A Distributed Backend for Halide

by

Aaron Epstein

Submitted to the Department of Electrical Engineering and Computer Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

© Massachusetts Institute of Technology 2015. All rights reserved.
A Distributed Backend for Halide

By

Aaron Epstein

Submitted to the Department of Electrical Engineering and Computer Science on

May 22, 2015

In partial fulfillment of the requirements for the Degree of Master of Science in Electrical Engineering and Computer Science.

ABSTRACT

Halide is a tool that makes writing efficient code easier to design, maintain, and operate. However, Halide is currently limited to building programs that run on one machine, which bounds both the size of the input and the speed of computation. My work involves adding distributed computation as an option for running Halide programs. Enabling distributed computation would allow Halide to be run at a much larger scale and solve larger problems like weather simulation and 3D rendering. This project has dealt with how Halide can analyze what data programs will need to share and how they can efficiently share it. Using my work, a programmer can easily divide their input into several small chunks and start an instance of their program to solve each chunk with synchronization and data sharing handled through Halide.

Thesis Supervisor: Saman Amarasinghe
Acknowledgements

Completion of this thesis would not have been possible without the generous help of multiple individuals who I would like to thank here. My thesis advisor, Saman Amarasinghe, helped me find this project and directed its course over the last year. Several coworkers and members of the Halide developer community, including Tyler Denniston, Jonathan Ragan-Kelly, Shoaib Kamil, Andrew Adams, and Zalman Stern, were instrumental in helping me work through issues. Finally, my friends and family were supportive to me throughout all of my endeavors. Thank you.
Contents

1 Introduction ........................................................................................................................................... 10
  1.1 Motivation ......................................................................................................................................... 10
  1.2 Why Halide? .................................................................................................................................... 10
  1.3 Solution Overview .......................................................................................................................... 11
  1.4 Thesis Outline ................................................................................................................................... 12

2 Halide ............................................................................................................................................. 13
  2.1 Program Structure in Halide ........................................................................................................... 13
    2.1.1 buffer_t ....................................................................................................................................... 13
    2.1.2 Functions ..................................................................................................................................... 14
  2.2 Compiling a Halide Function ........................................................................................................... 16
    2.2.1 Lowering ....................................................................................................................................... 17
    2.2.2 Compiling ..................................................................................................................................... 17

3 Message Passing Interface .................................................................................................................. 18
  3.1 Initialization ......................................................................................................................................... 18
  3.2 Communication ................................................................................................................................. 18
    3.2.1 Point-to-Point ............................................................................................................................... 18
    3.2.2 Collective ...................................................................................................................................... 19

4 Changes to the Interface to Halide ..................................................................................................... 20
  4.1 Using MPI Tools ............................................................................................................................... 20
  4.2 Inputs .................................................................................................................................................. 21
  4.3 Outputs ................................................................................................................................................ 21
  4.4 Only Images Are Distributed ........................................................................................................... 22

5 Additions to Halide ............................................................................................................................... 23
  5.1 New Image Checks ............................................................................................................................ 23
  5.2 Replacing the Buffer ........................................................................................................................ 23
  5.3 Filling the Collected Buffer .............................................................................................................. 24

6 Collecting the Distributed Image ........................................................................................................ 25
  6.1 Initialization ......................................................................................................................................... 25
6.2 Creating the Map of Needs and Local Regions .................................................. 25
6.3 Sends and Receives ......................................................................................... 26
6.4 Filling in the Buffer ....................................................................................... 26

7 Performance and Possible Optimizations ....................................................... 27
  7.1 Dense stencil ............................................................................................... 27
  7.2 Sparse Stencil .............................................................................................. 28
  7.3 Degenerate Pattern ...................................................................................... 28
  7.4 Reductions .................................................................................................. 29
  7.5 Potential Optimizations ............................................................................... 29

8 Numerical Results ........................................................................................... 31
  8.1 Experimental Setup .................................................................................... 31
    8.1.1 Shape .................................................................................................... 31
    8.1.2 Collection System ................................................................................ 31
  8.2 Experimental Results .................................................................................. 33
    8.2.1 Collect Performance ........................................................................... 34
    8.2.2 BLOCK vs CYCLIC ............................................................................ 35
  8.3 Control ......................................................................................................... 35

9 Summary of Contributions and Future Work ............................................... 36
  9.1 Contributions ............................................................................................... 36
  9.2 Future Work ................................................................................................ 36

10 References ...................................................................................................... 37

11 Appendix ........................................................................................................ 38
List of Figures

