HalideTuner: Generating and Tuning Halide Schedules with Opentuner

by

Giuseppe Paolo Ernesto Toffanin Zingales

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

September 2015

© Giuseppe Paolo Ernesto Toffanin Zingales, MMXV. All rights reserved.

The author hereby grants to MIT permission to reproduce and to distribute publicly paper and electronic copies of this thesis document in whole or in part in any medium now known or hereafter created.

Signature redacted

Author .............................................................

Department of Electrical Engineering and Computer Science

Signature redacted

July, 14, 2015

Certified by............. ........................................

Saman Amarasinghe
Professor
Thesis Supervisor

Signature redacted

Accepted by ............ ........................................

Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee
HalideTuner: Generating and Tuning Halide Schedules with Opentuner

by

Giuseppe Paolo Ernesto Toffanin Zingales

Submitted to the Department of Electrical Engineering and Computer Science on July, 14, 2015, in partial fulfillment of the requirements for the degree of Master of Engineering in Computer Science and Engineering

Abstract

Halide is a programming language designed to make it easier to write high-performance image processing code. Halide separates the scientific problem of how to write the image processing algorithm to get the required result from the engineering problem of how to schedule the different parts of this algorithm to get a fast solution. This allows Halide’s users to test out different schedules for a single image processing algorithm to get the fastest program. Unfortunately, the amount of possible schedules is staggeringly large, and there is no deterministic way to find the fastest schedules. This leads to a lot of man-hours spent trying to find good Halide schedules.

As Halide grows in popularity and the platforms on which it runs diversify, the need for Halide schedules will increase. As the need for good Halide schedules increases, so will the need for good schedule designers. However, given the elusiveness of highly performant schedules, machine generated schedules will become highly preferred to human trial and error. HalideTuner solves this problem by automating the entire process.

The automation of the the schedule generation and tuning is done by a machine learning framework called Opentuner. Before Opentuner can operate, defining how to iterate through the search space needs to be codified. In this paper we present HalideTuner a codification of how Opentuner can manipulate Halide schedules to improve them in the process of clever trial and error. HalideTuner serves as the foundation for generating and tuning Halide schedules.

Thesis Supervisor: Saman Amarasinghe
Title: Professor
Acknowledgments

Saman Amarasinghe
Shoaib Kamil
Jason Ansel
Jonathan Ragan-Kelley
Tyler Denniston
Contents

1 Introduction .................................................. 15

2 Halide .......................................................... 19
   2.1 What is Halide ............................................ 19
   2.2 Single-Function Schedules ............................... 22
      2.2.1 Reorder LoopNodes ................................ 24
      2.2.2 Change Loop Type ................................ 24
      2.2.3 Split LoopNode .................................... 25
      2.2.4 Tiling ............................................. 26
   2.3 Multi-Function Schedules ................................. 28
      2.3.1 ComputeNode ....................................... 29
      2.3.2 StoreNode .......................................... 29
      2.3.3 Inlining a Function ................................ 30
      2.3.4 Lower ComputeNode ................................ 30
      2.3.5 Lower StoreNode ................................... 30
   2.4 Conclusion ................................................. 32

3 Opentuner ...................................................... 33
   3.1 What is Opentuner ....................................... 33
   3.2 Defining the Search Space ............................... 34
   3.3 Calculating the Value ................................... 34
   3.4 Opentuner Techniques ................................... 35
   3.5 Why Choose Opentuner ................................... 35
9 Conclusion

A Tree Schedules for Halide Users

B Opentuner Search Space
   B.1 Transformation Tuple
   B.2 Manipulator
   B.3 Base Schedule

C Meta Schedule

D Lanka

E Graphs of results
   E.1 Bilateral Grid
   E.2 Blur
   E.3 Laplacian
   E.4 Wavelet
# List of Figures

1-1 Example of a basic black and white filter ............................................ 16
2-1 Example of Call Graph, Loop Order, and Tree-Schedule. .................... 20
2-2 A key of what the colours and shapes mean in a Tree-Schedule. ....... 21
2-3 Example of single function loop structure ......................................... 23
2-4 Tree being transformed by simple reorder ......................................... 25
2-5 Applying a Split node transformation ............................................... 26
2-6 An example of how a tree mutates before and after a tiling occurs .... 28
2-7 Example of a StoreNode being lowered ............................................. 32
4-1 Example of a Opentuner search space for HalideTuner ....................... 39
4-2 This graph shows the initial boot of HalideTuner and Opentuner ....... 40
4-3 Graph showing one iteration in the HalideTuner loop ....................... 41
7-1 Blur from 1 to 24 hours ................................................................. 52
7-2 Wavelet from 1 to 24 hours .............................................................. 53
7-3 Bilateral Grid from 1 to 24 hours ..................................................... 54
7-4 Local Laplacian from 1 to 24 hours ............................................... 55
A-1 This is a basic example of Halide schedule as a Tree-Schedule tree . . 62
A-2 This is a more complex example of Halide schedule a Tree-Schedule . 62
E-1 Bilateral Grid from 0 to 1 hour ......................................................... 72
E-2 Bilateral Grid from 1 to 24 hours ..................................................... 72
E-3 Blur from 0 to 1 hour ................................................................. 73
E-4  Blur from 1 to 24 hours ........................................ 73
E-5  Local Laplacian from 0 to 1 hour ............................. 74
E-6  Local Laplacian from 1 to 24 hours .......................... 74
E-7  Wavelet from 0 to 1 hour ...................................... 75
E-8  Wavelet from 1 to 24 hours .................................... 75
Chapter 1

Introduction

Image processing has become immensely widespread as it is heavily used in applications such as digital photography, video editing and robotic vision. As these applications get more advanced, so does the requirement to build faster image processing routines. Halide is a novel solution to this problem. One of the biggest time sinks for building fast image processing routines is to change how the computer iterates over the image while maintaining the algorithm’s correctness. Halide solves this problem by separating the two concerns. On one side, we have the algorithm of the image processing routine that describes how the image should be transformed and on the other side, we have how the computer iterates over an image, which is called the schedule. An example of the algorithm is a camera filter which makes an image black and white. The schedule could instead describes how the computer iterates over the image: it could start at the top left most image and go down, or alternatively start at the top left and go to the right. Figure 1-1 contains a visual aid.

