Experimental Implementations of Stereo Matching Algorithms in Halide

by

Min Zhang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science

Certified by..
Saman Amarasinghe
Professor
Thesis Supervisor

Accepted by..
Albert R. Meyer
Chairman, Masters of Engineering Thesis Committee

Experimental Implementations of Stereo Matching Algorithms in Halide

by

Min Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on , in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

Currently, most stereo matching algorithms focus their efforts on increasing accuracy at the price of losing run-time performance. However, applications such as robotics require high performance stereo algorithms to perform real time tasks. The problem is due to the difficulty of hand optimizing the complicated stereo matching pipelines.

Halide is a programming language that has been widely used in writing high-performance image processing codes. In this work, we explore the usability of Halide in the area of real-time stereo algorithms by implementing several stereo algorithms in Halide. Because of Halide's ability to reduce the computation cost of dense algorithms, we focus on local dense stereo matching algorithms, including the simple box matching algorithm and the adaptive window stereo matching algorithms. Although we have found Halide's limitation in scheduling dynamic programming and recursive filters, our results demonstrate that Halide programs can achieve comparable performance as hand-tuned programs with much simpler and understandable code. Lastly, we also include a design solution to support dynamic programming in Halide.

Thesis Supervisor: Saman Amarasinghe
Title: Professor

Acknowledgments

I would like to express my sincere gratitude to my adviser Saman Amarasinghe, for his constant guidance and insights throughout my research. I would like to thank Shoaib Kamil for his timely feedbacks and advisory throughout the many discussions which helped me out when I get stuck. I would also like to thank Sudeep Pillai for sharing his insights on stereo algorithms with me when I was looking for the appropriate algorithms in this field to study. I would also like to thank Gaurav Chaurasia for his inputs on writing high performance Halide programs when I was new to this field. Without any of you, this thesis would not be possible.

Finally I would like to thank my family and friends for their constant love and support throughout this entire process.

Contents

1 Introduction **11**

2 Background **13**

2.1 Stereo Matching Algorithms 13

2.2 Halide 16

3 Implementing Stereo Algorithms **19**

3.1 Box Filter 19

3.2 The Box Matching Algorithm (stereoBM) 21

3.2.1 Algorithm Description 21

3.2.2 Scheduling 23

3.3 The Guided Filter Matching Algorithm (stereoGF) 31

3.3.1 Guided Filter 31

3.3.2 Algorithm Description 34

3.3.3 Scheduling 35

4 Experimental Results **37**

4.1 Evaluation of stereoBM Implementations 37

4.2 Evaluation of stereoGF Implementations 42

5 Design for Supporting Dynamic Programming in Halide **45**

6 Conclusion **49**

List of Figures

2-1 Example Inputs and Output of Stereo Matching Algorithms. The two input images are rectified. (c) is the ground truth depth image where the depth information is represented by color. Blue corresponds to large disparity values. Red corresponds to small disparity values. . . . 13

2-2 A Simple Example Stereo Matching Pipeline 14

3-1 Box filter defined as two-pass sum, $r = 2$ 20

3-2 Box filter using integral image technique, $r = 2$ 20

3-3 Schedule 1 of stereobm, $r = 2$. For simplicity, we only split the `y` direction in this figure, but the actual schedule splits both `x, y` directions. Tiles are evaluated in parallel. Red pixels are the pixels being evaluated at the current iteration. Yellow pixels are the ones used at this iteration. Blue pixels are the ones stored but not used at this iteration. 23

3-4 Schedule 2 of stereobm, $r = 2$. Tiles are evaluated in parallel. For simplicity, we only visualize computation of the top-left tile. Green pixels are the overlapping region between tiles. The `d` dimensions of **`vsum`** and **`cSAD`** are not plotted because their intermediate memory allocations have width 1 along the `d` dimension. 25

3-5 Schedule 3 of stereobm, $r = 2$. Tiles are evaluated in parallel. 27

3-6 Schedule 4 of stereobm, $r = 2$. Tiles are evaluated in parallel. The innermost dimension `di` is plotted here. All computations are vectorized along `di`. 28

3-7 Performances of Schedule 2, 3, 4 of stereoBM for different window sizes 30

4-1 Accuracies of Halide and OpenCV's implementations of stereoBM . . . 38
4-2 Runtime of Halide and OpenCV's implementations of stereoBM . . . 38
4-3 The throughput of Halide and OpenCV's implementations of stereoBM vs. number of cores . . . 39
4-4 The throughput of Halide and OpenCV's implementations of stereoBM vs. resolution scales. Q, H, F stand for quarter resolution, half resolution and full resolution, respectively. . . . 40
4-5 The throughput of Halide and OpenCV's implementations of stereoBM vs. disparity range . . . 41
4-6 The best runtimes of optimized Halide implementation for quarter-resolution images . . . 42
4-7 The speedup curves of stereoGF for quarter-resolution images vs. number of cores . . . 43
4-8 Performances of stereoGF for images with different resolutions vs. number of cores . . . 44
4-9 Performances of stereoGF vs. disparity range . . . 44

Chapter 1

Introduction

Stereo matching has been well studied as many vision applications require high-accuracy dense disparity maps in real-time. However, most state-of-the-art stereo algorithms have focused their efforts on improving the algorithm accuracy and avoided optimizing the run-time performance. Most top ranked submissions in the Middelbury and KITTI datasets take seconds or even minutes to process a 720×480 image [6, 12], which is far from meeting the requirements of real-time applications. While many implementations are unoptimized for run-time, the performance difference between a naive implementation of a given algorithm and a highly optimized one can be an order of magnitude or more.

The reason behind avoiding optimizing for performance is the complexity of state-of-the-art stereo matching algorithms. Many are almost infeasible to optimize by hand. Aside from the challenge of writing in a low-level programming language like C, efficient implementations also require global optimization of computation organization and data structures based on the entire pipeline, instead of simply combining the locally optimized components. Thus, the complexity of optimizing an algorithm grows exponentially with the length of the algorithm. Even for the simplest box matching algorithm, although the OpenCV implementation (stereobm) achieves peak performance by hand-writing parallel and vectorized code in low-level C, the implementation is over 1000 lines long and methods are twisted together which makes it extremely difficult to understand or modify for customized usage. Hand optimizing

the state-of-the-art stereo algorithms would be a formidable task, considering the complexity of those algorithms.

Halide is a new programming language designed as a tool to make it easy to write high performance code for image processing tasks [9]. It provides options for programmers to write simple code to explore different optimization tricks such as storage and computation reordering, parallelism and vectorization. In respect to such structure, the power of Halide is maximized for dense algorithms where identical computations are applied to all pixels. In applications such as bilateral filtering and local laplacian, Halide shows 2-4$\times$ speedup comparing to the expert tuned code while the expert tuned code is around 4$\times$ longer than the Halide programs.

Although it has already shown great success in many image processing applications both in simplifying the implementation process and achieving the same or better performance as hand-optimized code, it is rarely applied in vision algorithms. As we have discussed, real-time stereo algorithms are in great demand while most stereo algorithms do not have an optimized implementation. In this thesis, we explore the opportunities of developing high performance implementations of stereo matching algorithms in Halide. We particularly choose to study dense stereo algorithms which use the local approach since Halide is best suited for such algorithms.

We begin this thesis by a review of current state-of-the-art stereo matching algorithms and a brief introduction to Halide to explain why we choose dense stereo algorithms to study. In chapter 3, we discuss the implementation details of the two algorithms we implemented, stereoBM and stereoGF. In chapter 4, we compare the performance of our implementation with other hand-written implementations. Lastly, in chapter 5, we discuss the problems we encountered when developing Halide programs for stereo matching algorithms and propose potential solutions for future work.

Chapter 2

Background

2.1 Stereo Matching Algorithms

(a) Input image taken from left viewpoint (b) Input image taken from right viewpoint (c) Output depth image (ground truth)

Figure 2-1: Example Inputs and Output of Stereo Matching Algorithms. The two input images are rectified. (c) is the ground truth depth image where the depth information is represented by color. Blue corresponds to large disparity values. Red corresponds to small disparity values.

Stereo matching studies the problem of extracting depth information from two images from different viewpoints. Given two images taken from different viewpoints, stereo matching algorithms output the correspondence between pixels in the two images. To simplify the problem, the input images are usually rectified to align the epipolar lines with the horizontal coordinate axis so that the matching point of a given point (x_0, y_0) in one image lies on the horizontal line $y = y_0$ in the other image [5]. The disparity level (depth) for a given point is then defined as its distance from

its matching point. The output of the stereo matching algorithms in this case is a disparity map mapping each pixel to its disparity level. Figure 2-1 shows an example of inputs and outputs of stereo matching algorithms.

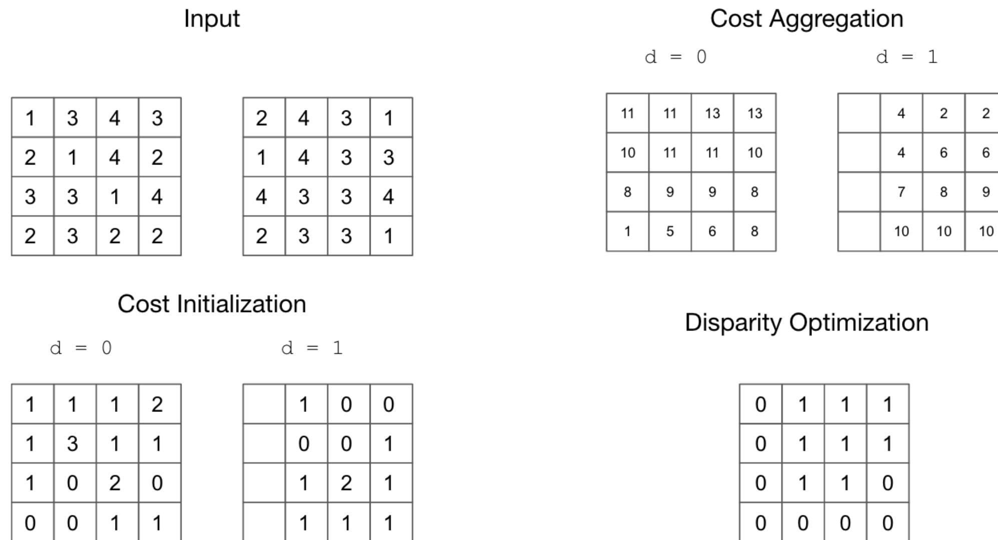


Figure 2-2: A Simple Example Stereo Matching Pipeline

For the rest of this section, we provide an overview of existing stereo matching algorithms. As [13] suggests, classical stereo matching methods generally consists of the following four steps. Figure 2-2 is a simple example stereo matching pipeline with 4×4 gray scale input images.

