Schedule Visualization and Analysis for Halide Image Processing Language

by

Jovana Knezevic

S.B, Massachusetts Institute of Technology (2012)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2013

Author ....
Department of Electrical Engineering and Computer Science
July 12, 2013

Certified by ....
Prof. Fredo Durand
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Prof. Dennis M. Freeman
Chairman, Masters of Engineering Thesis Committee

Schedule Visualization and Analysis for Halide Image Processing Language

by

Jovana Knezevic

Submitted to the Department of Electrical Engineering and Computer Science
on July 12, 2013, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Image processing applications require high performance software implementations in order to satisfy large input data and run on smaller mobile devices that require high efficiency. Halide is a language and compiler for optimizing image processing pipelines. Halide introduces a separation between algorithm, the logics behind the program, and a schedule, the order of execution. This thesis focuses on providing interactive GUI for visual analysis of Halide schedules. It creates a visualization of the order of execution and provides tools for analyzing three important aspects of image processing schedules: redundancy, locality and parallelism. Tool is designed for Halide programers who want to gain better understanding of scheduling in Halide and receive guidance for schedule optimizations.

Thesis Supervisor: Prof. Fredo Durand
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I would like to thank my thesis supervisor, Prof. Durand, and PhD student, Jonathan, for working with me on developing the ideas for this thesis and supporting me throughout my MEng journey. I would also like to thank Prof. Saman Amarasinghe and Sylvain Paris for being quick to come up with great ideas and advices.

I thank my lab mates Abe Davis, Nick Chornay, Michael Gharbi, Vladimir Bychkovsky, and Manohar Srikanth for bringing the fun back to work, karaoke, photography sessions, enjoyable lab meetings and even more enjoyable evenings.

I thank my friends and roommates, Ranko and Alex, for their immense support and the best two years I've had at MIT.

Finally, I thank my parents, who are the reason I am who I am. This thesis is every bit theirs as much as mine. Even though they are far away, they're celebrating this success with style (and a couple of good beers). I thank them because they are my support and my best friends.

Contents

1 Introduction

Image processing applications are becoming more popular every day. With growing number of smartphone users, almost everyone has access to a decent camera with up to 8-13 MP resolution. SLR and point and shoot cameras are becoming cheaper and more accessible, while camera sensors keep evolving and increasing in resolution and frame rate. Millions of users use applications like Instagram and Photoshop, but image processing application are also used in medical imaging, object and pattern recognition, etc. Image processing applications require high performance software implementations in order to satisfy large input data and run on the shrinking cameras and mobile devices that need high efficiency.

Nowadays, most of the image processing algorithms are implemented and then hand-tuned for a specific platform. In order to achieve very high performance, developers spend months writing vectorized, parallel, tiled and fused C code, trying to find an order of execution that might yield the best performance for a given platform. As a result, expert developers write algorithms that are efficient, but take months to write, are overly long and complex, and most importantly, lack flexibility and portability. Application code ends up being complex and highly platform specific. It becomes very hard to test alternative orders of execution and in order to port the application to a different platform, engineers write the application from scratch re-optimizing it for the new platform.

Halide changes the level of abstraction and decouples algorithm from its execution strategy, allowing for easier search for the optimal performance and better portability.

1.1 Separation of Algorithm and Schedule

Halide is a C++ functional embedded language developed by MIT graduate students Jonathan Ragan-Kelley and Andrew Adams. Halide represents images as functions defined over an infinite domain, where functions are mappings from coordinates to values, i.e. pixel coordinates to pixel's color. Halide tried to facilitate writing high performance image processing algorithms by decoupling algorithms from schedules. [1]

- *Algorithm* represents the logic of the program and the dependencies between different functions and function values. It specifies *what* gets executed and *how* (using which dependencies and which arithmetic operations)

- *Schedule* describes the order of execution of the algorithm. It commands when and where should value at each coordinate in each function be computed, where should those values be stored and how long they are cached before they are consumed[1]

Program's algorithm and schedule are implemented independently. The separation of the two allows the programer to explore different scheduling possibilities and seek an optimal solution without modifying the algorithm. Halide makes the code more readable, while facilitating the exploration of possible execution orders. Developers can reach high performance much quicker with cleaner, more readable and shorter code.

To illustrate this, PLDI paper on Halide uses an example implementation of local Laplacian filter. In order to implement this feature, one of the top developers in Adobe worked for three months and wrote hundreds of lines of C++ code with dozens of loops nests for high performance. Halide's version of local Laplacian took one intern day to write and ended up being 5 times shorter in lines of code, and 1.7 times faster.[1]

1.2 Thesis Goal and Guide

Halide is innovative because it introduces the separation between algorithms and schedules, while offering a way to describe image processing pipelines in a simple functional style. Halide enables a programmer to modify and try out many different schedules, all with the goal of finding the best one.

This thesis implements a tool for visualizing and analyzing Halide schedules. It should be used by a Halide developer in order to gain deeper understanding of order of execution, but also to get useful statistics on three main aspects of Halide's schedules: redundancy, locality and parallelism.

First we describe the schedule characteristics and the tension between locality, parallelism and redundant computation. The next section describes the general application setup and infrastructure. The sections that follow describe different modules of the GUI, the way data is processed in the background and the way it is visualized and presented to the user. Finally, results section demonstrates schedule analysis using the tools developed as a part of this thesis.

2 Schedules: Redundancy, Locality and Parallelism

This section describes Halide schedules in more details and demonstrates a tight relationship between locality, parallelism and the amount of redundant computations in every schedule. We will use an example of a simple two-stage blur algorithm, that computes a 3x3 box filter as two 3x1 passes. Input image is represented as a three-dimensional function from which we calculate the first stage of the algorithm, *blurx:*

```
blurx(x,y) = input(x-1,y) + input(x,y) + input(x+1,y);
```

blurx represents the input image blurred along the x axis using a 3x1 kernel. In the second stage, we compute the output of the two stage blur by blurring the y dimension of *blurx:*

```
blury(x,y) = blurx(x,y-1) + blurx(x,y) + blurx(x,y+1);
```

These two lines of code are what we call the *Algorithm* portion of the Halide code. Figure 1 sketches the dependencies between three functions where input function was 16x16 pixels.

There are many different ways in which this simple pipeline can be scheduled for execution on the processor. All available schedules cover a three-dimensional space of possible schedules along the three axis: redundant computation, locality and parallelism axis.

We want to minimize the total number of computations. Ideally, we would like to calculate each value exactly once. It turns out that sometimes, introducing redundant computation i.e. calculating the same values multiple times, can decouple dependencies and enable them to execute in parallel. On the other hand, we want to improve producer-consumer locality. Values that are computed should be used to calculate their descendants as soon as possible, increasing the chances of them still being in cache when reused. Great locality minimizes the number of cache misses and decreases the number of accesses to the slow, disk memory. Simultaneously, we want to have a lot of parallelism available i.e. we want independent operations on independent regions being executed on multiple threads. We would like to maximize all three, but have to face some of the tradeoffs. In order to gain on redundancy and parallelism we might need to sacrifice some of the locality, and vice versa.

The following subsections describe three different schedules that favor redundant computation, locality and parallelism respectively and demonstrate the trade-off relationship between these three variables. All three schedules are applied to the same two-stage blur described above.

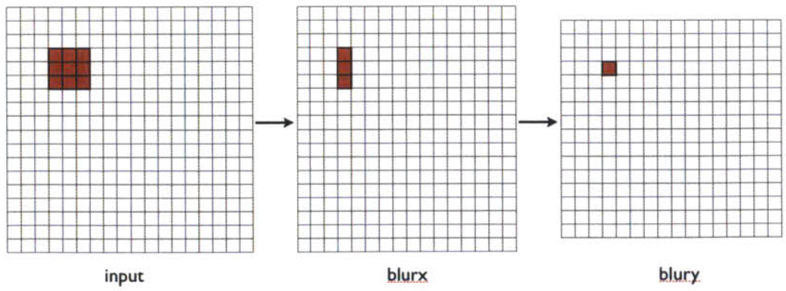

Figure 1: Visual representation of a two-stage blur dependencies

2.1 Strategy 1: Redundancy

The first simple approach a programer can take is to calculate all the values of *blurx* first, before proceeding to calculate *blury*. This is a breadth-first execution approach.

```
blur_x.chunk(root, root);
```

Every value of *blurx* is calculated independently using the corresponding pixels from the input, thus each of those calculations can be executed in parallel. Each *blurx* and *blury* value is calculated and stored only once, therefore achieving the optimal number of computations. This schedule performs well in terms of parallelism and redundancy, it maximizes parallelism and minimizes redundant computation, but fails short when it comes to producer-consumer locality. *blurx* pixels will be evaluated and stored, but the schedule will compute and store the whole *blurx* before reusing *blurx* values to calculate pixels in *blury*. It is likely that by the time *blurx* pixels get reused, they got out of cache forcing the program to access the disk memory. In other words, it takes a long time (or more specifically a large number of operations) between the times when the values are computed and used, thus damaging the locality. In this extreme case, locality was sacrificed to get optimal redundant computation and improve parallelism.

2.2 Strategy 2: Locality

The second approach tries to achieve the optimal locality. In this strategy, *blury* computes each point in *blurx*, immediately before the point which uses it.

```
blur_x.chunk(x, x);
```

That would mean that before computing little red cell in *blury*, three red cells from *blurx* will be computed from 9 red cells in input. (Figure 1) This stencil is repeated for every pixel in *blury* independently. The strategy is knows as depth-first execution approach and it insures maximized locality. Pixels in *blurx* are calculated as needed and used immediately, thus producing the optimal locality. Since computations across *blury* are done independently for every pixel, this schedule has a lot of avaiable parallelism. However, because *blury* pixels have overlapping dependencies in *blurx* and these are calculated independently, many of the *blurx* values will be computed multiple times, once for each *blury* pixel that uses it. This introduces redundant work across *blurx*. Therefore, this depth-first strategy achieves high locality and parallelism, while introducing a lot of redundant work.