While converting an image to black and white is an extremely simple image processing routine, most algorithms are significantly more complex. The implications of a separate schedule is that it gives the programmer flexibility and ease of mind. Thanks to Halide’s separation of algorithm and schedule a programmer can try out a series of different schedules without ever once having to worry about the correctness of their routine, since one a Halide algorithm is correct it is correct forever. This allows a programmer to iterate very quickly through schedules that would normally
Func BlackAndWhite(x, y) = \text{avg}((\text{InputImage}(x,y).\text{rgb}()))

BlackAndWhite.reorder(x, y)  BlackAndWhite.reorder(y, x)

\begin{tabular}{|c|c|c|c|}
\hline
0 & 1 & 2 & 3 \\
\hline
0.1 & 1.0 & 2.1 & 3.1 \\
0.2 & 1.2 & 2.2 & 3.2 \\
0.3 & 1.3 & 2.3 & 3.3 \\
\hline
\end{tabular}

*In a reorder outside loop goes last

Figure 1-1: Example of a basic black and white filter

have taken much longer to get working.

Halide’s library now provides an easy way to define different schedules, but the number of possible schedules for any non trivial application is virtually limitless. Luckily the number of useful schedules is more manageable, but it is still too many to be computed by hand. This would not be problematic if Halide schedules were not such subtle creatures. It is often impossible to intuit what a good schedule is. It furthermore requires time to understand the problem, think of the machine, and try out dozens of different schedules. Even after all this, the schedule is only optimized for a set of specific architecture properties. As architectures evolves and to keep a Halide schedules as preformant as possible it usually needs to be updated to reflect these new features.

In this thesis we introduce HalideTuner, which allows novice Halide users to generate an autotuned schedule in less than 24 hours with very little work from the programmer. HalideTuner accomplishes this with the help of Opentuner using machine learning techniques to intelligently search through thousands of Halide schedules to find the fastest one for a given algorithm\(^1\)

\(^1\) HalideTuner tries on order of 7-8 thousand schedules in a 24 hour period, assuming the runtime
of the average schedule is on the order of 5 seconds.
THIS PAGE INTENTIONALLY LEFT BLANK
Chapter 2

Halide

This chapter will be dedicated on helping understand what a Halide schedule is, and how different functions modify the schedule to make it faster\(^1\).

2.1 What is Halide

Halide is an image processing language, which is embedded and domain-specific. It works by de-coupling the algorithm of an image processing routine and the schedule of the routine. The schedule is basically how one manipulates the loop structure and computation of the image processing routine. The de-coupling allows the user to create the algorithm for a routine once, and then easily test different schedules to find the fastest one without having to worrying about correctness.

One should think of a Halide schedule\(^2\) as a tree which interleaves three things: it interleaves a call graph for a set of functions, the looping structure for these functions, and the storing/execution of these functions. In Figure 2-1 one can see an example of a call graph, a basic looping order, and a complete Halide schedule.

To help one understand Halide schedules, we shall deconstruct a schedule into

---

\(^1\)This chapter will be very analogous to (and is completely based on) chapter 7 from Jonathan Regan-Kelley’s Thesis [1].

\(^2\)This paper will talk exclusively about the Tree-Node representation of Halide schedules described in JRK’s work and not the C++ implementation of schedules. Switching between one and another is simple since all the commands are a combination of commands. Please refer to Appendix A for more detailed explanation.
Figure 2-1: Example of Call Graph, Loop Order, and Tree-Schedule.
Figure 2-2: A key of what the colours and shapes mean in a Tree-Schedule.
parts and slowly show what each transformations do to the schedule. Here is a full list of transformations that one can apply to a schedule:

They are clustered as pairs of inverses, those without pairs are their own inverses.

- reorder
- change loop type
- split/fuse node(s)
- raise/lower storage
- raise/lower compute
- de/inline function

### 2.2 Single-Function Schedules

Let us being with single functions schedules with only LoopNodes. While these schedules are not particularly interesting, they let us explore what the tree structure strives to convey. Throughout this chapter blur will be used as the example of choice to help describe the structure and the transformations. Blur is comprised of two functions, but for now we will only talk about BlurHorizontal. BlurHorizontal takes an average of three neighboring pixels. This function has to iterate over the entire image so by default it has two LoopNodes described below:

```plaintext
Func BlurHrz(x,y) = (InputImage(x-1,y) + InputImage(x,y) + InputImage(x+1,y))/3
LoopNode BlurHorizontalX = {func = BlurHrz, var = x, extent = image.width}
LoopNode BlurHorizontalY = {func = BlurHrz, var = y, extent = image.height}
```

The above code tree representation would look like Figure 2-3.

Therefore the analogous pseudo-code for the loop structure can be seen below:

---

3This paper shall gloss over the requirements for each transformation. If you are interested in this please refer directly to the code because it has been painstakingly double checked. One can see most of the checks for each transformation by looking at can-<transformation> functions in HalideSchedule.py.

---

22
for BlurHorizontalY.var in range BlurHorizontalY.extent:
    for BlurHorizontalX.var in range BlurHorizontalX.extent:
        Image[BlurHorizontalX.var, BlurHorizontalY.var] = BlurHorizontal(BlurHorizontalX.var, BlurHorizontalY.var)

There are a series of important things to note from this Figure. First, each LoopNode signifies a for-loop in the BlurHorizontal function. A nesting of all these LoopNodes creates the entire domain which the function must iterate over. Second, if two loops are nested in the tree-structure they are a parent-child pair. On the other hand, if the loop nodes are serial they will be siblings in the tree-structure.

The ovals in the figure signify LoopNodes. The rhombus in the figure signifies a ComputeNode. While in this example they are not necessary, they signify the location where the computation for the BlurHorizontal actually takes place. This node can only be placed after all the required variables are defined.

There are four transformations one could apply to a one-function schedule:

1. reorder
2. change loop type

This is not represented in the figure above but it is true nonetheless.
3. split
4. fuse.

While individually these functions might seem frivolous it is their combination in which they become powerful.

2.2.1 Reorder LoopNodes

Reorder is the most straightforward transformation to understand so it shall be described first. In a Halide schedule the idea of reordering loops is represented by switching a child and parent. This essentially means that it rearranges the nesting-order of the loops⁵.