1. **Cost Initialization**, which generates a 3D cost volume by calculating the matching costs for assigning different disparity levels to different pixels. In the example, the cost of matching two pixels are defined as the absolute difference between their values and the disparity range is from 0 to 1.

2. **Cost Aggregation**, which aggregates the initial matching costs spatially for each pixel. In the example, the aggregated cost for a pixel is calculated as sum of the costs in a 3×3 squared window centered at that pixel with repeated boundary condition.

3. **Disparity Optimization**, which selects the best disparity hypothesis for each

pixel to minimize a local or global cost function. In the example, we simply select the disaprity level with the lowest cost.

4. **Disparity Refinement**, which applies post-processing to the disparity maps generated from the last step to remove discrepancies or provide sub-pixel estimates.

Depending on the disparity optimization step, stereo matching algorithms can be divided into two categories, local and global methods. Global methods work by adding a pairwise smoothness term to the cost function which enforces spacial continuity across pixels and aligned assignments across edges [10]. Although global methods generally perform better than local methods in handling object boundaries and avoiding unambiguous matching, these advantages come at a price. The global optimization problem is usually NP-hard. An estimated solution usually involves using global energy minimization approaches such as graph cut or belief propagation [4, 15], both require expensive computation that are non-parallelizable. As a result, most existing real-time algorithms still rely on the simple winner-takes-all (WTA) approach for disparity optimization. For the same reason, we focus our efforts on local methods in this work.

The cost aggregation step can be considered as applying a spacial filter to the cost volume [10]. Which filter to use highly affects the performance of the resulting stereo matching algorithm. The most straightforward approach is to use a box filter, which uses the average of costs of assigning disparity d to all pixels within a squared window centered at pixel p as the cost of assigning d to p. This is referred to as the box matching algorithm. The implicit assumption of this method is that all pixels in the squared window share similar disparities, which does not hold at boundaries of objects, resulting in blurred edges in the generated disparity maps.

To overcome the above limitations, Yoon and Kweon [17] first adopted the edge-preserving *bilateral filter* for cost aggregation and showed that their algorithm outperformed many global optimization based approaches. However, the naive implementation of bilateral filter is very time consuming. Richard et al. [11] provided a real-time

implementation as an approximate of the bilateral filter, although the approximation degrades the accuracy of the algorithm, diminishing the edge-preserving advantages of the bilateral filtering. To overcome this problem, Rhemann et al. [10] proposed a solution that uses a *guided filter* to smooth the cost volume. The guided filter was first introduced by He et al. in [3] as an alternative to the bilateral filter. It not only provides the edge-preserving property but also can be implemented efficiently. Rhemann et al. showed that by using the guided filter, the local WTA method could achieve state-of-the-art performance.

In this work, we implement two algorithms in Halide, the simple box matching algorithm (stereoBM) and Rhemann et al.'s algorithm using guided filter (stereoGF). We specifically choose these two algorithms because they serve as good representatives of the family of those stereo algorithms which use different filtering techniques for cost aggregation and then use the local WTA approach. StereoBM is the simplest algorithm in this family and is used as one component in many more complicated stereo algorithms such as SGBM (semi-global box matching) while stereoGF is the algorithm that performs the best among all existing local approaches which have reasonable runtime. We believe that if Halide could perform well on these two algorithms, it is strong evidence that Halide could work on other stereo algorithms using the local approach as well.

2.2 Halide

Halide is an open source programming language that is originally designed for writing efficient image processing code that takes advantages of memory locality, vectorization computation and parallelism on CPUs and GPUs [8, 9]. Comparing with traditional hand-tuned codes, programs written in Halide are usually much simpler, many times faster, and highly portable. The front-end of Halide is embedded in C++, and it can be compiled to back-end languages including x86/SSE, ARM v7/NEON, CUDA, Native Client, and OpenCL on different architectures from mobile platforms, multicore SIMD to GPUs.

The main principal of Halide is to decouple the definition of the algorithm from the way how its calculation is organized [8]. This decoupling brings two benefits: i) the algorithm definition is simplified since irrelevant information such as boundary conditions is eliminated; ii) programmers can explore scheduling strategies separately without worrying about changing the algorithm. The Halide scheduling language provide options to adjust storage granularity and compute granularity to achieve the right tradeoff between parallelism, locality and redundant recomputation.

As a result of the above structure, Halide target in applications where iterated homogeneous computations are applied to high dimensional data points because techniques such as reodering data dimensions, interleaving memory allocations between stages and vectorization work the best on such computation and data structure. For example, Halide perform extremely well on image processing applications in the form of graphs of stencil computations. On the contrary, sparse algorithms such as [7] are not in Halide's target domain because of their heterogeneous computation structure. This is also one of the reasons that we restrict our attention to dense local stereo matching algorithms.

We do not include the details of the Halide language here. More information about Halide design can be found in [8, 9]. A tutorial of Halide's syntax and usage can be found in [1].

Chapter 3

Implementing Stereo Algorithms

Before we describe the implementation details of the stereoBM algorithm and the stereoGF algorithm, we will begin this chapter by discussing different ways to implement box filter, as it is involved in both algorithms.

3.1 Box Filter

The most straight forward implementation would be to use a two-pass approach – the first pass computes the sum along the horizontal scanline, while the second pass computes along the vertical scanline. Figure 3-1 shows a illustrative interpretation of this algorithm.

```
RDom rk(-r, 2*r+1, "rk");
Func vsum("vsum"), box_sum("box_sum");
vsum(x, y, d)    = sum(in(x + rk, y, d));
box_sum(x, y, d) = sum(vsum(x, y + rk, d));
```

A valid schedule which exploits both parallelism and memory locality for the above algorithm is

```
box_sum.compute_root()
       .tile(x, y, xo, yo, xi, yi, x_tile_size, y_tile_size)
       .vectorize(xi, vector_width).parallel(yo).parallel(xo);
vsum   .compute_at(box_sum, xo).vectorize(xi, vector_width);
```

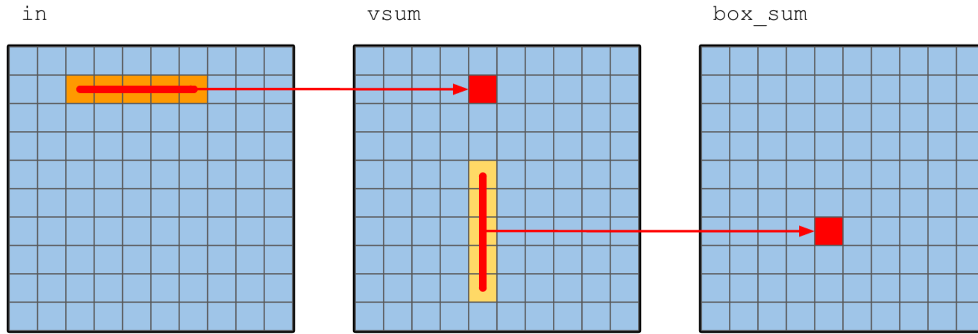


Figure 3-1: Box filter defined as two-pass sum, $r = 2$

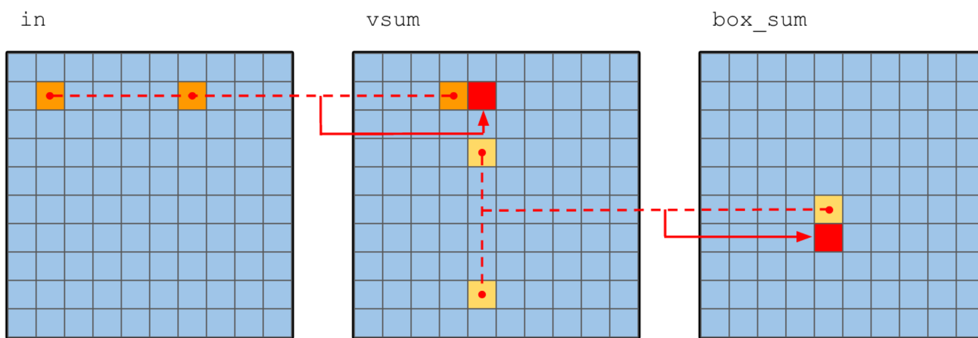


Figure 3-2: Box filter using integral image technique, $r = 2$

The above strategy achieves quite good locality and parallelism by interleaving the computation of **box_sum** and **vsum** at the level of tiles. However, it has a major pitfall: redundant recomputation. To compute **box_sum** and **vsum**, the program needs to do $2r + 1$ additions at each pixel. As in both stereoBF and stereoGF, the box filters used have large window size, the above strategy is not satisfying.

A more efficient way is to use the integral image technique [2] as follows.

```
vsum(x,y,d)     = vsum(x-1,y,d)     + in(x,y,d)   - in(x-r-1,y,d);
box_sum(x,y,d) = box_sum(x,y-1,d) + vsum(x,y,d) - vsum(x,y-r-1,d);
```

The complete algorithm in Halide is,

```
RDom rk(-r, 2*r+1, "rk");
RDom rx(1, width-1, "rx"), ryi(1, height-1, "ry");
vsum(x,y,d)  = undef<ushort>();
vsum(0,y,d)  = sum(in(rk, y, d));
```

```
vsum(rx,y,d) = vsum(rx-1,y,d) + in(rx,y,d) - in(rx-r-1,y,d);

box_sum(x,y,d)  = undef<ushort>();
box_sum(0,y,d)  = sum(vsum(rk,y,d));
box_sum(x,ry,d) = box_sum(x,ry-1,d) + vsum(x,ry,d)
                                    - vsum(x,ry-r-1,d);
```

We will discuss various scheduling strategies for the box filter at the next section since it is important to consider the entire pipeline when seeking for the optimal performance.

