MASTER

Synchronous dataflow models for Halide programming environment

Zhou, H.

Award date: 2016
Synchronous dataflow models for Halide programming environment

Master Thesis

Hanrong Zhou

Supervisors:
Dip. Goswami
Amir R. B. Behrouzian
Hadi Alizadeh Ara

Eindhoven, August 2016
Acknowledgement

These few months I spent as a graduate student have been sometimes challenging but very interesting and satisfying when I did my master project. During the project I received help and support from several people. I would like to thank my supervisor, Dip. Goswami, for his helpful guidance and encouragement. My gratitude goes to Ph.D. students, Amir R. B. Behrouzian and Hadi Alizadeh Ara, for their prompt answers to my questions and the advice they provided. A particular mention goes to my classmate, Sander Vocke, for his help to fix some tiny problems about the tools I used in the project.

I would also like to acknowledge all my roommates, Da Niu, Yujie Wang and Feng Liu, for their encouragement, their sincere friendship and the superb laughs.

Finally, I am very grateful to my parents for their constant support and endorsement throughout all these years.
Abstract

Image processing algorithms are time-consuming in general. In domains like healthcare, image processing algorithms are required to run in real-time. Efficient image processing is very important to assure the correct functionality of such machines. With the advent of multi-core technology, there have been technologies to perform the processing in parallel in different cores to reduce the computation time and improve efficiency. The crucial question here is scheduling decisions to map the processing stages to cores that optimises the efficiency of the image processing. Halide programming environment is a state-of-the-art technology to perform scheduling of image processing tasks to the processing cores. This thesis focuses on analyzing the performance (i.e., throughput) of the schedules of Halide programs. Dataflow is a well-known formalism for throughput analysis of imaging pipelines. Performance analysis of obtained models is much faster than doing the same analysis on the real platform. Utilizing such model-based analysis speeds up the exploration of the optimal schedules.

In this work, we analytically obtain dataflow models from Halide programs and generate dataflow models automatically when input image size or schedule changes. We analyse the performance of Halide programs with different schedules by using dataflow models and obtain dataflow models for two-stage pipelined algorithms. We show that the performance analysed by dataflow models is close to reality. We measure the performance of different schedules on the real platform and obtain the same results from dataflow analysis when comparing the relative performance of schedules.
# List of Figures

1.1 Interventional X-ray machine ........................................ 1  
1.2 The state-of-the-art approach of design .......................... 2  
1.3 Model-based analysis .................................................. 2  
2.1 An example of Halide program ...................................... 6  
2.2 Computation Order(1) .................................................. 6  
2.3 Computation Order(2) .................................................. 7  
2.4 Schedule method (1) .................................................... 7  
2.5 Schedule method (2) .................................................... 8  
2.6 An example of SDF(1) .................................................. 9  
2.7 An example of SDF(2) .................................................. 9  
2.8 State graph of Figure 2.7 .............................................. 10  
2.9 Screen shot of the output from $SDF^3$ ............................ 10  
3.1 Algorithm ................................................................. 12  
3.2 Process of calculating first line in consumer ..................... 13  
3.3 Process of schedule method two .................................... 15  
3.4 Process of schedule method three .................................. 17  
3.5 Process of schedule method four ................................... 19  
3.6 Input image divided into tiles with tile size $2 \times 2$ ................. 21  
3.7 Input image divided into tiles with tile size $3 \times 3$ .................. 21  
3.8 SDF model of schedule method one ................................. 22  
3.9 SDF model of schedule method two ................................ 23  
3.10 SDF model of schedule method three .............................. 24  
3.11 SDF model of schedule method four .............................. 25  
3.12 SDF model of schedule method five ............................... 27  
3.13 Trace of schedule method two ..................................... 28  
4.1 Automate analysis of throughput .................................... 30  
4.2 Automate analysis of run-time ..................................... 31  
4.3 Run-time of Load ....................................................... 31  
4.4 Run-time of Save ....................................................... 32  
4.5 Run-time of Run ........................................................ 32  
4.6 Run-time of Run calculated in SDF ................................. 33  
4.7 Run-time of Run measured in Halide for window size $2 \times 2$ ..... 34  
4.8 Run-time of Run calculated in SDF for window size $2 \times 2$ ..... 34  
4.9 Run-time of Run measured in Halide for window size $5 \times 5$ ..... 36  
4.10 Run-time of Run calculated in SDF for window size $5 \times 5$ ..... 36  
4.11 Run-time of Run measured in Halide for window size $10 \times 10$ .. 37  
4.12 Run-time of Run calculated in SDF for window size $10 \times 10$ .. 37
List of Tables

4.1  Comparison of analysing time in models and real run time . . . . . . . . . . . . . . . . . . . . . 39
Chapter 1

Introduction

1.1 Background

For most of the history of medicine, physicians relied on their senses, primarily vision and touch, to diagnose illness, monitor a patient’s condition, and perform invasive procedure. During the last few decades, various 3D medical imaging techniques, such as CT, MRI, and ultrasound, have become available that allow a physician to see and diagnose disease that is hidden from normal view. 3D biomedical images are now being used not only for diagnosis, but for planning and conducting treatment strategies and surgeries. This technology is referred to image-guided interventions.

For example, interventional X-ray uses image-guided interventions (see Figure 1.1). In this thesis, we focus on implementation of low-latency image processing pipelines.

![Interventional X-ray machine](image)

Figure 1.1: Interventional X-ray machine

1.2 Motivation

The image processing pipelines are given in Halide environment. Halide is a state-of-the-art programming language designed for easier coding and optimization of high-performance image processing code on modern machines. Halide program always consists of two main parts. One part is called the Algorithm part, which is the specific algorithm in the image processing program. The other part is called the Schedule part, which tells compiler when to proceed each stage in the algorithm, or whether the program will be paralleled to proceed. Usually, the algorithm of one Halide program is given while the schedule can be optimized, and different scheduling methods

Synchronous dataflow models for Halide programming environment
leads to different performance such as the run-time. Our project will focus on trying to find the image processing program with less run-time which are realized using Halide.

In the state-of-the-art approaches, when we run Halide program, the Halide code generator uses the Halide algorithm and schedule to generate equivalent C code and run the code. And then we measure the time to see which image processing method runs fast. If we don’t get the optimal schedule, then the optimizer modifies the schedules to find better ones.

![Figure 1.2: The state-of-the-art approach of design](image1)

In our project, we will use a model of computation to model the Halide program and compute the run-time based on the model, because using model-based approaches, the overall design process becomes efficient.

![Figure 1.3: Model-based analysis](image2)

We use the model-based analysis for comparison of run-time of a number of image processing approaches. For example, there are two image processing methods, Method 1 and Method 2. We will choose the one with less run-time. First we generate models for both of them and calculate the run-time by using models. We assume the run-time of Method 1 and Method 2 from the models are $t_1$ and $t_2$ respectively while the real run-time are $T_1$ and $T_2$. If $t_1$ is more than $t_2$, then we can say $T_1$ is also more than $T_2$. In this way, we can find the image processing method with least run-time.
1.3 Problem statement

In this thesis, we aim to analyse performance such as run-time of a Halide implementation, i.e., algorithm and schedule, of an image processing pipeline. We plan to analyse various scheduling options and obtain the optimal schedule in terms of run-time in an efficient fashion. We use a modelling theory to generate models automatically and do performance analysis between various schedules and then choose the optimal one.

1.4 Contributions

Our project targets to automatically generate SDF model for some useful functionalities in Halide and analyse program’s performance using dataflow theory, so the contribution of our project is mainly about transforming Halide program to SDF model automatically, validating the SDF model by visualization and choose optimal schedule method according to model-based analysis.
Chapter 2

Related work and literature review

2.1 Related work

Recently a number of efforts to automatically generate efficient image processing pipelines from high-level programs have been proposed.

