Scheduling Image Processing Algorithms in Halide for x86, AVX and RISC-V RVV Targets

Sonali Naphade

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-228
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-228.html

August 16, 2023

Copyright © 2023, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

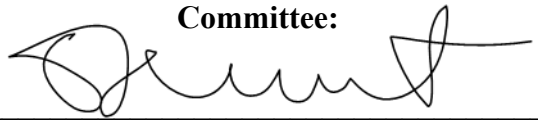
Scheduling Image Processing Algorithms in Halide for x86, AVX and RISC-V RVV Targets
by Sonali Naphade

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Borivoje Nikolic
Research Advisor

(8/15/2023)

* * * * * * *

Professor Krste Asanović
Second Reader

(8/15/2023)

Scheduling Image Processing Algorithms in Halide for x86, AVX and RISC-V RVV Targets

by

Sonali Naphade

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master's of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Borivoje Nikolic, Chair
Professor Krste Asanović

Summer 2023

Scheduling Image Processing Algorithms in Halide for x86, AVX and RISC-V RVV Targets

Copyright 2023

by

Sonali Naphade

Abstract

Scheduling Image Processing Algorithms in Halide for x86, AVX and RISC-V RVV Targets

by

Sonali Naphade

Master's of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Borivoje Nikolic, Chair

As computer vision expands and becomes an ever more important field, the speed of image processing and specific image processing and graphics accelerators becomes important. However, scheduling image algorithms by hand while maintaining correctness can become quite tedious, and the most efficient schedules are often found through experimentation. Therefore, specific high performance computing languages like Halide have been developed to allow the programmer more flexibility in finding efficiency by separating the algorithm from scheduling. In this work, the specific image kernels of unsharp filter, harris corners, and non-linear means were implemented and scheduled in Halide. Two main approaches of either splitting or tiling the output stage and computing producer stages with respect to the output were implemented. Furthermore, the performance of different schedules were evaluated for x86 AVX and RISC-V RVV targets. A comparison of the most efficient schedules for the variety of targets reveals the strengths and difficulties of Halide and possible points of inefficiency in Halide-generated RVV code.

Contents

Acknowledgments

I would like to thank my research advisor Professor Borivoje Nikolic for mentoring and supporting me in my research as a 5th Year M.S. student. I truly appreciate him giving me the opportunity explore and learn in this field, and his guidance has made my research journey more meaningful and insightful. I would also like to thank Professor Krste Asanović for being the second reader for my technical report.

Additionally, much of my work would not have been possible without the assistance of wonderful graduate and postdoctoral students of SLICE lab. In particular, I would like to thank Miles Rusch for his guidance and patience. I am deeply grateful for the help at every step of the way, from set up and debugging, to understanding Halide and Exo scheduling. I also want to thank Grace Dinh, Adrian Castello, and Gilbert Bernstein, for their insights and help with this project.

Finally, I want to thank my parents, Vijaya and Jay, for always supporting, encouraging, and believing in me. They have been my cheerleaders from day one, nurturing my passions and letting me explore freely. For that, I am forever grateful, and I love them infinity always.

Chapter 1

Introduction

Image processing is the process of extracting useful information from an image in its digital form. As more and more applications today utilize computer vision and computational photography, the speed of image processing algorithms is critical. In practice, these algorithms must be very efficient, from running on mobile devices with power constraints to workloads with many images. In low-level languages like C, the programmer can restructure naive code to achieve an optimized algorithm with much better performance. However, this requires careful thought on the programmer's part to maintain correctness in addition to efficiency, and generates complex, inflexible code. For image processing pipelines which consist of many different stages, the most optimal code would be extremely difficult to write.

Halide is a domain-specific language designed to make it easier to write efficient image and array processing code on modern machines. The algorithm is separated from its scheduling, allowing the programmer to more flexibly experiment with different choices for optimal performance. Furthermore, code can be generated for a variety of CPU and GPU architectures, allowing for a comparison of performance across different targets. Vector architectures can lend great speedup to image pipelines, so the standardized x86 AVX and variable-length vectors of RISC-V RVV targets are of interest. In this work, we aim to experiment with different ways of optimally scheduling specific image processing algorithms in Halide. We aim to generate code in the x86 AVX and RISC-V RVV targets, comparing their performances and analyzing generated RVV code for inefficient trends.

Chapter 2

Background

2.1 Image Processing Algorithms

In image processing, digital image data is operated on to extract useful information or transform the image. Image kernels operate on each pixel of the image, which may be in grayscale or color (three values at each location for each of three RGB color channels). There are many different categories within image processing [8], from classification, to enhancement and feature extraction algorithms. In this work, I focus on three specific image processing algorithms: image sharpening with unsharp masking, harris corner detection, and image denoising with non-local means. The sections below go into more detail on each algorithm as they will be used in later chapters.

2.1.1 Image Sharpening with Unsharp Masking

Image sharpening is a contrast enhancement technique aiming to accentuate details and edges in an image. Common image sharpening techniques can be separated into two categories: modifying pixel values with histogram re-scaling approaches or separating and emphasizing an image's high frequency components. Both approaches to sharpening have their own pros and cons, and further details about image sharpening methods can be found in [12].

In this paper, we focus on image sharpening with unsharp masking. Low frequencies of an image are obtained by convolving the original image with a low-pass filter. The resulting blurred image is called the unsharp mask. The high frequencies of the image, containing edges and fine detail information, are then obtained by subtracting the unsharp mask from the original image. A sharpened image is obtained by accentuating the high frequencies in the original image. The basic unsharp masking procedure can be described by,

$$y(u,v) = x(u,v) + \lambda z(u,v)$$

where $y(u,v)$ is the sharpened output pixel value, $x(u,v)$ is the original image input pixel value, λ is the gain factor, and $z(u,v)$ is the high-frequency value at a pixel. λ must be

tuned well for good sharpening results, and adaptive approaches seek to vary the gain factor in different image areas [14, 12].

(a) original image input

(b) high frequencies obtained from subtracting blurred image from original image

(c) sharpened output image

Figure 2.1: Process of sharpening with unsharp mask. Figure from [12]

There are various ways to compute z, from linear and quadratic operators to laplacian and normalized methods [3]. In this work, we use a normalized method to compute λz and we convolve the image with a 7x7 gaussian kernel before subtracting the blurred image from the original to extract its high frequencies. The below equation describes this process,

$$\lambda z(u,v) = \frac{gray(u,v) - blur(u,v)}{gray(u,v)} x(u,v)$$

where $gray(u,v)$ is the original image in grayscale.

2.1.2 Harris Corner Detection

Harris corner detection is a feature extraction approach for identifying corners within an image. Identifying corners is useful for a variety of purposes in in image processing and computer vision, from image matching and mosaic construction [7].

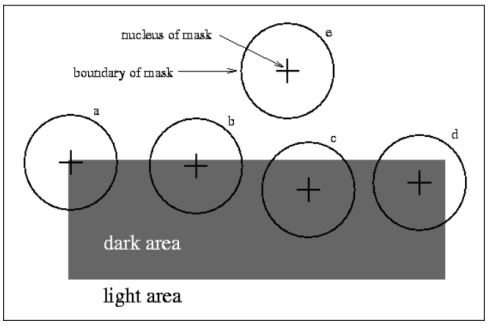
Broadly, a corner can be described as an area that is distinct from its surroundings. If an area of an image is a corner, shifting a small window in all directions around the area should result in a large change. This precise idea is captured by Harris and Stephens in their paper [10, 9]. Mathematically, the change in appearance of a window W for a small shift $[u,v]$ can be given by

$$E(u,v) = \sum_{(x,y)\in W} [I(x+u, y+v) - I(x,y)]^2$$

. Substituting first-order Taylor approximations for a small change $[u,v]$, E simplifies to

$$E(u,v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix}$$

(a) image with output corners marked [7]



(b) identifying a corner, possibilities of what a window can encounter [7].

Figure 2.2: Harris corner detector output.

where M is the second moment matrix involving image derivatives in the x and y directions, I_x and I_y:

$$M = \sum_{(x,y)\in W} \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

From matrix M, we can calculate corner response R

$$R = Det(M) - kTr(M)^2$$

$$Det(M) = I_x^2 I_y^2 - (I_x I_y)^2 = \lambda_1 \lambda_2$$

$$Tr(M) = I_x^2 + I_y^2 = \lambda_1 + \lambda_2$$

Eigenvalues λ_1 and λ_2 in the diagonalization of M give a rotationally invariant measure of intensity change. When both eigenvalues are large, corner reponse R will be large, indicating presence of a corner. In this work, we use this algorithm for Harris corner detection with $k = 0.04$ on grayscale images. Furthermore, we use the Sobel operator to compute gradients I_x and I_y, as described in [17].