3.2 The Box Matching Algorithm (stereoBM)

3.2.1 Algorithm Description

As we introduced earlier, stereoBM is the simplest stereo matching algorithm using the local approach. Like all local method, it has four stages: cost initialization, cost aggregation, disparity optimization and disparity refinement. It uses the WTA method for disparity optimization and box filtering for cost aggregation. Implementation details of initialization and disparity refinement may vary between different implementations. In our work, we follow OpenCV's implementation details since we use it as a baseline for performance comparison. We first apply a Sobel filter [14] to compute the gradient of intensities along the x axis and then compute the cost of assigning disparity level d to pixel (x, y) as the absolute difference between gradients of pixel (x, y) at the left image and pixel $(x - d, y)$ at the right image, i.e.,

$$C(x, y, d) = \left| \frac{\partial I_0}{\partial x}(x, y) - \frac{\partial I_1}{\partial x}(x - d, y) \right|.$$

For disparity refinement, we simply run a left right consistency check to detect mismatch.

The entire pipeline is in the following.

```
// prefilter the image using the Sobel filter
```

```
Var x("x"), y("y"), c("c"), d("d");
Func gradient0 = prefilterXSobel(I0, width, height);
Func gradient1 = prefilterXSobel(I1, width, height);

// cost initialization
Func diff("diff");
diff(x,y,d) = cast<ushort>(abs(gradient0(x,y) - gradient1(x-d,y)));

// cost aggregation
Func vsum("vsum"), cSAD("cSAD");
RDom rk(-r, 2*r+1, "rk");
RDom rx(1, width-1, "rx"), ryi(1, height-1, "ry");
vsum(x,y,d)  = undef<ushort>();
vsum(0,y,d)  = sum(diff(rk, y, d));
vsum(rx,y,d) = vsum(rx-1,y,d) + diff(rx,y,d) - diff(rx-r-1,y,d);

cSAD(x,y,d)  = undef<ushort>();
cSAD(x,0,d)  = sum(vsum(x,ry,d));
cSAD(x,ry,d) = cSAD(x,ry-1,d) + vsum(x,ry,d) - vsum(x,y-ry-1,d);

// disparity optimization
RDom rd(minDisparity, numDisparities);
Func disp_left("disp_left");
disp_left(x, y) = {cast<ushort>(minDisparity), cast<ushort>((2<<16)-1)};
disp_left(x, y) = tuple_select(
        cSAD(x, y, rd) < disp_left(x, y)[1],
        {cast<ushort>(rd), cSAD(x, y, rd)},
        disp_left(x, y));

// diparity refinement
Func disp("disp");
disp(x, y) = disp_left(x, y)[0];
```

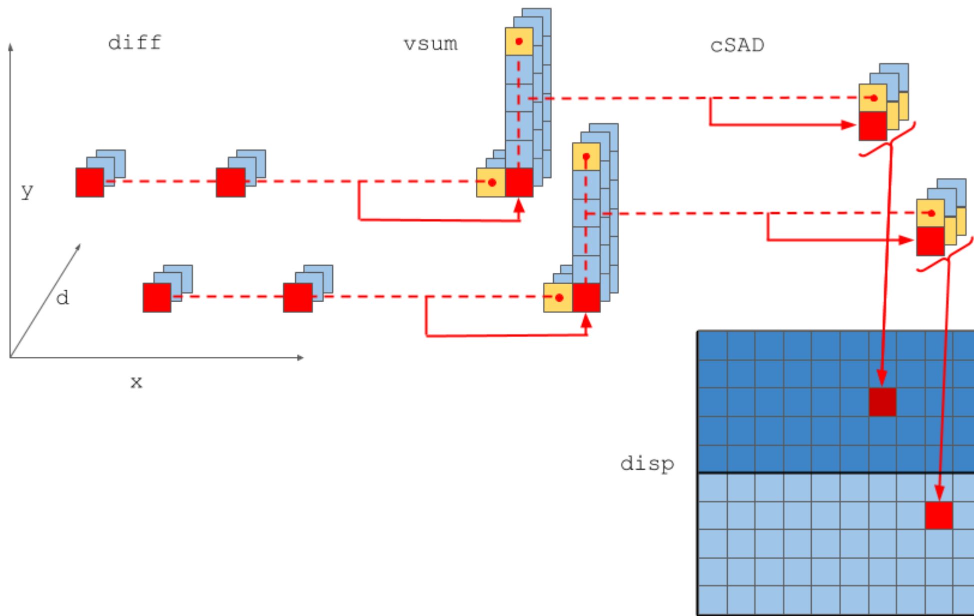


Figure 3-3: Schedule 1 of stereobm, $r = 2$. For simplicity, we only split the `y` direction in this figure, but the actual schedule splits both `x, y` directions. Tiles are evaluated in parallel. Red pixels are the pixels being evaluated at the current iteration. Yellow pixels are the ones used at this iteration. Blue pixels are the ones stored but not used at this iteration.

3.2.2 Scheduling

The most difficult part of scheduling the stereoBM pipeline lies on how to schedule the box filter. An optimal schedule needs to make tradeoffs between locality, parallelism and computation. We have tried different schedules during implementation and we will list all of them here and discuss the advantages and drawbacks for each one of them.

Schedule 1

Note that to compute **vsum**`(x,y,d)` only requires **vsum**`(x−1,y,d)`, **in**`(x,y,d)`, and **in**`(x−r−1,y,d)` and that to compute **cSAD**`(x,y,d)` also only requires **cSAD**`(x,y−1,d)`, **vsum**`(x,y,d)` and **vsum**`(x,y−r−1,d)`. Thus, we can use a sliding window to store **vsum**`(x,y−r−1...y−1,d)`. We compute **diff** on demand instead of memoizing it because computing **diff** takes little computation.

The schedule is illustrated in Figure 3-3. The pseudo-code is in Algorithm 1.

```
for y = ymin...ymax do
    for x = xmin...xmax do
        for d = dmin...dmax do
            calculate diff(x, y, d)
            calculate vsum(x, y, d)
            calculate cSAD(x, y, d)
            if cSAD(x, y, d) < minDisp(x, y) then
                disp(x, y) = d
                minDisp(x, y) = cSAD(x, y, d)
            end
        end
    end
end
```

Algorithm 1: Pseudocode for Schedule 1 of stereoBM

We omit parallelization and vectorization code here to keep the pseudo-code terse. To parallelize the process, we tile the whole image to small tiles and carry the above computation strategy in parallel for all tiles. For vectorization, we simply vectorize the innermost dimension `d`.

The above schedule achieves near-maximal locality as the program only allocates about `(2r+2)×diparities` for intermediate memory and most functions are consumed immediately after it is produced. By choosing reasonable tile sizes according to the image size and the window size, it can also achieve near-optimal parallelism and negligible redundant recomputation. In fact, this is the schedule that OpenCV uses for their stereobm implementation.

Unfortunately, the above schedule cannot be expressed by the current Halide language. In this schedule, we keep **`vsum`**`(x, y−r−1...y−1, d)` to compute **`cSAD`**`(x, y, d)`, i.e., the footprint of **`cSAD`** is variable as the algorithm iterates through y. However, Halide only supports constant footprints. We will discuss this again in Section 5 and propose possible solutions to resolve this issue.

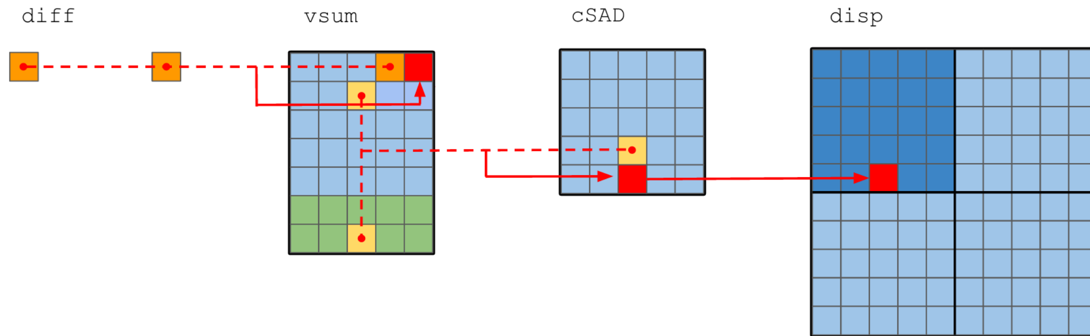


Figure 3-4: Schedule 2 of stereobm, $r = 2$. Tiles are evaluated in parallel. For simplicity, we only visualize computation of the top-left tile. Green pixels are the overlapping region between tiles. The d dimensions of **vsum** and **cSAD** are not plotted because their intermediate memory allocations have width 1 along the d dimension.

Schedule 2

Another strategy is to put d to the outermost dimension, tile the image and compute **vsum** and **cSAD** at the tile level. Figure 3-6 illustrates the schedule. The code of the schedule also follows.

```
disp.compute_root()
    .tile(x, y, xo, yo, xi, yi, x_tile_size, y_tile_size)
    .vectorize(xi, vector_width)
    .parallel(yo).parallel(xo);

disp_left.compute_at(disp).vectorize(x, vector_width)
         .update()
         .reorder(x, y, rd).vectorize(x, vector_width);

cSAD.compute_at(disp_left, rd).vectorize(x, vector_width)
    .update() .vectorize(x, vector_width);
cSAD.update(1).reorder(x, ry, d).vectorize(x, vector_width);

vsum.compute_at(disp_left, rd).vectorize(x, vector_width);
```

In fact, the above schedule will not work in Halide and we need to do some tricks to make it work. The bound inference process in Halide will infer that the footprints

of **vsum** and **cSAD** are the whole image because of the range of the reduction domains `rx`, `ry` in their update definitions. Thus, even though our desired schedule is to compute **vsum** and **cSAD** at each tile, the actual code produced by the above schedule would compute **vsum** and **cSAD** for the whole image at each tile, resulting in unnecessary redundant recomputation. To solve the above problem, we need to modify the definitions of **vsum** and **cSAD** to explicitly tile the image in the function definitions as follows, by which we force Halide to maintain the footprints of **vsum** and **cSAD** at the tile level.

```
RDom rx(1, x_tile_size-1, "rx"), ryi(1, y_tile_size-1, "ry");
vsum(xi,yi,xo,yo,d) = undef<ushort>();
vsum(0, yi,xo,yo,d) = sum(diff(rk,yi,xo,yo,d));
vsum(rx,yi,xo,yo,d) = vsum(rx-1,yi,xo,yo,d) + diff(rx,yi,xo,yo,d)
                                  - diff(rx-r-1,yi,xo,yo,d);

cSAD(xi,yi,xo,yo,d) = undef<ushort>();
cSAD(xi,0, xo,yo,d) = sum(vsum(xi,ry,xo,yo,d));
cSAD(xi,ry,xo,yo,d) = cSAD(xi,ry-1,xo,yo,d) + vsum(xi,ry,xo,yo,d)
                                  - vsum(xi,y-ry-1,xo,yo,d);
```

This schedule achieves a good balance between locality and parallelism if we choose reasonable tile sizes. The pitfall in this schedule is that it does not vectorize the second update stage of **vsum** because the innermost dimension `rx` is recursive and vectorizing `y` yields no performance benefit. By not vectorizing this stage, we lose the significant improvement in performance that would have been provided by vectorization.