Figure 2.1 A Simple Halide program................................................................. 14
Figure 2.2 One Algorithm, Two Schedules ..................................................... 16
Figure 7.1 At the Corners, a Stencil Needs Additional Values ....................... 27
Figure 7.2 Communication Under Different Distributions .............................. 28
Figure 7.3 A Degenerate Case........................................................................... 29
Figure 8.1 Execution Time on SQUARE Regions .......................................... 33
Figure 8.2 Execution Time on SLICE Regions ............................................... 34
Figure 8.3 Single Machine Comparison.......................................................... 35
Chapter 1

1 Introduction

1.1 Motivation

For problems of a large enough scale, it is simply not feasible to use a single machine to solve the entire problem. Instead, multiple machines are coordinated to complete the task. Sometimes this can be as simple as dividing the labor between independent machines. For example, compressing a million files on a thousand machines can be done by giving each machine a thousand files to compress and collecting the results when they have finished. However, other problems, such as simulation of weather across a region, require the machines to interact and share data. Programs like this are not easy to write or maintain.

If one has the opportunity to leverage a supercomputer or computing cluster against a problem, one should try to use it efficiently. While most programs could benefit from optimization, it is particularly important that code running on large and expensive hardware not waste time. Therefore, programmers might be interested in tools that allow them to write efficient distributed programs more easily. My work has been to integrate a tool for writing cross-platform and parallel programs with a system for writing distributed programs.

1.2 Why Halide?

One tool that might be helpful for the authors of such programs is Halide. Halide is a platform for writing optimized and parallel code in a way that is easy to maintain [1]. The fundamental idea behind Halide is that the algorithm that the program is trying to accomplish should be specified separately from the organization of the computation. For instance, a Halide program for a blur has two sections, one that states that pixel values are averages of their neighbors, called the algorithm, and another that states that the
computation should be computed over x and y tiles, called the schedule. If the programmer wants to adapt the program to compute a gradient instead of a blur, only the algorithm needs to be changed to compute a difference. The optimal schedule for the blur should be the same as computing a gradient, so the schedule can remain untouched. To optimize the performance for the correct algorithm, only the schedule needs to be changed. In addition, the optimizations are mostly cross platform, so the program can be expected to run reasonably quickly, if not optimally, on different platforms. Even if a different schedule is required, a program using Halide's schedule abstraction can be tuned far more cleanly than a program that is written explicitly. [1]

I believe Halide's algorithm/schedule separation also lends itself to a clean interface for writing distributed code. Algorithms specified in Halide are very local. Since Halide already provides excellent ways to specify the granularity for operations like vectorization and parallelization, it would be elegant to use a similar system to specify the degree of distribution among different machines. For example, Halide allows the programmer to break a large loop into a smaller one over tiles. I use a similar interface to allow programmers to specify how to break storage up into tiles. In addition, Halide is already designed to support multiple runtimes on different types of hardware, which makes using a homogenous distributed system significantly easier.

1.3 Solution Overview

In order for Halide to be a viable tool for writing distributed programs, it would need to hide the explicit details of transferring data around the cluster from the programmer and accomplish those tasks automatically. To that end, I have integrated it with MPI, the Message Passing Interface, which is a platform that enables communication between processes. My code allows programmers to write programs that process a very large image, while only storing a small region of it on each machine. When Halide programs make use of data outside of their local region, the new additions will enable the Halide library to collect the necessary data from other processes.
1.4 Thesis Outline

The rest of this document is organized as follows. Chapter 2 introduces the fundamentals of the Halide library. Similarly, Chapter 3 introduces the relevant components of MPI, the Message Passing Interface. In Chapter 4, I detail how programmers can write distributed programs with the newly created interface. Chapter 5 describes the additions within the Halide library which detect distributed inputs and modify the program to use the data collection procedure. Chapter 6 contains information on the data collection procedure itself. In Chapter 7, I discuss the theoretical performance of my approach in various common and uncommon cases. In Chapter 8, I show some performance measurements on actual data. Chapter 9 summarizes my contributions and gives possible avenues for future development.
Chapter 2

2 Halide

Before detailing how Halide integrates with MPI, the Message Passing Interface, it is necessary to explain some of the basic elements of Halide’s functionality.

2.1 Program Structure in Halide

2.1.1 buffer_t

In general, Halide programs input and output "images." These are referred to as images because Halide was originally designed as an image processing library, but they are simply buffers which hold data about their size and shape. The low level structure containing these buffers is buffer_t.

A buffer_t has three arrays describing the shape of each dimension. These arrays all have length four, as Halide currently only supports four-dimensional inputs and outputs. The min array describes the top left corner of the region held by the buffer, as the smallest member of the domain in each dimension. The extent array holds the width of the buffer in each dimension, with the total size of the buffer being the product of all nonzero extents. The stride array holds, for each dimension, how far in memory one travels when moving one index in that dimension. The strides are used to calculate the linear memory address of a four-dimensional address.