2.3 Strategy 3: Parallelism

The third strategy is similar to the previous one; it calculates *blurx* values as they are needed, but it stores them instead of calculating them independently for each *blury* pixel. In this fashion, *blurx* values will be calculated using the sliding window approach.

```
blur_x.chunk(root, x);
```

Thirs strategy calculates each *blurx* value exactly once and reuses it in the following computation. This schedule still achieves high locality. The distance between a *blurx* value being produced and consumed is up to three lines of *blurx*. This is much better that the whole image (Strategy 1) but it's not as good as depth-first approach (Strategy 2). However, in order to minimize the redundant work from Strategy 2, independence between calculations is introduced. *blury* values have to wait for the previous ones to be computed so that the values can be reused, which means that they can't be executed in parallel any more. Therefore, the third strategy minimizes the number of redundant computations, improves on locality compared to breath-first approach, but it completely destroys parallelism.

2.4 The tradeoff

Two-stage blur example with three different schedule possibilities shows several features of Halide language. First, it demonstrates the independence between algorithm and order of execution. Program can specify and freely change and try out any schedules without modifying the initial blur algorithm. Second, Halide's functional nature provides clean and clear representation of images as functions and generates highly readable algorithms. Third, Halide's keywords and schedule abstraction allow us to describe a huge space of different execution possibilities while getting rid of a very messy syntax. Halide schedules look simple and are much easier to understand. Finally, all the previous features give programmer enough freedom to experiment and try out many different schedules before finding an optimal one.

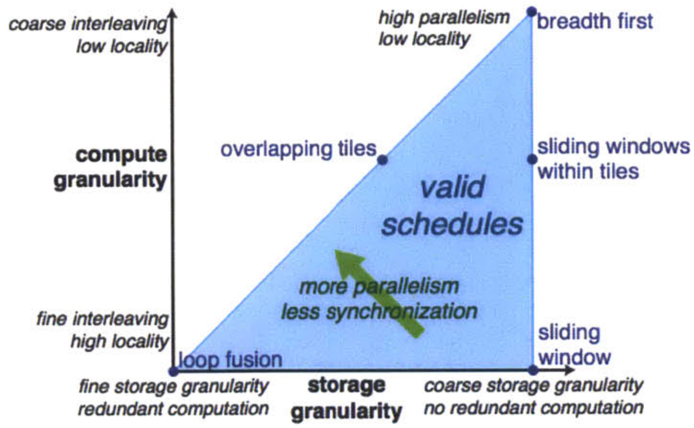

Figure 2: Breadth-first execution (Strategy 1) has poor locality, depth-first execution (Strategy 2) often does redundant work and using sliding windows (Strategy 3) to avoid redundant recompilation constrains parallelism by introducing dependencies across loop iterations. [1]

Every schedule represents a fine tension between redundancy, locality and parallelism. Figure 2 extracted from the PLDI paper graphically describes this tension. Three examples above demonstrate extreme cases when one is highly optimized while the other two variables suffer. Most of the time, the best schedule won't be in these extremes; it will be a combination of contributions of all three attributes, with different contribution across individual functions and across fusion between those stages.

The work described in the following chapters help the programer understand the way redundancy, locality and parallelism interact in a given schedule. Even though we can often sit down and reason out the tradeoffs between these three variables, it takes time and effort to truly deduce what is happening for each schedule we might play with. One of the goals of this thesis is to calculate and display statistics that might help developers gain better understanding of how these three parameters interact within a schedule. As a consequence, this analysis tool should give guidance into how to improve upon existing schedules.

3 Schedule Visualizer

Schedule visualizer processes and extracts information about execution of Halide's programs suppling the developer with visualizations of schedules and statistical data. This section will talk about its features, GUI design, and data processing in the background. We start off by describing the input files in subsection 3.1 followed by general description of frameworks and coding infrastructure in subsection 3.2. Schedule visualizer is made out of several different visualizers, each of which is described in a separate subsection, 3.3, 3.4, 3.5 and 3.6.

3.1 Execution traces

Halide programs can generate trace files that describe program's execution. By specifying setenv("HL_TRACE_ENABLED=2") program will generate an output file. This slows down the execution due to all the print statements, but it gives insight into what happened during the execution. Traces contain information about 4 different kinds of operations:

- *Allocation* describes the action of allocating memory for values to be computed. It is described in trace file as:

  ```
  Allocating function_name over x_min y_min x_max y_max
  ```

- Freeing is the opposite of allocation. It describes when some function range is released from memory:

  ```
  Freeing function_name over x_min y_min x_max y_max
  ```

- *Load* signifies when a value or values are loaded into memory. This means they were previously stored, either as input values, or were computed, stored and are being loaded to be used in the next computation. Loads are described as:

  ```
  Loading function_name at x y
  ```

  or as:

  ```
  Loading function_name at [x1, x2, x3, x4] y
  ```

  if multiple values are being loaded simultaneously, in case of vectorization.

- *Store* labels when function value is stored in memory:

  ```
  Storing function_name at x y
  ```

  or as:

  ```
  Storing function_name at [x1, x2, x3, x4] y
  ```

  if multiple values are being stored simultaneously, in case of vectorization.

Depending on the size of the images/functions and the complexity of algorithm and schedule, trace files can be a few MBs in size up to a couple of GBs. Performance and responsiveness of the GUI become an issue when trace files are too large. We managed to deal with trace files that are up to 1GB large and future work should include performance improvement.

Schedule visualizes doesn't interact with Halide code directly. It only takes the data provided by the execution trace and analyzes it in various ways. Trace files often change with different version of the compiler. That's because Halide language is still in the development phase and a lot of things will be improved upon and changed until it reaches its final version. Before trace files are supplied to the visualizer at input, they are preprocessed in two different ways.

First, simple python script parses the trace, generating two different files that we call header file and body file. Header file of the two-stage blur example is depicted in Figure 3. Header file describes the functions that participate in the algorithm, including their names, the number of dimensions and the range of values that are going to be used or calculated in each dimension.

```
3
0 blur_y_0.blur_x_0 2 0 15 -1 16 0 0 0 0
1 .m0 2 -1 16 -1 16 0 0 0 0
2 blur_y_0 2 0 15 0 15 0 0 0 0
```

Figure 3: Header file of one of the two stage blur execution traces

Second file is the body file. It contains information about all the data manipulation, formatting the four operations into lines of integers of constant size. Each operation, allocation, freeing, storing and loading, is converted into a sequence of 10 integers per file line. Each operation is represented with a number where 0 corresponds to allocation, 1 to loading, 2 to storing and 3 to freeing. This further enables the conversion of body file to binary body file, which enables a very quick load of all the operation into our visualizer C++ code. Each operation is represented as a struct depicted in Figure 4. Operations are read very efficiently into memory using *fread* C++ stream function.

It seems like a lot of effort, but it's well worth it. Initially, the file parsing took a long time because of the sheer size of the trace files. With trace preprocessing, loading of the trace files takes significantly less time. Files need to be preprocessed only once and then freely reused at each run of the visualizer.

3.2 Infrastructure

Schedule visualizer is implemented in C++ using Qt for graphical user interface. Qt is a cross-platform application and UI framework for developers using C++ or QML, a CSS & JavaScript like language. [2]Project repository is at *https://github.com/mit-gfx/halide-visualizer* and is available for all Halide developers who would like to use it. In order to run the visualizer, users need to have Qt installed along with Qwt library. Schedule Visualizer displays a lot of statistical data and it uses Qwt library to do so. The Qwt library contains GUI Components and utility classes which are primarily useful for programs with a technical background and include GUI components for histograms and plots, among other things.[3]

3.3 Execution Visualizer

First idea of this project is to help developers visualize the execution of their Halide program. Different schedules will cause different order of execution of the same algorithm, produce different trace files and lead to different visualizations. Order of execution will change within functions, but also across stages. We are interested in seeing when pixels are stored (produced) and when they are loaded (consumed). Since Halide programs mostly deal with images, the most natural way of visualizing this is representing each function as a grid of pixels. Figure 4 shows the representation of 2D functions of two-stage blur inside of the execution visualizer.

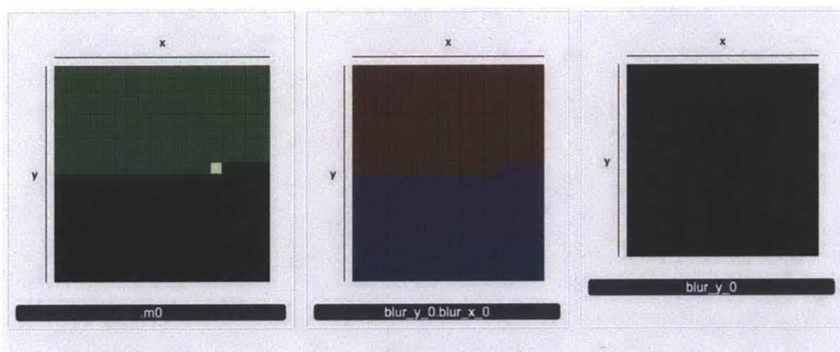

Figure 4: Halide functions are visually represented as a grid of blocks/pixels

Each function is represented as a QPixmap inside of code and each cell correlates to the coordinates of the functions inside of the program. Functions will be of various sizes, depending on the algorithm and the size of input images. That is why user has an option to zoom in or zoom out using plus and minus keyboard keys. By zooming in, each function cell becomes larger, and vice versa, by zooming out they become smaller until they reach 1x1 pixels minimum available size. This enables better control over the layout of the visualization and how much is seen at a time. In some cases, it might be beneficial to zoom out so that more functions are visible in the GUI simultaneously. However, as cells get smaller, it is harder to see what is going on per individual function region. User might decide to zoom in and observe a specific region in enlarged view and more details.

