COMPILING IMAGE PROCESSING AND MACHINE LEARNING APPLICATIONS TO RECONFIGURABLE ACCELERATORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Jeff Setter
December 2023

This dissertation is online at: https://purl.stanford.edu/bn933dk7986

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark Horowitz, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Fredrik Kjoelstad

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

Approved for the Stanford University Committee on Graduate Studies.

Stacey F. Bent, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format.

Abstract

Recent vision applications provide exciting new opportunities in photography, autonomous driving, and image generation. In turn, new hardware platforms have similarly risen to efficiently execute this class of applications. Effective compilers are necessary to seamlessly run these new applications on hardware accelerators. However, hardware accelerators use specialized memories to increase efficiency, which makes them challenging compilation targets. Addressing this issue requires changing the abstraction that the compiler uses to represent memories.

In this dissertation, we present such a compiler. It compiles image processing and machine learning applications to dataflow accelerators. To create this system, we extend the Halide domain-specific language (DSL) to target streaming accelerators. Using new scheduling primitives, the user has full control over optimization decisions. These optimizations can be tailored to new hardware accelerators. We introduce a unified buffer abstraction to provide an interface between application definition and hardware memory configuration. This abstraction enables efficient hardware implementations while supporting the generality of applications that are represented in our abstraction. Our compiler enables compute sharing by generating designs that time-multiplex compute operations with low utilization. We demonstrate the effectiveness of this compiler by running applications on the Amber Coarse-Grained Reconfigurable Array (CGRA), designed by the Agile HArdware (AHA) group at Stanford.

Acknowledgments

I would like to express my deepest gratitude to each and every person at Stanford University who has played a part in my incredible journey during my PhD program. The support, guidance, and inspiration I have received from all of you have been instrumental in shaping my academic and personal growth. I am truly grateful for the opportunity to be a part of this vibrant and intellectually stimulating community.

First and foremost, I extend my gratitude to my advisor, Mark Horowitz. Upon joining Stanford, my initial goal was to design RTL hardware for emerging application spaces. However, over the course of several years at the university, my focus expanded to encompass the development of complete systems, spanning the creation of applications, compiler development, and hardware mapping. The collaborative execution of a substantial system with the AHA group, alongside interactions with esteemed professors and industry leaders, was only possible under Mark's mentorship. He provided invaluable guidance on the fundamentals of image processing applications and optimal strategies for crafting efficient, single-ported memory. The opportunity to learn the entire software-hardware stack broadened my perspective, leading me to discover a keen interest in compilers. Throughout our meetings together, Mark offered pertinent and candid feedback on my design ideas, fostering the evolution of a more effective application compiler. I value the collaboration and accomplishments we have shared and anticipate continuing to rely on his advice as I progress in my career.

I thank my dissertation readers, Christos Kozyrakis and Fred Kjolstad. Christos was an immense reassurance during my introduction to Stanford; his students welcomed me to the university and helped me feel included. Christos's exciting attitude piqued my interest in the recent trends in computer architecture, and helped guide my early research direction. Fred Kjolstad provided helpful feedback and expert insight on my compiler. Furthermore, I enjoyed his unique perspective on software design and methodology that took a more systematic approach to software engineering. Incorporating these ideas have enhanced the quality of my work.

I want to extend my appreciation to Abbas El Gamal for chairing my oral defense. He was the professor for my first Stanford class (EE 278) when I entered my first year, and so it was fitting that he bookended my Stanford journey by chairing my oral defense. He brought a humorous take in the classroom and oral defense that brought just the right amount of lighthearted spirit. Priyanka

Raina was the final member on my oral defense committee as well as a collaborator on papers and in the AHA group. I am grateful of all of her suggestions and commitment to our research projects so that they maintained the highest rigor and excellence.

During my undergraduate years at Cornell University, I was guided by my research advisor José Martínez and Professor Chris Batten. Both encouraged me in research and provided the foundation I needed to succeed at Stanford. They never hesitated when I asked for second opinions even after I had graduated from Cornell. I also received help and suggestions from Andrew Adams about Halide implementation. His expertise helped guide my modifications to the Halide compiler to ensure that my hardware extensions fit within the Halide framework.

I met many great students while at Stanford. I first want to thank the people that I collaborated the most for paper submissions, including Jing Pu, Xuan Yang, Qiaoyi Liu, Dillon Huff, Maxwell Strange, and Kathleen Feng. Working on such large systems meant trusting others to implement and design code. Luckily, I found that every student I worked with was willing and capable. We learned from each other in order to create full system from high-level software application down to running our own silicon hardware. I am grateful for their willingness to work tirelessly up to submission deadlines for the last bit of data, and revise our papers to come under the page limit.

I enjoyed learning and creating the systems in the AHA group at Stanford. Group members include Alex Carsello, Po-Han Chen, Ross Daly, David Durst, Jake Ke, Taeyoung Kong, Kalhan Koul, Ankita Nayak, Ritvik Sharma, Kavya Sreedhar, Keyi Zhang, Richard Bahr, Clark Barrett, and Steve Richardson. With this talented group, we were able to design an image-processing and machine learning accelerator, create an application compiler to our hardware, and then successfully tape-out and evaluate a chip. I am proud of this group and of the collective successes we have had.

I also am grateful for an entertaining research group. Other members of the Horowitz research group that I interacted with are Steven Bell, Heonjae Ha, Daniel Stanley, Zach Myers, and Sneha Goenha. Having a group with a diverse set of backgrounds, I learned about many topics across electrical engineering, as well as grew as a person during the engaging discussions at group lunches.

My family helped me, including my Mom and Dad. My parents were patient and understanding of my graduate education. They provided advice and support whenever I needed it, and encouraged me throughout my studies. My siblings, Jason and Renée, were also supportive during my PhD. Jason, and his wife Sakshi, provided insight into the hardware industry and gave valuable feedback on my presentation practice runs. Renée has kept me grounded and focused on my research as we reflect on academia and research.

And finally, I extend my deepest gratitude to my wife, Judy. She has persevered through so much struggle and success with me. She has been a comforting presence beside me during long nights working on projects and repeated practice for upcoming talks. Her love and support have encouraged me throughout this demanding journey. I owe her a great deal for being supportive and patient as I have worked on my PhD at Stanford.

Contents

List of Tables

List of Figures

List of Code Snippets

Chapter 1

Introduction

In recent years, we have witnessed remarkable advancements in the field of vision applications, encompassing image processing [37,102], autonomous driving systems [9,93], and generative image creation [5,12,83]. Traditional image processing techniques have been augmented with the integration of machine learning, leading to significant improvements in accuracy and performance [32,61,98]. To meet the growing demands of these vision applications, hardware accelerators have emerged as key enablers, offering enhanced computational power and efficiency. These accelerators span a range of platforms, from small systems on the edge like mobile phones [6,41], to large-scale data centers [47,74].

Some recent studies have shown Coarse-Grained Reconfigurable Accelerators (CGRAs) as a promising hardware platform that can accelerate a wide range of applications [21,48,74,99]. CGRAs are hardware accelerators consisting of a 2D array of computational and memory tiles that are connected with programmable wires. CGRAs benefit from computation elements that efficiently perform stencil and tensor computation, while also maintaining the flexibility that eases mapping applications. These aspects make CGRAs a nice blend of the flexibility of CPU processors while seeing some of the efficiency gains from ASICs.

While the pace of designing applications and hardware has accelerated, designing the compilers necessary to map these applications to custom hardware remains a challenge. CGRA accelerators usually suffer from poor programmability. Many designers choose to use custom software languages rather than provide compilers from more popular front-ends [19,97]. Building a robust and effective compiler is necessary to support a wide array of applications and properly utilize the available computational power of the accelerator.

One gap in accelerating an algorithm is taking an existing algorithm and porting it to more efficient hardware. The current process typically consists of creating an algorithm for more generic hardware, such as a CPU, and then making the modifications necessary to run on the target accelerator [7,23,47]. However, this porting process can lead to lower than expected performance

gains unless the algorithm is modified significantly to match the underlying accelerator. Ideally, an application developer would be able to use the same algorithm with scheduling directives to target an accelerator.

Once an appropriate algorithm is found, another challenge is memory mapping. Prebuilt hardware accelerators have trended towards using complex address generators and controllers bundled with the SRAM for efficiency [15,73]. Standard compiler flows use intermediate representations at a lower level, which makes it harder for them to map their results to this higher-level object. Instead, we would prefer a compiler that works on higher-level memory primitives rather than at the level of individual loads and stores.

Finally, some applications have low hardware utilization, which leads to inefficient usage of the computation elements. These situations arise from the popular mapping technique where each operation maps to its own computation element [76,109] combined with the fact that not all operations need to process the same amount of data. In these types of situations, we would like to allow the user to use hardware scheduling to improve compute utilization by instructing multiple algorithmic sections to use the same hardware elements.

Our research group developed a CGRA for image processing and machine learning applications [15]. I helped build the application compiler that we needed to map applications to our custom hardware. During this process, we made key design decisions to construct the compiler to address the aforementioned issues. In this dissertation, I describe our compiler that maps to our custom CGRA. This compiler uses additional scheduling and memory abstractions to incorporate new efficient hardware designs seen on CGRAs.

My main contributions shown in this dissertation are:

- Extend the Halide scheduling language to map to CGRAs.
- Develop the unified buffer abstraction to encapsulate the memory transfers and dependencies for image processing and machine learning.
- Enable compute sharing for reconfigurable accelerators through Halide scheduling and proper mapping through the compiler system.

My contributions on the front-end compiler are part of a larger compilation system. The penultimate chapter describes how the Halide front-end fits within the application compiler. I then describe the strategy I use to schedule applications to the CGRA. The full compiler and CGRA hardware are then evaluated on image processing applications and machine learning kernels. I finally conclude with what I learned while working on this dissertation.

Chapter 2

Background

Hardware accelerators have seen a rise in popularity as more complex algorithms clash with the desire for more energy-efficient devices. Due to the end of Moore's Law and Dennard Scaling, hardware devices are no longer getting dramatically more efficient from process technology [42]. At the same time, applications are becoming more complex with the rise of many new image [37, 102] and video algorithms [32] as well as the growing use of deep neural networks (DNNs) [25,43]. For users to be able to run these new applications on their portable devices, they need to run on hardware accelerators, which have proven to be more energy efficient and faster than general compute [42]. However, once a hardware accelerator is designed, there is still the arduous task of mapping applications to the new accelerator, a task usually handled by a compiler for that accelerator. It must find a way to map the application while utilizing the accelerator's special hardware features to provide performance and energy efficiency gains.

The next section, Section 2.1, describes foundational image processing and machine learning applications that our compiler must handle. With these motivating applications, Section 2.2 describes how hardware designers have created specialized, efficient accelerators for vision applications. These accelerators show the potential of hardware designs, along with their drawbacks in flexibility. Section 2.3 then surveys different reconfigurable accelerators that have better flexibility over the domain-specific accelerators. Our primary target is the Amber CGRA, which is described in that section. And finally, Section 2.4 details previous work on compilers for hardware accelerators and reconfigurable architectures as well as some of the abstractions needed for these compilers. We use these abstractions, along with the contributions of this dissertation, to form a complete application compiler for reconfigurable accelerators.

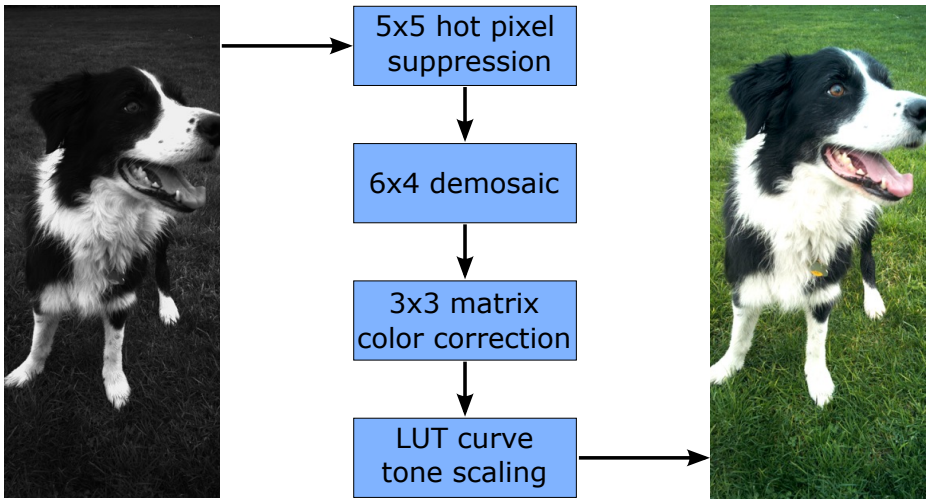


Figure 2.1: This is a sample stencil application called *camera_pipeline* that transforms a mosaicked, raw image taken from a camera sensor and converts it to an RGB image. It exhibits a pipeline of four kernels with stencils of different sizes.

2.1 Applications

The key to developing efficient hardware accelerators is understanding the applications that are running on devices. Today, people are using many devices with image processing, video processing, and machine learning applications. These tasks include image enhancements on smartphone cameras, vision systems on autonomous vehicles, and generative images built using large language models (LLMs). While the number of tasks using visual data has not diminished, many applications are not using the pure image processing techniques of the past. A new class of vision applications [61,80,98] take concepts of traditional image processing algorithms and combine them with the power of DNNs. DNN models have been trained to do noise reduction, super resolution, and segmentation to widen the capabilities of image applications. Understanding both of these domains is important for constructing new vision applications and hardware.

Image Processing

Image processing is a vast class of applications that ranges from improving image quality by removing noise, to stitching multiple images together to create a composite. Most of these applications share similar characteristics. They use image data as an input, and make heavy use of data in the vicinity of a pixel to determine intermediate results for an image patch. The overall computation is usually described as a series of computational kernels, each of which only uses a small stencil of data around each input pixel to generate the kernel's output. Thus, most of these applications can be described by a pipeline of stencil computations, as illustrated with the example in Figure 2.1. These stencils are

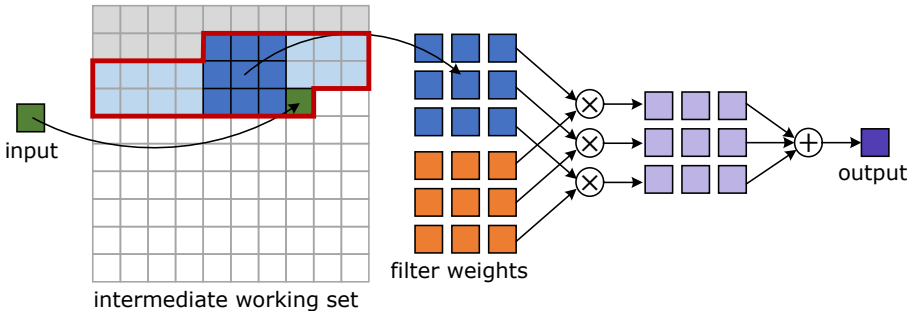


Figure 2.2: This line buffer efficiently stores values for a 3×3 stencil by keeping a little over two lines in its working set. The output of the line buffer memory is convolved with filter weights to compute each output pixel.

commonly convolution kernels, which consist of an element-wise product of weights and input data followed by a sum. With this pipeline of kernels, efficient execution looks for the fastest execution and the minimum memory required for each of the required kernels.

To find an efficient hardware execution for image processing applications, we need to use a proper memory implementation. One fundamental memory implementation used with stencil pipelines is a line buffer, as shown in Figure 2.2. Line buffers use the fact that many image processing applications read their input images in scanline order (left to right for each row, then from top to bottom). A stencil computation waits as the input pixels are scanned row by row. The first stencil computation can then start once enough input pixels are available to fill a stencil. Once a single stencil is completed, the next stencil computation overlaps with the last stencil's data. Only a column of pixels to the right needs to be added. The crux of achieving efficiency in a line buffer is exploiting the overlap between successive stencils, and realizing that only a few lines of the image need to be stored at any time.

We use streaming memories to optimize the hardware efficiency of performing convolutions in hardware. Line buffers improve the efficiency of successive stencils by avoiding refetching overlapping data. Figure 2.2 depicts a 3×3 stencil where each pixel is used nine times in an application: three times consecutively in a row, and for three separate rows. A pixel is stored temporarily in a stencil register so that it can be used in consecutive clock cycles. Since the pixel is reused in the two subsequent cycles, the pixel is retained using a chain of two shift registers. This reuse ensures no refetching of the pixel as the stencil computes horizontally across a row. Reuse of pixels is also possible vertically on successive rows. Pixels are later used when the stencil returns for computation on the next row. We need to calculate the time before the pixels are reused, known as the reuse distance. Since the image is read in scanline order, these pixels are delayed for the number of pixels in a line. The name "line buffer" comes from the storage elements holding pixels for reuse after a scanline. By using line buffers with stencil registers, pixels are only streamed into the accelerator

a single time, and the pixel is available from various memory ports at the correct times. Due to the gained efficiency of line buffers, they have become the basic hardware building block for image processing pipelines.

Image processing applications are defined by a series of simpler computation kernels. Understanding these individual kernels provides a foundation for constructing efficient implementations of larger applications. We next go through a set of simpler image processing kernels that are later used in the evaluation of our compiler system. One of the simplest examples is basic filtering with convolution kernels to improve the input images. Gaussian blur uses a weighted average of adjacent pixels to reduce visible noise. Unsharp masking is an image processing application that subtracts a blurred version of the input image to produce a sharpened output image. Finding corners is an important component of image processing as it provides a preliminary step for image alignment, registration, and tracking. Harris corner detector [36] is a pipeline of gradients and blurs that are combined together to identify where corners exist on an image. FAST corner detection [85] is another technique that uses twelve circular pixel comparisons to determine a corner. Camera pipeline [3] is a series of hot pixel suppression, demosaicking, and gamma correction to take raw pixel values to a recognizable color image. Each of these image processing applications are implemented in hardware with a pipeline of stencil computations connected by line buffers.

Pyramid Computation

Stencil processing provides a great framework for applying local computations that focus on small patches in an image, but are not the right structure to use for computations that need longer range or global computation. A common method for constructing global image processing applications is pyramid image processing. Pyramid image processing retains the same stencil techniques and small local patch computation, but makes global computation feasible by creating a stack of images, where image resolution decreases as you move up the stack. Thus the same size stencil has a larger reach as you move up the pyramid. The first step for pyramid applications is creating an image pyramid through successive stages of Gaussian filtering followed by downsampling kernels. The filtering is done to reduce aliasing artifacts in the downsampled images. Each pyramid level creates an image of half the width and half the height of the previous level. This process is repeated several times to create the same image, but of different sizes. The pyramid varies from the original image size at the lowest level, to a much smaller version for the highest level. If desired, one can continue this pyramid to a single pixel to find the average value of the original image. An example application is shown in Figure 2.3.

The most basic image pyramid, the Gaussian pyramid, creates a multi-scale representation of the input image using a Gaussian blur. Another common type of image pyramid is a Laplacian pyramid. This pyramid is constructed by first creating a Gaussian pyramid. Then, for each level in the pyramid, the smaller level is upsampled (becoming the same size as the next larger level)

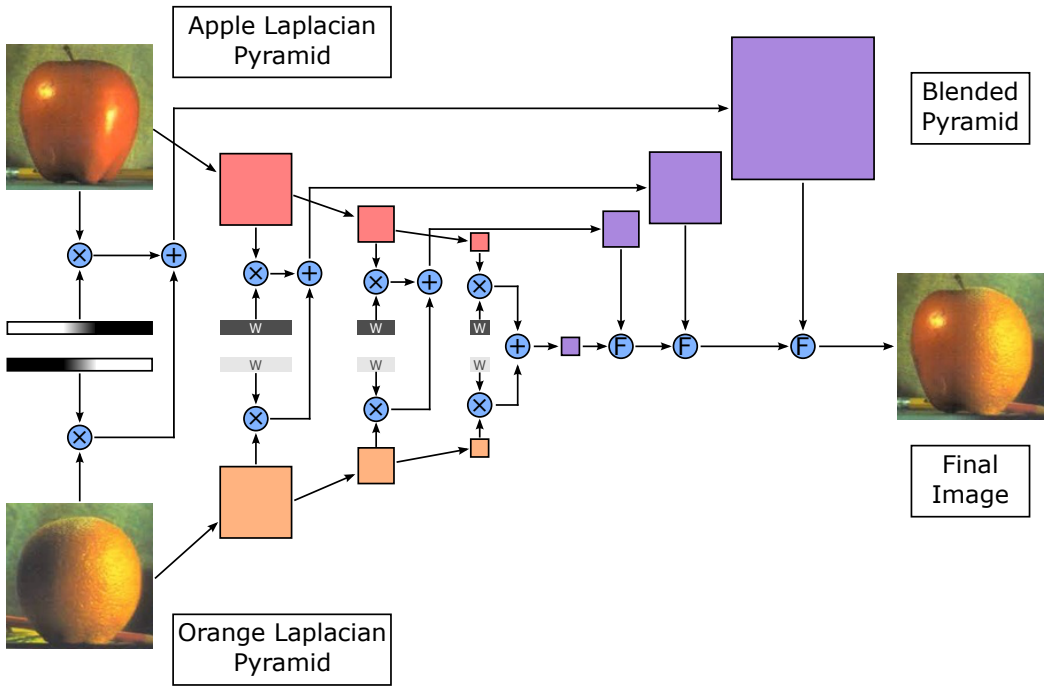


Figure 2.3: An example pyramid application that blends two images together. Each input image is decomposed into a Laplacian pyramid. Each pyramid level is then convolved with specific weights to provide the feathered blending. Next, each level is summed with the corresponding level on the other input. Last, the blended pyramid is flattened using a series of upsamples and addition to create the final image.

and subtracted from this level. The difference in pixel values becomes the value of the Laplacian pyramid of the larger level. The Laplacian pyramid along with the smallest level of the image is able to reconstruct the original image by summing the upsampled image with the Laplacian level, and repeating until it is the size of the original image. Another property of the Laplacian pyramid levels is that they represent the different spatial frequencies of the image with finer features (higher spatial frequencies) captured in the larger image levels.

Once an image pyramid is created, certain image processing algorithms work on the image pyramid. These algorithms have the benefit of working on patches that occur from images of different sizes [4]. This leads to algorithms that are multi-scale, meaning they are able to identify features regardless of their patch size in the actual image. Typically, the same computation kernel is performed at all levels of the pyramid. The same features are found and refined at each level.

There are numerous uses of pyramids in applications. These use cases include: using the inherent decomposition of frequencies from Laplacian pyramids; utilizing the hierarchical nature of image pyramids to reduce computation; or applying algorithms on different pyramid levels to identify features of various sizes. Pyramid blending [13] takes two images and seamlessly combines them

together using their Laplacian pyramids, as depicted in Figure 2.3. Burt and Adelson in [11] extend the Lucas-Kanade optical flow algorithm [60] used for stereo vision to hierarchically find the pixel displacement between two images. Google HDR [37] uses pyramids to hierarchically align image bursts. Implementing these pyramid applications with specialized hardware could follow similar line-buffered pipelines as seen in image processing. However, constructing and processing smaller, downsampled images in an image pyramid can lead to inefficient hardware. We revisit this dilemma in Chapter 5.

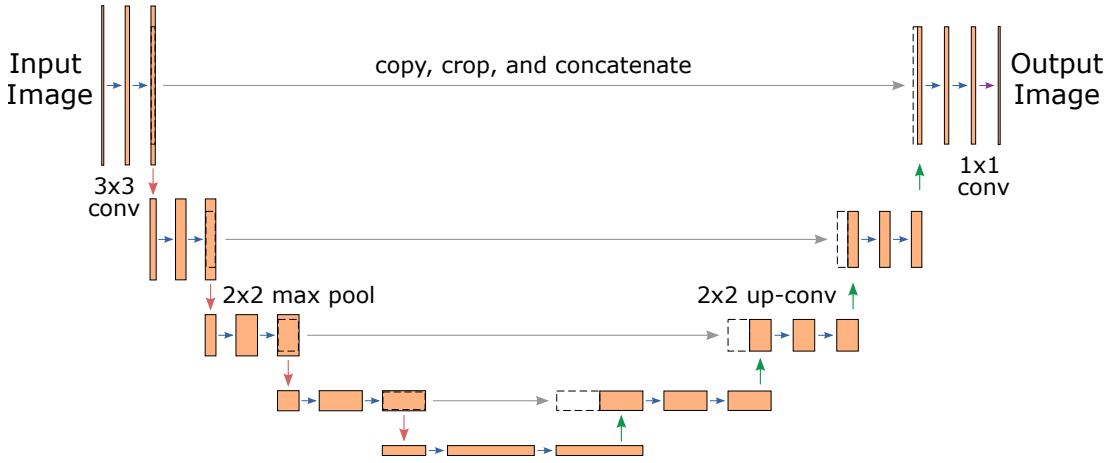


Figure 2.4: An example DNN application showing U-Net. This application takes an image input and outputs a cropped image with a segmentation mask around target regions. The application consists of 19 convolutional layers, 4 downsampling max-pool layers, and 4 upsampling concatenate layers. The series of layers resembles a U, hence its name.

DNNs

More recently, machine learning has become an important application for hardware acceleration. One popular implementation for machine learning is a deep neural network (DNN). These implementations use a large amount of data in the form of tensors, which are n-dimensional arrays. The data is then fed into many computation layers. The *deep* descriptor in DNN comes from the large number of layers in a DNN as compared to previous architectures that had far fewer layers. Researchers found that more layers had better result accuracy, but they also require far more training and computation than early network architectures.

The computation in a DNN is commonly a convolutional neural network, which consists of a series of convolutions similar to image processing as shown in Figure 2.4. However, there are some important differences. First, these convolutions compute on input tensors, not images. Each input tensor consists of multiple channels, where each channel is a conventional 2D image. Similarly the output is also a tensor, again consisting of multiple channels of 2D images. If we attempted to use

```
// Host loops, a single iteration since inputs fit within the accelerator memory
for y_host = 0:1
 for x_host = 0:1
  // Iteration loops in outermost accelerator memory
  for yo = 0:2
   for xo = 0:2
    for wo = 0:4  // output channel
     // Innermost loops on innermost accelerator memory
     for ro.z = 0:2  // reduction loop of input channel
      for r.y = 0:3  // reduction loops of conv kernel
       for r.x = 0:3
        for yi = 0:26
         for xi = 0:26
          unroll for wi = 0:8  // output channel; parallel execution
           unroll for ri.z = 0:8  // parallel execution of input channel
            // Define the overall variables after considering tiling
            host_tilesize_x = 26 * 2;  outer_tilesize_x = 26;
            x = x_host * host_tilesize_x + xo * outer_tilesize_x + xi + r.x
            host_tilesize_y = 26 * 2;  outer_tilesize_y = 26;
            y = y_host * host_tilesize_y + yo * outer_tilesize_y + yi + r.y
            outer_tilesize_w = 8
            out_ch = wo * outer_tilesize_w + wi
            outer_tilesize_z = 8
            in_ch = ro.z * outer_tilesize_z + ri.z
            // Computation for convolution
            output(x, y, out_ch) += weights(r.x, r.y, in_ch, out_ch) * input(x, y, in_ch)
```

Figure 2.5: Sample loop iterations for a convolution layer in a DNN. Loops are tiled, reordered, and performed in parallel on hardware accelerators.

a line buffer with DNNs, we would need to use every input channel while computing each output channel simultaneously. This almost always is not possible because the computation is too large. Instead, we carefully block the computation to reduce the storage working set, and reorder the computation loops. This blocking means that input image tiles are reused multiple times, which means that we need to store the full tile, and not just a few lines. Thus, we use a different memory structure, known as a double buffer, that allows data to be reused multiple times.

A single layer in a DNN has thousands of times more computation than a convolution in an image processing application. Because of this, accelerating a DNN typically focuses on just a single DNN layer at a time. Even with a single DNN layer, the inputs and weights must be tiled and accumulated over multiple accelerator executions due to their sheer size. By tiling these memories, we can keep a small working set of values that are reused many times. Reuse of memory values is key to creating an energy-efficient implementation of an application. The multi-dimensional data along with tiling creates many loops for computation. The order of the loops affects the opportunity for reuse, and becomes an important factor for the efficiency of the hardware acceleration. Figure 2.5 shows some pseudocode on how tiling, iteration order, and computation look for convolution layers.

To optimize the efficiency of DNN applications, we look again towards the data reuse as we did for image processing. Matrix multiplications as well as multi-channel convolutions have different

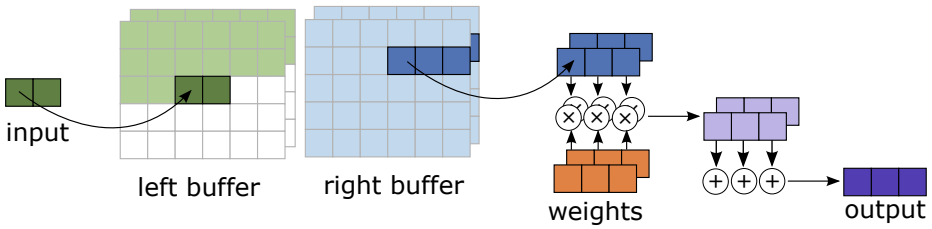


Figure 2.6: A double buffer efficiently stores and distributes input values for DNN applications. The two buffers alternate in phases where one buffer distributes values to the compute kernel (right buffer in example), whereas the other buffer stores input values for the next tile. After both buffers have completed the phases, they switch roles where the filled input tile is now fed into the compute kernel. Each of the memory operations are potentially vectorized to provide parallel execution. Vectorization can be performed independently with a mismatch of rates; however, it is more optimal with a similar number of iterations on each side.

reuse patterns as compared to stencil pipelines. Memories are blocked into a memory hierarchy and loops are reordered to achieve as much reuse as possible. With DNN loops, we have utilized parallelism in output and input channels and perform an iterative reduction on the convolution kernel loops. This leads to a different memory access pattern that cannot be optimized by line buffers. Line buffers are optimized for parallelizing computation in the x and y spatial dimensions; meanwhile, DNN layers can be reordered such that we compute pixels across channel dimensions where we see more parallelism. Rather than reusing a single block as we stream through an image, we ensure that a steady state of tensor blocks have a high throughput. These blocks take many cycles to initialize, accumulate, and output. To ensure the computation kernels always are busy, these phases are commonly overlapped with loop pipelining. To overlap this computation, we use a different streaming memory, called a double buffer, so that two blocks can operate at once as shown in Figure 2.6. In this configuration, half of the buffer feeds values into the computation kernel; meanwhile, the other half of the buffer initializes the subsequent block of values. This overlap is necessary to ensure that communication (initializing the next tensor) is not a bottleneck for the computation pipeline. We can increase bandwidth of the memory transfers into the double buffer as well as overlap these data transfers to ensure that useful computation is occurring on each cycle. Similar to image processing, this memory technique is used to ensure the compute kernel is always highly utilized.

Recently, there have been many advances in DNN applications. This recent surge in DNN advancement began with convolutional neural networks tackling the problem of image classification. The ImageNet competition [24] challenged researchers to build systems to identify 1000s of classes in an image database. AlexNet [52] found that creating a convolutional neural network (CNN) along with increased training time on GPUs could improve the error rate dramatically. VGGNet [91] improved upon their results by increasing the number of layers from 5 in AlexNet to 16 layers and using only 3×3 convolutions. Network architectures continued to grow in depth with each

year. Furthermore, new architecture techniques were introduced such as 1×1 convolutions by Inception [95] and skip connections by ResNet [38]. With the growth of all of the architectures, the number of parameters was growing rapidly and not amenable to mobile platforms. Although model accuracy continued to improve, they were taking larger computers and more time to train. MobileNets [43] suggested methods to reduce model sizes by using separable filters and a resolution hyper-parameter to reduce overall model size. These separable filters use less computation than the convolution neural networks in other models, and their execution looks more like image processing by first performing a normal convolution and then a 1×1 convolution (DNN convolution layer with 1 input and 1 output channel). However, not all models can be distilled into fewer parameters with the techniques introduced in MobileNets.

Apart from CNNs working on images, DNNs have also been used on sequence data, such as speech [86], text [35], and translation [94]. The structure of these algorithms is different, since they need to keep track of context and state. Due to these unique needs, early architectures used recurrent neural networks (RNNs) that took advantage of long short-term memory (LSTM) to keep track of temporal behavior. Later works improved upon RNNs and instead used attention modules in a Transformer architecture to improve training speeds [100]. BERT [25] enhances the performance of even more NLP tasks by using a bidirectional architecture. GPT-3 [12] introduced a large pre-trained language model that performs exceptionally well across NLP tasks, which has started the craze for highly capable large language models (LLMs).

Although the application space of DNNs is large and continuing to grow, their hardware acceleration follows a consistent implementation. A large array of multipliers and adders connected to double buffers remains the fundamental component of DNN hardware accelerators. By creating a hardware accelerator with these fundamental computation and memory blocks, we are able to efficiently run many of these applications.

2.2 Domain-specific Hardware Accelerators

With the large number of applications in image processing and DNNs, hardware designers have created domain-specific accelerators (DSAs) to run these applications efficiently. Each of these hardware implementations strive to accelerate a class of applications to remain useful across many tasks.

Image Processing Accelerators

Image processing accelerators take a variety of strategies to improve efficiency. Some accelerators simply provide many vector processing units to provide the compute necessary for these data-parallel applications. NVIDIA's programmable vision accelerator [87] uses this approach of creating systems with simply more compute. Microsoft's Hololens Processing Unit [96] employs this same strategy

of providing ample SIMD units for computation, but also includes application-specific hardware for image processing. There is a hardened joint bilateral filter and neural network to improve the efficiency of these operations even further. Finally, some accelerators include specialized compute as well as specialized memory systems to accommodate image processing applications. The research accelerator Convolution Engine [77], Google's Pixel Visual Core [79], and Movidius's vision processor [68] all include processing units as well as line buffer implementations. These accelerators all use these efficient hardware implementations to provide better performance as well as include the flexibility to handle a wide array of image processing applications.

DNN Accelerators

As DNN applications have become popular, ways to accelerate DNNs have also emerged. Neuflow [28] is one of the earliest accelerators developed by university researchers. This accelerator uses a 2D array of processing elements that can execute multiply-accumulates (MACs), normalization operators, and non-linear operators. This flexibility allows all layers of a convolution neural network (CNN) to run on the chip. DaDianNao [18] is a 2D array of tiles that have abundant local storage along with the compute units to maintain adequate memory bandwidth. Eyeriss [17] studied the role of dataflow and reusing locally stored elements to improve accelerator efficiency. By interchanging loops in a convolution layer, an application can keep the weight, output, or input stationary within a compute unit to increase reuse. TETRIS [31] is a proposed accelerator that uses 3D memory layout and hierarchical tiles to gain more performance and efficiency for large DNN layers.

In addition to university DNN accelerators, companies have created even larger DNN accelerators. NVIDIA Research created Simba [88], which uses an even larger systolic array of multiply-accumulates. Tesla created neural processing units [9] to run vision applications on their self-driving vehicles. Google found that their cloud services were running more DNNs, so they created tensor processing units [47] to achieve greater efficiency and cost savings over their datacenter GPUs. With applications seeing no size limit, Cerebras unveiled their Wafer Scale Engine [58], which creates a giant systolic array with MACs and local storage across an entire silicon wafer.

Hardware accelerators have started to converge towards a systolic array structure with local connections between tiles with MACs and storage. This hardware can efficiently execute both convolution layers and matrix multiplication. From the description of all of these accelerators that have been developed in academia and industry, one may suspect that their usage is widespread. However, the opposite is true: most DNN implementations are running on NVIDIA GPUs. The main reason is the difficulty of compiling applications onto these accelerators, which will be discussed further in Section 2.4.

2.3 Reconfigurable Accelerators

Although domain-specific accelerators gain tremendous efficiency in a single domain, they are much more rigid in their dataflow. On the other side of the spectrum, CPUs provide the most flexibility with control-flow instructions and programmable instruction streams. Reconfigurable accelerators strike a middle ground between generic CPUs and dedicated ASICs. They remain programmable to support a wide range of applications, but also are limited enough in scope to allow for high-performance and energy-efficient hardware implementations compared to generic CPUs. This group of accelerators includes FPGAs and CGRAs. They get their flexibility from programmable wires that connect their generic compute elements. This flexibility comes with drawbacks though. Programmable wires and muxes require silicon area to implement and additional energy to run an application. With image processing and DNN applications spanning different domains, the additional flexibility is useful for accelerating new vision applications.

FPGAs

FPGAs, or Field Programmable Gate Arrays, are a flexible hardware platform for running applications. They can be used to verify new hardware architectures before tapeout [29], or efficiently implement ever-changing algorithms in production [20].

FPGAs are composed of lookup tables (LUTs), digital signal processors (DSPs), flip-flops (registers), and Block RAMs (BRAMs). LUTs implement bit-level logic. By composing many LUTs together using the FPGA's programmable wires, one can create fundamental gates (such as ANDs, ORs, and XORs) as well as more recognizable operators (such as 16-bit adders and comparison operators). As gates get larger, the overhead of combining so many bit-level gates becomes expensive, so DSPs are used. DSPs typically implement a multi-bit multiplier. For memory, FPGAs use shift registers and BRAMs. Shift registers are used on the FPGA fabric for pipelining while BRAMs provide storage for larger arrays.

There are two major FPGA products on the market today. These are Altera FPGAs [46] sold by Intel and Xilinx FPGAs [105] sold by AMD. Each of these FPGAs are composed of similar hardware units as well as software to map and integrate applications with FPGAs. The generality of the compute elements allow them to run any computation, while the programmable wires can implement deep image processing pipelines or systolic arrays for DNNs. Lowering applications to FPGAs is relatively straightforward, since LUTs and DSPs implement all computation, while BRAMs and registers hold all state in the application.