A buffer_t also has a pointer to the data contained in the buffer and the size of each element. In addition, I’ve added fields to the buffer_t that describe how the data is distributed among the different processes. These fields are described in detail in Chapter 4. The buffer_t has fields to help synchronize it with a counterpart buffer on the GPU, but these fields are not relevant to my project.
A greyscale image would be represented as a `buffer_t` with two dimensions, with a `min` array containing only zeroes and an `extent` array whose first two elements contain the width and height of the image. The state of a weather simulation at any given time might be four dimensional, with the first three dimensions representing space and the fourth separating temperature and pressure into different channels.

### 2.1.2 Functions

A key feature of Halide is the separation of code into an algorithm and a schedule.

```c
Func blur_3x3(Func input) {
    Func blur_x, blur_y;
    Var x, y, xi, yi;

    // The algorithm - no storage or order
    blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
    blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;

    // The schedule - defines order, locality; implies storage
    blur_y.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    blur_x.compute_at(blur_y, x).vectorize(x, 8);

    return blur_y;
}
```

**Figure 2.1** A Simple Halide program

The algorithm describes the operations needed to transform the inputs into the output. For a blur, as in Figure 2.1, this is an averaging of neighbors. For a gradient, it is a difference of neighbors. For a histogram, this is the collection of values into bins. In all cases, the algorithm describes statements that the programmers expect to be true about the output, regardless of how it's divided or optimized.

The schedule describes how a function should be evaluated, which determines what kind of loops will iterate over the function domain. Here is where optimization is placed. In Figure 2.1, the blur is reorganized to loop over tiles to improve locality, the innermost calculations are vectorized, and one of the dimensions is parallelized.
1. The Algorithm

The algorithm section contains symbolic definitions of one or more functions. One of these functions will be the output function. The algorithm can be described as a tree with the output function at the root, with children being the functions that it calls. At the leaves are functions which are calculated without the calls to other Halide functions, although they may make use of input images. The tree of functions is called a pipeline.

Although the code in Figure 2.1 looks like C++ code that does math operations, the operators used are actually overloaded so that the math is symbolic. The operands are functions and expressions, stored as objects of type Halide::Func and Halide::Expr, and the operators join these objects into a tree that describes the algorithm. When Halide compiles the pipeline, it uses this data structure.

2. Schedule

By default, every function but the output is inlined, meaning that each value of the function is computed when a consumer requires it. Alternatively, a function may be marked as a root, so that all required values of the function are computed before any consumers are called. This default schedule produces a program that loops over the range of elements of the output function. At each iteration of this loop, it calculates the values of any producers required to find the value of that element.

More complex scheduling options allow loops to be split into multiple levels or fused with other levels. Loops can be reordered, unrolled into sequential statements, parallelized, etc. The ability to change execution so much with simple descriptors is one of the major strengths of Halide.

Here is an example of one algorithm scheduled two different ways.
Algorithm

\[ \text{producer}(x, y) = \sqrt{x \cdot y}; \]
\[ \text{consumer}(x, y) = (\text{producer}(x, y) + \text{producer}(x, y+1) + \text{producer}(x+1, y) + \text{producer}(x+1, y+1)); \]

**DEFAULT SCHEDULE**

float result[4][4];
for (int y = 0; y < 4; y++) {
    for (int x = 0; x < 4; x++) {
        result[y][x] = (sqrt(x*y) + sqrt((x+1)*y) + sqrt(x*(y+1)) + sqrt((x+1)*(y+1)));
    }
}

Figure 2.2 One Algorithm, Two Schedules

The default schedule compiles to the simple loops as described. The more advanced schedule on the right changes the storage of the producer function to eliminate some redundant work.

2.2 Compiling a Halide Function

A Halide function can be compiled in one of two ways. It can be compiled just-in-time by the program that defines it, or it can be compiled to an object by one program and then used by another. Fortunately, use of the approach to distribution described in this document is not affected by the choice of compilation method. In any case, compiling proceeds through the following steps.
2.2.1 Lowering

The input to the Halide library is the tree of functions and expressions described in the previous sections. Since these objects are also the internal representation used by the Halide library, there is no need for a lexical phase. Instead, the tree goes through an iterative process called "lowering" which transforms the program into something much easier to compile.

The basics of lowering, skipping phases that are irrelevant to this work, are as follows:

- Lowering starts by ordering the functions by their dependencies and creating storage nodes for each function that needs one.
- The bounds of each function are calculated. These are symbolic and will gain values when the function is actually called with an output. Until the function is actually used, Halide doesn’t know over what region the function will be evaluated.
- Checks are added to make sure that the functions don’t access input images outside of their bounds. These are runtime checks which use the values of the previous phase. This phase is where I determine whether an image is distributed, see Chapter 5.
- Halide determines how much memory each function will require.
- My added phase sets up a new buffer for distributed images and the call that will populate them. It also replaces all accesses to the original image with accesses to this new image, see Chapter 5.
- Halide performs optimizations and simplifications on the lowered code.
- Halide changes multi-dimensional buffer accesses into single-dimensional ones using the stride information in the buffer.
- Again, the code is simplified and optimized.