Figure 5 shows a complete appearance of the execution visualizer. Trace files contain all the operations written in the order of the program's execution.

The list of operations in the order of execution is called a *timeline*. Slider is positioned below functions and slider's value represent a position inside of a timeline. When slider is dragged all the way to the end, we see the results of the complete program execution. Using the slider, user can go forward and back in time, following and observing order of execution.

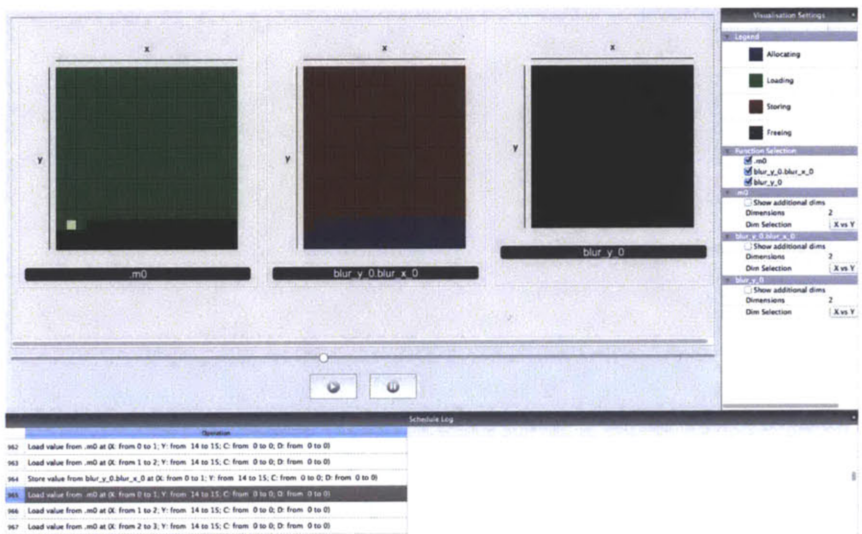

Figure 5: GUI of the execution visualizer showing all the components

Additionally, user can go through the timeline step by step in two different ways. First, keyboard keys A and S move the visualization in time by exactly one time step i. e. one operation. User can move the slider to the point of interest and then trace through the execution step by step, forward or backwards in time. Second, user can click on "play" and "pause" buttons located below the slider. Clicking on play causes visualization to play through the execution automatically, while the user can sit back and observe. Pause button pauses the animation. During the automatic execution visualization, operation is executed every 100 ms.

Execution visualizer is color coded. Function cells are colored differently depending on the operation being executed. The legend at the top right corner shows what each color represents. Pixels are colored red when stored, green when loaded, blue when allocated and dark gray when freed. Functions are initially black. That means no operation has been executed in those regions. Blocks in lighter colors represent the operation happening at the current slider position. Once the operation has finished, the cells assume the darker color of their last operation. In Figure 5, highlighted pixel of .m0 is the one being loaded at the moment. We can see that majority of input function.m0 has been loaded and used for calculating and storing pixels of *blur_y_0.blur_x_0*. *blur_y_0* hasn't been touched at this point, because the example schedule shows the Strategy 1 described in Section 2. First, the whole intermediate function is calculated and stored before proceeding to using those pixels to calculate *blur_ y_ 0* values.

Around the main visualization segment, Schedule Log and Visualization Settings widgets are placed. They can be closed or moved around, if not needed, to make more screen space for the visualization itself. Schedule Log contains a readable version of the trace file. It shows all the operations in order while table selection moves concurrently with the slider, with current operations always being the highlighted one. User is also able to scroll up and down the log, select any operation and slider will immediately follow up, placing the timeline at that particular selected operation. Schedule Log familiarizes the developer with trace files and the correlation between trace files and the actual execution.

Functions are ordered in the order of their dependence. In our example above, input image is first followed by *blur_ x* that's derived from *.m0*, followed by the output function. Names of the functions are extracted from the trace and they depend purely on how Halide decided to name them.

Visualization Settings widget offers users a few additional options. By default, all the functions are displayed as part of the visualizer space. Sometimes, due to their size, they can't all fit into the screen space. By placing QPixmaps on top of a QScrollWidget, we enabled user to scroll left to right, up and down and be able to observe all the functions. Additionally, user can click on the checkboxes in function selection to select which function should be displayed. This is especially useful when the algorithm deals with a lot of functions; user might want to observe only two whose pixels are being loaded and stored at the time.

Visualization Settings also display some basic info about each function. Halide functions can have up to 4 dimensions. Commonly, they will have three, representing an image and its x and y coordinates and three color channels, but they can have anywhere from 1-4 dimensions. User has an option to select which two dimensions should be plotted on a function's 2D cell grid. Figure 6 shows all possible selection for a 4D function. User can choose any two dimensions to use them as function coordinates and observe operations in the space of those 2 dimensions.

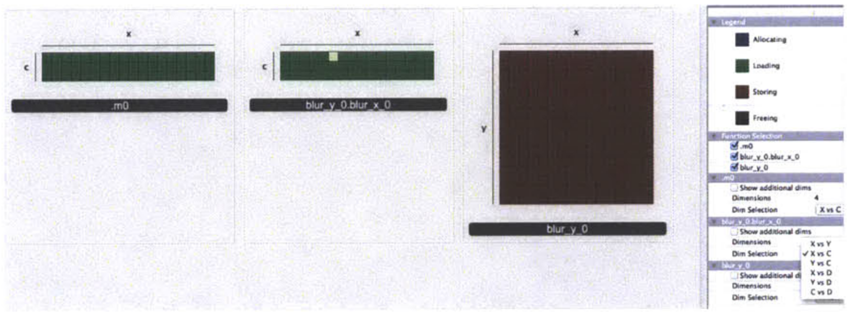

Figure 6: User can choose to observe operations in the space of any 2 function dimensions

Often, one might want to know what is happening in every dimension. 2D grid looks good but it is not enough to represent all the cells of a 4D function. Instead, it's possible to use two 2D grids, where each grid is a combination of two of the four dimensions. This feature is enabled by selecting *Show additional dims* checkbox. Figure 7 demonstrates this example. User can select cells in x and y grid and see the state of the other two dimensions at given point in time. Using this one, and the previous feature, it is possible to make any possible graphical combination of the four dimensions using 2D grids of colored cells.

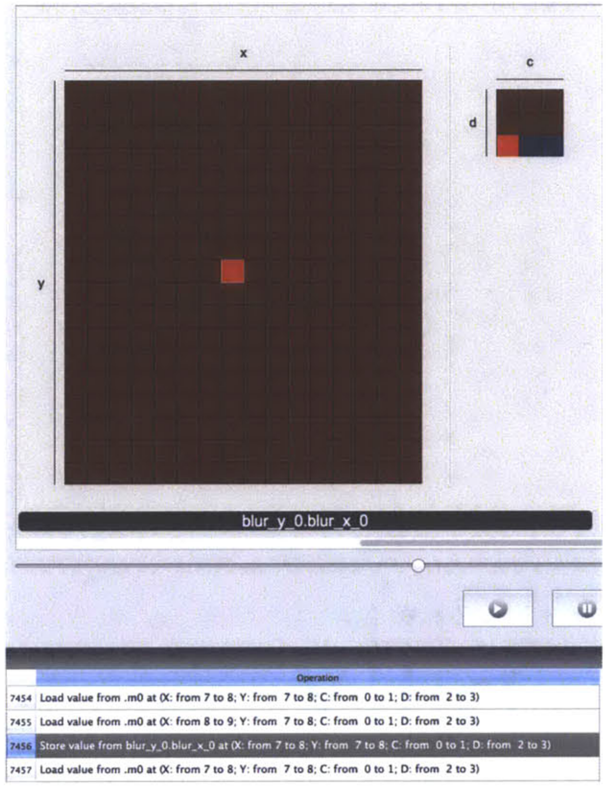

Figure 7: Figure shows *blur_ x* visualized as a two 2D grid. First one shows the operation in the x and y dimensions, and the second one c and d. Selecting a cell *(x1,y1)* inside of the first grid, displays the state of *(x1, y1, c, d)* grid for c and d dimensions

Schedules can specify vectorization of a dimension. This means that vectors of cells will be processed, loaded or stored, simultaneously. In that case, visualization will successfully highlight multiple pixels as demonstrated in Figure 8. Figure 8 also shows an example of tiling. *blur_ y* is separated into four different sections (tiles), each of which is processed in vectors along x axis.