In pseudo code a reorder(x, y) would take this code:

```python
for BlurHorizontalY.var in range BlurHorizontalY.extent:
    for BlurHorizontalX.var in range BlurHorizontalX.extent:
        Image[BlurHorizontalX.var, BlurHorizontalY.var] =
            BlurHorizontal(BlurHorizontalX.var, BlurHorizontalY.var)
```

and transform it into this code:

```python
for BlurHorizontalX.var in range BlurHorizontalX.extent:
    for BlurHorizontalY.var in range BlurHorizontalY.extent:
        Image[BlurHorizontalX.var, BlurHorizontalY.var] =
            BlurHorizontal(BlurHorizontalX.var, BlurHorizontalY.var)
```

The tree representation would mutate as shown in Figure 2-4

2.2.2 Change Loop Type

This transformation changes how a variable will loop over its extent. The default type is Serial, which essentially says that it will loop over the entire domain one at a time. You could however Vectorize the loop which means that it will loop over its

⁵It is important to remember that rearranging these loops does not impact the correctness of the solution.
extent using vectorized instructions. The third type is Parallelized which means that
the loop will be split amongst different threads and computed in parallel. The final
type is Unrolled which simply unrolls the loop.

Since these are intrinsic properties of the loop, the tree structure does not change.
To denote these different states the colour of the LoopNode will change. A program-
mer does not need to worry about correctness or boundary conditions, since this is
handled behind the scenes by Halide.

2.2.3 Split LoopNode

Splitting a node essentially means covering the extent of one node with two nodes.
This split is not just dividing the extent in half and having one node do the first half
and one node do the second half\(^6\). Instead, this creates two nested loops which cover
the extent in sections. Below there is an example in pseudo-code which starts with a
normal LoopNode \(y\) with extent 200.

\[
\text{for } y \text{ in } [0..200]: \\
\quad \text{new-image}[x][y] = (\text{image}(x-1,y) + \text{image}(x,y) + \text{image}(x+1,y))/3
\]

A split of value 10 is applied to our LoopNode \(y\), and the two new vars are called

\(^6\)If that was the case the tree node structure would show the nodes appearing one after the other
as siblings, not as a parent child pair.
Figure 2-5: Applying a Split node transformation

\[
\begin{align*}
\text{for } y_{\text{Outer}} \text{ in } [0..20]: \\
& \hspace{1em} \text{for } y_{\text{Inner}} \text{ in } [0..10]: \\
& \hspace{2em} y = y_{\text{Outer}} \times 10 + y_{\text{Inner}} \\
& \hspace{2em} \text{new-image}[x][y] = (\text{image}(x-1,y) + \text{image}(x,y) + \text{image}(x+1,y))/3
\end{align*}
\]

As one can see, these two nodes (\(y_{\text{Outer}}\) and \(y_{\text{Inner}}\)) still cover the entire extent that the previous node (\(y\) and the computation is spanned by two nodes).

The tree-structure format for this is substantially terser and can be seen in Figure 2-5.

**Fuse**

Fuse is simply the inverse of a split, it takes two nodes and merges them into one node.

**2.2.4 Tiling**

A good application of these functions would be tiling. This takes a large area and divides it into tiles. These tiles are parallelized and computed individually. The
benefit of this is the ability to divide and conquer, and of cache-locality that you get when you tiling since each tile fits into cache.

Below you can see the Halide Schedule and pseudo-code for a tiled and parallelized Halide Schedule. The resulting tree-schedule can be seen in Figure 2-6.

---

### Halide Schedule:

```
split(x, x_outer, x_inner, 64)
split(y, y_outer, y_inner, 64)
reorder(x_inner, y_inner, x_outer, y_outer)
fuse(x_outer, y_outer, tile_index)
change_loop(tile_index, parallelize)
change_loop(x_in, vectorize)
```

### Pseudo-code:

```
parallelized for tile_index in [0..(4*3)]:
    y_outer = tile_index / 4
    x_outer = tile_index % 4
    for y_inner in range [0..64]:
        for x_temp in [0...64/4]:
            #vectorized X_inner
            x = x_outer * 64 + x_temp*4
            x_inner[4] = {x + 0, x + 1, x + 2, x + 3}
            y = y_outer * 64 + y_inner
            vectorized do {
                Func(x_inner[0], y)
                Func(x_inner[1], y)
                Func(x_inner[2], y)
                Func(x_inner[3], y)
            }
```

This concludes all the transformations that can be applied to single-function schedules.
2.3 Multi-Function Schedules

Halide developers recommend having multiple small schedules rather than one big one. This is because checking correctness of a bunch of small functions is easier than for one massive function. As a result most schedules are multi-function schedules. Blur for instance has two functions BlurHorizontal and BlurVertical. When a schedule is multi-functional it has and these schedules have a few more transformations. The new transformations are as follows:

- raise/lower storage
- raise/lower computation
- de-inline/inline function

It is important to understand a couple of things about these new schedules. Every Node in the tree corresponds to one and only one function, and there are two new Nodes that you need to understand: ComputeNodes and StoreNodes.

We will continue to use Blur as the base for examples below is a full definition of Blur, which includes both of its functions.

```python
Func BlurVert = {name: BlurVertical, vars = [x, y]}
```
LoopNode BlurVerticalX =
   {func = BlurVert, var = x, extent = image.width}
LoopNode BlurVerticalY =
   {func = BlurVert, var = y, extent = image.height}

Func BlurHrz = {name: BlurHorizontal, vars = [x, y]}
LoopNode BlurHorizontalX =
   {func = BlurHrz, var = x, extent = image.width}
LoopNode BlurHorizontalY =
   {func = BlurHrz, var = y, extent = image.height}

2.3.1 ComputeNode

ComputeNodes are nodes that describe to the user where the specific function will be computed. Therefore a ComputeNode will always be after all of the variables in its argument list (otherwise how could you compute the result?). By moving where we compute the function we can take advantage of the cache to reduce time spent getting this information from memory.

2.3.2 StoreNode

StoreNodes signify the granularity at which resulting values of the function are stored\(^7\). The StoreNode's relative size can be calculated by the number of LoopNodes of the same function that are below it. For example, if the StoreNode is the ancestor of all the LoopNodes of the same function, this is equivalent of memoizing the entire range of that function. On the other hand if the StoreNode is below all of the LoopNodes this is equivalent to not storing any result.