2.1.3 Non-Local Means Denoising

Image denoising techniques aim to remove noise in an image while retaining details like texture and edges. Local mean filters denoise by taking the mean value of pixels in an area to smooth out noise. By contrast, non-local means takes a global mean of the image weighted

$\frac{1}{8}$

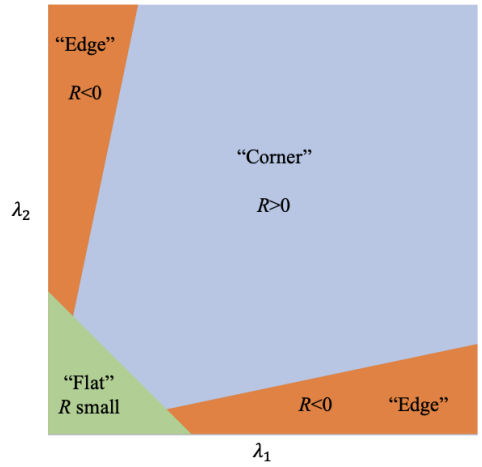
-1	0	1
-2	0	2
-1	0	1

$\frac{1}{8}$

1	2	1
0	0	0
-1	-2	-1

(b) Sobel operator

(a) The Sobel filter [17]



(b) The space of possible corner responses R. When R is high, this indicates both eigenvalues are large and that there is a corner [17].

Figure 2.4: Weights are computed as a similarity measure between two patches, meaning the weights for patches two and three will be large while those between patches one and three will be small. Image generated based on weight comparison image in [4].

by similarity of each pixel to the target pixel, as shown in Figure 2.4. More information about these various denoising methods can be found in [6].

The non-local means algorithm can be expressed by

$$NL(i) = \sum_{j \in I} w(i,j)v(j)$$

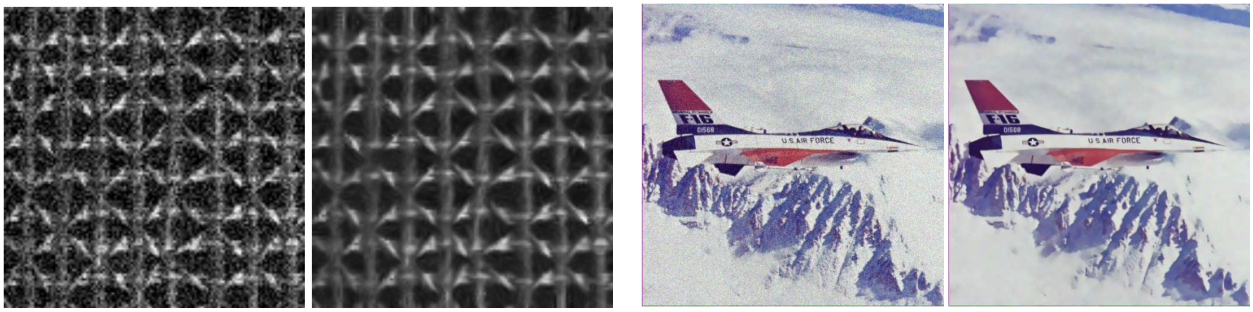
where v is the input noisy image and $w(i,j)$ is a weight representing similarity of pixels i and j. Similarity between pixels depends on the similarity of gray level vectors of the pixels' encompassing patch area, represented by N. Very similar windows will have a large weight while contrasting windows will have small weights. The weight can be defined as

$$w(i,j) = \frac{1}{Z(i)} e^{-\frac{||v(N_i) - v(N_j)||^2_{2,a}}{h^2}},$$

where h is a filtering parameter and $Z(i)$ is the normalizing constant

$$Z(i) = \sum_j e^{-\frac{||v(N_i) - v(N_j)||^2_{2,a}}{h^2}}$$

. This derivation comes from Buades' and Morel's work, and more specific details about this calculation can be found in [4, 5].



(a) grayscale images (b) color images

Figure 2.5: Input images and output nl-means denoised images [6].

2.2 Halide

Writing fast image processing pipelines can be very tedious and challenging. Programmers must generally take into account both parallelism (distributing work across threads, utilizing SIMD vectors) and locality (tiling computations such that intermediate results are retained in local cache). These optimizations can yield immense speedup, yet even in a low-level language like C, optimized code can be complex and inflexible, as seen in Figure 2.6. For multi-stage image pipelines, optimization often requires experimentation, and programming in this way is impractical.

Halide is a programming language embedded in C++ that offers a solution to this problem. Halide separates an intrinsic algorithm from scheduling code, making it much easier to write efficient, high-performance image processing code for modern machines. Scheduling is much more flexible and choices can be easily changed without impacting algorithm correctness, allowing the programmer to try and analyze different optimization techniques quickly.

```
void blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 Image<uint16_t> bh(in.width(), in.height();

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
   for (int x = 0; x < in.width (); x++)
    bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;
}
```

(a) Naive C++: 6.5ms per megapixel

```
void fast_blur(const Image<uint16_t> &in, Image<uint16_t> &bv) {
 __m128i one_third = _mm_set1_epi16(21846);
 #pragma omp parallel for
 for (int yTile = 0; yTile < in.height(); yTile += 32) {
  __m128i a, b, c, sum, avg;
  __m128i bh[(256/8)*(32+2)];
  for (int xTile = 0; xTile < in.width(); xTile += 256) {
   __m128i *bhPtr = bh;
   for (int y = -1; y < 32+1; y++) {
    const uint16_t *inPtr = &(in(xTile, yTile+y));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_loadu_si128((__m128i*)(inPtr - 1));
       b = _mm_loadu_si128((__m128i*)(inPtr + 1));
       c = _mm_load_si128 ((__m128i*)(inPtr));
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(bhPtr++, avg);
     inPtr += 8;
    }}
   bhPtr = bh;
   for (int y = 0; y < 32; y++) {
    __m128i *outPtr = (__m128i *)(&(bv(xTile, yTile+y)));
    for (int x = 0; x < 256; x += 8) {
       a = _mm_load_si128(bhPtr + (256 * 2) / 8);
       b = _mm_load_si128(bhPtr + 256 / 8);
       c = _mm_load_si128(bhPtr++);
     sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
     avg = _mm_mulhi_epi16(sum, one_third);
     _mm_store_si128(outPtr++, avg);
}}}}}
```

(b) Fast C++ (for x86) : 0.30ms per megapixel

```
Func halide_blur(Func in) {
 Func bh, bv;
 Var x, y, xi, yi;

 // The algorithm
 bh(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 bv(x, y) = (bh(x, y-1) + bh(x, y) + bh(x, y+1))/3;

 // The schedule
 bv.tile(x, y, xi, yi, 256, 32)
   .vectorize(xi, 8).parallel(y);
 bh.compute_at(bv, x).vectorize(x, 8);

 return bv;
}
```

(c) Halide : 0.29ms per megapixel

Figure 2.6: Comparison of optimizing code in C++ and Halide from [16].

An image pipeline algorithm can be written in Halide as a a series of functions, where each 'Func' object corresponds to a single pipeline stage. Functions are defined at integer coordinates of 'Var' object variables as described by an 'Expr' object computation using these variables. Halide scheduling operators allow for a wide variety of choices. To actually compute pixel data and generate an output image, we realize the final function of an image

pipeline with specified input image dimensions. More details on the specifics of Funcs, Vars, Exprs, and processing images in Halide can be found in [15].

The schedule of the image pipeline is defined separately. Unscheduled stages are computed inline by default. When scheduling, the programmer must balance tradeoffs between redundant computation, temporary memory usage, and memory bandwidth for their specific purposes. The choice space for scheduling an image pipeline falls into two categories:

1. The ordering of loops to compute values within this stage
2. The ordering of computation of different producer-consumer stages within a multi-stage pipeline

An overview of the primary scheduling calls within these choice spaces is given in the following sections.