2.2.2 Compiling

Once lowering is complete, the function is still contained in the Halide internal representation, but in a way that is much closer to the representation used by LLVM, the compiler used by Halide to actually generate machine code. Code generation walks through the tree and converts the representation to a similar tree using the LLVM representation. Then, LLVM compiles the program into machine code.
3 Message Passing Interface

To support communication between processes, I chose Message Passing Interface (MPI). MPI is a standard created by the MPI Forum which provides an interface for processes running the same program to work together. [2] To use MPI, programs are compiled with the MPI library and run with `mpirun`, which coordinates the starting of multiple instances of the program on multiple machines.

3.1 Initialization

MPI programs start by calling `MPI_Init()`. While programs can call `MPI_Init()` at any time, they are discouraged from accessing any external systems (like file IO) before doing so.

Once initialized, MPI programs can use `MPI_Comm_rank()` to get their rank, which serves to uniquely identify each process. While rank is a one-dimensional ID, the programmer can describe the topology of the processors in a Cartesian grid. I have not used this feature, as I chose to set the topology in the `buffer_t`, as described in Chapter 5. If MPI implementations can make use of the topology to enhance performance, then it may be wise to also describe the topology to MPI.

3.2 Communication

In my project, I use two types of communication supported by MPI.

3.2.1 Point-to-Point

In point-to-point communication, one process sends and another receives. Both processes must agree on the size of data to be sent and tag their message with the same integer. On the surface, it appears that one process is copying memory into a buffer in the other process. An important detail to note here is that a buffer cannot simply wait and listen for messages. All `MPI_Send()` calls must have a corresponding `MPI_Recv()` on the other end.
Point-to-point communication can be either synchronous or asynchronous. In my project, many processes must communicate at the same time in no particular order, so I chose to use asynchronous.

3.2.2 Collective

Collective communication involves all processes at once. Of relevance to this project specifically is the `MPI_Allgather()` call. For this call, each process starts with one piece of data, and ends with a buffer containing the data from every process. I use this call so that each process can know the region that each other process has and needs, as detailed in Chapter 6.
4 Changes to the Interface to Halide

Using an MPI to distribute a Halide program requires some additions to that program. One might wish it was as simple as "Distribute!", but for reasons I explain, some operations are left to the programmer.

4.1 Using MPI Tools

The program is an MPI program, meaning it must be compiled and linked with the MPI library. In addition, to take advantage of distribution, it must be invoked with mpirun. The mpirun program is responsible for launching multiple instances of the Halide program based on a specified topology. Since it's not possible to guess the topology of the cluster with Halide, it's left to the user to call mpirun. In addition, it's likely that users of computing clusters already have experience and infrastructure for distributing programs with MPI, and I would like to allow them to leverage that to the best of their ability when running Halide programs.

In addition, I require the programmer to call MPI_Init() and MPI_Finalize() in the client code. Halide assumes that if it is working with distributed inputs, then it can freely make MPI calls. There are a few reasons that Halide does not call MPI_Init(). The first is that MPI programs are strongly discouraged from accessing the system, such as using file IO, before calling MPI_Init(). However, the programmer will likely want the freedom to make system calls before running a Halide function, for example, to load the input to the function from disk. The second is that the programmer may be using the Halide function as part of a larger distributed program with other needs involving MPI. The programmer may wish to organize the processes into communicators. If the program was structured such that processes needed to communicate to decide whether to call the Halide function at all, then it would seem rather silly to have Halide initialize the communication. Another minor reason is that the user may want direct access to any error information provided by MPI during setup.
4.2 Inputs

The most important change I've made to the interface for calling a Halide function is in the input. Halide functions often take buffer_t structures as inputs, and it is these functions for which my work is targeted. When such a function is required to run on an input too large for one machine, each machine must be given only a chunk of that input. This input is assumed to look like any other buffer_t, in that it has four or fewer dimensions and represents a rectangular region in space. Since the programmer knows the problem and the cluster topology, the programmer is responsible for splitting the input into chunks. However, those chunks are assumed to be congruent, rectangular sub regions of the global image.

When the program runs, it loads only the chunk it is responsible for into a buffer_t. This buffer_t has some additional requirements placed on it. I've added a field to the buffer_t called d_extent, short for "distributed extents". This field describes, for each dimension, how many chunks the input is split into. It is these fields from which Halide gleans the topology of the image with respect to the rank of each process. In addition, the programmer is required to set the min field such that it accurately represents the chunk's position in the larger image. The end result is that the accesses to the image are replaced at runtime with accesses to a new image, which holds all the data it needs as collected from other processes. This process is described in detail in Chapter 6. However, a programmer must still take care not to access elements outside of the bounds of the global image, in the same way that writers of non-distributed programs must not overstep the bounds of their input.

