EVALUATING SPATIALLY PROGRAMMABLE ARCHITECTURE FOR IMAGING AND VISION APPLICATIONS

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

Artem Vasilyev
March 2019

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.

Pat Hanrahan

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.

Ofer Shacham,

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Fixed function hardware for image processing (ISP) has been used ever since digital cameras were introduced. This solution is efficient but the resulting systems are slow to adapt because creating a new chip takes a long time. While historically this wasnt a problem, the explosion of new applications from computational photography and computer vision fields has made this approach difficult. To address the need for greater flexibility, a number of specialized accelerators were created: PVC, Hexagon, Myriad, etc. These chips rely on programmable SIMD VLIW architectures for increased flexibility.

This thesis explores and evaluates an alternative approach that is more similar to traditional ISP engines - a spatially programmable architecture which is in the class of coarse-grain-reconfigurable array (CGRA) machines. We demonstrate that CGRA can be programmed as easily as a conventional CPU or GPU by using a domain specific language (DSL) for image processing like Darkroom or Hailade and leveraging an FPGA development flow based on the VPR toolset. Our evaluation framework shows that programming in space with CGRA archives a modest improvement over programming in time with SIMD: about 1.6x better energy efficiency and 1.4x better area efficiency. However the cost of programmability is still high: compared to an ASIC, CGRA has 6x worse energy and area efficiency, and this ratio would be roughly 10x if memory dominated applications were excluded. To reduce this gap requires each compute unit to do more computation making it more specialized and less flexible. How to accomplish this, while still retaining enough programmability, is the key challenge for future research.

Acknowledgments

Finishing this thesis has been my dream, my goal and my challenge for years. I wouldn't have been able to do it without help from many people. I thank my wife for her support and taking care of so many things; my parents and grandparents for encouraging me and for everything they have done for me; my friends and colleagues for keeping me focused and on track; my advisor for believing in me and mentoring me.

This work is dedicated to my son, who is my inspiration and my reason.

Research taught me many different things, some more useful than others, and a couple of principles I want to mention. Turn your passion into motivation but keep a cool mind to get things done. Above all, stay on course and don't forget that every challenge presents an opportunity.

THANK YOU!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Custom hardware for processing images (image signal processing or ISP) has been used ever since digital cameras[79] were introduced. This approach creates area and energy efficient engines for the large computation that digital cameras require. Recently this approach has started breaking down: supporting the rapid growth of imaging applications and the growing the types of computation required is becoming increasingly difficult. These hardware engine are difficult and expensive to adapt to the changing market conditions and constant innovations in the field, like new computational photography methods and introduction of CNN for image recognition. As a result there is increasing interest in creating efficient, but more programmable image processing engines. This thesis explores using a spatially programmed array to address this need. Similar to fixed function ISP implementations, programmable in space architecture relies on pipeline parallelism and static data flow for control while using flexible wires for programmability, rather than instructions found in traditional programming in time machines like CPU and GPU.

In this chapter we'll look at the imaging applications and describe their structure as well as some development process. Next, we'll provide an overview of various hardware options available for implementation of these algorithms focusing on two kind of programmable architectures: programmable in time and programmable in space. Finally, we'll conclude the introduction with the motivation for the rest of this thesis.

1.1 Imaging Applications

In this section we'll take a closer look on the application landscape. We will start from the applications run on a traditional ISP and then move to more recent applications from computational photography and computer vision domains. As we will see, all these applications have similar structures and can be viewed as a generalization of the traditional ISP structure. Finally we'll give an overview of the application development process.

1.1.1 Simple ISP

The main task of an ISP always has been to convert the sensor data into a good looking picture. Early versions used relatively simple digital signal processing techniques to archive this objective. The narrow task definition, large data volumes, and need for low power consumption lead to creation of custom hardware to implement this task. While the sophistication of the algorithms to complete this transformation has grown over time, its main function has remained the same: to correct numerous problems with image data such as:

- Estimating the missing color information at each pixel with "demosaic"
- Removing noise by filtering the data
- Correcting sensor and lens defects, like "dead" pixels, lens distortions, etc.
- Perform a number of image enhancement steps like sharpening, color correction and tone mapping to make the picture "look good" to the user.

These functions are performed in sequence and form steps of what is known as an ISP pipeline (Fig. 1.1). Each step can be viewed as a certain transformation of the incoming image and as a whole this structure gradually transforms the incoming data into a better and better looking image. We can think of each stage as gradual improvement of the data until we get to the final result. Thus each stage uses as an input the entire image read from the previous stage and the output is the entire image written to the next stage.

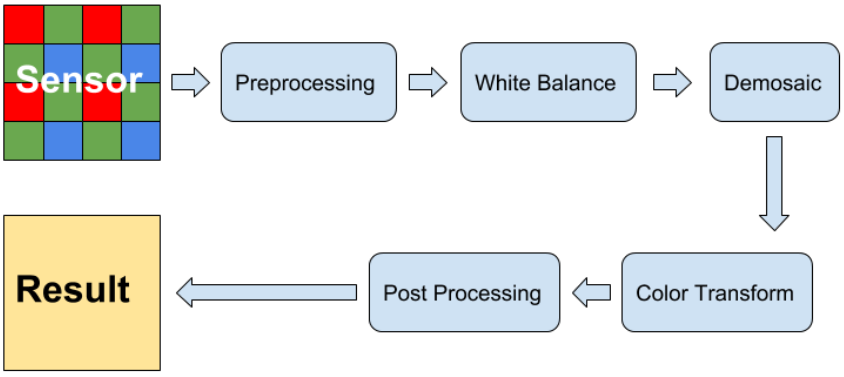

Figure 1.1: Simple ISP pipeline consist of several stages that take in a full image and produce an improved version of it (see Ramanath et al [88]).

The ISP also collects various image statistics to enable camera functions as Auto Focus, Auto Exposure and Auto White Balance. Ramanath et al[88]. and Gentile et al. [39] describes these steps in more detail and give two examples of an ISP. Modern ISPs have many more steps and a bit

more complex structure (see Fig. 1.2) than a simple pipeline. Typical implementations are quite complicated requiring 100s, even 1000s operations per pixel (See Tab. 12 in Chapter 6.2.1). As a result, the computation load for modern sized images (10Mpx) is very high especially for video applications which require 30-60fps:

$$1000\text{Op/pxl}*10\text{Mpx/frame}*30\text{ frame/s} = \mathbf{300\ GOP/s}.$$

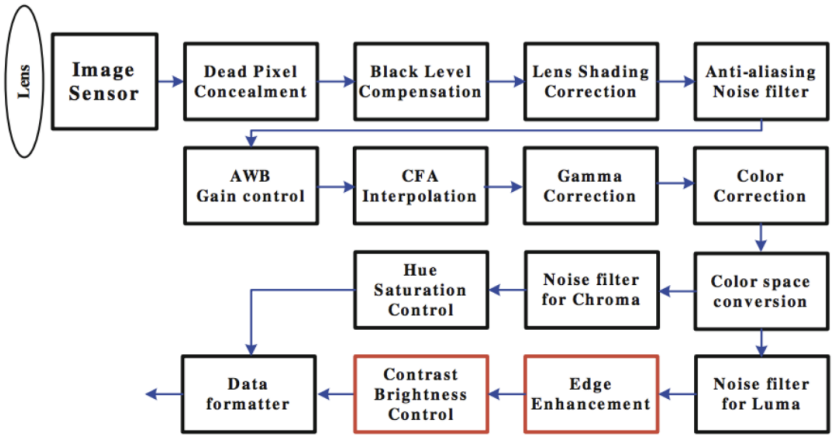

Figure 1.2: Modern, more complicated ISP has many more kernels and a more complex structure (see Park, H. S. [84]).

1.1.2 Structure of ISP application

Fortunately, the computations required to compute each pixel has a high spatial locality, requiring input data only from a small region around the pixel, called the stencil, and also has high parallelism, since all pixel outputs can be computed in parallel. Furthermore, the computations are typically done with low precision integer operations, because the input data coming from the sensor is only 10-14 bit per pixel and the output is a multi channel image with 8-16 bit per color channel per location. These characteristics enable efficient hardware implementations of these algorithms.

The ability to perform computation efficiently doesn't guarantee that application overall is implemented efficiently. If the input image to a kernel is kept in DRAM, the energy of accessing it might overwhelm the total energy of a kernel, because an arithmetic operation costs over 1000x less than the energy of reading or writing a single pixel to DRAM (see Tab. 6.2 in Section 6.2.2). Thus ISP engines are optimized to minimize DRAM access and rely on local storage and cheaper ways to capture locality and data parallelism. To capture data locality, ISP relies on the data locality of imaging kernels: for every pixel in the output, it's values depends only on a small region around corresponding location in the input image. This region is called *stencil* and the specialized local storage is called *line buffer* (see Chapter 4.6). The line buffer abstraction allows one to view an

application as a DAG of kernels which take as an input a full image produced by prior kernel and produce a temporary full image for later kernels. This is just an abstraction, because line buffer doesn't store a full image; instead it captures only a few rows at a time and exploits spatial data locality to produce a stencil need by the kernel. Line buffers also allow ISP engine to further improve efficiency by leveraging producer consumer locality between two image kernels and allow the computation to be structured as a hardware pipeline of kernels and line buffers. Kernels perform the desired data transformation and line buffers capture temporary data reuse and minimize the required energy. This leads to a pipeline of kernels and line buffers (see Fig. 1.3).

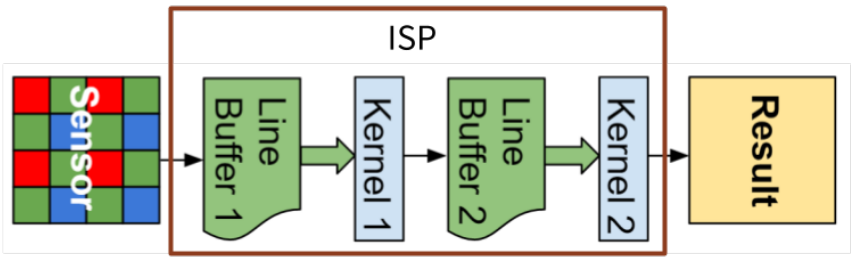

Figure 1.3: Structure of the simple ISP - pipeline of kernels separate by line buffers.

As we saw in Fig. 1.2, modern ISPs are more complex than the straight pipeline, but they still exploit spatial locality and data parallelism. In general case, an application can be represented by Directed Acyclic Graph (DAG) with two types of nodes: Kernels which perform computation and Line Buffers which provide data and capture data locality (Fig. 1.4). Kernels don't communicate with each other directly - only through a line buffer. A Line Buffer can have only one kernel as an input, but multiple kernels can be reading from the same line buffer.

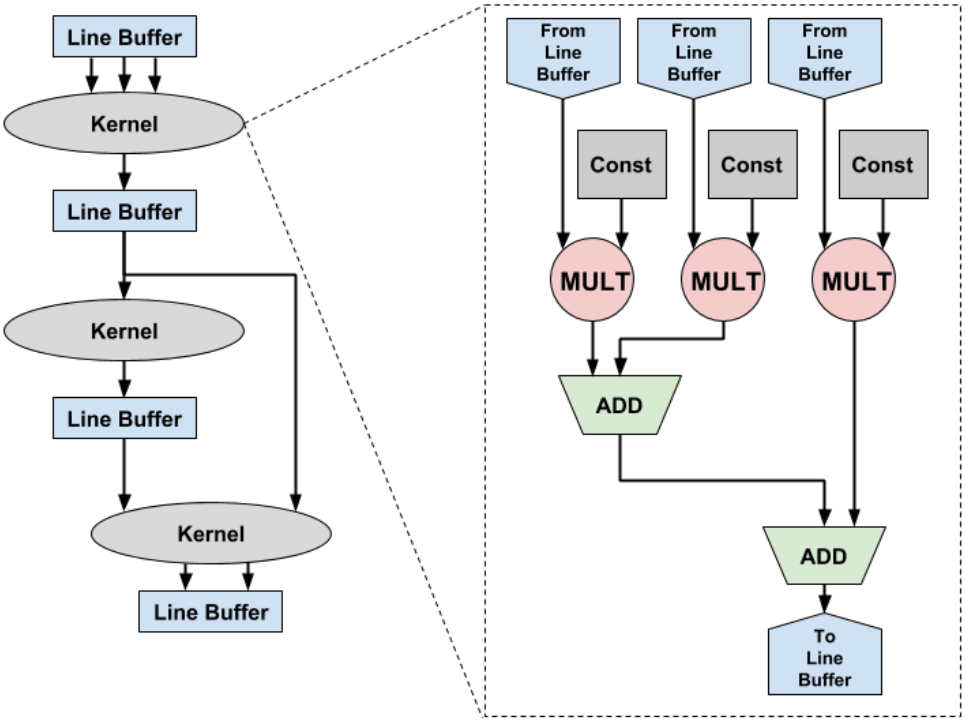

Figure 1.4: ISP is organized as a DAG of kernels and Line Buffers and each kernel is a DAG of operations.