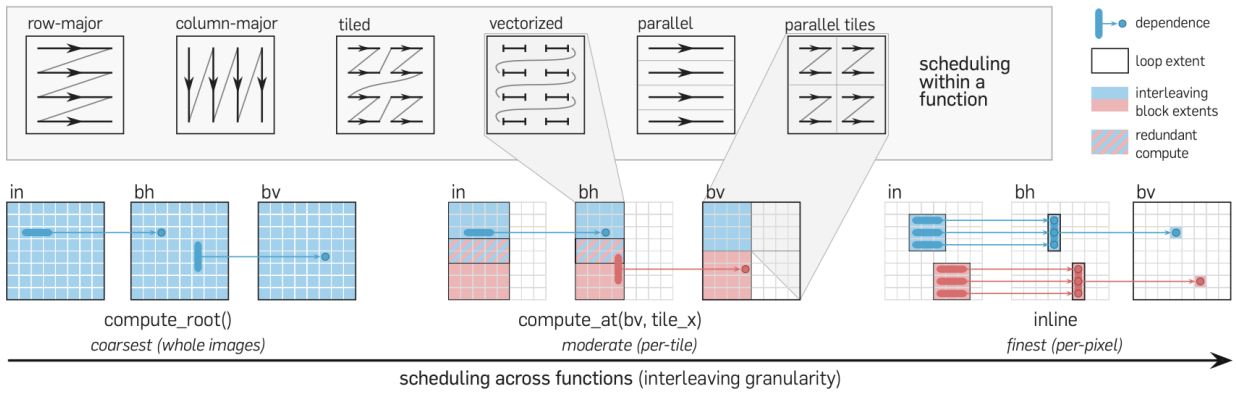


Figure 2.7: Visualization of different how different scheduling directives work from [16].

2.2.1 Single-Stage Scheduling Operations

These are some basic Halide directives used to schedule a single image pipeline stage by making changes within the stage.

- **reorder(vars)**: reorder loop nesting for the function in the given order of variables, from innermost out.
- **split(oldVar, outer, inner, factor)**: split a loop over the given variable oldVar into nested loops over inner and outer sub-variables. The inner loop increments from zero to the given split factor, and the outer loop increments from zero to the original extent of oldVar divide by the split factor.

- **fuse(inner, outer, newVar)**: the opposite of split, fuse merges loops over inner and outer sub-variables into a single loop over the newVar variable with the product of the extents.
- **tile(oldVar1, oldVar2, outer1, outer2, inner1, inner2, factor1, factor2)**: splitting in two dimensions, by the given factors and ordering nested loops to be inner1, inner2, outer1, outer2, from innermost out. This is shorthand to obtain a tiled traversal of a domain, where blocks are computed at a time.
- **vectorize(var, factor)**: vectorizes computation along a dimension by splitting the variable according to the factor and vectorizing the inner variable loop.
- **unroll(var, factor)**: unrolls computation along a dimension by splitting the variable according to the dactor an unrolling the inner variable loop.
- **parallel(var)**: parallelizes computation along the dimension given by the variable.

2.2.2 Multi-stage Scheduling Operations

These are some basic Halide directives used to schedule a multi-stage image pipeline, by indicating when a stage should be computed. We will describe the relationship between stages as a producer-consumer relationship, where the producer stage computed intermediate results used by the consumer stage.

- **compute_root()**: evaluate the current producer stage fully before the consumer.
- **compute_at(consumer, var)**: evaluate the current producer stage as needed for each variable value of the consumer.
- **compute_with(func, var)**: If current stage and given stage have the same schedule from the outermost dimension until the given variable, the two stages can be fused such that both computations occur within only one set of nested loops.
- **store_root()**: store all of the producer computations in a buffer at the outermost level (to avoid redundant computation).
- **store_at(consumer, var)**: store producer computations in a buffer just within the consumer's loop over the given variable (to avoid redundant computation).

2.2.3 Generators and Targets

Halide is supported by the LLVM compiler structure. Halide functions expressed in C++ and scheduling are transformed into LLVM bitcode that LLVM transforms into code for

specified targets [include reference]. Halide generators are a structured method for ahead-of-time Halide pipeline compilation, and can be used to compile object files for a variety of targets, such as x86, ARM, and CUDA.

In this work, we generate code for the targets of x86 AVX and RISC-V's Vector (RVV) extension. With Spike, we are able to attain cycle counts to compare the efficiency of code for each target. Our goal is to experiment with optimal scheduling of the specific image algorithms from section 2.1 for both of these targets. Additionally, we want to compare Halide-generated RVV code and analyze possible inefficiencies compared to Halide-generated x86 AVX code as standard.

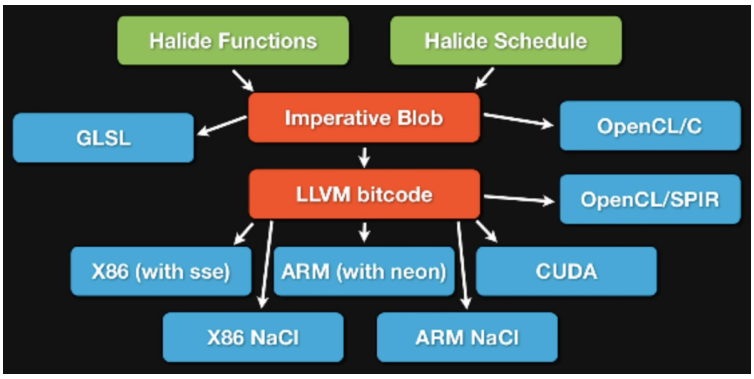


Figure 2.8: The Halide compiler process

Chapter 3

Scheduling Experiments and Performance Evaluation of Image Processing Algorithms in Halide

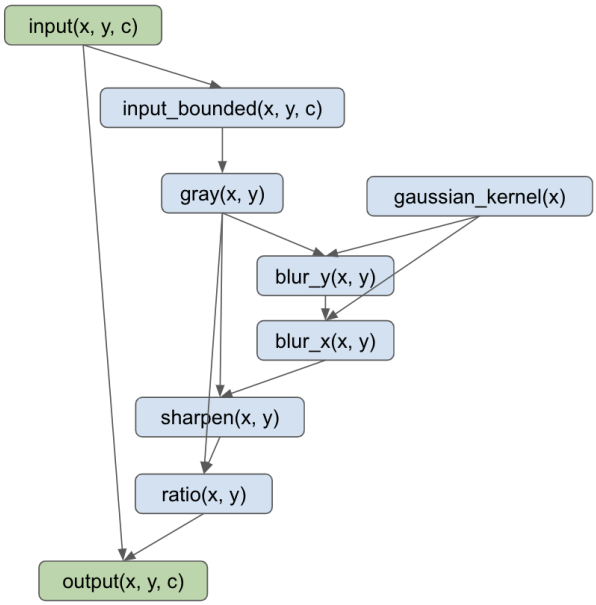
A variety of different schedules were tried for each of the unsharp mask sharpening, harris corner detection, and non-local means denoising algorithms. The main two approaches to scheduling for optimized performance were splitting or tiling the final output stage for blocked traversal, and computing certain producer stages as needed per block. Additionally, storing intermediate results from producer stages and vectorizing loops often led to improved performance. Although Halide provides parallel computation scheduling directives, these were not used in this work and only single-core schedules were considered.

To evaluate the performance of a schedule, assembly code was generated for x86 scalar and AVX vector targets in addition to the RVV vector target. Code was run on the Intel Xeon 6354 CPUs, notably with a 48 KB L1 data cache. This code was compiled and simulated with Spike, outputting a cycle count. Out of many scheduling experiments, the best schedules were determined by their cycle count results for an input matrix of size 64x64, before being further evaluated for square matrices of size from 64 to 512, in increments of 32. Each of the following section details the specific optimal schedules and analyzes their performance for each image processing algorithm.

3.1 Unsharp Mask Image Sharpening

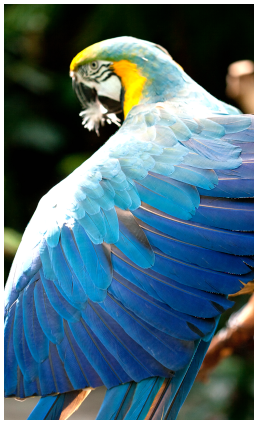
The unsharp masking algorithm described in Section 2.1.1 was implemented in Halide and sharpened output was generated on a sample color image.

After experimenting with different schedules, these are the optimal schedules whose performances were further evaluated.



(a) Unsharp mask algorithm major stages

(b) Our input image



(c) Our generated output image

Figure 3.1: Unsharp Mask algorithm and realized output.

1. **split-yi16-vec16**: split y loop of output stage into nested yo and yi loops, where yi traverse a wide image swath of width 16. Output stage nested loops are reorder to be x, c, yi, yo, innermost out. The gray, blur_y, and ratio stages are computed as needed with respect to the yi loop of the output stage and are stored within the yo loop of the output stage. The output, gray, blur_y, and ratio stages are each vectorized in variable x with vector size 16.