Auto-tuning guided by genetic search to automatically generate Halide schedules was proposed by Ragan-Kelley. Auto-tuning is competitive with hand-tuned implementations in performance. But this method required a day or more to find optimal quality schedules. Later a more general Halide auto-tuner was implemented with the OpenTuner framework and this system was able to find efficient schedules for simpler pipelines. But for more complex pipelines, OpenTuner requires five to ten times slower than hand-tuned implementations [5].

There are some other recent efforts that achieved high performance by limiting the space of image processing programs. For example, Darkroom limits pipelines to contain only fixed-size stencil operations and no resampling. Darkroom adopts a line-buffered scheduling strategy that is ideal for the performance concerns of FPGA architectures [5].

PolyMage extends polyhedral analysis techniques to schedule image processing pipelines implemented in a Halide-like dataflow language. PolyMage demonstrates that good solutions lie in a subspace of schedules that consider only pipeline stage fusion and overlapped tiling of the output image [5].

2.2 Halide

Halide is a new programming language designed to make it easier to write high performance image processing code on modern machines. Its current front end is embedded in C++. The reason why using Halide is it takes advantage of memory locality and parallelism [4]. The locality here refers to the data generated in the current stage is stored temporally in the buffer and will be used as input in the next stage. Parallelism means the program distribute the data and run the program with distribution of data instead of running it sequentially. Parallelism requires to consider little data dependencies between each other. Halide has two main parts, algorithm and scheduling. We will explain them using illustrative examples.

Another reason is that experimenting with changing the schedule of image processing requires programmer to modify large parts of the algorithm if we use traditional programming languages. Halide separates the image processing algorithm from the scheduling. In Halide, changing the schedule does not require any changes in the algorithm and this allows the programmer to experiment with scheduling and find the optimal scheduling method. Here the algorithm means what
to compute during the program and the schedule refers to when and how to compute. Figure 2.1 shows a small Halide program and it is used to blur the input image. In this example, we can see clearly that the algorithm and schedule are separated from each other. The codes showing the equations of calculating blur \( x \) and blur \( y \) is the algorithm part and the codes followed these two equations is the scheduling part. We can try different scheduling strategies by modifying the schedule part of the program.

In general, an image processing program consists of some stages which are pipelined. Computation of pixels in an image can be in different orders and a Halide program computes pixels in different orders depending on the selected schedules.

When exploring the scheduling strategies, there are two aspects that need to be considered. The first one is in what order a stage should compute its values. Every stage has its own order to compute values. We list some common computation orders in the following.

The most common traverse strategy is called “scanline” order and we call this order as “Serial \( y \), Serial \( x \)”. It means we traverse the original function sequentially across \( y \) dimension and within that sequentially across \( x \) dimension. We can also transpose \( x \) and \( y \) dimension to make column-major traversal and we call this as “Serial \( x \), Serial \( y \)”. It means we traverse the function sequentially across \( x \) dimension and within that sequentially across \( y \) dimension. These two orders are explained in Figure 2.2 and the numbers inside each block in the figure refers to the order of computation.

The order “Serial \( y \), Vectorize \( x \)” still uses scanline order, but traverses \( x \) dimension vectorized by a factor. The order “Parallel \( y \), Vectorize \( x \)” distributes the scanlines with parallel threads and it also traverses \( x \) dimension vectorized by a factor. We can also split the \( x \) dimension or \( y \)
dimension to make components and one of this kind order strategy is “Split x, Split y, serial outer y, serial outer x, serial inner y, serial inner x”. It means we split the data into components and for outer components we traverse in scanline order as well as we traverse the inner components. These three orders are indicated in Figure 2.3 as following.

![Figure 2.3: Computation Order(2)](image)

The second aspect during the scheduling strategy is when and how we should compute these values. Here we also take the example in Figure 2.1. This example is a simple 2-stage blur algorithm, which computes a 3 × 3 box filter. The first stage, \( \text{blur}_x \), computes a horizontal blur of the input by averaging over a 3 × 1 window and the second stage, \( \text{blur}_y \), computes the final blur by averaging a 1 × 3 window of the output from the first stage. Now how should we compute its input in these two stages? There are two obvious methods for this example.

First, it can compute and store every required pixel in \( \text{blur}_x \) before computing any pixel in \( \text{blur}_y \). This requires large memory size to store the output in the first stage and this method shares little locality in the second stage. As shown in Figure 2.4, the size of input image is 6 × 6. When applying the algorithm, it will compute all the \( \text{blur}_x \). When all the \( \text{blur}_x \) are finished, it starts computing \( \text{blur}_y \). Here we need a larger buffer to store all the value of \( \text{blur}_x \) temporally.

![Figure 2.4: Schedule method (1)](image)

The second method is we can compute the pixels in \( \text{blur}_x \) first and immediately compute the pixels in \( \text{blur}_y \). This method shares maximum locality within two stages but it also leads to redundant computations. As shown in Figure 2.5, step (1) shows the procedure of computing the
first line of $blur_y$. In step (1), we compute all the needed $blur_x$ and immediately use them to compute the first line in $blur_y$. We just need a smaller buffer to store $blur_x$ temporally. Following steps are similar but there are redundant calculation of $blur_x$ shown by the green dots.

![Figure 2.5: Schedule method (2)](image)

### 2.3 Synchronous Data Flow

#### 2.3.1 SDF graph

A model is a mathematical artificial structure that reflects some aspects of the behavior of a real-life system. A model is an abstraction and an approximation, it is used to reason about and analyze a system. A model can also be used to derive an implementation.

Models of computation are means to describe the process of computation. They limit the freedom of expressing behavior to facilitate analysis and synthesis. They are used to reason about the implementation of a system before building it. There are many methods about models of computation and SDF (Synchronous dataflow) is chosen for the project because of its good property in expressiveness and succinctness, analysability and implementation efficiency.

SDF is used to model multiple applications running on a multiprocessor platform. This requires that every application running on the system has a predictable timing behavior which is independent of other applications running on the same system [1].

By using SDF, we can model different schedules of tasks in a program and we can easily analyze the performance metrics such as the latency of the whole program or the throughput of the final output. So we can use SDF to find our optimal schedule of the tasks with lower latency.

An example of an SDF is depicted in Figure 2.6. The nodes A, B and C are called actors, which are used to model individual tasks. Actors can also model just functions, computations or context switching and they communicate using tokens sent from one actor to another over the edges. The
edges with arrow typically model data dependencies or control dependencies between two actors. An essential property of SDF is that every time an actor fires, it consumes the same amount of tokens from its input edges and produces the same amount of tokens on its output edges. These amounts are called the rates. In an SDF, the rate at which tokens are produced on an edge may differ from the rate at which tokens are consumed from the edge [1]. For the sake of clarity in explanation, we omit the tokens in the figures when the rate is 1.

In Figure 2.6, there is a special edge in every actor that both the source and destination actor of the edge are same. This kind of edge is called self-edge. On self-edge, both the consuming rate and producing rate are usually 1 and the token number on it is also 1. Typically, self-edge is to make sure that actors will fire sequentially because it will produce 1 token on self-edge during each firing and this token will continually be consumed in the next firing when the tokens are available on all the consuming edges.

### 2.3.2 Throughput

The throughput here refers to the output we generate per time after running the program. The value of the throughput can indicate whether the program is running well. If the throughput is high, it means we can get more output within a period of time. Throughput is an essential metrics in our project.