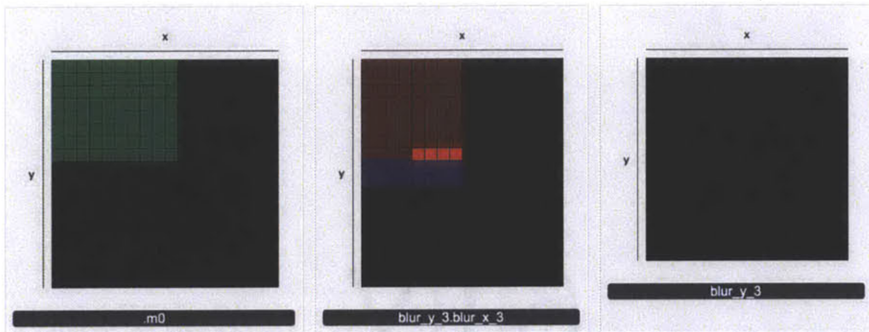

Figure 8: Vectorization and tiling

Execution visualizer has multiple benefits. It helps users understand better what is going on during the execution. Instead of manually going through thousands of lines of trace files, which is how code is often debugged, visualization processes all that data and graphically visualizes execution. It is much easier to observe and understand in which way programs are executed and to understand constructs like vectorization and tiling. Execution visualizer also helps debug the schedules. If a programer expected tiling but doesn't observe it where desired, he can trace back to that place in the schedule log and see what happened instead and why. Execution visualizer will also help users who are new to the world of Halide and Halide's schedules. By visualizing selected schedules they gain deeper understanding into how Halide schedules work.

3.4 Dependency Visualizer

Dependency visualizer gives insight into dependencies of the algorithm itself. It's representation doesn't change with the schedule, but with the algorithm. For that reason, it is considered to be less of a schedule analyzer and more of an algorithm analyzer. Dependency visualizer is an attempt to derive as much of useful information from the trace as possible. It visualizes the dependencies between values of functions and its UI layout is derived from the execution visualizer.

Value dependencies are deduced from the trace. Traces are ordered in the order of execution. Therefore, before a value is calculated and stored, all its dependencies have to be loaded into memory in order to be used. All the values that are loaded immediately before some other value is stored, are considered "parents" of the stored value. Vice versa, the stored value is considered a "child" of all the loaded values. The same pixel can be a child or a parent of many others. The following is the excerpt from a trace file.

```
Loading .m0 at -1 -1
Loading .m0 at 0 -1
Loading .m0 at 1 -1
Storing blur_y_0.blur_x_0 at 0 -1
Loading .m0 at 0 -1
Loading .m0 at 1 -1
Loading .m0 at 2 -1
Storing blur_y_0.blur_x_0 at 1 -1
```

It would be deduced that pixel of *blur_y_0.blur_x_0* at *(0, -1)* depends on pixels of function *.m0* at *(-1, -1)*, *(0, -1)* and *(1, -1)*. Similarly, pixel of *blur_y_0.blur_x_0* at *(1, -1)* is a child of pixels of function *.m0* at *(0, -1)*, *(1, -1)* and *(2, -1)*. All of the dependency information is stored in memory and accessed during the visualization.

Dependency visualizer, in a similar way as execution visualizer, represents all the functions as QPixmaps, 2D grids of cells. Function with more than two dimensions are represented with two different QPixmaps, one showing X and Y dimensions, and the other showing C and possibly D. In the beginning all the cells are gray and the user can choose whether to make functions visible by selecting or deselecting checkboxes in the right side dock widget. By default, all the functions are displayed.

User then has the option of clicking on cells whose dependencies they are interested in analyzing. With the right click of a mouse, the cell becomes highlighted blue, with all its children drawn in red and all its parents drawn in green. Figure 9 shows a dependency visualization for two-stage blur on two-dimensional functions.

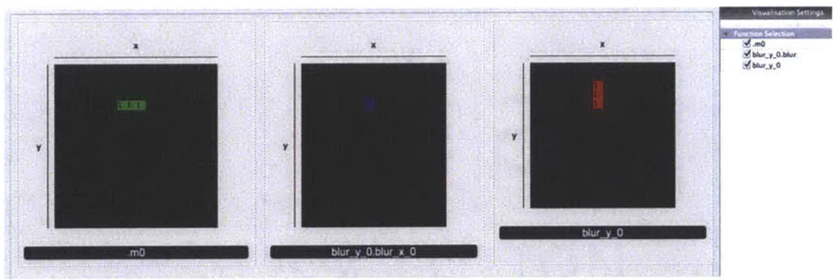

Figure 9: Dependency visualizer showing the dependencies on a two-stage blur. User selected the blue cell; green are it's dependency ancestors; red are it's dependency descendants. Number 1 represents direct dependence.

Blue square was selected by user by right clicking on a cell. Green cells show blue cell's parents, and red cells show its children. The number on each cell represents a generation distance. In this case, since all the cells have number 1 inscribed, they are directly related to the computation of the blue square. Dependency visualizer can also show descendants in further generation i.e. children of children, etc. Figure 10 demonstrates this ability. Children labeled with number two are "grandchildren" of the blue cell.

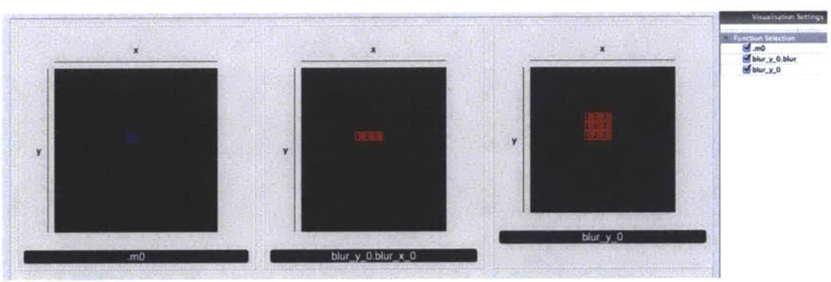

Figure 10: Dependency visualizer showing the dependencies on a two-stage blur. Cells labeled with number 1 are direct children of the blue cell. Cells labels with number 2 are children of the cells with label 1.

These dependencies are consistent with two-stage blur algorithm. *.m0* is an input image, and all the pixels in *blur_y_0.blur_x_0* are derived from taking three cells in *.m0* along the x axis and averaging their value. This means that most of *.m0* cell (excluding border cases) participate in calculations of three *blur_y_0.blur_x_0* cells, as demonstrated by the dependency visualizer. Further more, each of the *blur_y_0.blur_x_0* contributes to three different cells in *blur_y_0* along the y axis. As a result, we see a 3x3 square in *blur_y_0*. Blue square value was used in calculations of 9 different cells around that region in *blur_y_0*. That is exactly what two-stage blur does: it applies 3x3 kernel to average neighboring pixels and produces blur. Dependency diagram helps to establish correctness of an algorithm.

Dependency visualizer can cope with functions that have up to 4 dimensions, supporting all Halide functions. Figure 11 represents the layout in case of the same algorithm, two-stage blur, but with 3D functions. QPixmaps on the left of each function segment show X and Y dimensions, and the right one shows C dimension. In the case of more than two dimensions, user needs to specify (right click select) both values in X and Y, and C domain. After that, all the ancestors and descendants will be labeled in the X, Y grids. To examine the dependencies of C and D dimensions, user needs to click on individual children/parents and observe C, D dimension for the selected (X, Y) function of that cell. For those selections, user uses left click and the white border appears around the selected cell. In the case of Figure 11, user selected the bottom right corner of the 3x3 green square and verified that first cell in C dimension of function *blur_y_0* is a descendant of first cell in C dimension of the input *.m0*. Similarly, user chose the bottom green cell of the X, Y grid of function *blur_y_0,blur_x_0* to confirm the same relationship in C dimension.

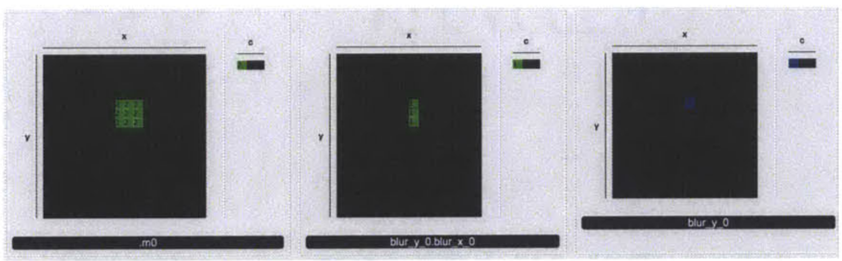

Figure 11: Dependency visualizer showing the dependencies for two-stage blur for 3D functions

Results obtained for two-stage blur are consistent with the expected results pictured in Figure 1. In summary, dependency visualizer can display all Halide functions and show the dependencies between individual values. This tool is primarily used to verify the logic behind the algorithm.

3.5 Statistics Tool

Statistics tool derives the statistical data related to the three schedule properties mentioned, locality, redundancy and parallelism. Statistics are presented in an interactive and visual way, that helps developers analyze schedule properties, identify week points and give intuition for improvements. Statistics tool is divided into three independent sections that represent the data for these three variables, redundant computation, locality and parallelism.

3.5.1 Redundant Computation

As explained in section 2, optimal number of computations per each function value is one. Values could be loaded and used multiple times, but ideally we want to calculate values exactly once, compute them and store them. Redundant computation is introduced in some schedules to improve locality or increase available parallelism. Statistics tool offers a way to observe the amount of computation done per function and in total, and diagnose the areas where redundant computation is performed.

Trace files are preprocessed in order to extract information about computations. Each operation is observed and different information is stored per each pixel: the number of loads, the number of stores, the times of each operation. Then, per pixel information is processed to achieve statistics on the function level: total number of loads and stores, average number of load and stores, etc. This information is presented as a histogram of the number of pixels per number of stores. User can select to display a histogram for a specific function, or to show a total histogram combining the values of all the functions.

Figure 12. shows an example of stores visualization. It is an instance of two-stage blur as before, with a schedule whose values are calculated once, two times or three times. The number of pixels that were stored more than once represents the amount of redundant computation of this schedule.

Figure 12: Histogram representing the number of calculations per pixel in a two-stage blur schedule

Ideally, one will want to minimize the amount of redundant computation. Such histogram would show only a peak at number 1 on x axis. Schedule in Figure 12, on the contrary, introduces a lot of redundant computation since most of the pixel values are calculated three times.