Coarse-Grain Reconfigurable Architecture

The Coarse-Grain Reconfigurable Architecture (CGRA) is a class of reconfigurable accelerator that provides higher-level primitives than FPGAs. Instead of bit-level LUTs, CGRAs use multi-bit

operators such as adders and multipliers. This sacrifices the efficiency that precise bitwidths can achieve, but suits the needs of applications that are accustomed to bitwidths of common datatypes (such as 8, 16, 32, or 64 bits wide). Additionally, creating operators of higher bitwidths improves the energy efficiency over the equivalent configuration needed on an FPGA. CGRAs remain a research topic with no standard design for the processing elements (PEs) or memory units.

ADRES [63] is one of the earliest CGRAs; it is built with an array of tiles with processing and memory combined with a flexible interconnect. The CGRA is built with the host processor in mind with a shared memory for communication. FPCA [21] increased the complexity of each PE by chaining together operators to increase potential usable transistors over an ALU design where only a single operator is used. FPCA memory is stored in local memory banks with configurable address generators. DySER [34] increased the PE size differently by using vector SIMD units. Plasticine [75] similarly uses vector units in each PE. For memory, Plasticine uses scratchpad memories and on-chip address generators to generate different memory patterns. Plasticine has been adapted for commercial use with SambaNova's RDU [74], which uses similar tiles but with a much larger grid of 10× as many tiles. 4D-CGRA [48] uses a custom processor in each PE that tracks loop iteration and branch divergence. Combined with the spatial dimensions in the array, the 4D-CGRA works with data in four dimensions. SNAFU [33] and Ultra-Elastic CGRA [97] both aim to create very low power CGRAs by reducing buffering and using DVFS, respectively. SNAFU [33] and DSAGEN [103] build a framework where a designer can create unique PEs and compose their own CGRA fabric. These frameworks create the abstractions necessary to integrate your own PE implementation as well as the compiler necessary to use the generated CGRA. To make compilation easier, we found it useful to create our own abstractions, which we introduce below in Section 2.4.

Amber CGRA

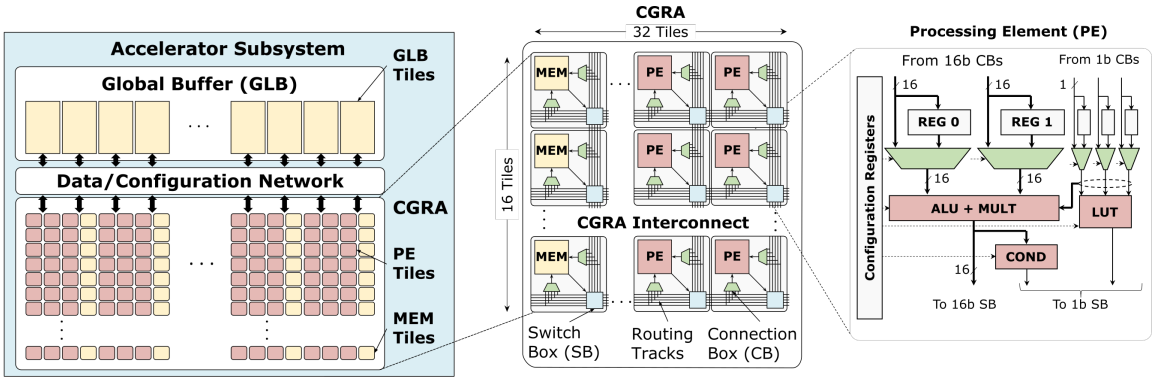


Figure 2.7: Our Amber CGRA is a 16 × 32 array of processing element (PE) and memory (MEM) tiles. One-fourth of the tiles are MEMs and the rest are PEs. The memory tile contains the optimized Physical Unified Buffer (PUB) depicted in Figure 2.8.

We constructed a new reconfigurable accelerator called Amber [15] in the Stanford AHA group. This new Coarse-Grain Reconfigurable Array demonstrates how a flexible hardware fabric can run an entire class of applications. The first iteration of this CGRA accelerated just image processing applications [99]. The Amber CGRA runs image processing and deep neural network applications and has many updates to the processing elements, memory tiles, and memory hierarchy. The Amber CGRA is our main motivating hardware target, so we describe its components in detail below.

The Amber CGRA consists of a 16×32 sized grid of tiles that are connected together by programmable wires. As shown in Figure 2.7, processing element (PE) and memory (MEM) tiles are connected by routing tracks through switch boxes and connection boxes. The computing fabric consists of 16-bit tracks as the primary data width, as well as 1-bit tracks for boolean values. Each PE is a simple computing element with an ALU and multiplier for 16-bit values as well as a 3-input LUT to perform logic operations. Before accelerator execution, configuration registers are set to choose which operation each PE performs.

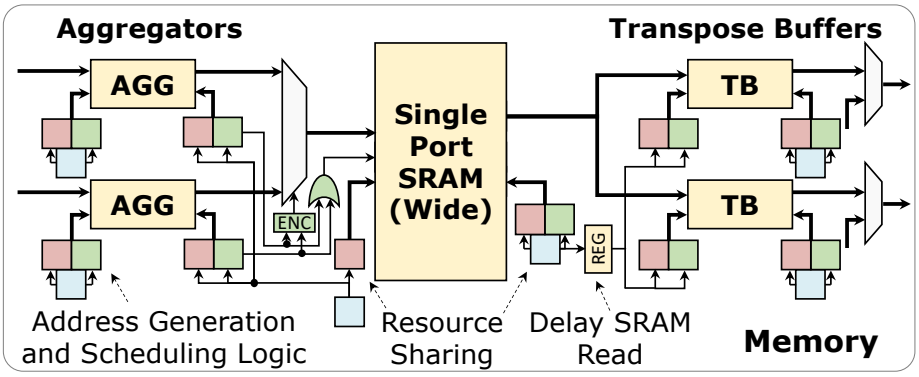


Figure 2.8: Diagram of a Physical Unified Buffer (PUB) with a wide-fetch single-port SRAM, aggregator (AGG), and transpose buffer (TB). Sets of ID/AG/SG controllers (blue/red/green blocks respectively) control the input and output of each sub-component.

Every fourth column on the CGRA fabric consists of memory tiles, which we call Physical Unified Buffers (PUBs). As shown in Figure 2.8, each memory tile is not simply a bare SRAM. Instead, each memory tile consists of SRAM connected to address generation and scheduling logic that is programmable during configuration time. This way, the memory reads and writes have pre-programmed locations and timing. The memories are controlled by an iteration domain (ID), address generator (AG), and schedule generator (SG). The ID sequentially counts through the tensor; the AG produces address values for the memory operations; and the SG calculates when each memory operation occurs. We go into more detail about calculating memory tile parameters in Chapter 4. In addition, the SRAM has a single wide-fetch port with four 16-bit words written or read every cycle. This configuration increases the power efficiency of the memory operations, but comes at additional complexity in scheduling the reads and writes. To accommodate the wide fetch and

single port, words are accumulated in aggregators (AGGs) and transpose buffers (TBs). The AGGs accumulate adjacent words over several cycles before writing all four words to the SRAM. Similarly, the TB holds four adjacent words so that they can be transferred out of the memory tile over four cycles. Overall, the memory tile is able to achieve high efficiency with its embedded controllers and wide-fetch SRAM port, but it also requires a configured static schedule to dictate the memory operations. It becomes a challenge to compile applications to a memory tile like this with a complex configuration space.

Connecting the PEs and MEMs is an interconnect of switch boxes and connection boxes. The programmable wires are configured before execution to connect a tile output to any set of tile inputs on the CGRA fabric. The role of place-and-route (PnR) is to find an efficient placement of the application's PEs and MEMs that can then be routed with these flexible wires. Registers also exist in the PEs and switch boxes to allow applications to be pipelined to meet timing. A set of 32 input/output (IO) tiles are above the top row of tiles. These allow for data to be streamed into and out of the the global buffer (GLB). The GLB consists of 16 large SRAM banks each with their own dedicated controller, similar to the memory tiles. The GLB then is connected to DRAM through the host processor. It is the role of the application compiler to find an efficient way of configuring the PEs, MEMs, and routing resources to execute the application.

2.4 Compiling to Hardware Accelerators

To bridge the gap between applications and target hardware architectures, compilers must lower and map input applications. It remains a challenge to automatically compile these domain-specific applications to new hardware accelerators. Accelerators differ from traditional CPU hardware in their method of computational parallelism as well as their memory structures. We find that CPUs and hardware accelerators are trying to solve different problems. CPUs have powerful control flow mechanisms to handle any type of program, and then use branch prediction and decoupled fetch units to alleviate any memory stalls. Hardware accelerators take a different approach with very wide compute parallelism. Since missing data would end up stalling the very large compute resources, they prevent memory stalls by construction with software-managed memories, which makes the compiler job much more difficult. These differences require compiler back-ends that are tailored for each of their unique requirements. When mapping an application to physical hardware, one must consider four components:

- **computation kernels** that perform numerical operations on data;
- **memories** that store values for reuse and possible reordering;
- **addressors** and **controllers** that calculate memory addresses and execution timings;
- **network** that wires together different elements on the accelerator fabric.

Understanding these fundamental hardware building blocks is critical to developing an effective compiler. The granularity and complexity of these blocks impose restrictions on the applications and hardware mappings. Furthermore, abstractions are necessary to bridge the gap between software concepts and their eventual hardware implementation. Different languages and techniques have been developed to facilitate this complex series of steps.

Mapping Applications to FPGAs

FPGAs originally were programmable solely by hardware languages such as Verilog, VHDL, and SystemC. However, there was a desire to create hardware implementations from software languages, known now as high-level synthesis (HLS). Zhang worked on creating the first system [109] that took a subset of C++ code and created hardware that mapped to FPGAs; this work eventually evolved into Vivado HLS.

Today, HLS is a common way to produce designs for FPGAs. Designers create an application and testbench using a subset of C++. These designs are modified with special hardware types to specify exact bitwidths. Furthermore, pragmas and directives are used to decorate and direct the HLS tool to produce the desired hardware results. The commercial tools created by hardware vendors are Vivado HLS (now Vitis HLS) [104] for Xilinx FPGAs, and OpenCL [45] for Altera/Intel FPGAs. Another popular HLS tool is Catapult HLS by Mentor Graphics [67] which can target Xilinx FPGAs, Altera FPGAs, or even generate custom ASICs. MaxCompiler [62] and LegUp [14] are other HLS compilers for FPGAs.

Due to the success of HLS compilers, many projects have built on top of these compilers to map to FPGAs. These projects start from higher-level languages and domain-specific languages, then generate HLS C code, and lastly use the vendor HLS compilers to map to FPGAs. Hipacc [81] uses C++ templates to implement an image processing domain-specific language (DSL) that creates HLS C. SODA [19] uses a custom high-level DSL for describing stencil applications to generate HLS C whose parameters are automatically tuned for high performance. HeteroCL [55], HeteroHalide [57], and Halide HLS [76] use a separate algorithm and schedule to independently create the algorithm from the hardware schedule. We share this strategy and explain its benefits in Section 3.1. All of these systems use HLS to ease the compilation process to hardware since HLS C is semantically closer to the DSL languages than RTL hardware languages. Therefore, many systems compile their DSL language to HLS C, and then use high-level synthesis to generate hardware. Furthermore, the languages that use a separate schedule have a clear mechanism for generating the pragmas and directives used in HLS. Combining application schedules with DSL front-ends allow users to create efficient FPGA implementations from a higher-level language.

Other projects map to FPGAs but elect to not use HLS for mapping. Instead, they produce hardware designs directly and then map the RTL to the FPGA. Darkroom [39] and Rigel [40] use a novel high-level DSL with structural composition to create line-buffered image processing pipelines

in Verilog. Aetherling [27] and μIR [89] create Chisel [8] code. Both Verilog and Chisel are hardware design languages that can then produce an ASIC design or map to an FPGA. Their DSL front-ends already closely align with the structural hardware semantics, so generating RTL is a straight-forward process.

DNNs can also be mapped to FPGAs. DNNWeaver [90] creates a tiled and batched schedule to efficiently run on an FPGA's limited memory resources. DNNBuilder [107] builds a pipeline of DNN layers made possible by low-bit quantization. Both of these projects use parameterized hardware generators that are tailored for the application and available hardware resources on the FPGA.

FPGAs enable users to create hardware accelerators for image processing and DNNs starting from DSLs. By generating an RTL design, FPGAs can map the hardware components to the flexible FPGA fabric. With recent advancements in HLS, compilation from software DSLs to FPGAs has become much easier. The simplicity of the compute and memory elements allow for flexible arrangements that can implement any application. FPGAs provide a high performance with the flexibility to accommodate many applications. However, to improve energy efficiency further, we look towards more specialized accelerators such as programmable domain-specific accelerators and CGRAs. These hardware platforms improve energy efficiency and performance, but due to their specialized nature, also make a more difficult compilation target.

Compiler Systems for DSAs and CGRAs

Domain-specific accelerators (DSAs) and CGRAs use specialized computation to gain performance and energy efficiency benefits over FPGAs. However, these unique computation blocks also become more difficult compilation targets. This issue was identified very early on with the ADRES paper [63] in 2003, stating that "we believe the compiler is even more important than the architecture." Most CGRA papers acknowledge the difficulty of compilation and discuss their compilation strategy in their papers [21, 33, 48, 63, 75, 97, 103]. The compiler for CGRAs are challenging, because the PEs, memory units, and address generators create a large configuration space with their own set of constraints and limitations. The growing number of hardware accelerators increases the importance of creating better hardware/software abstractions and sophisticated compilers.

Due to the complexity of mapping applications to CGRAs, many systems have accompanying papers that describe the compiler for their hardware accelerator.

- Google uses Tensorflow [1] as a language to describe DNNs. In order to map these applications to CPUs, GPUs, and custom TPU [47] accelerator, they use the XLA compiler [56]. XLA helps to dynamically configure applications and uses kernel fusion to ensure high performance on each target platform.

- TVM [16] has become a popular framework for generating efficient hardware schedules for DNN applications. TVM enables applications written in popular DNN front-end languages to

compile through the TVM system. TVM then automatically tunes the schedule parameters and searches for the highest performing schedule. VTA-TVM [69] is an extension to this system that allows TVM to target efficient hardware-accelerated executions that run on FPGAs.

- MAERI [54] introduces a unique DNN accelerator that uses a distribution and reduction tree to implement different DNN layers. The many configurable switches require a robust compiler to map each of the different DNN layers in an application. MAESTRO [53] is an analytical model that is able to evaluate the performance of different dataflows on the accelerator to make improvements on the accelerator schedules for MAERI.

- Plasticine [75] is a CGRA with an interlaced array of vector PEs and memory tiles. Address generators control the memory tiles, while switches control the flow of data between tiles. Spatial [50] provides a high-level interface to create applications for reconfigurable hardware, including Plasticine. SARA [108] extends Spatial by abstracting the resources to allow for better partitioning and scheduling of resources.

My effort has focused on creating a compiler for image processing and DNN applications for our own Amber CGRA. To build this application compiler required integrating different abstractions that were built by other members of Stanford AHA. Understanding some of these other components in the compiler help provide context for the front-end compilation steps. We next describe some of the abstractions that are needed in a complete application compiler.

Compute Representation

Our compiler for the Amber CGRA uses CoreIR [22] to represent the application graph. Compute kernels are directly mapped to CoreIR after the application compiler. CoreIR is again used after memory mapping to construct the full application. CoreIR defines a set of basic computation primitives. This set is composed of operations based on smt-lib [10].

The Amber CGRA uses PEak [26] to formally define the available compute operations, and then rewrite rules can map the compute operations in CoreIR to the PEs on the Amber CGRA [51]. With the compute kernels defined, and the rewrite rules providing the mapping to the PEs, we are next faced with mapping the memory units.

Streaming Memory Representation

Memory mapping is a critical and challenging step of the hardware mapping process. Hardware accelerators typically use optimized memory structures that buffer streams of data. The Buffets paper [73] describes a memory primitive that can express many types of buffers, such as line buffers and double buffers. Buffets provide storage for explicitly decoupled data orchestration (EDDO), which is commonly found in hardware accelerators. EDDO memory systems push data directly by

address, and allow for overlapping write and read phases. PolyEDDO [72] provides the framework for how to compile to Buffets. PolyEDDO introduces an abstraction called Hardware Space-Time to allocate across spatially-instantiated compute units and time during application execution.

Polyhedral analyses have been developed to understand memory operations, and calculate when to schedule operations for optimal execution. Clockwork [44] is one such compiler that utilizes polyhedral analysis. We utilize Clockwork in our compiler system to perform loop fusion as well as schedule our accelerator. We elaborate on the usage of Clockwork more in Chapter 4.

Outline of the Next Chapters

Previous work has shown exciting new applications in image processing and machine learning. Following the interest in these applications, researchers have designed programmable accelerators to improve efficiency on constrained devices. Efficient memory implementations come at the cost of compilation difficulty. Furthermore, we strive to give the user choices to trade off application resource usage and runtime performance. In this dissertation, we build a system where a high-level Halide application is scheduled to an efficient implementation on a CGRA. The upcoming chapters are as follows:

- **Halide**: application language for image and array processing
- **Unified buffers**: memory abstraction to describe flexible memory primitives
- **Shared hardware**: extension to Halide scheduling and memory mapping for shared compute
- **Application flow**: full system built to target CGRAs from Halide

Chapter 3

Halide: Scheduling to Hardware

Halide [78] is a high-level language for image and array processing applications. Its convenience for creating high-performance code has led to its adoption at Adobe, YouTube, and Qualcomm. The high productivity and performance comes from its unique property of separating the algorithm from the schedule. The *algorithm* describes what operations to perform (to specify what the output values should be). The *schedule* then specifies how to perform these operations (in terms of loop optimizations and blocking) to ideally achieve a high performance on the specified hardware target. We use Halide as our front-end due to its expressiveness for describing image processing applications and DNNs, as well as its extensibility to target new hardware. In this chapter, my main contributions are extending the scheduling language to CGRAs with new scheduling primitives and new declarative scheduling using semantic sugar. Below we introduce the semantics of Halide, go in more depth about constructing algorithms and schedules, and then discuss the extensions to the Halide primitives for targeting hardware accelerators.

3.1 Halide Overview

The Halide language can create high-performance code for several hardware targets. These targets include CPUs (x86, ARM), GPUs (CUDA, Apple Metal), and DSPs (Qualcomm Hexagon). Halide applies optimizations to applications in its own internal intermediate representation called HalideIR. Each of these hardware targets have their own instruction set which HalideIR can generate. In addition, each hardware target has different memory sizes and parallelism. The differences between hardware targets motivate separating the algorithm from the schedule. For an application, we can create a different schedule to specialize the loop transformations for the specific hardware target. Since the algorithm and schedule are separate, we can use the same algorithm for every hardware target to make easy comparison and verification for an application evaluated on different hardware targets. This is because each schedule is ensured to not change the output values described in the

algorithm [82].

The basic element of each Halide algorithm is a Func. A Func is a named storage element that is defined in a multi-dimensional space. For images, this might be three dimensions: two spatial dimensions, x and y, and a dimension for the color, c. In the algorithm definition, the bounds for these dimensions are not defined. In fact, the size of the index variables and iteration in terms of for-loops are absent from the algorithm. Instead, the bounds of each index variable is defined and calculated during compilation or runtime. A computational pipeline is created by indexing Funcs by variables in each dimension. This provides an easy way to define every point in the space. This method of definition of multi-dimensional spaces lends itself well for defining algorithms in image processing and machine learning. Images and tensors consist of multi-dimensional arrays where the computation is repeated for every point.

Halide scheduling is important for defining the iteration loops and optimizing them for high-performance code. Even with the algorithm defined, the values of the index variables still have not yet been determined. Using scheduling primitives `bound` and `tile`, the size of for-loops can be defined. These work by defining the size of the output. `bound` specifies the absolute bounds of a variable at runtime. `tile` splits a loop variable into two parts with the inner loop variable having a specified bounded length. Bounds analysis is then propagated through the definition of the producer Funcs and input Funcs to calculate the bounds of each multi-dimensional Func. This step is important for determining how large each Func needs to be, and in turn how much storage is necessary to hold all intermediate values.

There are additional scheduling primitives, `store_at` and `compute_at`, that further refine the storage size of each Func. These are unique scheduling primitives introduced by the Halide language that trade off storage with recomputation. `store_at` defines a loop level at which a memory for a Func should be created. Defining the storage level to outer loops increases the storage required to hold all values, but reduces recomputation. `compute_at` defines the loop level where a Func should calculate and populate its memory. By defining the compute level to be the inner loop, the user can improve locality at the cost of making parallelism more difficult. Setting the compute level at the outer loop allows for greater parallelism, but also inhibits optimizations to reduce storage.

Exploring the trade offs can lead to higher performing code. The user is expected to try different schedules to optimize the performance. However, generating good schedules can be difficult. Several papers [2, 70] have aimed to automatically find good schedules for any application. These auto-schedulers relieve the burden of creating good schedules by generating them after exploring through the complex search space.

Halide is our choice for front-end language for our applications. It is able to succinctly describe both image processing and deep neural networks. The specialization for each back-end is a useful way to optimize for each hardware architecture. In this dissertation, we extend the Halide compiler to enable compilation to hardware accelerators, and specifically generate applications for our CGRA.

```
1  using Halide::ConciseCasts::u16;  // cast to 16-bit unsigned int
2
3  /* Algorithm */
4  Var x("x"), y("y");
5
6  // Define 3x3 kernel weights
7  Func kernel;
8  RDom r(0, 3, 0, 3), r2(0, 3, 0, 3);
9  kernel(x,y) = 0;
10 kernel(0,0) = 1;       kernel(0,1) = 2;       kernel(0,2) = 1;
11 kernel(1,0) = 2;       kernel(1,1) = 4;       kernel(1,2) = 2;
12 kernel(2,0) = 1;       kernel(2,1) = 2;       kernel(2,2) = 1;
13
14 Func conv1, conv1_norm, conv2, conv2_norm;
15 Func hw_input, hw_output;
16 hw_input(x, y) = u16(input(x, y));
17
18 // First convolution
19 conv1(x, y) = u16(0);
20 conv1(x, y) += u16(kernel(r.x, r.y)) * hw_input(x + r.x, y + r.y);
21 conv1_norm(x, y) = conv1(x,y) / 16;
22
23 // Second convolution
24 conv2(x, y) = u16(0);
25 conv2(x, y) += u16(kernel(r2.x, r2.y)) * conv1_norm(x + r2.x, y + r2.y);
26 conv2_norm(x, y) = conv2(x,y) / 16;
27
28 hw_output(x, y) = conv2_norm(x, y);
29 output(x, y) = cast<uint8_t>(hw_output(x,y));
```

Code 3.1: Halide algorithm code for the cascade application. It consists of two 3×3 convolutions using predefined weights that are normalized after convolution.

3.2 Halide Algorithm

We use Halide to define image processing and deep neural networks applications. As introduced in Section 3.1, Halide uses a functional style code with Funcs being the means of storing intermediate values. Index variables, such as x and y, are used on each Func, but their bounds are only defined later. By hiding the inherent loops from the algorithm, we can later manipulate the nature of loops to improve our hardware implementation during Halide scheduling.

Code 3.1 shows the algorithm for the cascade application. It consists of two 3×3 convolutions that use `kernel` as the weights for those convolutions. Since the weights sum up to 16, this value is normalized out on lines 21 and 29. The convolutions are performed using reduction domains (`RDoms`), which introduces a set of reduction variables ($r.x$ and $r.y$) as well as loops to perform the reduction. The reduction domain definition, `RDom r(0, 3, 0, 3)`, defines a two-dimensional reduction domain with initial value 0 and extent 3, meaning each is a 3×3 reduction. The usage of the reduction domains on lines 20 and 25 with the += operator shows that the reduction is adding each partial result a defined Func. The `RDom` accumulation on line 20 is an update to the initial value for Func `conv1`. Line 19 shows the initialization stage where each index point is set to 0; then,

line 20 is the accumulation to the same Func. These two lines produce two separate loopnests, and are scheduled separately using `conv1` and `conv1.update()` respectively. In the next section, we see how scheduling Funcs change their loopnests and execution.

3.3 Halide Scheduling

Halide scheduling is a unique aspect of the Halide language where loop transformations are added separately from the algorithm definition. Halide schedules strive to make the code run faster. Schedules alter the branching structure, affect locality for memories, and create parallel code. The success with separated scheduling in Halide has inspired other works to also use a decoupled schedule [16,49,55,92]. Below we explore the scheduling of the cascade application in Code 3.1 by introducing scheduling primitives gradually while measuring their effects on runtime.

`split()`, `reorder()`, and `tile()`

`split`, `reorder`, and `tile` are used to modify loops by strip mining and interchanging their order. In Halide, execution on a smaller tile of the input image is commonly beneficial since the full input image does not fit in a CPU cache. `split` is used on a loop to break it into two separate loops: an inner loop and an outer loop. `split` is then combined with `reorder` to have the inner loops executed before the outer loops. Using two `split`s and a `reorder` on a 2D image can create a loop order x_{inner}, y_{inner}, x_{outer}, y_{outer} from innermost to outermost. Due to the frequency of this exact scheduling sequence occurring, `tile` is the syntactic sugar to do these splits and then reorder.

Code 3.2 shows how the cascade algorithm in Code 3.1 can be tiled. The scheduling to generate the tiling is shown on line 3 (on the left). Alternatively, one could use `split` and `reorder` as shown in lines 6-9. The original x and y loops are broken into rectangular tiles of 512×32. Lines 1-4 of the generated loopnest (pseudocode on the right) show the iteration order. The variables for the number of tiles in the x and y directions are calculated at runtime based on the input image size. Lines 5 and 6 show how the tile indices are used with the inner indices to calculate the original image indices. The generated loopnest, however, provides little benefit on its own. All computation is by default computed inline leading to lots of recomputation. The nested reductions lead to 81 loads and multiply-accumulates for each output pixel even though much of the computation overlaps with previous iterations. We use the next scheduling primitives to overcome this issue.

`store_at()` and `compute_at()`

`store_at` and `compute_at` are used to assign where buffers are created and populated. `store_at` denotes which Funcs should be stored as intermediates in buffers. The default is that Halide inlines all computation resulting in recomputation of previously computed values. By creating intermediate

```
// Schedule 1: tiled loops
output
  .tile(x,y, xo,yo, xi,yi, 512,32);

// This is equivalent to:
// output
//  .split(x, xo, xi, 512)
//  .split(y, yo, yi, 32)
//  .reorder(xi, yi, xo, yo);

// Manually-tuned time: 3643.11ms
```

```
for yo = 0 to img_y_tiles:
  for xo = 0 to img_x_tiles:
    for yi = 0 to 32: // inner tiled loops
      for xi = 0 to 512:
        int x = xo * 512 + xi;
        int y = yo * 32 + yi;
        allocate conv2[uint16 * 1];
        for r2.y = 0 to 3:
          for r2.x = 0 to 3:
            allocate conv1[uint16 * 1];
            for r.y = 0 to 3:  // all inline
              for r.x = 0 to 3:
                allocate kernel[int32 * 9];
                kernel = ...
                conv1[0] += kernel(r.x, r.y)
                      * input(x + r.x, y + r.y);
            allocate kernel[int32 * 9];
            kernel = ...
            conv2[0] += kernel(r2.x, r2.y)
                        * conv1[0] / 16;
        output(x, y) = conv2[0] / 16;
```

Code 3.2: An example schedule for the cascade app shown in Code 3.1. This Halide schedule (on the left) tiles the `output` into 512×32 blocks. The default scheduling leaves all producers computed inline with `output`. This leads to excessive recomputation and a slow baseline execution time. The generated loopnest (pseudocode on the right) shows the tiled loops and computation locations.

buffers, the CPU stores the computed values. `store_at` takes an argument on which loop level to allocate the buffer. Halide's loopnests are constructed based on the output and producer-consumer dependencies, so the `store_at` loop level must be one of the loops in its consumer chain. The Func allocation then is sized such that all calculation of Func within the buffer's scope are stored. Thus, the user can influence the buffer size based on the bounds of the enclosed calculations. This is helpful for ensuring that buffers fit in the cache to decrease execution time due to temporal locality.

`compute_at` is used in conjunction with `store_at` to specify at which loop level a buffer is filled. `compute_at` takes an argument on which loop level to fill the buffer with values. All producer values needed to create an iteration of the consumer is populated into the buffer. By defining at what granularity the buffer is populated, both sliding window and storage folding optimizations can improve the generated code. Sliding window ensures that values stored in the buffer are not recomputed. Storage folding then determines if the buffer size can be reduced by implementing it as a circular buffer where consumed data space can be reclaimed and reused for future computation. Both of these analyses are done at compile time so that execution and buffer sizing can be statically known. Computing at inner levels provides greater opportunity for storage folding, but also leads to a serial dependence where parallelism is hampered. The extreme case of never recomputing output values leads to an inability to parallelize work, since each output tile overlaps slightly with its neighbors. Instead, it is common to tile the output to allow for parallel threads to work on different tiles and allow for some redundant work at the boundary of tiles.

```
// Schedule 2: memory granularity
output
  .store_root()
  .compute_root()
  .tile(x,y, xo,yo, xi,yi, 512,32);

// allocate and compute at xo
conv1_norm
  .store_at(output, xo)
  .compute_at(output, xo);

kernel
  .store_at(output, xo)
  .compute_at(output, xo);

// Manually-tuned time: 394.655ms
```

```
for yo = 0 to img_y_tiles:
  for xo = 0 to img_x_tiles:
    allocate kernel[int32 * 9]; // allocate at xo
      kernel = ...
    allocate conv1_norm[uint16 * 514 * 34];
    for yi = 0 to 34:
      for xi = 0 to 514:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv1[uint16 * 1];
        conv1[0] = 0;
        for r.y = 0 to 3:
          for r.x = 0 to 3:
            conv1[0] += kernel(r.x, r.y)
                      * input(x + r.x, y + r.y);
        conv1_norm(x,y) = conv1[0] / 16;
    for yi = 0 to 32:
      for xi = 0 to 512:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv2[uint16 * 1];
        conv2[0] = 0;
        for r2.y = 0 to 3:
          for r2.x = 0 to 3:
            conv2[0] += kernel(r.x, r.y)
                  * conv1_norm(x + r.x, y + r.y);
        output(x,y) = conv2[0] / 16;
```

Code 3.3: Schedule 2 (on the left) adds on storage and computation locations for `conv1_norm` and `kernel`. This stores and computes a tile of the first convolution that is small enough that it is cached by the time it is used in the next convolution. This speeds up the computation time by 9.23×.

Code 3.3 shows store and compute levels added to the cascade app. The `output` is by definition the root level, so `store_root` and `compute_at` are provided simply for clarity. Both `conv1_norm` and `kernel` are stored at the *xo* tile level, leading to their allocations within that loop level. They are computed at the same level, leading to a tile of `conv1_norm` being calculated before a tile of the `output` is calculated. Notice that in lines 6 and 7 of the loopnest, the producer tile is 514×34. This is because the 3×3 convolution creates a halo of a slightly larger producer tile than consumer tile. These halos lead to recomputation of border pixels in the producer, but this is worth the benefit of caching the results.

`unroll()`

`unroll` removes iteration loops by duplicating the statements in the body. Loop unrolling is helpful because it removes branch instructions and jumps in the program. Branches disrupt the predictable instruction sequence leading to worse execution time. When unrolling a loop, the iteration variable is replaced by all values that it would have had. Replacing iteration variables with constants can then lead to further simplification of the generated code. Generally, small loops are unrolled to improve the execution time, and LLVM provides some of these optimizations by default. In our example schedule, we disable this LLVM optimization to show how this optimization affects runtime

```
// Schedule 3: unrolled reductions
output
  .store_root()
  .compute_root()
  .tile(x,y, xo,yo, xi,yi, 512,32);

// unroll reduction
conv2.update()
  .unroll(r2.x).unroll(r2.y);

conv1_norm
  .store_at(output, xo)
  .compute_at(output, xo);

// unroll reduction
conv1.update()
  .unroll(r.x).unroll(r.y);

kernel
  .store_at(output, xo)
  .compute_at(output, xo);

// Manually-tuned time: 124.455ms
```

```
for yo = 0 to img_y_tiles:
  for xo = 0 to img_x_tiles:
    allocate kernel[int32 * 9];
      kernel = ...
    allocate conv1_norm[uint16 * 514 * 34];
    for yi = 0 to 34:
      for xi = 0 to 512:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv1[uint16 * 1];
        conv1[0] = 0;
        // unrolled reductions for 9 updates
        conv1[0] += input(x,y)
                    * kernel(0,0);
        ...
        conv1[0] += input(x+2,y+2)
                     * kernel(2,2);
        conv1_norm(x,y) = conv1 / 16;
    for yi = 0 to 32:
      for xi = 0 to 512:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv2[uint16 * 1];
        conv2 = 0;
        // unrolled reductions for 9 updates
        conv2 += conv1_norm(x,y)
                    * kernel(0,0);
        ...
        conv2 += conv1_norm(x+2,y+2)
                    * kernel(2,2);
        output(x,y) = conv2 / 16;
```

Code 3.4: Schedule 3 (on the left) adds in `unroll` directives so that the small reduction loops occur on individual instructions rather than a loop. The generated loopnest (on the right) consists of nine updates for each convolution; we show just two each for conciseness. By removing the branches from the loops, our speed increases by another 3.17×.

when the user is given full control of the scheduling.

Code 3.4 shows the cascade application after both convolutions are fully unrolled. Instead of the reductions being two loops, the reductions consist of nine updates to the value. The `unroll` scheduling primitive takes an optional parameter to unroll a given number of iterations; otherwise it defaults to unrolling all iterations. Note that in the Halide schedule, the `unroll` is performed on the `update()` stage of each Func. This refers to the algorithm's accumulation statements on lines 20 and 25 of Code 3.1 rather than the initializations. This optimization leads to another 3.17× reduction in execution time.

`vectorize()` and `parallel()`

`vectorize` and `parallel` increase the parallel execution of the generated code. With these scheduling primitives the code generator adds the necessary decorators to the generated C++ code. The Halide schedule by default has all iteration loops run serially. By adding these scheduling primitives,

```
// Schedule 4: loop parallelism
const int vec = 16;
output
  .store_root()
  .compute_root()
  .tile(x,y, xo,yo, xi,yi, 512,32)
  // perform vector instructions
  .vectorize(xi, vec)
  // threads run tiles in parallel
  .parallel(yo);

conv2.update()
  .unroll(r2.x).unroll(r2.y);

conv1_norm
  .store_at(output, xo)
  .compute_at(output, xo)
  // perform vector instructions
  .vectorize(x, vec);

conv1.update()
  .unroll(r.x).unroll(r.y);

kernel
  .store_at(output, xo)
  .compute_at(output, xo);

// Manually-tuned time: 2.96118ms
```

```
parallel-for yo = 0 to img_y_tiles: // in parallel
  for xo = 0 to img_x_tiles:
    allocate kernel[int32 * 9];
      kernel = ...
    allocate conv1_norm[uint16 * 514 * 34];
    for yi = 0 to 34:
      for xi = 0 to 32:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv1[uint16 * 16];
        conv1[x16] = x16(0); // vector instructions
        conv1[x16] += input[x16](x,y)
                      * x16(kernel(0,0));
        ...
        conv1[x16] += input[x16](x+2,y+2)
                      * x16(kernel(2,2));
        conv1_norm[x16](x,y) = conv1[x16] / x16(16);
    for yi = 0 to 32:
      for xi = 0 to 32:
        int x = xo*512 + xi; int y = yo*32 + yi;
        allocate conv2[uint16 * 16];
        conv2[x16] = x16(0); // vector instructions
        conv2[x16] += conv1_norm[x16](x,y)
                      * x16(kernel(0,0));
        ...
        conv2[x16] += conv1_norm[x16](x+2,y+2)
                      * x16(kernel(2,2));
        output[x16](x,y) = conv2[x16] / x16(16);
```

Code 3.5: Our final schedule (on the left) adds in vector instructions for the convolutions, as well as sets the tiles to run in parallel on separate threads. This dramatically increases the parallel computation, leading to an increase of speed by another 42×. Overall, 1230× faster than the original schedule in Code 3.2.

the loops are modified. `vectorize` replaces several iterations of the specified loop and runs them using SIMD instructions. These instructions work on vectors of data allowing the CPU arithmetic units to execute multiple iterations simultaneously. Vectorizing works best with inner loops so that spatial locality can exploit cache lines prefetching data stored adjacent to each other. LLVM again will attempt to vectorize the code, but we disable this to make the contribution of vectorization more evident.

Halide's `parallel` scheduling primitive instructs the generated code to assign different iterations of the loop to different threads on the machine. Each thread runs different iterations simultaneously leading to better execution times. It is best to assign the outer loops to different threads to ensure that the threads do not collide when trying to access the same memory locations.

Code 3.5 shows how we can use these scheduling primitives on the cascade application. The vectorization takes a vector width, which we use as 16 words in this instance. Each of the computed blocks are vectorized, leading to all of the inline computation also being vectorized, as seen on the loopnest on the right. The casts to vectors are shown with `x16()` while vector memory accesses

are shown with `[x16]`. The vectorized execution decreases the innermost loop iterations from 512 to 32. Furthermore, the `conv1` and `conv2` allocations for accumulation are expanded to vectors to accommodate the vector execution. The Halide `parallel` primitive decorates the outermost loop to be run in parallel, which does not change the generated loopnest significantly, but helps the execution time dramatically. By increasing the parallelism in our code, the code runs another 42× faster. Figure 3.1 shows the progress of execution time by adding Halide scheduling. Overall, the scheduling increases the execution time by 1230× over the default Halide schedule.