A kernel is also a DAG and consist of three types of nodes: input-from/output-to the line buffer, operations on data (such as MULT, ADD, etc.) and constants (also known as immediate values) which can be used as inputs to some operations. Each kernel represents program that computes a pixel (or one color channel of a pixel) at the x, y location in the resulting image based on small region near x, y location in the input image. This region is often called stencil.

1.1.3 Modern imaging applications: Computational Photography and Computer Vision

Within the last decade, the landscape of imaging applications has changed with the introduction of the Computational Photography methods and wide adoption of Computer Vision techniques like CNNs for image classification and analysis. These applications usually have a more complicated structure than an ISP and we can view them as a generalization of the simple structure, so the old methods are still useful.

Computational photography (CP) or computational imaging is defined as: "digital image capture and processing techniques that use digital computation instead of optical processes."[114] It is widely

used to enhance an image or even to generate an image that would be impossible to capture with given camera. Examples of the first kind include: High Dynamic Range or HDR[29] imaging, burst photography[46], panorama stitching[16] and the second kind includes applications like: obstructions removal[124] and light field rendering[66]. CP algorithms often use more global information than a simple stencil requiring multiple passes over the same pixel data(for example when image pyramids[1] are computed); additionally the application can operate on multiple images from a burst. This means that there is less locality and we can't rely solely on internal storage. In practice this results in a DAG that has several input and output streams. In addition to a new pixel from image sensor, there are several data words that are read from DRAM on every cycle, similarly, there several data words stored to DRAM on every cycle besides the final result. Such applications are still very efficient because the compute density is at least an order of magnitude more than typically done in ISP. So the increased energy cost due to extra DRAM accesses is amortized across 1000s, even 10000s operations.

Computer vision (CV) is a field that develops algorithms for image understanding. Early work concentrated on finding "features" - interesting points in the image and creating "descriptors" based on the image content around those points, such as SIFT[70]. However most modern methods use use CNN such as AlexNet[57] or VGG[99] to automatically create models that are used for image classification and even transformation.

CNN applications have two variants: training and inference. The goal of training is to find CNN parameters that give best classification results on a training data set. These parameters are often called "weights" and the training is done using backpropagation which often involves floating point computation. Training is a long process, but it's typically done once using powerful computers and clusters of computers using previously generate and labeled images. Inference phase on the other hand is performed much more often - every time there is a new image to analyze. It applies the weights to the real images, often in a much less powerful mobile environment in order or classify them. During inference, the weights are constant and often computation is done using small integers like 8 or 16 bit. This work focuses on applications that can be performed near the sensor, similarly to ISP, so we are interested in the inference phase. Most of the kernels in CNNs' DAG are convolution layers, but unlike ISP where the image dimension remain roughly the same between kernels, CNNs reformat the data at every layer, making it smaller but tall. Generally this requires blocking[125] and a more sophisticated memory subsystem, but the rest of the application structure is the same. Since each layer of the CNN is a convolution, it's possible to use line buffers, however the resulting implementation will not be as efficient as the one using blocking.

Computer vision (CV) and image processing are becoming increasing intertwined. Images are generally processed before running CV, and many CV techniques can be helpful in traditional image acquisition pipelines. For example CNN can be used in image pipeline to perform white balance step[9], or Face Detection[112] methods as commonly used in digital cameras during autofocus and

image processing in general[103]. Besides cameras, CV and image processing are widely used in cars to implement advanced driver assistance system[80] (ADAS).

Usually imaging applications are even more complex than the ISP described earlier, but they still exploit spatial locality and data parallelism. An application can still be represented by the structure and look like a DAG of kernels and buffers (line or double buffers). The difference is that in general this DAG has several image inputs and outputs and a line buffer in some cases might perform data reshuffling to support blocking. Just like before, kernels don't communicate with each other directly - only through a buffer. A Buffer can have only one kernel as an input, but multiple kernels can be reading from the same buffer.

1.2 Image application development

Digital image processing is a well established field going back to the later 1960s[92]. The complexity of application development was always a concern because these algorithms are created by imaging engineers whose primary focus is the quality of the result, rather than performance. So initially, applications were implemented using general purpose programming languages and deployed on general purpose computers. Coding wasn't easy because the languages didn't use the right abstractions for the domain and run time was long because of high data volumes and computation complexity. As a result, in order to use these methods in the actual products, a specialized hardware implementations has to be created which lead to dedicated ISP ASICs in the first digital cameras.

Application developer needed a quick and simple way to test their ideas with minimal effort. Over time, domain specific libraries (e.g. OpenCV[15]) and even specialized computing environment like Matlab and Simulink were created. Matlab was primary created to simplify matrix operations and overtime became popular in scientific communities for developments of various signal processing applications, including ISP. The main problem with this approach is that the application would have to be developed again when engineers needed to convert the Matlab model into a high performance production grade implementation.

In order to make the code faster, developers had to take advantage of the special feature in the target hardware like special instructions and SIMD processor extensions, i.e. NEON[7], SEE[31], AVX[89] which over time were added to the CPUs to improve performance. Later, more specialized architectures were created that better suited high throughput applications like imaging. A primary example of such architecture is gpGPU. Even though these hardware enhancements did improve the run time on general purpose systems, custom ASIC implementations were often still needed due to energy and power savings. To take advantage of these hardware capabilities developers often this required the use of explicit assembly intrinsics or use specialized languages like CUDA/OpenCL and embrace the new heterogeneous nature of compute. As a result the implementation of high performance imaging application quickly grew in complexity, becoming more and more difficult to

write. Because of this difficulty, initial algorithm implementation and production code were different programs and required testing and debug to make sure they produce the same result.

The increased popularity of imaging applications has led to creation of new languages optimized for this class of applications. Such languages as referred to as Domain Specific Language or DSL and are designed to make programming easy and efficient by taking advantage of certain characteristics common in a particular domain. An examples of such DLS for imaging are Halide[87] and Darkroom[48]. Halide is a functional language that separates algorithm and scheduling. Thus it's easy to create a working version which can then be optimized for a certain architecture by providing a scheduling detectives, like *vectorize, reorder, parallelize*. There is no need to use assembly or duplication work - the algorithm portion remains as is. Support for different platforms, including new specialized accelerators like PVC[96] or Hexagon DSP[26], is also simple with various backends supported by the compiler, all that is needed from the developer is to create a schedule for the new architecture. Darkroom operates similarly, but it's a more restrictive and it can be seen as a subset of Halide, targeting stencil operations and line buffered pipelines. This restriction makes the language even easier to use.

1.3 Hardware architectures for imaging

Image processing pipelines have been implemented on many different types of computing platforms. These span a wide range of approaches from ASIC ISPs with limited programmability, to CPU, and GPU engines which can run applications from Halide and other imaging DSLs. While CPU and GPU are the most flexible computational engines, these engines are much less efficient (in energy/op or performance/area) than the more specialized ISP architectures. This efficiency loss can be critical, especially in power constrained form factors like cellphones. As a result, there has been a recent effort to create specialized compute engines for image processing to address the need of high performance, energy efficiency and programmability.

1.3.1 Taxonomy of existing image processors

To understand the landscape of various implementations, we will group them based on the number of processing elements in the compute engine, and the number of instructions executed by the application to compute each output pixel(see Fig. 1.5). Here, by processing element we mean some hardware unit that is visible to and controlled by the user. And an instruction means a command or configuration that is sent to any of the hardware units in order to compute the result. Using these definitions, a completely fixed function ASIC ISP which needs no configuration data is mapped at the (0,1) coordinate: it requires zero instructions and contains a single hardware unit. Most ASIC ISPs require some configuration information to run, moving it to the (1,1) coordinate. If the hardware is not monolithic and consist of 20 individual IP cores that can be used to form an application, and

the connection between these IPs and the IP function needs to be configured once before processing as many pixels and images as we want, would occupy the coordinate (1, 20).

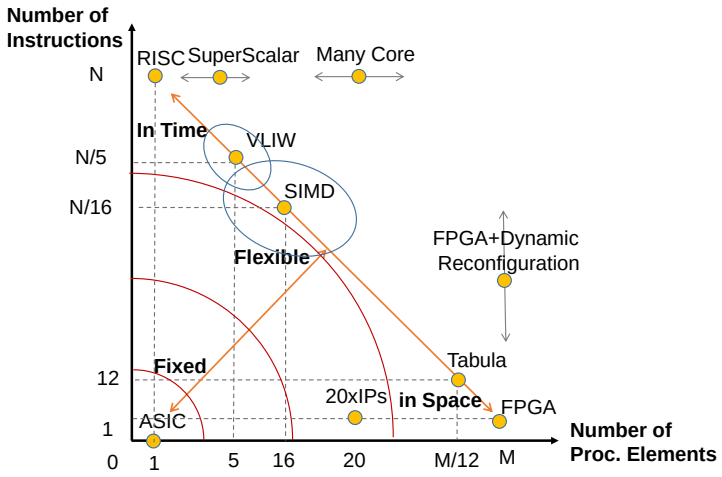

Figure 1.5: Taxonomy of image processing engines. Flexible solutions represent a spectrum between programmable in space' and in time' solutions.

This rough diagram shows that these platforms can be classified into three main categories depending on how they are programmed and how flexible they are: **ASIC**, or non-programmable (fixed function) architectures; **programmable in time** machines, including multicore, vector, SIMD and VLIW architectures; and **programmable in space machines** like FPGAs.

ASIC fixed-function solutions achieve the best efficiency and performance, and they are widely used to execute image processing and computer vision algorithms. These solutions implement the image processing pipeline directly in hardware. They have deep pipelines with line buffers in between compute nodes, where each node performs a specific operation in the application's data flow graph. Well known examples of ASIC solutions include image signal processor chips used in digital cameras, like Canon's DIGIC[6] and Fujitsu's Milbeaut[56]. These later appeared as fixed function blocks in mobile SoCs such as TI's OMAP[27][118], and NVIDIA's Tegra[32]. A similar approach was used in automotive computer vision systems for advanced driver assistance (ADAS), e.g. EyeQ from Mobileye[102] or some models of Analog Devices' Blackfin[18] line. More recently, such chips have been used for low power implementation of modern algorithms like HDR[90]. Efficiency, however, comes at the price of flexibility. Most ASICs typically only allow a few configurable parameters like filter coefficients. As a result, adding new features like face detection might require a new hardware implementation. As imaging and vision have become more dynamic, the delay required to create a new chip is no longer acceptable, which creates a strong need for an efficient programmable ISP.

Programmable-in-time approaches use a computing engine with instructions to execute the

image application. This includes multicore CPUs like Adapteva[82] and Kalray MPPA-256[28]. Most of the solutions in this class use some kind of SIMD or vector approach to amortize the energy and area overhead caused by the instruction fetch over many pixel-parallel data operations. VLIW solutions try to increase utilization of ALUs by allowing multiple sub-instructions to simultaneously use CPU resources after compile time reordering[26]. Some architectures such as Cadence Vision P5[30], Movidius Myriad[51], Ceva XM4[98] and CogniVue[14] contain both SIMD and VLIW engines and support longer instructions to increase throughput, efficiency, and utilization. GPUs[81] add threads to SIMD by increasing the register file so as to allow multiple threads to be resident, which enables fast thread switching. This allows GPUs to have very high function unit utilization, since they context switch to hide memory and other long latency operations. As we will see, this decision increases energy cost, and is not energy efficient for this class of applications.

Spatially programmable architectures reconfigure their connections to support programmability and include traditional FPGAs such as Altera Stratix V[4], Xilinx Virtex 7[77], as well as time-multiplexed FPGA like Tabula[42]. Another example of spatial architecture is coarse grain reconfigurable array or CGRA. Past and present CGRA chips targeted for image processing include RaPID[34], PiCoGA[19], RICA[55], MorphoSys[101] and ADRES[111]. Recently, deep learning applications for imaging have inspired CGRA accelerators for convolutional neural networks including NeuFlow[36], Neuro CGRA[52], Origami[20] and Diannao[22].

1.3.2 Programmable in Time

The task for any programmable machine is to be able to compute a DAG of kernels, where each kernel can be an arbitrary DAG of operations. Programmable in time is the most common approach to map these types of graphs, and is used in such familiar architecture as CPU and GPU. Here the operations are serialized, and instructions are used to select the needed operation each cycle. The values on the graph edges are stored in memory (register file or SRAM), so the size of memory is set by the largest set of live values needed for the chosen serialization. Arbitrary graphs can be executing by choosing instructions with the right operation that fetch from the correct memory locations. The DAG of kernels is then created by interleaving execution of the different kernels so that only a line buffer worth of data needs to be stored in the local processor caches for every kernel.

This architectures often use SIMD approach in order to improve efficiency[43]. Another popular optimization technique is VLIW - it improves performance by reducing the total number of instructions required. One of the most common VLIW applications is to overlap compute and data transfers by having “arithmetic” and “memory” slots for simultaneously controlling ALUs and Load/Store unit. Both SIMD and VLIW are used in general purpose machine, chips specialized for imaging additionally are optimized for low precision integer math, and often they implement a custom memory system optimized for working with 2D stencil data instead of standard caches, an example of such system is NVidia Patch Memory[25].