Once the user has observed the histogram values, they can closely examine where the redundant computations are coming from. Below the histogram there is a QPixmap representation of the function, like before. User can click on particular histogram values and the pixels that were stored that many times will be highlighted in red. Figure 13 shows an example of this interaction with the previous histogram of *blur_y_1.blur_x_1*. As Figure 13 shows, after clicking on the histogram bar with x value 3, all the pixels associated with that number of stores are colored red. Additionally, user can click on pixels themselves to reveal more detail. Selected pixel's border is colored in white and its chain of operations is reproduced below emphasizing where the histogram values are coming from. In the case of the selected pixel in Figure 13, after observing red borders around store operations in its chain of operations, it is obvious that this pixel's value was calculated and stored three separate times.

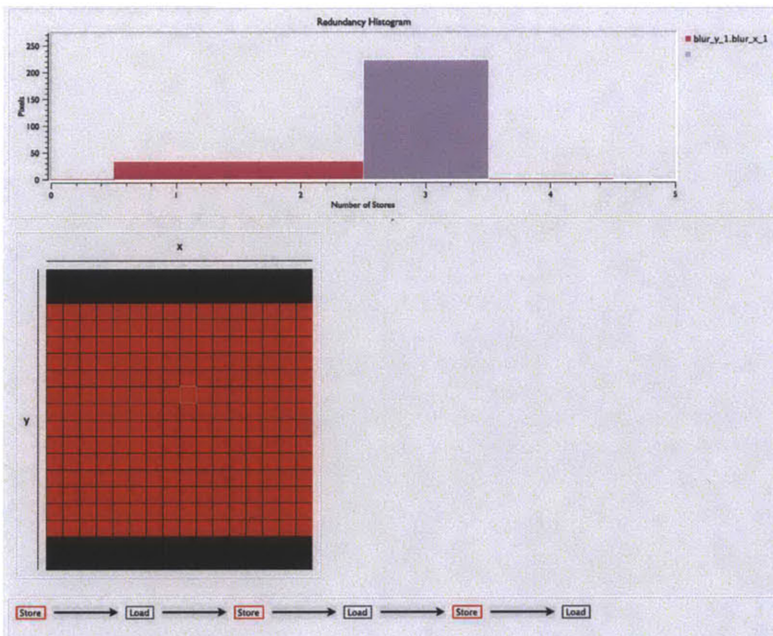

Figure 13: Interaction with the stores histogram

Another example of redundant computation analysis is shown in Figure 14. This figure also represents the same algorithm, but a different schedule. This schedule tiles the *blur_y.blur_x* function in 4 different regions, called tiles. Each tile is computed separately, so all the redundant computation happens where the tiles overlap: those pixels' values were computed once in each tile.

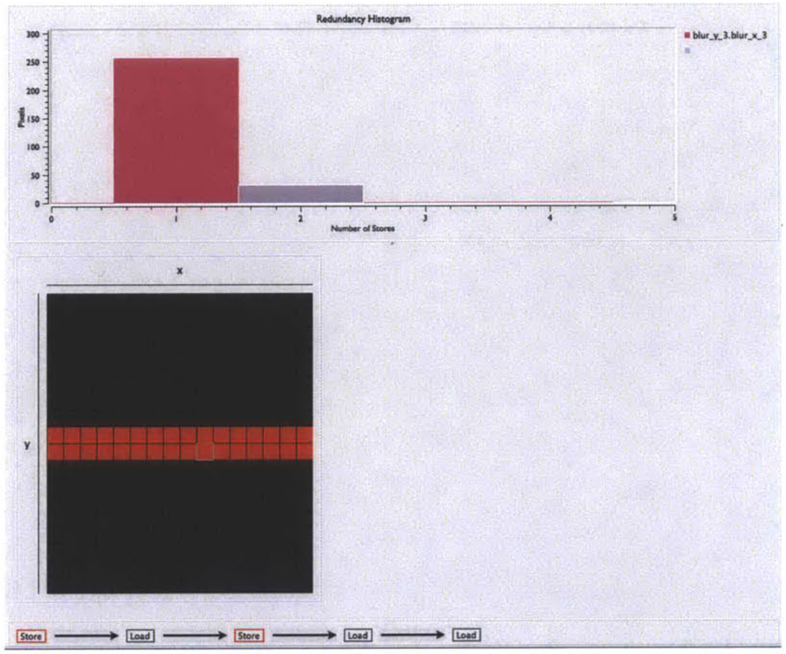

Figure 14: Interaction with the stores histogram of a tiling schedule

Users can also choose to visualize a total redundancy histogram (Figure 15). This is a cumulative histogram of number of stores of each function, combined together in different colors. This histogram is not interactive, but it helps observe the total redundancy of a schedule and the way functions relate to each other. One function when put against the others can be a major contributor to all the redundant work, and therefore we might want to analyze that function closer. If the user wants to analyze any aspect in more details, they can easily switch to per function mode and proceed as described previously.

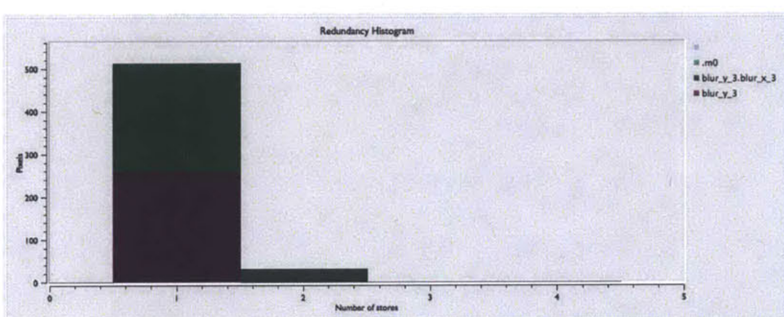

Figure 15: Total redundancy histogram

Redundant work happens only when a schedule does multiple stores per pixel. However, it is possible to choose to observe the load distribution of each function. GUI works in a similar way as for stores, but this time Load operations are highlighted instead of Store operations. Figure 16 shows a visualization for *blur_y.blur_x* loads of the same schedule as Figure 14 and Figure 15. In a similar way, total loads histogram is available as well.

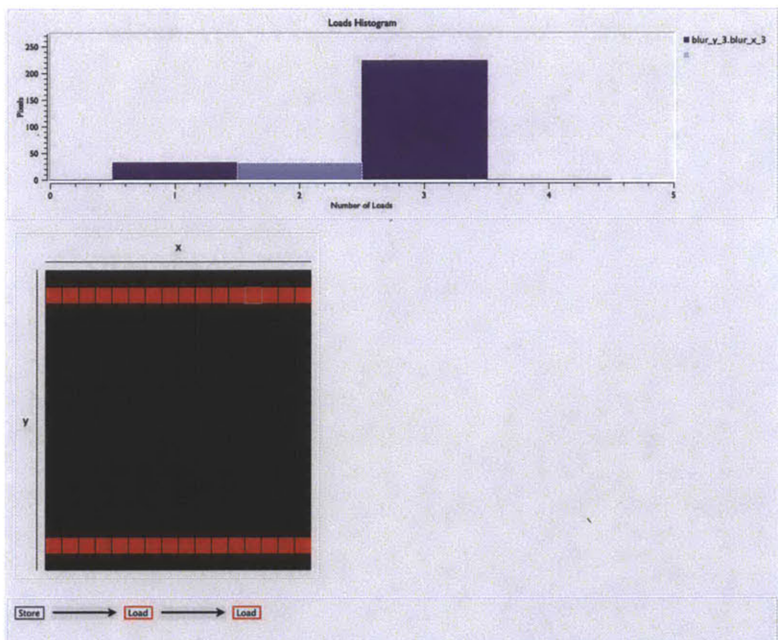

Figure 16: Histogram interaction and visualization of load operations

This visualization and analysis tool helps understand the distribution of store and load operations of a given schedule. It provides information about the amount of loads and stores per schedule, per function and per pixel. It calculates and demonstrates the amount of redundant work being done, but most importantly, it helps to diagnose critical areas where the redundant work is coming from. This leads to developers better understanding one of the most important schedule features, redundant work, and knowing where and what to change in order to improve it.

3.5.2 Locality

Good locality is achieved with keeping the number of cache misses as low as possible. Cache is a fast memory and less cache misses means less disk accesses and faster load times. In order to achieve this, values should be used soon after they are computed or previously used, increasing the chance that they are still located inside of the cache. Locality analyzer extracts, from the trace, information about producer-consumer locality and represents it in a visual and interactive way.

Locality analyzer calculates the distances between pixel accesses. In other words it finds, between two accesses to the same pixel, how many different pixels were accessed. When pixel A is stored or loaded, the copy of its value stays in cache. Every time a different pixel is loaded or stored, they are added to the cache as well. In our calculation, we simulate behavior of a simple LRU (least recently used) cache. If many different pixels are loaded/stored (more than the size of the cache), pixel A will removed and replaced by newly accessed pixels. The next time A is loaded, cache miss occurs. If however, the number of newly inserted pixels is not larger than cache size, A will be in cache and provide faster load time. This number of different pixels loaded or stored between two accesses to the same pixel is what we call the distance between accesses. For optimal locality, the distance should be as small as possible.

The distance between accesses is calculated for every pixel separately, and for every two accesses to that pixel. Trace is processes by observing each pixel individually and saving the time of every operation on that pixel. Every time a load operation happens, algorithm counts how many other, different pixels were accessed, since the last time a load or a store operation was performed on this pixel. This is repeated for every pixel and every pair of store/load, load-/load operations. Eventually, each pixel will have a locality vector of distances associated with it. Out of that data, locality map is produced that maps each distance to the number of pixels that have that distance in their locality vector.