2. **split-yi16-vec8**: similar to split-yi16-vec16 schedules, except with output, gray, blur_y, and ratio stages each vectorized in variable x with vector size 8.

3. **tile-16x32-vec16**: tile output stage by splitting x and y loops into nested xi, yi, xo, and yo loops. xi spans [0, 16) and yi spans [0, 32), such that the output stage is computed in blocks of size 16x32 at a time. Output stage nested loops are reorder to be xi, c, yi, xo, yo, innermost out. The gray, blur_y, and ratio stages are computed as needed with respect to the xo loop of the output stage, without storing. The output, gray, blur_y, and ratio stages are each vectorized in variable x with vector size 16.

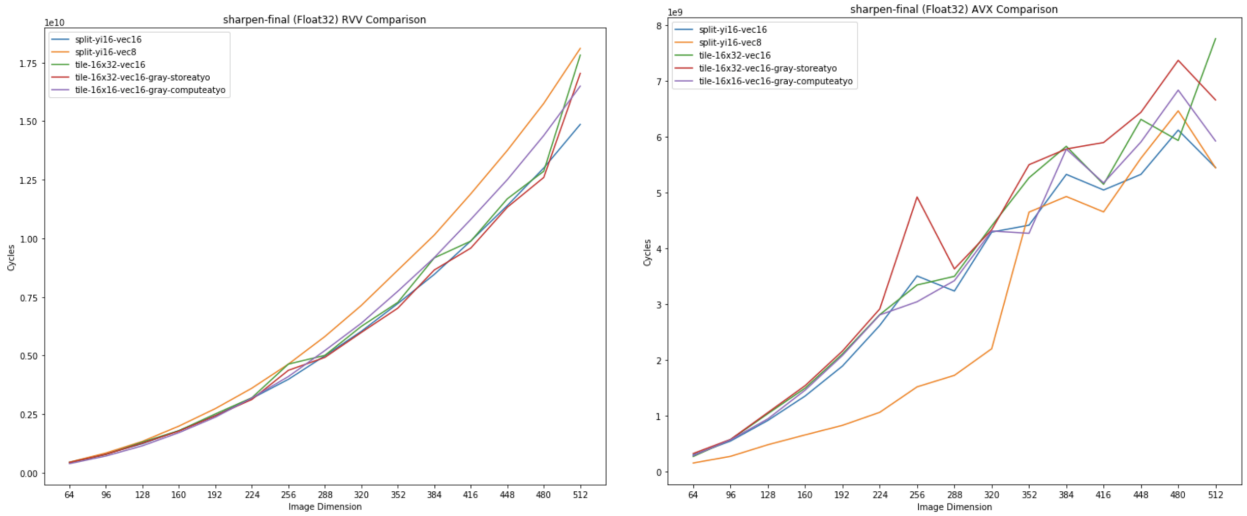
4. **tile-16x32-vec16-gray-storeatyo**: similar to tile-16x32-vec16 schedules, except the gray stage's results are stored within the yo loop of the output stage.

5. **tile-16x16-vec16-gray-computeatyo**: similar to tile-16x32-vec16 schedule, except the yi loop of the output stage spans [0, 16) and the gray stage is computed as needed with respect to the yo loop of the output stage. Additionally, the gaussian kernel stage is computed as needed with respect to the yo loop of the output stage and is stored at the outermost level of the algorithm.

Performance Evaluation

Schedule	x86 scalar cycles	x86 avx cycles	rvv cycles
split-yi16-vec16	6069796	301456	442881
split-yi16-vec8	2441012	156228	444127
tile-16x32-vec16	6069212	273580	419097
tile-16x32-vec16-gray-storeatyo	6074588	327044	428240
tile-16x16-vec16-gray-computeatyo	9867308	288140	378991

Table 3.1: Performance of unsharp mask schedules for 64x64 images.



(a) RVV Performance

(b) x86 AVX Performance

Figure 3.2: Performance of unsharp mask schedules for images of size 64x64 to 512x512. True cycle count is table value multiplied by 1000.

In terms of RVV code, all five schedules gave similar performance. As seen in Table 3.1, for an input image of size 64x64 pixels, all the schedules yielded around 4×10^8 cycles. In Figure 3.2.a, we see that each schedule follows a similar exponential trend as the input dimension

size increases. If examining the graph closely, the split-yi16-vec8 schedule takes slightly more cycles than the rest, since vector size 8 does not utilize the full capacity of a register. The Halide directive natural_vector_size<float>() calculates the optimal vector size as vector register length (512 here) divided by data type size (float 32 in this case), which would be 16. The tile-16x32-vec16 and tile-16x32-vec16-gray-storeatyo are less smooth compared to the other schedules, like tile-16x16-vec16-gray-computeatyo. The blur_x and blur_y stages utilize a larger window of gray stage results. These two schedules compute the gray stage for every xo loop iteration of the output stage, so a narrower window of values is computed compared to as in tile-16x16-vec16-gray-computeatyo. Additionally, the block size of 16x32 means overlapping edges of the gray stage used for multiple blocks of blur_x and blur_y are larger, and more values need to be recomputed or stored.

There is much more variation among the results of the schedules for in x86 AVX code. For size 64x64 inputs, Table 3.1 shows the schedules approximately output 3×10^8 cycles. However, the split-yi16-vec8 schedule seems to be an outlier with an output of 1.5×10^8 cycles, about half compared to the others which use vector size 16. This is interesting because the vector size 16 is the natural vector size calculated for the data type and the target, as mentioned above. Yet in Figure 3.2.b, vector size 8 in split-yi16-vec8 does much better than any schedule with size 16 vectors until input sizes larger than 352x352 pixels. Examining the AVX code, code with larger vector size 16 had 50 to 70 more vector instructions like vpextrq, vpinsrb, vmovq to manipulate vector values within inner loop sections. It is unclear why this difference happens for small to medium input sizes, but this would be significant across multiple iterations. Otherwise, all five schedules seem to perform similarly, the cycle counts increasing about linearly on average, if not smoothly, as input size increases. tile-16x32-vec16-gray-storeatyo does have an unusual spike at input size 256. A comparison of the AVX assembly for input sizes 256 and 288 yields no difference at all, so the spike must be due to storing the gray stage for each iteration of output's yo loop causing extra time to be taken.

Furthermore, we can compare the code generated for the RVV target to code generated for x86 AVX by taking a ratio of RVV cycles to AVX cycles, shown in Figure 3.3. As x86 AVX is rather standardized and widespread, we can use it as a standard to compare RVV code with. RVV code cycle counts for the schedules are between one to three times that of AVX code. The split-yi16-vec8 has an outlier, higher trend due to the extremely low AVX cycle counts. Otherwise, all schedules have RVV code that approaches AVX performance at a sort of absolute minima at input side dimension of 224 to 256. RVV code seems to be the most efficiently generated for medium sized inputs. tile-16x32-vec16-gray-storeatyo even dips below a ratio of one at an input dimension of 256, meaning its RVV code performs better than AVX code at that point.

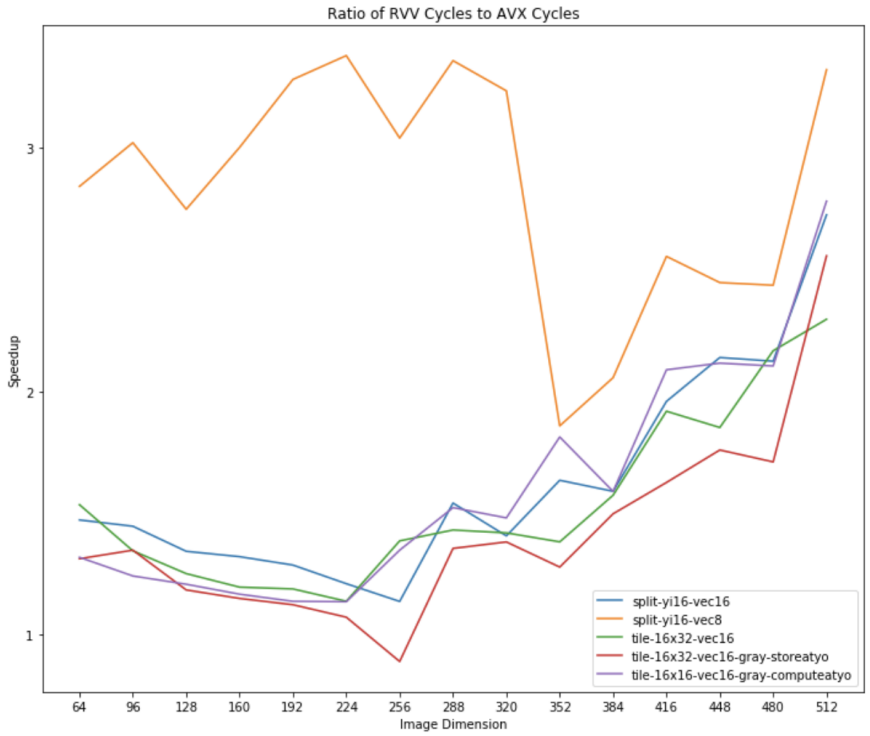


Figure 3.3: Comparison of RVV and x86 AVX performance for unsharp mask schedules on images of size 64x64 to 512x512.