\(^7\)Every function that is not the output function has a store node.
2.3.3 Inlining a Function

Inlining a function F simply means that each value of F will be computed as necessary and these values will not be stored. Thus they must be recomputed upon demand. This is often used if a function is a very simple conversion of another function, like a scalar multiple. Spending time and energy creating and storing the intermediate values is more expensive than computing them on the fly.

De-inlining a Function

De-inlining a function is the exact opposite of inlining a function. It removes all inlined ComputeNodes from their various locations and creates one basic subtree. This is only useful if you wanted to undo an inlining operation.

2.3.4 Lower ComputeNode

Lowering a ComputeNode means it lowers the computation by one node in the tree of its caller. This means that more of the function will be computed closer to the ComputeNode of the caller. This essentially refines the granularity at which the function is computed.

Hoist ComputeNode

Hoist ComputeNode is the inverse of Lower ComputeNode.

2.3.5 Lower StoreNode

Lowering a StoreNode essentially decreases the size of what you are storing. This operation will move a StoreNode down the tree exactly one of the LoopNodes in its function. The visualization of this in a tree-node looks like a StorageNode and a LoopNode switch places, as can be seen in Figure 2-7.

Below we have a loop level pseudo-code that displays the affect of the transformation.
BlurVert Storage = [BlurVerticalX][BlurVerticalY]

for BlurVerticalX.var in range BlurVerticalX.extent:
    for BlurVerticalY.var in range BlurVerticalY.extent:
        BlurVertStorage[BlurVerticalX.var, BlurVerticalY.var] =
        BlurVertical(BlurVerticalX.var, BlurVerticalY.var)

for BlurHorizontalX.var in range BlurHorizontalX.extent:
    for BlurHorizontalY.var in range BlurHorizontalY.extent:
        Image[BlurHorizontalX.var, BlurHorizontalY.var] =
        BlurHorizontal(BlurVertStorage[BlurHorizontalX.var][BlurHorizontalY.var])

transforms into:

for BlurVerticalX.var in range BlurVerticalX.extent:
    BlurVert Storage = [BlurVerticalY]
    for BlurVerticalY.var in range BlurVerticalY.extent:
        BlurVertStorage[BlurVerticalX.var, BlurVerticalY.var] =
        BlurVertical(BlurVerticalX.var, BlurVerticalY.var)

for BlurHorizontalX.var in range BlurHorizontalX.extent:
    for BlurHorizontalY.var in range BlurHorizontalY.extent:
        Image[BlurHorizontalX.var, BlurHorizontalY.var] =
        BlurHorizontal(BlurVertStorage[BlurHorizontalX.var][BlurHorizontalY.var])

Hoist StoreNode

Hoist StoreNode is the inverse of Lower StoreNode, which basically just increases the size of what is being stored.
Fig. 2-7: Example of a StoreNode being lowered

Imperfection of Inverses

While every Halide transformation has an inverse, it is important to note that these inverses are not always perfect. They have to be applied correctly and with the schedule in a specific state, to be perfect inverses. For example, if you inline and then de-inline a function, all the differences in loop types will be lost. So de-inlining a function is only a perfect inverse of inlining a function if the function has only Serial LoopNodes. This is true for many other use cases.

2.4 Conclusion

Halide schedule allows you to manipulate and interleave the loop structure for an algorithm. However, there are many limitations of what can be manipulated and interleaved. The transformations thus described are the only ones available in Halide-Tuner. This is an interesting mix of power and limitation that must all be encoded and understood by Opentuner.
Chapter 3

Opentuner

In addition to Halide, the other important library that will be used in HalideTuner is Opentuner. Opentuner is the pack mule that does the grunt work of trying thousands of different schedules, and recording their results. However, to understand how HalideTuner integrates with Opentuner there are some key components that one should understand.

3.1 What is Opentuner

Opentuner is an extendable autotuner; it is a machine learning framework. For Opentuner to work properly, one needs to codify three things:

1. The search space
2. The value, which is computed from running an instance in this search space.¹
3. The strategies on how to construct different, better instances in the search space.

With this information Opentuner will optimally search through the search space to get the best results.

¹Generally this is just the runtime, but this value could represent anything of interest.
3.2 Defining the Search Space

Opentuner needs to know the breadth of the search space it must look through. To inform Opentuner of the search space, one tells it of several independent parameters, each of which expresses the possible range for itself. The most basic parameter is an IntegerParameter which has a minimum and maximum bound. For example, if you wanted to tune the size of a three dimensional array the search space would be three IntegerParameters each representing a different dimensions. Then, one would set the minimum and maximum for each dimension.

The codification of this search space would look something like this:

```
IntegerParameter("width", 1, max_width)
IntegerParameter("height", 1, max_height)
IntegerParameter("depth", 1, max_depth)
```

Now there are multiple other parameters like EnumParameter, FloatParameter, PowerOfTwoParameter, and ScheduleParameter. Each of these parameters is considered independent by Opentuner and it selects an element using different techniques. Over many selections it learns which values work best and which do not.