There are two other scheduling options in which we can vectorize **vsum**,

- Use the simple sum definition instead of the image integral trick for **vsum**.
- Split `d` to two dimensions `d_i` and `d_o` and reorder the storage such that `d_i` is the innermost dimension. This way, we can vectorize `d_i`.

We will discuss these two options in Schedule 3 and 4.

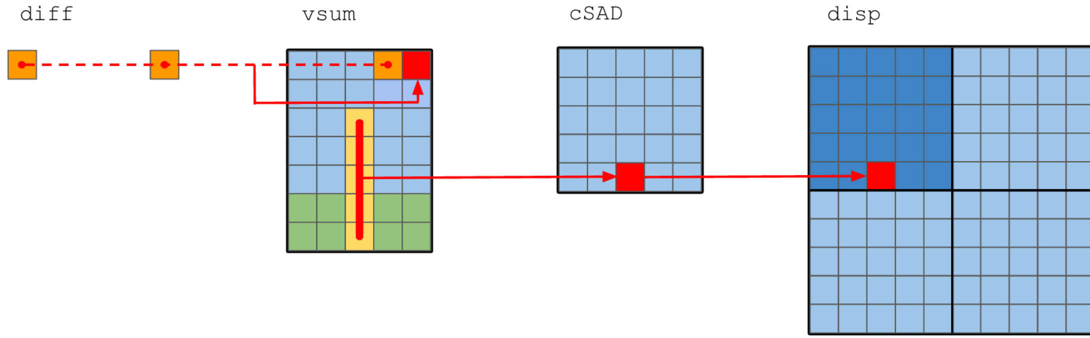


Figure 3-5: Schedule 3 of stereobm, $r = 2$. Tiles are evaluated in parallel.

Schedule 3

The schedule is similar to Schedule 2, except that we change the definition of **vsum** to summations to vectorize **vsum**. The scheduling code follows. This schedule also achieves good locality and parallelism. Its disadvantage results from the redundant recomputation caused by the sum definition of **vsum**. As the window size increases, the schedule will result in longer runtime.

```
RDom rx(1, x_tile_size-1, "rx"), ryi(1, y_tile_size-1, "ry");
vsum(xi,yi,xo,yo,d) = sum(diff(rk,yi,xo,yo,d));

cSAD(xi,yi,xo,yo,d) = undef<ushort>();
cSAD(xi,0, xo,yo,d) = sum(vsum(xi,ry,xo,yo,d));
cSAD(xi,ry,xo,yo,d) = cSAD(xi,ry-1,xo,yo,d) + vsum(xi,ry,xo,yo,d)
                                        - vsum(xi,y-ry-1,xo,yo,d);

vsum.compute_at(disp_left, rd)
    .reorder(xi,  yi, xo, yo, d).vectorize(xi, vector_width);
cSAD.compute_at(disp_left, rd)
    .reorder(xi,  yi, xo, yo, d).vectorize(xi, vector_width)
    .update()
    .reorder(xi, xo, yo, d).vectorize(xi, vector_width);
cSAD.update(1)
    .reorder(xi, ryi, xo, yo, d).vectorize(xi, vector_width);
```

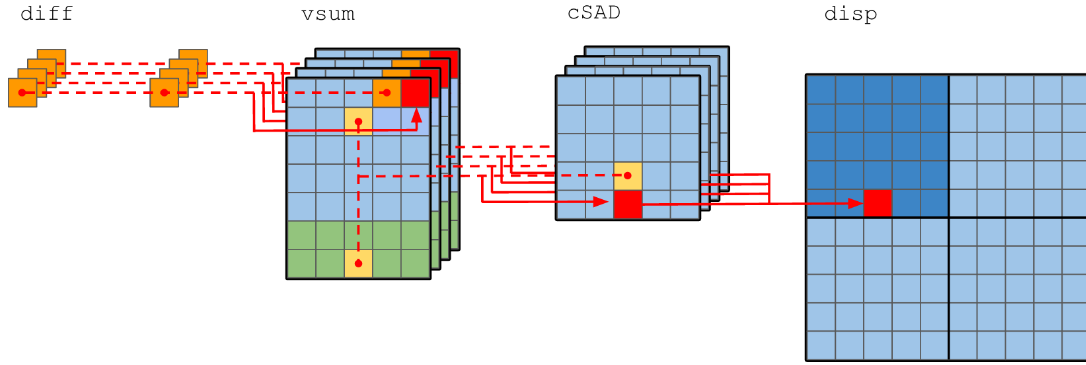


Figure 3-6: Schedule 4 of stereobm, $r = 2$. Tiles are evaluated in parallel. The innermost dimension di is plotted here. All computations are vectorized along di.

Schedule 4

The key idea of this schedule is to split the dimension d into to dimensions di, d_o and make di the innermost dimension so that we can vectorize di. We include the pipeline in the following.

```
/***************************schedule 4**********************/
//cost aggregation
Expr d_ = di + d_o * vector_width + minDisparity;
vsum(di, xi, yi, xo, yo, d_o) = undef<ushort>();
vsum(di, xi, 0, xo, yo, d_o) = sum(diff_T(d_, xi, rk, xo, yo));
vsum(di, xi, ryi, xo, yo, d_o) = vsum(di, xi, ryi-1, xo, yo, d_o)
                               + diff_T(d_, xi, ryi+win2, xo, yo)
                               - diff_T(d_, xi, ryi-win2-1, xo, yo);

cSAD(di, xi, yi, xo, yo, d_o) = undef<ushort>();
cSAD(di, 0, yi, xo, yo, d_o) = sum(vsum(di, rk, yi, xo, yo, d_o));
cSAD(di, rxi, yi, xo, yo, d_o) = cSAD(di, rxi-1, yi, xo, yo, d_o)
                                 + vsum(di, rxi + win2, yi, xo, yo, d_o)
                                 - vsum(di, rxi - win2 -1, yi, xo, yo, d_o);
//disparity optimizaiton
RDom rd(0, vector_width, 0, numDisparities/vector_width);
Expr d_ = rd[0] + rd[1] * vector_width + minDisparity;
disp_left_raw(di, xi, yi, xo, yo) =
               {cast<ushort>(minDisparity), cast<ushort>((2<<16)-1)};
disp_left_raw(rd[0], xi, yi, xo, yo) = tuple_select(
  cSAD(rd[0], xi, yi, xo, yo, rd[1]) < disp_left_raw(rd[0], xi, yi, xo, yo)[1],
  {cast<ushort>(d_), cSAD(rd[0], xi, yi, xo, yo, rd[1])},
  disp_left_raw(rd[0], xi, yi, xo, yo));

RDom rdi(0, vector_width);
```

```
disp_left(xi, yi, xo, yo) =
        {cast<ushort>(minDisparity), cast<ushort>((2<<16)-1)};
disp_left(xi, yi, xo, yo) = tuple_select(
        disp_left_inter(rdi, xi, yi, xo, yo)[1] < disp_left(xi, yi, xo, yo)[1],
        disp_left_inter(rdi, xi, yi, xo, yo),
        disp_left(xi, yi, xo, yo));

disp(x, y) = select(
        x>xmax-xmin || y>ymax-ymin,
        cast<ushort>(FILTERED),
        cast<ushort>(disp_left(x%x_tile_size, y%y_tile_size,
                               x/x_tile_size, y/y_tile_size)[0])
                  );

//schedule
disp_left.reorder_storage(xi, yi, xo, yo);
vsum     .reorder_storage(di, xi, yi, xo, yo, d_o);
cSAD     .reorder_storage(di, xi, yi, xo, yo, d_o);

disp_left.compute_at(disp, xo)
         .reorder(xi, yi, xo, yo).vectorize(xi, vector_width)
         .update()
         .reorder(rdi, xi, yi, xo, yo).unroll(rdi);
disp_left_raw.compute_at(disp_left, xo)
             .reorder(di, xi, yi, xo, yo).vectorize(di, vector_width)
             .update()
             .reorder(rd[0], xi, yi, xo, yo, rd[1]).vectorize(rd[0]);

cSAD.compute_at(disp_left_inter, xo)
    .reorder(di, xi, yi, xo, yo, d_o) .vectorize(di, vector_width)
    .update()
    .reorder(di, yi, xo, yo, d_o)     .vectorize(di, vector_width);
cSAD.update(1)
    .reorder(di, rxi, yi, xo, yo, d_o).vectorize(di, vector_width);
vsum.compute_at(disp_left_inter, xo)
    .reorder(di, xi, yi, xo, yo, d_o) .vectorize(di, vector_width)
    .update()
    .reorder(di, xi, xo, yo, d_o)     .vectorize(di, vector_width);
vsum.update(1)
    .reorder(di, xi, ryi, xo, yo, d_o).vectorize(di, vector_width);
```

The pitfall of this schedule is that it uses more intermediate memory than the other schedules. In schedule 4, **vsum** and **cSAD** both need `vector_width` $\times$ `x_tile_size` $\times$ `y_tile_size` memory while in schedule 2 and 3, they only need

`x_tile_size` × `y_tile_size` memory.