While programming in time is the most natural way to create a flexible engine, historically it has not come close to the energy efficiency of ASIC solutions. The reason might be because programming in time approach is fundamentally different from the way fixed function ISPs are built:

- we use multiple identical compute resources (ALUs) performing the same operation specified by the instruction, which changes every cycles instead of multiple specialized resources performing different operations which don't change over time
- we use storage (register files) instead of direct connections to pass data between elements
- we rely on data parallelism (SIMD) instead of pipeline parallelism and building data flow graphs

In order to improve, a different approach to flexibility and programmable hardware is needed.

1.3.3 Programmable in Space

Unlike a programming in time model where programmability is achieved by executing different operations (instructions) over time on a single processing element, programming in space employs many processing elements, each executing different operation which doesn't change over a long period of execution. Here we build an application graph from various hardware elements as a large pipeline and programmability is achieved by connecting different elements together. It's similar to creating an electronic circuit, where each component represents different operations (instructions) and the circuit as a whole is a graph of these operations, representing the entire application. We can also think of it as a custom vector engine, where the connection of the different hardware elements creates a very complex vector instruction (computes the entire kernel) and the data streaming through it forms the vector, thus each pixel is process by one complex vector instruction.

Another commonly used term for this class of computing engines is a Reconfigurable Architecture (RA)[104]. There are two examples of such architectures: FPGAs and Coarse Grain Reconfigurable Architecture (CGRA). FPGAs were initially used as "glue" logic or as an IC emulation platform[54]. It performs bit level operations that can implement any logic and typically has a capacity of many millions of gates. Overtime FPGAs were adopted for computation[35] and their architecture was changed to include specialized elements (like memories, standard IOs such as PCIe and DDR, dsp blocks, and even full ARM cores) to better support this use model. At least one study has shown that the price of FPGAs flexibility is high compared to ASIC: 35x more area, 4.6x longer delay, 14x more dynamic power[58], and as such they are not very efficient for computation. The same study showed that the efficiency improves if FPGA uses "hard macro" blocks like BRAM (memory) and DSP (multipliers). CGRA is essentially an FPGA consisting largely of these hard macro blocks and thus performing more complex operations than single bit logic. CRGA uses a lot of the same ideas and techniques as FPGA but offers better efficiency as a computation platform.

When compared to traditional programming in time models, RAs removed the overhead of instruction fetch+decode and access to large register file and add the overhead of configurable muxes and wires. So if the average length of a wire is small, RA will be more efficient. A number of workload classes such as signal processing, media codecs, pattern matching and sorting[83] are known to map well to spatial architectures. A detailed overview of reconfigurable architectures and their applications can be found in Tessier's survey[104]. We use this approach to create and evaluate our programmable ISP.

1.4 Thesis Overview

Decades after digital imaging became widespread it continues to be a dynamic field with new products released each year that use advanced imaging as a distinctive feature. Many new algorithms are created to improve the quality and understanding of images. However there is no commonly accepted flexible implementation of such algorithms - an analog of GPU for graphics doesn't exist in image processing. This thesis leverages recent work in creating the imaging DSLs Halide and Darkroom, and extensions that enable these compilers to map applications to ASIC hardware to explore and evaluate a different approach for flexible image processor using a programing in space approach. We will use the abstract hardware model of a fixed function ISP and FPGAs' approach to programmability to create a compute substrate optimized for the imaging applications. Our architecture belongs to CGRA class of machines. A high-level view of this architecture and a programming flow for it based on existing imaging DSL compilers and FPGA tools (VPR) are presented in Chapter 2. The main component of programmability is flexible wires, their implementation is presented in Chapter 3. This architecture uses customized and programmable processing elements, described in Chapter 5.5, and memory subsystem, described in Chapter 4.6. Chapter 6 presents a methodology to compare the programming and space and programming in time approaches and presents these results.

Chapter 2

A Spatially Programmed ISP

Both spatially programmed hardware, and ISP design are well researched areas and we heavily leverage these previous research results. This chapter starts by reviewing the micro-architecture used in many ISPs, using the notation from J. S. Brunhaver[17] which we will follow. Next it reviews the spatial approach to programmability used by both FPGAs and CGRAs. We'll describe how memory, logic and wiring is created based on application requirements, as well as the basic application mapping flow used for FPGAs. Our CGRA uses many of these same design approaches, which allows us to leverage many tools created to explore FPGA designs. Section 2.4 then reviews a process of generating a CGRA hardware implementation of an imaging application written in domain specific languages. This review points out a number of challenges that need to be addressed to create this flow. We will address these challenges in the subsequent chapters.

2.1 ISP micro-architecture

A micro architecture for hardware implementation of ISP was described by J. S. Brunhaver[17] (Fig. 2.1). It reflects the overall structure of ISP (see Section 1.1.2) as a DAG of kernels and Line Buffers. To this basic microarchitecture it adds a few implementation details. One is related to the implementation of the line buffer abstraction and how a stencil window can be efficiently generated in hardware, by combining a normal SRAM with a shift-register array. The shift-register captures the strong locality of convolution operations. Another is to generalize fixed function hardware by storing some constant values in "tap registers" to allow these parameters to be tuned over time. An example would be filter weights that depend on the lens used in the actual system. Even though the algorithm treats these values as constants, there should be a way to change them after the hardware is deployed.

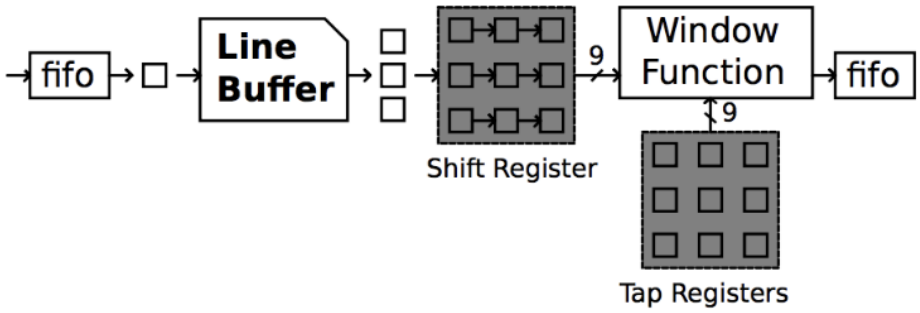

Figure 2.1: Micro-architecture of fixed function ISP consists of line buffer for storage, shift registers for preparing the stencil, window function for performing computation and tap registers for allowing configuration and small adjustments (taken from Brunhaver, J.[17]).

The ISP is built as a long hardware pipeline using a static data flow architecture[109]. It takes advantage of the fact that "window function" (which we refer to as kernels) can be statically scheduled and enables the "shift register" optimization.

We extend this work to create a programmable in space rather than fixed function implementation based on the same micro-architecture. Because our solution will flexible, we don't need to explicitly separate "tap register" - these special values can be embedded in the DAG representing the kernel. Thus we can slightly simplify the abstract machine model which now looks like Fig 2.2.

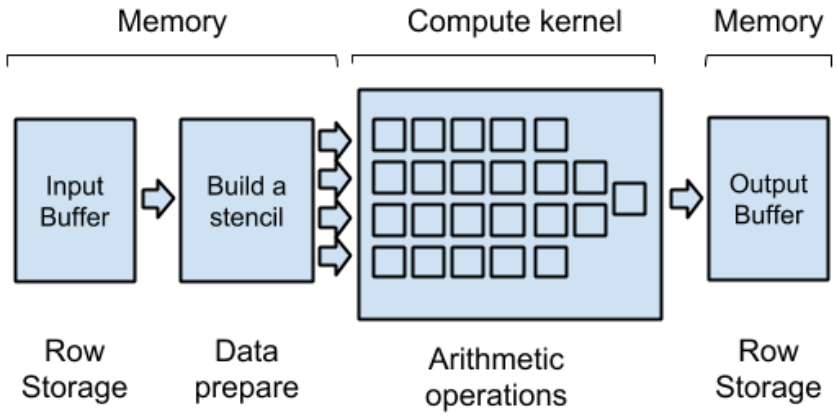

Figure 2.2: Abstract machine model for CGRA is a simplified ISP micro-architecture and includes line buffer for storage, data preparation for building a stencil and number of pixel data operations that implement a window function.

The memory subsystem implements a line buffer abstraction and consists of a "Row Storage" and "Data Preparation" components. Row Storage keeps the data locally on chip and enable data

reuse across different rows of the image. Data Preparation captures the data reuse between adjacent stencil invocations during raster order computations (across columns). It mostly consists of the same shift registers as in fixed function hardware(see Fig 2.1). In our case, data preparation might also do some amount of stencil manipulation to handle border conditions, by filling in the missing data. Compute kernels still consist of a collection of the arithmetic and logic operations that implement required computation. Our hardware has the same control mechanism using static data flow as before and the implementation still looks like the long pipeline.

To make the chip programmable, we need to find a way to configure the Memory, which is discussed in Chapter 4.6, and specify the correct operation for all the nodes that constitute kernel (see Chapter 5.5). But most importantly, there must be a way to establish connections between various elements and realize logical order of operations and producer-consumer relationships described by user program. For this task we borrow many concepts from FPGA design, which are reviewed in the next section. Routing for the CGRA is further described in Chapter 3.

2.2 FPGA approach to programmability

The most common organization of FPGAs is island style (Fig. 2.3). It is used in many commercial chips such as Altera Stratix V[4], Xilinx Virtex 7[77] and many others. One of the key advantages of this organization is high regularity which allows replication and scaling to larger chips as well as easier optimization of hardware and simplified timing analysis.

Island style FPGA consists of tiles placed on two dimensional grid. Tiles are connected by wire segments which form a 2D mesh interconnect network. Segments are laid on W tracks, which is one of the key architecture parameters. Inside each tile there are three types of elements: logic blocks, connection blocks (CB) and switch blocks (SB).

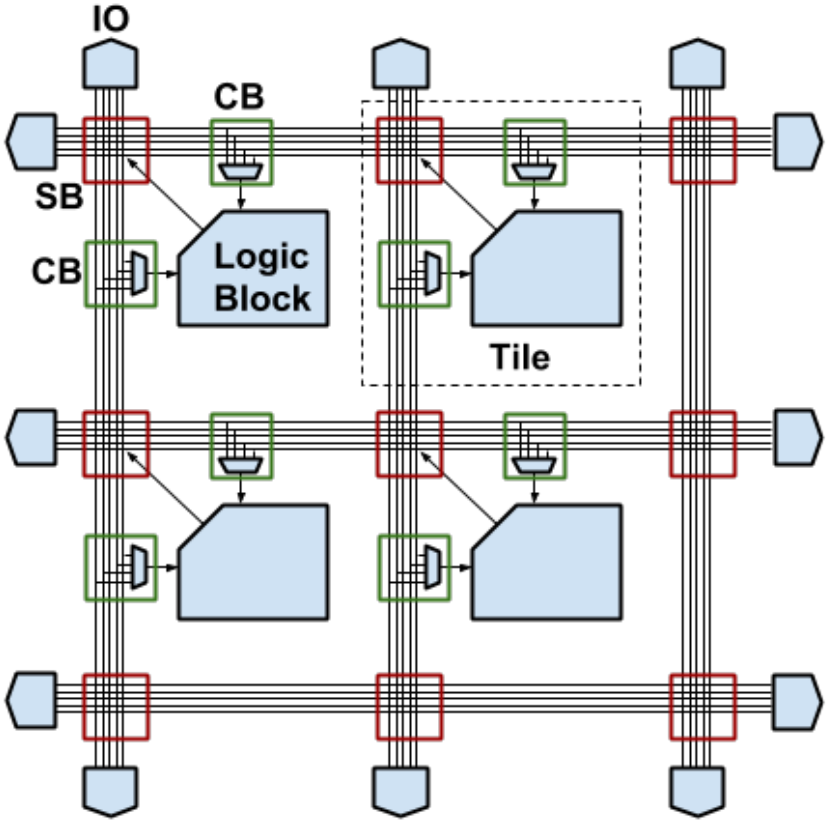

Figure 2.3: Island style organization of FPGAs consisting of tiles placed on 2D connected together by wires using mesh topology.

The logic block implements some function on the data, for example adding two values together. The data comes on wire segments which are connected to logic block by a CB. Each CB handles one input port of the logic box and connects it to one of the wire tracks. A switch block facilitates connection between horizontal and vertical wire segments as well as the output of the logic block. Note that traditionally a SB only deals with wire segments and connections between logic block output and interconnect is facilitated by an *output connection box*. Our definition of SB functionality merges traditional SB and output CB - such structure is sometimes referred to as *routing driver block*[59]. We chose this SB definition because it reflects the design choice of physical implementation for the interconnect - *single-driver routing architecture* - which we use in our design. This tile structure separates computation (Logic Block) from data transfer (CB and SB) and each tile implements both functions at the same time.

A connection between two logic blocks is constructed out of multiple *wire segments* going in different directions, which allows the connection to make turns. There is no restriction on how many segments can be used, which provides a convenient abstraction that **any two logic blocks**

can be connected, no matter how far apart they are. Of course long connection will have large propagation delays and dissipate more energy, so there is an incentive to keep them short. In addition to higher cost, we can't have too many long wires because that would require more routing tracks than available on the chip (W).