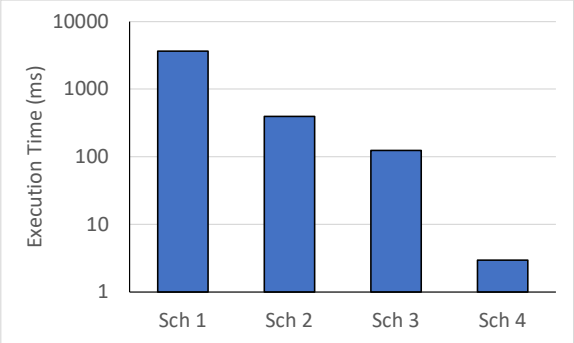


Figure 3.1: Execution time for the sequence of Halide schedules. On a log scale, the execution time consistently decreases as proper scheduling is added.

3.4 Scheduling to FPGAs

Halide's scheduling targets include CPUs, GPUs, and even the Hexagon DSP. However, no back-end for FPGAs exist in the mainline branch of Halide. Several research projects [57,76,92,106] extend the Halide compiler to add FPGAs as a back-end target. Halide HLS [76], written by Jing Pu, brings line-buffered image processing pipelines to Xilinx FPGAs by generating HLS C for Vivado HLS. Standard Halide scheduling primitives are used, but interpreted in a similar style to HLS. Notably, in HLS the unroll primitive is used for hardware duplication due to the semantics of translating HLS C into hardware. Our work builds upon Jing's work in Halide HLS, so we focus on the scheduling primitives added in that work. Below, we go through newly introduced scheduling primitives, as well as how existing primitives perform differently when targeting FPGAs.

`accelerate()`

`accelerate` is a new scheduling primitive that creates the overall accelerator interface in a Halide application. Within the overall application, the hardware accelerator runs a subset of the entire application. This scheduling primitive specifies which Funcs and loops are in the accelerator. The

scheduling primitive is: `accelerate(input_funcs, cycle_var, block_var)`. The scheduling primitive is performed on the output Func for the FPGA accelerator. The output Func is a data stream, while `input_funcs` is a list of Funcs that are streamed into the accelerator. `cycle_var` refers to the inner variable, corresponding to a single cycle of accelerator execution. `block_var` denotes the loop level outside the bounds of the accelerator. Many times the input images are too large to fit in the accelerator's memory. Therefore, the input image is tiled and a single tile is streamed into the accelerator on each execution. To complete the entire application, the host runs the accelerator on each tile. `block_var` shows which output loop variable represents a single execution of the accelerator.

`split()`, `reorder()`, `tile()`, and `bound()`

Similar to CPU execution, FPGAs need scheduling to split loops and determine the loop order. FPGAs benefit from memories where the size is statically known. `split` and `tile` set the inner loop to a static size which helps create static buffers. Similarly, `bound` sets the expected iteration range of an index variable for the input or output. All of these variables help to create memories and iteration counts of a fixed size. `reorder` rearranges the loop variables to ensure that loop variables determined at runtime, such as the number of tiles, are placed outside the bounds of the accelerator.

`store_at()`, `compute_at()`, and `linebuffer()`

`store_at` and `compute_at` create memories for CPU implementations. For image processing applications, line buffers efficiently buffer data between computation kernels as described in Section 2.1. Line buffers use far less memory than a complete tile due to the smaller working set size during execution. Furthermore, shift registers and buffer banking for multiple consumers increase the bandwidth of a memory. These lead to less of a concern for the loop levels. Instead, we can reuse all values sent into the accelerator by using a `store_at` loop level of the full accelerated tile; and we can populate the memory on every cycle, meaning we `compute_at` the innermost loop. Since these two options are so common for buffering memories, the `linebuffer` scheduling primitive was introduced by Halide HLS as syntactic sugar for these options for store and compute level.

`fifo_depth()`

`fifo_depth` creates a fixed-length FIFO between a producer-consumer pair. The user must create FIFOs when reconvergent data streams require additional buffering. This scheduling pass informs the code generator to create additional buffering on the FPGA. Halide HLS creates streams using ready-valid interfaces, but the application deadlocks without the appropriate FIFO insertion when streams reconverge with different delays. The user is expected in Halide HLS to calculate this FIFO depth manually, but integer linear programming could also calculate the necessary delay [40].

`in()`

`in` is a scheduling primitive useful for creating memory hierarchies. Accelerators typically have a memory hierarchy where subsets of data are copied from one memory to another. The FPGA accelerator designs in Interstellar [106] use explicit copies between push memories. Halide provides the `in` scheduling primitive to create these duplicate copies of data. These copies perform no computation, but provide the user with a way to represent the memory hierarchy transfers that exist in the hardware accelerator.

`unroll()`

`unroll` increases parallelism using hardware duplication. Unrolling a loop reduces the number of iterations by duplicating the body of the loop along with advancing the index variable. For an HLS-style interpretation of the program, this results in more computation hardware in the mapped application. `unroll` is thus used to increase the parallelism of the application to make the execution faster by using more resources. Unrolling computation also changes the memory bandwidth needed to serve the computation kernel hardware. Halide HLS creates additional memory streams to facilitate this increased traffic. Memory ports are shared whenever multiple consumers need the same data, such as for overlapping stencils. This effectively allows for all aspects of the memory, computation, and network to be duplicated to ensure that parallel hardware does not bottleneck at any particular resource.

We use `unroll` to increase the parallelism of the application. However, note that for FPGAs there is no interpretation of `vectorize` or `parallel`. Their absence is due to no SIMD processing elements or parallel threads on the FPGA. Instead, `unroll` provides all means of parallelism by duplicating compute hardware. Unrolling all hardware components by n leads to hardware that executes $n\times$ faster. We use similar HLS semantics as we extend Halide lowering to CGRA hardware.

3.5 Scheduling to CGRAs

We extended the Halide compiler further to target our custom Amber CGRA, whose hardware features are described in Section 2.3. Scheduling for CGRAs is fairly similar to FPGAs, since both are reconfigurable accelerators. We take a similar approach to Halide HLS [76] by using HLS semantics. This means that unrolling loops leads to hardware duplication and data is streamed between compute kernels. CGRAs differ from FPGAs due to higher-level compute blocks and specialized memory configurations. Many of these complexities lead to a specialized mapper. In the Halide scheduling, we assist the back-end compiler by tagging memories with their intended memory modules to simplify mapping. We also clean up the scheduling semantics by separating the accelerator definition into two scheduling primitives. Code 3.6 shows how all of these scheduling

```
/* CGRA Schedule */
Var xii, xio, yii, yio, xi, xo, yi, yo;
output.bound(x, 0, outImgSizeX)
      .bound(y, 0, outImgSizeY);

// Accelerate 360x360 tiles in GLB
hw_output.in().compute_root()
    .tile(x,y, xo,yo, xi,yi, 360,360)
    .hw_accelerate(xi, xo);

// Send 60x60 tiles to CGRA fabric
hw_output
    .tile(x,y, xio,yio, xii,yii, 60,60)
    .compute_at(hw_output.in(), xo)
    .store_in(MemoryType::GLB);

// conv2 kernel with unrolled reduction
conv2_norm.compute_at(hw_output, xio);
conv2.update()
    .unroll(r2.x).unroll(r2.y);

// conv1 kernel with unrolled reduction
conv1_norm.compute_at(hw_output, xio);
conv1.update()
    .unroll(r.x).unroll(r.y);

// MemoryTile and GLB for input stream
hw_input.in().in()
    .compute_at(hw_output, xio);
hw_input.in()
    .compute_at(hw_output.in(), xo)
    .store_in(MemoryType::GLB);
hw_input.accelerator_input();

kernel.compute_at(hw_output, xio);
```

Code 3.6: An example CGRA schedule for Code 3.1. This algorithm runs at 1 pixel/cycle with two buffers: one for each 3×3 convolution. A memory hierarchy is created for both the `hw_input` and `output`. Tiled execution sends 360×360 tiles to the GLB, which in turn sends 60×60 tiles to the CGRA fabric.

primitives create a CGRA schedule for the cascade application. Below we describe how each of the scheduling primitives are used for CGRA scheduling.

`hw_accelerate()`, `stream_to_accelerator()`, and `accelerator_input()`

`hw_accelerate`, `stream_to_accelerator`, and `accelerator_input` denote the output and input Funcs for the hardware accelerator. In this way, hardware accelerator input and output bounds can be defined within applications, with the rest of the application run on the host CPU. `hw_accelerate` denotes the output stream of the application while `stream_to_accelerator` identifies each of the input streams. As an alternative for specifying accelerator inputs, `accelerator_input` is a more precise way of defining inputs, especially when there is a complex memory hierarchy. `accelerator_input`

denotes the boundary Func on the host side before any memory copies to the accelerator. However, this primitive requires any memory copies to be specified by the user. `stream_to_accelerator` is a simple primitive for small test cases that bakes in an assumption for the memory hierarchy: a single copy from the host to the accelerator.

These scheduling primitives for defining accelerators have many similarities to their corresponding FPGA definitions, but with some important differences. `hw_accelerate` is very similar to the FPGA `accelerate` call, except it does not include the input Funcs or a cycle variable. The innermost loop for each loopnest is assumed to iterate for each cycle. If an inner loop should be entirely executed in a single cycle, the user should unroll that loop. Another change is removing the list of input Funcs. Instead, `stream_to_accelerator` or `accelerator_input` is used on each input Func. One reason for this change is that Halide's scheduling language tends to be written from output to input. The scheduling calls then specify the consumer Func for storage and compute loop levels. `accelerate` broke from this convention by specifying *producer* Funcs. The new `stream_to_accelerator` and `accelerator_input` adhere to this convention by using the scheduling primitives directly on the producer Funcs.

`store_at()` and `compute_at()`

Both `store_at` and `compute_at` are again used in tandem to define intermediate memories. We schedule our application memories that consist of line buffers and double buffers, as introduced in Chapter 2. Our target CGRA contains memory primitives that implement efficient line buffers with adequate memory bandwidth, as well as double buffers for DNN applications. Instead of specifying the exact memory scheduling, we leave this memory mapping step to a separate compiler stage.

In order to leave this memory scheduling decision for later, we require that the user schedule memories in serial order. This means that in Halide, the `compute_at` variable is always the same variable as that provided for `store_at`. This generates a schedule where each Func is computed with a serial loopnest with no interleaving. This corresponds to `compute_at(block_var).store_at(block_var)`, where `block_var` is the accelerator loop variable for a block run on the CGRA fabric. Later during mapping, the loops are analyzed to determine if loop fusion or loop pipelining should happen. In both cases, the loopnests are later scheduled such that efficient hardware implementations are used.

`in()` and `store_in()`

Our target CGRA consists of multiple levels of hierarchy, including the host DRAM, the accelerator's global buffer, memory tiles on the CGRA fabric, and ponds within processing elements. There are several scheduling primitives used for creating these memory hierarchies. `in` is a scheduling primitive that generates a copy instruction, which is the computation that occurs for transfers in the memory hierarchy. Using `in`, the user can create multiple copies of data as it travels through the memory hierarchy from the host to the CGRA fabric.

`store_in` helps the memory mapper determine what type of physical memory to implement for a Func. Our CGRA consists of different memory primitives of different sizes, as described in Section 2.3. The **GLB** is the highest level that communicates between the **Host** memory and CGRA fabric. The CGRA fabric consists of **MemoryTile**s that store values on a wide SRAM. Each processing element has a smaller register file, known as a **Pond**. Finally, a special implementation of the memory tile is a **ROM**, where the addressing is controlled by external calculation. These bolded memory types are added to Halide's existing set of memory types to assist the subsequent mapping phase to these memory primitives.

`unroll()`

Similar to the FPGA, we use `unroll` to create parallelism for the CGRA. `unroll` informs the compiler to duplicate the hardware to increase the rate of a component. Small reduction loops are typically unrolled to allow the hardware to run at a single pixel per cycle. The innermost loop can then be unrolled even more to increase the rate further. It is important to unroll each kernel to match the rates. Without proper rate matching of memory transfers, initialization, and kernel computation, a scheduled application will be slowed to the constricting rate. For DNNs, unrolling increases the number of multiply-accumulate operators that are generated for the compute kernel. Creating a grid of multipliers is as simple as reordering two loops to become the innermost loops, and then unrolling the innermost reduction loops. This provides a large computation kernel that can map to the CGRA.

For our CGRA, neither `vectorize` or `parallel` is used. However, other CGRAs use vector processors, so mapping to them would have a use case for the `vectorize` scheduling primitive. `parallel` could also be implemented at the host-CGRA interface to denote multiple iterations of an application running on a single CGRA. This can be useful since smaller applications with less unrolling have better PnR and pipelining results. Implementing parallel instances of a smaller application could be another way to achieve parallelism and hardware utilization on a CGRA.

`compute_share()`

Our applications can normally be mapped to efficient implementations with the scheduling primitives above. One exception, though, is when compute kernels have low utilization. In these instances, creating hardware that is exclusively used by a single compute kernel may not be an efficient use of valuable compute tiles. Instead, it would be best if we could share compute tiles among several underutilized compute kernels.

`compute_share` is a new scheduling primitive that we introduced to provide an efficient implementation for underutilized compute kernels. This scheduling primitive allows multiple exact matching compute kernels to share the same compute tiles. We go into more detail about the implementation and compiler implications in Chapter 5.

```
output.bound(x, 0, 368 * 16);
output.bound(y, 0, 196 * 20);
hw_output.in().compute_root();
hw_output.in()
  .tile(x, y, xo, yo, xi, yi, 368 * 16, 196 * 20)  // GLB size
  .reorder(xi, yi, xo, yo)
  .hw_accelerate(xi, xo);
hw_output.in()
  .unroll(xi, myunroll, TailStrategy::RoundUp);
hw_output
  .tile(x, y, xo, yo, xi, yi, 368, 196)  // tile size
  .reorder(xi, yi, xo, yo);
hw_output.compute_at(hw_output.in(), xo);
hw_output.store_in(MemoryType::GLB);
hw_output.unroll(xi, myunroll, TailStrategy::RoundUp);

cim.compute_at(hw_output, xo).unroll(x, myunroll, TailStrategy::RoundUp);
lgxx.compute_at(hw_output, xo);
lgyy.compute_at(hw_output, xo);
lgxy.compute_at(hw_output, xo);
lgxx.update().unroll(box.x).unroll(box.y).unroll(x, myunroll, TailStrategy::RoundUp);
lgyy.update().unroll(box.x).unroll(box.y).unroll(x, myunroll, TailStrategy::RoundUp);
lgxy.update().unroll(box.x).unroll(box.y).unroll(x, myunroll, TailStrategy::RoundUp);
lgxx.unroll(x, unroll, TailStrategy::RoundUp);
lgyy.unroll(x, unroll, TailStrategy::RoundUp);
lgxy.unroll(x, unroll, TailStrategy::RoundUp);
lxx.compute_at(hw_output, xo).unroll(x, myunroll, TailStrategy::RoundUp);
lyy.compute_at(hw_output, xo).unroll(x, myunroll, TailStrategy::RoundUp);
lxy.compute_at(hw_output, xo).unroll(x, myunroll, TailStrategy::RoundUp);

kernel_x.compute_at(hw_output, xo);
kernel_y.compute_at(hw_output, xo);
kernel_x.unroll(x).unroll(y).unroll(x, myunroll, TailStrategy::RoundUp);
kernel_y.unroll(x).unroll(y).unroll(x, myunroll, TailStrategy::RoundUp);
kernel_x.compute_at(hw_output, xo);
kernel_y.compute_at(hw_output, xo);
kernel_x.unroll(x).unroll(y).unroll(x, myunroll, TailStrategy::RoundUp);
kernel_y.unroll(x).unroll(y).unroll(x, myunroll, TailStrategy::RoundUp);

grad_x_unclamp.compute_at(hw_output, xo);
grad_y_unclamp.compute_at(hw_output, xo);
grad_x_unclamp.update().unroll(r.x).unroll(r.y).unroll(x, myunroll, TailStrategy::RoundUp);
grad_y_unclamp.update().unroll(r.x).unroll(r.y).unroll(x, myunroll, TailStrategy::RoundUp);
grad_x_unclamp.unroll(x, myunroll, TailStrategy::RoundUp);
grad_y_unclamp.unroll(x, myunroll, TailStrategy::RoundUp);
gray.compute_at(hw_output, xo).unroll(x, myunroll, TailStrategy::RoundUp);

hw_input.in().in().compute_at(hw_output, xo); // represents the mem tile
hw_input.in().in()
  .unroll(c)
  .unroll(x, myunroll, TailStrategy::RoundUp);
hw_input.in().compute_at(hw_output.in(), xo); // represents the glb level
hw_input.in().store_in(MemoryType::GLB);
hw_input.in().unroll(c)
  .unroll(x, myunroll, TailStrategy::RoundUp);
hw_input.compute_root()
  .accelerator_input();
```

Code 3.7: Full original schedule for Harris with a memory hierarchy, unrolled rate, and buffers.

3.6 Declarative Scheduling

While the above scheduling primitives all work for describing a schedule with a hardware target, some of the scheduling is not intuitive. Creating a memory hierarchy with tiled loops is cumbersome, and is a difficult schedule to read. Code 3.7 shows the schedule for the Harris application. Due to the large number of buffers, there is a lot of duplication needed to create each of the buffered intermediates. Additionally, each of the streams are unrolled to match the rates.

```
// Declarative Schedule:
// Create three level memory hierarchy with iteration_order (and set rate)
auto level1 = IterLevel("CGRA", {{x, 368}, {y, 196}});  // tile size
auto level2 = IterLevel("GLB",  {{x, 1}, {y, 1}});  // No GLB-specific tiling
auto level3 = IterLevel("host", {{x, 16}, {y, 20}});  // host tiling
hw_output.output_rate(myunroll)
  .iteration_order({level1, level2, level3});

// Specify compute variables for memory hierarchy
auto x0 = Var("x0");
auto compute_cgra = hw_output.get_looplevel("CGRA", x0);
auto compute_glb = hw_output.get_looplevel("GLB", x0);
auto compute_host = LoopLevel::root();

// Create hardware accelerator
hw_output.get_memory_level("GLB")
  .hw_accelerate(Var("x1"), x0);

// Create memories (and set rate)
hw_output.get_memory_level("GLB").output_rate(myunroll)
  .create_memories({cim, lgxx, lgyy, lgxy, lxx, lyy, lxy,
                    grad_x_unclamp, grad_y_unclamp, gray}, compute_cgra);
kernel_x.compute_at(compute_cgra).unroll(x).unroll(y)
        .unroll(x, myunroll, TailStrategy::RoundUp);
kernel_y.compute_at(compute_cgra).unroll(x).unroll(y)
        .unroll(x, myunroll, TailStrategy::RoundUp);

// Stream input to accelerator (and set rate)
hw_input.output_rate(myunroll)
  .stream_to_accelerator({"CGRA", "GLB", "host"},
                         {compute_cgra, compute_glb, compute_host});
hw_input.in().get_memory_level("CGRA").unroll(c);
hw_input.get_memory_level("GLB").unroll(c);
```

Code 3.8: Full declarative schedule for Harris. This schedule is equivalent to Code 3.7, but with new syntactic sugar. These new scheduling primitives make assumptions common for the CGRA.

To provide an alternative to this scheduling, we also developed some syntactic sugar to provide these schedules with a more readable syntax. These new scheduling primitives do not implement anything new, but simply call the above scheduling primitives internally. One motivation for this new scheduling scheme is to specify properties of the hardware in a more declarative way. The scheduling primitive arguments are exactly the target loop iterations, loop ordering, and output rate that the user declares. Code 3.8 shows this new syntax, which implements the same schedule as

```
// Define iteration levels in the memory hierarchy
auto mem_lvl = IterLevel("CGRA", {{x, 368},{y, 196}});  // tile size
auto glb_lvl = IterLevel("GLB",  {{x, 1},   {y, 1}});    // no specific GLB tiling
auto hst_lvl = IterLevel("host", {{x, 16}, {y, 20}});   // host tiling
// Set the output memory hierarchy order
hw_output.iteration_order({mem_lvl, glb_lvl, hst_lvl});

// Get handles for different loop levels
Var x0 = Var("x0");  // Internal variable x in compiler
auto x_cgra = hw_output.get_looplevel("CGRA", x0);
auto x_glb  = hw_output.get_looplevel("GLB", x0);

// Use the memory level to define the accelerator
Var x1 = Var("x1");
hw_output.get_memory_level("GLB").hw_accelerate(x1, x0);
```

Code 3.9: Define a memory hierarchy using new declarative memory scheduling. `IterLevel` and `iteration_order` defines the memory hierarchy on the output. `get_looplevel` and `get_memory_level` provide handles to variables and Funcs for further scheduling.

Code 3.7. Next, we go into detail for each of the new scheduling primitives by going through each of the components of this new Harris schedule.

`IterLevel()` and `iteration_order()`

One of the unique elements of hardware accelerators is the memory hierarchy. We see in both GPUs and hardware accelerators that an application needs to take values from the CPU host memory and copy those values into a software-managed memory hierarchy. One common hierarchy structure has memory levels each with their own number of iterations, and each larger memory level includes the values of the lower memories.

`IterLevel` is a way of defining an iteration level in a memory hierarchy. It consists of a name (so that the generated loops have more understandable names), and a sequence of iteration loops. The loops are listed from innermost to outermost level, and each loop variable is bundled with its size at that level. Note that a variable can be used multiple times to specify multiple loop splits. Multiple iteration levels are created, and then they can be ordered on the full computation using `iteration_order`. The combination of `IterLevel` and `iteration_order` provides an alternative to `split`, `reorder`, and `in`. Code 3.9 shows how a set of ordered loops can be defined with these new scheduling primitives.

`get_memory_level()` and `get_looplevel()`

With the creation of loops using a new `iteration_level` scheduling primitive, we have created new loops and memory levels. However, we also need a way of scheduling with these newly generated loops and memory levels. Therefore, we introduced two scheduling primitives to provide handles

to those components. `get_looplevel` provides a handle to the new loop variable at a defined loop level name. `get_memory_level` provides a handle to a Func (memory level) that is created with `IterLevel`. Code 3.9 shows how these scheduling primitives are used.

`stream_to_accelerator()`

The input memory hierarchy is similar, but our original scheduling also used `accelerator_input` to specify which memory level involved the transfer from host to accelerator. For our new declarative syntax, we again use the iteration levels defined using `IterLevel`. The `stream_to_accelerator` function is then redefined with two new parameters: an input array of memory level names, and then the equivalent output loop levels used as each of the compute levels. This syntactic sugar takes the iteration levels and loop names to create the `compute_at` statements needed by Halide to create a memory hierarchy. We show how to use the new `stream_to_accelerator` to define the accelerator interface below:

```
hw_output
  .iteration_order({mem_lvl, glb_lvl, host_lvl});
hw_output.get_memory_level("GLB")
  .hw_accelerate(x1, x0);
hw_input
  .stream_to_accelerator({"CGRA", "GLB", "host"},
                         {x_cgra, x_glb, x_host});
```

`create_memories()`

Our above syntactic sugar focuses on the memory hierarchy. Another component of hardware schedules is the repetition of key scheduling arguments. An example of this repetition is in the creation of memories between compute kernels. Each of these memories uses the same store and compute level. To reduce repetition, we introduce `create_memories` to specify a list of Funcs that should have a memory, and then a loop level where they should be created. For example, in the Harris application, we have multiple intermediates (seen as `compute_at` in Code 3.7 on lines 17 to 46) that can be scheduled as CGRA memories using:

```
hw_output_glb
  .create_memories({cim, lgxx, lgyy, lgxy, lxx, lyy, lxy,
        grad_x_unclamp, grad_y_unclamp, gray}, compute_cgra;
```

output_rate()

In addition to the repetition of memories using `store_at`, we also see lots of repetition of `unroll`. Unrolling duplicates compute kernels and IO bandwidth. For image processing applications, we typically want to match all of the rates so each of the compute kernels have a similar compute utilization. Using the original scheduling primitives, the entire compute pipeline repeats the same `unroll` primitive for every memory.

Instead, using the new declarative scheduling primitive, we can specify an output rate. We use the output rate in conjunction with `create_memories` to specify that each should be unrolled by the same amount. Similarly, we can apply an `output_rate` to an output iteration order and input streaming declaration to increase the bandwidth of these interfaces. Below is an example:

```
hw_output.output_rate(8)
  .iteration_order({mem_lvl, glb_lvl, host_lvl});
hw_output.get_memory_level("GLB").output_rate(8)
  .create_memories({blur, blur_out}, hw_output_cgra);
hw_input.output_rate(8)
  .stream_to_accelerator({"CGRA", "GLB", "host"},
                         {x_cgra, x_glb, x_host});
```

Note that all previous scheduling primitives have fairly straightforward Halide scheduling using the original primitives. However, achieving a set rate by matching unrolling and loop iteration counts is a little more difficult. Iteration counts are not expected to be exactly equal due to convolution kernels shrinking produced images at the edges. Furthermore, the rate of a DNN layer is a lot harder to quantify due to the bursty output of the tiles during the readout phase. Currently, you can set the rate for an image processing application. This will properly unroll all of the compute kernels and memory transfers for an application. For DNNs, output rate is harder to define as a single integer, so `output_rate` does not work for these applications. This scheduling primitive is a work in progress, but is the start of redefining Halide scheduling for hardware accelerators in a declarative way.

Generating Halide Schedules

As discussed above, the declarative scheduling primitives provide a new interface for the user, but in reality are simply semantic sugar for the original scheduling primitives. When the Halide front-end is given these new scheduling primitives, internally the compiler calls the corresponding original scheduling primitives.

For example, `IterLevel` specifies the iteration loops at different levels. These structs are then provided to the output Func. This results in a tiling of the output Func using the `IterLevels` provided. To calculate the values of the original tiling call, the compiler takes the product of variables

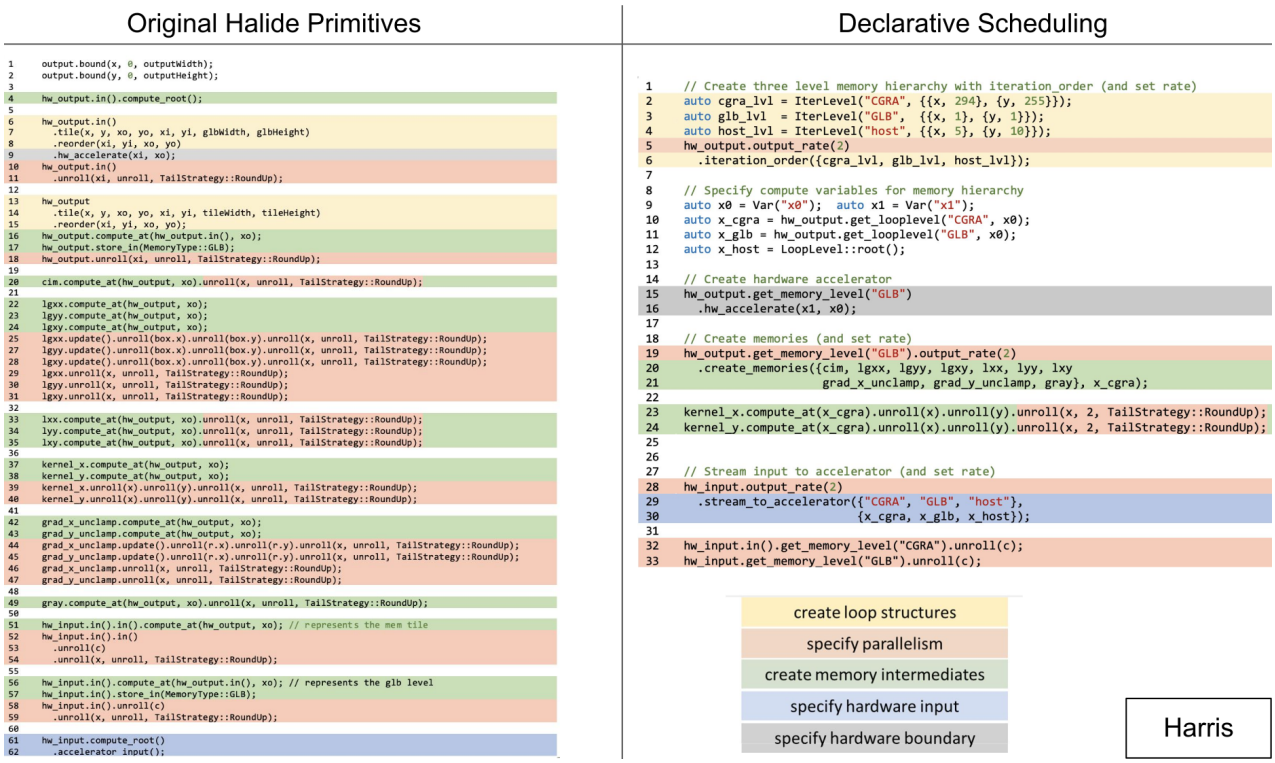


Figure 3.2: Declarative schedule for Harris application. Left uses original Halide primitives while the right uses the new declarative scheduling. The scheduling lines are classified using different colors to show that concepts are further grouped with our new declarative scheduling.

of the same name (all x's multiplied together, and all y's multiplied together). In our example in Code 3.8, we have three `IterLevel`s, which generate multiple `in`, `compute_at`, and `store_in` calls for our memory hierarchy. The corresponding memory hierarchy created on the input side is created by the new `stream_to_accelerator`. The memory hierarchy levels are given corresponding names, and the compute level variables needed to create these original scheduling calls.

The next set of scheduling calls generate the intermediate memories and compute kernel parallelism. `create_memories` and `output_rate` are provided the necessary parameters to generate the original scheduling. `store_at` and `compute_at` use the provided memory level, while `unroll` uses the output rate. Each of these scheduling primitives are applied to each Func in the list provided `create_memories`. In all, the declarative scheduling primitives have strong correspondence to the original scheduling primitives, but provide a more compact and direct interface for scheduling to our CGRA.

Full Application: Harris

With these scheduling primitives, we can schedule a full application. Code 3.7 and Code 3.8 show two versions of the same Harris schedule: original scheduling primitives and the new declarative schedule. Figure 3.2 shows the mapping of these scheduling primitives between the two schedules. Notice that the number of lines in the schedule decreases, mainly due to less repetition of memory creation and unrolling. More importantly, I find that the declarative schedule is easier to read and understand.

3.7 Summary

Halide is a high-level language for image and array processing applications known for its ability to generate high-performance code by separating the algorithm from the schedule. Halide schedules invoke loop transformations to optimize code execution by altering memory locality and parallelism. It supports various hardware targets such as CPUs, GPUs, and DSPs. Extensions to Halide have been made to target FPGAs using HLS as an intermediate. We choose to extend the Halide compiler to support reconfigurable accelerator targets more generally, including the Amber CGRA. Our set of scheduling primitives for reconfigurable accelerators refine the scheduling used previously for FPGAs, including the ability to define accelerator scope within an application, defining memory hierarchies, and providing hardware duplication for additional parallelism. With declarative scheduling, we simplify the specification of these hardware schedules with more readable syntactic sugar. These schedules provide the necessary tools for application designers to create efficient application executions on reconfigurable accelerators.

Chapter 4

Unified Buffers: a Memory Abstraction

CGRAs pose a unique challenge for application mapping due to their higher-level hardware primitives. In Section 2.3 we introduced our primary target, the Amber CGRA. This CGRA consists of simple PE tiles that directly map from compute primitives. On the other hand, the memory tile is more complex due to its shared SRAM port for both reads and writes, wide SRAM port that fetches four words at a time, and embedded address generators. These complex memories are commonplace on accelerators, which strive for high performance streaming of large amounts of data. In order to more easily map an application to hardware, we created the **unified buffer abstraction** [59]. Our unified buffer abstraction describes the data movement for every memory element in the application. We found that the description of the memories and address generators were linked, so our abstraction captures the intent of both of these components. Thus, the job of the front-end layer of the compiler is to map the high-level application to a collection of unified buffers. Then separately, a hardware mapper maps from the unified buffers to the custom hardware.

In the previous chapter, we introduced Halide as our front-end language. To lower Halide applications to the unified buffer abstraction, Halide runs optimization passes and a code generator. The Halide passes transform the intermediate representation to fit the unified buffer abstraction. After the code transformations, we can output the unified buffers and computation kernels in a format that is ready to map to our hardware accelerator using our memory mapper called Clockwork [44]. Our group collaborated to design the unified buffer abstraction and compilation scheme, where my contributions focus on the Halide front-end and codegen.

This chapter first specifies the parameters for the unified buffer abstraction. Then, we explain how Halide is used to lower the application memories to the unified buffer abstraction. And lastly, we describe the code generation to create the unified buffers that can be mapped using Clockwork.

4.1 System Overview

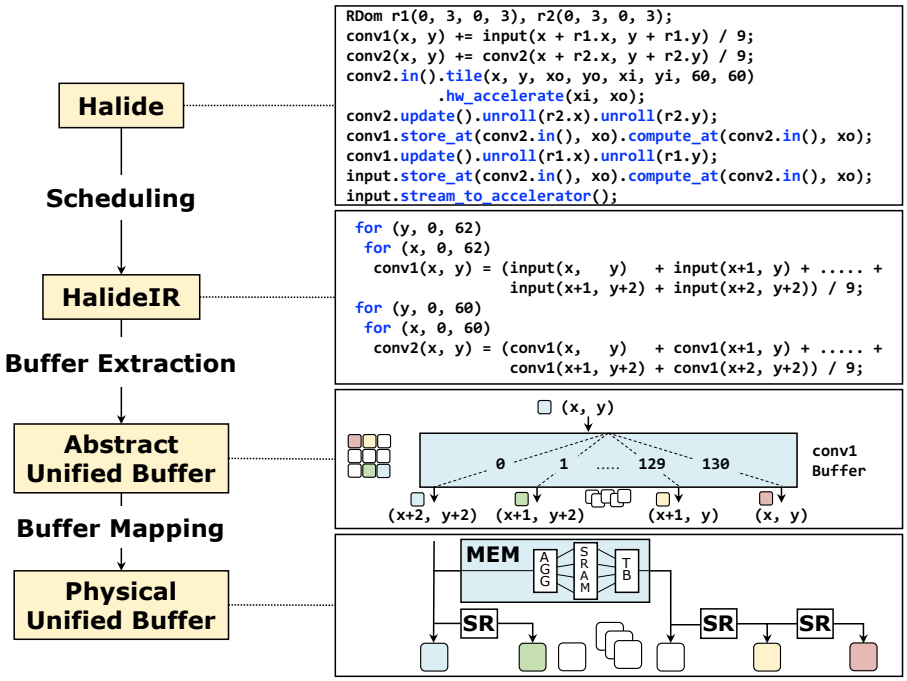


Figure 4.1: The three compiler steps for the *cascade* example application. Scheduling generates tiled loops, from which buffer extraction emits the `conv1` unified buffer. This is mapped to shift registers (SR) and our optimized memory tile (MEM) with aggregator (AGG) and transpose buffer (TB).

Our compiler system uses a modified Halide compiler as the first stage in translating Halide code to the unified buffer abstraction. This modified Halide front-end parses application code, then schedules and transforms it using an internal imperative intermediate representation known as HalideIR, and finally generates a hardware representation of the application. The hardware representation consists of two files: one representing the computation kernels, and one representing the buffers. Figure 4.1 shows these steps for a small sample application.

During application compilation, we manipulate and modify HalideIR to create a version of the application that can be readily converted to hardware primitives. The desired intermediate representation for our compute kernels in HalideIR, right before conversion into hardware primitives, is a compute pattern commonly used in high-level synthesis (HLS). In this interpretation of HalideIR, for-loops are eventually mapped to control flow while mathematical operations become distinct hardware computation units. Whereas CPUs run through a block of statements in sequential order, our desired intermediate representation (IR) creates a hardware processing element for each statement. A block of n statements can thus be performed in a single cycle. Due to these semantics, common loop transformations done on software code have hardware interpretations, as seen in HLS pragmas.

Most notably, loop unrolling reduces loop iterations and causes hardware duplication in HLS. We see how Halide scheduling affects the generated hardware in Figure 4.3.

After representing the Halide application in our IR, we need to extract unified buffers from our application. To lower the IR needed for Clockwork, several Halide passes are used to analyze and modify the loopnests as described in Section 4.4. The codegen step of the compiler takes the final version of HalideIR and generates separate memory and compute files for Clockwork. Here, the memory dependencies are organized by buffer name and the unified buffer parameters are extracted and calculated.

The unified buffer is a critical interface between the application representation of memory operations and mapping these functions onto the hardware accelerator. After defining the Halide application, we only need to focus on representing the application with unified buffers. After that, Clockwork takes these unified buffer properties and optimizes them by performing loop fusion and scheduling to determine an absolute order of the application execution. Clockwork then performs memory mapping by taking the unified buffers and mapping them to configurations on the hardware accelerator. The final steps for mapping to the Amber CGRA are described in Section 6.1, where we describe the full compiler system. Since the unified buffer abstraction provides an important division between the front-end and back-end of the compiler, we start our description with properties of the unified buffer abstraction.