Choosing the schedule

Each of the above schedule has its own advantages and drawbacks. To decide which schedule to use, we implemented all of them and tested their performances for different window sizes. Information of the test machine and the test cases is included in Section 4. We run the tests using 4 cores on 15 test images which have up to 750×500 pixels. The average runtime is shown in Figure 3.2.2.

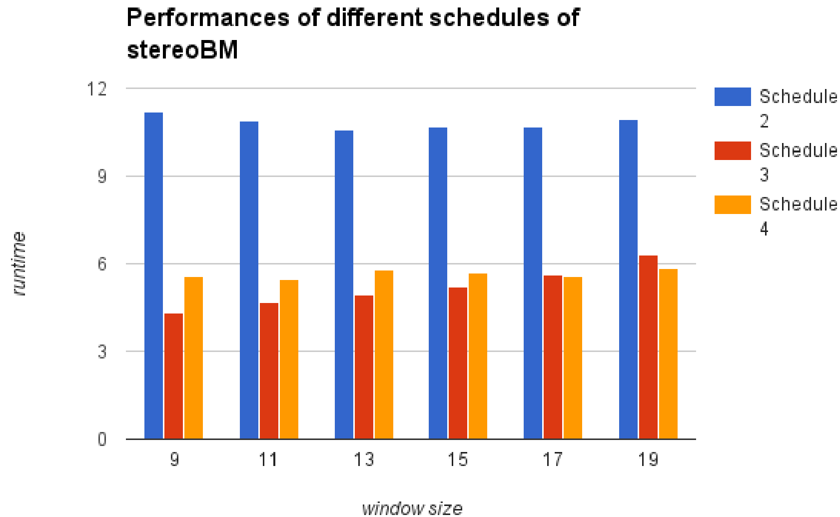


Figure 3-7: Performances of Schedule 2, 3, 4 of stereoBM for different window sizes

We observe that Schedule 2 is always outperformed by Schedule 3 and 4, which is expected since Schedule 2 bears the loss of no vectorization of **`vsum`**. Another important observation is that the runtime of Schedule 2 and 4 change very little by window size, while the runtime of Schedule 3 increases as window size increases. When window size is 9, Schedule 3 outperforms Schedule 4. When window size is 17, they have similar performance, and Schedule 4 outperforms Schedule 3 when window size is 19. In our setting, we usually set window size to 11, thus we choose Schedule 3.

3.3 The Guided Filter Matching Algorithm (stereoGF)

In this section we describe the stereoGF algorithm that was introduced in [10]. We start by introducing the guided filter.

3.3.1 Guided Filter

Guided filter is an edge preserving filter that was first introduced by [3]. It follows a local linear model, where the output of the filter at a pixel p is a weightd average of all pixels in a squared window centered at p. The weights are determined by the contents of a guidance image, which could be the filtered image itself or another image, in a way such that the weights reflect edges in the guidance image.

Denote the guided image as I and the image to be filtered as p. Then, the filtering output q at pixel $i = (x, y)$ is expressed as,

$$q_i = \sum_{j \in \omega_i} W_{i,j}(I) p_j, \tag{3.1}$$

where ω_i is a squared window with dimensions $(2r+1) \times (2r+1)$, centered at pixel i. The weights $W_{i,j}(I)$ is given by,

$$W_{i,j}(I) = \frac{1}{|\omega|^2} \sum_{k: i,j \in \omega_k} \left(1 + (I_i - \mu_k)^T (\sigma_k + \epsilon U)^{-1} (I_j - \mu_k)\right), \tag{3.2}$$

where μ_k and Σ_k are the mean and the covariance matrix of I in the window ω_k, $|\omega| = (2r+1)^2$ denotes the number of pixels in ω_k and ϵ is a smoothness parameter. Note that here I_i, I_j and μ_k are 3×1 color vectors and the covariance matrix Σ_k and the identity matrix U are 3×3 matrices.

[3] also showed that the guided filter can be implemented efficiently as a sequence of box filters. We also present the technique here because we are going to use it in our implementation.

The filter can be expressed in this way.

$$a_k = (\Sigma_k + \epsilon U)^{-1} \left(\frac{1}{|\omega|^2} \sum_{i \in w_k} I_i p_i - \mu_k \bar{p}_k \right), \tag{3.3}$$

$$b_k = \bar{p}_k - a_k^T \mu_k, \tag{3.4}$$

$$q_i = \bar{a}_i I_i + \bar{b}_i, \tag{3.5}$$

where $\bar{p}_k = \frac{1}{|\omega|} \sum_{i \in \omega_k} p_i$ is the mean of p in ω_k, $\bar{a}_i = \frac{1}{|\omega|} \sum_{j \in \omega_i} a_j$, and $\bar{b}_i = \frac{1}{|\omega|} \sum_{j \in \omega_i} b_j$. All summations here are box filters and can be computed efficiently using the techniques in Section 3.2. We include the Halide pipeline for the guided filter in the following.

```
Func left_gradient = gradientX(left);
Func right_gradient = gradientX(right);

Func cost_left("cost_left");
Func diff("diff");
diff(x, y, c, d) = abs(left(x, y, c) - right(x-d, y, c));
Expr color_diff = clamp((diff(x, y, 0, d) + diff(x, y, 1, d) + diff(x, y, 2, d))/3, 0, threshColor);
Expr gradient_diff = clamp(abs(left_gradient(x, y) - right_gradient(x-d, y)), 0, threshGrad);
cost_left(x, y, d) = (1 - alpha) * color_diff + alpha * gradient_diff;

Var x("x"), y("y"), c("c"), d("d");
float scale = 1.0f/(2*r+1)/(2*r+1);

Func mu = mean(I, r);
Func square("square");
Func ind2pair("ind2pair"), pair2ind("pair2ind");
ind2pair(c) = {0,c};
ind2pair(3) = {1,1};
ind2pair(4) = {1,2};
ind2pair(5) = {2,2};
pair2ind(c, d) = undef<int>();
pair2ind(0, 0) = 0;
pair2ind(0, 1) = 1;
pair2ind(0, 2) = 2;
pair2ind(1, 0) = 1;
pair2ind(1, 1) = 3;
pair2ind(1, 2) = 4;
pair2ind(2, 0) = 2;
pair2ind(2, 1) = 4;
```

```
pair2ind(2, 2) = 5;


Expr row = clamp(ind2pair(c)[0], 0, 2), col = clamp(ind2pair(c)[1], 0, 2);
square(x, y, c) = I(x, y, row) * I(x, y, col);
Func s_m = mean(square, r);

Func sigma("sigma");
sigma(x, y, c) = s_m(x, y, c) - mu(x, y, row) * mu(x, y, col);
Expr a11 = sigma(x, y, 0) + epsilon, a12 = sigma(x, y, 1), a13 = sigma(x, y, 2);
Expr a22 = sigma(x, y, 3) + epsilon, a23 = sigma(x, y, 4);
Expr a33 = sigma(x, y, 5) + epsilon;

Func inv_("inv_"), inv("inv");
inv_(x, y, c) = undef<float>();
inv_(x, y, 0) = a22 * a33 - a23 * a23;
inv_(x, y, 1) = a13 * a23 - a12 * a33;
inv_(x, y, 2) = a12 * a23 - a22 * a13;
inv_(x, y, 3) = a11 * a33 - a13 * a13;
inv_(x, y, 4) = a13 * a12 - a11 * a23;
inv_(x, y, 5) = a11 * a22 - a12 * a12;
Expr det = a11 * inv_(x, y, 0) + a12 * inv_(x, y, 1) + a13 * inv_(x, y, 2);
inv(x, y, c) = inv_(x, y, c) / det;

Func prod("prod");
prod(x, y, c, d) = I(x, y, c) * p(x, y, d);
Func prod_m = mean(prod, r);
Func p_m = mean(p, r);

Func temp("temp");
temp(x, y, c, d) = prod_m(x, y, c, d) - mu(x, y, c) * p_m(x, y, d);

Func a("a"), b("b");
RDom k(0, 3, "k");
a(x, y, c, d) = sum(inv(x, y, clamp(pair2ind(c, k), 0, 5)) * temp(x, y, k, d));
b(x, y, d) = p_m(x, y, d) - sum(a(x, y, k, d) * mu(x, y, k));

Func a_m = mean(a, r);
Func b_m = mean(b, r);

Func q("q");
q(x, y, d) = sum(a_m(x, y, k, d) * I(x, y, k)) + b_m(x, y, d);
```

3.3.2 Algorithm Description

The framework of this stereo matching algorithm still follows the local approach.

Cost Initialization: The matching cost is a combination of a truncated absolute difference of the color and the gradient at the matching pixels.

$$\begin{aligned} C(x, y, d) = (1-\alpha) \cdot \min(||I_0(x, y) - I_1(x-d, y)||, \tau_1) \\ +\alpha \cdot \min(||\frac{\partial I_0}{\partial x}(x, y) - \frac{\partial I_1}{\partial x}(x-d, y)||, \tau_2) \end{aligned} \tag{3.6}$$

Cost Aggregation: We then use the guided filter to filter the cost volume, using I_0 as the guidance image.