4.3 Outputs

To evaluate a Halide function, the programmer gives the function a buffer to fill and the function fills it. No additional requirements are placed on the output buffer besides the fact that it should accurately describe the region of space within the global space. The output of the distributed function is a buffer_t, as is usual for Halide functions. Each process is left with one buffer, so collecting the results is left for the programmer, who knows what the data is needed for.
If the Halide function represents only one iteration of a loop, then output is already in the form required to be inputted back into the loop.

4.4 Only Images Are Distributed

While designing this system, I had several drafts for interfaces where individual functions in a pipeline could be marked as having their storage distributed. In these designs, communication could happen during a pipeline. However, no compelling use was found for this feature. It seemed that it would always be possible to collect the data required before execution. In addition, a function that would be distributed would have to be both stored and computed as root, as having communication and synchronization within a loop would be extremely inefficient. If a function is computed and stored as root, it’s not a large change to merely split the pipeline at this point, making it an output function itself. With this strategy, one can use my code to insert communication in the middle of a pipeline.
5 Additions to Halide

New code in Halide is responsible for preparing distributed images for use by client programs.

5.1 New Image Checks

One of the early stages in Halide lowering calculates the region of inputs required to make the outputs. With this information in hand, the library puts checks into the function to make sure that it doesn’t use more input than it has. However, when using distributed images, the input is expected to require more of the input than it has loaded. Therefore, for distributed images, these image checks are replaced with new ones that take into account the size of the global image. In addition, a new node, called an MPI_Share node, is inserted into the code that marks the buffer for transformation.

5.2 Replacing the Buffer

When the MPI_Share node is processed, it is turned into multiple statements that have the combined effect of replacing the input buffer with a new, larger one with data from all required sources.

The first step in this process is to create and allocate a new buffer. Since Halide already has plenty of code that deals with making buffers, this step only requires inserting a new Realize node of the correct dimensions. The correct dimensions are the exact dimensions calculated to be required by the input function. Because there are checks in place to make sure this isn’t outside the grander image, we know that these bounds are not too large to fill (although, as discussed in Chapter 7, it may be obnoxiously large). Because the required bounds are calculated to be the max and min possibly required, we know that this region isn’t too small.

Now that we have a new buffer, which is named after the original input with the "_collected" appended to the name, it is necessary to make sure it is used. The MPI_Sharing lowering stage walks the tree of statements looking for accesses to the
original image. When it finds one, it simply changes the target image to the collected image. At this stage, accesses are still in the multidimensional space, which is shared by the input and collected image. Therefore, there is no need to change the indices of the access, only the image that it refers to.

5.3 Filling the Collected Buffer

For the collected buffer to be useful, it must be filled with data. To this end, a call to my external library, described in Chapter 6, is inserted with the new buffer and old buffer as arguments. When the program runs, these will be pointers to the buffer_t structure in memory. The two buffer_t structures together contain all the information required to determine what data needs to be requested and where to find it. When this call returns, the collected buffer is filled with the data required by the pipeline.

All of this replacement is completed before any loops that access the image, so by the time those loops execute, the collected image is ready for use. Note that since all MPI communication is done by an external function, there is no need for Halide to link with MPI. If ahead-of-time compiling is used, there is no need for the generating program to link with MPI either. Only the code using the Halide function, which now includes MPI communication, needs to be linked with MPI and run with mpiexec.
6 Collecting the Distributed Image

When the Halide library sees that the Halide program has marked an image as distributed, it inserts a call to `halide_mpi_collect()` into the function before any input images are accessed. This function resides in an external library I call `libcollect`. This separation saves Halide from the burden of linking with MPI.

This function's inputs are the process's original input `buffer_t` and the additional `buffer_t` created to hold the collected data. The collected buffer has the correct shape for use by the Halide function, but has no data in it. This function has the responsibility of collecting the data and putting it in the buffer.

6.1 Initialization

The first step is to perform some simple steps to make sure the inputs are valid. These include checks that the input pointers are not null and that they refer to `buffer_t` structures with non-zero dimensions. Then `libcollect` situates itself with MPI by checking that it is initialized. It asks MPI what the current process's rank is. The shape of the global image and the local chunk's place in it are still stored in the `buffer_t`, where the programmer specified it initially, as described in Chapter 4.