Locality map data is presented in a histogram of distances vs. number of pixels. User is able to interact with a histogram, not only by selecting individual data points, but also by selecting a range along x axis. Range selection can be performed by clicking, dragging and releasing the mouse. The click represents a range starting point, and release marks the ending point. Once the range is selected, pixels whose distances belong to that range are highlighted with red on a QPixmap.

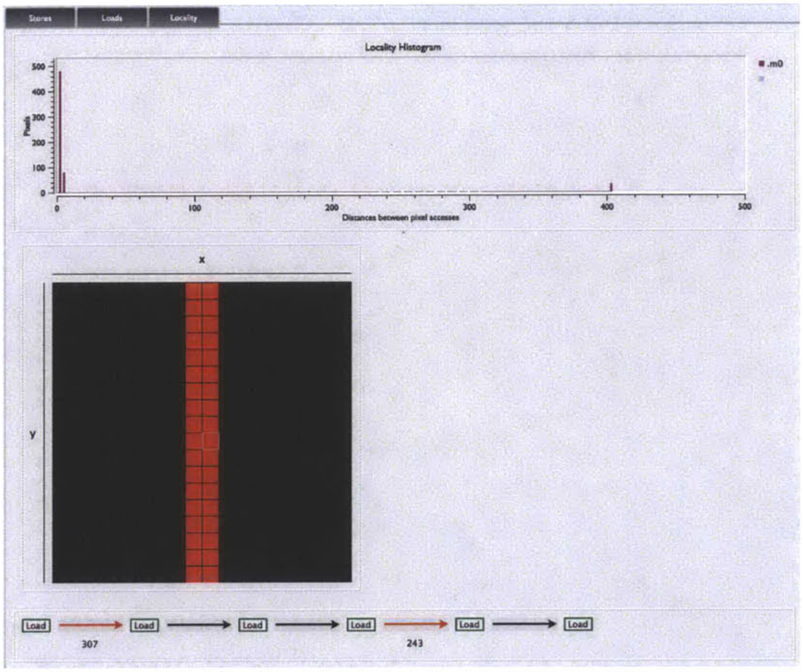

Figure 17: Locality histogram interaction and visualization of distances between pixel accesses

In Figure 17 of a locality visualization of input function in two-stage blur, user selected a range of histogram values from ~230 to ~310. All the pixels whose distances between some operations belong to that range are highlighted in red. User selected the white-bordered cell to visualize where the distances are coming from. Chain of operations for that pixel shows all the operations in order and red arrows mark the distances that belong to the selected histogram range. These distances are 307 and 243 and happened between first and second load, and fourth and fifth load. These numbers represent the number of different pixels processed and cached between the operations around the red arrows.

Application, like prevous modules, supports functions with up to 4 dimensions (Figure 18). GUI interaction is similar as in 2D case only this time, user has to select not only X, Y location but coordinates form other dimensions as well.

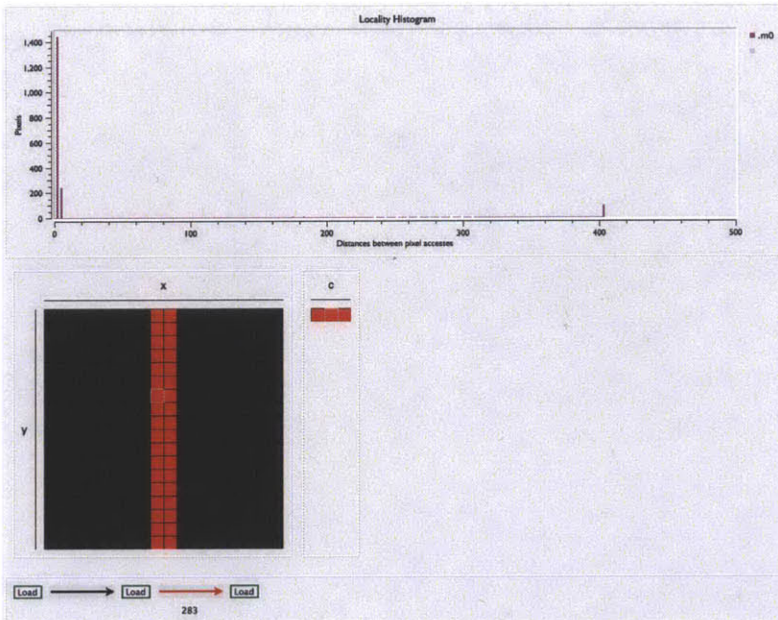

Figure 18: Locality histogram interaction and visualization of distances between pixel accesses in 3D functions

Locality visualization helps evaluate schedule's locality and diagnose critical areas that might suffer from week locality. This helpes developers further understand and improve their schedules.

3.5.3 Parallelism

Unfortunately, given the nature of the trace files, we were unable to extract useful information about parallelism. It is possible to, using the dependencies calculated in the dependency visualizer, calculate the maximum parallelism available in an algorithm. This number would purely depend on the algorithm and wouldn't help analyze Halide's schedules. Further more, it was clearer what the optimal values for locality and redundancy were: we want to minimize the amount of redundant work and minimize the number of cache misses. It is not as clear what the optimal value for parallelism would be. While it is good to have some parallelism, introducing too much of it with small granularity can cause more performance harm than benefits. These are the reasons why parallelism wasn't pursued in this thesis.

It is possible for the developer to gain some insight about parallelism simply from observing the execution visualizer. With enough understanding and intuition, Halide developer can deduce some information about parallelism by observing the execution timeline and tiling and vectorization occurences.

3.6 Schedule Comparison

This tool so far provided ability to analyze and visualize individual schedules. In order to find an optimal schedule, it is useful to compare two different ones in terms of redundant work and locality. Doing the comparison can give insight into whether we are moving in the adequate position, and give better understanding of the tradeoff between the two variables.

Every time a trace is loaded into the application we can save the processed data into a binary file. Application saves all the histogram data containing information about redundancy and locality. Later on, when a different schedule is being analyzed, we can load the data from the previous schedule and perform a data comparison.

Module for comparison consists of displaying three kinds of histograms. First two are the ones described before that contain the information about the number of stores and the distances between pixel accesses. The third one is a variant of a locality histograms. Distance data is processed and a reverse cumulative histogram is created. Figure 19 shows an example of all three histograms. The third histogram represents the number of cache misses depending on a cache size. It decreases with the size of the cache. Smaller cache means less data can fit into cache, which leads to a quick disposal of unused values. Larger cache means we can keep the unused data in it for a longer period of time. Pixels with large distances between loads will get thrown out of small caches.

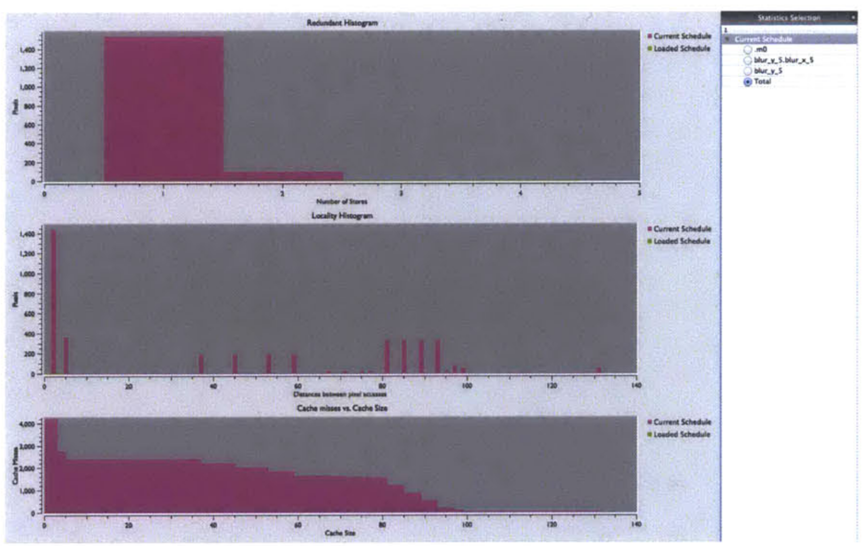

Figure 19: Redundancy, distances between accesses and cache misses histograms

By default, these histograms represent the total histograms of a schedule, summarized data across all functions. User can choose to view only the data from a particular function by making a selection in the radio buttons on the side.

When a data from another schedule is loaded, histogram data of the loaded schedule is laid over data from the current schedule (Figure 20).

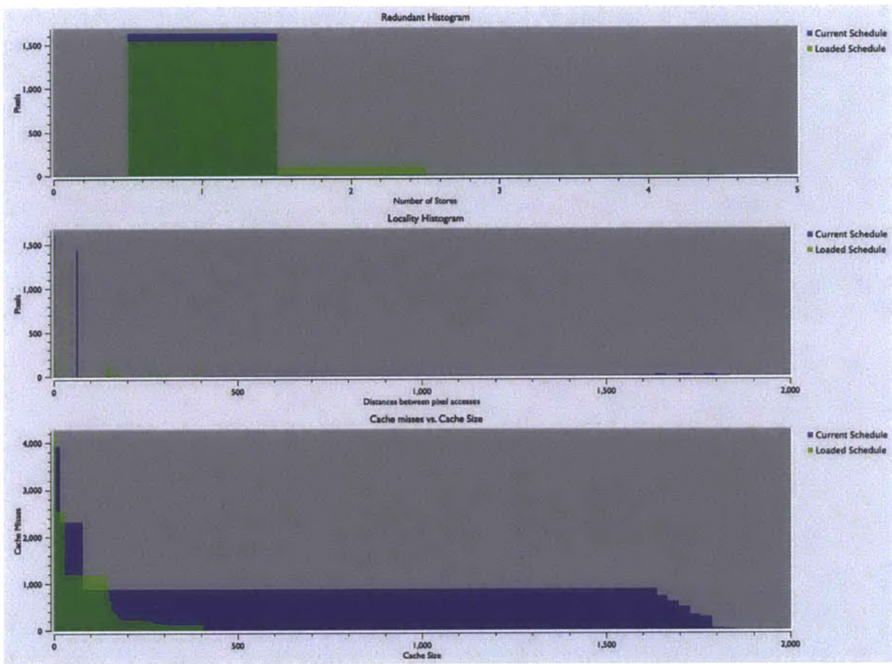

Figure 20: Comparison of two schedules

Comparing them, it becomes more obvious what are the advantages and disadvantages of each of the schedules, and gives programmers an opportunity to choose the appropriate one.