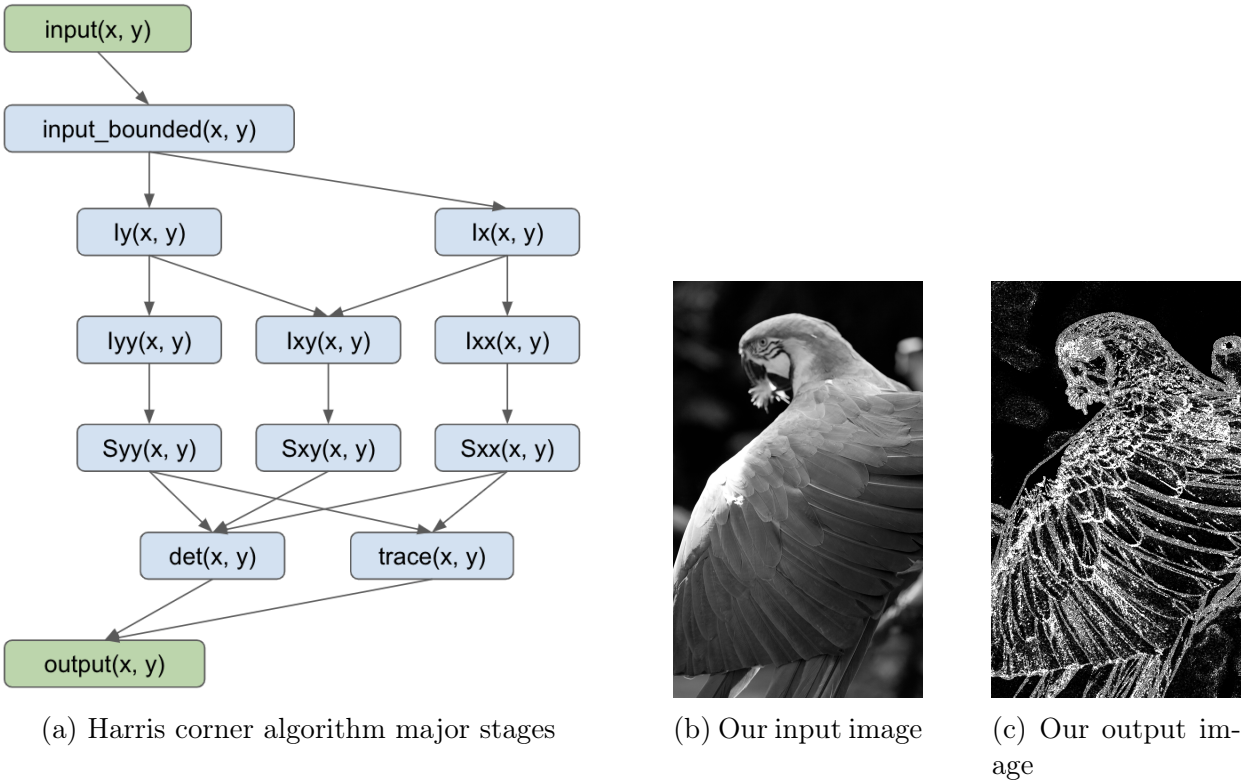
3.2 Harris Corner Detection

The Harris corner detection algorithm described in Section 2.1.2 was implemented in Halide for grayscale images and output was generated on a sample image.

After experimenting with different schedules, these are the optimal schedules whose performances were further evaluated.

1. **split-yi8-vec16**: split y loop of output stage into nested yo and yi loops, where yi traverses a wide image swath of width 16. Stages Ix and Iy are computed as needed with respect to the yi loop of the output stage and are stored within the yo loop of the output stage. Ix and Iy are computed together within a single set of nested loops. The output, Ix, and Iy stages are each vectorized in variable x with vector size 16.

2. **split-yi16-vec16-computeatyo**: similar to split-yi8-vec16, except yi traverse a wide image swath of width 8 and stages Ix and Iy are computed as needed with respect to the yo loop of the output stage, with no storing.



(a) Harris corner algorithm major stages

(b) Our input image

(c) Our output image

Figure 3.4: Unsharp Mask algorithm and realized output (white areas indicate high corner response, indicating corners/fine edges).

3. **split-cascade-yi16-vec16**: split y loop of output stage into nested yo and yi loops, where yi traverses a wide image swath of width 16. Every stage of the pipeline is scheduled in a cascaded order:

   - det and trace stages computed as needed with respect to the yo loop of the output stage.
   - Syy, Sxx, and Sxy stages computed as needed with respect to the y loop of the det stage.
   - Iyy, Ixx, and Ixy stages computed as needed with respect to the y loop of the Syy stage.
   - Iy and Ix stages computed as needed with respect to the y loop of the Iyy stage.

   Each group of stages is computed in one set of nested loops and is stored within the yo loop of the output stage. Additionally, each stage mentioned above including the output stage is vectorized in variable x with vector size 16.

4. **tile-16x16-vec16-computeatyo**: tile output stage by splitting x and y loops into nested xi, yi, xo, and yo loops. xi spans [0, 16) and yi spans [0, 16), such that output stage is computed in blocks of size 16x16 at a time. Stages Ix and Iy are computed as needed with respect to the yo loop of the output stage, without storing. The output, Ix, and Iy stages are each vectorized in variable x with vector size 16.

5. **tile-cascade-16x32-vec16**: Similar to split-cascade-yi16-vec16, except the output stage is tiled with nested xi, yi, xo, and yo loops, where xi spans [0, 16) and yi spans [0, 32), such that output stage is computed in blocks of size 16x32 at a time. Additionally, the Iy and Ix stages are left unscheduled and are computed inline by default.

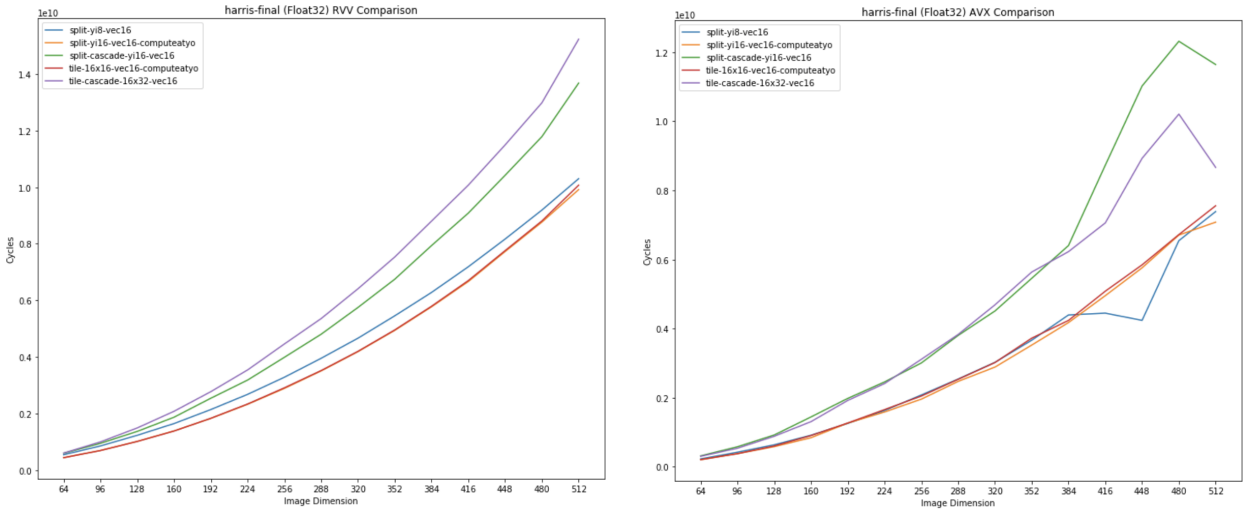
Performance Evaluation

Schedule	x86 scalar cycles	x86 avx cycles	rvv cycles
split-yi8-vec16	1070844	224484	537154
split-yi16-vec16-computeatyo	1062636	201668	433964
split-cascade-yi16-vec16	1069488	313416	590700
tile-16x16-vec16-computeatyo	1075440	204076	432303
tile-cascade-16x32-vec16	1204808	300248	596239

Table 3.2: Performance of Harris corner detector schedules for 64x64 images.