Modern FPGA have 3 kinds of logic blocks, depending on the functions they perform: memory, DSP and LUTs (Look Up Tables). Typically they are organized in columns: all the tiles in a column perform the same kind of logic. The majority of tiles are LUTs - this is the most general purpose kind of logic block, and can be used to implement arithmetic functions found in the kernels. Memory and DSP tile implement dedicated functions: bulk storage and multiplication, respectively. They typically are built for a certain granularity, but can be broken into a few smaller configurations, or combined into a larger one. For example, memory could be a dual port 36kb RAM which can be viewed as two independent single port 18kb blocks and multiple RAM can be combined to create any capacity in the 18kb increments.

2.2.1 Logic block : LUTs

Traditionally, FPGA were used as a "glue" logic and had to be able to implement any combination of gate and flip flops that form the user's digital circuit. This had a big influence on a design of the logic block in an FPGA. The main element was chosen to be an K input Look Up Table (LUT) because it can implement any function of K boolean inputs and the output of a LUT can optionally go into a flip flop which is used a storage/delay element or to improve the critical path timing by pipelining. Usually, N LUTs are organized into *cluster* (see Fig 2.4) in order to improve important chip area and delay/performance[2]. K and N are the major design parameters, and were shown to have optimal ranges[2]: 4-6 for K and 3 to 10 for N.

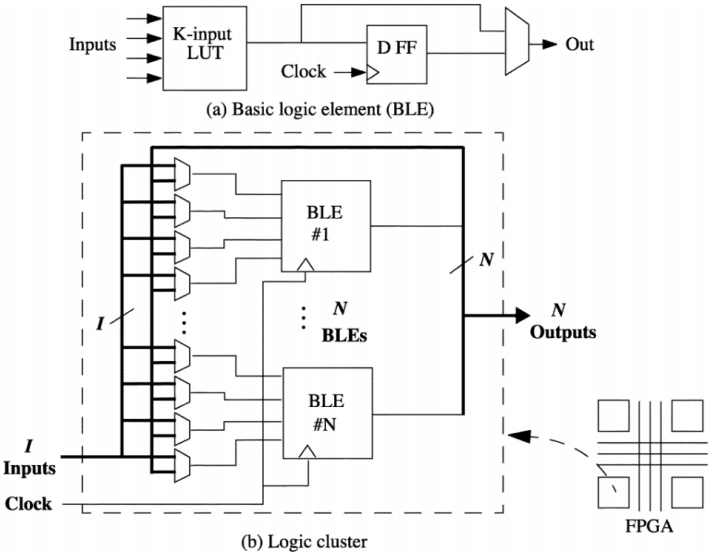

Figure 2.4: FPGA basic logic element and cluster. LUTs in the cluster can be cascaded using multiplexers on the inputs (taken form Ahmed, E[2]).

The advantage of clusters have been show in FPGAs[10] and speed/area tradeoff studied by Marquardt et al.[73]. As a result, many commercial FPGAs use this approach and add hierarchy to the design of their logic blocks. For example, Stratix fpga[68] uses *logic array block* (LAB) costing of 10 4-input look-up tables (LUT) with some local routing within LAB. In Xilinx 7 series chips[121] the basic building block is a 6-input 2-output LUT, four such LUTs are grouped into a *slice* and two slices constitute a *configurable logic block*(CLB).

Modern commercial FPGA starting with Stratix II and Virtex 5 use 6 input LUTs[67] with 2 outputs which can be split into two LUTs with smaller number of inputs, as long as some inputs are shared for certain configurations. For example we can have two 5 inputs LUTs that share 4 inputs or two 4 input LUTs that share 2. Often the basic structure also includes some additional logic for betters support of certain functions: in case of Virtex 5 it is XOR that help with carry chains[120].

Such LUT based structures are very flexible and can be used to implement any logic, however this flexibility is not free and the area cost can be pretty high. For example, LUTs can be used for computation, but implementing a simple 16 bit ripple adder, would require at least sixteen 3 input LUTs with 2 outputs which takes as much area as a full 16 bit integer arithmetic logic unit (ALU) covering a set of basic operations math (see Tab. 5.1), similar to the one used by CPUs (see Fig. 2.5).

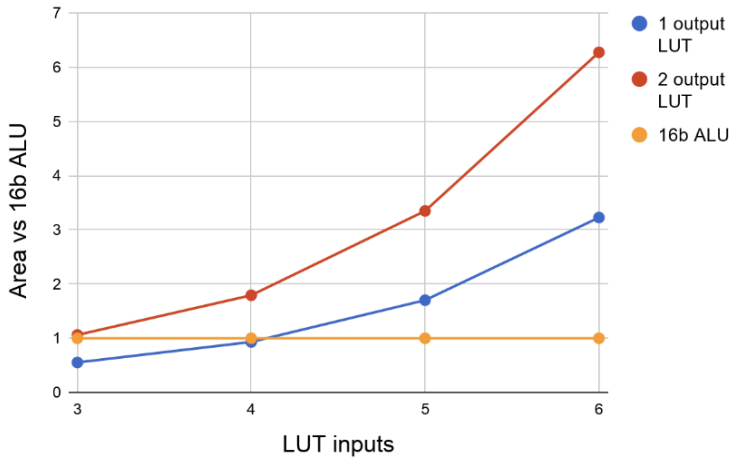

Figure 2.5: Area of 16 LUTs with 4 or 6 input configuration used by FPGA is larger than simple 16 bit integer ALU

To alleviate this inefficiency, FPGA include a number of specialized DSP[47] blocks that implement some common arithmetic operations like add and multiply. For flexibility reasons dedicated DSPs usually support operations in several bit width, for instance: 9, 18 and 36 bits[3] which adds a certain amount of control overhead. The number of DSP blocks varies based on a model within FPGA family and it is typically much smaller than the number of LUT clusters. This leaves an obvious optimization opportunity for applications dominated by computation, like image processing, to replace the fine grain LUTs and limited functionality DSPs with a block better suited for wide range of coarser gain arithmetic and logic operations, similar to ALU in programming in time architectures (See Chapter 5.5). The resulting structure is called a Coarse Grain Reconfigurable Architecture[45] or CGRA and it belongs to a broader class of Reconfigurable Computing (or RC).

2.2.2 Prior work on CGRA and reconfigurable computing

A CGRA can be viewed as a variant of FPGA, specialized for data processing. Many CGRAs were created for image processing and media applications (see Section 1.3.1) and over the years their scope was expanded to support other high-performance computing applications. Examples of these more general engines include PipeRench[40], Dyser[41], and triggered instruction architectures[83]. Recently, deep learning applications for imaging have inspired CGRA accelerators for convolutional neural networks including NeuFlow[36], Neuro CGRA[52].

Today reconfigurable computing is a very rich field[104] and multiple techniques have been proposed over the years to expand the number of cases that can be covered. Beside specialization, the most noticeable ones are time multiplexing[106] (or multi context) and run time reconfiguration[37],[65] (or RTR). Both these techniques are aimed at making the spatially programmable chip *virtually*

larger that it actually is by reusing the same hardware across different times and thus enabling larger, more complicated applications.

With RTR, the application graph is divided into two or more sections that can be viewed as a separate, smaller programs that fit into available resources. The data is processed by the first program and the output is stored in the memory, than the chip (or part of the chip) is reconfigured with the second program which is used to process output of the first stage and so on until we get the final result. Usually, each program operates on batches of data to amortize the high energy and performance cost reconfiguration. Using this approach the energy overhead comes from the memory access to save and restore data during chip reconfiguration. Typical FPGA use model assumes that configuration is not changed very often, thus it takes relatively long time to reconfigure, depending on the chip size it can take 100s of milliseconds.

Time multiplexed FPGA were commercialized by Tabula[42] and they operate similarly, but on a much finer scale: they have special resources that allow them to configure multiple times per clock cycle which virtually increased the amount of resources proportionally to how many times reconfiguration is done. Because the reconfiguration is so often, there is no need to batch the data to get the same throughput as in RTR case, but the energy cost of reconfiguration is not amortized anymore. Typically, there is a limited number of *contexts* - times reconfiguration can be performed each clock cycle and switching each context is much cheaper that reconfiguring the chip in RTR case. Each context can be run at a certain maximum frequency, and the frequency of the entire program is just the frequency of each context divided by the number of contexts.

2.3 Application mapping to spatial architectures with VTR

Spatial architectures use an application mapping process which is very similar to the ASIC development. It can be viewed as a pipeline which consists of the five major steps: synthesis, technology mapping, placement, routing and timing analysis (see Fig. 2.6). All the FPGA vendors have tools flows that implement all these steps but only support chips from that company, for example Vivado[38] from Xilinx or Quartus II[86] Altera/Intel. In this thesis we will be using the VTR[72] toolset to implement these steps for our chip. VTR is one of the most popular open source tools developed in academia for FPGA research. It is very flexible and supports a wide range of island style FPGA architectures with row based organization described by *comprehensive architecture file*. The architecture file allows the description of custom logic blocks, thus enabling us to use it with our CGRA.

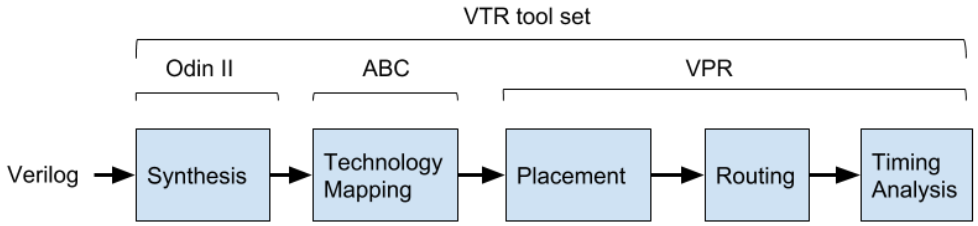

Figure 2.6: FPGA application mapping process with VTR toolset

The first step of the application mapping is **synthesis**. It takes a circuit description in a hardware description language like Verilog, parses and elaborates it, producing a graph of hardware primitives supported by target platform. This graph is often called a netlist. As a part of creating the netlist, the tool has to express all elements of the design in terms of a limited set of primitives, for instance this means that large memories are expressed in terms of 36kb pieces and multiplications are split in 36 bit ones.

Next, the netlist goes through **technology mapping** step. Here the hardware primitives are optimized and mapped into various blocks and resources that exist on the target chip. The most common example of this is "LUT packing" - converting logic into appropriately sized LUTs and identifying flip flops. The result is a netlist of elements that exist in the target architecture and that correspond to pieces of in logic block from island style model. Blocks are not yet assigned a location on the chip yet and all the connections are abstract - there are no SBs, CBs or wire tracks in this netlist.

To convert the mapped netlist into an actual implementation, we need to perform two more steps:

- placements, which finds a location for each element in the input netlist
- routing, which establishes required connections between elements using flexible routing resources: wire tracks, SBs and CBs

Placement and Routing is a well studied problem for FPGAs and many approaches[105] have been create for it. In the VTR flow it is performed by the VPR[11] tool suite.

The goal of **placement** is to assign each nodes in the netlist the coordinate of the tile that will implement this block and to guarantee that this assignment is valid. Sometimes a single hardware blocks can implement one complicated function or several simpler ones (for example DSP unit), in this case placement step would also be responsible for finding optimal mapping - the one that minimizes the total number of hardware blocks required. Such optimization can be done at the mapping stage as well, however doing it during placement has an advantage of being able to balance the node density of the various parts of the chip. Mapping doesn't have access to this density information and can only minimize the total number of blocks in the output.

Routing finds a valid a path for each connection between nodes by allocating wire tracks, SBs and CBs. The paths are then optimized. There are many different optimization goals a designer is interested in improving. For instance to minimize critical path delay and to minimize the number of routing channels required. Routing is NP-complete problem[110], but over the years there have been many algorithms and methods developed to solve it including: Maze router[61], PathFinder[76], BoxRouter[24]. These approaches span from simulated annealing[94] developed in the eighties to more modern techniques like ILP[126]. Due to complexity of the problem, tools often don't guarantee finding an optimal solution and instead just produce a result that satisfies certain requirement, like maximum delay on the critical path.

2.4 Mapping imaging applications from DSL to hardware

As we previously discussed (see Section 1.2), the complexity of application development is a major concern for the users. Nowadays, many applications are written in dedicated DSLs: using a hardware description language (HDL) like Verilog is too low level. Unfortunately, this low level description is required by all the hardware development tool set to create an ASIC or FPGA implementation. This problem can be solved by taking advantage of the structure of imaging applications (see Section 1.1.2) and converting DSL code into an intermediate representation (we refer to it as a Abstract IR) which represents the program as kernels and line buffers and then turning it to a hardware description language(HDL).

This approach was first developed by J. Brunhaver as the Darkroom to hardware[48] using custom intermediate format called DPDA[17] and a set of generators that converted it to Verilog. Standard commercial tools could then turn this Verilog into ASIC or FPGA. Later, J. Pu[85] used a similar method for Halide and compiled it to a structured C++ implementation that described kernels and line buffers and used Vivado HLS[123] synthesis to turn it into Verilog and create an FPGA implementation. Overall this process consists of three steps (see Fig. 2.7): DSL Compilation, HDL Generation, and hardware Implementation.