4.2 Unified Buffer Abstraction

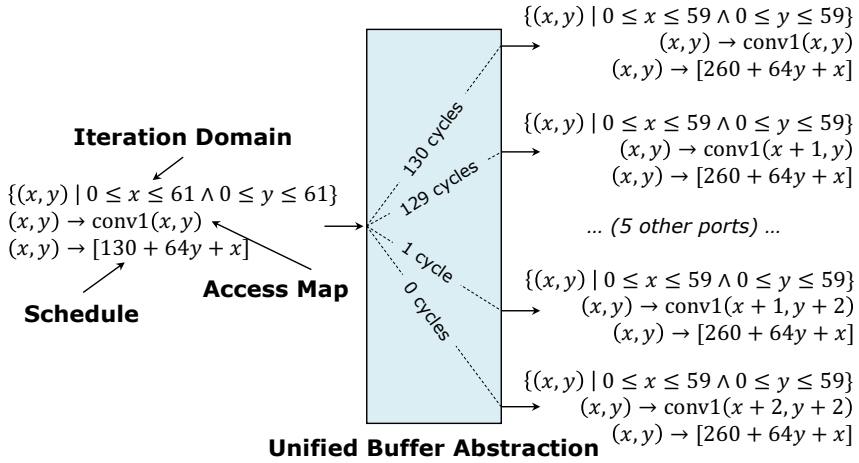


Figure 4.2: The unified buffer specifies the data movement between the first 3×3 compute kernel `conv1` (shown above) and the second 3×3 compute kernel `conv2` as illustrated in Figure 4.1. Each port is defined by a polyhedral iteration domain and access map that describe the data written to/read from the buffer. The schedule describes when those values arrive at each port.

The Buffets paper [73] describes our target accelerators as explicit decoupled data orchestration (EDDO). These memories take streams of data, store a working set of values, and then after some latency, output a possibly reordered stream of the data. The goal of the unified buffer abstraction is to fully represent this type of memory—representing the storage and controllers needed to properly depict the data as it flows through the hardware over time. The abstraction encapsulates both the memory space needed as well as the addressor configurations necessary to control where and when data is written/read. Grouping these components together is critical for CGRAs and other accelerators where, for efficiency, the memory components have complex addressing and scheduling bundled together. Memory tiles in the Amber CGRA couple the SRAMs and addresses generators for efficiency, so lowering to this high-level hardware primitive requires a similarly high-level memory abstraction. More broadly, Buffets [73], Plasticine [75], and other efficient accelerators also create higher-level memory structures. Therefore, we find that a unified buffer abstraction is a concept that can be used for compiling to many of the image processing and machine learning accelerators designed recently.

Figure 4.2 shows the unified buffer abstraction for the `conv1` Func. The buffer consists of write ports (on the left) and read ports (on the right). Each port on the buffer has parameters to describe how these streams of data interact with the buffer. The unified buffer abstraction defines these streams using three types of parameters: the iteration domain, which describes the data which needs to be stored; the access functions, which describe the order of data in the stream; and the operation schedule, which describes when the data will be valid. Finally, each of the write and read ports are connected to compute kernels, and in turn, other unified buffers to describe the memory dependencies.

This abstraction provides a nice interface between the application's desired memory function and the implementations of both the compute kernels and the storage implementations. Outside the abstraction are the compute units. The compute operations performed between the buffers do not affect unified buffer parameters, except for the latency of the compute units. Therefore, the only information kept about a compute unit is its latency. Furthermore, knowledge about how hardware implements unified buffers is also not needed for this specification. With this interface, a full application can be separated into its computation kernels and memories. Next, we go into detail about the three unified buffer parameters.

Iteration Domain

The first important property for storing a variable in memory is how many elements exist. The **iteration domain** determines the total number of values for a particular memory. An iteration domain consists of index variables and the range of values for each variable. The iteration domain is represented using multiple index variables to maintain the natural multi-dimensional variables that are used in the application. For example, this means that a color image is specified using spatial

Table 4.1: Unified buffer parameters for the *cascade* application. The cascade application consists of two 3×3 convolutions each with an access map with nine output ports. The memory size of `input` and `output` are 0, resulting in wires; the `hw_input` memory has a capacity of 130 and the `conv1` memory has a capacity of 126.

Buffer	Dir.	Iteration Domain	Access Map	Schedule
`input`	in	$(0 \le x \le 63) \cap (0 \le y \le 63)$	$(x, y) \to \text{input}(x, y)$	$(x, y) \to [64y + x]$
	out	$(0 \le x \le 63) \cap (0 \le y \le 63)$	$(x, y) \to \text{input}(x, y)$	$(x, y) \to [64y + x]$
`hw_input`	in	$(0 \le x \le 63) \cap (0 \le y \le 63)$	$(x, y) \to \text{hw\_input}(x, y)$	$(x, y) \to [64y + x]$
	out	$(0 \le x \le 61) \cap (0 \le y \le 61)$	$(x, y) \to \text{hw\_input}(x, y)$ $(x, y) \to \text{hw\_input}(x + 1, y)$ *... (5 other ports) ...* $(x, y) \to \text{hw\_input}(x + 1, y + 2)$ $(x, y) \to \text{hw\_input}(x + 2, y + 2)$	$(x, y) \to [130 + 64y + x]$
`conv1`	in	$(0 \le x \le 61) \cap (0 \le y \le 61)$	$(x, y) \to \text{conv1}(x, y)$	$(x, y) \to [130 + 64y + x]$
	out	$(0 \le x \le 59) \cap (0 \le y \le 59)$	$(x, y) \to \text{conv1}(x, y)$ $(x, y) \to \text{conv1}(x + 1, y)$ *... (5 other ports) ...* $(x, y) \to \text{conv1}(x + 1, y + 2)$ $(x, y) \to \text{conv1}(x + 2, y + 2)$	$(x, y) \to [260 + 64y + x]$
`output`	in	$(0 \le x \le 59) \cap (0 \le y \le 59)$	$(x, y) \to \text{output}(x, y)$	$(x, y) \to [260 + 64y + x]$
	out	$(0 \le x \le 59) \cap (0 \le y \le 59)$	$(x, y) \to \text{output}(x, y)$	$(x, y) \to [260 + 64y + x]$

variables x and y, as well as a color channel, c.

Each variable is specified as a range of values. For images and deep neural networks tensors, these spaces are typically rectangular prisms. However, the concept of iteration domains comes from polyhedral models, where the only limitation of the iteration domain is that it is convex. Being convex means that any linear interpolation between two iteration points is also a point within the iteration domain. In reality, a large fraction of interesting programs fit within the definition of linear programs, and if we make this assumption of linear relations, we can make optimizations in the hardware.

The iteration domain is important to the unified buffer configuration to determine the extent of the iterators. The iteration space determines how many written values exist for each buffer. Table 4.1 shows the iteration domains for each buffer in the cascade application, which performs two 3×3 convolutions. The `hw_input` buffer has an output iteration domain with index variables x and y between 0 and 61, meaning each output port is read $62 \cdot 62 = 3844$ times. Notice that the iteration domain shrinks in later stages of the application due to convolution kernels reducing the defined iteration domain extent based on their kernel size ($e_{consumer} = e_{producer} - (k_{conv} - 1)$).

Access Functions

The multiple streams of data interacting with the unified buffer are usually related in some manner. The **access functions** specify the relative offsets of streams for the output ports in relation to the input ports. An input port on a unified buffer is used to store the result of a computation kernel. Multiple input ports occur when there are multiple writing streams, such as when a memory address is updated several times. An output port on the unified buffer is created whenever a computation kernel uses the buffered intermediate. Multiple output ports are created when several values from the same buffer are used by another memory (or memories). Stencils are a common computation pattern in image processing applications that result in multiple output ports.

The listed relationships between the input and output port map the iteration domain values to the actual indices needed by that port. For example in Table 4.1, the third output port has access function: $(x, y) \rightarrow hw\_input(x + 1, y)$ and iteration domain $x = 0$ to 61 and $y = 0$ to 61. While the iteration domain specifies that there are 62 values in the x dimension, instead of taking the first 62 values in each input row, instead we offset our index by 1, and take values in indices 1 to 62. Each of the access functions map the iteration domain to the actual indices on the input.

The input and output ports that are created for the unified buffer abstraction are logical ports, and do not directly translate to that quantity of physical ports on a physical memory unit. Instead, these ports are later analyzed during the memory mapping phase to reduce the number of physical ports needed to support these logical ports. For a convolution, our memory mapper replaces many of the logical ports with stencil registers to implement the unified buffer abstraction in a more feasible and efficient manner.

The access map is used to select a group of indices for the compute kernel. Table 4.1 shows the access map for each of the unified buffers in the cascade application. The first 3×3 convolution uses nine indices of `hw_input` to calculate `conv1`. Using this access map, when $x = 0$ and $y = 0$ we need the `hw_input` indices from $(0, 0)$ to $(2, 2)$. The size of the access map is later used by Clockwork scheduling to determine how many pieces of data must be buffered before all outputs are available.

Operation Schedule

The final piece of our abstraction defines the order of memory operations on a unified buffer. The **operation schedule** specifies when each value is written to and read out of the memory. The order of memory stores/loads is calculated using the index variables in the iteration domain. The equations that are typically formed by these schedules are affine equations. These affine schedules appear due to the rectangular size of the iteration domain and linear access functions, leading to periodic execution. Inherent to the operation schedule are the loop order and loop tiling. Each index variable is used in the affine operation schedule to specify when loads and stores occur. The index variable with the smallest stride is the innermost loop for the iteration order.

In Table 4.1 we show the final, optimized schedule for each of the unified buffer outputs. The

Table 4.2: Halide code supported by our compiler toolchain. Italicized fields are not supported by Clockwork during memory mapping.

Use case	Halide algorithm code	Support?	Memory data	Write Address	Read Address
Affine indexing	`out(x) = in(2*x) + 5 * x`	Yes	Data dependent	Affine	Affine
Stencil taps	`taps(x) = 1; taps(1) = 2;` `out(x) =` $\sum_{i=0}^{2}$ `taps(i)*in(x+i)`	Yes	Constant	Constant	Constant
Lookup table	`lut(x) = max(512, x*x);` `out(x) = lut(in(x))`	Yes	Constant	Constant	Data dependent
Non-affine indexing	`out(x) = in(x*y)`	No	Data dependent	Affine	*Non-affine*
Histogramming	`out(bin(x)) += 1`	No	Data dependent	*Data dependent*	*Data dependent*
Recursion	`fib(x) = fib(x-1) + fib(x-2)`	No	Affine	*Affine*	*Affine*

output of $x = 0$ and $y = 0$ occurs on $130 = 130 + 64 \cdot 0 + 0$. Note that each compute kernel must have all of its input values before executing, which explains why we wait until 130 before reading `hw_input`; it takes two full lines of length 64 and two additional pixels before all nine stencil pixels are ready.

The schedule of the unified buffer shows the absolute order for all operations in the application. By directly converting the values to cycle counts, we have a possible cycle accurate schedule for the accelerator. However, we may need to modify the schedule to account for any hardware-specific implementations. For example, we assume zero cycles needed for each compute kernel, but pipelining might lead to multiple cycles to execute a compute kernel.

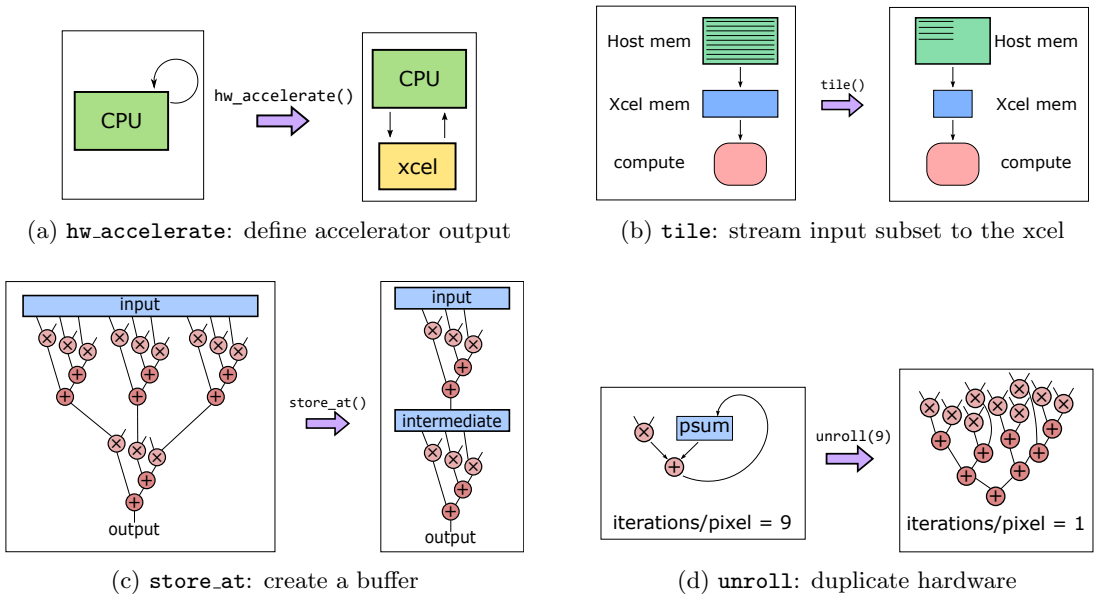
With this abstraction, we can determine the configurations needed for the memories on the CGRA. The storage capacity can be calculated from the schedule difference in write and read times of each iteration point. The controller needed for writing and reading values is determined directly by the schedule.

4.3 Halide Algorithm and Scheduling for Hardware

Using Halide as a front-end language, we are able to readily define image processing and deep neural network applications. These applications can be tested using the default CPU codegen with test images. The same algorithm can then be scheduled for the CGRA. All that is needed is a unique schedule to conform and take advantage of the hardware accelerator.

Not all Halide code is right now supported by our current hardware accelerator. Several computing patterns and indexing strategies cannot be mapped to our memory tiles on the Amber CGRA. Specifically, the address pattern into the memory must follow an affine index pattern. Although we can express non-affine address patterns in Halide and in the unified buffer abstraction, the eventual mapping to memory tiles fails. Similarly, data dependent indexing can be expressed in the front-end, but mapping to hardware has not been implemented by the memory mapping yet. A representative set of examples of the supported Halide code is shown in Table 4.2.

Once an algorithm has been created, we must schedule the code to target our hardware accelerator



(a) `hw_accelerate`: define accelerator output

(b) `tile`: stream input subset to the xcel

(c) `store_at`: create a buffer

(d) `unroll`: duplicate hardware

Figure 4.3: Depicted are the effects of four Halide scheduling hardware when added to a Halide schedule that targets CGRA hardware as described in Section 3.5. A user adds each scheduling primitive to a loop in order to transform the generated application running on the accelerator.

using our extension to the Halide scheduling primitives described in Section 3.5. When scheduling buffers, the Halide schedule for unified buffers is stored and computed at the outermost tile level, since we perform loop fusion later in the compiler during memory scheduling in Clockwork.

Figure 4.3 visually shows each of the hardware scheduling primitives, while Code 3.6 gives a full example of the Halide schedule for the cascade application. Using these scheduling primitives, we can create a schedule tailored for our CGRA.

4.4 Halide Compiler Passes

Once the user writes an application and schedules it for hardware, our modified Halide compiler performs transformations for the target hardware.

Accelerator Extraction

The first step is to create a separate scope for the hardware accelerator. This boundary is defined by the `hw_accelerate` scheduling primitive. Once the target output Func is found, the compiler pass traverses through the producer loopnest to find the loop where the accelerator begins. `_hls_target`, a new identifier in HalideIR, is used to enclose the full scope of the hardware accelerate block.

Once the scope of the accelerator has been created in the IR, the boundaries are checked. A

Table 4.3: Classification of different memory types based on their memory usage and index pattern. The italicized memory types are not handled by the Amber's memory tiles.

		Memory Usage			
		initialized	streaming	RMW	iterative
Index Pattern	constant	tap register	—	—	—
	affine	tap memory	buffer	accumulation	*recursion*
	non-affine	*tap memory*	*buffer*	*accumulation*	*recursion*
	data dependent	LUT	*RAM*	*histogram*	*pointer chasing*

closure is conducted on the enclosed block of code to determine the input Funcs that are necessary to compute the accelerator. Every data dependent stream must be identified as an input stream. Any input stream that fails this assertion results in a termination of the compiler for the user to fix the Halide schedule. This check is performed so that the user is fully aware of which input streams are needed for the hardware accelerator.

Multi-dimensional Indexing

Typically, Halide's built-in passes for CPU processing lower down the multi-dimensional indexing to a one-dimensional index. This works better for the C codegen where the sizes of one-dimensional buffers are more clear. However, our compiler system prefers using multi-dimensional indices to do polyhedral analysis in Clockwork. Therefore, this compiler pass ensures that every Func that is in the accelerator retains its multi-dimensional indexing. This is accomplished by tagging each of the Funcs used by the accelerator with `.stencil`, so that these multi-dimensional index calls are not lowered to one-dimensional loads. All Funcs outside the accelerator scope (and run on the CPU host) are lowered to one-dimensional loads.

Memory Classification

To ensure that we have covered different Halide use cases in terms of memory manipulation, we classify memory operations used in an application. Identifying the indexing patterns of memory calls is useful for eventual scheduling and mapping to memory tiles. This compiler pass analyzes the usage pattern of each memory and classifies them. There are two parts to this classification: the memory usage and the index pattern. This pass analyzes the memory data, write addresses, and read addresses of each Func to classify them as single-initialization, streaming, RMW, or iterative memories with constant, affine, non-affine, or data dependent addressing. A full table of the intersection of these two properties is shown in Table 4.3. This classification categorizes some of the use cases shown in Table 4.2.

The **memory usage** of a Func is based on how the the algorithm performs initialization and updates to the values in the memory. Halide supports many variations of initialization and updates, which we have categorized here. By categorizing the space of memory usage in Halide, we can have

better confidence that the Halide language space is covered by our compiler system. Both constant registers and ROMs are initialized to particular values, and then the values are never modified during the execution of the application. Streaming memories encompass most of the memories on our accelerator, where data dependent values are written into a memory that are later read out for computation. RMW memories use values that are written back to the same memory location. And finally, iterative memories use stored values to calculate other values in that same memory. For example, a Fibonacci sequence can be defined as an iterative algorithm where $F_n = F_{n-1} + F_{n-2}$ meaning two previous values (F_{n-1} and F_{n-2}) are needed to calculate a new value (F_n).

The **index pattern** of a Func is determined by the read and write address calculations for a memory. The index pattern is defined by the addressing function used in index calculation, and later becomes a constraint when mapping to specific memory addressors. These addressing functions must be mapped to address generators in our hardware, so classifying the different types of addressing schemes shows which parts of the language any given specialized hardware can accelerate. A constant index is an unchanging value seen in single-initialization memories such as the write patterns of constant registers and ROMs. An affine address uses an affine expression of index variables, as commonly seen with stencil applications. Non-affine addresses use index variables for address calculations but without affine expressions. Note that this non-affine group is defined mainly based on the capabilities of the address generator. Our CGRA can only handle affine addresses, so most other index calculations are lumped into this unsupported group, but other hardware that is able to calculate a larger set of index calculation would include more supported categories. Finally, data dependent index expressions use values from other memories to determine the index.

Identifying the memory usage and index pattern is important for later analysis and optimization. Our Halide compiler identifies constant registers as single-initialization with constant indexing. Similarly, ROMs are single-initialization with constant write addresses and data dependent read addresses. Both constant registers and ROMs are generated in the computation files instead of being treated as memories. This allows for Clockwork to focus on the streaming memories with more complicated dependencies rather than these simple cases that can be generated without much analysis.

Memory identification also helps with memory scheduling. Iterative and RMW memories must be scheduled to ensure values have been written back to a memory before trying to use them for calculation. Data dependent addresses become a challenge to the scheduler because the addresses cannot be pre-computed, so vectorizing memory reads and writes is more difficult. Our CGRA has affine memory addressors, but non-affine indexes do not map to these dedicated addressors. Our memory analysis is performed in Halide and provided to the memory scheduler and mapper to guide compilation to the hardware.

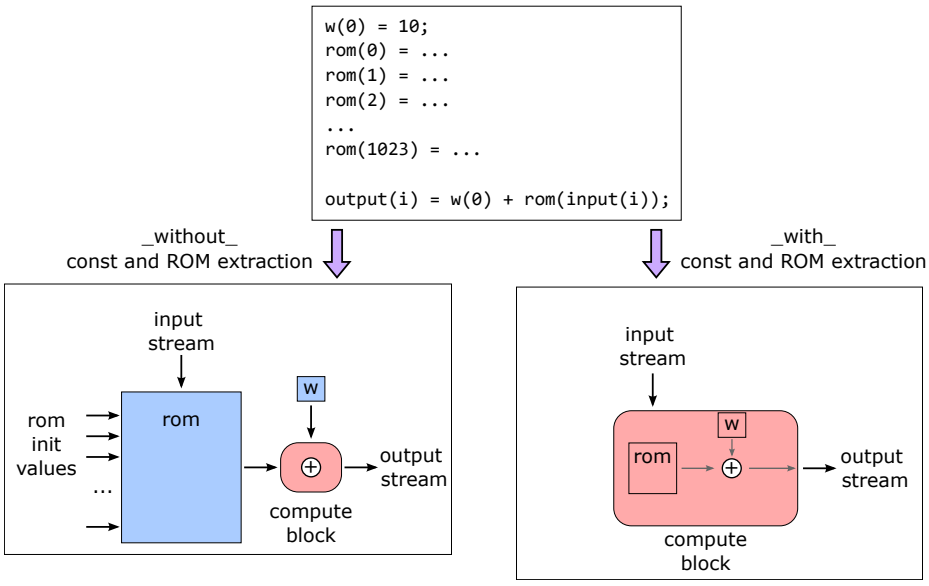


Figure 4.4: Our Halide pass removes ROMs and constant registers and directly maps them to their components on the CGRA. These mappings are stored directly with the compute mappings so our memory mapper does not need to analyze them, simplifying the memory mapping process.

ROM and Constant Register Extraction

After identifying different memory structures, we extract ROMs and constants so we can later classify them as compute kernels in the codegen. Both ROMs and constants differ from the other memories, since they do not require the built-in addressors. Instead, ROMs use data dependent values to index an SRAM, and constants are always the same, so they can be stored in registers.

Using the Memory Classification pass, we are able to determine which Funcs are ROMs or constants. The realization nodes for ROMs and constants are removed so that they are not created as memories during codegen as shown in Figure 4.4. Instead, we will later place the ROMs in the file with computation kernels, and the ROM kernel will be implemented using an SRAM on a memory tile using a special mapping process. The stores into the ROMs and constants become preloaded initialization values during memory configuration.

Update Merging

Update merging combines statements created after unrolling reduction domains. Reduction domains exist in the Halide algorithm to more elegantly create stencils. A reduction domain creates multi-dimensional index variables with a user-specified range of values and creates additional loops that iteratively accumulate values. Unrolling small reduction domains is a common strategy for hardware to use parallel computation units to perform the reduction in fewer cycles.

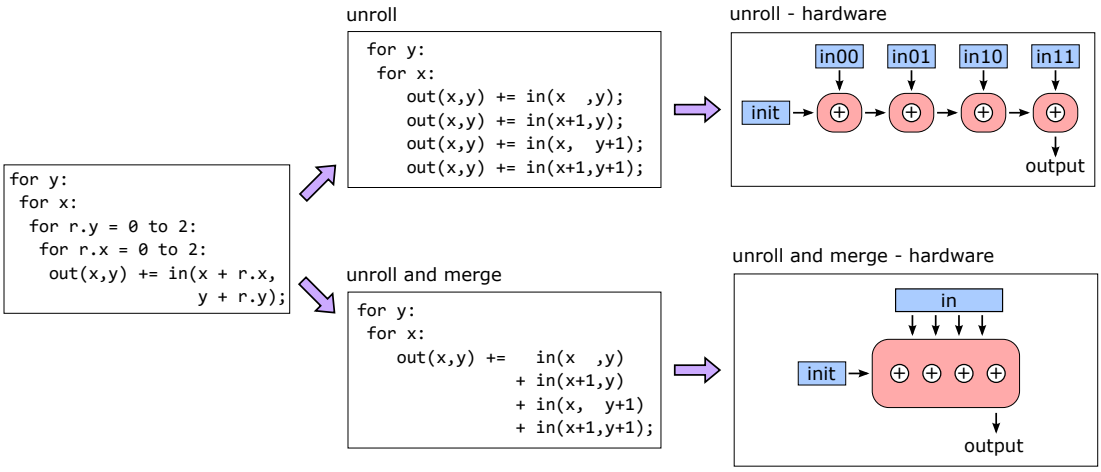


Figure 4.5: Our Halide pass merges together unrolled accumulation operations into a single compute kernel. This provides the memory mapper with stencil memory operations that can be more efficiently analyzed.

When reduction domains are unrolled, separate statements are created for each iterative update in the reduction. Using our HLS interpretation of the IR, this is equivalent to multiple computation kernels and more memory operations on the same memory address. The hardware-efficient solution is to not save the unrolled intermediate results, but simply build a hardware unit which produces a single output. Thus in hardware we desire a single compute kernel with the entire reduction and a single memory that is accessed at several indices based on the reduction domain.

This update merging pass identifies repeated read-modify-write operations, and combines them together. Namely, statements matching $x = x + a; x = x + b;$ are coalesced to $x = x + a + b$. This simplification is done by first unrolling the reduction domain. Then in the unrolled loop body, the compiler remembers each assignment into the Func. The next usage of that Func is replaced with the right-hand-side of the store statement. This complete process is illustrated in Figure 4.5.

Memory Index Shifting

One final memory modification before codegen is shifting the memory indices. When a Func has been tiled, the relative placement of the indices is held in HalideIR. However, when we use the tile in the hardware accelerator, this relative location is not known, and instead the top-left corner of the tile is assumed to be the zero point. Therefore, we must shift the memory indices that Halide uses to access the Func.

This pass is performed during the allocation of the Func. First, the minimum index for each of the dimensions is found. Then, each memory operation that uses the Func has its index shifted by the minimum index. This effectively realigns the indexing to properly index the tile given to the hardware accelerator with a redefined zero point.

BFloat Emulation

Our compiler supports compilation of bit-exact execution of bfloat operations. The CGRA hardware natively supports 16-bit bfloat addition, subtraction, and multiplication using dedicated bfloat hardware. The Halide language was extended to support the bfloat datatype so that we could write applications with that data. During compilation, bfloat operations that are run on our target hardware accelerator are unmodified. They are represented as normal IR adds, subtracts, and multiplies, except they are operating on bfloat data.

BFloat execution on the CPU requires emulation by conversion operations to properly compute bfloat results. In this BFloat Emulation pass, any bfloat operations outside of the accelerator block are replaced by datatype casts before and after every bfloat computation. The CPU emulates bfloat16 computation by running float32 instructions. The conversion from bfloat16 to float32 involves simply appending zeros to the mantissa. For conversion from float32 to bfloat16, values are rounded to match the precision of the bfloat datatype. We round numbers to even values to match the rounding performed by the CGRA hardware. Using this emulation, we are able to perform the same bfloat computation on the CGRA and CPU. By scheduling loops so the computations run in the same order, we can perform bit-exact comparisons of the results.

4.5 Codegen to Clockwork

The last step of the Halide compiler is the codegen. Our codegen produces code for the CPU host, compute kernels for the CGRA, and unified buffers for the Clockwork memory scheduler and mapper.

Clockwork: Memory Mapping

Before we talk about Halide's codegen to Clockwork, let's review Clockwork. Clockwork is a DSL that uses C++ objects and functions to describe hardware kernels and dependencies. The creation of an application is centered around an *op*, short for compute operation. Each op associates a compute kernel, consisting of a single C++ function, with several inputs and a single output value. The particular computation is not very important for Clockwork; instead the graph of input and output dependencies plays an important role for Clockwork's scheduling. Each op denotes the name of all memory inputs (the memory loads) as well as the name of the memory output (memory store). The memory operations are indexed by multi-dimensional variables with offsets to create stencils.

With the application ops and dependencies defined, Clockwork runs to determine an efficient execution ordering. By analyzing the dependencies, op execution can be fused so that their execution can be overlapped with each other. After op fusion, Clockwork uses polyhedral scheduling and isl [101] to calculate the exact timing of all computation. The schedule obeys all dependencies from the application definition, and overlaps the computation as much as possible to reduce memory

requirements and decrease application runtime. Our objective in the Halide codegen is to produce a format that Clockwork can use as an input to then assign a schedule and map to memories.

Accelerator Generation

We generate the multiple files from our final form of HalideIR. By traversing through the nodes in the HalideIR application, we are able to create the requisite host code, Clockwork memory file, and CoreIR compute file. For an accelerated application, the accelerator is defined by the `_hls_target` IR node. Outside of the accelerator, all generated code goes into a standard CPU C executable file. In place of the `_hls_target` node and its entire code body, the generated CPU code includes a call to the accelerator with all necessary input variables.

The call to the accelerator is typically within a set of loops in order to execute the accelerator for each tile of output. A single accelerator call requires index variables to indicate which tile of the input and output images are needed. The example host code for the cascade application is shown in Code 4.1. This host code iterates over the input tiles, and then runs the hardware accelerator with the input data, while collecting the output. RDAI helps define the interface of our accelerator call.

Our application compiler uses a library developed in our research group that provides a reconfigurable device access interface (RDAI) [51]. RDAI provides the calls to configure and start an accelerator from the host machine. RDAI is further modularized to provide a single interface for any of the possible runtimes that are later run on each specific device. Each platform-specific runtime is implemented in RDAI to provide the correct calls needed to configure devices, move data, and start the device for an accelerator run. RDAI is a subset of OpenCL [45], which abstracts accelerator calls in a similar manner.

Code 4.1 contains multiple RDAI classes and calls, each of which is prepended with `RDAI_`. `RDAI_MemObject` defines the size of each of the inputs and output buffers needed for an accelerator call. `RDAI_PlatformType` on line 25 then specifies which runtime platform is being used, in this example, Clockwork. `RDAI_VLNV` on line 27 then defines what specific application is being used (in a similar fashion to Vivado's vendor, library, name, and version). The most important call is `RDAI_device_run` on line 33, which runs the hardware accelerator given the set of inputs and output buffers. One can see how these calls are very generic functions which would be necessary to implement on an accelerator-host interface with any device. RDAI provides the implementations for the CGRA and other devices that we are interested in.

Once the host CPU code is generated, the accelerated code is created for the hardware accelerator. This accelerator code is generated into its own files: one for memory and two versions of the compute.

Memory

After creating the host code, the next step is generating the code for the hardware accelerator. The Halide codegen uses a visitor pattern to generate the Clockwork files. The Halide compiler follows

```
// Allocate and provision _hw_input_stencil.
RDAI_MemObject *_hw_input_stencil = RDAI_mem_shared_allocate(10183592);
for (int _hw_input_y = 0; _hw_input_y < 0 + 499; _hw_input_y++) {
 for (int _hw_input_x = 0; _hw_input_x < 0 + 5404; _hw_input_x++) {
  int32_t input_addr = calc_input_addr(_input_buffer, _hw_input_x, _hw_input_y);
  uint8_t _252 = ((const uint8_t *)_input)[input_addr];
  uint16_t hw_input_value = (uint16_t)(_252); // computation outside accelerator

  uint16_t *_hw_input_stencil_host = (uint16_t *) _hw_input_stencil->host_ptr;
  int32_t hw_input_addr = calc_hw_input_host_addr(_hw_input_x, _hw_input_y);
  _hw_input_stencil_host[hw_input_addr] = hw_input_value;
 } // for _hw_input_x
} // for _hw_input_y

// Allocate and produce buffer _hw_output_stencil.
RDAI_MemObject *_hw_output_stencil = RDAI_mem_shared_allocate(5346000);
for (int _hw_output_yo = 0; _hw_output_yo < 0 + 5; _hw_output_yo++) {
 for (int _hw_output_xo = 0; _hw_output_xo < 0 + 9; _hw_output_xo++) {
  RDAI_MemObject* _yo_obj = RDAI_mem_shared_allocate(1);
  _yo_obj->host_ptr = (int32_t*) _hw_output_yo;
  RDAI_MemObject* _xo_obj = RDAI_mem_shared_allocate(1);
  _xo_obj->host_ptr = (int32_t*) _hw_output_xo;

  // Define and set up the hardware accelerator that we will run.
  RDAI_PlatformType platform_type = RDAI_PlatformType::RDAI_CLOCKWORK_PLATFORM;
  RDAI_Platform **platforms = RDAI_get_platforms_with_type(&platform_type);
  RDAI_VLNV device_vlnv = {{"aha"}, {"halide_hardware"}, {"cascade"}, 1};
  RDAI_Device **devices = RDAI_get_devices_with_vlnv(platforms[0], &device_vlnv);
  // Define the input variables needed for this accelerator call.
  RDAI_MemObject *mem_obj_list[5] = {
    _yo_obj, _xo_obj, _hw_input_stencil, _hw_output_stencil, NULL };
  // Execute a tile on the hardware accelerator.
  RDAI_Status status = RDAI_device_run(devices[0], mem_obj_list);
  // Clean up RDAI call after execution.
  RDAI_free_device_list(devices);
  RDAI_free_platform_list(platforms);
 } // for _hw_output_xo
} // for _hw_output_yo

// Produce final output by copying from the accelerator output.
for (int _output_y = 0; _output_y < 0 + 495; _output_y++) {
 for (int _output_x = 0; _output_x < 0 + 5400; _output_x++) {
  int32_t hw_output_addr = calc_hw_output_addr(_output_x, _output_y);
  uint16_t *_hw_output_stencil_host = (uint16_t *) _hw_output_stencil->host_ptr;
  uint16_t hw_output_value = _hw_output_stencil_host[hw_output_addr];

  int32_t output_addr = calc_input_addr(_output_buffer, _output_x, _output_y);
  uint8_t _435 = (uint8_t)(hw_output_value); // computation outside accelerator
  ((uint8_t *)_output)[output_addr] = _435;
 } // for _output_x
} // for _output_y
```

Code 4.1: The host code includes the code outside the accelerator as well as the looped calls to the accelerator for each tile.

```
// Pseudocode for generating Clockwork Memory and Compute files.
// Entry to the visitor implementation is at the bottom using 'Main'.

// Append given compute kernel to ComputeFile.
EmitComputeKernel(compute, loads, store):
  interface = CreateInterface(loads, store);  // Create inputs and output interface
  EmitToComputeFile(interface);  // Generate compute kernel interface to ComputeFile
  for (rom in compute):
    EmitToComputeFile(rom);  // Generate rom used in this compute kernel
  EmitToComputeFile(compute);  // Generate compute operations

// Translate 'provide' statement.
Visit(provide):
  // 'provide' consists of computation on the RHS, and store on LHS.
  compute = provide.rhs;
  loads = Closure(compute);  // Determine loaded memories using a closure
  store = provide.lhs;
  Emit(provide);  // Generate 'add_op' and 'add_function' for this provide
  for (load : loads):
    Emit(load);  // Generate 'add_load'
  Emit(store);  // Generate 'add_store'
  EmitComputeKernel(compute, loads, store);  // Generate compute in a separate file

// Translate loop and then codegen loop body.
Visit(loop):
  Emit(loop);  // Generates 'add_loop'
  Visit(loop.body);

// Entry function for codegen, implemented using visitors. Generates primarily
// to MemoryFile, but also generates a separate ComputeFile.
Main(program):
  Visit(program.body);  // Visit the first HalideIR node
```

Code 4.2: Pseudocode for the codegen. Implemented using a visitor pattern that visits each of the HalideIR nodes. The codegen generates two Clockwork files: a memory file and a compute file. The HalideIR example in Code 4.3 would start with the accelerator node, then visit the loops, and last visit the provide statement.

the psuedocode in Code 4.2 to codegen the Clockwork files from the HalideIR. First, the Halide compiler analyzes and codegens based on the block of code labeled from our original `hw_accelerate` scheduling primitive. The first IR nodes in the accelerator block are the iteration loops as shown in Code 4.3. These are represented in HalideIR as for-loops that give the initial value and extent for each iteration variable. These loop variables are then used in the memory loads and stores. These surrounding loops become the iteration domain for the unified buffer. An example of these loopnests are seen in the example generated Clockwork input in Code 4.4.

Within the body of the for-loops are memory loads and stores. HalideIR statements consist of a memory store on the left-hand-side, while the memory loads and computation are on the right-hand-side. These computation statements become the basis of the codegen. For each memory store statement encountered in HalideIR, the memory file generates a function (`add_function` in Code 4.4). This function encloses a computation block and is populated with as many memory

```
_hls_target:  // entry point for the accelerator
 for y = 0 to 99:  // outer loops
  for x = 0 to 600:
    // Next 5 lines are a 'provide' statement.
    //   Left-hand-side: store
    //   Right-hand-side: computation on loads
    conv2_stencil_1(x, y) =
      conv1_stencil(x, y)     + 2*conv1_stencil(x+1, y)   +   conv1_stencil(x+2, y) +
      2*conv1_stencil(x, y+1) + 4*conv1_stencil(x+1, y+1) + 2*conv1_stencil(x+2, y+1) +
      conv1_stencil(x, y+2)   + 2*conv1_stencil(x+2, y+2) +  conv1_stencil(x+1,y+2) +
      conv2_stencil(x, y);
```

Code 4.3: Example HalideIR for the accelerator depicting the `conv1` kernel for the cascade application. The accelerator begins with the `_hls_target` node. A `provide` statement defines each compute kernel where the LHS is a memory store, and the RHS performs computation on memory loads.

loads and stores as in the statement. A closure is used on the single expression to determine any memory loads and ROMs that are needed. As specified earlier, the ROMs are treated in a special manner and are codegen'ed into the compute file. Each memory operation consists of a named memory buffer and multi-dimensional indexing into the buffer. In addition, a unique computation kernel is created for the statement, which is generated in its own file. Code 4.4 shows a computation kernel with nine `add_load`s and a single `add_store`. The first argument in each memory operation is the buffer name, and then is a list of multi-dimensional indices. For this example, the indices are affine addresses, but these indices could just as well be non-affine. Furthermore, these indices are indexed by iteration variables. When the codegen finds data-dependent addresses, we emit `add_dynamic_load` and `add_dynamic_store`. In those examples, there are multiple buffer names in a single call, and a line defines multiple memory ports in the application.