4 Results

Section 1 described three extreme schedules for two-stage blur algorithm. Strategy 1 favored redundancy while sacrificing locality. Strategy 2 improved the locality by introducing some redundant work. Strategy 3 performed well in terms of redundancy and locality, but it completely got rid of all available parallelism. Analysis was done using the intuition behind schedule loop synthesis. Developers can often try to analyze schedules in this way, but it becomes a tedious job for large input files, multi-dimensional functions and complex schedules.

Visualization and analysis tool enables developers to understand schedules and their advantages and disadvantages in a comprehensive, graphical and easy way. Analysis of Strategy 1 produced the Figures 21 and 22.

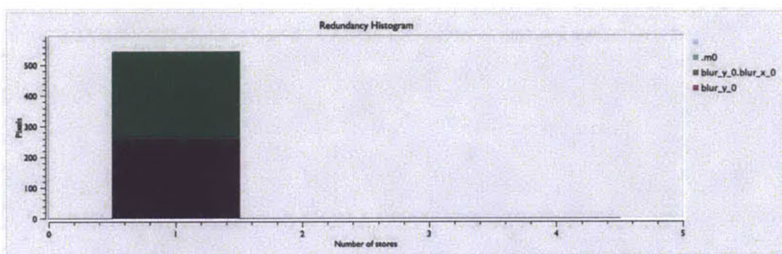

Figure 21: Redundancy total histogram for Strategy 1

Figure 1 shows that all the computation in Strategy 1 happens exactly once i.e. no redundant work is introduced. On the other hand, locality histogram for *blur_x* (Figure 22) shows some smaller peaks of large distances (red rectangle on the histogram). Chain of operations shows that these large distances are coming from *blur_x* data being used long after it is computed. The distances between the first store and the first load is large, which is consistent with our intuitive analysis. Whole *blur_x* is computed and stored before any of its values are used, making the locality between the first and second access rather poor.

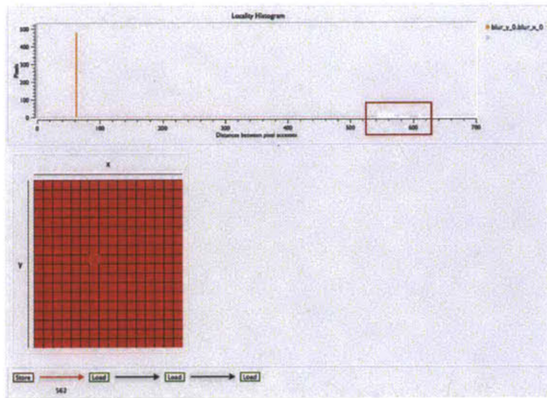

Figure 22: Locality histogram of blur_x for Strategy 1

Strategy 2 introduces redundant computation as visible in the Figure 23. Many pixels are recalculated two or three times.

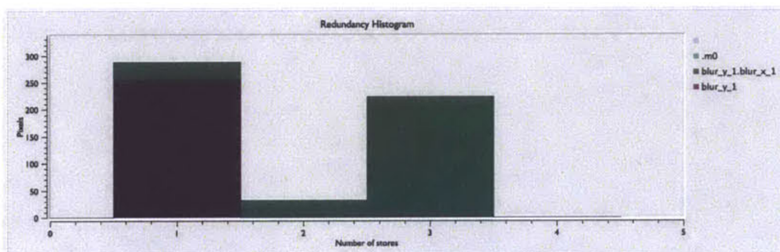

Figure 23: Redundancy total histogram for Strategy 2

On the other hand, Strategy 2 significantly improves locality. The distances between pixel accesses are at most 8, significantly better than locality of Strategy 1. (Figure 24)

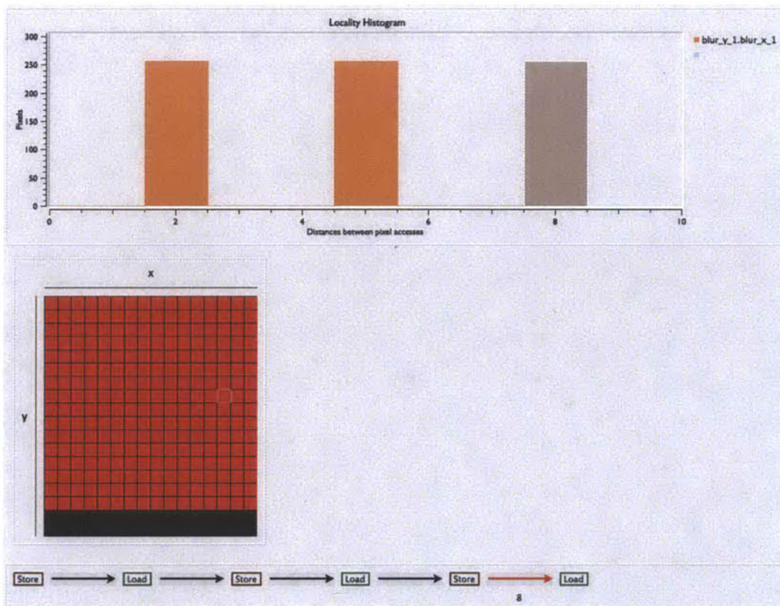

Figure 24: Locality histogram of *blur_ x* for Strategy 2

Comparison of the two schedules gives a nice summary of both properties. In Figure 25 histograms of the two schedules are overlaid on top of each other. Pink schedule represents Strategy 1 while purple one represents Strategy 2. Purple schedule introduces redundant computation on almost half of all its pixel values. Cache misses histogram shows that for small cache sizes, two strategies perform similarly. As the cache grows, cache misses are completely eliminated from Strategy 2, making it superior in terms of locality.

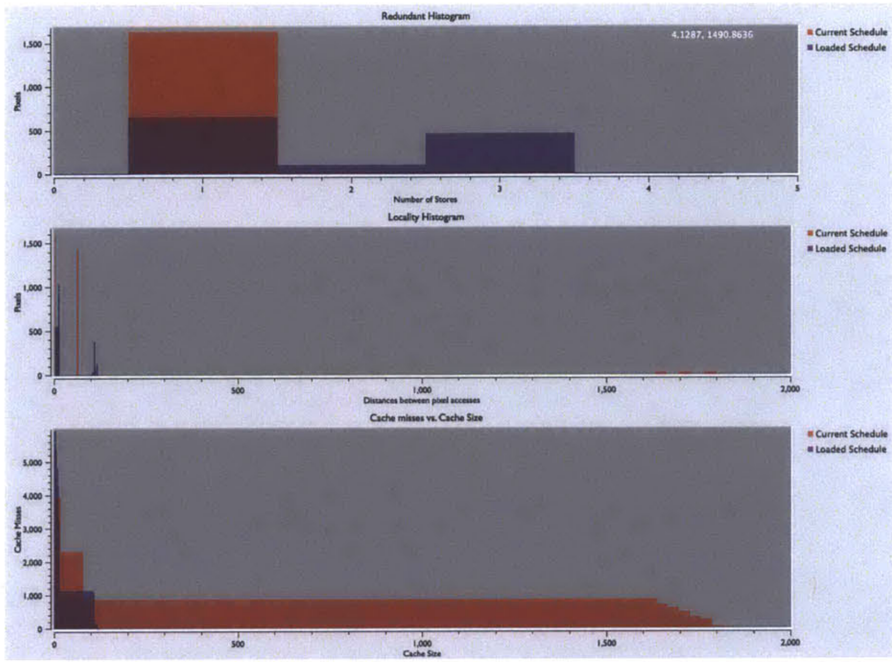

Figure 25: Strategy 1 vs. Strategy 2

Finally, Strategy 3 also achieves optimal amount of work like Strategy 1. (Figure 26)

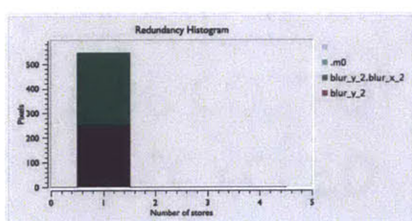

Figure 26: Redundancy total histogram for Strategy 3

However, even though its locality is not as good as Strategy 2's, it is significantly better than the first Strategy in that regard. Figure 22 shows that Strategy 1's distances between pixel accesses goes up to ~550. Strategy 2, even though spread out, achieves maximal distances between accesses of approximately 120. (Figure 27)

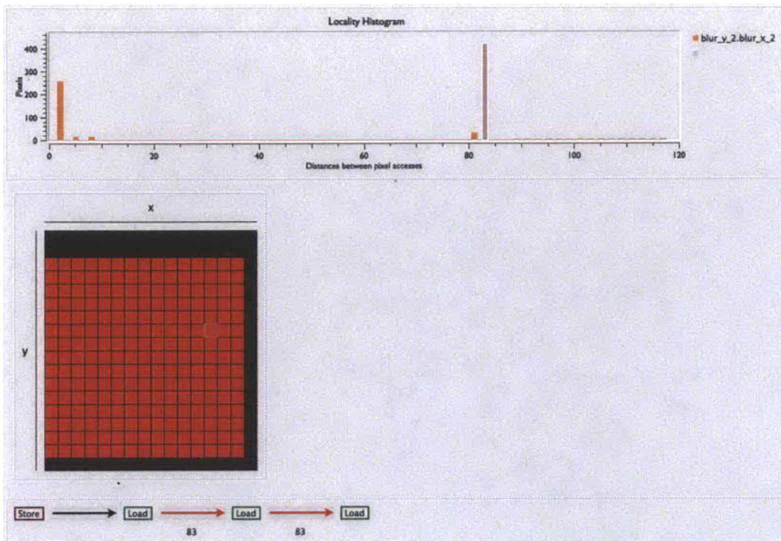

Figure 27: Locality histogram of *blur_x* for Strategy 3

To further amplify the results, we can compare every schedule with each other. (Figure 28 and Figure 29) Strategy 2 achieves better locality than any of the other ones, but it performs more significantly more redundancy than both of them. Strategy 3 locality is better than Strategy 1 while keeping redundant work almost optimal.