6.2 Creating the Map of Needs and Local Regions

Because MPI point-to-point communication requires that both parties acknowledge the transfer before any transfer takes place, it is necessary for the process to calculate not only what it wants to receive but also what it must send. It is not possible for processes to contact each other and ask for data. Therefore, my system requires all processes to exchange their required region with each other process. To this end, each process allocates an array of `buffer_t` structures large enough to hold one from each process, and then makes a call to `MPI_Allgather()`. `MPI_Allgather()` takes as input one object from each process and an empty array of that type of object. It then fills the array with the input data from every
process. In other words, if there are $n$ processes, process $i$ starts with data $i$ and ends with data $[0, n]$. A similar method is used to share what data each process has and can send. An alternative method is discussed in Chapter 7.

With this information in hand, each process can determine which processes it needs data from and which processes it needs to send data to. These are determined by a simple loop which uses a simple comparison of min and extent in each dimension to determine whether buffers overlap. The input chunk is tested against the required region of each other processor to see if a send is required. The required region is tested against the local region of each other process to see if a receive is required.

6.3 Sends and Receives

Once each process knows which other processes to receive from, it calculates the region it needs to receive from each process. This calculation is accomplished by finding the overlap between the regions this process needs with the region local to the other process. Then, it allocates space for the received data and calls MPI_Irecv().

After setting up all the receives, the process moves on to the sends. The sends use a similar overlap calculation on the local region with the requested regions of the other processes. When sending, a process must also copy out the data from its local region into a new buffer which holds the overlap. The copy allows MPI to send from a contiguous region of memory. Then the process points MPI at this region with MPI_Isend().

Before proceeding, all sends and receives must be completed. MPI_Waitall() is used to give MPI time to resolve all the requests.

6.4 Filling in the Buffer

When the receives terminate, each process will have many buffers filled with fresh data from other processes. The sum of these buffers, possibly combined with the local region, will comprise the collected region. Values from these buffers are copied into the collected buffer provided by Halide. After these copies, control is passed back into the Halide function so it can use the data.
7 Performance and Possible Optimizations

In this section, I will discuss some expected use patterns and how my code performs on them. This chapter will focus on counting how many messages and of what size are sent. Numeric results will be presented in Chapter 8.

7.1 Dense stencil

The first and most common usage pattern is a simple stencil. In a stencil operation, the value of each element depends on the values of a rectangular region around that element, usually a weighted average. A perpetual question about stencil operations is what to do at the borders, where the element depends on values that are outside of the image. When distributing an image, my code is responsible for extending the borders of the image by collecting those values into a single buffer. Figure 7.1 describes this problem.

![Figure 7.1 At the Corners, a Stencil Needs Additional Values](image)

In this type of problem, libcollect copies exactly what is required. Each process will send and receive to processes that own regions in a rectangle around itself. To determine how far that rectangle extends left in dimension 0, one would divide how far the stencil extends in that dimension by the width of a local region and round up. One would also calculate how far the stencil extends in the other direction and round up. The sum is the width of the rectangular region in that dimension. The number of messages sent is the product of the sides of this rectangular region. Each message contains all the data that the
receiver needs from the sender. Often, each process will need data from only the neighboring regions. In this case, the region is three processes wide in each dimension, and the volume of the whole region is $3^n$, where $n$ is the number of dimensions. Since a process doesn't need to send to itself, the number of messages sent is $3^n - 1$. The primary way to optimize this case is to reduce the number of dimensions over which the input is distributed. Figure 7.2 demonstrates this technique with splits along two dimensions and three dimensions.

![Diagram showing communication under different distributions](image)

**Figure 7.2 Communication Under Different Distributions**

### 7.2 Sparse Stencil

A stencil doesn't necessarily need to make use of every value in the stencil region. One could imagine a stencil that averages each element with the value of the element in a local region two regions to the right of it. With this pattern, my system will collect the region as though it were dense, resulting in unnecessary messages and bandwidth. Removing this excess would require a new representation of stencils that Halide does not currently have.

### 7.3 Degenerate Pattern

While Halide programs tend to have some kind of symmetry in their accesses, like in the stencil, one can imagine access patterns that are completely degenerate. For example, imagine a one-dimensional `buffer_t` that is distributed in that dimension. Let the width of the `buffer_t` and the number of buffers be the same. We could then address each point by
where \( x \) is the index of the element in the buffer and \( y \) is the index of the buffer in the global image. The transformation \( f(x, y) = f(y, x) \) makes it so that each process requires one element from every other process. This pattern requires \( n^2 \) messages, and there is no apparent way to fix it with my strategy. It may be that this case is too strange to have an elegant solution that does not reduce performance in more normal situations.

Figure 7.3 demonstrates this problem.

![Figure 7.3 A Degenerate Case](image)

### 7.4 Reductions

Another type of distributed function one might want to evaluate is a reduction. In a reduction, the output is significantly smaller than the input. For example, one might have several terabytes of data distributed among the different machines in a cluster and simply desire the average value of this data. My distributed strategy does not address this type of need at all. However, for many cases, it is likely possible to use MPI's reduction features using values calculated by Halide.