The complete pipeline expressed in Halide is in the following.

```
Var x("x"), y("y"), c("c"), d("d");
Func left_gradient = gradientX(left);
Func right_gradient = gradientX(right);

Func cost_left("cost_left"), cost_right("cost_right");
Func diff("diff");
diff(x, y, c, d) = abs(left(x, y, c) - right(x-d, y, c));
Expr color_diff = clamp((diff(x, y, 0, d) + diff(x, y, 1, d) + diff(x, y, 2, d))/3, 0, threshColor);
Expr gradient_diff = clamp(abs(left_gradient(x, y) - right_gradient(x-d, y)), 0, threshGrad);
cost_left(x, y, d) = (1 - alpha) * color_diff + alpha * gradient_diff;
cost_right(x, y, d) = cost_left(x + d, y, d);

Func filtered_left = guidedFilter(left, cost_left, r, epsilon);
Func filtered_right = guidedFilter(right, cost_right, r, epsilon);

RDom rd(0, numDisparities);
Func disp_left("disp_left"), disp_right("disp_right");
disp_left(x, y) = {0, INFINITY};
disp_left(x, y) = tuple_select(
        filtered_left(x, y, rd) < disp_left(x, y)[1],
        {rd, filtered_left(x, y, rd)},
        disp_left(x, y));

disp_right(x, y) = {0, INFINITY};
disp_right(x, y) = tuple_select(
        filtered_right(x, y, rd) < disp_right(x, y)[1],
        {rd, filtered_right(x, y, rd)},
        disp_right(x, y));
```

```
Func disp = postprocessing(disp_left, disp_right);
```

3.3.3 Scheduling

We will not include the complete schedule here because of the length of the pipeline. The general idea is to tile the image and use Schedule 3 in Section 3.2 for all box filters. We include the schedule of the major functions in the following. All other intermediate functions are either scheduled inline or at the tile level with the guideline of minimizing redundant recomputation and maximizing locality.

```
disp.compute_root().vectorize(x, vector_width);
disp_left.compute_root().vectorize(xi, vector_width)
         .update().reorder(xi, yi, rd, xo, yo)
         .vectorize(xi, vector_width)
         .parallel(yo).parallel(xo);
filtered_left.compute_at(disp_left, rd).vectorize(xi, vector_width);

left.compute_root().vectorize(x, vector_width);
mean_left.compute_root()
         .tile(x, y, xo, yo, xi, yi, x_tile_size, y_tile_size)
         .reorder(xi, yi, c, xo, yo).vectorize(xi, vector_width)
         .parallel(yo).parallel(xo);
inv_left.compute_root()
        .tile(x, y, xo, yo, xi, yi, x_tile_size, y_tile_size)
        .reorder(c, xi, yi, xo, yo).vectorize(xi, vector_width)
        .parallel(yo).parallel(xo);
cost_left.compute_at(disp_left, rd).vectorize(xi, vector_width);
```

Chapter 4

Experimental Results

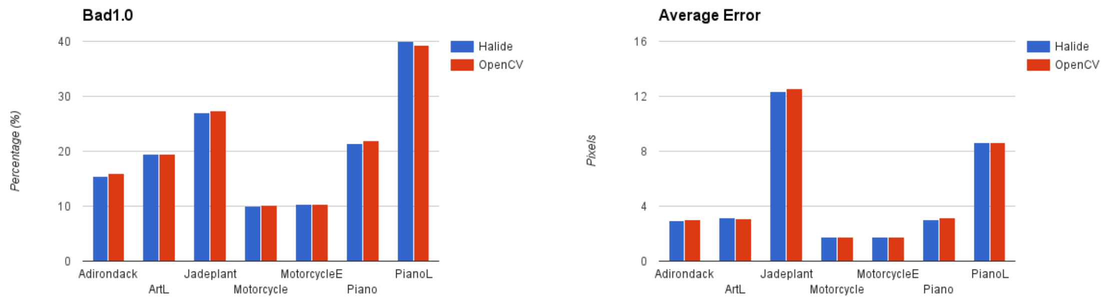
We conducted several experiments to evaluate the performances of the algorithms we implemented in Halide as discussed in Section 3. All experiments are performed on an Intel Xeon E5-2680 CPU with 48 cores. We use the 2014 Middlebury stereo evaluation datasets [12]. Since we did not modify the algorithms, we focus our attention on the runtime performance of our implementations. We include accuracy measurement of our implementation only to verify the correctness of our implementations.

4.1 Evaluation of stereoBM Implementations

To test the performance of our implementation of the stereo box matching algorithm (stereoBM), we use OpenCV's implementation cv::StereoBM as baseline. We choose Schedule 3 in section 3.2 as Halide's implementation to compare. We evaluate both implementations on their performances under different settings with different image sizes, disparity range and number of cores.

First we include the accuracy measurement of both implementations. Figure 4-1 shows that the output of these two implementations are essentially equivalent. The slight difference may be caused by difference between the two implementations in converting images to gray-scale.

The computation time required by both implementations is plotted in Figure 4-2. It is evaluated with window size 11×11 using 1, 4, 16, and 48 cores. The following

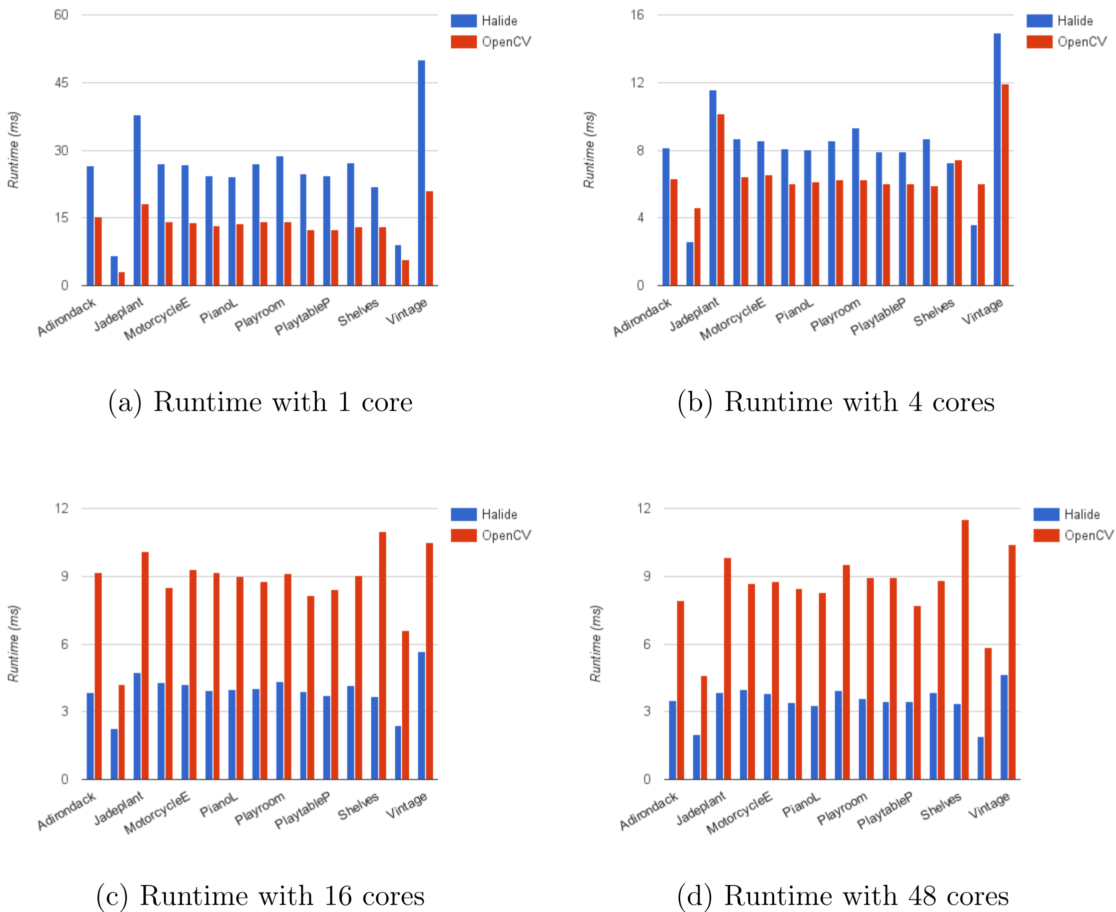


(a) Percentages of bad pixels with error threshold set to 1

(b) Average error between output disparity map and the ground truth

Figure 4-1: Accuracies of Halide and OpenCV's implementations of stereoBM

can be observed from the results.



(a) Runtime with 1 core

(b) Runtime with 4 cores

(c) Runtime with 16 cores

(d) Runtime with 48 cores

Figure 4-2: Runtime of Halide and OpenCV's implementations of stereoBM

- When using only a single core, OpenCV's implementation outperforms Halide's implementation, because OpenCV's implementation maximizes memory locality and minimize redundant recomputation, as we discussed in Section 3.2.
- The two implementations have similar performances with 4 cores and our implementation outperforms OpenCV's with more than 16 cores, indicating that the former exploits parallelism better than the latter. In OpenCV's implementation, the images are divided into stripes by splitting the y direction, while in our implementation, images are divided into tiles by splitting both x and y directions.

To further see the scalability of these two algorithms with different number of cores, we plot the relationship of their throughput vs. number of cores in Figure 4-3. The throughput is calculated as the number of megapixels times the number of disparities being processed per second. We can observe that Halide's implementation scales to 16 cores while OpenCV's implementation only scales to 4 cores and its performance drops from 4 cores to 8 cores.

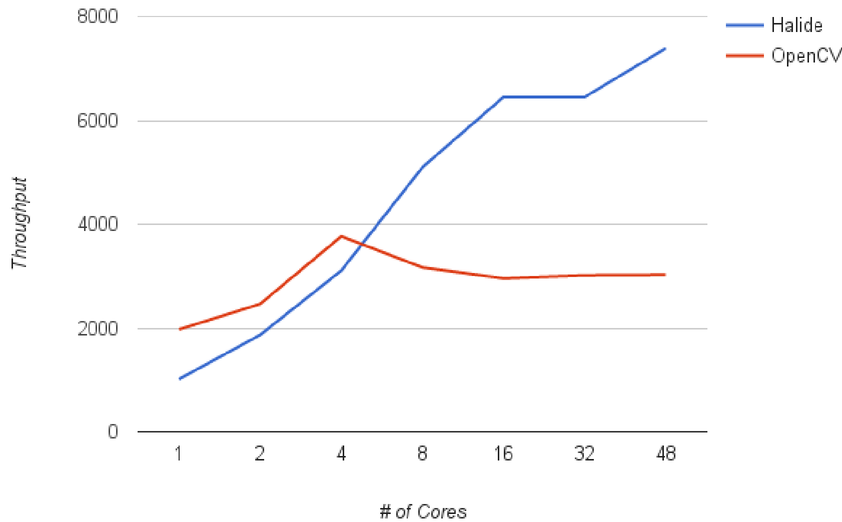
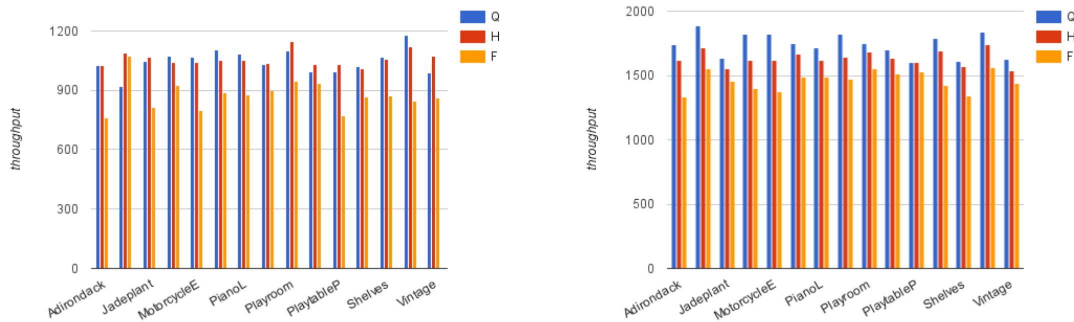