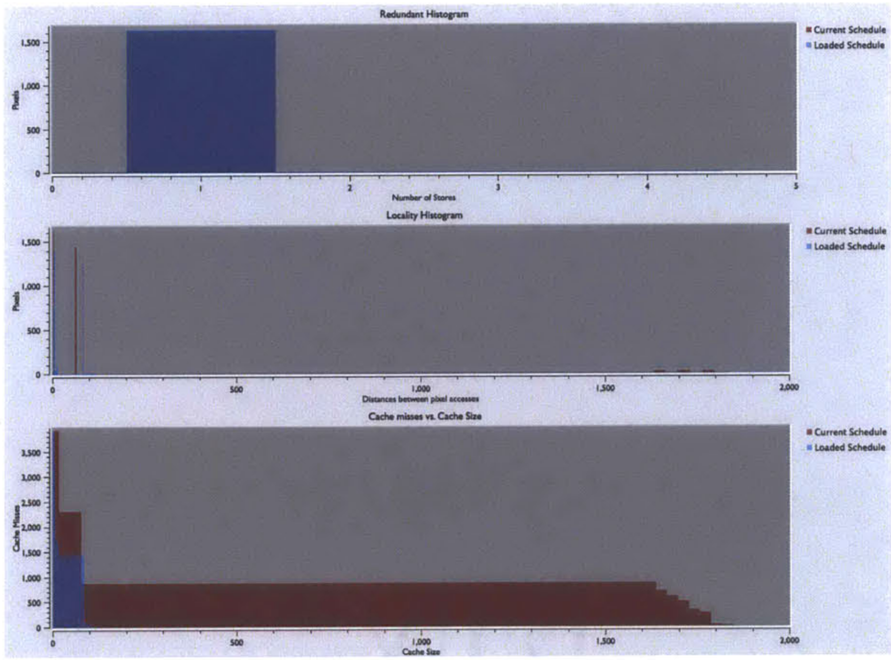

Figure 28: Strategy 1 vs. Strategy 3

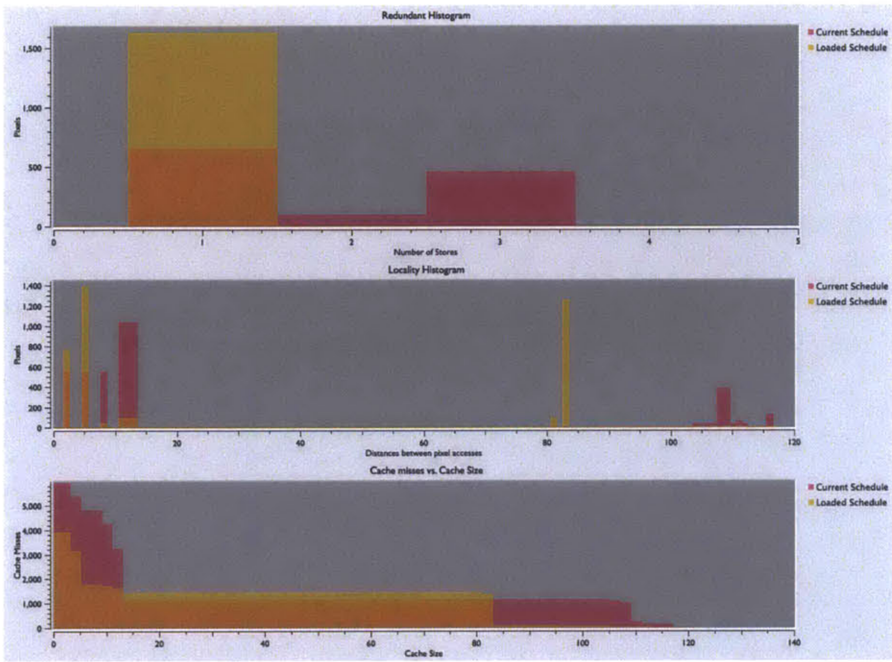

Figure 29: Strategy 2 vs. Strategy 3

These results are consistent with our intuitive analysis from Section 1. We have to keep in mind that these analysis are not complete since we are lacking parallelism data. However, a lot about parallelism can be deduces from observing the visualization of the schedule.Visualization provides insight into what schedules apply sliding window strategy that minimizes locality. In this case, even though Strategy 3 would be deemed superior in terms of locality and redundancy, it will constrain the amount of parallelism and thus not be the best choice for optimal two-stage blur.

Most of the time, the best schedules will not lie in the corners of the redundancy-locality-parallelism, but somewhere in the area of the triangle. The best strategy for two-stage blur won't be any of the strategies analyzed above. One of the better ones is a tiling schedule that divides *blur_x* into 4 separate regions. Tiling is easy to spot in visualizing module, and the developer can deduce that tiles can be computed i parallel, while within the tiles work can be done sequentially or in parallel. This introduces some amount of parallelism, while behaving well in terms of locality and redundancy. Figure 30 shows the total redundancy and locality of tiling schedule. It's not as optimal in redundancy and locality as some of the strategies mentioned above, but as a combination of all three parameters, it performs well.

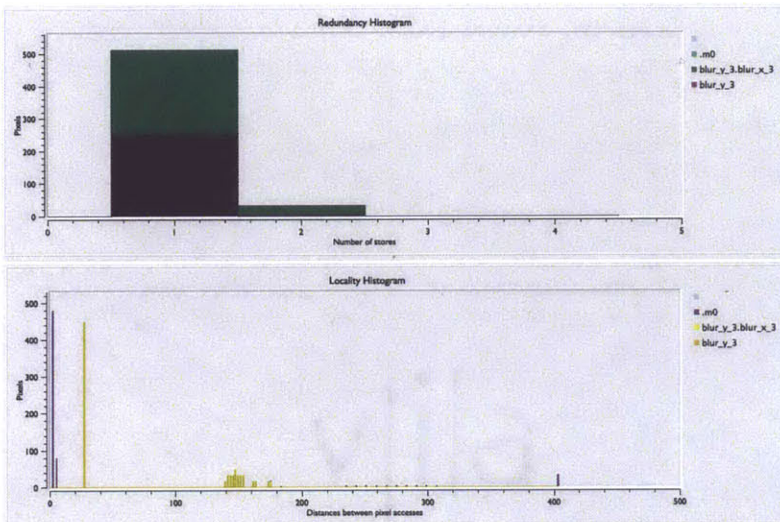

Figure 30: Redundancy and Locality total histograms for a tiling strategy

Finally it's all about the trade-off between locality, redundancy and parallelism. This thesis provided a tool to better analyze these properties for every schedule, ability to understand schedules better by visualizing the execution and the ability to compare schedules and the three variables. Overall, it provides a good and detailed solution to schedule analysis, minimizing the tedious intuitive analysis. Sometimes, we might have a wrong intuition, and schedule analyzer helps us understand what is really going on.

5 Conclusion

The tool presented in this thesis helps visualize, understand and evaluate properties of Halide schedules. Future work should deal with the performance of the visualization tool. Disk storage versus RAM storage would have to be carefully designed for each module to minimizes RAM consumption, and to reduce waiting times. This would also provide insight into complex schedules with large data sets. Visualization tool still performes well with moderately complex schedules and it provides good analysis of some of the most important aspects of Halide schedules.

Execution visualizer will be especially helpful to people who are just entering the world of Halide and algorithm-schedule separation. It will lead them to better understanding of what is going on in the background of a Halide program. Execution visualizer is also useful for detecting the overal flow of the execution and noticing when tiling and vectorization are being used. This additionally helps gain visual intuition into how much parallelism is available; it is easier to deduce while looking at the visualization of the execution than by picturing it in one's head. Dependency visualisation is the only module that focuses on Halide algorithms by providing insight into te inner dependencies of the algorithm. Statistical analyzer represents crucial data about redundancy and locality in a graphical way. It not only presents useful data to the users, but it enables them to diagnose places of failure by highlighting critical regions within functions. Finally, tool enable schedule comparison, all in service of minimizing programmer's manual work and laying out useful data in front of him. By comparing different schedules, programmers get a better understanding of the advantages and disadvantages of each schedule, and helps them move in a rght direction towards a better scheduling option.

References

[1] J, Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand and S. Amarasinghe. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *PLDI 2013*

[2] http://qt-project.org/

[3] http://qwt.sourceforge.net/