3.3 Calculating the Value

For Opentuner to calculate the value, it needs a method that converts an instance of the search space into a value. This method is the run command which takes a dictionary of values (an instance of the search space) and gets a result which is then returned to Opentuner. Opentuner saves this result and dictionary into a database and then uses it to learn and create better instances. In our example, a possible argument to the run command would be a dictionary with these values:

```
{
  "width" : 7,
  "height" : 40,
}```

\(^2\)The names are pretty self explanatory but refer to the Opentuner docs for more info.
It would then run the program and return the runtime of 0.80 seconds.

3.4 Opentuner Techniques

Techniques are really where the learning happens. Opentuner techniques can search through the database of previous instances (also called configurations) and runtimes and use these as information to create better configurations. Luckily, Opentuner already provides you with a slew of good techniques. It also provides you the interface to create your own technique if you think it would benefit your search space. A couple Opentuner Techniques will be described in Chapter 6.

3.5 Why Choose Opentuner

Since the goal was generating the best Halide schedules, there were many options to choose from including trying to create my own. It was immediately clear that creating a new tuner was not a good idea. Halide schedules is complex enough that creating a new solution with no or little experience would not have ended well. This meant that an already existing solution was the correct route. There are two reasons why Opentuner was the best option. First, there was already an existing Opentuner solution which had generated good results on a specific set of schedules\(^3\). Second, it was the most extensible option. Creating HalideTuner with all its different caveats was easily done in Opentuner.

---

\(^3\)This solution was convoluted enough and specific enough that it could not easily be generalized for all Halide schedules.
Chapter 4

HalideTuner

HalideTuner is the codification of the 'define the search space', and 'calculate the value' steps that Opentuner framework requires. HalideTuner defines the search space as a series of 100 transformations. It then supplies functions that given a list of transformations will create a schedule, compile it, run it, and return the average runtime. With these two functions Opentuner is able to probe the search space with different configurations and through a series of probings gain an understanding of the space, and how to manipulate its further probings. On top of this, HalideTuner also provides techniques, and associated tools, which help Opentuner search through the space more effectively.

4.1 Aims of HalideTuner

When creating HalideTuner there are a couple of goals we tried adhere to. First, is that that HalideTuner results are simple to reason about. Second, because of the fickle nature of Halide schedules that HalideTuner is not capped by how well any single programmer understand Halide schedules. Finally that HalideTuner was flexible enough to search the entire space. While following these goals we believe that HalideTuner is not only a good tuner of Halide schedules, but a tuner that is easy to improve upon and reason about so that the project can be taken improved upon by multiple different developers.
4.2 Difficulty of the Space

To understand certain design choices we made it is important to understand the particular difficulties of generating a tuner for this particular space.

1. The search space is too large to search exhaustively.
2. The best schedules differ dramatically from one algorithm to another.
3. Defining the parameters for Opentuner, or any tuner for that matter, is rather complicated. These schedule do not lend themselves particularly well to being fully described.

4.3 Defining the Search Space in Opentuner

Considering the difficulties described above, how the search space was defined in Opentuner was really the crux of the problem. With an improper description of the search space it would be impossible for us to adhere to all the goals that we had set for ourselves. After some discussion we settled upon a series of transformations as the description of the search space.

The reason for this choice is that Opentuner needs the description to have a constant number of parameters. Furthermore, the encoding of the space needed to contain all possible schedules in this set of fixed parameters. In the search space each transformation is described by a MetaParameter. Please refer to Figure 4-1 for an example.

4.4 Integration

The glue that combines Opentuner, Halide and the Tree-Node representation is HalideScheduleTuner. HalideScheduleTuner performs a series of tasks to make sure HalideTuner keeps running smoothly. It deals with the initial booting of HalideTuner (more on that later), and it handles removing bad schedules and getting the run times for good ones.
4.4.1 Argument Parsing

The arguments passed into HalideTuner serve a plethora of uses. The most important of which are some basic information on the algorithm you are tuning, how to compile and run the program containing the algorithm, setting up Opentuner to run properly with HalideTuner and much more. Unfortunately, there are too many arguments to reasonably describe here. The full list of arguments can be found by running python HalideScheduleTuner.py --help.

4.4.2 Booting up Opentuner

Once all the proper arguments have been passed into HalideTuner, the booting up, interfacing and informing Opentuner of all the information it needs begins. Below is a basic list of the things HalideTuner does, in Figure 4-2 there is a more graphical representation of the same information.

1. HalideTuner makes sure that the supplied call graph is correct\(^1\). This call graph contains all the functions that Opentuner can manipulate. For each function there is a list of dependencies and a list of arguments the function takes\(^2\).

\(^1\)In code this is called deps.json
\(^2\)Eventually it will also have a list of reduction steps.
Figure 4-2: This graph shows the initial boot of HalideTuner and Opentuner.

2. Using this call graph and the arguments passed, HalideTuner describes the search space to Opentuner\(^3\). The search space is a list of transformations\(^4\), which is called a configuration.

3. It generates the base-schedule\(^5\).

4.4.3 Converting Configurations to Results

Once both HalideTuner and Opentuner are properly up and running the work of iterating through thousands of schedules begins. This process involves converting the configurations generated by Opentuner into solid runtimes, that Opentuner then uses to generate better configurations. In a configuration, each transformation needs to be applied to generate a new schedule. The transformations are applied consecutively starting with the base-schedule as seed. After all these transformations are applied we have a tree-node representation of the new schedule. This tree-node representation

---

\(^3\)Please refer to Appendix B.2 for more info.

\(^4\)The default number of transformations is 100.

\(^5\)More about the base-schedule in Appendix B.3.
is first verified then converted into C++ so that it can be injected into the original source code of the algorithm. This new source is then compiled, linked, ran so that the runtime of the algorithm can be reported to Opentuner. Figure 4-3 displays the information that is passed between Opentuner and HalideTuner during a typical iteration, for more precise description of this process one can look at the run-cfg method in the source code.

4.5 Schedule Verification

The schedule verifier plays a pivotal role in HalideTuner. Its work can be separated into two parts: the first is verifying that a particular transformation can be applied, and the second part is verifying that a schedule is valid. Verifying that a transformation can be applied is an extremely important role in the HalideTuner project, because without it essentially all series of transformations would generate nonsensical schedules. The basic logic of the tree-node transformations moves a schedule from one valid state to another, however, most transformations are not valid at any given time. This
verification step allows greater flexibility to Opentuner when it generates schedules, since it does not need to worry about generating invalid schedules. Furthermore, it is not detrimental to Opentuner because it slowly learns which transformations have affect and which do not.

Even though it's theoretically impossible for a transformation to make an invalid schedule, in practice it happens and so we need to verify a schedule before sending it to be compiled and run. This is caused by the non-perfect transition from theoretical to actual. Some of these are limitations in the machine, like amount of memory, some of these are limitations of the specific image that is being used, Halide cannot split an image with 1000 pixel into 2000 different variables, and some of it are bugs in HalideTuner or even potentially in Halide. Below are a few lists of potential errors the schedule verifier might raise.