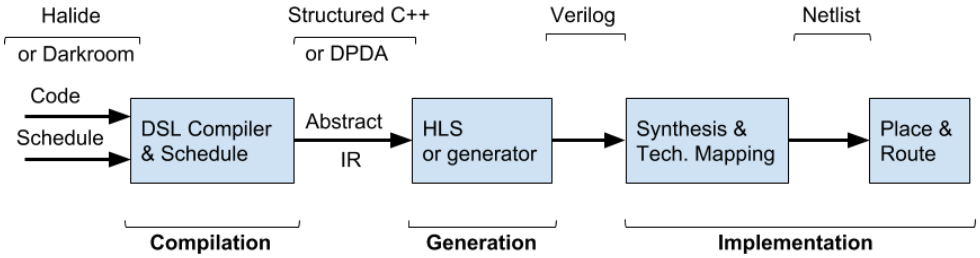

Figure 2.7: Mapping imaging DSL to hardware implementation

We can apply the same process overall process to our CGRA - we just need to use VTR/VPR during implementation step instead of vendor specific tools. To do that we first need to define how our chip implements memory structures (see Chapter 4.6) and individual operation of the kernels (see Chapter 5.5), then create an architecture description file for VTR and finally provide a way to convert the Abstract IR into a Verilog that can be synthesized by VTR to the netlist of our GCRA primitives. We implement this last step with a set of custom scripts that generate Verilog from DPDA and structured C++, this Verilog consists of explicit invocations of various architecture primitives, so netlist generation in VTR becomes just a trivial 1-to-1 conversion and all the actual technology mapping is done in our scripts which are aware how to map Abstract IR to the blocks in CGRA.

The Abstract IR for our CGRA is the same as DPDA. Based on how various elements are implemented on our chip, we group them into 3 categories: connections, memory and compute (Tab. 2.1). Connections are handle by the routing resources (Chapter 3), memory - by the memory subsystem (Chapter 4.6) and compute by processing elements (Chapter 5.5). The next chapters will describe these categories individually.

	Name	**Input**	**Output**	**Notes**
Connections	Wire	any data type	same type	Connection between elements
	Input port	N/A	any data type	Connection to the outside world
	Output port	any data type	N/A	
	Constant/Tap	N/A	any data type	Doesn't change during execute, can be changed before the launch
Memory	Line Buffer	Scalar, Stencil	Stencil, X-Y-C position	Input/output data sizes can be any size, entire stencil output is ready at the same time
	Move	any data type	any data type	Type cast. The dimensions have to match
Compute	Basic arithmetic	2 scalars	scalar	+,-,*, /, %, >, <, etc
	Special functions	Scalar, [const]	scalar	abs(), ReLu(), etc.
	Select	3 scalar	scalar	Res = c ? a : b
	Reduction	Stencil	salar	sum(), min(), max(), argmin(), argmax()

Table 2.1: Abstract IR elements for CGRA are the same as DPDA.

Chapter 3

Routing

A key attribute that makes FPGAs programmable is their routing architecture and ability to establish connections between any blocks on the chip. For CGRA we use the same approach as FPGAs: island style architecture with Switch Block(SB), Connection Block (CB) and heterogeneous wire segments all of which are reviewed in more details in this chapter. We optimize routing by reducing the number of connections by embedding constants in the compute blocks and by using two techniques previously proposed for FPGA: bus based routing and pipelined wires. None of these enhancements is supported by the standard VTR distribution, so we had to add these steps to the flow.

This chapter builds on the discussion of FPGA routing from Chapter 2, and introduces CGRA specific architectural optimizations. After that we present the final version of the application mapping process and review how the application changes as it goes through these mapping stages.

3.1 Routing resources

Flexible routing is what makes FPGAs programmable and it's also one of the biggest parts of the layout, taking up to 80% of area[12]. Over the years, FPGA researchers explored various methods to optimize routing resource and reduce their overhead. In our CGRA we rely on these past results and utilize the same approach to flexible routing. Our design is built using the same island style (see Section 2.2) organization which uses wire tracks, connection blocks(CB), switch block(SB) to route signals between logic blocks. This section reviews the routing resources of the FPGAs which our CGRA borrows.

3.1.1 Wire segments

Since the number of possible point-to-point combinations is extremely large, having dedicated wires for every possible combinations is impossible and the only option is to use a number of small

wires, typically called *wire segments*, and programmable connections to form a longer wire. Due to repetitive nature of island style organization, the length of each wire segments is usually expressed in units tied to the physical dimensions of a tile, thus a 1-long segment is long enough to connect to the next tile, a 2-long segment - to the tile after the next and so forth.

In the simplest case, all wire segments would have the length of 1 for the greatest flexibility in routing. However, modern FPGA use *heterogeneous segments* - a mix of wire segments with different lengths[12] because it results in smaller area of routing resources and better wire performance. The savings result from the fewer number of programmable segment-to-segment connections required in case of the longer wires and from the fact that typical applications have a noticeably greater than 1 average length of connections between elements.

Typically, the segments are chosen by optimizing the implementation results of some benchmark application[12]. Alternatively, Rent's rule which models general wire length distributions can be used to select the distribution of segments lengths[60]. This rule models the relationship between the amount of logic - G and the number of required pins - P and says that $P = K * G^B$, where K is a scaling constant and B is known as Rent's parameter. The value of B depends on the circuit and was found to be between 0.5 and 0.75[8]. The lower values of B generally indicate the high degree of locality in a design. Rent's rule can be applied to find the wire length distribution in VLSI chips[33] and to show that the fraction of segments with length L is: $f_L = L^{2B-3}$.

When using heterogeneous wires segments, the length of the longest wire segment is limited by wire resistance. The resistance of VLSI wires means that excessively long wires require *repeaters* along the way to prevent the degradation of the signal due to resistance as it propagates through the wire. The area cost of a repeater is comparable to the cost of programmable segment-to-segment connection: the former is just a buffer and the later is a multiplex followed by a buffer. For this reason it doesn't make sense to consider segment lengths that require the use of a repeater. The only exception to this rule is special purpose "global" signals like clock or reset, but they are not routed on general purpose resources and instead use specially designed connections. For CGRAs, where the tile size is larger than FPGAs, this means that the max segment length is a small integer, usually less than 4.

3.1.2 Connection block

Wire segments are connected to the inputs of a logic block by the connection blocks (CB). In island style organization there is one CB per input of a logic block. Each CB is usually associated with one side of the tile and only connects to wires running in that wiring track. This decision makes tile layout and physical design of the chip easier.

The major design parameter of a CB is the fraction of the wire channels it connects to, $F_{c_{in}}$. The value of this parameter is a trade off between the area of CB and routability of the chip[91]. Typically FPGAs have a very large number of wire tracks and not all of them can be selected in

a CB (thus $F_{c_{in}}$ <1.0). This is often called depopulation[13]. Overall, one can think of a CB as a simple K:1 multiplexer with $K = F_{c_{in}}$ * *number_of_tracks*.

3.1.3 Switch Block

The connections between wire segments is done by Switch Block which act as a programmable routing switch. As described by Lewis et al[68], there are only 5 electrically possible implementations for these switches but direct drive muxes and single source drivers for wire segments result in a smaller area and shorter connection delay. A wire segment is used to connect two logic blocks in FPGA or CGRA, but there still could be two connections options, depending on the direction of the signal: which block is the source and which one is the sink. To cover these cases we can use bidirectional drivers or double the number of wires and assign each wire to a single direction (see Fig. 3.1). Lemieux et al[63] showed that despite increasing the number of wiring channels the use of directional drivers in FPGA yields about 25% better area due to transistor savings in buffers, reduction in delay and capacitance on the wires "due to reduced switch loading and physical wire length shrinkage". Moreover, in combination with single driver, it archives both better area and delay (see [63] and [68]). For our CGRA architecture we will also use *directional single driver* configuration of wire segments.

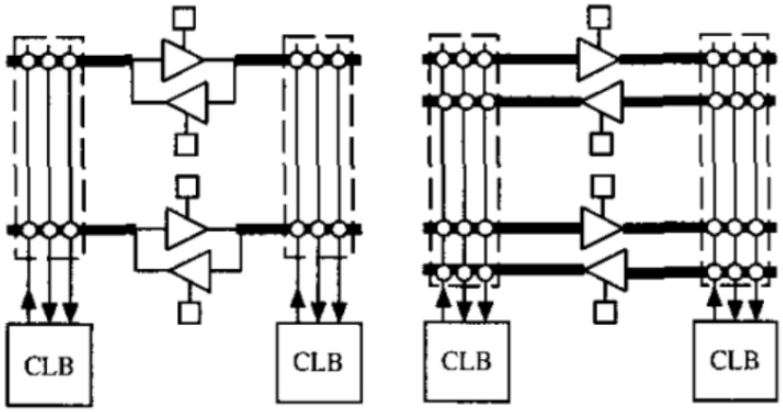

Figure 3.1: Bidirectional and directional wire segment implementation (taken from Lemieux et al[63])

The wire segments are connected together in the SB, so its design and organization greatly affects routability in the chip and the total of tracks required to implement a given circuit. The main parameters of the SB are: switch block flexibility (F_s) and connection pattern. Flexibility is defined as the number of possible connections between a wire segment and all other wire segments and typically $F_s = 3$. This flexibility means each output wire track can only connect to a single

track from each of the other sides. The connection pattern defines exactly which other segments a segment can be connected to, which is important because some patterns add diversity for the router. A few examples of the patterns used in SB are: Disjoint[119] , Universal[21] and the Wilton[117] (see Fig. 3.2).

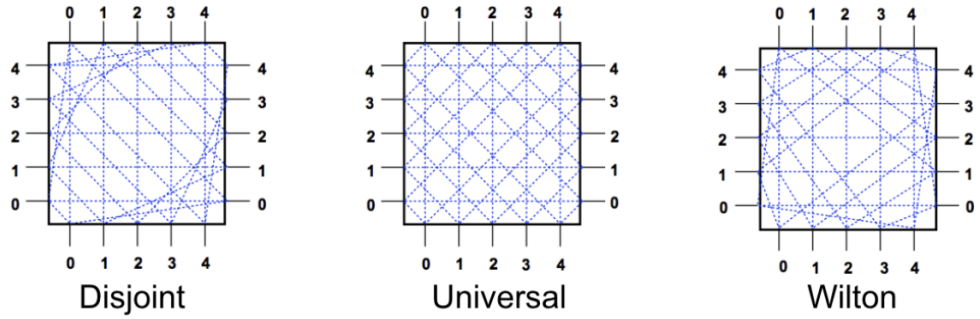

Figure 3.2: Disjoint, Universal and Wilton SB organization (taken from Wilton, S.[117])

The Wilton pattern generally achieves better routability than disjoint, because it allows connections to use wire segments of different length. Other organizations such as Imran[75] and Shifty[69] are possible, if midpoint connections between segments are allowed (with multiple possible drivers of the segment and tristate buffers). Different SB patterns are needed only with heterogeneous wire segments, because these patterns define a set of lengths which can be combined. In Disjoin it's not possible to switch to a different track, so all the segments will have the same length. In Universal there could only be 2 different lengths (for example 4 and 0) while with Wilton all segment lengths can be used, for example track #4 in top left corner on Fig. 3.2 can go to track 1 or 3, which than co go to track 2 or 0. More details about switch block design can be found in Lemieux et al[64].

In our CGRA the SB is merged with the output connection block (this structure is also known as the *routing block*). We set $F_s = 3$, which is the minimum value of this parameter that allows a connection to change direction. Each outgoing wire segment can be be connected to one of the segments that input the tile from the opposite side and just continue the signal in same direction. It also can be different from the two perpendicular sides allowing the signal to turn (see Fig. 3.3). Since one wire segments from each direction is possible as signal source, the connection pattern is important when those segments are not identical. For better routability we use the Wilton pattern in our design. Each SB is a just a collection of 4:1 multiplexers followed by the buffer that drives the wire segment. Each multiplexer selects between 3 tracks, chosen according to the Wilton pattern and an output from the Logic Block.

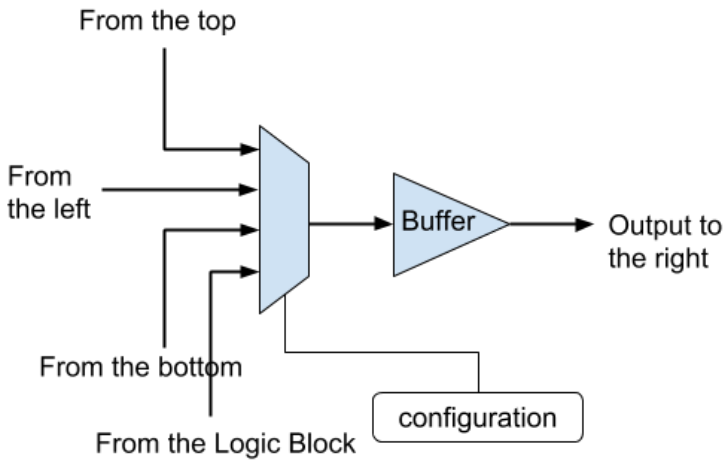

Figure 3.3: SB consists of 4:1 multiplexer allowing an output to be connect to the logic block or one of inputs from different directions. This enables turns in the direction of a signal and composing multiple segments to form a longer connection.