The access maps are the next parameter in the unified buffer. They are determined for each unified buffer based on the loads and stores. The codegen creates the application centered on the computation kernels. The unified buffer abstraction reorganizes the memory operations around each named buffer. All stores and loads to a buffer are grouped together to help determine any reuse later in the memory mapping phase. At this stage of the compiler, a single buffer might contain many loads, such as nine loads for a 3×3 stencil. The physical limitations of a memory tile, such as how many read ports it has, is resolved later during the memory mapping phase.

The last unified buffer parameter is the unified buffer schedule. HalideIR provides only a little insight into these values based on the dependencies created in the algorithm. These dependencies are implicitly defined by showing that a store into a named memory requires several loads from other memories. The default schedule that Halide provides is a sequential schedule where all producer values are calculated before the consumer is started. The default schedule is not shown in the generated Halide file, but instead is assumed by Clockwork. This assumption is the simplest schedule that adheres to the dependencies outlined by the construction of the memory transfers. Clockwork then later optimizes the timing of loads and stores, but this requires greater knowledge into the

```
// Cascade has a unified buffer named "conv2_stencil" storing 16-bit values
prg.buffer_port_widths["conv2_stencil"] = 16;

// These loops correspond to the iteration domain
auto y = prg.add_loop("y", 0, 99);
auto x = y->add_loop("x", 0, 600);

// The function indicates which compute kernel to use
auto hcompute_conv2_stencil_1 = x->add_op("op_hcompute_conv2_stencil_1");
hcompute_conv2_stencil_1->add_function("hcompute_conv2_stencil_1");

// These loads are access maps for outputs of the "conv1_stencil" buffer
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "y", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 1)", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "x");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "(x + 2)");
hcompute_conv2_stencil_1->add_load("conv1_stencil", "(y + 2)", "(x + 1)");
hcompute_conv2_stencil_1->add_load("conv2_stencil", "y", "x");

// This store is an access map for the input of the "conv2_stencil" buffer
hcompute_conv2_stencil_1->add_store("conv2_stencil", "y", "x");
```

Code 4.4: This is a sample of generated Clockwork input code. Above is the loopnest for a compute kernel that takes nine values from `conv1_stencil` and calculates a value of `conv2_stencil`. From the generated code, one can readily extract the iteration domain and access maps for this op.

hardware timing of the ports. Therefore, optimization of the unified buffer schedule is performed later during compilation.

Compute Kernel

The compute kernel is created using the computation blocks between the loads and stores into memory. The interface to each compute kernel is a set of memory inputs and outputs. When generating the compute kernels, two representations are created: a Clockwork C representation, and a CoreIR representation. The Clockwork C file is used during pre-mapping simulation to ensure there are no translation differences from Halide to Clockwork. The CoreIR representation is closer to a hardware representation where hardware modules are instantiated and then wired together.

HalideIR consists of mainly recognizable operations that match with the CoreIR instructions. These include arithmetic instructions (add, multiply, divide), sinusoidal functions (sin, tan, acos, cosh), and comparison operators (equal, greater than, less than or equal). A full list of the operations are in Table 4.4. The HalideIR input bitwidth (1, 8, 16, or 32 bits) and data type (unsigned, signed, bfloat) are used to determine the exact operator used in CoreIR.

Instruction selection for more complex operations also takes place during codegen. We ensure that any complex operator is not lowered until absolutely necessary to ensure the compiler does

Table 4.4: Available CoreIR operators created by Halide during codegen. The *signed* column indicates if the operators exist with signed and unsigned variants. The *bfloat* column indicates if the operators also have variants that work on bfloat numbers.

Library	Category	CoreIR Operators	Symbols	Signed?	Bfloat?
coreir	arithmetic	add, sub, mul	$+ \ - \ \times$		yes
		div, rem, mod	$/ \ \%$	yes	yes
	comparison	eq, neq	$== \ \neq$		yes
		lt, gt, le, ge	$< \ > \ \leq \ \geq$	yes	yes
	shift	shl, lshr, ashr	`<<` `>>` `>>>`		
	bitwise	inv, and, or, xor	$\sim$ `&` `|` `^`		
	ternary	mux	$c \ ? \ a : b$		
corebit	boolean	not, and, or, xor	`!` `&&` `||` `^`		
commonlib	add with carry	adc	$a + b + 1$		
	multiply	mult_middle	i16((i32(a) $*$ i32(b)) >> 8)		
		mult_high	i16((i32(a) $*$ i32(b)) >> 16)		
	minmax	min, max, clamp	min max clamp	yes	yes
	absolute	abs, absd	$\|a\|$ $\|a - b\|$		yes
float	round	frnd, fflr, fceil	round $\lfloor a \rfloor$ $\lceil a \rceil$		yes
	exponential	fsqrt, fsqr, fpower	$\sqrt{a}$ a^2 a^b		yes
		fexp, fln	exp ln		yes
	sinusoid	fsin, fcos, ftan, ftanh	sin cos tan tanh		yes
		fasin, facos, fatan, fatan2	arcsin arccos arctan		yes

not need to lift complexity later. Some operators exist in HalideIR, but are not base primitives in CoreIR. These include `min`, `max`, `abs`, and `absd`. For these operators, we create an extension library in CoreIR (called commonlib) that implements these operators using more basic CoreIR primitives.

The CGRA contains some complex operators that do not cleanly map to HalideIR. The CGRA multiplier is able to multiply two 16-bit numbers and output the bottom 16-bits, middle 16-bits, or top 16-bits. The equivalent Halide algorithm to perform this computation involves casting the inputs to 32-bits, multiplying, shifting the result, and casting back to 16-bits. In the codegen, we match this exact sequence of operations and replace the operators with the `mult_middle` or `mult_high` CoreIR operators based on the shift amount.

ROM Creation

In the Halide compiler, ROMs are identified and treated differently from other memories. As opposed to keeping multi-dimensional indexing, ROMs are lowered to one-dimensional memories. During codegen, these ROMs are treated in a special manner. Instead of being created as part of the memory file, they are directly mapped to a memory tile and stored with the compute kernels.

There are several conditions that need to be met to properly extract a ROM from HalideIR. The extent must be a fixed value, since the initialization of the memory is performed during configuration before data has been streamed through the system. Furthermore, the ROM initialization indices and values should be precomputed. The user can do that with Halide scheduling by unrolling the

computation if it is defined using an index variable.

The configurations needed for the ROMs are the overall ROM length as well as the the initial values. Initial values are collected through the unrolled initialization statements. The values are then stored in a JSON structure. The depth is found in the realization node and then rounded up to a multiple of four to ensure proper configuration of the memory tile.

Clockwork Memory Mapping

After the Halide compiler, we use Clockwork to do loop fusion and memory mapping to hardware. With the original Clockwork compiler [44], we can generate FPGA code. This process does loop reordering, kernel grouping, unrolling, kernel scheduling, line buffer synthesis, and code generation to HLS C. However, Clockwork's original system uses a different front-end with fewer user controls on loop scheduling. With the Halide loop scheduling, we reduce the role of Clockwork to just loop fusion and scheduling the timing for each kernel execution.[1]

In regards to the unified buffers, Clockwork performs three jobs: scheduling the kernels, composing the unified buffers, and mapping the buffers to hardware. The first step, scheduling the kernels, creates and then optimizes the op schedule, the last unified buffer parameter. The generated Halide code is created sequentially with disjoint loopnests and no loop fusion. Clockwork analyzes the kernels using polyhedral analysis to determine when each kernel can run. For stencil pipelines, overlap can occur and loop fusion takes place. If the most compute-intensive kernel does not have full utilization after loop fusion, then double buffers with loop pipelining is used. In all cases, Clockwork scheduling determines the exact cycles when each load, compute, and store occurs during accelerator execution.

After Clockwork scheduling, all of the dependencies on a memory are collected together to generate the unified buffers. Importantly, little hardware information has been used to generate the unified buffers. The unified buffer abstraction holds just the information on loop sizes, memory dependencies, and the sequence of computation. These properties are then mapped to either the original FPGA codegen to create code in HLS C, or it goes through a specialized mapper for the CGRA memories. The CGRA mapper consists of an extension to the original Clockwork system to map memories to our Amber CGRA, described in Section 2.3. In both cases, the same unified buffers hold the information needed to fully understand and encapsulate the application. We describe the full mapping process to the final CGRA bitstream in Section 6.1. Most importantly, the unified buffers provide a common interface for the software lowering to either FPGA or CGRA.

[1] Note that Halide *scheduling* and Clockwork *scheduling* have different intentions despite the same word choice. Halide scheduling consists of the user-added primitives to dictate the loop transformations for the generated code. Clockwork scheduling uses polyhedral analysis on the dependencies of the compute kernels to determine when each of the kernels can run; kernel execution scheduling consists of the exact cycle scheduling when each kernel executes.

Table 4.5: Halide applications used in the evaluation section.

Application	Type	Description
gaussian	stencil	3×3 convolutional blur
cascade	stencil	back to back gaussian convolutional blurs
harris	stencil	corner detector using gradient kernels and non-maximal suppression
upsample	stencil	up sampling by repeating pixels
unsharp	stencil	mask to sharpen the image
camera	stencil	camera pipeline with demosaicking, image correction, and tone scaling
nlmeans	stencil	edge-aware denoising using non-local means
histogram	data-dep	bin input pixel values into a histogram
bilateral	data-dep	edge-aware smoothing of the image using histograms
gaussian pyramid	hierarchical	convolution and downsampling
laplacian pyramid	hierarchical	pyramid decomposition of image
exposure fusion	hierarchical	pyramid blending of images based on brightness
image blend	hierarchical	stitching of two images using pyramid decomposition
optical flow	hierarchical	pyramid search of direction of pixel movement
gemm	DNN	general matrix multiplication
resnet	DNN	ResNet layers using multi-channel convolution
mobilenet	DNN	MobileNet layers using separable, multi-channel convolution
jitnet	DNN	layers in JITNet [71], including convolution and upsampling
unet	DNN	layers in U-Net [84], including conv, maxpooling, and transposed conv

4.6 Evaluation

This section evaluates the front-end application compiler, looking at the effectiveness of each step.

Halide Algorithm and Scheduling

We use Halide to describe image processing and deep neural network applications. A list of the applications we have created is in Table 4.5. Here, we have created stencil applications that use line buffers as memories; this includes the cascade application, which has been our running example. Some image processing applications use data dependent indices, like the histogram example which counts the number of data values in each data range. A final common stencil application type is hierarchical image processing. These hierarchical applications create image pyramids by downsampling the images and then processing each level. These levels are useful for understanding the images in high versus low frequencies as well as pixel versus patch comparisons.

As explained in Section 2.1, deep neural networks (DNNs) consist of convolution layers and matrix multiplication. We can accelerate a single DNN layer and find large benefits in runtime. Using Halide, we are able to describe this wide array of applications that have ample data-parallel computation that can be accelerated by hardware. Using our new set of Halide scheduling primitives, we are able to compile all of the applications in Table 4.5 to Clockwork. A large subset of these applications also meet the constraints of the CGRA, and can be further mapped to the hardware.

The hardware schedule that we create for each application must conform to the memory hierarchy. Each application uses the global buffer to input values onto the CGRA fabric. Each application then uses memory tiles to maximize the reuse of computed values rather than create more compute. The

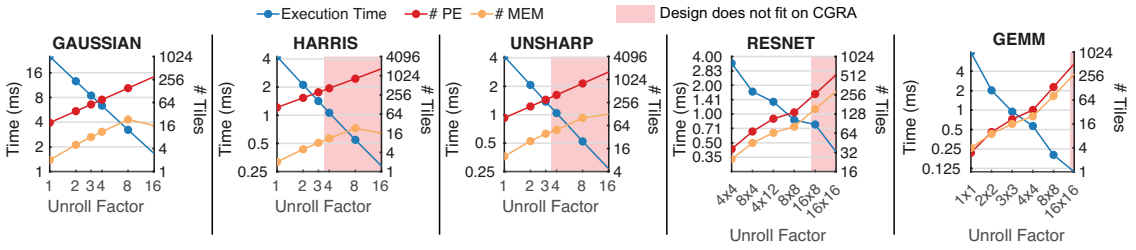


Figure 4.6: Execution time versus resource utilization tradeoff by using Halide's scheduling. At high unrolling factors, designs do not fit on the CGRA (384 PEs, 128 MEMs).

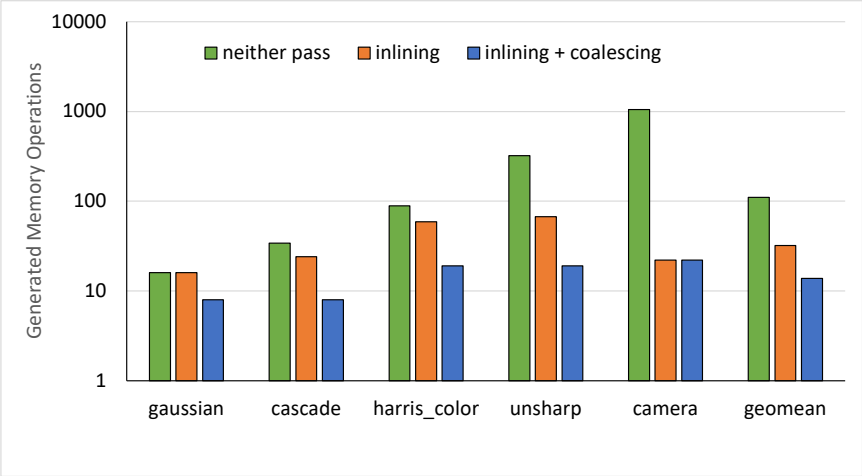


Figure 4.7: There is a reduction in memory statements by using the Halide passes. The two passes are: (1) inlining kernel and LUTs and (2) coalescing reduction updates into a single statement.

computation is tiled such that the buffer capacities fit into memory tiles. Furthermore, we can use Halide scheduling to perform hardware duplication. As shown in Figure 4.6, several applications can be unrolled to dramatically increase performance. As the unroll is increased, the runtime decreases at the cost of a larger number of tiles needed to implement the schedule. This shows that Halide scheduling empowers the user to create an application design that can run on the CGRA as well as make decisions on how to trade off characteristics of the final design.

Halide Compiler Passes

Next, we evaluate the Halide passes that transform HalideIR into usable Clockwork code. Figure 4.7 shows that the Const/ROM Inlining and Update Coalescing passes work to reduce the number of generated memory operations. These passes help Clockwork by removing Funcs, so that there are fewer memory statements that need to be scheduled using polyhedral analysis. Const/ROM inlining takes Funcs that can be mapped without a schedule, and automatically creates their mapping in

Table 4.6: The characteristics of our physical unified buffer (PUB) memory primitive and alternative memory implementations. Our compiler, using the unified buffer abstraction, supports more memory implementations as compared to FPGA compilers and other accelerator compilers.

Memory Back-end	**PUB (ours)**	DP-SRAM + AG	DP-SRAM + PEs	Ready-valid (Buffet)	BRAM + LUTs
SRAM Macro	**SP**	DP	DP	DP	DP
Built-in AG	**Yes**	Yes	No	No	No
Control Protocol	**Static**	Static	Static	Ready-valid	Static
Accelerator Architecture	**CGRA**	CGRA	CGRA	ASIC	FPGA
Unified Buffer	✓	✓	✓	✓	✓
Vivado HLS [104]	✗	✗	✗	✗	✓
SODA [19]	✗	✗	✗	✗	✓
PolyEDDO [72]	✗	✗	✗	✓	✗

the compute file. Update Coalescing removes Funcs that are generated as intermediates, which are remnants of using unrolled reduction domains.

In Figure 4.7, the gaussian application is unaffected by the Const/ROM Inlining pass. The gaussian application contains no LUTs or ROMs that can be inlined, and also does not use a Func to store the convolution weights. This means that the inlining pass is not needed for this particular application. The camera application has a large decrease in memory operations due to the inlining pass. This is because there are three LUTs that are simplified into ROMs during the inlining pass. Each ROM is initialized for 1024 addresses before being used for computation. The large number of initialization operations correspond to the large reduction in memory operations due to the inlining pass. Another aspect of the camera application is that there is no effect of coalescing. This is because none of the stencil computations are represented in Halide as unrolled RDoms, so there are no computation kernels to coalesce during this pass.

Unified Buffer Abstraction

After the Halide lowering passes and code generation, we construct the unified buffers for our application. The unified buffer abstraction provides a framework to describe the memories. From this abstraction, Clockwork maps to memory back-ends including FPGAs and CGRAs.

Table 4.6 shows the wide array of hardware platforms that can be generated from our unified buffer abstraction. These include FPGA, ASIC, and CGRA memory primitives. FPGAs use BRAM and LUTs to store memory and generate address controllers (AGs) from basic compute gates. The original Clockwork compiler compiles to FPGAs using HLS. Common in ASICs are latency-insensitive memories that use ready-valid protocols between modules. Using our specialized mapper, the cycle accurate schedule can be stripped away to build ready-valid memories.

CGRAs can be grouped based on their address generators and SRAMs. Three strategies for address generation include building counters from processing elements (PEs), building from PEs including specialized address generation operators, and building address generation into the memories for maximum efficiency. Our CGRA mapper supports all three of these addressing modes. We also

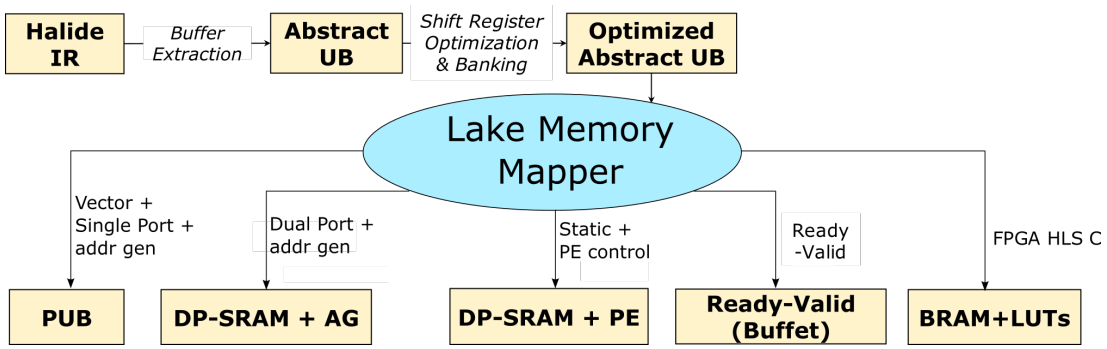


Figure 4.8: The Lake memory mapper provides the final mapping step to the specific hardware. The optimized abstract unified buffer is analyzed by Lake, which then specializes the configuration values for each memory type.

can support buffers built using SRAMs that are single-ported (SP) or dual-ported (DP). Scheduling a buffer with a dual-port SRAM is easier to configure because reads and writes can happen simultaneously. However, a single-ported SRAM is far more energy and area efficient. Our compiler is able to create the configurations for all of these different buffer types, including our highly specialized physical unified buffer (PUB), where the addressing logic is packaged in the same tile as a single-ported SRAM. Figure 4.8 shows how our Lake mapper takes the unified buffer abstraction and identifies different memory properties to target each of the specialized accelerator memories.

Our unified buffer abstraction and physical unified buffer implementation support both image processing and DNNs. By using the unified buffer abstraction, we keep our memory abstraction at a high level, allowing users to lower it to a wide variety of accelerator targets as well. We attribute this portability to the careful lowering of the IRs to the unified buffer abstraction before specializing the memory mapping to our target architecture.

4.7 Summary

We discuss the challenges of mapping applications to CGRAs with higher-level hardware primitives, especially complex memory tiles. To address this, we introduce a unified buffer abstraction that describes the data movement for every memory element in the application, capturing the requirements on both the memory and address generators. We utilize a modified Halide compiler to transform memories in HalideIR to fit the unified buffer abstraction. The final code generator produces unified buffers for the Clockwork memory scheduler and mapper. The compiler system excels in its versatility, supporting a wide range of applications and hardware accelerators. This is achieved through its unified buffer abstraction and physical implementation, which enable efficient mapping of applications to different reconfigurable architectures.

Chapter 5

Shared Hardware: Multiplexing Underutilized Compute

While looking at our applications, our strategy of creating a dedicated compute kernel for each stage has some inefficiencies. Some kernels rarely perform useful computations since the required data rate of these kernels is much less than other kernels in the application. One situation where this occurs is in pyramid applications, such as the one shown in Code 5.1. To enable an application to work with an image at different spatial scales, these applications create a set of versions of the input image downsampled into smaller image sizes. Processing each of these different scaled images enables the algorithms to be multi-scale. Due to the small image sizes in most of these pyramid levels, there are fewer computation resources needed for the inner compute kernels. Therefore, each dedicated computation kernel sits idle as the other levels perform computation.

Qiuling Zhu investigated different potential strategies of accelerating pyramid applications [110]. She found that some accelerators create dedicated hardware as described above. Another strategy was to utilize a single compute kernel that sequentially performs each pyramid level. Sharing a single compute kernel is possible because pyramid applications conveniently have similar computation performed at each pyramid level. This solution keeps the compute blocks busy during the entire application execution. However, this strategy also requires additional storage for intermediates, which can be reduced by interleaving the computation of the different pyramid levels.

Ideally, we want our hardware to have a high compute occupancy as well as limited storage for intermediates. This solution would have the compute blocks running useful computation most of the time. Additionally, the memory needed for the intermediates would be minimal. The final solution proposed in [110] is to share a single computation kernel to perform the calculations for multiple stages but interleave the computation to minimize storage requirements. To properly share a computation kernel, it needs to be scheduled so that it performs only one stage at a time. Based

```
// Algorithm: Each pyramid layer is successively blurred
blur0(x, y) = (k(0,0) * hw_in(2*x,2*y)   + k(1,0) * hw_in(2*x+1,2*y) +
               k(0,1) * hw_in(2*x,2*y+1) + k(1,1) * hw_in(2*x+1,2*y+1)) / 4;
blur1(x, y) = (k(0,0) * blur0(2*x,2*y)   + k(1,0) * blur0(2*x+1,2*y) +
               k(0,1) * blur0(2*x,2*y+1) + k(1,1) * blur0(2*x+1,2*y+1)) / 4;
blur2(x, y) = (k(0,0) * blur1(2*x,2*y)   + k(1,0) * blur1(2*x+1,2*y) +
               k(0,1) * blur1(2*x,2*y+1) + k(1,1) * blur1(2*x+1,2*y+1)) / 4;
blur3(x, y) = (k(0,0) * blur2(2*x,2*y)   + k(1,0) * blur2(2*x+1,2*y) +
               k(0,1) * blur2(2*x,2*y+1) + k(1,1) * blur2(2*x+1,2*y+1)) / 4;
hw_output(x, y) =  blur3(x, y);
```

Code 5.1: Pyramid blur application which applies four levels of blur on the input image.

on Qiuling's findings, I made modifications to the application compiler to enable compute sharing.

In this chapter, we introduce how the user can specify in Halide which computation kernels should be shared together. In this way, the user can choose to share computation kernels to increase compute occupancy as well as choose the desired interleaving. Implementing this function required changes to the Halide front-end as well as extending Clockwork scheduling to implement the new execution order specified in the Halide schedule. The modified compiler back-end then generates new connections for the application hardware graph to account for the sharing of the compute kernel. We show optimizations on our Amber CGRA to provide the compute kernel sharing with minimal hardware overhead and then show results on the effectiveness of compute sharing.

5.1 Halide Scheduling

Modifying the Halide front-end scheduling is our first step. New Halide scheduling primitives allow users to specify which compute kernels should be shared together. In keeping with Halide philosophy, we enable compute sharing with a new scheduling primitive that the user can control. In addition to specifying which compute kernels should be shared, the user also must specify the interleaving of the computation.

The *interleaving granularity* refers to how often the computation kernel switches between which stage it is computing. The interleaving options for an image are interleaved after every pixel, line of pixels, or block (tile) of pixels. Each of these interleaving choices change the number of cycles spent computing a certain pyramid level before switching to the next pyramid level. The choices are described based on the number of output pixels that are calculated in a single iteration.

Figure 5.1 shows four different interleaving decisions for three levels of 2×2 convolution down-sampled by 2 in each level. The horizontal axis shows the progression of time while the colors on different rows in a grouping depict the different compute kernels as they execute. The top choice is how the three compute kernels would execute with exclusive compute kernels. Each compute kernel executes whenever its requisite input pixels are ready. The red input pixels stream in continuously, and the compute kernels in green, blue, and purple execute once the necessary values have entered.

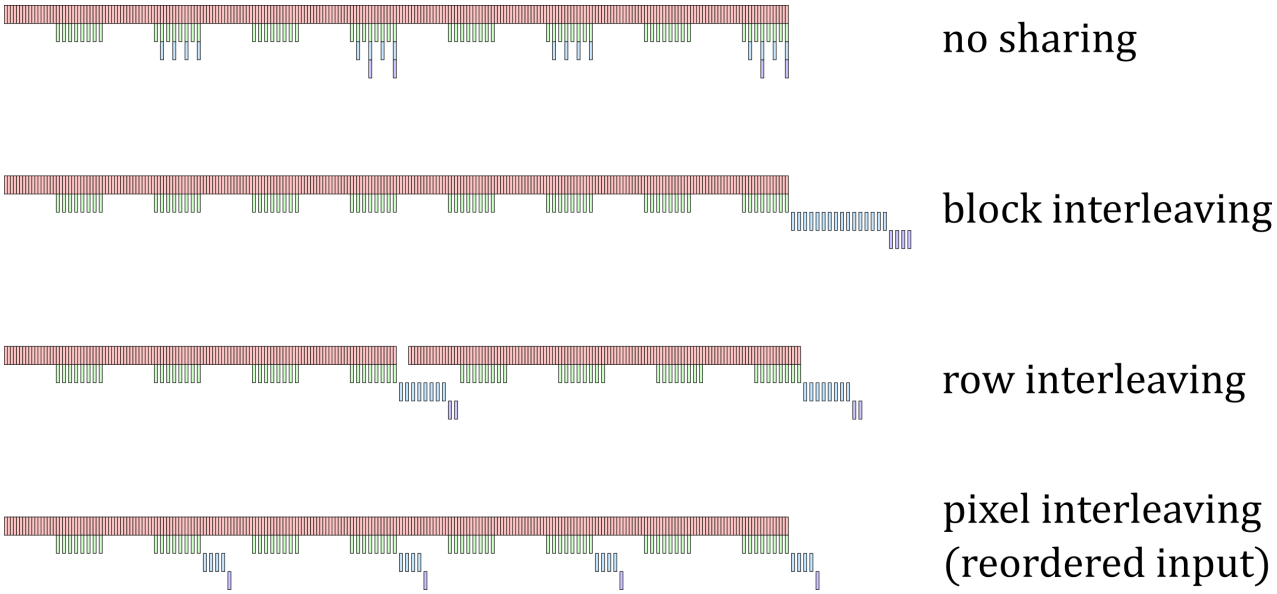


Figure 5.1: How interleaving choice affects the order of computation. Note that pixel interleaving additionally requires streaming the input data in a different order, which may not be feasible.

Notice that different compute kernels can align vertically since they can execute at the same time.

The second grouping of executions in Figure 5.1 shows sequential execution. In my Halide-specified scheme, this is interleaved by block. The full tile of the first convolution in green executes before the next convolution in blue starts. Last, the four purple output pixels are calculated. In this scheme, the execution of each pyramid level runs entirely before the next level runs. Note that the total execution time can be visually observed by the horizontal space. Tile interleaving has a modest increase in the execution time compared to the exclusive kernel execution, but this is compensated with fewer compute resources required for the shared compute version.

Row-interleaved execution is the third group of executions in Figure 5.1. Here, just the necessary input for a single row of output at the highest level of the pyramid is read. Once the first convolution has executed for four rows, the subsequent convolutions execute to calculate a row of output. The input then resumes its reads and repeats to calculate the second row of output. With this interleaving, the computation switches between kernels based on generating a single line of output. Notice that the input pixels can overlap the execution which helps reduce the total runtime. Note that when we try to schedule each of the kernels to an affine schedule, we end up with a small gap in input scans and kernel computation. This gap is needed to ensure that there is no resource conflict. For example, for the row interleaving version, 8 rows of the input are read. During this execution, 4 rows of the largest pyramid level are created (in green). At the end of this execution, we can create the next two pyramid levels. This is by design; we run all of the producers for enough iterations until we can create a single output row. However, this also means that we must serially produce the

```
// Schedule: Share the compute kernels
hw_output
  .compute_root()
  .tile(x, y, xo, yo, xi, yi, 8, 8)
  .hw_accelerate(xi, xo);

blur3.store_at(hw_output, xo).compute_at(hw_output, xo);
blur2.store_at(hw_output, xo).compute_at(hw_output, xo);
blur1.store_at(hw_output, xo).compute_at(hw_output, xo);
blur0.store_at(hw_output, xo).compute_at(hw_output, xo);

// Share the compute kernels
blur3.compute_share_root(y);
blur2.compute_share(blur3);
blur1.compute_share(blur3);
blur0.compute_share(blur3);

// Assign coarse-grain loops
std::vector<Func> funcs = {blur3, blur2, blur1, blur0, hw_in.in()};
for (auto& func : funcs) {
  func.coarse_grain_loop(y);
}
hw_output.coarse_grain_loop(yi);

hw_in.stream_to_accelerator();
```

Code 5.2: Schedule for compute sharing for Code 5.1. Each level shares a single compute kernel, and has the same coarse-grain loop using the y loop.

output (since we only have a single compute kernel). This means we must hold back the execution of the next input rows until we have completed the output row (purple), so that we don't have a resource conflict when we get to the next computation of the largest pyramid level (green). Notice that we hold back the input pixels (red) for just enough cycles so that compute kernel is available for the green computation precisely when the stencil is ready.

The last interleaving in Figure 5.1 is executed with a pixel interleaving. Each output pixel is calculated before proceeding to the other pixels in the input image. However, unlike the previous schedules, this one is not possible with our current input traversal order. Reading just the input needed to produce a single pixel on the output requires a subset of the input; for example, our first output pixel requires the upper-left quadrant of the input image. However, our hardware usually prefers reading the input data in row-major order. Even if we were to gather the input data in a different order, our problems continue if we are using line buffers. Line buffers work based on row-major traversal order. When you reorder the input traversal, the reuse pattern of the input becomes more complicated when the stencil size is larger than the stride size. Due to these issues, we will not consider a pixel interleaving in our final evaluation, even though it has the lowest storage requirements and overall latency.

We have seen that the different interleaving choices affect the order of computation and the resulting execution time. In addition, the choice in interleaving affects the required memory size.

```
// HalideIR mutator that annotates IR with each user-provided coarse-grained loop.
InsertCoarseLoops(program, coarse_loop_names):
  for (loop : program):
    if (loop.name in coarse_loop_names):
      InsertAnnotation(loop, "coarse_grain_loop_tag");

// HalideIR mutator that annotates IR with each compute share mapping
// during memory definition.
InsertKernelMappings(program, compute_share_mappings):
  for (memory_definition : program):
    if (memory_definition.name in compute_share_mappings):
      // Compute share mapping: original_kernel name -> new_kernel name
      //                        and compute share looplevel
      InsertAnnotation(memory_definition,
                       compute_share_mappings[memory_definition.name]);

// Change to codegen: new 'provide' visitor pass. Note the changes in the middle.
Visit(provide):
  // 'provide' consists of computation on the RHS, and store on LHS.
  compute = provide.rhs;
  loads = Closure(compute);
  store = provide.lhs;

  // compute_share_mappings populated based on HalideIR annotations.
  if (provide.name in compute_share_mapping):
    // Generate 'add_function' pointing to the shared kernel
    Emit(compute_share_mapping[provide.name]);
  else:
    Emit(provide);  // Generate unique 'add_op' and 'add_function' for this provide

  for (load : loads):
    Emit(load);
  Emit(store);
  EmitComputeKernel(compute, loads, store);
```

Code 5.3: Pseudocode for the Halide compiler changes for shared scheduling. HalideIR is mutated with annotations for coarse-grain loops and compute kernel mappings to the shared kernel. The codegen for `provide`, described in Code 4.2, is modified to emit the shared kernel based on the HalideIR annotations.

As the interleaving size increases, the number of values that are stored for the next stage increases. When interleaving by tile, we must buffer an entire tile's pixels before they are consumed. On the other extreme, we minimize the producer-consumer length by interleaving by pixel. Reducing the time between production and consumption of the intermediates reduces the memory requirement up to a point: given that we are performing convolutions, we always need to have line buffers for each level of the pyramid storing the data that will need to be reused by future rows. Thus, our choices have minimal memory requirements when interleaving by pixel, modest when interleaving by line, and a large memory requirement when interleaving by tile. We examine the different tradeoffs in more detail later in Section 5.4.

At the Halide stage, the user identifies which kernels can be shared and the interleaving loop level. The schedule for Code 5.1 is shown in Code 5.2. The new Halide scheduling primitive is

```
// Clockwork input file with shared compute kernels
auto blur0_y = prg.add_loop("blur0_y", 0, 32);
blur0_y->coarse_grain_loop_tag();
auto blur0_x = blur0_y->add_loop("blur0_x", 0, 32);
auto hcompute_blur0_1 = blur0_x->add_op("op_hcompute_blur0_1_stencil");
hcompute_blur0_1->add_function("hcompute_blur3_1");    // Shared function
hcompute_blur0_1->add_load("blur0_1_stencil", "(blur0_y*2)", "((blur0_x*2) + 1)");
hcompute_blur0_1->add_load("blur0_1_stencil", "(blur0_y*2)", "(blur0_x*2)");
hcompute_blur0_1->add_load("blur0_1_stencil", "((blur0_y*2) + 1)", "((blur0_x*2) + 1)");
hcompute_blur0_1->add_load("blur0_1_stencil", "((blur0_y*2) + 1)", "(blur0_x*2)");
hcompute_blur0_1->add_store("blur0_1_stencil", "blur0_y", "blur1_1_s0_x");

// blur1_y definition, blur1_y->coarse_grain_loop_tag(), blur1_x definition
auto hcompute_blur1_1 = blur1_1_s0_x->add_op("op_hcompute_blur1_1_stencil");
hcompute_blur1_1->add_function("hcompute_blur3_1");    // Shared function
// Loads and stores:  4x [ ->add_load() ] + 1x [ ->add_store() ]

// blur2_y definition, blur2_y->coarse_grain_loop_tag(), blur2_x definition
auto hcompute_blur2_1 = blur2_1_s0_x->add_op("op_hcompute_blur2_1_stencil");
hcompute_blur2_1->add_function("hcompute_blur3_1");    // Shared function
// Loads and stores:  4x [ ->add_load() ] + 1x [ ->add_store() ]

// blur3_y definition, blur3_y->coarse_grain_loop_tag(), blur3_x definition
auto hcompute_blur2_1 = blur2_1_s0_x->add_op("op_hcompute_blur3_1_stencil");
hcompute_blur2_1->add_function("hcompute_blur3_1");    // Basis function
// Loads and stores:  4x [ ->add_load() ] + 1x [ ->add_store() ]
```

Code 5.4: The generated input code to Clockwork. Note that each generated kernel in the pyramid has the same shared function. Generated by pseudocode in Code 5.3.

`compute_share(root_func, loop_level)`. The root Func is the last Func in the chain of shared compute stages. And the loop level specifies the granularity of sharing: the innermost loop level to share by pixel, the outer loop level to share by tile. The scheduling `coarse_grain_loop` specifies which loop is fused together to create the desired interleaving.

In the Halide compiler, we must annotate these new shared kernel mappings and loop level granularity to pass to the codegen. These steps are shown as pseudocode in Code 5.3. The shared kernels and loop level granularity information is stored within the HalideIR nodes. These nodes are simply tagged with the compute kernel and loop levels so that this information can inform the Clockwork codegen. During codegen, the compute kernels in the Clockwork memory file are annotated with `->add_function()`, which points to the execution function. When two compute kernels point to the same function rather than their own dedicated function, then these compute kernels share an execution unit. The compute granularity is labeled on each of the loop levels where the coarse-grain loop is for each of the kernels. Clockwork scheduling and hardware generation will use this information to create the modified schedule timing and hardware design. Code 5.4 shows the generated annotations for our pyramid downsample example.

```
// Original loops

for y = 0 to 63:
  for x = 0 to 63:
      hw_in = ...

for y = 0 to 31:
  for x = 0 to 31:
      blur0 = ...

for y = 0 to 15:
  for x = 0 to 15:
      blur1 = ...

for y = 0 to 7:
  for x = 0 to 7:
      output = ...
```

```
// Split on coarse-grain loop
for y_o = 0 to 7:
  for y_i = 0 to 7:
   for x = 0 to 63:
      hw_in = ...
for y_o = 0 to 7:
  for y_i = 0 to 3:
   for x = 0 to 31:
      blur0 = ...
for y_o = 0 to 7:
  for y_i = 0 to 1:
   for x = 0 to 15:
      blur1 = ...
for y_o = 0 to 7:
  for y_i = 0 to 0:
   for x = 0 to 7:
      output = ...
```

Code 5.5: Compute kernel loops before and after refactoring for later coarse-grain loop fusion. This example uses row interleaving with a coarse-grain loop of y. The coarse-grain loops are split such that each shared compute kernel has the same sized outer loop.