### 7.5 Potential Optimizations

A future implementation of libcollect would not need the `MPI_Allgather()` calls for the stencil case. Halide could analyze whether the access pattern can be proven constant and
equal for each element. If so, the needs and haves of each region could be calculated
deterministically from the shape of the stencil and distribution.

Another possible optimization involves forgoing the use of MPI for communication on
the same node. If processes could access each other’s memory directly, they could save much
of the repackaging done for the benefit of creating MPI messages. However, this kind of
optimization would require significant additions to the current system and is beyond the
scope of this project.
8 Numerical Results

8.1 Experimental Setup

In order to investigate the performance of various aspects of the image collection system, I performed some experiments on the Lanka Cluster. My experimental setup involves running an 8x8 box blur on 1 MB images using four nodes of the Lanka Cluster. Each node has two Intel Xeon E5-2695 v2 @ 2.40GHz processors with 12 cores each and 128 GB of memory. The blur is repeated 1000 times in order to better estimate the real performance. In all but the serial control trial described in Section 8.3, four nodes were used with 16 processes per core.

The goal was to measure the performance impact of distribution with respect to a few different parameters.

8.1.1 Shape

The first parameter was to measure two different shapes of local regions. In SQUARE experiments, the local region used was 1024x1024 elements and the global region was 8x8 local regions. In SLICE experiments, local regions were 131072x8 and the global region was 1x64 local regions. In SQUARE experiments, each process loads a total of

\[ (1024+8) \times (1024+7) - 1024 \times 1024 = 15416 \text{ bytes} \]

from 8 total neighbors.

In SLICE experiments, each process loads a total of

\[ 131072 \times 8 + (131072 \times 7) = 1966080 \text{ bytes} \]

from only 2 total neighbors. The purpose of this parameter was to investigate the relative importance of the quantity of messages and the size of those messages.

8.1.2 Collection System

The second parameter controlled which portions of the collection system were used. The hope was to isolate the effects of various components of the collection process on performance. The different values for this parameter are:
- **NONE** - The `d_extent` and `d_stride` parameters were not set. Halide compiles this program as a completely local program using none of my additional code.

- **GATHER** - The `d_extent` and `d_stride` parameters are set, but all operations of the collection library are disabled except the distribution of needs and local regions using `MPI_Allgather()`. Because `libcollect` does not execute in its entirety, the resulting output isn't correct. However, this experiment is useful for isolating the relative effects of the various stages of the collection process.

- **COPY** - The images are set to be distributed, but the buffer sends and receives are disabled. The difference between these experiments and GATHER experiments are that these allocate and copy to and from receive and send buffers, even though those buffers do not have the correct data from the point-to-point messages.

- **FULL** - The images are set to be distributed, and the resulting blur is accurate.

The third parameter is whether the BLOCK or CYCLIC method was used to distribute ranks among processes. With BLOCK, processes on the same node have one continuous interval of ranks. With CYCLIC, ranks are distributed to alternating nodes in round robin fashion. In BLOCK distribution experiments, processes communicate almost entirely with other processes on the same node. In CYCLIC experiments with the SLICE shape, every process communicated exclusively with processes on the other nodes. In SQUARE shaped CYCLIC experiments, processes communicated with a mix of processes on the same node and processes on different nodes. It may have been possible to maximize the internode communication for these experiments, but the data as it stands does not suggest that this would be productive.

The fourth parameter is whether a time consuming point operation was weaved in between the blur operations. Specifically, the time consuming operation was to take the `sin` of the element, then the `sqrt` of the element, and finally the `asin` of the element. The thought behind adding this step was to detect any strange effects caused by a pipeline with almost no compute load. No effects of this nature were discovered, so I will ignore this parameter in the analysis.
8.2 Experimental Results

The raw data for all experiments can be found in the Appendix. The following two charts summarize the most interesting results.

![SQUARE Regions Chart]

**Figure 8.1** Execution Time on SQUARE Regions
8.2.1 Collect Performance

For experiments with SQUARE shaped regions, the difference between the distributed and non-distributed execution was approximately 20%. Whether this result is fast enough will depend on the problem, but it does mean that the majority of the time in execution was spent in the local part of the code. As the amount of computation done on the data increases, the ratio spent on communication will decrease.

Of the time spent in communication, it appears that 75% of it was spent in the copying and allocation of send and receive buffers. For this problem, it seems that the penalty paid for distribution is not the time spent sharing data, but the time spent reshaping the data so it can be more conveniently transferred and read. To address this, the number of tasks per node should be decreased so that there is less reshaping overhead.

The SLICE experiments showed a similar pattern to the SQUARE experiments.