For the RVV target, with an initial input size of 64x64, the schedules perform on the order of 10^8 cycles in Table 3.2. There is some variability, from about 4×10^8 to 6×10^8 cycles. This initial difference between schedules is maintained and reflected across small, medium, and large input sizes in Figure 3.5.a. All the schedules follow a smooth exponential trend, yet no lines overlap, with split-yi16-vec16-computeatyo and tile-16x16-vec16-computeatyo having the best performance and tile-cascade-16x32-vec16 and split-cascade-yi16-vec16 on the worse end.

In AVX, Table 3.2 shows the performances range from 2×10^8 to 3×10^8 cycles for input size 64x64. Over many input sizes in Figure 3.5.b, all the schedules follow an exponential trend. The performances separate into two categories, where the cascade schedules (split-cascade-yi16-vec16 and tile-cascade-16x32-vec16) do worse. The cascade schedules had about 6500 lines of AVX code, almost double compared to around 3300 for the rest. Most of this difference, in both the RVV andd AVX code, was extensive prologues and epilogues for setting up the cascade loops in addition to many extra spills and reloads (over 1000 compared to about 200 for the non-cascade schedules). split-yi8-vec16 has a sudden drop in AVX cycles at input size 448. A comparison with the AVX code for input size 416 showed the inner loop sections had long series of vector multiplies (vfmadd) and stores to memory (vmovups),



(a) RVV Performance

(b) x86 AVX Performance

Figure 3.5: Performance of Harris corner detector schedules for images of size 64x64 to 512x512.

about five times as many as for input size 448. This is the cause of the difference, although it is unclear why this particularly happens at size 448.

Comparing code generated for the RVV target to code generated for x86 AVX, Figure 3.6 shows the ratio of RVV to AVX cycles. RVV code cycle counts are between one to three times those of AVX code. As input size increases, the RVV to AVX ratio decreases and RVV code is generated more efficiently in all schedules. The largest input sizes of 448 to 512 is where RVV performance is closest to AVX performance. On average, split-yi8-vec16 had the worst ratio. However, split-cascade-yi16-vec16 had the best ratio for various input sizes, with ratio less than one for sizes 448 and 480, such that RVV code performed better than AVX. split-yi8-vec16 particularly spikes at input size 448 due to its dip in AVX cycles taken. Oppositely, split-cascade-yi16-vec16 dips around input size 448 due to its steeper increase in AVX cycles. Interestingly, within both RVV cycles and AVX cycles, split-yi16-vec16-computeatyo and tile-16x16-vec16-computeatyo did about the exact same despite having different approaches. The tiling approach had more loop overheads but also utilized the cache more efficiently with 16x16 blocks of the image computed at once rather than wide rectangular sections with splitting.

3.3 Non-local Means Denoising

The non-local means denoising algorithm described in Section 2.1.3 was implemented in Halide and denoised output was generated on a randomized input array.

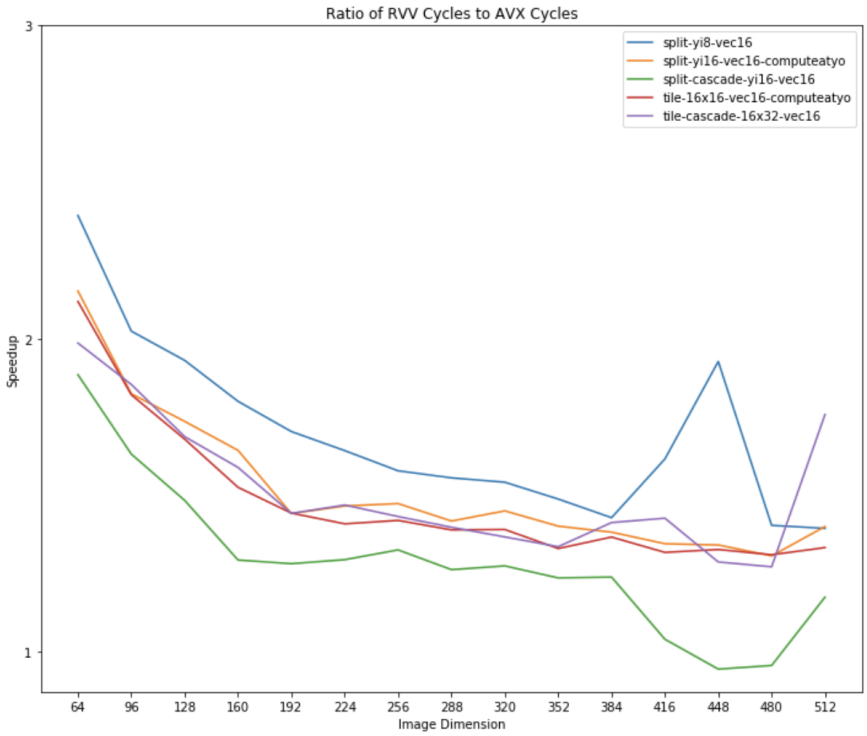


Figure 3.6: Comparison of RVV and x86 AVX performance for Harris corner detector schedules on images of size 64x64 to 512x512.

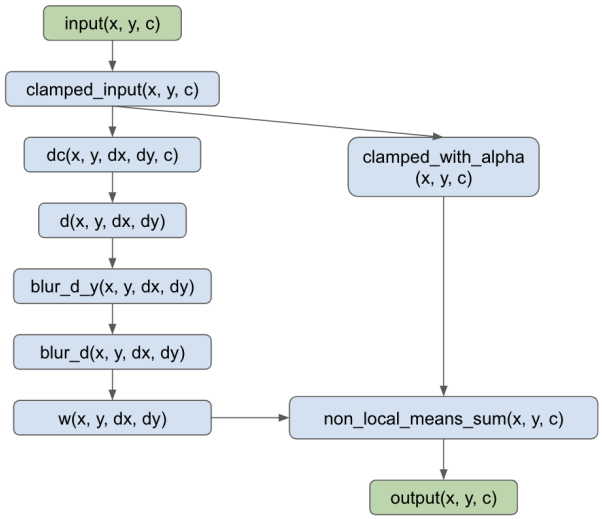


Figure 3.7: Non-local means algorithm major stages.

After experimenting with different schedules, these are the optimal schedules whose performances were further evaluated.

1. **split-yi2-vec16**: split y loop of output stage into nested yo and yi loops, where yi traverses a wide image swath of width 2. Stages non_local_means_sum, blur_d_y, and d are computed as needed with respect to the x loop of the output stage. The blur_d stage is computed at needed with respect to the x loop of the non_local_means_sum stage. Additionally, updates in the non_local_means_sum stage are reordered in order of c, x, y, innermost out, along with unrolling the c loop. The d stage is reordered in order y, x. The above scheduled stages are each vectorized in variable x with vector size 16.

2. **tile-16x8-vec16**: similar to split-yi2-vec16, except the output stage is tiled with nested xi, yi, xo, and yo loops, where xi spans [0, 16) and yi spans [0, 8), such that output stage is computed in blocks of size 16x8 at a time. blur_d_y and d stages are computed as needed for the output stage at xo, non_local_means_sum is computed as needed for the output at xi, and blur_d is computed as needed for non_local_means_sum at x. Instead of reordering the d stage, the blur_d_y stage is reordered in order y, x.

3. **tile-16x16-vec16-compatyo**: similar to tile-16x8-vec16 except in the output stage yi spans [0, 16), such that output stage is computed in blocks of size 16x16 at a time. Furthermore, the output loop order is reordered to be yi, c, xi, yo, xo, innermost out. The blur_d_y and d stages are computed as needed for the output stage at yo and both are reordered in order y, x.