3.2 CGRA routing optimizations

In our CRGA we use several less common optimization: embedding constants, bus based routing and pipelined wires. These were previously proposed for FPGA, but they work even better for CGRA due to its coarse grain structure. This section describes each of these techniques and how they are implemented in our design.

3.2.1 Embedding constants

We can reduce the number of signals to be routed by examining the data which is passed on them. In particular, we are looking for data that never changes, such as constants and immediate values in the program (see Tab 3.1). We notice that overall the ratio of constants is significant: 1-9%. For image processing pipelines it's on the higher end of the ranger: 9% for ISP and 5% for FCam. The details about these applications will be described later (see Section 6.2.1).

Application	16 bit Bus	1 bit Wire	Constants
Stereo	0.94	0.03	0.03
Canny	0.07	0.87	0.05
Harris	0.87	0.06	0.07
ISP	0.82	0.09	0.09
FCam	0.74	0.21	0.05
FAST	0.11	0.88	0.01

Table 3.1: Statistics of connections used in application by type: constant values vs variables 16 bit or 1 bit data.

Instead of routing signals that never change, we can take advantage of the static pipeline computation model for our chip and store constants locally in the tile that needs them. Since the logic block in each tile is doing the same operation every cycle, this saves us communication energy and reduces the demand for routing resources. Notice that, in fixed function implementations (ASICs) the constant values are also not routed and embedded in computation blocks (which lead to additional area savings). In Programmable in Time machines constants don't produce any savings in hardware since the operation and all its operands change every clock cycle, the constants have to be stored in the register file like any other value.

3.2.2 Bus based routing

In FPGAs every 1 bit wire is routed individually and the total number of signals is equal to the total number of bit in all the connections. This gives the ultimate routing flexibility and is the best suited for connecting individual logic circuits. However, in the case of imaging applications for our CGRA, most of the blocks operate on a multi bit data (16 bit in our case) which allows one to amortize routing control across all bits in the multibit data block. This approach also guarantees that the propagation delay is the same on all the bits - something that could be difficult to maintain in the routing toll if bits were individually routed.

Bus based routing was first suggested for FPGAs[23] and later work[127] estimated that despite 20-30% increase in the number of tracks, the routing area is reduced by 14% due to "more sparse bus-based connections" and "sharing of configuration memory among each set of bus-based connections". The same study recommended using 4 bit granularity and dedicating 40-50% of the tracks to buses because of a substantial amount of non-bus control signals. But for imaging applications in our CGRA, the busses are wider (16 bits vs 4 bits) and typical applications (FCam, ISP) have more than 70% of all connections (irrespective how wide) in the program are 16 bit busses (see Tab. 3.2). There are few examples dominated by 1 bit connections (like Canny), but they tend to be smaller and simpler, thus requiring fewer tracks (see Tab. 3.2). Since the number of tracks required is set by

the worst case applications which are dominated by wide connections, we expect to see the benefit if the majority of the interconnect resources are dedicated to buses.

Harris	**Canny**	**Fast**	**Stereo**	**FCam**	**ISP**
86	86	80	124	160	120

Table 3.2: Minimal number of tracks per application reported by VPR.

The reason why bus based routing works and leads to smaller area is because it allows amortization of control circuits in SB and CB. This results in smaller area for the multiplexers inside SB and CB and smaller amount of state to hold the configuration. In case of SB and 16 bit busses, we replace sixteen 1 bit multiplexers with one 16 bit multiplexer which used only 2 bits of state instead of 16*2 bit=32 bits. But the biggest benefit is in CB design where the number of inputs is proportional to the number of tracks. Without bus based routing each 16 bit PE input requires sixteen W:1 1 bit multipliers with $log_2(W)$ bits of state. Clearly using bus routing saves a large amount of logic by leveraging the fact that there is never a need for a CB responsible for bit i of the 16 bit bus to have as inputs wire tracks that carry other bits of any other bus.

FPGAs which use 100s of tracks must deal with this issue by using CB depopulation and connecting each input only to a subset of tracks(use $F_{c_{in}} < 1.0$). Depopulation reduces CB area but it also makes signal routing harder, because now the router not only must establish the connection path, but also guarantee that the end point is at one of the tracks that are used in the CB. Often this leads to increase in the total track count which in turn increase the CB area negating some initial savings.

We evaluated CB depopulation on our benchmarks using VPR. For this experiment we created a version of CGRA architecture with the same fine gain routing as FPGAs which allowed us to use VPR without any modifications. For each benchmark application VPR estimated the minimal number of wire tracks required to route and the area of routing resources. We compared the configuration without depopulation ($F_{c_{in}} = 1.0$) with the one where each CB connects only to 15% of wire tracks nearby ($F_{c_{in}} = 0.15$), the same ratio is used by canonical FPGA architecture that comes with VPR and is in line with commercial chips. With depopulation the minimum number of tracks for the worst case has increase by 50% as a result the total routing area savings due to sparse connections was quite small - only about 6%.

With the bus based routing with N 16 bit bus tracks, a CB uses a single N:1 16 bit multiplexer. The area for this multiplexer is smaller than a sixteen N:1 1 bit multiplexer that would be needed for each input wire if we did not use buses. Including registers to store the configuration, we measured that the area of CB was reduced by **24x** compared to fully connected CB without bus grouping. Since now the multiplexing occurs of the 16 bit signals, not only have we amortized the control circuit, but have also eliminated all the cross bit redundancy, by guaranteeing that bit 0 of the CB output

can only connect to bit 0 in all of the input buses. The benefit of this technique is that the resulting multiplexers in the CB are relatively small and we can maintain full connectivity($F_{c_{in}} = 1.0$) for the bus connections in the CB without paying too much area. Full connectivity makes the routing task easier and avoids the increase in total number of tracks which we saw with depopulation.

The downside of bus based routing is that it's not efficient for 1 bit signals. To solve this problem, we add a second routing network which uses the same parameters as the first one except it uses 1 bit connections - just like FPGA. However, the total number of tracks is much smaller than in FPGAs: about 20 vs several 100s. This means that all the signals are separated into two groups: buses and 1 bit-signals. These groups interact only inside logic blocks and are completely independent in the routing resources.

As previously mentioned, VPR currently doesn't support bus based routing and we used 1bit wires for all connections. Because of this, the result of routing will only be valid if we use two independent routing networks(16 bit and 1 bit) with the same parameters (like number of tracks), which leads to inefficiency in some cases. If an application is dominated by only one kinds of connections, we'll have routing resources that we can't use. For example, in FCam only about 20% of signals are 1 bit (see Tab. 3.1), as a result, 1 bit routing resources are underutilized (see Fig. 3.4). If the number of tracks is dictated by the 16 bit buses, this inefficiency is small compared to fine grained routing case even with depopulation. In fact, since both groups of routing resources are scaled to cover all the application, the inefficiency is bounded by the ratio of the tracks in 1 bit group over the tracks in buses: $W_{1bit}/W_{bus} = N*1/(N*16) = 6.3\%$, assuming the same parameters of both groups. Of course, if the worst case application in the benchmark sets was dominated by 1 bit signals, rather than 16 bit buses, the overhead would be horrible. But this case breaks one of the main design assumptions of our CGRA that the applications mostly consist of 16 bit computations.

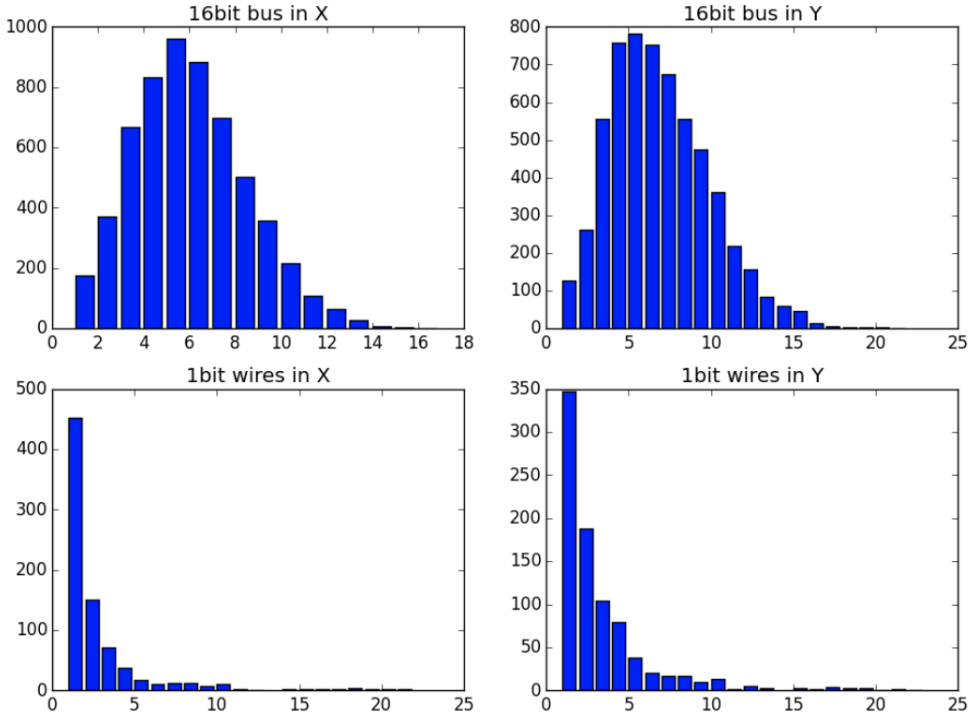

Figure 3.4: Histogram of routing utilization of buses and 1 bit wires in case of FCam.

3.2.3 Pipelined wires

When an imaging application is represented in Abstract IR form (see Section 2.4), there is no notion of *time delay*, only causality between nodes in the DAG. This makes the compiler's job easier because it doesn't need to know the physical characteristics of the hardware implementation. During hardware generation, this graph becomes a synchronous circuit and the standard rules of timing analysis apply. The performance of the implementation is set by the critical path delay - $T_{critical}$ which is a path delay between two synchronous elements in the circuit (like registers or memories).

Since programming model doesn't constrain the number of operations between these memory elements, the generated circuit might have multiple 16 bit arithmetic operations on the critical path which would cause poor timing. To fix the problem, the design has to be pipelined. For the ASIC flow this can be done by adding several register stages at the end of each kernel and applying register retiming[62] which is well supported by commercial tools. In this process these register are distributed along the critical path to minimize the maximum delay between two registers, after that a certain number of registers is added to other paths through the graph to make sure that all the inputs to any node arrive in the same cycle.

In case of the FPGAs, the situation is worse. First, register retiming is not usually supported by open sources tools which forces users to do it manually. Second, registers are usually located in the logic block, which means that adding pipelining might increase the total number of tiles required. And finally, routing resources may add a significant amount of delay which users don't see in their logical design and therefore they can't pipeline it even if they wanted to. We can handle wire delay, and avoid increasing the number of function tiles required by adding pipeline registers to the interconnect resources. This approach was previously proposed to improve traditional FPGAs[107] and it's currently implemented in Stratix 10 chips with Hyperflex[49] technology.

In our CGRA we add pipeline capabilities to the routing by modifying the SB design (see Fig. 3.3) to include an optional register after the multiplexer (see Fig. 3.5). This register is added at the SB on a wire track only if it is the beginning of the wire segment, so the total number of registers in SB is typically less that *4*W*. With heterogeneous segments, only a portion of W tracks originate at a tile - the rest are "pass through" and don't have a driver at that location. These long segments that span several tiles, have one optional register only at the SB which drives that segment. Each pipeline register is optional - it can be enabled or bypassed by the tools without the need to change routing results.

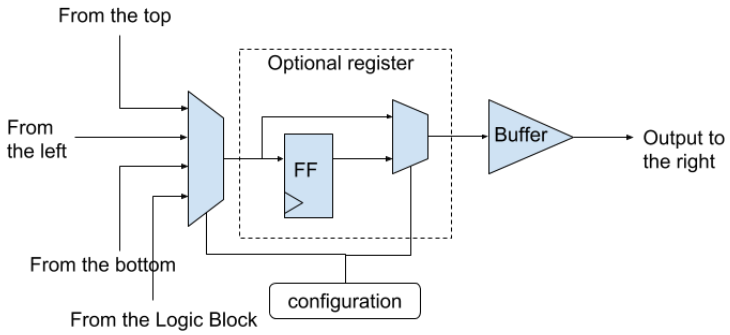

Figure 3.5: Modified SB design with the optional pipeline register.

Our wire pipelining approach is similar to the one previously explored for traditional FPGA which has shown a 25% increase of the clock frequency at the 10% area cost[100]. The authors found that only 12-25% of the SB need a registers. Unlike them, we assume that the registers are added to *every* SB after every multiplexer. This leads to a larger number of unused registers in case of small and simple applications (like Harris), but it reduces the chances that we would not have enough registers on any given path to balance the data arrival delay in order to guarantee causality. It's possible to reduce the number of registers, but it might require a change to the routing tools such that certain connections can be made *longer* than they would normally be to give opportunity for enabling more register.