Considering the throughput computation by hand, a self-timed execution of an SDF is used to compute the throughput of the graph. In this type of execution, an actor fires as soon as sufficient tokens are present on all its inputs. The firing ends when time has advanced with the execution time of the actor. At that moment, the actor produces tokens on all of its outputs. In order to compute the throughput, the state graph of an SDF is used, which is described by the distribution of tokens over the channels and the remaining execution time of all active actor firings. To compute the throughput, states visited during the self-timed execution are examined and a small subset is stored till a recurrent state, which always exists, is found. At that moment, all reachable states are found and the throughput can be computed from the periodic part of the state graph.
CHAPTER 2. RELATED WORK AND LITERATURE REVIEW

[1].

Figure 2.8: State graph of Figure 2.7

Figure 2.7 shows us an example of SDF. The execution time of $a_1$, $a_2$ and $a_3$ is 1, 1, and 2 respectively. Figure 2.8 shows the state graph of SDF, which is used to calculate the throughput. From the periodic part of the graph, we know that we can generate the output every 2 time units. So the throughput will be 0.5. Actually, calculating throughput by hand is only suitable for some small and easy SDF. When considering some normal SDF, it will become much more complicated if we use this method. Instead, the tool $SDF^3$ can help us to achieve the goal.

2.3.3 SDF$^3$

$SDF^3$ is a tool for performance analysis of SDF graphs. The user can control the characteristics of the graphs by specifying bounds, averages and variance on various aspects of an SDF graph. The tool implements a library offering many SDF analysis and transformation techniques as well as a function to visualize SDF graphs. It is very simple to extend the graph generation tool and integrate other techniques on top of the library. Following shows the throughput of an SDF model analysed by $SDF^3$ together with the analysing time.

Figure 2.9: Screen shot of the output from $SDF^3$

2.3.4 TRACE

The order of the actor execution obtained from the SDF model can further be visualized in a tool called TRACE. Generally, it visualizes the actor execution for the given execution time. The visualization helps to verify the properties like latency and throughput. With the help of visualization, exact bottle necks like critical path can be identified. We compare the order generated from TRACE with the order generated by Halide function to make sure both the orders are same. In this way we validate the SDF models with reality.
Chapter 3

Parameterization and SDF models of a Halide program

This chapter firstly introduces the Halide program that we analyse in the project. In this chapter, we have a brief overview of the algorithms considered in this project and put forward five schedule methods. We model all the schedules with SDF theory and we also validate the correctness of the models by trace visualization.

3.1 Algorithm study

In Halide program, algorithm is about what to calculate and which function we call to proceed. Following is the main part of our Halide program:

```cpp
using namespace Halide;
int main(int argc, char **argv) {
    Var x("x"), y("y");
    
    Func producer("producer_default"), consumer("consumer_default");
    producer(x, y)=sin(x * y);
    consumer(x, y) = { producer(x, y)+producer(x, y+1)+producer(x+1,y)+producer(x+1, y+1) }/4;
    consumer.realize(4, 4);
}
```

While the algorithm is just:

```cpp
producer(x, y)=sin(x * y)
consumer(x, y) = { producer(x, y)+producer(x, y+1)+producer(x+1,y)+producer(x+1, y+1) }/4;
```

Here the variable x and y refer to the position of each pixel. For example, if a pixel is positioned at second row and third column, then the value of x will be 3 and y will be 2. In our program, there are only two stages, producer and consumer. The first stage calls sin() function and the second stage calculates the average value of first stage with a window size 2 by 2. After the second stage is finished, we get the output image.

Now let’s have a look at the pixel number we need in width and the height of each stage. For example, we assume that the width and height of our output image is 5 and 4 respectively. Since the output image is consumer, then the pixel number in width and height of consumer is 5 and 4. But the number in width and height of producer is 6 and 5 according to the algorithm.
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

consumer will calculate the average value with a window size of 2 by 2 on producer, while the producer doesn’t have enough pixels when it comes to the boarder of the producer like the last row and last column. So we need to add an additional row and an additional column to our producer and that’s why the pixel number both in width and height are increased by 1. This is shown in Figure 3.1. The red pixels in producer are the additional pixels we add.

![Figure 3.1: Algorithm](image)

3.2 Schedule study

As we mentioned in Chapter 2, schedule means in which order and when each stage will proceed. In our original program, there is no code dealing with schedule. We need to put our own code to schedule the program. In this part, we put forward some schedule methods and parameterise all the schedules.

3.2.1 Schedule method one

This is a new schedule that we first put forward because it is easy to understand and it is also the base of other new schedules. Let’s take an example and assume that the input image size is 4 * 6. Here it means the pixel number in height is 4 and the pixel number in width is 6. Following images are also described in this way. We also assume the window size we use in stage consumer is 2 * 3.

In this schedule, our program will first calculate all the pixels in producer that will be used in stage consumer. And then after all the pixels in producer are finished, the program begins to calculate the pixels in consumer. Figure 3.2 shows the process of calculating the first line in consumer. In Figure 3.2, the red dots indicate the pixels within the window we use to calculate the average value in consumer stage. From Figure 3.2 we know that in order to calculate all the pixels in the first line of consumer, we need 4 more pixels in producer.
Considering the whole program, we need one additional row and two additional columns of pixels in the producer. In our example, the final image size of producer will be $5 \times 8$. Following is the conclusion we have and this conclusion can also be used in the other schedule methods.

- In vertical direction of producer, the number of pixels is calculated as the height of input image plus the height of window and then minus 1.

- In horizontal direction of producer, the number of pixels is calculated as the width of input image plus the width of window and then minus 1.

**Parameterization:** we assume that the input image size is $a \times b$ and the window size is $m \times n$. These variables should meet the following requirements:

$$a \geq m, b \geq n, m \geq 1, n \geq 1$$

then the size of producer is:

$$(a + m - 1) \times (b + n - 1)$$

This value is also the number of calculations in stage producer.

Similarly, the size of consumer is:

$$a \times b$$

This value is also the number of calculations in stage consumer. We use these parameters in the SDF models to analyze a given Halide program and input image. Based on the parameterization above, schedule method one can be described as following:
Algorithm 1 Schedule method one

1. **Load** input image
2. **Get** the number of rows in input image and set to \( a \) \( \triangleright \) get image size
3. **Get** the number of columns in input image and set to \( b \)
4. **Get** the number of rows in the window and set to \( m \) \( \triangleright \) get window size
5. **Get** the number of columns in the window and set to \( n \)
6. **Declare** array \( \text{producer}[a + m - 1][b + n - 1] \) \( \triangleright \) declarations
7. **Declare** array \( \text{consumer}[a][b] \)
8. **for** \( i \leftarrow 1 \) to \( a + m - 1 \) **do** \( \triangleright \) calculate all pixels in \( \text{producer} \)
9. Call \( \text{sin()} \) function with position of pixels in row \( i \) and store values in row \( i \) of \( \text{producer} \)
10. **end for**
11. **for** \( j \leftarrow 1 \) to \( a \) **do** \( \triangleright \) calculate all pixels in \( \text{consumer} \)
12. Calculate the average value within the window in \( \text{producer} \) and store values in row \( j \) of \( \text{consumer} \)
13. **end for**
14. **Save** output image

3.2.2 Schedule method two

With input image size \( a \times b \) and window size \( m \times n \), schedule method two can be described as following:

Algorithm 2 Schedule method two

1. **Load** input image
2. **Get** the number of rows in input image and set to \( a \) \( \triangleright \) get image size
3. **Get** the number of columns in input image and set to \( b \)
4. **Get** the number of rows in the window and set to \( m \) \( \triangleright \) get window size
5. **Get** the number of columns in the window and set to \( n \)
6. **Declare** array \( \text{producer}[m][b + n - 1] \) \( \triangleright \) declarations
7. **Declare** array \( \text{consumer}[a][b] \)
8. **for** \( i \leftarrow 1 \) to \( a \) **do**
9. **for** \( j \leftarrow 1 \) to \( m \) **do** \( \triangleright \) calculate pixels in \( \text{producer} \) needed for one line in \( \text{consumer} \)
10. Call \( \text{sin()} \) function with position of pixels in row \( i + j \) and store values in row \( j \) in \( \text{producer} \)
11. **end for**
12. Calculate the average value within the window in \( \text{producer} \) and store values in row \( i \) of \( \text{consumer} \)
13. **end for**
14. **Save** output image

The process of schedule method two is shown in Figure 3.3. Here we also take the example we use in schedule method one. In Figure 3.3, step (1) to step (8) shows how we calculate the pixels in the first line of \( \text{consumer} \). Step (9) to step (16) shows how we calculate the pixels in the second line of \( \text{consumer} \). The green dots in step (1) show that the first two lines in \( \text{producer} \) which are needed in the following steps are already calculated. The red dots from step (2) to step (7) show the window we take that we will calculate the average value inside the window. The green dots in
step (8) show that all the pixels in the first line of consumer are finished. The process from step (9) to step (16) is similar.

**Parameterization:** schedule method two with input image size $a \times b$ and window size $m \times n$ as following:

- Total number of calculation in producer: $a \times m \times (b + n - 1)$
- The number of calculation in producer in each iteration in order to calculate the pixels in each line of consumer: $m \times (b + n - 1)$
- The number of iterations: $a$
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

3.2.3 Schedule method three

With input image size \( a \times b \) and window size \( m \times n \), schedule method three can be described as following:

**Algorithm 3** Schedule method three

1. **Load** input image
2. **Get** the number of rows in input image and set to \( a \) \( \triangleright \) get image size
3. **Get** the number of columns in input image and set to \( b \)
4. **Get** the number of rows in the window and set to \( m \) \( \triangleright \) get window size
5. **Get** the number of columns in the window and set to \( n \)
6. **Declare** array \( \text{producer}[a + m - 1][b + n - 1] \) \( \triangleright \) declarations
7. **Declare** array \( \text{consumer}[a][b] \)
8. **for** \( i \leftarrow 1 \) to \( m - 1 \) **do**
9. Call \( sin() \) function with position of pixels in row \( i \) and store values in row \( i \) in \( \text{producer} \)
10. **end for**
11. **for** \( i \leftarrow 1 \) to \( a \) **do**
12. Call \( sin() \) function with position of pixels in row \( m - 1 + i \) in \( \text{producer} \) and store values in row \( m - 1 + i \) in \( \text{producer} \)
13. Calculate the average value within the window and store values in row \( i \) in \( \text{consumer} \)
14. **end for**
15. **Save** output image

The process of schedule method three is shown in Figure 3.4. Here we also take the example we use in schedule method one. In Figure 3.4, step (1) to step (9) shows the process of calculating the first line in \( \text{consumer} \). The green pixels of the first two lines in \( \text{producer} \) are calculated first and stored temporally. The process of calculating the average value of inside the window is described by the red dots. Step (10) to step (17) shows the process of calculating the second line in \( \text{consumer} \) and it is similar to the process of calculating the first line of \( \text{consumer} \).
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

Parameterization: schedule method three with input image size $a \times b$ and window size $m \times n$ as following:

- The number of calculation in producer we first calculate and store temporally is: $(m - 1) \times (b + n - 1)$

- The number of calculation in producer remained during the whole program is: $a \times (b + n - 1)$

- The number of iterations to calculate the consumer in the following sequence is: $a$

- The number of calculation in producer in each iteration is: $b + n - 1$

3.2.4 Schedule method four

With input image size $a \times b$ and window size $m \times n$, schedule method four can be described as following:
Algorithm 4 Schedule method four

1: **Load** input image
2: **Get** the number of rows in input image and set to $a$ \(\triangleright \) get image size
3: **Get** the number of columns in input image and set to $b$
4: **Get** the number of rows in the window and set to $m$ \(\triangleright \) get window size
5: **Get** the number of columns in the window and set to $n$
6: **Declare** array $\text{producer}[a + m - 1][b + n - 1]$ \(\triangleright \) declarations
7: **Declare** array $\text{consumer}[a][b]$
8: Call $\text{sin()}$ function with position of pixels needed to calculate first pixel in $\text{consumer}$ and store values in $\text{producer}$
9: Calculate the average value within the window size and store value in first pixel in $\text{consumer}$
10: \hspace{1em} for $i \leftarrow 1$ to $b - 1$ do \(\triangleright \) calculate the other pixels in the first line of $\text{consumer}$
11: \hspace{2em} for $j \leftarrow 1$ to $m$ do
12: \hspace{3em} Call $\text{sin()}$ function that needed for the next pixel in $\text{consumer}$ and store values in $\text{producer}$
13: \hspace{2em} end for
14: \hspace{1em} end for
15: \hspace{1em} Calculate the average value within window size and store values in $\text{consumer}$
16: \hspace{1em} for $i \leftarrow 1$ to $a - 1$ do \(\triangleright \) calculate pixels in the other lines of $\text{consumer}$
17: \hspace{2em} for $j \leftarrow 1$ to $n$ do \(\triangleright \) calculate the pixels in $\text{producer}$ needed for the first pixel in other lines of $\text{consumer}$
18: \hspace{3em} Call $\text{sin()}$ function and store values in $\text{producer}$
19: \hspace{2em} end for
20: \hspace{1em} end for
21: \hspace{1em} Calculate the average value within the window size and store value in $\text{consumer}$
22: \hspace{1em} for $k \leftarrow 1$ to $b - 1$ do \(\triangleright \) calculate the other pixels in other lines of $\text{consumer}$
23: \hspace{2em} Call $\text{sin()}$ function and store values in $\text{producer}$
24: \hspace{3em} Calculate the average value within the window size and store values in $\text{consumer}$
25: **end for**
26: **Save** output image

The process of schedule method four is shown in Figure 3.5. Here we also take the example we use in schedule method one. In Figure 3.5, the red dots in step (1) show the pixels we need in $\text{producer}$ to calculate the first pixel in $\text{consumer}$ and the green dot in step (2) shows the first pixel that we already calculated. Step (3) and step (4) show the process of calculating the second pixel in horizontal direction in $\text{consumer}$. The rest steps show the similar process of calculating the other pixels in $\text{consumer}$.

**Parameterization:** schedule method four with input image size $a \times b$ and window size $m \times n$ as following:

- The number of calculation in $\text{producer}$ we first calculate which are needed for the first pixel in $\text{consumer}$ is: $m \times n$
- The number of calculation for the remaining pixels in $\text{producer}$ which are needed to calculate the second pixel in the first line in $\text{consumer}$ is: $m$
- The number of remaining pixels needed to be calculated in the first line in $\text{consumer}$ is: $b - 1$
- From calculating the second line in $\text{consumer}$, the number of calculation which are needed for the first pixel is: $n$
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

3.2.5 Schedule method five

With input image size $a \times b$, window size $m \times n$ and tile size $p \times q$, schedule method five can be described as following:

Algorithm 5 Schedule method five

1. Load input image
2. Get the number of rows in input image and set to $a$ \(\triangleright\) get image size
3. Get the number of columns in input image and set to $b$
4. Get the number of rows in the window and set to $m$ \(\triangleright\) get window size
5. Get the number of columns in the window and set to $n$
6. Divide input image to tiles
7. Get the number of tiles
8. Assign tiles into different cores on the platform \(\triangleright\) Here there are 2 cores in the project
9. Get the number of tiles on core1 and set to $k_1$ \(\triangleright\) the value can be calculated shown in following Parameterization
10. Get the number of tiles on core2 and set to $k_2$
11. for $i \leftarrow 1$ to $k_1$ do \(\triangleright\) the following two parts are paralleled
12. Processing image with schedule method shown in Algorithm 1
13. end for
14. for $i \leftarrow 1$ to $k_2$ do
15. Processing image with schedule method shown in Algorithm 1
16. end for
17. Save output image

Here we also take the example we use in schedule method one. In our example, the input image size is $4 \times 6$. Figure 3.6 shows how the input image is divided into tiles when we define the tile size as $2 \times 2$. The red dots in the figure indicate the tiles and there are 6 tiles finally. But if we change
the tile size, then the number of tiles will be changed too. For example, in Figure 3.8, if the tile size is $3 \times 3$, then the number of tiles will be $4$.