In neither case was the time spent sending the messages significant. Most likely, the MPI implementation supports shared memory message passing, where messages to processes on the same nodes are not copied but simply shared between processes. Therefore, most of the time is spent repacking that data for transfer unnecessarily. Without
the shared memory optimization described in Chapter 7, the system described in this paper works best when the communication is primarily between processes on different nodes.

8.2.2 BLOCK vs CYCLIC

In the SQUARE data, the difference between the BLOCK and CYCLIC experiments was negligible. However, the difference was significant, although only 11% of the slowdown from distributed in general, for the SLICE experiments. This contrast would suggest that for the network, the bandwidth is far more important than the quantity of messages. However, network use did not seem to be the major bottleneck in any case.

8.3 Control

<table>
<thead>
<tr>
<th>TIME</th>
<th>SHAPE</th>
<th>RANK</th>
<th>COLLECT</th>
<th>TIME</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>ONE</td>
<td>FULL</td>
<td>52.909</td>
<td></td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>BIG</td>
<td>N/A</td>
<td>39.25888</td>
<td>628.142</td>
</tr>
</tbody>
</table>

**Figure 8.3 Single Machine Comparison**

As a control for my experiments, I measured the execution time of the blur on a single machine, once using libcollect to distribute to different processes, and another time with a serial program. For the serial program, I used a 16 Mb image, instead of a 1 Mb, to represent the global image that the distributed case was working on. This experiment used a SQUARE shape. After dividing by the factor of 16 that represents the per node parallelism of every distributed trial, I found that this serial program took approximately 75% as long as the distributed version per element processed. This result is similar to the non-distributed parallel case, which is the NONE case in the previous figures. I suspect that the serial version is faster per operation than the parallel case without distribution due to multiple processes sharing memory bandwidth.
9 Summary of Contributions and Future Work

9.1 Contributions

In this thesis, I have described a system created for enabling the construction of distributed Halide programs. Programmers can now write programs that process extremely large data sets, but have each machine load only the necessary region. Ideally, this will enable the portability and optimization of Halide to be leveraged against much larger problems.

9.2 Future Work

While my system is currently useable, there are a few adjustments that should be made. One is the automatic stencil detection described in Chapter 7, which would reduce the communication required between processes. Currently, Halide limits itself to image sizes that can be addressed with only 32 bits. Since local regions are addressed as though they are part of a larger, global image, they must obey this 32 bit limitation. I have not investigated the difficulty of extending the address space to 64 bits. Additionally, if desired, a more convenient method than splitting the pipeline could be used to enable distribution of function outputs in addition to input images.
10 References


11 Appendix

<table>
<thead>
<tr>
<th>TIME CONSUME</th>
<th>SHAPE</th>
<th>RANK</th>
<th>COLLECT</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>SQUARE</td>
<td>CYCLIC</td>
<td>FULL</td>
<td>91.538</td>
</tr>
<tr>
<td>YES</td>
<td>SQUARE</td>
<td>BLOCK</td>
<td>FULL</td>
<td>92.581</td>
</tr>
<tr>
<td>YES</td>
<td>SQUARE</td>
<td>N/A</td>
<td>COPY</td>
<td>91.536</td>
</tr>
<tr>
<td>YES</td>
<td>SQUARE</td>
<td>N/A</td>
<td>GATHER</td>
<td>85.283</td>
</tr>
<tr>
<td>YES</td>
<td>SQUARE</td>
<td>N/A</td>
<td>NONE</td>
<td>82.478</td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>CYCLIC</td>
<td>FULL</td>
<td>53.233</td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>BLOCK</td>
<td>FULL</td>
<td>53.936</td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>N/A</td>
<td>COPY</td>
<td>53.192</td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>N/A</td>
<td>GATHER</td>
<td>46.56</td>
</tr>
<tr>
<td>NO</td>
<td>SQUARE</td>
<td>N/A</td>
<td>NONE</td>
<td>45.052</td>
</tr>
<tr>
<td>NO</td>
<td>SLICE</td>
<td>CYCLIC</td>
<td>FULL</td>
<td>167.904</td>
</tr>
<tr>
<td>NO</td>
<td>SLICE</td>
<td>BLOCK</td>
<td>FULL</td>
<td>161.619</td>
</tr>
<tr>
<td>NO</td>
<td>SLICE</td>
<td>N/A</td>
<td>COPY</td>
<td>160.921</td>
</tr>
<tr>
<td>NO</td>
<td>SLICE</td>
<td>N/A</td>
<td>GATHER</td>
<td>128.404</td>
</tr>
<tr>
<td>NO</td>
<td>SLICE</td>
<td>N/A</td>
<td>NONE</td>
<td>114.005</td>
</tr>
</tbody>
</table>