5.2 Clockwork Scheduling

Once we have outputted the needed information from Halide, we use Clockwork to generate the new schedules based on a shared compute kernel. The new schedule needs to be modified such that (1) the shared compute kernel does not execute on overlapping clock cycles, and (2) the frequency of compute kernel switching is based on the user-directed interleaving. Clockwork scheduling must assign the cycle times that each stage should execute on the shared compute kernel. Calculating the new loopnest schedules requires loopnest modifications and proper timing of the kernels. Next, we go over how Clockwork creates the new schedules.

Coarse-grain Loop Fusion

The first change to Clockwork scheduling is to modify the loopnests to account for the interleaving. The interleaving dictates several properties of the modified loopnest. Each new loopnest follows a sequence of steps to execute each of the shared compute kernels. Also, the last step of the sequence calculates a portion of the output pixels. Based on the interleaving, these steps are repeated to calculate the entire output block. For example, a coarse-grained tile interleaving only iterates a single time before the output is complete. On the other hand, when interleaving at a fine-grained pixel granularity, each sequence calculates a single output pixel and so it is repeated many times to construct the full output block. To reflect the repetition of the fine-grained interleaving, we split the outer loops, and then fuse the inner loops across different compute loops. The interleaving loops are coarse-grain loops that are split so that each of the shared loops can be fused together. For this reason, Halide labels the coarse-grain loops to guide this step.

Our loop fusion process mirrors the strategy taken for coarse-grain loop fusion for DNN applications. For both use cases, some of the outer loops are fused together for a set of pipeline stages, while the innermost loops are separate. This creates a loopnest that is not perfectly nested, but instead has distinct coarse-grain blocks as seen in the row interleaving in Code 5.5. Each of the compute kernels are executing the same number of coarse-grain loops (including all of the outer loops). However, the body of the coarse-grain loops differ for each distinct compute kernel. Their final Clockwork schedules will also differ to obey the producer-consumer dependencies and hardware constraints (only a single compute kernel for our use case with shared compute).

One complication of the coarse-grain loops is that the size of the loops is not the same across all pipeline stages. Due to downsampling and stencil contractions, the loop extents are different across stages. Therefore, the first step in Clockwork scheduling is to split the loops to create evenly-sized loops that can be fused. First, all of the coarse-grain loops are analyzed and the rate of the smallest loop is used as the loop length. For a series of downsamples, this is the size of the last, smallest pyramid level. All other coarse-grain loops are split such that the outer loop length is the determined size. In Code 5.5, the coarse-grain loop is y, and the last loop level has just 8 iterations. Therefore, we split the y loop for each function so that each outer iteration, y_o, has 8 iterations.

After all of the loops are split, the loops can be effectively fused. The outermost loops all match in size until the coarse-grain loop, and then the inner loops are distinct for each pipeline stage. Note that the difference in pipeline stages results in very different execution times for each pipeline stage. A series of four stages with downsampling in the x and y dimensions results in the first stage calculating 16 lines that each are 16$\times$ longer than the single output line that is calculated in a coarse-grain block. In this way, the computation has been fused such that a block of input lines is calculated for the next line of output pixels. The following section describes how the modified Clockwork schedule is calculated.

Coarse-grain Block Timing

The next step is calculating the block timing to schedule each pipeline stage within a periodic shared schedule of the compute kernel. Each pipeline stage was split so that the coarse-grain loops could be fused. Now, the execution timing must be determined so that there is no overlap in execution. Our interleaving strategy and coarse-grain loops determine an inner block that is repeated by each of the compute kernels. We expect that our schedule will be periodic using the outer, coarse-grain loops. To accomplish this, we calculate the execution time for each compute kernel within a single iteration of the coarse-grain loop. One other constraint is that every pipeline stage gets a contiguous slot of time to execute within the coarse-grain loop before switching to the next compute kernel. This constraint allows each schedule to be affine so that we can eventually map the schedules to the affine address generators on the Amber CGRA.

With these goals, we can use the address generator parameters with our desired constraint of

non-overlapping cycle assignments to formulate the new schedules. The timing of a pipeline stage is expressed using a cycle-accurate affine expression using the loop extent, stride, and delay. We represent each of the schedules using a vector with entries for each loop level. Each pipeline stage has their own vector for the extent, stride, and delay to calculate the timing of its execution. The loop extent, $\vec{e}$, represents the total iterations at each loop level. The stride, $\vec{s}$, is an initiation interval consisting of the computation interval and the loop execution time. This stride depicts how many cycles occur between executions at different loop levels. The computation interval, $\vec{q}$, refers to how often the computation executes; while the loop execution time, $\vec{l}$, is the number of cycles for a single iteration of each loop level. Finally the delay, $\vec{d}$, is the number of initial iterations of each loop level to stay idle in order to respect producer-consumer dependencies. Next, we go more in depth on calculating each of these vectors.

Loop Extent ($\vec{e}$): The loop extents are already defined by the loop iteration. The loop extents are simply how many iterations each for-loop must repeat the statements in its loop body. Each of these iteration domains of the produced output is used to calculate the bounds for its inputs. Every downsampling kernel results in a larger loop extent for its producer as compared to the loop extents of the consumer. These loop iteration values and bounds analysis are already computed and provided in Halide, so no further calculation is necessary in Clockwork.

Stride ($\vec{s}$, $\vec{q}$, $\vec{l}$): The stride ($\vec{s}$) is comprised of a computation interval ($\vec{q}$) and loop execution time ($\vec{l}$). The computation intervals denote the relative kernel sizes between producer and consumer. The larger the computation interval, the more producer pixels must be produced before the computation kernel can be executed. For example, a downsample of 2 results in a computation interval of 2. The loop execution time is how many cycles occur in each loop level. Calculating the loop execution time is different for the inner and outer loops. Since the inner loops inside the coarse-grain loop are run sequentially, this value is just the product of all of the loop extents up to that loop ($l_i = \prod_{j=1}^{i-1} e_j$), starting with $l_1 = 1$. For the coarse-grain loop, we must consider how much time a coarse-grain loop iteration takes. The length of the coarse-grain loop level is the sum of the executions of all of the pipeline stages that shared the kernel. Outside the coarse-grain loop, the execution is fused with all other pipeline stages. We multiply the previous loop level size by the maximum loop size to get the extent of each successive level. Putting these together, we can get the stride of the computation. The consumer kernel must wait for the loop execution time for each pixel in the computation interval, so the total stride for the consumer is the product of these values.

Delay: The delay for a kernel is derived using the stencil size and represented using a multi-dimensional value. For typical image-processing stencils, this is one less than the stencil size in each dimension. This means that a 3×3 stencil would have a delay of 2 rows in the y dimension and 2 pixels in the x dimension. The delay for shared kernels is based on the loop being inside the coarse-grain loop or not. Inside the coarse-grain loop, the stencil delay is set to zero, since we no longer fuse these inner loops. Instead, the granularity of delays begin at the coarse-grain loop. This

is because the innermost loops are all run sequentially to achieve the desired interleaving. At the boundary between shared compute kernels and dedicated kernels, delay is not set to zero, since the producers using dedicated compute kernels produce pixels at a consistent rate. For the innermost loop (at the cycle level) we have a phase delay accounting for the execution of the compute kernel within the shared compute pipeline. This delay is the cumulative execution times of all previous computation kernels.

Altogether, the execution time for N-dimensional point $\vec{p}$ is:

$$\sum_{i=1}^{N} l_i(q_i \cdot p_i + d_i)$$

Each of the N dimensions has its own loop extent ($\vec{e}$), computation interval ($\vec{q}$), loop iteration interval ($\vec{l}$), and delay ($\vec{d}$). Subbing in a multi-dimensional point into this equation gives the iteration time for that compute stage. This expression can be rewritten to more closely match the memory configuration values for the address generators in the Amber CGRA as:

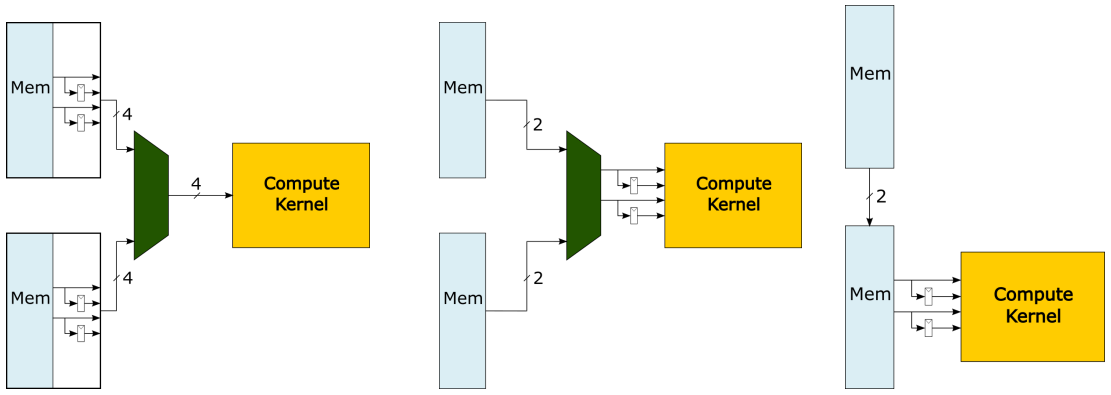
$$D + \sum_{i=1}^{N} s_i \cdot p_i$$

where the initial delay is:

$$D = \sum_{i=1}^{N} l_i \cdot d_i$$

and strides are calculated as:

$$s_i = l_i \cdot q_i$$



(a) buffers muxed together (b) buffers sharing registers (c) buffers chained

Figure 5.2: Multiplexed inputs, stencil register sharing, and memory chaining are ways to share a single compute kernel between multiple buffers.

5.3 Hardware Generation

The final part of Clockwork mapping is generating the hardware for the application. This hardware represents the MEM and PE tiles needed to execute the application as well as the connections between tiles. Sharing compute kernels alters the generated hardware where its primary purpose is to reduce the number of compute kernels. Instead of each pipeline stage having its own dedicated compute kernel, only a single compute kernel is created. Since the data from the different kernels might come from different unified buffers, hardware generation needs a mechanism to choose which unified buffer is feeding the compute kernel. The following section explores different approaches to accomplishing this data multiplexing.

Multiplexed Inputs

A set of multiplexers provides a traditional method of connecting all of the inputs to a shared compute kernel. Each of the pipeline stages maintains their own distinct memories, and during the scheduled period of time executes on the shared compute kernel. Effectively, the pipeline stages time multiplex the compute kernel.

Time multiplexing the inputs requires an N-input multiplexer for each input to the compute kernel. For example, if a 3×3 compute kernel is shared among 4 stages, then 9 different 4-input multiplexers are necessary. Furthermore, the multiplexers need signals to select which input is executing for each cycle. Each memory has already been scheduled based on the calculations shown above to exclusive time slices. Therefore, we need to simply output a signal from each memory when they are outputting values. Luckily, Clockwork already has a mechanism to create a signal when there are outputs from a memory. Collectively, the memories create a one-hot signal that directs the multiplexer on which stage is executing.

Using multiplexers effectively connects the memories to the shared compute kernel; however, there is some redundancy in the implementation with each unified buffer having their own stencil registers and memory. Furthermore, implementing the multiplexers and control signals requires additional PE resources on the CGRA fabric. Next, we show some optimizations to reduce the hardware overhead.

Stencil Register Sharing

One inefficiency of our first implementation is that each of the unified buffers have their own exclusive stencil registers. However, when interleaving at the row or block level, the stencil registers can be shared between pipeline stages. Our original mux design creates stencil registers for each of the pipeline stages, and then the outputs of each stencil register are multiplexed into the compute kernel. Instead, for row or block interleaving we can share each of the stencil registers by placing a

multiplexer between the memory tiles and the stencil registers. For this new hardware configuration, instead of N sets of stencil registers, only a single set of stencil registers is needed.

The reason this approach can't support pixel interleaving is that shared stencil registers can't hold any necessary data for a compute kernel during an interleaving transition. This constraint means that sharing stencil registers cannot occur at the pixel level since you need them to store previous pixels in the line. For a pixel interleaving, exclusive stencil registers are needed for each unified buffer to store values that are needed on successive computations, which makes pixel interleaving expensive.

Row interleaving works well, since stencil registers don't hold useful data once a row is complete. Stencil shift registers help for temporal locality in the innermost traversal dimension, but do not help for further rows. If our interleaving is by row or coarser, the stencil registers can be shared. With this scheme, the stencil registers are used by the currently executing pipeline stage, and then change ownership when the compute kernel switches to the next pipeline stage. This means that each stencil register isn't dedicated to any one compute kernel.

This concept of shared memory can be extended beyond stencil registers, but is not explored in our compiler. Taken one step further, a shared compute kernel interleaved by block could share entire line buffers created by memory tiles. This is because once a pipeline stage has executed, its values are no longer needed, so the memory can be reclaimed. Extending the sharing beyond registers requires more complicated addressing in the memory tiles, and so is not explored here.

Memory Chaining

Once we have reduced the number of stencil registers, we still have the overhead of the multiplexers. Ideally, we would want to optimize our hardware mapping to free up the resources used for implementing the multiplexers. Our target CGRA has memory tiles that have a chaining feature. The original intent of memory chaining was to allow memory tiles to implement buffers that are larger than a single SRAM block. Chaining multiple memory tiles together gives a unified buffer its cumulative SRAM size and is implemented using a special *chain input* data port. When chaining is enabled, the output port switches between output read from its own SRAM and data from the chain input. Internally, this multiplexer is controlled by the schedule of the memory tile, meaning whenever the memory tile reads out data from its own SRAM, it is outputted to the output port. Note that this gives priority to its own SRAM. At the end of the chain of memory tiles is a single output port where all of the chained data is used.

The simple implementation of memory chaining means that it can be used to multiplex any two memories. While the original intention was that the memory tiles were chained for capacity reasons, this is not enforced by the hardware. Instead, we can chain together the memories that are needed for each port of the shared compute kernel. The multiplexers that exist within the memory tiles replace the multiplexers that we were originally implementing on the CGRA fabric. The control of

the chaining multiplexers comes from the same schedule that determines when values are read from each SRAM. Since we scheduled the computation so that there is no overlap, there is no conflict with multiple inputs arriving at the same time to a multiplexer, so they can be chained in any order. The output of the final memory can then be connected to the shared stencil registers.

Memory Sharing

With changes to the memory tile, we can reduce the number of memory tiles needed for an application even further. The memory tiles that are chained together sometimes do not utilize the entire SRAM in the memory tile. Due to the downsampling that occurs in a pyramid application, the necessary memory tile capacity decreases with each stage; the last stage may have a very small utilization of the memory tile capacity. The central purpose of sharing the compute kernels was to improve utilization of the CGRA, so we should extend that consideration also to the memory tiles if possible.

The strategy for reducing the number of memory tiles is to store multiple unified buffers in a single memory tile. In this single memory tile, each pipeline stage gets its own exclusive range of addresses in the SRAM. Therefore, multiple unified buffers can be consolidated if the sum of their storage space is less than the capacity of a single memory tile.

There are multiple considerations on capacity to ensure that multiple memory tiles can be stored together. In order to fit multiple memories together, they first have to have small enough sizes to fit into a single memory tile. Line buffers for our stencil applications typically only hold a couple hundred pixels, and pyramid stencils have buffer sizes that shrink by 50% each successive stage. For our memory tiles that hold 2048 words, we can hold the buffers for many stages in a single memory tile. In addition to capacity, the loads and stores for each stage cannot overlap, otherwise there would be a conflict on using the input and output ports for the memory tile. Conveniently, our pipeline stages have been scheduled to not overlap when employing compute sharing.

Once the memories are merged together, the limiting factor becomes the address generator. Instead of mimicking a simple loopnest, the address generator is now responsible for alternating between a set of affine loops. Our current target CGRA only holds an address generator that can perform two affine loopnests independently. This capability is sufficient for creating the piecewise address pattern for merging two buffers together, but in our applications, we could benefit from merging many more buffers together. Providing this increased flexibility should be considered in future CGRAs.

This new strategy of storing multiple unified buffers in a single memory tile can be used in other applications as well. There are many unified buffers that are not able to utilize the entire available SRAM in a memory tile. Therefore, with this new strategy, it would be useful to add a new Halide primitive for *memory sharing*. This new primitive could be used to indicate to Clockwork that multiple unified buffers should be mapped to a single memory tile. Note that a major consideration for sharing memories is whether the memory usage is prone to overlap with each other. Using a

Table 5.1: Evaluation of different compute granularities for shared compute kernels. We see that sharing compute kernels use a single compute kernel over the original spatial scheduling choice. Additionally, sharing at a granularity per line gives a good blend of decreasing the latency, without reordering the input, and without a large increase in memory usage.

Application	Compute Granularity	Compute Kernels	Memory Size (words)	Latency (cycles)
cascade	spatial	2	256	4096
	shared - pixel	1	256	8192
	shared - line	**1**	**316**	**8064**
	shared - block	1	33974	7816
gaussian pyramid	spatial	3	115	4096
	shared - pixel	1	115	4106
	shared - line	**1**	**225**	**4244**
	shared - block	1	1345	4672

single read/write port for an SRAM would greatly hinder two memories trying to access an SRAM simultaneously and would cause the scheduling to increase execution times to ensure there are no conflicts on the SRAM port. However, a different physical unified buffer that uses multiple read/write ports would be able to implement memory sharing more easily.

5.4 Results

Next, we present results for using shared compute with different compute granularities, showing how it saves compute resources with minimal overheads. Below we evaluate our interleaving options based on the resulting memory size and the application latency.

Interleaving Comparison

When sharing a compute kernel, we also choose at what granularity we want to interleave the compute kernel execution. This interleaving choice leads to a question of what trade offs exist between the options. Table 5.1 numerically lists the results for the interleaving choices visually depicted in Figure 5.1 for gaussian pyramid as well as sharing the cascade application.

Our baseline comparison is a spatially scheduled application. These implementations use multiple compute kernels and thus have the lowest latency of 4096 cycles, since there is no contention for the compute units. When we interleave for the cascade application, we see that interleaving at a fine granularity (per pixel) has the lowest memory size and highest latency. The lower memory is explained by the smaller intermediates that are kept throughout execution, since we switch compute kernels to use these values instead of storing them. However, with these compute kernel switches, we also incur the largest latency of 8192 cycles. When we interleave at a coarser level, we also require

Table 5.2: Comparison of cascade and gaussian pyramid implemented with a spatial schedule versus a shared compute kernel interleaved by line. We see an increase in the frames per second per compute kernel for both applications.

Application	Schedule	Compute Kernels	MEMs	Latency (cycles)	Slowdown	fps / compute kernel
cascade	spatial	2	2	4096	-	111
	shared - line	1	2	8064	97%	113
gaussian pyramid	spatial	3	3	4096	-	74
	shared - line	1	3	4244	3.6%	214

that the intermediates are fully buffered. However, we also are able to skip any "dead" cycles used in the finest interleaving. The dead cycles are incurred during the first few lines of computation when the second compute kernel has no work to do. During execution of cascade, the first compute kernel takes 64 cycles for a single row. Since the resulting row output is only 62 pixels, computing a row of the next kernel only takes 62 cycles. In total then, we execute 64 lines with these slightly smaller lines ($(64+62)\times 64 = 8064$). If we interleave at the block level, we only need to compute 60 output lines, since the output of the second kernel is 60×60, yielding another additional savings. We find that interleaving per line has many advantages. Interleaving per line uses minimal memory size overhead due to the efficiency of line buffers, enables shared stencil registers and "free" multiplexers, and provides a small latency benefit from the coarser interleaving size (as compared to per pixel).

For gaussian pyramid, we again find that sharing compute kernels reduces the resources needed for our application implementation. Again we find that memory requirements increase as we have coarser interleaving, with a large jump in memory needed to store entire blocks of intermediates. Recall that sharing by pixel is largely infeasible for our hardware, due to a requirement of input reordering and problematic reuse patterns. Therefore, we focus primarily at the other interleavings. The latency of the gaussian pyramid interleaving is a bit different. We are able to interleave the computation in a more optimized way when scheduled by line. We are able to effectively fill the gaps in the original compute pattern. In fact, we see that interleaving by line with a shared compute kernel takes only 3.6% longer than the original spatial schedule, but also we have a third of the compute resources. The latency is able to come much closer to the spatial schedule in the gaussian pyramid, since many cycles have no active computation for many of the gaussian pyramid levels. Due to these observations, we recommend sharing compute kernels for pyramid applications, and interleaving per line.

Shared Compute Evaluation

We now look at the effectiveness of sharing compute kernels. Based on the results of interleaving above, we now focus just on the interleaving by line. Table 5.2 shows the compute kernels and

Table 5.3: The number of components used for the cascade application with compute sharing. Each optimization shifts the resource usage to a more available resource, or reduces the number of resources.

Optimization	# Kernels	# Shift Regs	# Muxes	# MEMs
no sharing	2	12	0	2
muxes	1	12	10	3
shared shift registers	1	6	10	3
memory chaining	1	6	0	3
memory sharing	1	6	0	1

memory used by each application and schedule. As expected, we see the number of compute kernels decrease when we share compute kernels. We also see that the number of memories does not change since the number of intermediates does not change between our schedules. The schedules are simply changing how we connect these same memories with the compute kernel(s).

In terms of latency, we find that cascade has a 97% slowdown while gaussian pyramid incurs a small 3.6% slowdown. We expect this result, because of the different structures of these applications. Cascade has similar compute kernels to be shared, but there is no utilization issue with the spatial schedule which instead has a high compute occupancy. Thus, when we share the compute kernel, we are increasing our latency. We see that we are utilizing the compute kernels in a similar manner when looking at the frames per second per compute kernel. Sharing a compute kernel for cascade is only beneficial for the reduction in compute resources.

For gaussian pyramid, we find that the latency is only increased by 3.6%. The lower compute occupancy of the pyramid levels means that we have a large opportunity when sharing compute kernels. This increase in compute utilization is best seen in the fps / compute kernel, where we increase from 74 to 214. This is almost a threefold increase since we are achieving almost the same runtime with a third as many compute kernels.

Hardware Optimization

In Section 5.3 we saw how compute sharing can be implemented with different techniques. Each of these optimizations try to use less additional hardware to implement the needed multiplexing. By reducing the overhead of compute sharing, we achieve our goal of reducing compute tile usage with compute sharing.

Table 5.3 shows a count of the hardware components needed for each optimization for the cascade application. Note that not all components are equally as valuable. In particular, memory tiles and compute tiles (for kernels and muxes) are less plentiful than stencil registers. We see from the table that each optimization reduces the number of resources or exchanges a valuable register for a more plentiful resource.

5.5 Extensions

Non-exact Kernel Sharing

The shared compute kernels in this chapter depend on being exactly the same in order to be shared. However, not all kernels are exactly the same. Some kernels are slightly different from one another, but share much of the same structure or computation. Some stencil computations differ just by the weights used, and some pyramid applications have a simpler first computation layer. One useful extension would be sharing computation kernels that have a few differences.

This feature can work by combining similar computation kernels together into a single computation kernel that can do multiple variants. This new computation kernel is slightly larger than each individual kernel, but not as large as the sum. The computation that the kernels have in common is not duplicated, while the differing parts are both created. A multiplexer can then choose which computation is being run at any given time. In this way, we have created a more powerful computation kernel that is capable of doing multiple types of computation based on the configuration that it is given.

This new feature would again affect the entire application compiler. In Halide, one would apply `compute_share` to two compute kernels. However, this time the compute kernels do not exactly match. One could then generate the hardware representation of both kernels, and use a metric similar to the frequent subgraph analysis in [64] to determine what merged compute kernel is best. From here, we would create a merged compute kernel as well as muxes that change their computation intent based on a control signal. During execution, the same signal that performs the memory muxing would also control which compute kernel is executing. This implementation allows a fine-grained switching of which compute kernel is executing at a cycle level, rather than a longer latency if you instead attempted an implementation that utilized fabric reconfiguration. One would be most interested in how fine-grained the execution switching can get, and then evaluate the resulting latency and runtime. Furthermore, since the non-exact compute sharing has additional muxes, we must also consider how much resource overhead is introduced. However, in all we expect this technique to be useful on applications like demosaic, where there are large compute kernels that differ only slightly from each other, and have a lower compute occupancy than the rest of the application.

If we take this concept even further, we could consider fusing compute kernels with little or no common computation. Fusing these dissimilar kernels results in a powerful compute block that does distinct computation, more similar to the generality of an ALU, but on larger groups of computation. Non-exact kernel sharing would allow for such complex components to be mapped onto the CGRA. This application of compute sharing allows the user to choose a design along the continuum of designs that are spatially implemented (unrolled completely) and temporally implemented (sharing of a few time-multiplexed compute kernels). One such use case would be when designing an accelerator that must implement several different applications/kernels. This fusion function would allow for a single

kernel block to be constructed that could be temporally scheduled along with the configuration of which compute kernel to execute at any given period of time.

CGRA-level Compute Sharing

Bringing compute sharing up in the hierarchy, we can apply some of the same principles to sharing the CGRA fabric over multiple DNN layers. For multiple layers of DNNs, we typically have a fixed number of compute resources that are used in a DNN layer, and maintain a fixed structure over multiple layers. Extending the compute sharing framework to multiple DNN layers uses the entire CGRA compute fabric as a single compute kernel. Instead of sharing per pixel or row, we now share the compute kernel over entire executions of the hardware accelerator. Our interleaving is elevated to a loop level that encompasses the full DNN layer execution. Therefore, a full DNN layer executes sequentially before continuing to the next layer.

The compute kernel in this case is a little different from those earlier in this chapter. Instead of just PE tiles being shared, we also want to include the memory tiles to keep the same connection structure across DNN layer executions. However, these layers have different configurations where the loop lengths are different for each DNN layer. In order to switch between these non-exact matching kernels, reconfiguration can be used.

The memories feeding the shared compute kernel are elevated from memory tiles to global buffer banks. The successive DNN layers are stored in the global buffer and are multiplexed into the CGRA fabric using the programmable global controller. Because of the size of the input, weight, and output buffers, it becomes necessary to reclaim global buffer banks after their use to ensure there is enough capacity for all of the data buffers.

There are several parallel concepts that we see between compute kernels and multiple DNN layer execution. The execution becomes a demonstration of the global buffer instead of the memory tiles. And the benefit becomes the consistency in CGRA routing structure between CGRA layers.

5.6 Summary

Sharing compute kernels aims to optimize computation in pyramid applications by sharing a single compute kernel for multiple stages while carefully interleaving their execution. This approach maximizes compute resource utilization and minimizes memory requirements. We extend Clockwork scheduling to accommodate the new execution order, ensuring that shared compute kernels run efficiently without overlapping. We also adjust hardware generation to represent shared kernels. Chained memory tiles then multiplex between different input sources, leading to minimal hardware overhead to implement compute sharing. Hardware sharing represents a scheduling primitive that is exclusively for hardware. Users can employ hardware sharing to optimize their target performance.

Chapter 6

Halide to Hardware System: Evaluating the Compiler

In the previous chapters, we have focused on the first few steps of the compiler. The major focus of my research was on these steps. However, my work is part of a larger system [51] that compiles Halide applications onto our CGRA. We first look at a brief overview of the full application compiler developed by our Stanford Agile HArdware (AHA) group. We again start with Halide, but continue our description of the application compiler with PnR and hardware execution on real silicon. This full system description gives the context necessary for understanding some of the later evaluations. Next, we describe the design philosophy for constructing our system, including our testing strategy of handcrafted concepts, unit tests, and full applications. While creating these application schedules, I found common strategies for creating schedules to create good implementations on the CGRA. We define these evaluation metrics and then list my recommendations on how to schedule to the CGRA. Finally, we list our motivating image processing and DNN applications, show the final CGRA schedule parameters, and evaluate the applications.

6.1 Compiler System for CGRA

This section describes the steps for compiling a Halide application to the CGRA depicted in Figure 6.1. The first steps revisit Halide applications and mapping to the unified buffer abstraction. The next compiler step maps the unified buffer onto CGRA hardware. Then, the application is placed and routed onto CGRA tiles. And finally, a configuration bitstream is created for the full application. Below, we briefly describe each of these compiler steps.

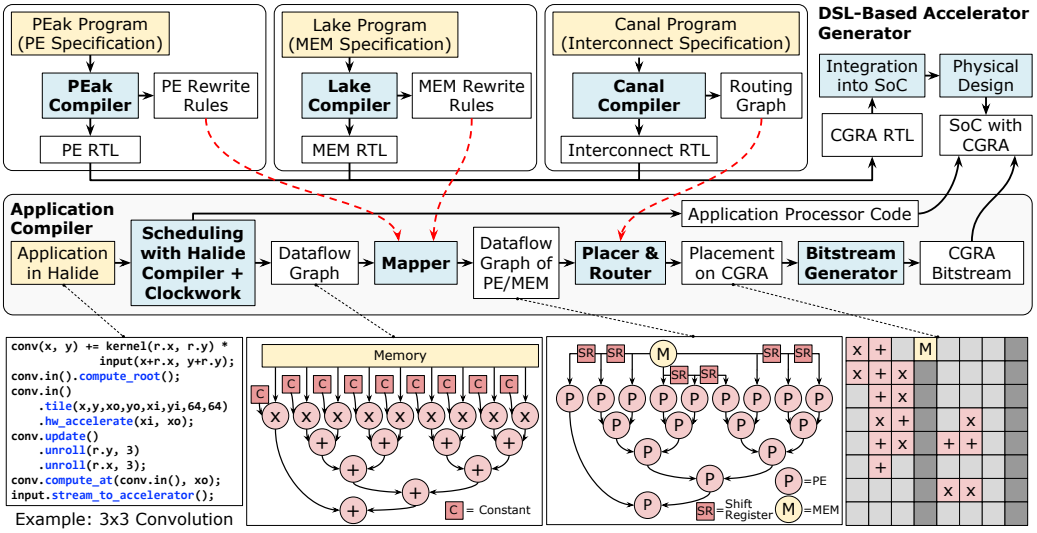


Figure 6.1: This figure shows all of the steps in the application compiler.

Halide Algorithm and Scheduling

First, an image processing or machine learning application is defined in Halide. As we saw in Chapter 3, the Halide algorithm encodes the user's target application. Here the mathematical operations performed on the data are translated into Halide operators. Halide provides relative indexing and reductions to make common computation patterns easier to express.

The purpose of the Halide scheduling is to optimize our generated algorithm. The scheduling primitives change *how* the algorithm is computed on hardware. When targeting the CGRA, we are typically looking for a schedule that is as fast as possible by using as much of the CGRA as possible. Furthermore, we use scheduling to match the constraints of the hardware resource units, such as the SRAM capacities and memory hierarchy structure.

Halide Compiler and Codegen

After the application has been fully created using the algorithm and schedule, we need to create the files for Clockwork, our memory mapper. As we saw in Chapter 4, each of the Halide compiler passes modify the HalideIR to better match the expected format that Clockwork wants. During codegen, the application is separated into the memory operations and the compute kernels. The generated memory file contains loopnests to describe the iteration domain, and indexed memory stores for the access maps. During Clockwork scheduling, the compute kernels are largely ignored, except for any cycle delay incurred in any compute kernel.

Clockwork Scheduling

Clockwork first fuses stages together, and then calculates the timing of each memory operation. As we saw in Chapter 5, the type of fusion can be automatic or directed by Halide tags. Stencils typically have compute stages that all can be run simultaneously, which leads to all compute stages being fused together at the innermost loop. DNN pipelines and shared compute kernel pipelines use coarse-grain fusion.

Once the pipeline stages are fused together, we calculate the timing of memory operations. The access maps of each memory dictate the dependency chain from inputs until the output stage. Furthermore, the iteration rate of each of the pipeline stages changes how often each of the stages performs its compute kernel. Using the dependency chain and iteration rate, we create a polyhedral model in Clockwork to calculate the schedule of each compute kernel in the application. After Clockwork scheduling, we have the complete information for the unified buffer. With the unified buffer, our application is fully specified for our hardware execution.

Clockwork Memory Mapping

In Clockwork memory mapping, the unified buffer specification is mapped to the target hardware. Clockwork is capable of targeting different hardware targets, such as an FPGA or CGRA. For my research, I focused mainly on the CGRA that was being constructed in our group. The unified buffer abstraction is meant to serve as a high-level representation of the application that any hardware accelerator could implement. However, each specific hardware designer might make different choices to create their hardware accelerator.

The hardware accelerator that our group designed, called Amber, has several important parameters for mapping. First, there are the sizes of components. The capacity of the global buffer and memory tiles limit how much can be stored. The address generator for the memory tile is configured for six-level affine loops. Finally, there are two input ports and two output ports for each of the memory tiles.

Each of these limitations in the hardware comes with hardware solutions to expand them for unified buffers that need more resources. The memories can be chained together in order to expand their size. Memory chaining combines the total SRAM space of multiple memory tiles. If a unified buffer uses more space than a single buffer, memory chaining enables memory tile data to be connected through the chaining input of one memory tile to another.

Nested loops can be coalesced given that the outer loop stride is equal to the extent of the inner loop. This helps when there are many loop levels in the iteration domain. When doing computation for neural network layers, there can be many levels of tiling and splitting where six levels are not enough. However, we can combine loops together using this coalescing technique.

The number of input and output ports required by each unified buffer can be solved using shift registers and banking. Each memory tile has just two input ports and two output ports. However,

many output ports are typically needed for stencil computation. A 3×3 stencil computation requires nine output ports, which far exceeds the two available output ports. However, by analyzing the access pattern of successive computation, we find that many of the output ports are reused from the last computation. Instead of rereading these values from the memory tile, we can locally store them in stencil registers. In fact, the purpose of the access maps being parametric indices is to make this stencil register optimization easier.

Another means of increasing the bandwidth of our memory is banking. This process analyzes the access maps and determines if the access maps are disjoint from one another. When they are disjoint and require more memory bandwidth, then they are separated into their own memory tiles. By storing them in their own memory tiles, the number of available memory ports increases for that unified buffer.

One might wonder why two input and two output ports are useful. One of the primary uses of the output ports is that a 3×3 line buffer can be implemented using a single memory tile by using the two output ports and an input stream to provide data for the three needed lines. Another use case is when accumulating values for DNNs. A memory tiles for a DNN go through three phases: initialization, accumulation, and read-out. Each of these phases needs its own ports and access patterns. Therefore, we use the four IO ports to cover all of the phases (one input port for initialization, an input and output port for accumulation, and one output port read-out).

The final part of mapping is to create the full application graph in terms of memory tiles and processing elements. The MEMs have been given the correct read/write timings, and the PEs have been configured with the correct compute operations. The connections of the memories to the compute kernels was specified during Halide codegen. Thus, all of the tiles can be wired together into the final CoreIR hardware design.

CGRA Placement and Routing

With our final graph of PEs and MEMs, we now need to create these physical connections on the CGRA. The CGRA fabric contains PEs and MEMs that are connected by configurable switchboxes and connection boxes that act as programmable wires. Next, we need to choose which PEs and MEMs in the application design will correspond to which physical locations on the CGRA fabric during placement and routing (PnR). Our group built a specialized PnR tool [66] that does placement and routing for our CGRA.

The first step in this process is *packing*. The PEs contain not just a compute operation, but also several registers. Registers that exist in the compute kernel can be packed into a PE. Furthermore, any constants (such as stencil taps), can be placed within a PE along with the compute operation. Each PE also has an independent 3-input LUT that is used to do logical operations such as comparisons and 1-bit operations. A LUT can be packed together within a PE that is using input registers and its 16-bit compute.

The next step is *placement* where tile locations are chosen on the CGRA fabric. Placement is done hierarchically, with global placement followed by detailed placement. Global placement places tiles using half-perimeter wirelength (HPWL) to determine the cost of a placement. In order to reduce HPWL cost, tiles that are highly connected with each other naturally favor shorter wire lengths. This is a good proxy for our true goal of reducing latencies from long wires. In detailed placement, simulated annealing refines the global placement to a fully legal placement of tiles using an iterative approach. A cost function again is used to reduce wire lengths and decrease the number of tiles that are used only for routing. By discouraging tiles used only for routing, the generated placement is able to power gate more tiles on the CGRA fabric.

The last step is *routing*. The tiles that are connected to each other must find an unused path through the switchboxes. This step is done iteratively, starting from the shortest wiring possible. The shortest wiring likely has overlapping wires, in which case these wires (nets) must be rerouted. Solving routing congestion is done by rerouting the nets with short wire lengths (most positive slack). This allows the most timing critical nets (longest wires) to use the most congested paths.

Pipelining

After placement and routing, our application has been fully created on the CGRA; however, it cannot run at the top frequencies, since we never incorporated timing information. Pipelining the application involves determining the locations where the delay through the compute units, memory units, and switchboxes exceed the frequency target of the CGRA. Our research group built a toolkit [65] that pipelines our CGRA applications.

Registers are placed throughout the CGRA design in order to meet the runtime clock frequency. Our CGRA was built where the delay through a single compute unit is one clock cycle, so we start by placing registers before and after each compute unit. However, since there are a different number of compute units on each path through the computation graph, additional registers must be placed to balance the delays. Even with the computation pipelined, some paths still can be too long. This is common for the wire that connects the global flush to all of the memory tiles. These remaining long wires are pipelined to ensure all wire delays fit within a clock cycle.