**Parameterization**: schedule method five with input image size $a \times b$, window size $m \times n$ and tile size $p \times q$ as following:

- We assume that the number of tiles in vertical axis is $t_1$, then we have: $t_1 = \lceil a/p \rceil$
- We assume that the number of tiles in horizontal axis is $t_2$, then we have: $t_2 = \lceil b/q \rceil$
- We assume that the number of tiles in total is $t$, then we have: $t = t_1 \times t_2 = \lceil a/p \rceil \times \lceil b/q \rceil$
- In our project, our program is running on the platform with two cores. If we parallel our program on these two cores, then the operating system will assign the tiles to the two cores on average. We assume that the number of tiles running on the first core is $k_1$, the number of tiles running on the other core is $k_2$, then we have: $k_1 = \lfloor t/2 \rfloor = \lfloor \lceil a/p \rceil \times \lceil b/q \rceil \rfloor / 2 \rfloor$ and $k_2 = t - k_1 = \lceil a/p \rceil \times \lceil b/q \rceil - \lfloor \lceil a/p \rceil \times \lceil b/q \rceil \rfloor / 2 \rfloor$
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

Figure 3.6: Input image divided into tiles with tile size $2 \times 2$

Figure 3.7: Input image divided into tiles with tile size $3 \times 3$
3.3 SDF models

In this part, we will use SDF graphs to model each process of our schedule methods. We have already parameterised the schedules that we want to study. In each model, we will use actors to represent each stage in our program. We also add some more actors to describe the transfer between stages and the change between iterations in order to make our model reasonable. We also calculate necessary consuming rate, producing rate or token number in terms of the parameters. It should be noted that the execution of the individual actors are not possible to measure (we do not have the necessary infrastructure to compute the actor execution times in isolation) and the presented analysis does not fundamentally depend on the values of the execution time. For the sake of the analysis, we make assumption on the execution time.

3.3.1 Model of schedule method one

Figure 3.8 is the SDF model for schedule method one. The self edges in the model mean the corresponding actors can only fire sequentially and they have the same meaning in the other following SDF models.

![Figure 3.8: SDF model of schedule method one](image)

Following are the legends of Figure 3.8:

- **Actor Load** represents loading the input image to our Halide program.
- **Actor Overhead** represents the overhead when processing the image. It could be setting up the resource required in the program.
- **Actor Save** represents saving the output image after the image processing is done.
- **Actor A** represents calling \( \sin() \) function in **producer** stage.
- **Actor B** represents the transfer from stage **producer** to stage **consumer**.
- **Actor C** represents calculating the average value in **consumer** stage.
- **Rate \( R_1 \)** means the total number of calculation that will be proceeded in stage **producer**. So \( R_1 = (a + m - 1) \times (b + n - 1) \)
- **Rate \( R_2 \)** means when all the calculation in stage **producer** is done, the program will continue to the process in stage **consumer**. So \( R_2 = (a + m - 1) \times (b + n - 1) \)
- **Rate \( R_3 \)** means the total number of calculation that will be proceeded in stage **consumer**. So \( R_3 = a \times b \)
- **Rate \( R_4 \)** means when all the calculation in stage **consumer** is done, then the program will save the output image. So \( R_4 = a \times b \)
3.3.2 Model of schedule method two

Figure 3.9 is the SDF model for schedule method two.

Following are the legends of Figure 3.9:

- **Actor Load** represents loading the input image to our Halide program.
- **Actor Overhead** represents the overhead when processing the image. It could be setting up the resource required in the program.
- **Actor Save** represents saving the output image after the image processing is done.
- **Actor A** represents calling \( \sin() \) function in *producer* stage.
- **Actor B** represents the transfer from stage *producer* to stage *consumer*.
- **Actor C** represents calculating the average value in *consumer* stage.
- **Actor D** represents the change from current iteration to the next iteration.
- **Rate \( R_1 \)** means the total number of calculation that will be proceeded in stage *producer*. So
  \[
  R_1 = a \cdot m \cdot (b + n - 1)
  \]
- **Rate \( R_2 \)** means when all the calculation in stage *producer* within one iteration is finished, the program will continue to the process in stage *consumer*. So
  \[
  R_2 = m \cdot (b + n - 1)
  \]
- **Rate \( R_3 \)** means the total number of calculation in stage *consumer* within one iteration. So
  \[
  R_3 = b
  \]
- **Rate \( R_4 \)** means when all the calculation in stage *consumer* within one iteration is done, the program changes to the next iteration. So
  \[
  R_4 = b
  \]
- **Rate \( R_5 \)** means the pixels we have in the output image when one iteration is done. So
  \[
  R_5 = b
  \]
- **Rate \( R_6 \)** means the number of tokens on the back-edge we produce when one iteration is finished. So
  \[
  R_6 = b
  \]
- Rate $R7$ means when all the calculation in stage consumer is done, then the program will save the output image. So

$$R7 = a \times b$$

- Token number $T1$ means the initial token number on the back-edge. These tokens will be consumed by actor $A$ one by one within an iteration, so it is same as the number of calculation in stage producer in one iteration. So

$$T1 = m \times (b + n - 1)$$

### 3.3.3 Model of schedule method three

Figure 3.10 is the SDF model for schedule method three.

![Figure 3.10: SDF model of schedule method three](image)

Following are the legends of Figure 3.10:

- Actor Load represents loading the input image to our Halide program.
- Actor Overhead represents the overhead when processing the image. It could be setting up the resource required in the program.
- Actor Save represents saving the output image after the image processing is done.
- Actor $A$ represents calling $\sin()$ function in producer stage.
- Actor $B$ represents that when the program finishes all the calculation in stage producer excluding the last line that are needed to calculate the first line in consumer, the program will continue its following process.
- Actor $C$ represents calling $\sin()$ function to calculate one line in stage producer.
- Actor $D$ represents the transfer from stage producer to stage consumer.
- Actor $E$ represents calculating the average value in consumer stage.
- Actor $F$ represents the change from current iteration to the next iteration.
- Rate $R1$ means the number of calculation in stage producer excluding the last line that are needed to calculate the first line in consumer. So

$$R1 = (m - 1) \times (b + n - 1)$$

- Rate $R2$ means when the calculation in stage producer excluding the last line that are needed to calculate the first line in consumer is finished, the program will move to its’ next step. So

$$R2 = (m - 1) \times (b + n - 1)$$
• Rate $R_3$ means the number of calculation remained in producer. So

\[ R_3 = a \times (b + n - 1) \]

• Rate $R_4$ means the number of calculation in remained producer of one iteration. So

\[ R_4 = b + n - 1 \]

• Rate $R_5$ means the number of calculation in stage consumer of each iteration. So

\[ R_5 = b \]

• Rate $R_6$ means when the calculation in stage consumer of each iteration is finished, the program will move to the next iteration. So

\[ R_6 = b \]

• Rate $R_7$ means that when the calculation in stage consumer of each iteration is finished, it will produce a number of tokens on the back-edge that will be consumed in the next iteration. So

\[ R_7 = b + n - 1 \]

• Rate $R_8$ means the number of pixels already calculated in stage consumer of each iteration. So

\[ R_8 = b \]

• Rate $R_9$ means when all the calculation in stage consumer is done, then the program will save the output image. So

\[ R_9 = a \times b \]

• Token number $T_1$ means the initial token number on the back-edge. These tokens will be consumed by actor $C$ one by one within an iteration, so it is same as the number of calculation in stage producer in one iteration. So

\[ T_1 = (b + n - 1) \]

### 3.3.4 Model of schedule method four

Figure 3.11 is the SDF model for schedule method four.