3.3 Final application mapping flow

Unfortunately, none of the three CGRA specific optimizations is supported by the standard version of VTR so we have to modify the flow from Chapter 2.3 (see Fig. 2.6). The final flow (see Fig. 3.6) still has the same three stages and goes through compilation, generation and implementation, but the implementation stage is now more involved. Overall, our application development relies on existing open source tools: DSL compilers to convert the code into our Abstract IR and VPR to perform place and route. Our contribution is a set of scripts to make everything work as one flow and implement some missing features and CGRA specific optimizations. Next, we'll describe how it all fits together.

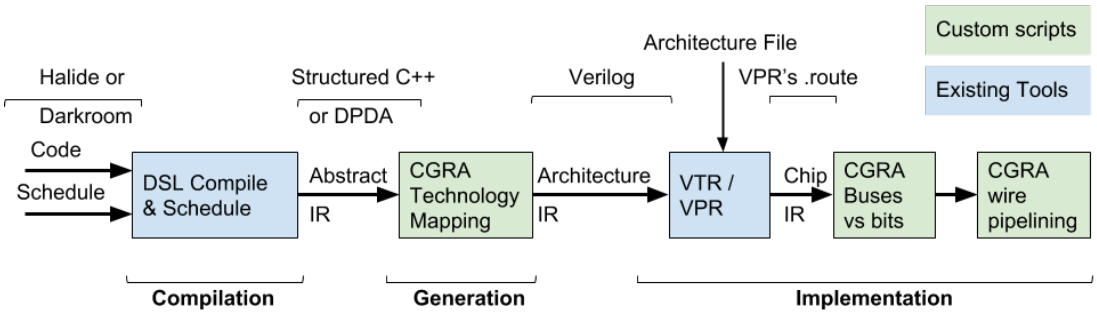

Figure 3.6: Final application mapping flow for CGRA is using DSL compiler and VTR/VPR FPGA tool set along with custom scripts that implement CGRA specific features.

The first step is DSL compilation. It converts the source code into DAG of kernels and Line Buffers (see Chapter 1.1.2) which is described using DPDA or Structured C++ according to our Abstract IR(see Chapter 2.4). With the Halide flow we use structured C++ because it's the standard output of the compiler and the resulting code has the same structure.

Next is the generation phase where we use a custom script to do CGRA technology mapping. This script translates the Abstract IR description of application into a graph of primitives supported by the hardware(Fig. 3.7). Thus this script must be aware of the target implementation and how various nodes in the IR are implemented on the CGRA. In particular, it should know how to implement a Line Buffer for any output stencil size and how all arithmetic operations are converted in to hardware blocks. This script also implements our "embedding constants" feature and removes all fixed values from the graph (it assumes that these values will later be written to appropriate registers which can be found based on unique names of all the hardware blocks). The output of this process is exactly the same graph of hardware primitives representing the application as the output of "technology mapping" step in VTR tool set, but we use Verilog to describe it instead of netlist format. Verilog allows us to simulate and verify the design and we still can run VPR by using VTR flow. For correct simulation our technology mapping script has an option to preserve the bus widths

and the constants.

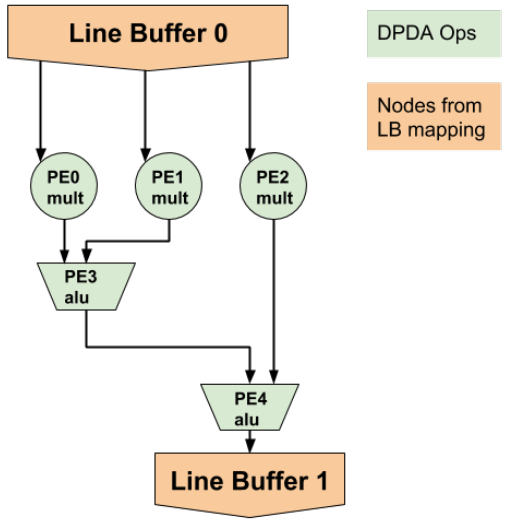

Figure 3.7: Application graph is mapped to CGRA primitives.

The next step is to perform place and route with VPR. Unfortunately, the current version of VPR doesn't yet support bus-based routing. To solve this limitation, our script used 1bit wires for *all* the connections and appended the connection names with the original width. The result of P&R process is the same graph of hardware primitives representing the application, but it now has the routing information for all connections, represented by locations of the SBs along each signal, and placements coordinates of all the nodes (Fig. 3.8).

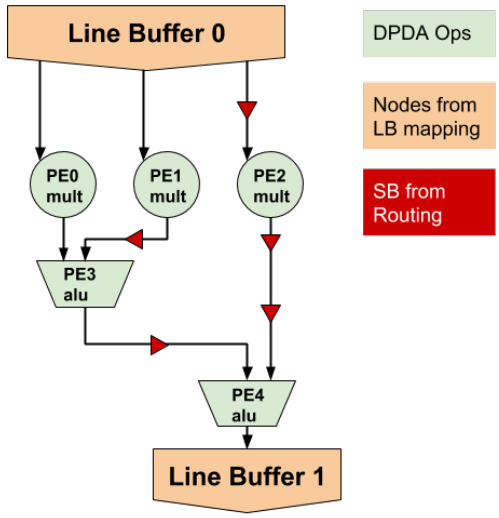

Figure 3.8: Place & Route adds routing information by inserting SBs involved in all connections.

Next we use another custom script to analyze the routing results and separate connections into two groups, one bit wires and buses, in order to correctly track the usage of routing resources. At this point the design *does NOT* include any pipeline registers and the maximum frequency is quite low. Next we perform a static timing analysis to determine the timing delay of all the signals in the graph. Every time the total delay reaches a certain threshold, the tool adds a register which resets the delay. Now we have a graph that meets timing but the design is broken because the data arrives at different clock cycles. To fix it, we calculate the cycle delay along each edge and determine how many registers have to be added on each connection. This gives us the graph that is causal and meets timing (Fig. 3.9). This greedy method is simple but it isn't optimal and doesn't guarantee the smallest number or pipeline registers.

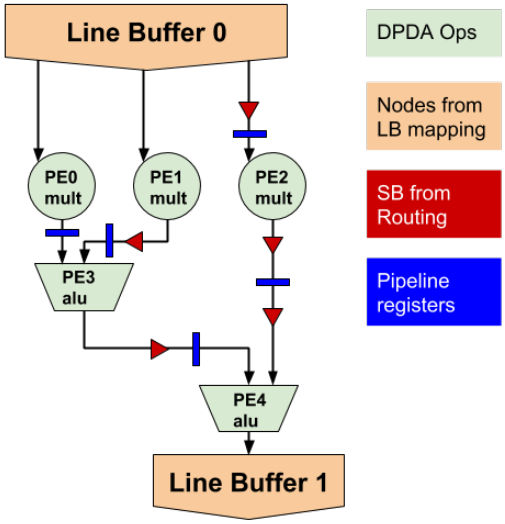

Figure 3.9: Pipelining stage improves critical path by enabling registers along signal path.

Since the pipeline registers are located in the interconnect network, they can be enabled **after** the routing is done. This allows us to guarantee that the design will meet the timing estimate set by the logic blocks and will not depend on application complexity, allowing us to maintain the simple abstraction that any chain of operation can be done in the same cycle. Our algorithm relies on the fact that application graphs don't have any feedback loops, so we can add any number of delays (registers) to any edge and then determine the required delays on every other edge without falling into infinite loop. It also exploits the abundance of pipeline registers along any connection paths allows us to use VPR place and route results unmodified. Besides registers in the SBs, we can use the registers inside logic box as an extra pipeline stage (see Section 5.1.2).

3.4 Summary

Flexible routing in our CGRA is implemented using the same approach as in modern FPGAs: island style with heterogeneous wire segments. Because of this, we were able to reuse many of the optimizations (SB patterns, heterogeneous wire segments with single driver) and development tools (VPR for Place & Route) previously created for FPGAs. For our CGRAs' interconnect we also took advantage of the some characteristics of imaging applications: like feed forward DAG structure with coarse grain operations (arithmetic). This has allowed us to effectively use several less common techniques: bus based routing and pipelined wires to increased area efficiency and clock frequency (bandwidth/performance) of our interconnect.

Chapter 4

Memory subsystem: Line Buffer

Our CGRA's memory subsystem is built around a Line Buffer abstraction commonly found in image processing applications. Just like in ASIC implementations, the LB is split into two elements: a bulk row storage unit (the *RS block*) and shift registers, but in CGRA both have to be flexible and support arbitrary sized stencils. We achieve this by reusing a basic RS block which we chain together to create a larger LB, and by implementing shift registers using pipeline wires. Each RS block is built such that it can store 2 rows of data with 4K pixels/row and uses the same single port SRAM based microarchitecture as an ASIC module. In addition to being flexible, the LB also can create an extended ring of pixels in the boundary region of the input image, to cause the kernel's output image to have the same dimensions as the input.

4.1 Line buffer abstraction

As was described in Section 1.1.2, a line buffer is a compact method of buffering the output from an image processing kernel, so it can be used by one or more kernels as their input. Rather than storing the entire image, it only keeps the working set needed by the consumer kernels, and discards the data that will never be read again. When the kernel process data in the raster scan (row major) order this working set is just a few rows (or lines) of the input image.

In one cycle, a kernel reads an $N \times M$ window of data from the LB, which corresponds to the size of its input stencil and generates an output pixel. In the next cycle, a new $N \times M$ window of data is read corresponding to the data window shifted one column to the right and a new output pixel is generated in raster order. When the data window reaches the right end of the image, it shifts one row down and back to the left end of the image, and the operation repeats. Thus, an image row is read M times for an $N \times M$ kernel. We can store $M - 1$ image lines in a local buffer (called the line buffer or LB) and exploit this data reuse. Similarly, we can use the fact that two neighboring stencils have significant amount of overlap - the data is mostly the same and the difference is only

one column (see Fig. 4.1). It means that a LB can be built by reading each stencil column only once from a memory and placing it in a small shift register. That data is used N times by the $N \times M$ kernel.

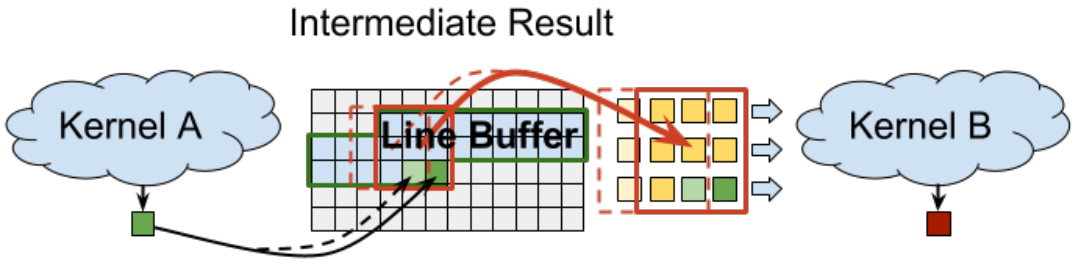

Figure 4.1: Line buffer only stores a few rows of pixels and produces stencils that differ only in one column.

Because image sensors produce data in raster (row-wise) order, the width of each line buffer must be equal to the image width, while its height is equal to the height of the buffer's consumer kernel window height minus one. Alternatively, a full frame of sensor output can be buffered in DRAM, and then processed in vertical strips, reducing the width of the required line buffers. This extra DRAM read and write usually makes the stripped system less energy efficient, although it does reduce the line buffer area. A comprehensive treatment of the tradeoffs involved with stripping can be found in Brunhaver's thesis[17].

Figure 4.2 shows the detailed operation and construction of a simplified ASIC line buffer feeding an $N \times N$ kernel. On each cycle, one pixel is driven from the sensor to shift register SR_0; from SRAM_1 to SR_1; and from SRAM_{N-1} to $\mathrm{SR}_N - 1$. These operations load a new column into the shift registers. During the same cycle the output of SR0 is written to SRAM_1, SR_1 is written to SRAM_2 , and SR_{N-2} is written to SRAM_{N-1} to move the data into the right SRAM for the next line.

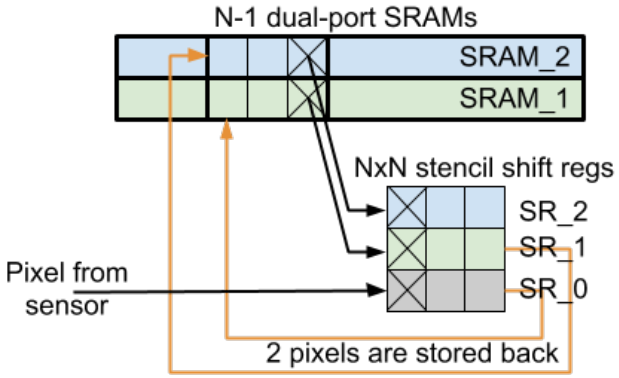

Figure 4.2: Multi-SRAM line buffer architecture and operation for 3x3 kernel.