Once additional registers have been placed within the design, the scheduling from Clockwork must be redone. The scheduling in Clockwork assigns precise clock timings, so the extra delay through compute units needs to be accounted for. In the Clockwork memory file, we annotate the calculated delays in each part of the application. Then, Clockwork reschedules the application so that the memories properly load and store data given the pipeline delay registers.

CGRA RTL Simulation

With the entire application mapped to the CGRA, we can simulate applications. Our system uses a script to generate a configuration bitstream based on the mapped application. During configuration,

the global buffer sends the configuration register values to each of the CGRA tiles and switchboxes. We are able to simulate the entire configuration process as well as the CGRA execution using an RTL simulator. The RTL simulator uses the same input image as the CPU simulation, so we can verify correct execution by checking that the output images/data are identical.

CGRA Hardware Execution

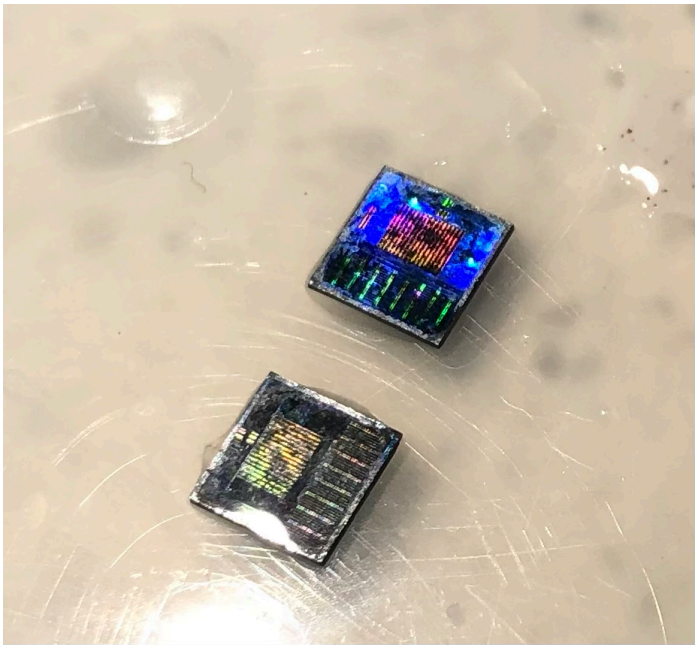


Figure 6.2: This is the CGRA hardware on a 5×5 mm die. We depackaged and delayered the chip to show the components: a central global buffer and CGRA tiles (eight column-groups of 1 MEM column + 3 PE columns).

The final step is running the application on the real CGRA hardware. Our research group is fortunate to have the hardware taped-out (as shown in Figure 6.2) so that we can test that the hardware design really works. To run the application on the hardware, there are some extra steps to transfer the data using DMAs onto the global buffer itself. From there, the application can be run on the hardware. With correct execution, the same input image will calculate the same output image, verifying that the execution is correct. This additional execution on hardware is important for truly testing that the routes all make timing.

6.2 System Design Methodology

A key part of this project was to create an automated software system that can compile applications to our CGRA. Since we were creating both the applications and the compiler, being able to test our applications at many stages through the compilation flow was essential. The code is written to make it easy to build each application and debug it. In the front-end compiler, a Makefile system is used for each file or action that a user wants to run. The dependencies of each Makefile rule allow slight modifications of the generated files, and then future objects that depend on these files are remade. This allows for easier debugging. The output files needed for Clockwork are created using `make clockwork`. Running this command in an application directory leads to the compilation of the Halide generator, and execution of the generator to create the input of Clockwork code.

With so many components in the compilation process, we found it useful to generate and test all of these parts. We created a Makefile with targets for each of the intermediate outputs. By creating a Makefile with properly defined dependencies, we can request the final output to be made, and all of the intermediate targets and final output are subsequently generated. Each of the projects in the application compilation process, from Halide to bitstream generation, use a set of scripts that are automatically called, similar to a Makefile. Each of these scripts are executed without user input or modification. A user is encouraged to simply write an application using a Halide algorithm and schedule, and then use a one-button command to compile it to hardware. This ease of use lends well for integration with automatic testing frameworks. Our projects are cloned, installed, and tested using continuous integration scripts using a triggered git repository.

Our testing strategy was to run applications at intermediate stages in the compiler. For each stage in the compiler, we should be able to feed in input data and retrieve output data. Given the same input data, we expect the output data to match exactly. At several intermediate points in the compiler, we have the capability of running the application in this manner:

- **Halide Reference**: golden reference model using the same algorithm run on a CPU
- **Clockwork Simulation**: input to Clockwork using unfused, sequentially run memories
- **CoreIR Simulation**: output of Clockwork using configured memories and compute operators
- **Mapped RTL Simulation**: design after placement and routing using a configured CGRA
- **Pipelined RTL Simulation**: mapped design on the CGRA with pipeline registers
- **Hardware Execution**: application running on real silicon hardware

The first testing target is Halide's default CPU target. Our system is built on an existing compiler from the Halide front-end to a CPU. We depend on the CPU implementation as a golden reference model. Using a CPU schedule, we can compare the performance of the CGRA to the CPU

execution. For verification that the output is correct, it is sufficient to use the CPU default schedule. For BFloat bit-exact comparisons, it is critical to perform the same loop optimizations since floating point arithmetic is not commutative.

When compiling to the hardware accelerator, we create several files for execution. The execution of the hardware accelerator is performed within a tiled loop in the host CPU code. This means that a file for the CPU is needed as well as the CGRA application execution. The Halide application is compiled and creates the host CPU wrapper in addition to the accelerated code. The production of these auxiliary files leads to the hardware execution being consistent with the original Halide application. The Halide application uses png image files, while the hardware accelerator uses raw bytes (from a pgm file). While the file formats differ, the generated accelerator wrapper code performs this conversion to ensure that the same data is used for each execution.

Once we create the input and expected output data, each step of the compiler is checked with a test execution. The input file is fed into the application execution, and each intermediate step of the compiler creates its own output file. The Halide application is codegen'ed to Clockwork C files, and the generated code is tested using this method. Testing gives us confidence that the generated application does not contain any logical differences introduced by the compilation of the original Halide application. The same testing and assurances are given as we test the application in each step through the application compiler. With all of these opportunities to execute the design, we are able to narrow down where a compilation bug has occurred.

Additional testing collateral is also useful for debugging applications with an issue in the compiler. Once the application is fully mapped to the CGRA, we can convert the hardware graph into a visualization. We use graphviz [30] and create a dot graph with arrows between wired components. Figure 6.3 shows an example visualization for the Harris application. This way we can quickly find obvious bugs through inspection, as well as understand what operators and connections have been created for our application.

6.3 Applications

In Halide, we created a suite of applications to test our compiler and the CGRA. Each of our applications use the Halide front-end to create a golden reference model, and each application can run through the entire compiler toolchain to the CGRA. We have three types of Halide applications:

- **handcrafted**: tests with user-implemented CoreIR designs
- **tests**: isolated usage of CGRA operators or computation patterns
- **applications**: implementations of real algorithms creating usable output

Handcrafted tests are ones where the compiler does not currently work. We used this early on to hand-create designs before spending effort on handling them in the compiler. This would involve

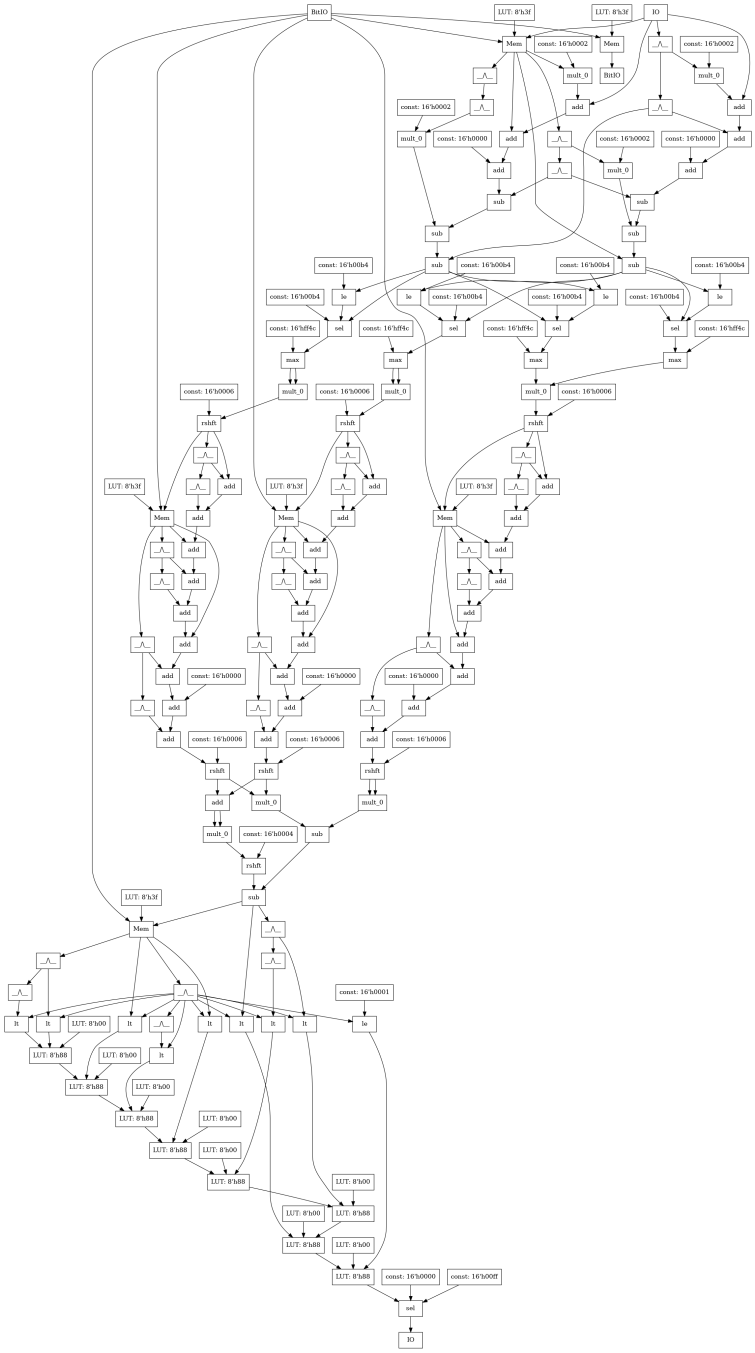


Figure 6.3: A dot graph made by graphviz show the connections between PEs, memory tiles, and registers. This shows the components on the CGRA fabric after memory mapping. Common symbols are used for PE operators while memory tile and register names are used for memory components. IO denotes the input and output stream of the application.

creating a Halide application. Then, instead of running the compiler, I would create the output of the Clockwork scheduler and mapper (namely, a CoreIR file with the scheduled memories and compute tiles). This strategy helped validate that certain schedules would work on the CGRA before implementing the lowering and optimization passes in the compiler.

Tests are simple or isolated versions of features that exist in applications. There are tests for each of the operators in Halide. These are meant as basic tests that can ensure that the hardware implementation is correct. The benefit of using a small unit test is isolating the issue to a single operator rather than debugging a full application. Furthermore, there are tests for subsets of an application (such as a difficult individual kernel) or a computation feature (such as stencils). These test the basic compiler functionality for a new feature as well as ensure that the hardware works for a simple case.

Applications are full algorithms using operators in a useful manner. The input is a reasonable set of values, and the output produces useful values. Our target set of applications are image processing and machine learning applications. The image processing applications use real photos (some of which are taken from my own camera) and apply traditional image processing algorithms. The resulting images can be visually inspected to ensure that the algorithm is performing the application. This means that gaussian creates a blurred image, harris creates an image of corner locations, and camera_pipeline produces an image from raw camera data (visually shown in Figure 2.1). Applications are created for real algorithms and to evaluate the system.

6.4 Halide Scheduling Strategies for CGRAs

In the previous chapters, we saw how Halide scheduling was extended so that an application can be mapped to hardware accelerators. However, these scheduling primitives require the user to appropriately place these scheduling primitives on their algorithm. Additionally, each scheduling primitive takes arguments (such as a tile size or unroll quantity). Below is guidance on where I have found these scheduling primitives most useful, and how to tune different arguments to maximize common performance metrics, such as application execution time.

Buffering Intermediate Values

We see that almost all values for image processing pipelines and machine learning pipelines should be buffered to avoid recomputation. Image processing line buffers are very efficient and there are plenty of memory tiles on the CGRA, so a line buffer should be placed between every pair of compute kernels that could use one. Code 6.1 shows a sequence of three schedules where buffers are successively added to the Harris corner algorithm. The algorithm is not too important for this illustration, but instead one should notice that we iteratively add more buffers. Harris has a depth of four compute kernels with `lgxx, lyy, lgxy` in the middle. We first have no buffers, then add

```
if (sch == 1 || sch == 2 || sch == 3) {
    // Add accelerator interface (no buffers).
    hw_output
       .tile(x, y, xo, yo, xi, yi, tileSize, tileSize)
       .hw_accelerate(xi, xo);
    padded16.stream_to_accelerator();
}

if (sch == 2 || sch == 3) {
   // Add three buffers in the middle of the application design.
   lgxx.store_at(hw_output, xo).compute_at(hw_output, xo);
   lgyy.store_at(hw_output, xo).compute_at(hw_output, xo);
   lgxy.store_at(hw_output, xo).compute_at(hw_output, xo);
}

if (sch == 3) {
   // Add all of the buffers to the application design
   grad_x.store_at(hw_output, xo).compute_at(hw_output, xo);
   grad_y.store_at(hw_output, xo).compute_at(hw_output, xo);
   lxx.store_at(hw_output, xo).compute_at(hw_output, xo);
   lyy.store_at(hw_output, xo).compute_at(hw_output, xo);
   lxy.store_at(hw_output, xo).compute_at(hw_output, xo);
   cim.store_at(hw_output, xo).compute_at(hw_output, xo);
   cim_output.store_at(hw_output, xo).compute_at(hw_output, xo);
}
```

Code 6.1: Halide schedule variations for Harris. The `sch` variable chooses which blocks of scheduling code are applied to each algorithm. Each schedule successively adds more buffering to reduce needed recomputation.

the buffers in the middle, and finally add all of the possible buffers. Table 6.1 shows the number of PEs, MEMs, and runtime with these three schedules. Notice how many PEs are saved by simply adding in some buffers. These intermediate buffers dramatically reduce the number of PEs with negligible increases in number of memory tiles and runtime.

Note that every computation kernel can be buffered, even element-wise operations. For compute kernels that do not need any buffering, the memory mapping analysis will recognize that no memory is needed and create a "memory" with no capacity, meaning a wire is created instead. This optimization occurs during memory mapping in Clockwork during dependency analysis. Due to this optimization, the user should not be worried about creating too many buffers, since any unneeded buffers are optimized away. The only drawback is slightly more compile time for Clockwork analysis.

Table 6.1: Compiler results for Harris application with different Halide schedules. Each subsequent schedule adds an additional memory using `store_at().compute_at()` as shown in Code 6.1.

Harris Schedule	# PEs	# MEMs	Runtime (cycles)
sch1: recompute all	769	3	4097
sch2: recompute some	145	5	4103
sch3: no recompute	83	5	4146

Buffering dramatically reduces the number of PEs needed, and buffering elementwise operators is optimized to wires. Due to these observations, I recommend to:

> **Recommendation 1:** Buffer after every compute kernel, even if you are unsure if it is necessary. This will prevent expensive recomputation, and any unnecessary buffering will be optimized away.

Tiling due to Memory Constraints

Once buffers are placed, we have the problem of how to fit the intermediates in memory. Our memory tiles on the Amber CGRA are limited to a capacity of 2048 kB. We can tile the output image, which in turn tiles all intermediates and inputs in Halide. These tiles help each unified buffer fit within an SRAM. However, how large should tiles be? We can tile computation with many small tiles or a few large tiles.

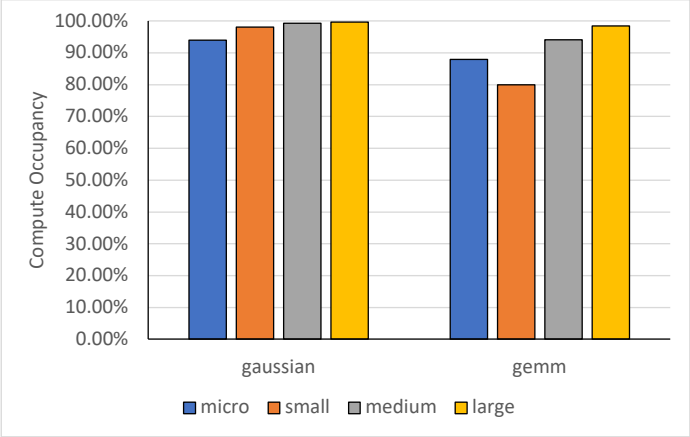


Figure 6.4: Scaling based on an increased image tile size. Compute occupancy is greatest for large input tiles for both gaussian blur and gemm.

Figure 6.4 shows how compute occupancy changes as we increase the size of a tile for two applications. Compute occupancy measures how often the hardware accelerator is outputting useful pixels. We find after scaling that larger tiles have better compute occupancy. Compute occupancy drops when computation does not produce useful output data. For image processing, the end of each input tile leads to several cycles of invalid outputs as the stencil computation wraps from one input line to the next. The fewer times the image wraps, the better the compute occupancy. Larger input tiles have a smaller percentage of the image classified as an edge. Thus, we prefer larger tiles.

One drawback of larger tiles are their longer compile time. Table 6.2 shows the compile times for gaussian and gemm. Compile time is longer since we must execute a full tile during Clockwork

Table 6.2: Compile times (in seconds) for different tile sizes. Compile time increases for larger tiles.

application	tile size	total compile time	app compilation	app generation	clockwork sched
gaussian	micro (64 × 64)	33	2.58	18.5	12.2
	small (240 × 180)	52	1.74	18.7	32.0
	medium (640 × 480)	183	2.55	18.3	162.5
	large (1920 × 1080)	1040	2.53	18.4	1018.9
gemm	micro (32 × 32)	33	2.58	17.0	13.8
	small (128 × 128)	38	2.39	16.9	18.7
	medium (512 × 512)	373	2.39	17.3	353.3
	large (2048 × 2048)	22386	2.42	18.1	22365.8

scheduling. However, generally we accept the longer compile time and prefer a solution with the greatest compute occupancy. Therefore, I recommend to:

> **Recommendation 2:** Tile algorithms with tiles as large as possible that fit within your memory capacity constraints. Large tiles have a greater compute occupancy, leading to a shorter total runtime for the entire application.

Code 6.2 shows how tiling is used on an application. Notice that the sizing creates a unified buffer that fits within a single 2048 kB memory tile. Furthermore, the output is tiled such that an integer number of tiles covers the entire output image. This ensures that each full run of the accelerator produces useful output values rather than computing on partial images.

```
// Input image size: 6000 x 4000
// Output tile size: 600 x 400
// Number of executions for full image: 10x10

int tileWidth = 600;
int tileHeight = 400;
hw_output_mem
  .tile(x, y, xo, yo, xi, yi, tileWidth, tileHeight)
  .reorder(z, xi, yi, xo, yo);

int glbWidth = tileWidth;    // GLB size same as tile
int glbHeight = tileHeight;
hw_output_glb
  .tile(x, y, xo, yo, xi, yi, glbWidth, glbHeight)
  .reorder(z, xi, yi, xo, yo);
```

Code 6.2: How to tile memories to fit on the CGRA. This sample code tiles the upsample application with $10 \times 10 = 100$ iterations to create a 6000×4000 output image. Due to line buffering, only a few lines are needed for the memory tiles; but the GLB holds the full tile for an accelerator execution.

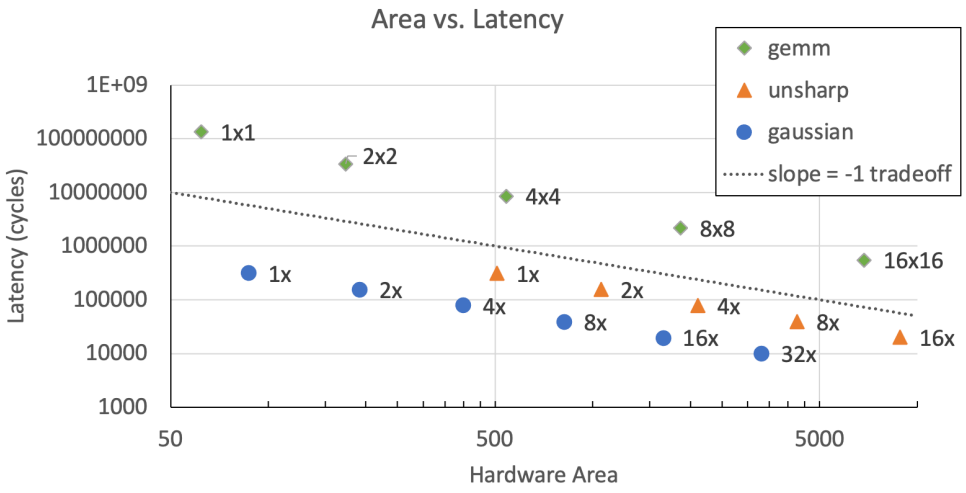
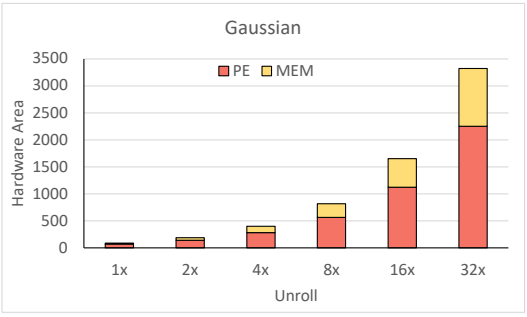


Figure 6.5: Unrolling compute hardware increases area and decreases latency by equal factors. The dotted gray trendline shows the slope of an equal factor of latency decrease for an increase in hardware area. All three applications follow the slope of this line.

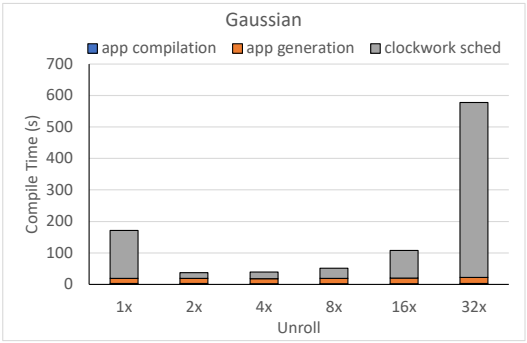
Unrolling to Duplicate Hardware

Besides buffering memory, another essential part of a hardware accelerator is using the available compute resources. For our Halide applications, we use the `unroll` scheduling directive to perform hardware duplication. Unrolling a loop increases the number of compute resources used in the loop body which in turn decreases the overall runtime, but it also affects other performance metrics.

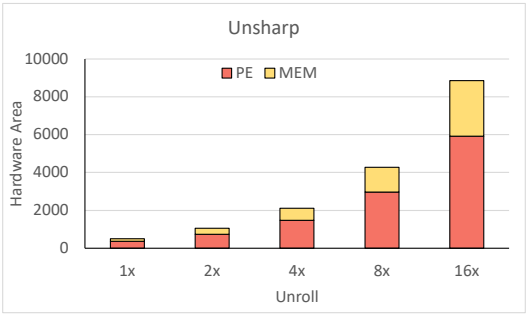
Figure 6.6 shows how unrolling compute affects the total hardware area as well as total compilation time. Each application uses a fixed input size, and then is scheduled with a varying unroll factor shared across every compute kernel and IO unit. Under this experiment, we start and end with values in the global buffer, which ignores the memory bandwidth bottleneck between the host and GLB. As expected, there is a linear increase in PEs and MEMs used as we increase the unroll factor. By unrolling, we create a design with better spatial utilization and faster execution time. Figure 6.5 shows how larger designs using more compute units decrease the latency by a proportional amount. This means that this usage of area has a direct benefit on the resulting latency, and is a great trade-off to make. That is, if an application pipeline fits on the CGRA fabric, one should also consider unrolling all compute kernels to increase throughput and duplicate GLB/CGRA connections to increase utilized memory bandwidth. Note that this analysis ignores the memory bandwidth that exists between the host processor and GLB. This bandwidth is difficult to simulate, but for demonstrations on real hardware the bandwidth between host/GLB is limited and becomes a bottleneck once all GLB/fabric is used. It becomes difficult for the host/GLB to keep up to this rate.



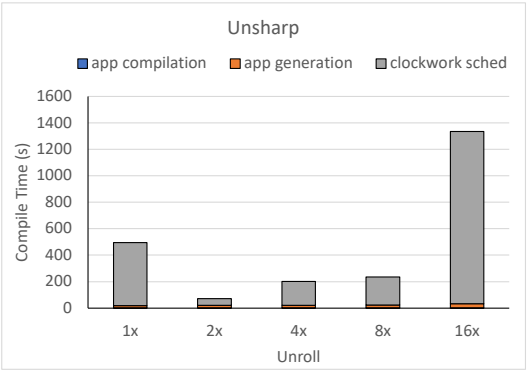
(a) gaussian hardware area



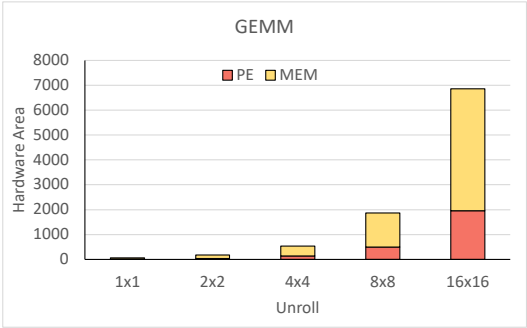
(b) gaussian compilation time



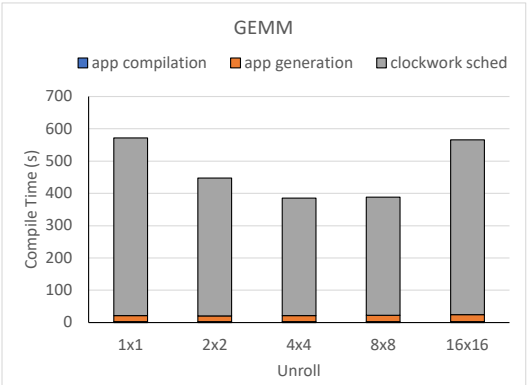
(c) unsharp hardware area



(d) unsharp compilation time



(e) gemm hardware area



(f) gemm compilation time

Figure 6.6: Hardware area and compilation time for gaussian, unsharp, and gemm as they unrolled. Gaussian and unsharp use input images sized 640×480 while gemm computes on 512×512 matrices. Unrolling applications increases the hardware area to implement them on the CGRA. Additionally, the compile time increases for larger hardware designs, mainly for Clockwork scheduling.

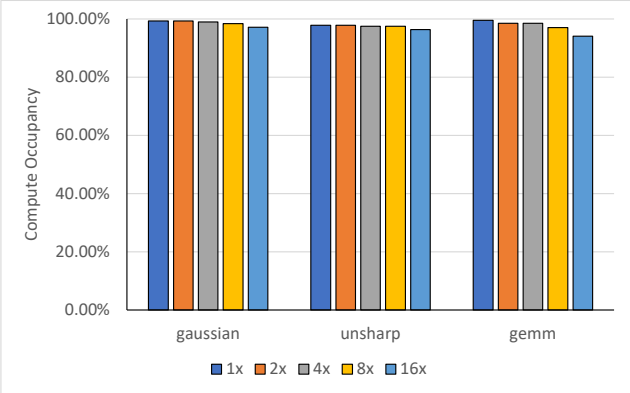
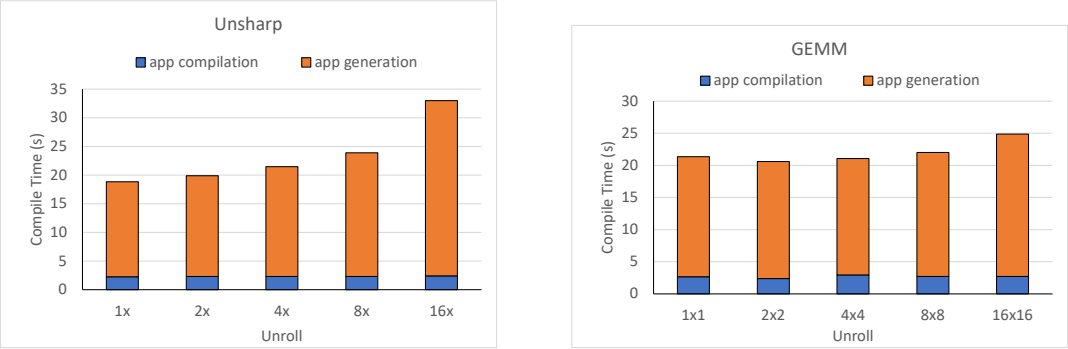


Figure 6.7: Compute occupancy with increasing unroll factor. A higher unroll factor has a lower compute occupancy for every application.



(a) unsharp front-end compilation time

(b) gemm front-end compilation time

Figure 6.8: Compilation time for the first two steps of the compiler. Modest increases in compile time as each application is unrolled more.

However, unrolling the application also slightly decreases temporal compute occupancy and increases compile time. Figure 6.7 shows the decrease in compute occupancy with increasing unroll factor. While keeping the input size fixed, unrolling a loop decreases the effective input tile size that each compute unit sees. Based on the findings on image size in the last subsection, we expect and observe a decreased compute occupancy with these smaller tile sizes.

Figure 6.6 shows the total compilation time for each application with increasing unroll factor. Note that Clockwork scheduling includes a required execution of the accelerator. We find that compilation time at first decreases and then increases. This is because Clockwork scheduling has an increased runtime for both longer loop iteration lengths as well as more hardware. When the algorithm is unrolled with a small factor of 1, the iteration time is large. When the unroll factor is a large factor of 16 or 32, the large amount of duplicated hardware causes long Clockwork scheduling times. In the middle with 2× or 4× unroll factors, we see a sweet spot with lower compile times.

The compile times in Figure 6.6 are dominated by Clockwork scheduling and it is difficult to see the compilation trends for the first two stages. Therefore, Figure 6.8 shows the compilation time for just the first two front-end compilation steps. The front-end compilation times have a modest increase with unroll factor, but not as drastic as Clockwork scheduling. The front-end compilation in Halide is not as affected by large designs as Clockwork scheduling is.

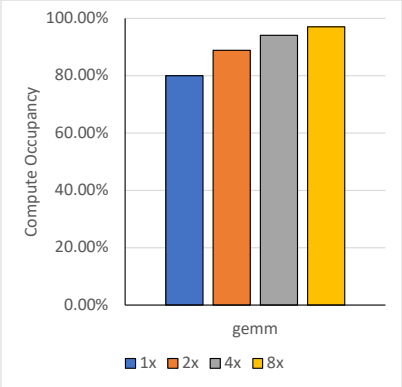


Figure 6.9: Increasing IO unroll factor on gemm using a compute unroll of 8×8 on 512×512 matrices. Greater IO unroll factor helps keep the compute kernels busy.

Although it is easy to focus on unrolling compute, it is also critical to unroll IO ports as well. Duplicating IO ports ensures that the compute and memory units are supplied with enough data so that they do not idle. Unrolling IO supports the compute rate for image processing pipelines, while unrolling IO for DNNs reduces the initialization and read-out phases. Figure 6.9 shows how compute occupancy of gemm increases as IO is unrolled with a greater factor. The target design has 8 input and 8 output channels unrolled for the compute kernel. The IO ports are then unrolled from 1 to 8. Unrolling the IO up to the rate of the compute kernel gives steady increases in the compute occupancy.

We observe that unrolling as much as the reconfigurable fabric can handle is best. When you try to unroll your application, you must consider how much compute and memory bandwidth is available. One will find that each application will be limited either by the amount of compute (PEs) or memory bandwidth (GLB IO tiles) based on the application. The arithmetic intensity is a calculation of how many arithmetic operations are needed for each transferred byte. This calculation helps calculate if an application will be limited based on compute or memory bandwidth once it is unrolled to the highest extent. Overall, unrolling decreases execution time by using the available compute units. The purpose of having a large compute array and many GLB IO tiles on a hardware accelerator is to spatially compute an application. Therefore, I recommend to:

Recommendation 3: Unroll compute kernels to fill the CGRA compute fabric as much as possible. Additionally, unroll IO to match the throughput of the compute kernels. Based on an application's arithmetic intensity, one will reach the limits of the compute or memory bandwidth resources on the CGRA.

Code 6.3 shows how we can unroll a DNN layer to create multiple MACs and use multiple IOs.

```
// Example tiling in conv3_1 layer of ResNet
// Unroll compute convolutions
output_cgra.update()
  .unroll(r.x).unroll(r.y);  // Unroll the convolution reduction fully

// Additionally, unroll input and output channels for compute
int k_oc = 8;
int k_oc = 8;
output_cgra
  .unroll(w, k_oc);  // Unroll the output channels partially
output_cgra.update()
  .unroll(w, k_oc)
  .unroll(rz_unroll, k_ic);  // Unroll the input channel reduction partially

// Unroll IO streams to match rates
int glb_o = 1;  // No duplication for output stream for this particular layer
int glb_i = 4;
hw_output.unroll(w, glb_o);
output_glb.unroll(w_cgra, glb_o);
input_glb.unroll(z, glb_i);
input_cgra.unroll(z_cgra, glb_i);
```

Code 6.3: Different use cases of unroll: duplicating convolution reductions, duplicating compute across channels, and matching rates for IOs.

Sharing Compute Kernels to Save Resources

The previous chapter, Chapter 5, described how compute sharing can be added to a Halide application's schedule.

We observe that compute sharing can be applied to pyramid applications and matching compute kernels to reduce necessary compute resources for large applications. When applications additionally have downsampling and reduced temporal usage of their compute kernel, this can be performed with minimal reduction in execution time.

Recommendation 4: Use compute sharing to increase utilization of compute elements on the hardware accelerator. The reduced resources are freed up with minimal increases in execution runtime. Compute sharing is most easily applied and has its largest impact on applications with pyramid structures.

Buffering to Create Memory Hierarchies

The Halide scheduling we have described so far are helpful strategies for making the most of the target accelerator. However, a hardware accelerator has a fixed memory hierarchy that must be followed. Following Halide's philosophy, we do not need to describe this target-specific hierarchy in the algorithm. Instead, we can use scheduling to alter the algorithm to accommodate the CGRA's memory hierarchy.

The CGRA's memory hierarchy consists of a host DRAM, global buffer, memory tiles, as well as small register files on each PE. Memory transfers between these entities should be depicted in our IR so that they can be mapped to our CGRA interconnect. Halide's `in()` schedule is used to represent these transfers in the memory hierarchy as demonstrated in Code 6.4. Each successive call to `in()` on a buffer refers to another copy of the same value. Using this scheduling primitive, the original buffer (`buf`) is the producer for an exact copy of its values (`buf.in()`). This scheduling primitive simply creates a statement that copies from one buffer to another with the understanding that each buffer copy will be mapped to its own memory in the hierarchy. For example,

```
hw_output.in()
```

refers to the output values on the host level while `hw_output` refers to the output values on the global buffer. Note that the call to `in` results in a copy that is after the Func. This means for the output, the copy (`hw_output.in()`) is an outer memory level. When we perform this scheduling copy on the input, we will similarly see the copy be performed after our Func (`hw_input.in()`), meaning an input copy will be on an inner memory level. Note that when we create these memory copies, we may use a different number of `.in()` based on how many copies we need. In the example in Code 6.4, the algorithm side of the output already contains the MEM level in the algorithm, so we have fewer `.in()` calls than the input.

Besides creating memory copies, it is also important to label buffered intermediates as memory elements in a particular part of the hierarchy. This is solved by using the parameter of `store_at()`. The `store_at()` scheduling primitive dictates that a memory primitive should be created while the first parameter (Func specified using the correct number of `in()`s) determines at which level in the hierarchy it should be placed. For example,

```
conv2.store_at(hw_output, xio)
```

specifies that `conv2` should be stored in an accelerator memory. We can then either rely on Clockwork to decide whether the buffer should be mapped to a memory tile, or a smaller Pond. Alternatively, we can add Halide scheduling with `store_in` to decide this as a user decision. The `store_in` scheduling primitive annotates the Func with this user decision to override any analysis that Clockwork may use to determine how to map a buffer.

```
// Output tiling and output stream
hw_output.in()        // host level
  .compute_root()
  .tile(x, y, xo, yo, xi, yi, 360, 360)    // GLB level sized 360x360
  .hw_accelerate(xi, xo);
hw_output             // GLB level
  .store_in(MemoryType::GLB)
  .tile(x, y, xio, yio, xii, yii, 60, 60)  // MEM level sized 60x60
  .compute_at(hw_output.in(), xo);

// Intermediate buffer at MEM level
conv2
  .store_at(hw_output, xio)
  .compute_at(hw_output, xio);

// Input stream
hw_input.in().in()    // MEM level
  .compute_at(hw_output, xio);
hw_input.in()         // GLB level
  .compute_at(hw_output.in(), xo)
  .store_in(MemoryType::GLB);
hw_input              // host level
  .compute_root()
  .accelerator_input();
```

Code 6.4: Memory hierarchy that creates the output stream, intermediate buffer at the MEM level, and an input stream.

The final element of constructing the hierarchy is creating the hardware accelerator interface. We use

```
hw_input.accelerator_input()
```

on the input memory tile buffer to specify that it is an input to the CGRA. Similarly,

```
hw_output.in().hw_accelerate(xi, xo)
              .store_in(MemoryType::GLB)
```

specifies that the output global buffer is the output of the accelerator. We observe that our accelerator interface and Halide's `in()` calls can represent an accelerator's memory hierarchy well.