![Figure 3.11: SDF model of schedule method four](image)

Following are the legends of Figure 3.11:

- Actor **Load** represents loading the input image to our Halide program.
- Actor **Overhead** represents the overhead when processing the image. It could be setting up the resource required in the program.
• Actor *Save* represents saving the output image after the image processing is done.
• Actor *A* represents calling *sin()* function in *producer* stage.
• Actor *B* represents calculating the average value in *consumer* stage.
• Actor *C* represents calling *sin()* function in *producer* stage for *m* times.
• Actor *D* represents calculating the average value in *consumer* stage.
• Actor *E* represents the change from current step to the next step.
• Actor *F* represents calling *sin()* function in *producer* stage for *n* times and calculating the average value in *consumer* stage once.
• Actor *G* represents calling *sin()* function in *producer* stage once and calculating the average value in *consumer* stage once.
• Rate *R1* means the number of calculation in *producer* that are needed to calculate the first pixel in *consumer*. So
  \[ R1 = m \times n \]
• Rate *R2* means when the pixels in *producer* needed for the first pixel in *consumer* are finished, the program will calculate the average value to get the first pixel in *consumer*. So
  \[ R2 = m \times n \]
• Rate *R3* means when the first pixel in *consumer* is done, the program will continue to calculate the other pixels in first line of *consumer*. The number of these pixels will be the number of iterations of the first circle in our SDF model. And Rate *R3* will describe the number of iterations. So
  \[ R3 = b - 1 \]
• Rate *R4* means when the first line in *consumer* is finished, the program will move to calculate the remaining pixels in *consumer*. So
  \[ R4 = b - 1 \]
• Rate *R5* means the number of iterations of the second circle in our SDF model. It is the number of lines of remaining pixels needed to be calculated in *consumer*. So
  \[ R5 = a - 1 \]
• Rate *R6* means the consuming rate when actor *F* fires. Since the consuming rate of the edge from actor *E* is 1, the value of *R6* is same as the initial token number. So
  \[ R6 = b - 1 \]
• Rate *R7* means the producing rate of actor *F*. Since actor *F* only fires once in each iteration, the value of *R7* will be the tokens produced by actor *F*. And the number of these tokens will indicate the number of firing of actor *G* since the consuming rate of actor *G* from *F* to *G* is 1. So
  \[ R7 = b - 1 \]
• Rate *R8* means the consuming rate of actor *Save*. So
  \[ R8 = (a - 1) \times (b - 1) \]
• Token *T1* means the number of initial tokens on the back-edge. So
  \[ T1 = b - 1 \]
3.3.5 Model of schedule method five

Figure 3.12 is the SDF model for schedule method five.

Following are the legends of Figure 3.12:

- **Actor Load** represents loading the input image to our Halide program.
- **Actor Overhead** represents the overhead when processing the image. It could be setting up the resource required in the program.
- **Actor Save** represents saving the output image after the image processing is done.
- **Actor A1** represents the first core in our platform receives a number of tiles which will be processed sequentially.
- **Actor B1** represents calling `sin()` in *producer* stage.
- **Actor C1** represents the transfer from *producer* stage to *consumer* stage.
- **Actor D1** represents the calculating the average value in *consumer* stage.
- **Actor E1** represents changing from the current step to the next step.
- **Rate R1** means the total number of calculation in *producer* that will be calculated on the first core. Since we have $k_1$ tiles processed on the first core, and each tile size is $p \times q$, then the value of $R1$ is
  \[
  R1 = k_1 \times (p + m - 1) \times (q + n - 1)
  \]
- **Rate R3** means the number of calculation in *producer* of one iteration. While the number of iterations is the number of tiles running on the first, so
  \[
  R3 = (p + m - 1) \times (q + n - 1)
  \]
- **Rate R5** means the number of firings of actor B1. So
  \[
  R5 = (p + m - 1) \times (q + n - 1)
  \]
CHAPTER 3. PARAMETERIZATION AND SDF MODELS OF A HALIDE PROGRAM

• Rate $R_7$ means when all the firings of actor $B_1$ are finished, then the program will change from stage producer to stage consumer. So

$$R_7 = (p + m - 1) \cdot (q + n - 1)$$

• Rate $R_9$ means the number of firings of actor $D_1$ and it is equal to the size of each tile. So

$$R_9 = p \cdot q$$

• Rate $R_{11}$ means when all the firings of actor $D_1$ are finished, then the program will finish the current iteration and go to the next iteration. So

$$R_{11} = p \cdot q$$

• Rate $R_{13}$ means when all the tiles processed by the first core are finished, then the work on the first core is done. So it is equal the number of tiles on first core, it is

$$R_{13} = k_1$$

• Actors from $A_2$ to $E_2$ have similar meaning as actors from $A_1$ to $E_1$. And the other rates, $R_2$, $R_4$, $R_6$, $R_8$, $R_{10}$, $R_{12}$ and $R_{14}$ also have the similar meaning as the previous rates. Here we will not describe them in detail.

3.4 Visualization in TRACE

In this part we use the tool TRACE to verify the trace of the process for each schedule method. Here we still use the input image with size $4 \times 6$. After using TRACE, we get the figure showing the trace of every schedule method. Here we choose the figure showing the trace of schedule method two as Figure 3.13. In the figure, we can see actor Load fires first, Overhead follows at second and Save fires in the end. Then in the middle period, there are four iterations and each actor from $A$ to $D$ will finish their firing work sequentially within each iteration. Here the length of each actor in horizontal axis indicates their execution time. Since we define the execution time per firing of each actor from $A$ to $D$ is the same, the different length means their number of firings is different. Clearly, our model matches with the TRACE based visualization of actor execution.

Figure 3.13: Trace of schedule method two

28 Synchronous dataflow models for Halide programming environment
Chapter 4

Performance analysis of Halide algorithm

This chapter introduces the performance analysis for Halide program with the five schedule methods explained in Chapter 3. Our goal is to validate the SDF model. In this way, we can use the analysis results from SDF model as input to optimizer function to obtain the schedule method with good performance.

4.1 Automate analysis of throughput

This section introduces the procedure of generating SDF model from Halide code and getting throughput by using $SDF^3$ automatically. With throughput as the input to optimizer function, we can obtain the optimal schedule method because throughput can be used to calculate the run-time as one divided by throughput. We automate the procedure by running a script. The procedure of the script can be described as following:

- Read Halide code
- Get parameters like input image size, window size or tile size
- Get the schedule method and copy the document containing the corresponding template of SDF model from a “library”. Here the “library” refers to a document containing the SDF model templates of all the five schedule methods
- Calculate parameters needed in SDF models like the consuming rates or producing rates, and modify them on the template
- Use $SDF^3$ to analyse the throughput

Figure 4.1 shows the procedure.
Here we have to mention that during the project, the throughput we get doesn’t match the reality. Because the run-time by using one divided by throughput doesn’t match the run-time we measure in Halide program. Then we need to find another method to continue the project.