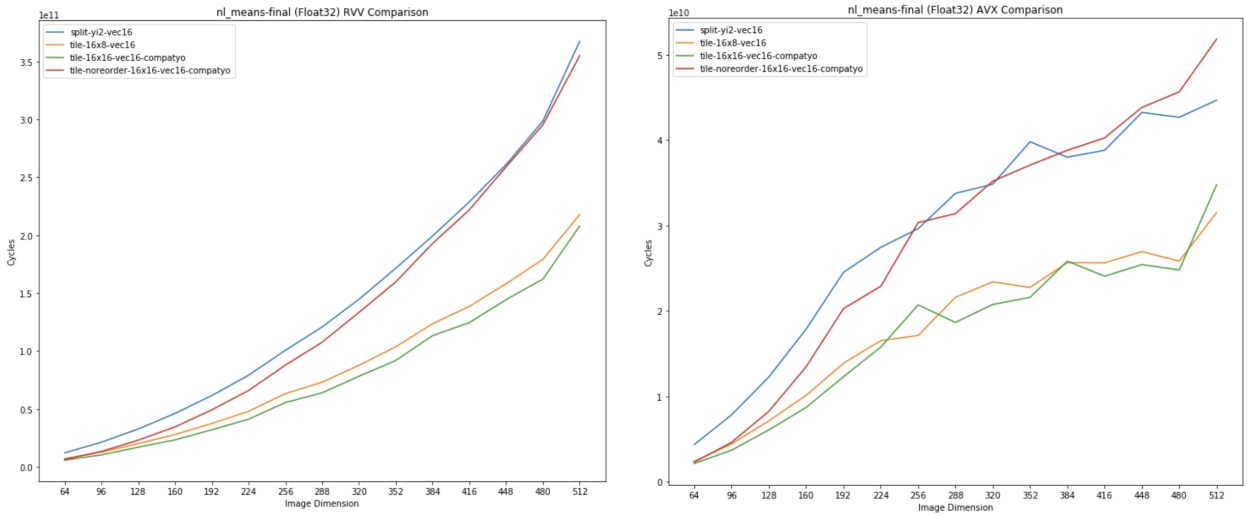
4. **tile-noreorder-16x16-vec16-compatyo**: similar to tile-16x16-vec16-compatyo without any reordering of the output stage's loops.

Performance Evaluation

Schedule	x86 scalar cycles	x86 avx cycles	rvv cycles
split-yi2-vec16	103985844	4336936	11955531
tile-16x8-vec16	104789090	2366228	6944837
tile-16x16-vec16-compatyo	111558376	2115412	5669392
tile-noreorder-16x16-vec16-compatyo	110556830	2282992	6347230

Table 3.3: Performance of non-local means denoising schedules for 64x64 images.

In terms of RVV code, for input size 64x64 pixels, the four schedules vary in performance from 5×10^9 to 11×10^9 cycles, with tile-16x16-vec16-compatyo taking half as many cycles as split-yi2-vec16 in Table 3.3. In Figure 3.8.a, we see the schedules follow pretty smooth



(a) RVV Performance

(b) x86 AVX Performance

Figure 3.8: Performance of non-local means denoising schedules for images of size 64x64 to 512x512.

exponential trends as input size increases. For input size greater than 192x192 pixels, a gap in performance occurs between different groups of schedules. Examining differences closely then, the tile-16x8-vec16 and tile-16x16-vec16-compatyo schedules do better.

Comparing AVX performance for input size 64, the schedules yield about 2×10^9 cycles, except for split-yi2-vec16 which takes double the amount to run, at 4×10^9 cycles, from Table 3.3. Looking at the performance across different input sizes in Figure 3.8.b, cycle counts increase steeply in small input sizes before increasing less steeply for input larger than 192 pixels per side. Similar to what was seen for RVV performance, the schedules separate into the same two groups with a gap in performance. For both RVV and AVX, examining the code between these two groups revealed that split-yi2-vec16 and tile-noreorder-16x16-vec16-compatyo had many more extraneous vector operations wihtin inner loops (vpinsrb, vextract, vpextrq) as the input sizes became larger. Sometimes the same value was calcualte multiple times when it could have been calculated once and stored.

In Figure 3.9, we see the ratio of RVV to AVX cycles, which can be taken as a statistic to gauge how efficiently code is generated for the RVV target. RVV code cycle counts are between two to seven times those of the generated AVX code. As input size increases, the ratio increases about linearly for each schedule, without much difference between schedules. Each schedule hits a absolute minimum ratio for an input size of about 192x192 pixels, where generated RVV code most closely replicates efficiently generated AVX code.

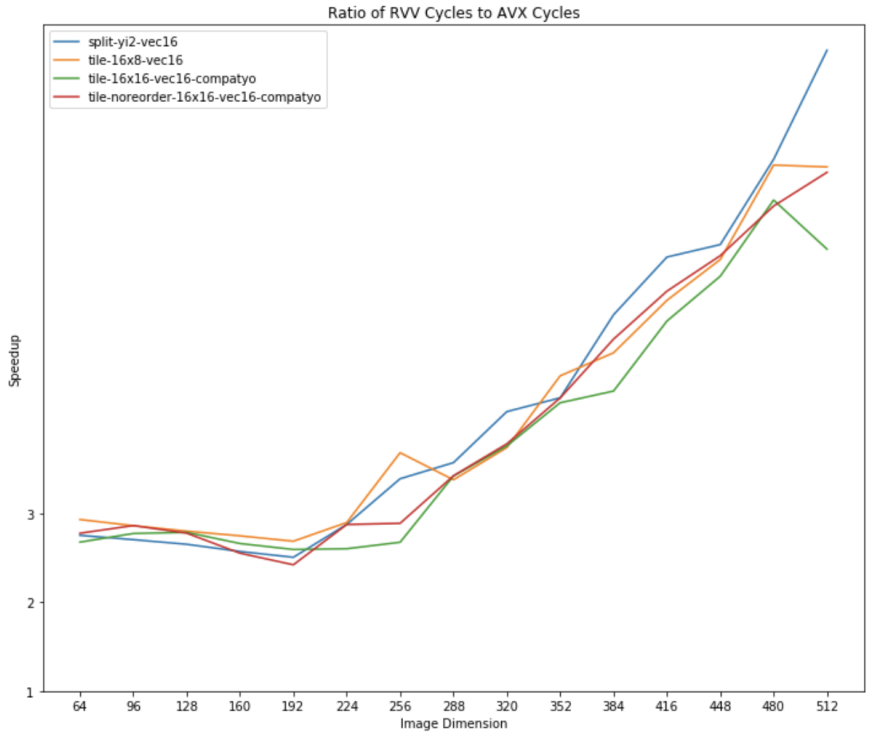


Figure 3.9: Comparison of RVV and x86 AVX performance for non-local means denoising schedules on images of size 64x64 to 512x512.

3.4 Inefficiences in Halide-generated RVV Code

As seen in the above sections, the Halide-generated RVV code usually takes two to three times the cycles compared to Halide-generated AVX code. Further examination of assembly generated files reveals inefficiencies that are the source [2, 13]. Mainly, repetitive inefficient portions of code were generally found in the prologues and epilogues before the major computation of the image processing algorithm.

In RISC-V, there are 32 integer registers (31 excluding the program counter, pc) and 32 vector registers. To maintain calling convention and ensure no values are lost even when different functions are called, the prologue before computation stores register values that could be overwritten on the stack and the epilogue after restores the values from the stack. The register sp is incremented and decremented to keep track of values on the stack. Additionally, when there are more local variables needed than registers available, some registers must be saved to stack to make space (register spill) and can later to restored (register reload).

Looking at the RVV assembly generated, there were many times a value was loaded into a specific register before a register spill saved it on the stack. After the spill, a new value would be loaded into the same register and the original value is never required in the same

vicinity. In the assembly fragments below, this process is shown happening repetitively. A more efficient approach would only load values into registers when required soon, avoiding filling up registers with extraneous values that then create excessive register spills.

```
ld  a2, 24(a0)
sd  a2, 696(sp)                    # 8-byte Folded Spill
lw  s1, 36(a0)
lw  a2, 8(s2)
sd  a2, 720(sp)                    # 8-byte Folded Spill
lw  a2, 24(s2)
sd  a2, 648(sp)                    # 8-byte Folded Spill
lw  a2, 40(s2)
sd  a2, 752(sp)                    # 8-byte Folded Spill
ld  s2, 16(a1)
ld  a2, 24(a1)
sd  a2, 728(sp)                    # 8-byte Folded Spill
lw  a2, 36(a1)
lw  a4, 8(t0)
sd  a4, 712(sp)                    # 8-byte Folded Spill
lw  a4, 24(t0)
sd  a4, 584(sp)                    # 8-byte Folded Spill
lw  a4, 40(t0)
sd  a4, 744(sp)                    # 8-byte Folded Spill
```

(a) register a2 is repetitively loaded into, where any loaded values immediately spill onto the stack.

```
vs1r.v  v8, (a0)                   # Unknown-size Folded Spill
li  a0, 2
vmv.s.x v8, a0
csrr    a0, vlenb
li  a1, 12
mul a0, a0, a1
add a0, a0, sp
addi    a0, a0, 880
vs1r.v  v8, (a0)                   # Unknown-size Folded Spill
li  a0, 3
vmv.s.x v8, a0
csrr    a0, vlenb
li  a1, 11
mul a0, a0, a1
add a0, a0, sp
addi    a0, a0, 880
vs1r.v  v8, (a0)                   # Unknown-size Folded Spill
```

(b) vector register v8 is repetitively loaded and spilled

```
addiw   s1, s1, -16
sd  s1, 624(sp)                    # 8-byte Folded Spill
srli    s1, a4, 30
sd  s1, 592(sp)                    # 8-byte Folded Spill
srli    a4, a4, 27
sd  a4, 576(sp)                    # 8-byte Folded Spill
slli    a4, a0, 2
sd  a4, 616(sp)                    # 8-byte Folded Spill
```

(c) each computation immediately gets stored on stack to free up registers

Figure 3.10: Portions of RVV code with unnecessary loads and repetitive spilling.