Figure 4-3: The throughput of Halide and OpenCV's implementations of stereoBM vs. number of cores

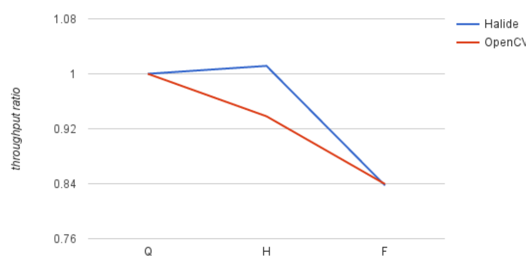
We have also studied the performances of both implementations for images of different resolutions from the Middlebury dataset. Quarter resolution images are up to 750×500, half resolution images are up to 1500×1000, and full resolution images are up to 3000×2000. To avoid the effects of parallelism, we have run all tests using only 1 core. We have also set disparity ranges of all tests to 64. The result is shown in Figure 4-4.

We can observe from the result that both implementations handle high resolution images well as the throughput does not drop by much for high resolution images. We also observe that Halide's implementations performs better for half-resolution images.



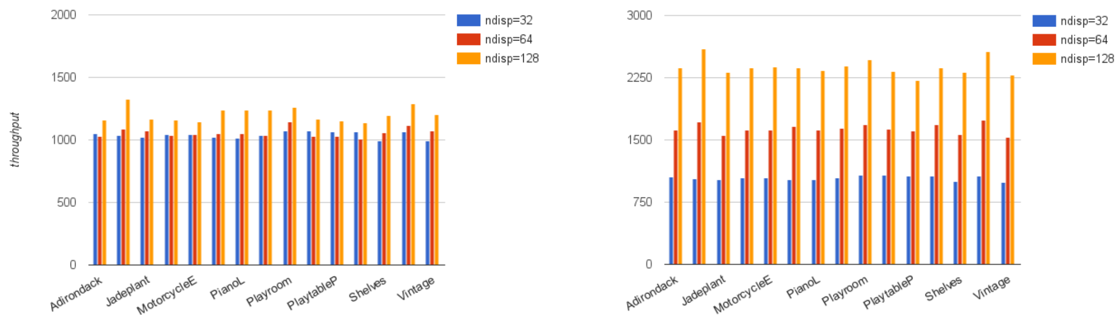
(a) The throughput of Halide for different resolutions

(b) The throughput of OpenCV for different resolutions



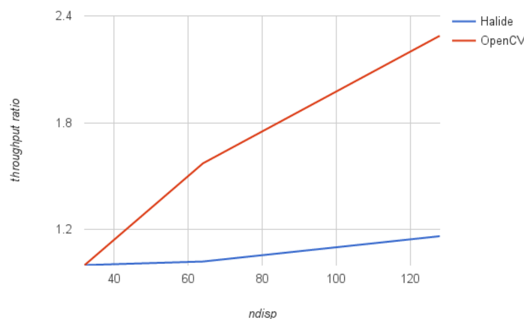
(c) The average throughput of Halide vs. OpenCV. The data is scaled so that the throughput of both for quater resolution is 1.

Figure 4-4: The throughput of Halide and OpenCV's implementations of stereoBM vs. resolution scales. Q, H, F stand for quarter resolution, half resolution and full resolution, respectively.



(a) The throughput of Halide with different disparity ranges

(b) The throughput of OpenCV with different disparity ranges



(c) The average throughput of Halide vs. OpenCV. The data is scaled so that the throughput of both for quater resolution is 1.

Figure 4-5: The throughput of Halide and OpenCV's implementations of stereoBM vs. disparity range

The performances of these two implementations for different disparity ranges are plotted in Figure 4-5. For both implementations, their throughput increases with larger disparity ranges, although OpenCV's implementation performs better when disparity ranges increase. This is because in OpenCV's implementation, dimension `d` is scheduled as the innermost dimension while it is the outermost in our implementation. Note that we have also proposed Schedule 4 in Section 3.2 which also schedules the `d` dimension to the innermost. An advantage of Halide is that we can always change schedules and tune parameters to find the optimal schedule according to the problem parameters and machine properties. If we run this algorithm with large dis-

parity size and on a machine with big cache sizes, we can expect that Schedule 4 will perform better and scale well with increasing disparity range.

Finally, we compare the length the Halide code and the OpenCV code for stereoBM. The Halide implementation for stereoBM is 83 lines long, literally translated from the pseudocode line by line while the OpenCV's one is 1278 lines long, involving low-level optimization instructions.

4.2 Evaluation of stereoGF Implementations

We cannot find any optimized CPU implementation of the guided filter algorithm. The original paper by Rhemann et al. [10] only released the MATLAB source code for their implementation. Therefore, we have implemented a naive schedule in Halide for the stereoGF algorithm that is equivalent to a naive C++ implementation without any optimization such as vectorization or parallelization. Every function is either computed at root or inline. We will use this naive implementation as a baseline to measure of the performance of our optimized implementation.

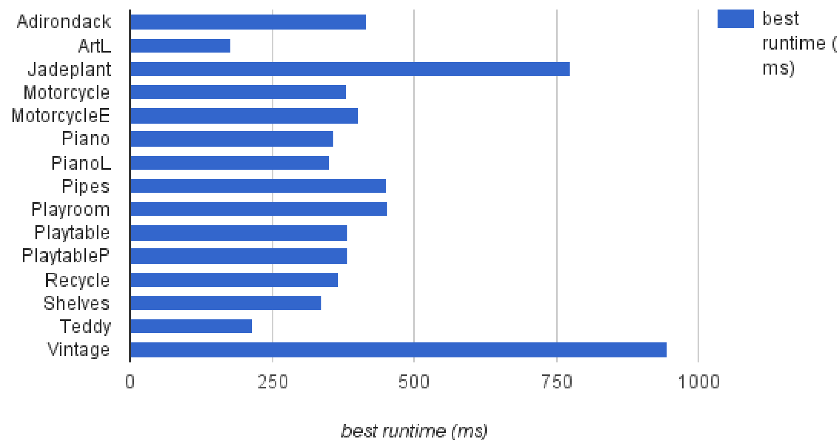


Figure 4-6: The best runtimes of optimized Halide implementation for quarter-resolution images

We run the optimized Halide implementation with different number of cores. Fig-

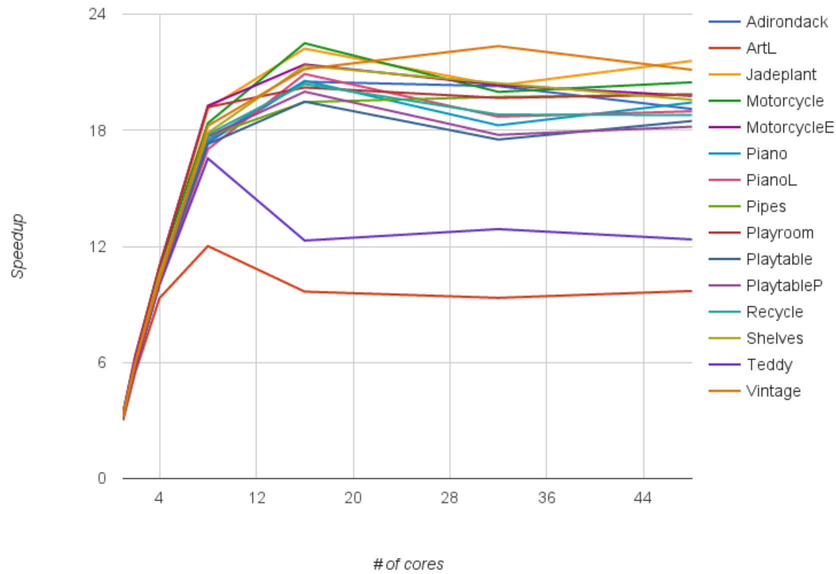


Figure 4-7: The speedup curves of stereoGF for quarter-resolution images vs. number of cores

ure 4-6 plots the best runtime for the quarter-resolution images among all of our runs. Figure 4-7 plots the speedup of the optimized Halide implementation compared to the baseline with 1, 2, 4, 16, 32 and 48 cores.

From the result, we observe that with parallelization, our optimized implementation achieves more than 10× speedup than the naive implementation for all test cases and 20× for most. The program scales up to 16 cores except for ArtL and Teddy, it scales to 8 cores. In fact, they are the two smallest images.

To further study the program's scalability for different size images, we test performance of the algorithm on images with three resolution scales: quarter-resolution (Q), half-resolution (H) and full-resolution (F). We set the number of disparities to 60 for all runs. The result is shown in Figure 4-8. As expected, the algorithm scales better for high-resolution images. Another observation is that when using 1 core, the algorithm maintains around the same throughput for all three resolution scales, indicating that the performance of the algorithm does not decay with image size.