Recommendation 5: Utilize Halide scheduling to construct a memory hierarchy and accelerator interface to match the target hardware accelerator.

Altogether, Code 6.4 shows how we schedule a single call to create a memory hierarchy using all of the above calls. Luckily, this memory hierarchy is the same for most accelerators using the same chip. Therefore, most applications can copy this exact set of calls to create the correct memory hierarchy.

Using Generator Parameters for Flexible Designs

Many of the applications we construct can be created with slight adjustments from previous applications. For example, ResNet has many layers, but many of these layers share the same structure. The differences are the number of input channels, output channels, and kernel size. Instead of creating many Halide applications, one for each layer, we can create a ResNet layer generator. This generator has a generator argument for each variable that changes from layer to layer. Then, we can call the application with different generator parameters for each distinct layer. This same technique can be used for image processing applications where the kernel size of a convolution could be changed. One such example in our application suite is a chain of convolution kernels where the convolution kernel size as well as number of successive convolution kernels are user-specified arguments.

Code 6.5 shows how these generator parameters are created in Halide, and then called for a simplified example layer in ResNet. This example uses the generator parameters to set the input image size (`in_img`), padding (`pad`), kernel size (`ksize`), convolution stride (`stride`), number of input channels (`n_ic`), and number of output channels (`n_oc`). The ResNet convolution layers are all very similar apart from these values. By representing these values with generator parameters, this single application can be called multiple times using different values in order to create the different convolution layer configurations. In the application, the `tilesize` is calculated using our generator parameters on line 27. The input and channel bounds are defined using Halide's `bound` scheduling primitive to specify how large the input channel (z) and output channel (w) dimensions are. The generator parameters provide a convenient way to calculate and schedule applications based on parameters. In the full ResNet application, we go even further to specify tile sizes in all dimensions, channel bandwidth unrolling, and specifying convolutions with distinct width and height values.

Recommendation 6: Construct extensible applications using generator arguments to reuse your code and create multiple DNN layers or convolutions from a single application.

Auto-scheduling Hardware Accelerators

With these observations on how to schedule image processing and machine learning applications, we have found a mechanical process in scheduling applications. New applications in our domains generally follow similar structures to other examples that we have seen. Since this guidance works with many applications, we could also look to auto-schedule these applications. An auto-scheduler and auto-tuner would take in the hardware constraints (such as number of compute tiles, capacities in the memory hierarchy, and number of levels in the memory hierarchy). Then, an input application would be tiled and unrolled to fit the target accelerator size. The application would try to minimize

```
// make clockwork HALIDE_GEN_ARGS="in_img=14  pad=1 ksize=3 stride=1 n_ic=256 n_oc=256"

// in_img determines the input image size
GeneratorParam<int> in_img{"in_img", 56};    // default: 56
// pad determines the padding to the input image size
GeneratorParam<int> pad{"pad", 1};           // default: 1
// ksize determines the output stencil size
GeneratorParam<uint8_t> ksize{"ksize", 3};   // default: 3
// Stride determines the sampling rate for the down sample
GeneratorParam<int>  stride{"stride", 1};    // default: 1
// n_ic determines the total number of input channels
GeneratorParam<int> n_ic{"n_ic", 32};        // default: 32
// n_oc determines the total number of output channels
GeneratorParam<int> n_oc{"n_oc", 32};        // default: 32

// Algorithm simplified to highlight generator parameters.
//  Reduction domain, r, for convolution in x, y, and input channel dims.
RDom r(0, ksize, 0, ksize, 0, n_ic);

// Accumulation on output channel 'w' using input channels 'z'
output(w, x, y) +=
            kernel(r.z, w, r.x, r.y) *
            input(r.z, stride*x + r.x, stride*y + r.y);

// Selected usage of GeneratorParams in the application:
// Defining tile size based on input and computation size
int tilesize = floor( (in_img + 2*pad - ksize) / stride ) + 1;
output_glb
  .tile(x, y, x_glb,y_glb, x_cgra,y_cgra, tilesize,tilesize_y, TailStrategy::RoundUp)

// Bounding input and output channels
kernel_glb.bound(w, 0, n_oc);
input_glb.bound(z, 0, n_ic);

/* Omitted: rest of algorithm and schedule */
```

Code 6.5: Usage of generator parameters in Halide application to create extensible layers. This sample set of parameters is later used in the algorithm and schedule. Generator parameters can take user-specified values during compile time.

the execution time under the constraining size of the target accelerator. This auto-scheduling problem has a smaller search space by focusing on the guidance of the above observations. Furthermore, a user would be able to refine and improve the suggested schedule after it has been auto-scheduled.

Our extension of the Halide scheduling space with declarative scheduling (as introduced in Section 3.6) reduces the boilerplate with syntactic sugar. These new scheduling primitives pose additional assumptions on the user schedules that we have seen in most of our target applications. By restricting the space of the scheduling to many good choices on the Pareto front, auto-scheduling Halide applications is an easier task. We expect declarative scheduling primitives and auto-scheduling to make scheduling to CGRAs an easier task for future users.

Table 6.3: The lines of code at different intermediate representations of the application during the compilation process. Each application is unrolled to saturate the CGRA compute elements. The "Halide" and "Clockwork" columns sum up the individual components shown on the right of each. Each step from Halide to Clockwork to CoreIR hardware increases the lines of code dramatically.

Application	Halide	Algorithm + Schedule	Clockwork	Memory + Compute	CoreIR
gaussian	124 =	52 + 72	2158 =	857 + 1301	15245
harris	165 =	88 + 77	2226 =	853 + 1373	16590
camera_pipeline	425 =	292 + 133	26517 =	748 + 25769	14438
gemm	66 =	30 + 36	958 =	409 + 549	12291
resnet	64 =	29 + 35	1139 =	460 + 679	73853
geomean	170 =	89 + 81	2248 =	589 + 1659	17696

6.5 System Evaluation

We have seen how Halide applications are compiled to CGRA implementations, and then mapped to CGRA bitstreams. This section goes through an evaluation of the compiler usability and productivity. Then, we look at an overall evaluation of the hardware implementations for our suite of image processing and machine learning applications.

Design Productivity

One of the main goals of a compiler is to bring high-level code down to lower-level abstractions. This allows a user to make higher-level algorithm descriptions and decisions, and have the compiler's job be the lowering and mapping to the low-level hardware. It is difficult to quantify the effectiveness of this process, but one metric commonly cited is the lines of code at different stages of the process.

Table 6.3 shows several applications showing the lines of code in the Halide algorithm and schedule, in the middle-end Clockwork memory and compute code, and the CGRA hardware implementation in CoreIR. We see that there are magnitude jumps in the lines of code as we go from Halide software code to the CoreIR hardware code. The geomean application has 170 lines of Halide, 13.2× more lines of Clockwork, and an additional 7.87× more lines of CoreIR hardware. Halide is meant to be a compact representation of the application using reduction domains and functional-style expressions to reduce code duplication. Clockwork has many more lines mainly due to the verboseness of describing each memory load and store in the memory file, and static single-assignment (SSA) form in the compute file. One particularly large Clockwork compute file is camera_pipeline. This is

Table 6.4: The lines of code for original Halide schedules and the equivalent using the new declarative scheduling to implement the exact same schedule. The number of lines decreases for each application in addition to being more readable.

Application	Original	Declarative
gaussian	36	25
harris	62	33
resnet	48	31

Table 6.5: Application parameters used to maximize CGRA utilization.

Application	Output Rate	Input Size	Tile Size	Tiling Iterations
gaussian	16	6000×4000	992×799	6×5
cascade	12	6000×4000	600×799	10×5
upsample	4	$6000 \times 4000 \times 3$	$600 \times 400 \times 3$	$10 \times 10 \times 1$
harris	4	$1530 \times 2554 \times 3$	$296 \times 319 \times 3$	$5 \times 8 \times 1$
unsharp	3	$1536 \times 2560 \times 3$	$378 \times 319 \times 3$	$4 \times 8 \times 1$
camera	4	$2560 \times 1920 \times 3$	$496 \times 368 \times 3$	$5 \times 5 \times 1$
gaussian pyramid	1	64×64	64×64	1×1
gemm	8	512×512	512×512	1×1
resnet – conv3_1	8	$56 \times 56 \times 64$	$28 \times 28 \times 8 \times 8$	$2 \times 2 \times 8 \times 16$
unet – conv3_i0	8	$140 \times 140 \times 128$	$28 \times 28 \times 8 \times 8$	$5 \times 5 \times 16 \times 32$
jitnet – enc2_c	16	$160 \times 90 \times 64$	$20 \times 32 \times 8 \times 16$	$8 \times 3 \times 8 \times 4$
mobilenets – pw_1	8	$112 \times 112 \times 32$	$28 \times 28 \times 4 \times 8$	$4 \times 4 \times 8 \times 8$

because the LUTs are expressed for all 4096 values, and duplicated three times to account for each unrolled color channel. Unrolled compute units increase the throughput of the hardware design, and also multiplies the lines of code in the Clockwork and CoreIR designs. Overall, we find that Halide is a very expressive form that creates large hardware designs.

By applying our new declarative scheduling, introduced in Section 3.6, we can improve the readability and conciseness of our schedules further. Table 6.4 compares a schedule using the original scheduling primitives and the new declarative schedules. Each schedule creates the exact same hardware configuration for a CGRA. All applications show a decrease in the lines of code needed to implement them. And more importantly, my subjective view is that the readability is better with the new scheduling.

Application Evaluation

With the recommendations in Section 6.4, we scheduled and compiled our suite of applications to the CGRA. Our suite of applications include common imaging processing kernels as well as a representative set of convolution layers seen in our DNN examples. We first schedule the applications

Table 6.6: Size of the hardware implementation of the designs listed in Table 6.5.

Application	# Compute Kernels	# Memories	MEM (B)	GLB (kB)
gaussian	112	7	252	96.75
cascade	120	9	204	78.03
upsample	42	6	604	117.19
harris	92	15	300	46.11
unsharp	54	10	508	78.50
camera	84	21	500	89.13
gaussian pyramid	5	6	260	8.00
gemm	40	6	260	64.00
resnet – conv3_1	28	9	1800	24.50
unet – conv3_i0	22	9	1568	76.56
jitnet – enc2_c	72	9	1280	50.00
mobilenets – pw_1	49	13	1568	98.00

considering the application requirements and hardware constraints and show the generator argument values for each schedule in Table 6.5. We unrolled the applications to ensure that the output throughout was maximized. The GLB capacity dictates the ideal tile size along with the constraints of the input image and tensor size. The output tiles fit within a single GLB tile. The total number of GLB iterations cover the input size with minimal overlap to ensure that the computation is mostly useful and distinct.

The image processing applications have equivalently sized tiles for the GLB and memory tiles. This is possible because the GLB has a larger capacity, and then the memory tiles only need to retain a small number of lines, and so the wide tiles still fit within the much smaller memory tiles. Additionally, the output tiles represent the total CGRA output tiles which is spread across all of the parallel data streams based on the output rate. The tiling iterations refer to the multi-dimensional tiling (x and y) iterations that are executed by the host to fully cover the output image.

The DNN applications in Section 6.4 are multi-dimensional, reflecting the tensors with multiple inputs and channels. The tile size represents the spatial tile size as well as input and output channels that are brought into the memory tile. The memories are banked and parallelized, which allows the tiles to be distributed across the CGRA fabric. The tiling iterations here refer to the subsequent double buffering swaps with the GLB. In the DNN cases, we can actually fit the entire layer in the GLB in many cases.

Once we have scheduled the designs in Halide, we can compile our applications to the CGRA. Table 6.6 shows our collected results on how large the applications are based on our schedules in Table 6.5. We count each of the compute kernels that are provided to Clockwork. This counts all duplicated compute kernels based on the output rate as well as simple compute kernels used for initialization. After scheduling and mapping to the CGRA, many of the unified buffers that are described can be merged or removed. Following scheduling and mapping, we find how many

Table 6.7: Performance metrics for our evaluated applications. The spatial utilization and compute occupancy (temporal utilization) show the effectiveness for the applications in Table 6.5. Other performance metrics are latency to run the application on hardware, and total compile time.

Application	% PEs	% MEMs	Compute Occupancy	Latency (cycles)	Compile Time (s)
gaussian	79%	24%	98.2%	50463	280.90
cascade	94%	36%	95.7%	41755	252.55
upsample	0%	5%	N/A	6354	50.52
harris	91%	30%	95.6%	24700	110.35
unsharp	78%	23%	96.6%	41601	69.29
camera	78%	20%	95.9%	47564	149.50
gaussian pyramid	8%	3%	25%	16976	14.01
gemm	35%	63%	97.0%	2162688	20.61
resnet – conv3_1	69%	19%	93.4%	967160	266.88
unet – conv3_i0	35%	13%	83.5%	108210328	114.07
jitnet – enc2_c	71%	19%	57.5%	1804452	389.49
mobilenets – pw_1	39%	88%	40.4%	1984808	134.16
geomean	52%	20%	74.0%	213569	105.4

memories are used. We notice that the number of memory tiles is fairly low.

Additionally, the usage of memory tiles and global buffer banks are given in the fourth and fifth columns of Table 6.6. These columns list how many bytes are used for the largest buffers in each of the applications. One can notice that using line buffers for image processing applications results in very low memory sizes, meaning we have ample capacity to implement them within our 2048 B memory tiles. This also suggests that if our application domain was just image processing, we could consider creating memory tiles with a lower capacity, since even with the largest tiles that fit in our GLB, the memory tiles still have a much larger capacity. The DNNs have slightly larger memories, but again fit in single memory tiles. The resulting GLB sizes are shown in the last column. Again we notice that our scheduling choice resulted in unified buffers stored in GLB tiles having sizes that are below the GLB capacity of 128 kB.

In addition to collecting application quantities, we can evaluate the schedule. Our application metrics are spatial utilization, compute occupancy, application latency, and compilation time. Table 6.7 shows these values for our application suite.

Spatial utilization refers to what percentage of the available CGRA tiles are used to implement the application. Ideally, we want to ensure that the hardware that we designed for the CGRA is used (otherwise we did not need to fabricate those units). We notice that the PEs have a high utilization for most image processing applications, as well as resnet and jitnet. Upsample is the serious exception, since no PEs are necessary for the nearest neighbor strategy that implements this version of upsampling. Furthermore, the layers of unet and mobilenets use fewer compute resources. Mobilenets is mostly bounded by memory and does pointwise computation rather than the full

convolution kernels of the other DNNs. The second column shows the utilization of the memory tiles in our suite of applications. Besides mobilenets, we notice low memory utilization across the board. If we expect most applications to have a large compute bottleneck, the number of memory tiles provided on the CGRA is a bit higher than necessary for most of the applications. However, as we look forward to generative AI and LLMs of the future, we likely need larger amounts of memory on the accelerator. So to anticipate future application demands, the current proportion of memory to PE tiles is perhaps appropriate.

The compute occupancy metric refers to how much time the compute units are computing useful values. This is computed by calculating how many cycles the application would take if each PE was scheduled to execute in every cycle perfectly, and then comparing the number of cycles that our scheduled applications actually take. This metric can also be thought of as temporal utilization, since it is the proportion of time that is effectively used. For our applications in Table 6.7, we notice a very high compute utilization in the image processing applications, because we use line buffered pipelines on large tiles. The DNN compute occupancy is lower in some of the DNNs because the schedules struggle to find useful work on unaligned tiles, and mobilenets has not much computation in its application. Improving the compute occupancy for layers takes a multifaceted approach that should investigate improved Halide scheduling (with better IO duplication for memory bottlenecks; tile sizing to better match vectorization requirements) and more aggressive Clockwork scheduling (smaller initiation intervals; more overlap between accumulation phases).

The last two columns of Table 6.7 show the total runtime of the application on the CGRA, and the compile time from Halide to mapped CoreIR. We notice that the image processing applications have tight and efficient streaming execution. On the other hand, the DNNs have extremely large tensors with lots of reductions, leading to much longer total execution times. These large values are important to consider when we construct cycle-accurate schedules, since counting cycles could easily overflow if we are not careful. Finally, the compile time column shows that even for these larger applications filling the CGRA, we still compile within minutes for all applications. With the provided data, we find that our Halide compiler system has constructed reasonable application schedules that have utilized the CGRA resources with modest compile times.

Chapter 7

Conclusion

In this dissertation, I have described our application compiler that takes image processing and DNN applications and compiles them to our CGRA hardware accelerator. We set out to create a robust compiler, and in doing so found that the challenging aspects were (1) defining and creating good hardware schedules in Halide, (2) lowering, mapping, and assigning execution schedules for memories, and (3) enabling hardware sharing through an integrated approach of Halide user scheduling, new Clockwork scheduling, and hardware generation. By tackling these aspects of the compiler, we were able to take our suite of applications and compile them to our hardware accelerator.

One key finding of scheduling these applications is that a user-guided schedule allows for great flexibility and extensibility. The benefits of user-provided schedules are a methodology and framework that lend themselves well to mapping to custom hardware. Rather than baking in heuristics and optimizations into the compiler, a separated Halide schedule allows for a user to make important scheduling decisions. While each compiler engineer's intentions are to create robust, future-proof optimizations, instead the reality is that we later encounter completely new applications that differ greatly from initial assumptions, and hardware implementations that have constraints that are hard to generalize across generations. When writing Halide schedules that target hardware accelerators, I found that the same loop optimizations intended for CPU targets are also applicable when targeting custom hardware. Additionally, I appreciate the thought model of this system to bridge the gap between imperative software code and hardware design language (HDL) code. From this work I have found that Halide is a good starting point for creating a system that supports both CPU targets and hardware accelerator targets.

Another aspect of this work was creating the unified buffer abstraction. Our group decided early to create efficient memories that couple the addressors and SRAM components together. Compiling to these memories was very challenging, and only after constructing an understandable interface (the unified buffer abstraction), did more consistently create correct and efficient schedules for our hardware. This interface allowed us to ensure we were able to express applications on the front-end

compiler (in HalideIR), while concurrently creating mapping optimizations on the back-end compiler (in Clockwork). We did not find much difficulty in mapping compute elements, since most compute operators in Halide mapped directly to a single compute tile. However, the next generation of hardware (Onyx) creates a more capable compute tile where a multiply and add can be achieved using one tile. As the capability of the compute tile grows, we again will have to create an abstraction and mapper, this time to efficiently map the compute kernels.

The final goal of this dissertation was to unlock optimizations that only make sense for hardware designs. We see loop tiling and parallel execution both in CPUs and hardware accelerators, and so it makes sense that Halide had these scheduling primitives. When thinking beyond the existing Halide primitives, I tried to think of what additional capabilities hardware designers use to create performant and efficient hardware. For performance, we see hardware duplication with simultaneous execution; this is achieved using `unroll` in Halide to hardware. For efficiency, hardware designers can reuse the same hardware to maximize the utilization of each component. I demonstrated this with compute sharing, but this can also be extended to co-allocating unified buffers in the same memory tiles. And furthermore, you could find some uses of the muxes in the CGRA switching network for additional efficiency. I am excited to see what other hardware design ideas can be enabled by new Halide scheduling, along with the compiler transformations to achieve this unique mapping.

While Halide is able to describe hardware schedules with mainly its original primitives (besides our additional compute share), it still is difficult for a new user to learn. However, after creating many hardware schedules, I find myself following the same sequence of steps to create hardware schedules for new applications. I laid these out in Section 6.4. This led to me creating syntactic sugar for Halide schedules that make assumptions that I regularly make about hardware schedules. These steps hint that the space of efficient hardware schedules is really not as large as one would expect. This is a good realization, since this narrower scheduling space means we can feasibly create an auto-scheduler for hardware. This is great for new users, since it can recommend multiple good schedules, and allow the user to choose and explore further scheduling based on its suggestion. I look forward to an auto-scheduler that can generate good schedules for image processing applications and DNNs by possibly searching this new restricted scheduling.

Our goal from the outset was to use the Halide language to map image processing and DNN applications to our new CGRA. To that goal, we have mapped a healthy suite of applications with many layer sizes. However, despite the success of these applications, several Halide applications still do not map to the CGRA, or see degraded performance. Some of these issues are CGRA hardware limitations, such as the constrained space of addresses that an affine address generator brings. Other issues remain in the Clockwork scheduler and mapper, where we could hypothesize how we would ideally connect and configure the CGRA to execute an application, but creating a robust set of optimization and/or mapping passes has not been attempted yet. And finally, there

are some expressions from Halide, such as iterative computation and data-dependent addresses, that can be expressed in Halide, but the Clockwork understanding of these computation patterns does not exist. All of these issues are luckily not fundamental, and we can build off our extensible application compiler to support and optimize for more features in the future. For example, we could hypothesize a full-rate implementation of histogram accumulation that ping-pongs accumulation into two different memory tiles and then reduces those two entries during the readout phase. We would first create a handcrafted example to demonstrate that it works, then create unit tests to test the functionality of the system of this histogramming compute pattern, and then finally apply this histogramming feature to an application like bilateral grid.

As I wrap up with this dissertation, I am reminded that a compiler is a complex tool that is bookended by its motivating applications on its input, and by the innovative hardware designs as its target output. As we consider future hardware accelerators, we must revise our applications and hardware as previously state-of-the-art techniques become antiquated in comparison to emerging ideas. Just as DNNs were taking over image processing applications as the default execution pattern, we now see vision transformers replacing DNNs. These generative LLM techniques are quickly becoming the most important applications to consider for future hardware acceleration. Current hardware designs are built based on the computation sizes, memory capacities, and memory bandwidth of current chips. However, with new application requirements, this balance of computation and memory must be reconsidered. Along with new sizes, another challenge for the compiler is supporting new hardware blocks by constructing new mappings. Address generator loop-nest depth and complexity was a key focus of our compiler design. Future designs, though, will need to consider data-dependent addressing, dynamic image sizes, and potential sparse accesses. Most important is creating a robust compiler system that supports these features before optimizing the compilation for the most efficient usage of new hardware components. Overall, I have learned that building a robust compiler system that maps emerging applications to a new hardware accelerator takes a good conceptual understanding of the fundamental application operations in order to effectively map them to the features on the hardware. I look forward to people creating new applications, which will challenge us to compile them to new hardware designs.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283. Savannah, GA, USA, 2016.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. Learning to optimize Halide with tree search and random programs. *ACM Transactions on Graphics (TOG)*, 38(4), July 2019.

[3] Andrew Adams, Eino-Ville Talvala, Sung Hee Park, David E. Jacobs, Boris Ajdin, Natasha Gelfand, Jennifer Dolson, Daniel Vaquero, Jongmin Baek, Marius Tico, Hendrik P. A. Lensch, Wojciech Matusik, Kari Pulli, Mark Horowitz, and Marc Levoy. The Frankencamera: An experimental platform for computational photography. In *ACM SIGGRAPH*, New York, NY, USA, 2010. Association for Computing Machinery.

[4] Edward H Adelson, Charles H Anderson, James R Bergen, Peter J Burt, and Joan M Ogden. Pyramid methods in image processing. *RCA Engineer*, 29(6):33–41, 1984.

[5] Adobe Inc. Adobe Firefly. https://www.adobe.com/sensei/generative-ai/firefly.html, 2023.

[6] Apple Inc. Deploying transformers on the Apple neural engine. https://machinelearning.apple.com/research/neural-engine-transformers, 2023.

[7] Newsha Ardalani, Clint Lestourgeon, Karthikeyan Sankaralingam, and Xiaojin Zhu. Cross-architecture performance prediction (XAPP) using CPU code to predict GPU performance. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO)*, MICRO-48, page 725–737, New York, NY, USA, 2015. Association for Computing Machinery.

[8] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala

embedded language. In *Proceedings of the Design Automation Conference (DAC)*, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. Computer and redundancy solution for the full self-driving computer. In *IEEE Hot Chips Symposium (HCS)*, pages 1–22, 2019.

[10] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[11] Jean-Yves Bouguet et al. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. *Intel Corporation*, 5(1-10):4, 2001.

[12] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[13] Peter J Burt and Edward H Adelson. A multiresolution spline with application to image mosaics. *ACM Transactions on Graphics (TOG)*, 2(4):217–236, 1983.

[14] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*, page 33–36, New York, NY, USA, 2011. Association for Computing Machinery.

[15] Alex Carsello, Kathleen Feng, Taeyoung Kong, Kalhan Koul, Qiaoyi Liu, Jackson Melchert, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zachary Myers, Brandon D'Agostino, Pranil Joshi, Stephen Richardson, Rick Bahr, Christopher Torng, Mark Horowitz, and Priyanka Raina. Amber: A 367 GOPS, 538 GOPS/W 16nm SoC with a coarse-grained reconfigurable array for flexible acceleration of dense linear algebra. In *IEEE Symposium on VLSI Technology and Circuits (VLSI)*, pages 70–71, 2022.

[16] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An automated end-to-end optimizing compiler for deep learning. *arXiv preprint arXiv:1802.04799*, 2018.

[17] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(1):127–138, 2017.

[18] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. Dadiannao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 609–622, 2014.

[19] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with optimized dataflow architecture. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, New York, NY, USA, 2018. IEEE.

[20] Derek Chiou. The Microsoft Catapult project. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124. IEEE Computer Society, 2017.

[21] Jason Cong, Hui Huang, Chiyuan Ma, Bingjun Xiao, and Peipei Zhou. A fully pipelined and dynamically composable architecture of CGRA. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 9–16, 2014.

[22] Ross Daly, Lenny Truong, and Pat Hanrahan. Invoking and linking generators from multiple hardware languages using CoreIR. In *Proceedings of the Workshop on Open-Source EDA Technology (WOSET)*, 2018.

[23] Javier Delgado, Joao Gazolla, Esteban Clua, and S Masoud Sadjadi. A case study on porting scientific applications to GPU/CUDA. *Journal of Computational Interdisciplinary Sciences*, 2(1):3–11, 2011.

[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[26] Caleb Donovick, Ross Daly, Jackson Melchert, Lenny Truong, Priyanka Raina, Pat Hanrahan, and Clark Barrett. PEak: A single source of truth for hardware design and verification. *arXiv preprint arXiv:2308.13106*, 2023.

[27] David Durst, Matthew Feldman, Dillon Huff, David Akeley, Ross Daly, Gilbert Louis Bernstein, Marco Patrignani, Kayvon Fatahalian, and Pat Hanrahan. Type-directed scheduling of streaming accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming*

Language Design and Implementation (PLDI), page 408–422, New York, NY, USA, 2020. Association for Computing Machinery.

[28] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR WORKSHOPS*, pages 109–116, 2011.

[29] Harry D. Foster. 2018 FPGA functional verification trends. In *International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 40–45, 2018.

[30] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203–1233, 2000.

[31] Mingyu Gao, Jing Pu, Xuan Yang, Mark Horowitz, and Christos Kozyrakis. TETRIS: Scalable and efficient neural network acceleration with 3d memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 751–764, New York, NY, USA, 2017. Association for Computing Machinery.

[32] Michaël Gharbi, Jiawen Chen, Jonathan T. Barron, Samuel W. Hasinoff, and Frédo Durand. Deep bilateral learning for real-time image enhancement. *ACM Transactions on Graphics (TOG)*, 36(4), July 2017.

[33] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: An ultra-low-power, energy-minimal CGRA-generation framework and architecture. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 1027–1040, 2021.

[34] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *IEEE Micro*, 32(5):38–51, 2012.

[35] Alex Graves and Jürgen Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 21. Curran Associates, Inc., 2008.

[36] Chris Harris, Mike Stephens, et al. A combined corner and edge detector. In *Alvey Vision Conference*, volume 15, pages 10–5244, 1988.

[37] Samuel W. Hasinoff, Dillon Sharlet, Ryan Geiss, Andrew Adams, Jonathan T. Barron, Florian Kainz, Jiawen Chen, and Marc Levoy. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (TOG)*, 35(6), Dec 2016.

[38] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[39] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: compiling high-level image processing code into hardware pipelines. *ACM Transactions on Graphics (TOG)*, 33(4):144–1, 2014.

[40] James Hegarty, Ross Daly, Zachary DeVito, Jonathan Ragan-Kelley, Mark Horowitz, and Pat Hanrahan. Rigel: Flexible multi-rate image processing hardware. *ACM Transactions on Graphics (TOG)*, 35(4), July 2016.

[41] John L Hennessy and David A Patterson. Pixel visual core, a personal mobile device image processing unit. In *Computer Architecture: A Quantitative Approach (6th Edition)*, pages 579–592. Elsevier, 2017.

[42] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Communications of the ACM*, 62(2):48–60, Jan 2019.

[43] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[44] Dillon Huff, Steve Dai, and Pat Hanrahan. Clockwork: Resource-efficient static scheduling for multi-rate image processing applications on FPGAs. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 186–194, New York, NY, USA, 2021. IEEE.

[45] Intel Inc. Altera OpenCL. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html, 2023.

[46] Intel Inc. Intel FPGAs. https://www.intel.com/content/www/us/en/products/programmable.html, 2023.

[47] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan

Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 1–12, New York, NY, USA, 2017. Association for Computing Machinery.

[48] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 4D-CGRA: Introducing branch dimension to spatio-temporal application mapping on CGRAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8, 2019.

[49] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA), oct 2017.

[50] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Spatial: A language and compiler for application accelerators. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, page 296–311, New York, NY, USA, 2018. Association for Computing Machinery.

[51] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. AHA: An agile approach to the design of coarse-grained reconfigurable accelerators and compilers. *ACM Trans. Embed. Comput. Syst. (TECS)*, 22(2), Jan 2023.

[52] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, May 2017.

[53] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. MAESTRO: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings. *IEEE Micro*, 40(3):20–29, 2020.

[54] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 461–475, New York, NY, USA, 2018. Association for Computing Machinery.

[55] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 242–251, New York, NY, USA, 2019. Association for Computing Machinery.

[56] Chris Leary and Todd Wang. XLA: TensorFlow, compiled. https://developers.googleblog.com/2017/03/xla-tensorflow-compiled.html, 2017.

[57] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, page 51–57, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Sean Lie. Cerebras architecture deep dive: First look inside the HW/SW co-design for deep learning : Cerebras systems. In *IEEE Hot Chips 34 Symposium (HCS)*, pages 1–34, 2022.

[59] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Unified buffer: Compiling image processing and machine learning applications to push-memory accelerators. *ACM Transactions on Architecture and Code Optimization (TACO)*, 20(2), March 2023.

[60] Bruce D Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, pages 674–679, Vancouver, Canada, August 1981.

[61] Karima Ma, Michael Gharbi, Andrew Adams, Shoaib Kamil, Tzu-Mao Li, Connelly Barnes, and Jonathan Ragan-Kelley. Searching for fast demosaicking algorithms. *ACM Transactions on Graphics (TOG)*, 41(5), May 2022.

[62] Maxeler Inc. MaxCompiler. https://www.maxeler.com/products/software/maxcompiler, 2023.

[63] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, pages 61–70, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[64] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Clark Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis. *arXiv preprint arXiv:2104.14155*, 2021.

[65] Jackson Melchert, Yuchen Mei, Kalhan Koul, Qiaoyi Liu, Mark Horowitz, and Priyanka Raina. Cascade: An application pipelining toolkit for coarse-grained reconfigurable arrays. *arXiv preprint arXiv:2211.13182*, 2022.

[66] Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina. Canal: A flexible interconnect generator for coarse-grained reconfigurable arrays. *arXiv preprint arXiv:2211.17207*, 2022.

[67] Mentor Graphics Inc. Catapult high level synthesis. https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform, 2023.

[68] David Moloney, Brendan Barry, Richard Richmond, Fergal Connor, Cormac Brick, and David Donohoe. Myriad 2: Eye of the computational vision storm. In *IEEE Hot Chips Symposium (HCS)*, pages 1–18. IEEE, 2014.

[69] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[70] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling Halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4), July 2016.

[71] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, October 2019.

[72] Angshuman Parashar, Prasanth Chatarasi, and Po-An Tsai. Hardware abstractions for targeting EDDO architectures with the polyhedral model. In *International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Virtual Event, 2021. HiPEAC.

[73] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 137–151, New York, NY, USA, 2019. Association for Computing Machinery.

[74] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. SambaNova SN10 RDU: A 7nm dataflow architecture to accelerate software 2.0. In *IEEE International Solid- State Circuits Conference (ISSCC)*, volume 65, pages 350–352, 2022.

[75] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine: A reconfigurable architecture for parallel patterns. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 389–402, New York, NY, USA, 2017. Association for Computing Machinery.

[76] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3), Aug 2017.

[77] Wajahat Qadeer, Rehan Hameed, Ofer Shacham, Preethi Venkatesan, Christos Kozyrakis, and Mark A. Horowitz. Convolution engine: Balancing efficiency and flexibility in specialized computing. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, page 24–35, New York, NY, USA, 2013. Association for Computing Machinery.

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

[79] Jason Redgrave, Albert Meixner, Nathan Goulding-Hotta, Artem Vasilyev, and Ofer Shacham. Pixel visual core: Google's fully programmable image vision and AI processor for mobile devices. In *Proceedings in IEEE Hot Chips Symposium (HCS)*, pages 1–18, 2018.

[80] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[81] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-based image processing accelerators with Hipacc: (invited paper). In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1026–1033, 2017.

[82] Alex Reinking, Gilbert Louis Bernstein, and Jonathan Ragan-Kelley. Formal semantics for the Halide language. *arXiv preprint arXiv:2210.15740*, 2022.

[83] Robin Rombach, Andreas Blattmann, and Björn Ommer. Text-guided synthesis of artistic images with retrieval-augmented diffusion models. In *Proceedings of the European Conference on Computer Vision (ECCV) Workshop on Visart*, 2022.

[84] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, pages 234–241, Cham, 2015. Springer International Publishing.

[85] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Computer Vision – ECCV 2006*, pages 430–443, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[86] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342, 2014.

[87] Jagadeesh Sankaran, Ravi P Singh, Stanley TZENG, et al. Programmable vision accelerator. https://patents.google.com/patent/US20160321074A1/en, 2016. US Patent App. 15/141,703.

[88] Yakun Sophia Shao, Jason Clemons, Rangharajan Venkatesan, Brian Zimmer, Matthew Fojtik, Nan Jiang, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Stephen G. Tell, Yanqing Zhang, William J. Dally, Joel Emer, C. Thomas Gray, Brucek Khailany, and Stephen W. Keckler. Simba: Scaling deep-learning inference with multi-chip-module-based architecture. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 14–27, New York, NY, USA, 2019. Association for Computing Machinery.

[89] Amirali Sharifian, Reza Hojabr, Navid Rahimi, Sihao Liu, Apala Guha, Tony Nowatzki, and Arrvindh Shriraman. μir -an intermediate representation for transforming and optimizing the microarchitecture of application accelerators. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture (MICRO)*, page 940–953, New York, NY, USA, 2019. Association for Computing Machinery.

[90] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[91] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations (ICLR)*, 2015.

[92] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, Geoff Lowney, Adam Herr, Christopher Hughes, Timothy Mattson, and Pradeep Dubey. T2s-tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189, 2019.

[93] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, Vijay Vasudevan, Wei Han, Jiquan Ngiam, Hang Zhao, Aleksei Timofeev, Scott Ettinger, Maxim Krivokon, Amy Gao, Aditya Joshi, Yu Zhang, Jonathon Shlens, Zhifeng Chen, and Dragomir Anguelov. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[94] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 27. Curran Associates, Inc., 2014.

[95] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[96] Elene Terry. Silicon at the heart of hololens 2. In *IEEE Hot Chips Symposium (HCS)*, pages 1–26. IEEE Computer Society, 2019.

[97] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. Ultra-elastic CGRAs for irregular loop specialization. In *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 412–425, 2021.

[98] Ethan Tseng, Yuxuan Zhang, Lars Jebe, Xuaner Zhang, Zhihao Xia, Yifei Fan, Felix Heide, and Jiawen Chen. Neural photo-finishing. *ACM Transactions on Graphics (TOG)*, 41(6), Nov 2022.

[99] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. Evaluating programmable architectures for imaging and vision applications. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[100] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30. Curran Associates, Inc., 2017.

[101] Sven Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software – ICMS 2010*, pages 299–302, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[102] Neal Wadhwa, Rahul Garg, David E. Jacobs, Bryan E. Feldman, Nori Kanazawa, Robert Carroll, Yair Movshovitz-Attias, Jonathan T. Barron, Yael Pritch, and Marc Levoy. Synthetic depth-of-field with a single-camera mobile phone. *ACM Transactions on Graphics (TOG)*, 37(4), July 2018.

[103] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: Synthesizing programmable spatial accelerators. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 268–281, 2020.

[104] Xilinx Inc. Vivado high level synthesis. https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html, 2023.

[105] Xilinx Inc. Xilinx FPGAs. https://www.xilinx.com/products/silicon-devices/fpga.html, 2023.

[106] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, Christos Kozyrakis, and Mark Horowitz. Interstellar: Using Halide's scheduling language to analyze DNN accelerators. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 369–383, New York, NY, USA, 2020. Association for Computing Machinery.

[107] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, page 1–8. IEEE Press, 2018.

[108] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a reconfigurable dataflow accelerator. In *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, pages 1041–1054, 2021.

[109] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. *AutoPilot: A Platform-Based ESL Synthesis System*, pages 99–112. Springer Netherlands, Dordrecht, 2008.

[110] Qiuling Zhu, Navjot Garg, Yun-Ta Tsai, and Kari Pulli. An energy efficient time-sharing pyramid pipeline for multi-resolution computer vision. In *IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, pages 278–281, 2013.