Additionally, in some places the same values are loaded multiple times within a loop when they could be loaded once separately in registers. This would minimize the overall number of loads required.

Furthermore, RVV allows for a variable vector length to control vector size and how many elements are operated on within a vector. In generated RVV code, the vector length is changed frequently in order to move and arrange elements. However means the vector length must be reset over and over again. The same computation could be done in two parts perhaps, accumulating some vslideup and vslidedown instructions to happen together before the vector length is changed.

```
add s0, a6, a4
ld  a5, 40(a0)
lw  t1, 12(s0)
add a5, a5, a4
sw  t1, 12(a5)
lw  s1, 8(s0)
sw  s1, 8(a5)
lw  s1, 4(s0)
sw  s1, 4(a5)
lw  s1, 0(s0)
addi    a4, a4, 16
sw  s1, 0(a5)
bne a4, t0, .LBB0_56
```

Figure 3.11: register a0 is not altered throughout this loop, so lines like ld a5, 40(a0) which happen in every iteration are repetitive.

```
vsetivli    zero, 9, e8, mf4, tu, ma
vslideup.vi v9, v11, 8
vsetivli    zero, 1, e32, m1, ta, ma
vslidedown.vi   v11, v10, 9
vmv.x.s a4, v11
add a4, a4, s9
lb  a4, 0(a4)
vmv.s.x v11, a4
vsetivli    zero, 10, e8, mf4, tu, ma
vslideup.vi v9, v11, 9
vsetivli    zero, 1, e32, m1, ta, ma
vslidedown.vi   v11, v10, 10
vmv.x.s a4, v11
add a4, a4, s9
lb  a4, 0(a4)
vmv.s.x v11, a4
```

Figure 3.12: v11 continually slides up into v9 and v10 continually slides down into v11, with vector length readjusted in between.

Chapter 4

Conclusion

In this work, different scheduling methods were explored for unsharp mask sharpening, Harris corner detector, and non-local means denoising image processing algorithms. Performance of schedules was evaluated by generating code for the x86 AVX and RVV targets and obtaining cycle counts through Spike. Overall, both methods of splitting and tiling the output stage and computing producer stages as needed with respect to the output worked well. Additionally, reordering and vectorizing each stage made large impacts. Comparing Halide-generated RVV code to x86 AVX, RVV cycles across all three algorithms usually was between one to three times greater. Additionally, by considering cycle counts across a variety of input sizes from 64 to 512 pixels per side, we saw the ratio of RVV to AVX cycles had some minimum, where RVV cycles approached AVX cycles. At these points, for specific input sizes, prologue and epilogue inefficiencies with repetitive loads and spills are minimized.

4.1 Future Work

It would be useful to see how handwritten RVV code for inefficient existing areas impacts performance. There are still many places were performance is better or worse for a certain input, but it is unclear why. More analysis of the RVV generated assembly and replacing portions with handwritten code could help determine the causes.

Within Halide itself, there are many different things that could be further experimented with. More can be explored with scheduling with respect to caches sizes, including L1 and L2 caches. Currently, althought cache size was kept in mind, the best combination of split and tile sizes were often found through experimentation. In terms of scheduling, parallelization with multiple threads could be explored for additional speedups. Halide also provides auto-scheduling [1] which given estimates, should optimally schedule the program. It would be interested to see what schedule auto-scheduling generates and how those compare to the handwritten ones in this work. In terms of targets, RVV handwritten code could be useful in reducing the mentioned inefficiencies found in this work. Additionally, performance could

be evaluated for different targets like CUDA and GPUs.

Similar to Halide, Exo [11] is another language that allows for easy scheduling and optimization of algorithms. In Exo, algorithms are also separated from scheduling, but scheduling directive allow for finer control loop manipulation and storage. Certain scheduling choices that are difficult to implement in Halide may be easier in Exo, and vice versa. It would be useful to implement these same algorithms in Exo, to see if better scheduling and therefore performance can be attained. Additionally, it would help understand the class of alogrithms that each language is most useful for programming.

Bibliography

[1] Andrew Adams et al. "Learning to optimize halide with tree search and random programs". In: *ACM Transactions on Graphics (TOG)* 38.4 (2019), pp. 1–12.

[2] Neil Adit and Adrian Sampson. "Performance Left on the Table: An Evaluation of Compiler Autovectorization for RISC-V". In: *IEEE Micro* 42.5 (2022), pp. 41–48. DOI: 10.1109/MM.2022.3184867.

[3] M.A. Badamchizadeh and A. Aghagolzadeh. "Comparative study of unsharp masking methods for image enhancement". In: *Third International Conference on Image and Graphics (ICIG'04)*. 2004, pp. 27–30. DOI: 10.1109/ICIG.2004.50.

[4] A. Buades, B. Coll, and J.-M. Morel. "A non-local algorithm for image denoising". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 2. 2005, 60–65 vol. 2. DOI: 10.1109/CVPR.2005.38.

[5] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. "Non-Local Means Denoising". In: *Image Processing On Line* 1 (2011). https://doi.org/10.5201/ipol.2011.bcm_nlm, pp. 208–212.

[6] Antoni Buades, Bartomeu Coll, and Jean-Michel Morel. "On image denoising methods". In: 2004. URL: https://api.semanticscholar.org/CorpusID:17780447.

[7] Jie Chen et al. "The Comparison and Application of Corner Detection Algorithms." In: *Journal of multimedia* 4.6 (2009).

[8] B Chitradevi and P Srimathi. "An overview on image processing techniques". In: *International Journal of Innovative Research in Computer and Communication Engineering* 2.11 (2014), pp. 6466–6472.

[9] Konstantinos G Derpanis. "The harris corner detector". In: *York University* 2 (2004), pp. 1–2.

[10] Chris Harris, Mike Stephens, et al. "A combined corner and edge detector". In: *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[11] Yuka Ikarashi et al. "Exocompilation for Productive Programming of Hardware Accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718. ISBN: 9781450392655. DOI: 10.1145/3519939.3523446. URL: https://doi.org/10.1145/3519939.3523446.

[12] N. M. Kwok et al. "Intensity-based gain adaptive unsharp masking for image contrast enhancement". In: *2012 5th International Congress on Image and Signal Processing*. 2012, pp. 529–533. DOI: 10.1109/CISP.2012.6469772.

[13] Angela Pohl et al. "A Performance Analysis of Vector Length Agnostic Code". In: *2019 International Conference on High Performance Computing Simulation (HPCS)*. 2019, pp. 159–164. DOI: 10.1109/HPCS48598.2019.9188238.

[14] A. Polesel, G. Ramponi, and V.J. Mathews. "Image enhancement via adaptive unsharp masking". In: *IEEE Transactions on Image Processing* 9.3 (2000), pp. 505–510. DOI: 10.1109/83.826787.

[15] Jonathan Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *Acm Sigplan Notices* 48.6 (2013), pp. 519–530.

[16] Jonathan Ragan-Kelley et al. "Halide: Decoupling Algorithms from Schedules for High-Performance Image Processing". In: *Commun. ACM* 61.1 (Dec. 2017), pp. 106–115. ISSN: 0001-0782. DOI: 10.1145/3150211. URL: https://doi.org/10.1145/3150211.

[17] Javier Sánchez, Nelson Monzón, and Agustın Salgado De La Nuez. "An analysis and implementation of the harris corner detector". In: *Image Processing On Line* (2018).