4.2 Automate analysis of run-time

In this section since the throughput cannot be used, we change to analyse the run-time based on SDF models instead of the throughput. Because there are no pipeline between actors in models and actors execute sequentially, we can calculate the run-time by using formulas. We automate the procedure of generating SDF models from Halide codes and analysing performance of run-time by running a script. The procedure of the script can be described as following:

- Read Halide code
- Get parameters like input image size, window size or tile size
- Get the schedule method and copy the document containing the corresponding template of SDF model from a “library”. Here the “library” refers to a document containing the SDF model templates of all the five schedule methods
- Calculate parameters needed in SDF models like the consuming rates or producing rates, and modify them on the template
- Calculate the run-time based on SDF model

Figure 4.2 shows the procedure.

4.2.1 Run-time of loading input image

Different schedules only vary the way how we process the input image, so different schedules lead to different run-time in image processing part and they don’t change the run-time of loading and saving image. In order to prove the run-time of loading and saving image doesn’t vary due to different schedule method, we first divide our Halide program into these three parts as following and measure the run-time of loading image and saving image for every schedule method.

- **Load**: the part of loading the input image to Halide program.
CHAPTER 4. PERFORMANCE ANALYSIS OF HALIDE ALGORITHM

Figure 4.2: Automate analysis of run-time

- **Run**: the part of processing the input image based on the algorithm and schedule in Halide code.
- **Save**: the part of saving the output image.

We choose 64 input images with different sizes. Here the image size means the number of pixels inside the image. We run the Halide program with these 64 images as input and use Halide function to measure the run-time of **Load**. The result is shown in Figure 4.3.

From Figure 4.3 we can see the run-time of **Load** is the same for all the five schedule methods.

4.2.2 Run-time of saving output image

We choose the same 64 input images with different size, run the Halide program and use Halide function to measure the run-time of **Save**. The result is shown in Figure 4.4.

As we can see from Figure 4.4, the run-time of **Save** is the same for all the five schedule methods.
According to the results above, we can conclude that no matter which schedule method we use, the run-time of \textit{Load} doesn’t change for the same input image. And the run-time of \textit{Save} also doesn’t change. So we can only focus on the run-time of \textit{Run}.

### 4.2.3 Run-time of processing input image

We still choose the same 64 input images and measure the run-time of \textit{Run} for all the five schedule methods by using Halide function. The result is shown in Figure 4.5. In Figure 4.5, the run-time for the same input image changes a lot due to different schedule method. The run-time with schedule method two is the most while the run-time with schedule method five is the least. For the other three schedule methods, the run-time of schedule method one is nearly the same as schedule method three, and they are both less than the schedule method four.

Our goal is to validate that the run-time based on SDF model can match the run-time in Halide program, so we need to calculate the run-time of \textit{Run} based on SDF model.
CHAPTER 4. PERFORMANCE ANALYSIS OF HALIDE ALGORITHM

For schedule method one from Figure 3.8, we can calculate the run-time of Run as

\[ R1 \times T_A + R2 \times T_C + T_O \]

For schedule method two from Figure 3.9, it is

\[ R2 \times T_A \times a + R3 \times T_C \times a + T_O \]

For schedule method three from Figure 3.10, it is

\[ R1 \times T_A + R4 \times T_C \times a + R5 \times T_E \times a + T_O \]

For schedule method four from Figure 3.11, it is

\[ R1 \times T_A + R2 \times T_B + R3 \times (T_C + T_D) + R5 \times (T_F + R7 \times T_G) + T_O \]

For schedule method five from Figure 3.12, it is

\[ \max(T_{B1} \times R5 \times R13 + T_{D1} \times R9 \times R13 + T_O, T_{B2} \times R6 \times R14 + T_{D2} \times R10 \times R14 + T_O) \]

Here \( T_A, T_B, T_C, T_D, T_E, T_F, T_G, T_{B1}, T_{B2}, T_{D1} \) and \( T_{D2} \) refer to the execution time of actor \( A, B, C, D, E, F, G, B_1, B_2, D_1 \) and \( D_2 \) per firing respectively. Their values are all \( 10 \times 10^{-6} \) milliseconds defined in our SDF model. \( T_O \) refers to the execution time of actor \( \text{Overhead} \) and its value is \( 100 \times 10^{-6} \) milliseconds. The rates we use here of each model can be calculated as the equations shown in Chapter 3.

We calculate all the run-time of Run for the five schedule methods with the same 64 input images. The result is shown in Figure 4.6.

![Figure 4.6: Run-time of Run calculated in SDF](image)

In Figure 4.6, the results of schedule method one, schedule method three and schedule method four are almost the same. They are described by the line in the middle. From this figure, we can conclude that the run-time of schedule method two is the most for all the input images while
the run-time of schedule method five is the least. Obviously, the run-time for all the schedules is growing when the input image size is increased and the difference between schedule method two and schedule method five is also growing when we increase the input image. Although the run-time calculated based on SDF model is not the exact value measured by Halide function, the trend of run-time calculated from SDF model for every schedule method shown in Figure 4.6 really matches the trend measured by Halide function shown in Figure 4.5. That means by using the results generated by SDF model, we can predict which schedule method needs least execution time for this Halide program without measuring the real run-time by Halide function because the measurement in reality is much time consuming.

4.3 Advanced performance analysis

According to the previous experiments, schedule method five has a good performance with least run-time for all the input images we choose. The tile size in schedule method five is always $2 \times 2$ during the experiments. In the following experiments, we choose one particular input image and change the tile size. We also measure the run-time of Run by using Halide function and calculate this run-time based on SDF model. Here the size of input image is $4000 \times 4000$. The shape of the tile is a square and we change the tile size by increasing the length (the number of pixels) of side in the square from 2 to 30. In the experiments, we also change the window size in the algorithm of Halide program. They are $2 \times 2$, $5 \times 5$ and $10 \times 10$ respectively.

Figure 4.7 shows the result measured by Halide function when the window size is $2 \times 2$. In this figure, we compare the result with the run-time measured in Halide when we use schedule method one.

![Figure 4.7: Run-time of Run measured in Halide for window size 2 \times 2](image)

Figure 4.7: Run-time of Run measured in Halide for window size 2 \times 2
From Figure 4.7, the run-time when using schedule method five is always less than the time when we use schedule method one no matter which tile size it is. When we increase the tile size, the run-time decreases quickly when the tile size is less than 5. When the tile size is more than 5, the run-time decreases slightly. When the tile size is more than 17, the time doesn’t change and becomes saturated.

Figure 4.8 shows the result calculated in SDF model when the window size is $2 \times 2$. In this figure, we compare the result with the run-time calculated in SDF when we use schedule method one.

![Figure 4.8: Run-time of Run calculated in SDF for window size $2 \times 2$](image)

From Figure 4.8, the run-time when using schedule method five is also less than the time when we use schedule method one no matter which tile size it is. Before the point that the tile size is 5, the run-time decreases quickly. After this point, the run-time becomes saturated when we increase the tile size. Comparing Figure 4.7 and Figure 4.8, the result calculated based on SDF model shows the same trend of the result measured by Halide function.

Then we change the window size of the algorithm from $2 \times 2$ to $5 \times 5$. Figure 4.9 shows the result measured by Halide function when the window size is $5 \times 5$, we compare the result with run-time measured in Halide when we use schedule method one.

From Figure 4.9, the run-time when using schedule method five is more than the time using schedule method one when the tile size is less than 5. When the tile size is more than 5, the run-time using schedule method five becomes less than the time using schedule method one. Before the point that the tile size is 5, the run-time using schedule method five decreases a lot. While after this point, the time decreases slightly and becomes saturated when the tile size is more than 13.