The first category is general invalid trees. General invalid trees are trees whose structure is incomplete from tree level.

1. Child and parent do not point to each other.
2. A function is missing in schedule.
3. The split, fuse or reorder use arguments that do not exist.

The second category is invalid tree schedules. These are trees that are not correct according to Halide guidelines.

1. Vectorized or unrolled loop has a child.
2. Vectorize or unrolled loop does not have a constant extent.
3. A caller is computed before its dependency is computed.
4. A StoreNode is in an invalid location.
5. Missing store or compute node.
6. A node is not fully inlined or non-inlined.
7. Order of inlined nodes is invalid.

The third and final category is project based invalid tree schedules. These fall into trees that are incorrect because of errors generated by the specific image you are processing or the specific system you are running on.
1. Parallel LoopNode bisects a StoreNode and a ComputeNode causing possible race conditions.

2. Split extents are not monotonically decreasing.

3. Running out of memory.

4.6 Output

After Opentuner has run for the allocated time, HalideTuner saves the best schedule that was found to a C++ file. This allowing final product can then be used.
Chapter 5

Related Works

Before HalideTuner there have been two main Halide tuning solutions. The first, can still be found in the Opentuner repository. The second, was created by the Halide group but has long since deprecated.

5.1 Previous Opentuner Solution

The previous Opentuner solution [4] was purposefully geared towards certain types of solutions. This meant that it would only find fast solutions for problems that it was built for, not all schedules. For example, this solution could not parallelize more than one node per function. Usually this is enough since its uncommon to parallelize two nodes in one function, however, this does limit the search space and there is no way easy way of ever really knowing if schedules could benefit from having more nodes parallelized. This solution could also never fuse variables and splitting variables was restricted. These assumptions were infused in the parameters and so even discovering all the limitations and assumptions that were made in this tuner is non-trivial. And considering our goal of making sure it was easy to reason about it was decided to not continue this project, but start a different approach.
5.2 Built-in Halide Schedule Tuner

The built-in Halide schedule tuner was removed from the Halide repo on May 23rd, 2013 [5]. Given the content of the commit message, it is reasonable to assume that as the tuner was removed because it was not worth the time to maintain support for the tuner as Halide evolved. Reviving this tuner was a considerable amount of work, since the tuner did not work properly when it was deprecated, and on top of that Halide has changed dramatically since mid-2013. Unfortunatly, in the repository there is little information on the effectiveness of the tuner, so it was hard to understand how well it worked at its peak, and so it was decided to take the less risky option of generating our own tuner.

5.3 How HalideTuner addresses these issues

- HalideTuner can theoretically generate any schedule.
- HalideTuner is very pliable and dynamic in what strategies it uses to find new schedules, allowing flexibility in which schedules it generates.
- HalideTuner runs very few schedules which are invalid, which maximizes the time spent running valid schedules\(^1\).
- HalideTuner schedule verifications and schedule transformations are rather easy to read, making itself easy to be checked and improved.
- HalideTuner is only really tied to the API on which Halide works, which we expect to not change too dramatically and be backwards compatible.

---

\(^1\)Running invalid schedules will be further reduced when more bugs are ironed out.
Chapter 6

Halide Techniques

Opentuner allows the user to define techniques on how to improve Halide schedules. Techniques are the core of learning that Opentuner provides. Each technique generates different schedules. Opentuner uses previous schedules and runtimes as bases on which the techniques improves on and generates new and better schedules. Opentuner also intelligently switches between techniques. This allows the programmer to be liberal in adding techniques without being afraid of affecting performance. Most Halide technique should try to solve this problem: if the schedule you have is currently the best, what would you do to make it even faster? Techniques can have a variety of goals. There are some techniques whose sole purpose is discovery of different schedules, which are later improved upon to make the new best.

6.1 Generating Specific Techniques for HalideTuner

To facilitate future building HalideTechniques, we created HalideTechniqueBase which comes with a couple of useful methods. There a couple of common methods and some bookkeeping tasks that most HalideTechniques require. This allows a developer to create new techniques without having to worry about minute tid-bits while allowing for the flexibility of a more advanced developer to change it completely if s/he desired. To create a new HalideTechnique there are a couple of methods a developer should know.
6.1.1 Main-Generator()

Every technique needs to have a main-generator, HalideTechniqueBase has a default main-generator. It starts by calling configs-to-expand(). This method returns a series of configurations. These configurations need to be tested to make sure they have not already been used. If they have not, they get processed into a schedule object, and finally gets passed to the logic part of the technique. While doing all these things, the main-generator also handles the more tedious bookkeeping of: compressing and generating a schedule which represents the configurations, and properly initializing new configurations before passing them to Opentuner.

6.1.2 Config-to-Expand-Gen()

The config-to-expand-gen, is the method which creates new configurations in the main-generator. It provides a series of guarantees. This method returns 30 configurations. This will include both the best configuration so far, and 29 of the most recent configurations that were good\(^1\). If less than 30 good configurations were found, it pads this result with random configurations. The method of selection was half experimental and half random. It needed to improve on the best configuration, but also not fall into a rut where it would be able to find different solutions. However, testing against 29 completely random configurations was rather wasteful, since many of these configurations were very slow. Instead, it ran against something whose slowness was bounded, this gave a better guarantees that HalideTuner is not wasting its time.

6.1.3 Sub-Generator()

In HalideTechniqueBase, this method has no implementation and returns a NotImplementedException(). It is here where differing techniques should define what they are doing. The arguments passed into this method have been copied so there is no fear that this method has side-effects.

---

\(^1\)The term good refers to configurations that are at least half as fast as the best configuration.
6.1.4 Gen-New-Config()

There are two data-structures that we have to keep track of: the configuration and the HalideSchedule object. The configuration needs to be passed to Opentuner. While the HalideSchedule is not necessary, it is highly useful since it contains information and meta-data about the configuration. Gen-new-config makes sure that these two structures are synchronized before passing the information to Opentuner.\(^2\)

6.2 VectorizeSearch and ParallelizeSearch

The two most effective techniques which were created using the technique Generator are VectorizeSearch and ParallelizeSearch. VectorizeSearch simply vectorizes the innermost LoopNode of each function. This is a simple strategy that often gives noticeable speedup with little cost. So what does this mean for our result? This means that every once in a while, Opentuner will try and vectorize the best schedule it has found so far. Parallelize is similar in that it Parallelizes the outermost LoopNode of each function.

The reason HalideTuner benefits so much from these two techniques is two fold. First, most Halide schedules benefit from having the inner node vectorized and the outer node paralleled, and second that vectorizing or paralleling a node requires the nodes to be in a specific state. To vectorize a node the node must have 'constant extent' which essentially means that the inner node must be split. While Opentuner would eventually discover this, it helps significantly improve the speed at which these transformations are done and allows Opentuner to focus more on other parts of the schedule.

\(^2\)Appendix C also contains valuable information about keeping these two data-structures synchronized.
Chapter 7

Results

I tested 4 different Halide algorithms: blur, wavelet, bilateral-grid, and local-laplacian. For each of these algorithms were run on Lanka for 24 hours, 30 distinct times.

7.1 Overview