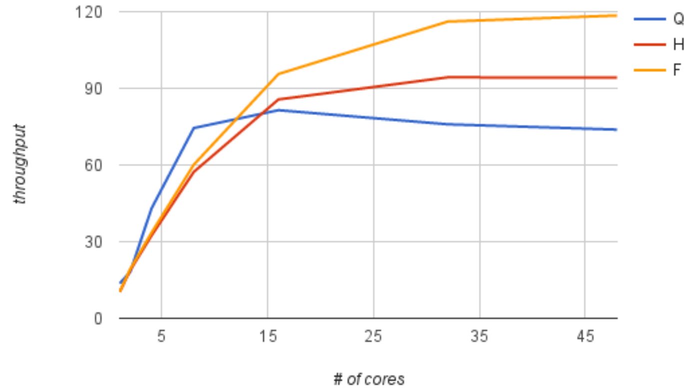


Figure 4-8: Performances of stereoGF for images with different resolutions vs. number of cores

Finally, we study the effect of disparity levels on the algorithm performance. We run our implementation for with disparity range equals to 20, 40, 60 and 80 on quarter-resolution images and plot the result in Figure 4-9. We can see that the algorithm throughput does not vary much with the disparity range, although high disparity ranges do bring a little performance benefit.

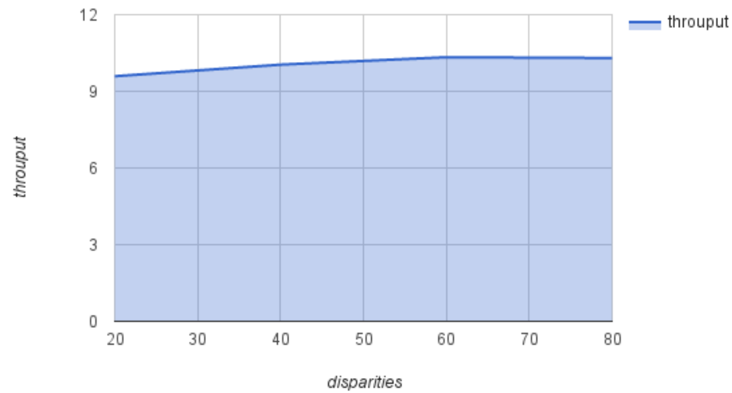


Figure 4-9: Performances of stereoGF vs. disparity range

Chapter 5

Design for Supporting Dynamic Programming in Halide

As we discussed in Section 3.2, we had problem scheduling the image integral definition of box filters in Halide because of its recursive nature. However, many stereo algorithms involve dynamic programming techniques [4, 16], whose recursive nature will also cause a problem in Halide. In this section, we will propose a design protocol in Halide to support representing and scheduling dynamic programming or recursive filters in Halide.

Currently, dynamic programming or recursive filters in halide can be expressed in Halide in the form of update definitions using reduction domain. The problem is to schedule them. Their recursive nature constrains the order of which pixels are computed. They have to be computed sequentially because all previous pixels have to evaluated before computing the next pixel. In this case, no matter which compute level we schedule the function at, Halide always computes from the first pixel; to avoid redundant recomputation, the only valid way is to schedule recursive functions at root, which limits both parallelism and memory locality.

For example, the classical dynamic programming problem of calculating the Fibonacci Series can be represented in Halide as follows.

```
f(x) = undef<int>();
f(0) = 0;
```

```
f(1) = 1;
f(rx) = f(rx-1) + f(rx-2);
g() = f(N);
```

The recursive definition of **f** is expressed in the third update stage of **f** with the use of reduction domain `rx`.

The problem is to schedule dynamic programming functions in Halide. An important technique in dynamic programming is memoization, which stores the value of solved subproblems for future use. For the Fibonacci example, by using memoization, we can store all values that have been computed to avoid recomputation. This corresponds to

```
f.compute_root();
```

where all values are computed only once. This algorithm takes $O(N)$ runtime and $O(N)$ storage space.

This algorithm can be further optimized by only storing two values `f(rx-1)` and `f(rx-2)` which improves memory locality by reusing the storage space. Ideally, this would correspond to Halide's schedule

```
f.store_root().compute_inline();
```

where Halide should detect and assign only the minimal storage space for `f`. However, in reality, the above schedule is identical to the first one.

The problem is with Halide's bounds inference procedure, during which Halide determines the allocation sizes and loop bounds for each function recursively back from the output. For non-recursive functions, Halide will apply an analysis called sliding window optimization which intelligently allocates minimal memory that is necessarily needed for the consumer, making no wastes. For example, for the simple box filter pipeline as we saw in Section 3.

```
RDom rk(-r, 2*r+1, "rk");
Func vertical_sum("vertical_sum"), box_sum("box_sum");
vertical_sum(x, y, d) = sum(in(x + rk, y, d));
box_sum(x, y, d)      = sum(vertical_sum(x, y + rk, d));
```

If we schedule this pipeline as follows, then instead of allocating memories for the entire image to **vsum**, Halide will notice that each pixel of **vsum** can be computed only once, and the maximum distance between a value being produced in **vsum** and consumed in **box_vsum** is only $2r+1$ and assign **vsum** memories for $2r+1$ scanlines. This process is called storage folding in Halide. However, if a function has a recursive update definition, Halide always determines the both the allocation sizes and loop bounds for this function as root without doing any further inference. Our goal is to modify the storage folding procedure to extend the above analysis to recursive functions as well.

```
box_sum.compute_root();
vsum.store_root().compute_at(box_sum, y);
```

In order to make storage folding work for recursive functions, we propose the following changes in the Halide compiler.

- Provide an interface in Halide that allows users to combine stages of a function. Currently, Halide generates a loop for each stage instead of each function during loop synthesis. However, if we want to compute a recursive function only once, all of its stages must be computed in a single loop in order to generate the final output at once. Note that to ensure that such change in the schedules does not affect the algorithm's outputs, Halide should check that i) each dimension occurs as reduction domain in at most one update stage; ii) in each update stage, the pixel being updated only depends on pixels that are computed before it. For the following example, it is illegal for both **f** and **g** to combine all of their stages into one as `f(x,y)` depends on `f(x+1, y)` in the update definition of **f** and dimension `x` appears as reduction domain `rx` in two update stages of **g**.

  ```
  f(x, y) = x;
  f(rx, y) = (f(rx+1, y) + f(rx-1, y))/2;
  g(x, y) = x;
  g(rx, y) =  g(rx-1,y);
  g(rx, y) =  (g(rx-1,y) + g(rx-2, y))/2;
  ```

- Modify the storage folding process to apply sliding window optimization to recursive functions whose stages have been combined. For non-recursive functions, the loop bound in one dimension of a function is the maximum distance in that dimension between a value being produced in this function and consumed in that consumer function. For recursive functions, since a value of this function may be consumed by the update stage of this function if the variable appears as a reduction domain, we also need to account for the distance between a value being produced and consumed in the update stage. For the following example, we need to allocate $\max(\texttt{s}, \texttt{t}) \times 1$ memory to function `f`.

```
f(x, y) = sin(x);
f(rx, y) = f(rx, y) - f(rx-t, y);
g(x, y) = f(x, y) + f(x+s, y);

g.compute_root();
f.store_root().compute_at(y);
```

The above proposal solves the issue of maximizing memory locality for recursive filters or dynamic programming functions. A more difficult task is to parallelize pipelines involving functions with recursive definitions. Because of their recursive nature, such functions can only be calculated sequentially. Therefore, it is not only a scheduling problem but also requires changes in the algorithm definition to parallelize such pipelines. A possible approach is to set division points and calculate the values on these points using non-recursive definitions. For example, when we tile the box filter, we compute the pixels at the tile boundary as the sum of pixels instead of using the recursive image integral definition.

Chapter 6

Conclusion

This paper implements two stereo matching algorithms, stereoBM and stereoGF, in Halide and evaluates their performances in terms of processing speed under different settings. The objective is to find out whether Halide can be applied to help researchers write simple but high performance stereo algorithms. We compare our implementations with highly optimized hand-written C implementations and naive C implementations. Our results show that Halide can achieve the same performance as the hand-tuned code and an order of magnitude performance improvement than the naive implementations. On the other hand, our code is much simpler and more understandable than the hand-optimized code, and can easily be customized to suit particular tasks or machines.

Nevertheless, we also find constraints of Halide in expressing recursive functions or dynamic programming algorithms. We claim that the issue can be solved by extending Halide's bound reference and storage folding processes to recursive functions and have proposed an execution plan for it.

Bibliography

[1] Halide tutorials. http://halide-lang.org/docs/examples.html. Accessed: 2016-06-04.

[2] Franklin C. Crow. Summed-area tables for texture mapping. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 207–212, New York, NY, USA, 1984. ACM.

[3] Kaiming He, Jian Sun, and Xiaoou Tang. Guided image filtering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(6):1397–1409, 2013.

[4] H. Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 807–814 vol. 2, June 2005.

[5] Charles Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision. Number MSR-TR-99-21, page 12. Institute of Electrical and Electronics Engineers, Inc., April 1999.

[6] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3061–3070, 2015.

[7] Sudeep Pillai, Srikumar Ramalingam, and John Leonard. High-performance and tunable stereo reconstruction. In *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016.

[8] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.*, 31(4):32:1–32:12, July 2012.

[9] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[10] C. Rhemann, A. Hosni, M. Bleyer, C. Rother, and M. Gelautz. Fast cost-volume filtering for visual correspondence and beyond. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 3017–3024, June 2011.

[11] Christian Richardt, Douglas Orr, Ian Davies, Antonio Criminisi, and Neil A. Dodgson. *Computer Vision – ECCV 2010: 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part III*, chapter Real-Time Spatiotemporal Stereo Matching Using the Dual-Cross-Bilateral Grid, pages 510–523. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[12] Daniel Scharstein, Heiko Hirschmüller, York Kitajima, Greg Krathwohl, Nera Nešić, Xi Wang, and Porter Westling. *Pattern Recognition: 36th German Conference, GCPR 2014, Münster, Germany, September 2-5, 2014, Proceedings*, chapter High-Resolution Stereo Datasets with Subpixel-Accurate Ground Truth, pages 31–42. Springer International Publishing, Cham, 2014.

[13] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1):7–42, 2002.

[14] Irwin Sobel. An isotropic 3 3 image gradient operator, 2015.

[15] Jian Sun, Nan-Ning Zheng, and Heung-Yeung Shum. Stereo matching using belief propagation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 25(7):787–800, July 2003.

[16] O. Veksler. Stereo correspondence by dynamic programming on a tree. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 384–390 vol. 2, June 2005.

[17] Kuk-Jin Yoon and In-So Kweon. Locally adaptive support-weight approach for visual correspondence search. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 924–931 vol. 2, June 2005.