Figure 4.10 shows the result calculated in SDF model when the window size is $5 \times 5$. In this figure, we compare the result with the run-time calculated in SDF when we use schedule method one.

From Figure 4.10, the run-time using schedule method five is more than the time using schedule method one when the tile size is less than 5. When the tile size is more than 5, the run-time using schedule method five becomes less than the time using schedule method one. Before the point that the tile size is 5, the run-time using schedule method five decreases a lot. While after this point,
the time decreases slightly and becomes saturated when the tile size is more than 13. Comparing Figure 4.9 and Figure 4.10, the result calculated based on SDF model shows the same trend of the result measured by Halide function.

Figure 4.11 shows the result measured by Halide function when the window size is 10 * 10. In this figure, we compare the result with the run-time measured in Halide when we use schedule method one.
From Figure 4.11, the run-time using schedule method five is more than the time when we use schedule method one when the tile size is less than 3. When the tile size is more than 3, the run-time using schedule method five becomes less than the time using schedule method one. Before the point that the tile size is 9, the run-time using schedule method five decreases quickly when we increase the tile size. While after the point, the time decreases slightly when the tile size is increased and the time becomes saturated when the tile size is more than 17.

Figure 4.12 shows the result calculated in SDF model when the window size is $10 \times 10$. In this figure, we compare the result with the run-time calculated in SDF when we use schedule method one.

From Figure 4.12, the run-time using schedule method five is more than the time using schedule method one when the tile size is less than 9. When the tile size is more than 9, the run-time using...
CHAPTER 4. PERFORMANCE ANALYSIS OF HALIDE ALGORITHM

Schedule method one is a bit more than the time using schedule method five. Before the point that the tile size is 9, the run-time using schedule method five decreases a lot when we increase the tile size. After the point, the time using schedule method five decreases slightly and it becomes saturated when the tile size is more than 17. Comparing Figure 4.11 and Figure 4.12, the result calculated based on SDF model shows the same trend of the result measured by Halide function. But the point of the tile size that the run-time using schedule method five becomes less than the time using schedule method one is different. In Figure 4.11, this point is the tile size of 3. While in Figure 4.12, this point is the tile size of 9.

According to the experiments above, if the window size in the algorithm is small, like $2 \times 2$, using schedule method five needs less run-time. The result calculated in SDF model can guide the designers to choose the schedule method five for the real Halide program.

If the window size is increased to $5 \times 5$, using schedule method five with small tile size needs more run-time than using schedule method one. The reason could be that when the window size becomes larger while the tile size is small, there could be more redundant calculation which increase the run-time. By increasing the tile size, the run-time using schedule method five decreases. The result calculated in SDF model can guide the designer to choose the tile size for schedule method five in order to make the run-time less.

If the window size is increased to $10 \times 10$, it is better to choose the schedule method five with the tile size more than 2 according to Figure 4.11. Similar to the case of window size $5 \times 5$, when we use schedule method five with small tile sizes, there could be more redundant calculation which leads to more run-time. The result calculated in SDF shown in Figure 4.10 can guide designers to choose schedule method five for the Halide program but cannot guide designers to choose the proper tile size for schedule method five.

4.4 Comparison of analysis time in models and real run time

Our goal is to use analysis from SDF models instead of doing the same analysis on the real platform because using such model-based analysis speeds up the exploration of optimal schedule. In the following, we choose five input images with different image size. We run the Halide program with five schedule methods in Chapter 3 and measure the run-time. Here the run-time is the sum of Load, Run and Save. We also do the analysis of throughput based on SDF model by using tool SDF$^3$ even though in our project the throughput analysis doesn’t match the reality. Here we just focus on that the procedure of analysis using SDF$^3$ also takes time and the tool SDF$^3$ also gives us this value together with the throughput just as shown in Figure 2.9

Table 4.3 shows the results of both analysis time in models and real run-time for all the five schedule methods. For each input image of every schedule method, the analysis time in models by using SDF$^3$ is much less than the real run-time in Halide program.
<table>
<thead>
<tr>
<th>Schedule method one</th>
<th>Input image size (number of pixels in million)</th>
<th>Analysing time by $SDF^3$ (ms)</th>
<th>Real run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>≤ 1</td>
<td>1483</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>≤ 1</td>
<td>5585</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>≤ 1</td>
<td>8127</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>15</td>
<td>11264</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>≤ 1</td>
<td>13955</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule method two</th>
<th>Input image size (number of pixels in million)</th>
<th>Analysing time by $SDF^3$ (ms)</th>
<th>Real run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>1000</td>
<td>1418</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>984</td>
<td>5744</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2000</td>
<td>8355</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1984</td>
<td>11426</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>2000</td>
<td>14489</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule method three</th>
<th>Input image size (number of pixels in million)</th>
<th>Analysing time by $SDF^3$ (ms)</th>
<th>Real run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>625</td>
<td>1390</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>640</td>
<td>5723</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>1265</td>
<td>8167</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>1250</td>
<td>11018</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>1250</td>
<td>13911</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule method four</th>
<th>Input image size (number of pixels in million)</th>
<th>Analysing time by $SDF^3$ (ms)</th>
<th>Real run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>218</td>
<td>1425</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>234</td>
<td>5921</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>437</td>
<td>8154</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>437</td>
<td>11108</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>437</td>
<td>14111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Schedule method five</th>
<th>Input image size (number of pixels in million)</th>
<th>Analysing time by $SDF^3$ (ms)</th>
<th>Real run time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>46</td>
<td>1374</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>31</td>
<td>5567</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>31</td>
<td>8039</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>46</td>
<td>10970</td>
</tr>
<tr>
<td></td>
<td>26</td>
<td>31</td>
<td>13924</td>
</tr>
</tbody>
</table>

Table 4.1: Comparison of analysing time in models and real run time
Chapter 5

Conclusion and future work

This chapter introduces conclusions and the future work of the project.

5.1 Conclusion

• The algorithm of the Halide program in the project only has two stages. We make actors to represent each stage and the relationship between each stage during the calculation in the algorithm. We define the values of the rates in SDF model according to different schedule methods. We make SDF model for every schedule method.

• We automate generating SDF models when the input image size or the schedule method changes. We also automate analysing the throughput using $SDF^3$ or the run-time calculated based on models.

• The visualization by using tool $SDF^3$ and TRACE validates the SDF models of all the schedule methods match the real Halide program.

• The run-time of loading image is always the same for the five schedule methods when input image doesn’t change and it is also the same case for the run-time of saving image. But the run-time of processing image is different for the five schedule methods when input image doesn’t change. The schedule method five costs least run-time.

• The run-time analysis based on SDF models can be used as input of optimizer function to obtain optimal schedule method.

5.2 Future work

• The algorithm in the project only has two stages. For the program with algorithm containing three or more stages, we need to model them using SDF theory as well.

• When we calculate the run-time based on SDF model, we define the value of execution time for all the actors because we cannot measure the exact value by using Halide function. Using the execution time we define, we can only conclude from the relative difference between different schedule method. If we can use the exact execution time, then we can get conclusion directly from the result.

• We only check the run-time of the program and analyse the performance of the program only based on run-time. We need analyse more performance metrics to evaluate Halide program.

• We need to check that why the throughput analysis by using $SDF^3$ doesn’t match the reality. The reason could be that the execution time of each actor that we define is not suitable, or there are some aspects that we don’t consider when we model the program.
• When we model the program, we don’t consider the resource limit, like the memory usage of the platform. And we don’t model the benefit of the performance about improving the locality when cache is used. If we consider more aspects into our model, then the analysis from the SDF model will be much more useful and convincible.
Bibliography