Of the 4 schedules tested, HalidTuner was able to beat the hand optimized schedules in 2 of the algorithms. However, we are certain that this lack of results in the other two schedules can be fixed by creating and improving new HalideTechniques. Considering the fact that HalideTuner was supplied only the most basic information, it is impressive that it has achieved faster runtimes than hand-optimized in the simpler schedules, and within 2x of the more complicated schedules.

7.2 Blur and Wavelet

Blur and wavelet fall are the two simpler schedules that HalideTuner beat. The first thing you might notice from the Graphs in Figure 7-1 and Figure 7-2 is how jagged the HalideTuner runtime is. The reason for this is that the graphs show you the best result found so far. So when a better result is found there is an immediate drop in

Please note that the source for all these schedules are in the Halide repository.

Please refer to appendix D for more information on the testing machine.

Graphs are also reproduced in Appendix E for easier comparison.
runtime. While it would have been more precise to display this information as data points, displaying it in this format is easier to understand.

**Blur**

The blur graph in Figure 7-1 is really an ideal graph. The improvements happen with an exponential decay like frequency: quick and large at the beginning, and then slow and small as it approaches a limit (the theoretically fastest schedule). This shows that HalideTuner is running efficiently and effectively. It first makes the large improvements, and then when there are no more large improvements to be made it starts working on finer solutions and these solutions take longer to come by. Seeing such a stretch of time with no improvement (8-24 hrs) should lead us to believe that HalideTuner has no more improvements to make.

**Wavelet**

Wavelet's graph in Figure 7-2 is good, but not as good as blur's. The improvements do not follow the exponential decay like curve of blur's improvements. There is also a 4 hour gap (starting just before the 5 hour mark) of no improvement followed by a huge improvement. This shows that while HalideTuner was finally able to find that
schedule, it took a considerable amount of time to find that improvement. While this is not a huge deal, it does show that the efficiency of HalideTuner can still be improved.

7.3 Bilateral Grid and Local Laplacian

Bilateral grid and local laplacian are the two schedules that did not beat the hand-optimized versions. These two algorithms are considerably more complicated than the previous two. Not only are their schedules more complex, but also their search spaces are dramatically larger. Search space size is based on the number of non-pointwise functions in the algorithm: blur has 2, wavelet has 3, bilateral grid has 7, and local laplacian has over 50.

Bilateral Grid

We will notice that in the bilateral grid’s graph in Figure 7-3 the HalideTuner schedule seems to have plateaued before beating the hand optimized schedule. It is important to notice the difference between this plateau and other plateaus. This plateau does not gradually occur it appears suddenly after an initial improvement. Furthermore,
it continuous to have minor improvements, of roughly the same size, throughout
the entire runtime of the program. If it was approaching the theoretical limit, we
would expect to see these improvements shrink in size. Furthermore, there are a fair
number of improvements even towards the end of trial. This leads us to believe that
if the trial had continued, there would have been more improvements. The total rate
of improvement in these algorithms is the top priority for this project, while being
within 2x of hand-optimized that is still not good enough.

Local Laplacian

The defects in local laplacian’s graph in Figure 7-4 bears striking similarities to that
of the bilateral grid graph: slow and small improvements throughout the trial. Again
this points to a lack in new schedule discovery, which can be fixed by building better
HalideTechniques. Considering how much larger the search space is for local laplacian,
it is impressive that its runtime did not preform worse when compared to bilateral
grid runtime.
7.4 A Call to Arms

In general, these graphs show that while HalideTuner is very capable, it is not yet perfected. There is still some work to be done in the HalideTechnique stage. Hopefully, this serves as a call to arms. As you can see, with the right techniques HalideTuner is able to perform and beat the hand optimized schedules with very little expertise or knowledge from the user. My hope is that Halide schedule experts will be excited by this project and create and improve the Halide techniques.
Chapter 8

Future Work

As mentioned before, HalideTuner is to be considered in its Alpha stages. The most important improvement is improving the techniques. After this improvements is done, there exist a list of useful side projects that could significantly improve HalideTuner.

The first improvement is that of adding update functions to the schedule tree. As of right now, HalideTuner completely ignores Halide update functions. While they are very similar to normal functions, there are a couple of extra checks that need to be added. This will allow HalideTuner to also tune update schedules, which will be necessary to come up with the best possible schedule.

The second such improvement would be to diversify the input images. HalideTuner currently uses a black image as an input image to tune against. I can imagine there exist algorithms in which this might lead to over-tuning. This improvement could grow into a suite of images that HalideTuner could tune against.

The third improvement is a series of small performance improvements. Currently, most tree transformations must traverse the entire tree. As the trees get longer, these operations get slower and slower\(^1\). The base-schedule also gets cloned at least twice per run, so there are some improvements to be made there.

\(^1\)So far, this has only been noticed in the schedule validation methods which take up to 0.3 seconds per validation.
Chapter 9

Conclusion

HalideTuner is meant to be your one stop shop for auto-generated schedules. It has high flexibility and has a reasonable set of defaults to allow both novice and advanced users to quickly start using it to generate fast schedules for their Halide algorithms. As Halide continues its explosive growth, the need for this software will be in high demand both for beginners who need help generating their first schedules, and for advanced users who want to use every single cycle of their CPUs to get the fastest possible Halide algorithm. While this project is still in its Alpha stages, the results are positive in showing that HalideTuner will soon be able to beat most, if not all hand optimized schedules. This cuts down time developers spend on Halide schedules and allow them to focus their time on the rest of their projects.
Appendix A

Tree Schedules for Halide Users

For people who are more familiar with Halide, you might find the chapter on Halide schedules a bit disorienting. I did not think it was worth the time of the reader to understand the difference between the tree schedule and the proper C++ Halide schedule. For Halide users however, the difference is a lot simpler to describe. C++ Halide schedules are basically a declarative language. The Tree Schedule on the other hand, is a procedural language. For example, instead of calling `store_at(var)`, you have to either raise or lower the StoreNode to the proper var (aka LoopNode).

To help your understanding further, it is probably beneficial for you to look over MetaSchedule.py (also described more in Chapter C) and CPPPrinter.py. These separate modules respectively take a C++ Halide Schedule and convert it into a Tree Schedule, and take a Tree Schedule and convert it into a C++ schedule. Hopefully, between the Halide Chapter and these two modules you will have a basic understanding of the slight difference between the two types of schedules. In Figures A-1 and A-2 you can see a C++ schedule and a tree-schedule side by side, which should help you understand some of the basics of how these trees work.
Func producer("producer_root"),
consumer("consumer_root");
    producer(x, y) = sqrt(x * y);
    consumer(x, y) = (producer(x, y) +
        producer(x, y+1) +
        producer(x+1, y) +
        producer(x+1, y+1));

//Schedule
producer.compute_root();

Figure A-1: This is a basic example of Halide schedule as a Tree-Schedule tree

producer(x, y) = sqrt(x * y);
consumer(x, y) = (producer(x, y) +
    producer(x, y+1) +
    producer(x+1, y) +
    producer(x+1, y+1));
Var yo, yi;
consumer.split(y, yo, yi, 16);
consumer.parallel(yo);
consumer.vectorize(x, 4);
producer.store_at(consumer, yo);
producer.compute_at(consumer, yi);
producer.vectorize(x, 4);

Figure A-2: This is a more complex example of Halide schedule a Tree-Schedule
Appendix B

Opentuner Search Space

B.1 Transformation Tuple

As I mentioned earlier, I created a MetaParameter in the Opentuner search space. What this means is that each MetaParameter is a specific set of Opentuner Parameters. While my code treats these as one unit, Opentuner considers these separate entities.

For reference, I call this MetaParameter a Transformation Tuple, since it is a tuple that describes one transformation. Each tuple looks like so:

```
EnumParameter(T-transform_FORMAT.format(slot_i), self.transform_types)
EnumParameter(FUNC_FORMAT.format(slot_i), funcs)

# if the transform is a split, choose the extent to split
# by (from 2 to max_extent)
IntegerParameter(EXTENT_FORMAT.format(slot_i), 2, self.max_extent)

# "CHANGE_LOOP", choose the target loop type.
IntegerParameter(LOOP_TYPE_FORMAT.format(slot_i), 0,
               len(HalideSchedule.LOOPTYPES) - 1)

# num_args_per_slot is the maximum number of nodes you can
# reference in a reorder transformation (must be at least
two, since a reorder of 1 node makes no sense).

max_arg_value is the maximum depth of a loop nest (and
equivalently, the maximum dimension of a function).

```python
for x in range(self.num_args_per_slot):
    # parameters can go negative here because the numbers are modded.
    IntegerParameter(VAR_FORMAT.format(slot_i, x), -self.max_arg_val,
                       self.max_arg_val)
```

## B.2 Manipulator

The manipulator serves to define HalideTuner's entire search space so that Opentuner can understand it and research through it. In HalideTuner, the search space is a predefined number of Transformation Tuples. Each Transformation Tuple is given a number such that the order of application is known.

This is codified in `HalideScheduleTuner::manipulator()`, essentially Opentuner defines a huge set of Transformation Tuples (whose code is above).

## B.3 Base Schedule

The base-schedule is the schedule which HalideTuner applies transformations to. Without this, HalideTuner transformations would not make sense. This is similar to a default schedule. However, the word default is not quite right because it implies that HalideTuner would always run the default first, where in reality, it would be surprising if the base-schedule was run at all. For this to happen, HalideTuner would have to generate a configuration of 100 no-ops.

This base-schedule was constructed to lend itself well to being transformed. The exact code can be seen in `HalideSchedule.py::Schedule::generate_default_schedule()`.

For people who are more familiar with Halide, the schedule is essentially a store-
root for all non-pointwise functions, and a compute-inline for all pointwise functions. But unless you disable it, pointwise functions are removed from the search space since 99 percent of the time, they should be inlined or they just clutter the search space with useless options.
Appendix C

Meta Schedule

The Meta-Schedule has two purposes. Its first purpose is to lock configurations and HalideSchedules together. Right now, when you modify schedules or configurations, the other data-structure is not modified. This means that if people are not careful, it can lead to an inconsistent state. The purpose of Meta-Schedule is to define one object which can be modified and keep track of the changes both to the configuration and to the HalideSchedule.

The second purpose of the Meta-Schedule is to give Halide users a completely Halide-like interface to create tree schedules. This will further reduce the learning curve of HalideTuner. This also further facilitates generating HalideTechniques because users can make techniques which look like HalideSchedules.

Unfortunately, Meta-Schedule is currently for illustrative purposes only because it has not yet been tested and there are still bugs that need to be ironed out.
Appendix D

Lanka

Lanka is a 24-node Intel Xeon E5-2695 v2 @ 2.40GHz Infiniband cluster with two sockets per node, each with 12 cores, for a total of 576 cores and a theoretical peak computational rate of 11059 GFlop/s. Each node has 128GB of memory, of which 120GB is safely usable.
Appendix E

Graphs of results

The plots displayed in this Appendix represent the average of the fastest time found so far against the amount of time the tuner has been running. Each algorithm has been averaged over 30 runs. The algorithms data is also divided into two plots. One between 0 and 1 hour and the second between 1 hour and 24 hours. The reason for this is that the scale of runtimes is too large to plot a useful graph. Even now some of the most extreme outliers that occurred during the first 2 minutes are not displayed.
E.1 Bilateral Grid

Figure E-1: Bilateral Grid from 0 to 1 hour

Figure E-2: Bilateral Grid from 1 to 24 hours
E.2  Blur

Figure E-3: Blur from 0 to 1 hour

Figure E-4: Blur from 1 to 24 hours
E.3 Laplacian

Figure E-5: Local Laplacian from 0 to 1 hour

Figure E-6: Local Laplacian from 1 to 24 hours
E.4 Wavelet

Figure E-7: Wavelet from 0 to 1 hour

Figure E-8: Wavelet from 1 to 24 hours
Bibliography


[4] https://github.com/jansel/opentuner/tree/f1643181e14c73cb100cf6e6d19284ba531b84fc/examples/ Link to Opentuner’s Halide tuning example

[5] https://github.com/halide/Halide/tree/3b54fcd2c104aa65b39a34e45ba3edcd52cc9f20 Link to last commit containing built-in Halide autotuner