This simple line buffer consists of $N-1$ dual-port SRAMs: SRAM_1 to SRAM_{N-1},each containing an image line; and N shift registers, SR_0 to SR_{N-1} holding N pixels each. During each cycle, the line buffer receives one input pixel. The incoming pixel goes to SR_0, which represents the newest row of the image, while the remaining shift registers SR_i $(0 < i < N)$ are fed by reading one pixel each from SRAM_i. This feeds a new column of pixels into the shift registers, so the needed N×N data window for the kernel is provided by these shift registers. As the new column of pixels is stored in the shift registers, the column of pixels shifted out of the registers is written back to the SRAM, but shifted by one row: each shift register SR_i is written into SRAM_{i+1}. This row shift moves the data into the right location for computation of the next line in the image (the data from the top line, SR_{N-1}, is not needed any more and is dropped).

Actually rewriting the data back into the SRAMs is not technically needed. Instead, since the oldest pixel (top-right in the stencil in the diagram) is no longer needed, the newest pixel (bottom-right) goes from SR_0 to SRAM_{N-1} for later usage, over-writing the dropped pixel. After completing the line, one can use an N×N crossbar to "relabel" the SRAMs (SRAM_i becomes SRAM_{i+1} mod N). This reduces the number of memory writes, but adds the cost of a crossbar between the RS units. Later we'll see that that by taking advantage of existing routing resources in our CGRA design, this crossbar becomes almost free.

As mentioned previously, imaging applications are more complex that a simple pipeline. In general they can be represented as a DAG of kernels and line buffers. Each kernel is writing its output to a single LB but a single LB might need to send its image data to multiple kernels, each with a different stencil. The LB has to support multiple kernels and provide producer/consumer relationships between all kernels connected to this LB. Since the LB in this case has one writer, but multiple readers, the data can't be freed from the LB until all consumers finished with it.

Another functionality which is assigned to LB is some implementations[25] is handling image borders. This situation occurs because a $N \times N$ stencil kernel "shrinks" the output data - for input

image of $W \times H$ pixels, it produces and output which is only $(W - N + 1) \times (H - N + 1)$. This is often an unwanted effect. To avoid this image shrinkage, the kernel input data must be extended by $(N - 1)/2$ pixels on each side to become $(W + N - 1) \times (H + N - 1)$. Missing pixels can be filled with constant (0 is often used) or replicated from pixels at the nearest image edge. A third choice used occasionally in software (often for FFTs) is to extend with data from the opposite edge of the image. This can't be done efficiently in hardware and won't be considered further.

4.2 Line buffer implementation

Like in an ASIC, our implementation of the line buffer will be split into two portions: the memory which we will call Row Storage (RS) and a Shift Register (see Fig. 4.2). RS is a bulk storage designed to keep several rows of pixels and it is responsible for outputting the column of new data, while the Shift Register is much smaller in capacity and it just combines the current column with the previous $N - 1$ columns to form the complete $N \times N$ stencil needed by the kernel. This design fits well with the raster (row major) order of the data coming from the image sensor and also with the abundance of the registers in the architecture due to pipelined wiring.

Since we have many registers in our programmable wires, we can easily distribute the Shift Register to improve the timing characteristics of the PEs that process this data. We can also replicate some registers if there is a large fanout on the path and the kernel is spread across the chip. In fact, we can treat the shift registers not as elements that have to be placed, but rather as parameters on the connections that can be accounted for *after* place & route, during the pipelined wires delay balancing stage. This procedure essentially removes the shift register stage from the design we need to place and route. To do this we "connect" all the kernel inputs that read from the same stencil row together, and connect them to the correct output of the RS block. We then annotate each of these inputs with the number of clock cycles that this input should be delayed to contain the desired stencil pixel (the delay sets the pixel's location in the row). This annotation is used during pipeline and retiming to create the right pixel delay effectively creating the needed shift register.

Building a flexible row storage unit is more problematic. It needs to support a variable capacity, both in terms of the row length (which depends on the image size), the number of rows that are needed (which depends on the stencil size). We also don't know how many LB instances an application will require. There are two approaches to solving this problem: creating large memory blocks that can hold many RS units, *or* creating small memory blocks which can be combined to create a larger RS unit. As we will see in Section 4.2.1, large blocks are not as efficient as smaller one which leads us to use smaller RS units as the basis for LB implementation.

4.2.1 Choosing the block size of the RS unit

The Row Storage block is bulk storage and just like most other on-chip memory subsystems, it's built using SRAM macros. There is a variety of library SRAM blocks available in different configurations. In general, blocks with smallest capacity have best *access energy per byte* while blocks with highest capacity have best *area per byte* (see Fig. 4.3).

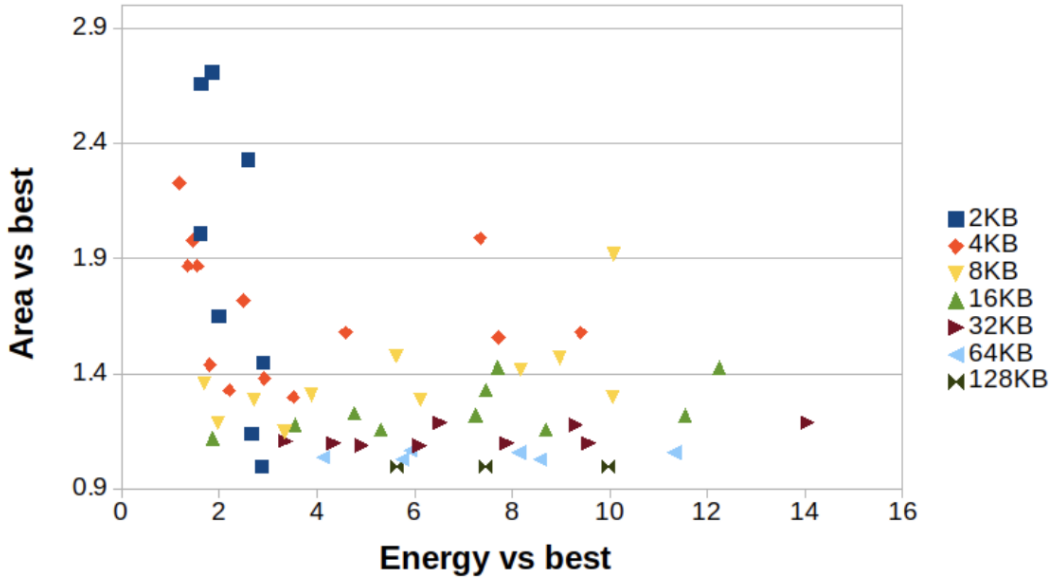

Figure 4.3: SRAM blocks energy/area based on 28nm library.

For a programmable fabric, *both* energy and area efficiency parameters are important, thus we are interested in the lower-left corner of the graph. Two blocks seem promising: 8KB and 16KB. Besides the total capacity, energy of an SRAM access also depends on the width of the interface. Looking at the selected 8KB and 16KB configurations (Fig. 4.4), we see that that the differences between configurations can be as much as 6x and at that the best ones use a wide 128 bit interface. This has a major effect on the overall design, since the pixel width is only 16 bit or 8 times smaller. If we want to use this block, we'd have to accumulate multiple pixels before every read, and similarly would have to use all 8 pixels we get from each read.

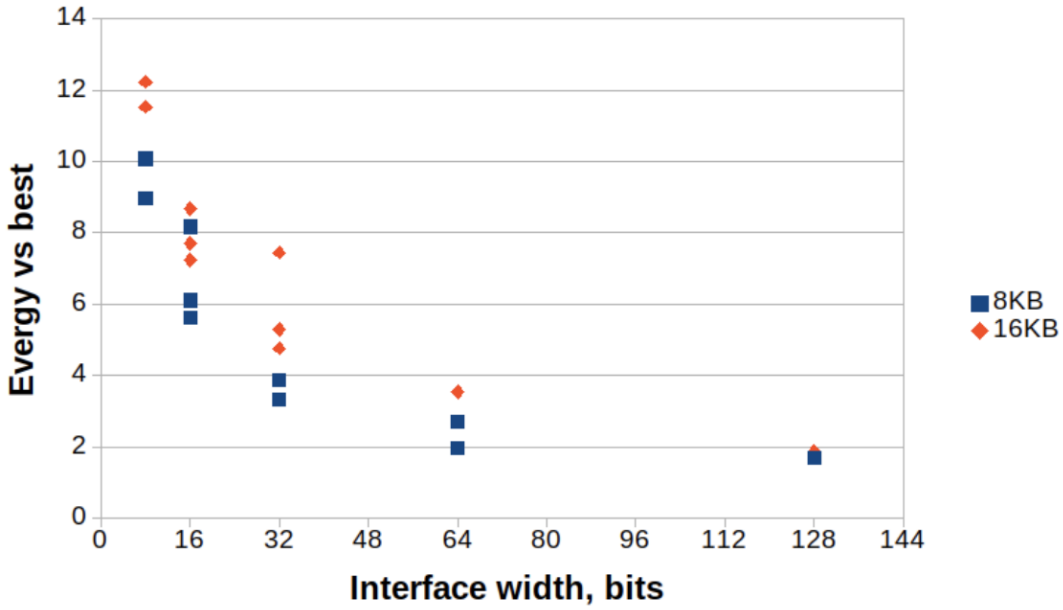

Figure 4.4: SRAM energy vs interface width.

Given that 8-16KB memory sizes represent a good energy/area tradeoff, we next explore how to use them to create the RS units we need. Modern smartphones use high resolution sensors for photography with at least 12MPx in rear camera and 8MPx in the front. So we expect that most of the algorithms will be processing 4K images (12MPx) and each pixel will be 2 Bytes (16bit). Additionally, we know that ISP pipelines are composed of kernels that often process odd sizes of stencils. A 3×3 stencil has to store two full rows or 2*4K*2B =16KB, so a 16KB macro will be enough to cover this basic case and for larger kernels we can use several such blocks by chaining' them to form a larges configuration that can hold more rows. This is similar to how FPGAs implement large capacity memories from several smaller modules.

In addition to photography, an ISP is often used in video processing applications which relay on older formats with smaller resolution. For example 1080p (HDTV) is still widely used and only has 2MPx frames with 1080 by 1920 pixels size. These smaller images might lead to large unutilized portions of the SRAM blocks and decrease area efficiency. Later in this chapter we will describe a way to mitigate this problem, by making more complex control logic that allows two independent RS units to share one storage element.

4.2.2 Basic RS block design

Design of the RS block will be based around our efficient 16kB single port SRAM with wide I/O, and can support a three row stencil. It adopts most of the same microarchitecture as previously

proposed for ASICs[17](see block diagram Fig. 4.5). The SRAM macro we have chosen has a *single* 128 bit read/write port, however, logically each cycle we need to write one 16 bit and read three 16 bit pixels, one of which is a copy of the input. So the bandwidth requirement is only 48 bit/cycle. To make this single ported block act as a dual port we need to divide the bandwidth among two ports and to add some buffering to eliminate bubbles.

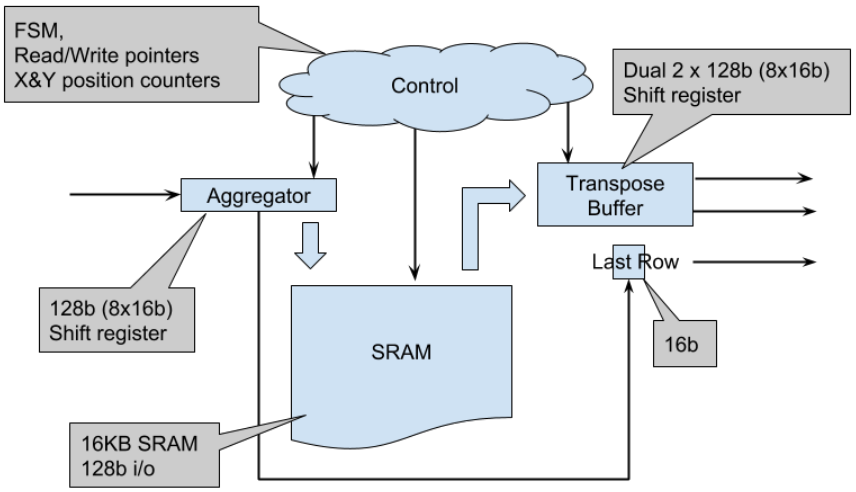

Figure 4.5: Design of an RS block based around single port SRAM with wide interface.

A line buffer always has a single kernel writing pixels into it, so the write port will need to accept 1 pixel (16 bit) per cycle. It will accumulate 8 pixels (128 bits) before performing an SRAM write, both to maximally use the SRAM bandwidth, and to avoid doing a masked write which further wastes bandwidth and energy. We refer to this write buffer to as the *aggregator* and implement it as a simple shift register inside each RS block. A copy of the write data will be also stored in the *last row* buffer to provide the data for the last, third row of the current stencil. This adds a third data output to the RS block. In most cases, this extra output is not required - due to static scheduling and known timing of the RS block, we can use the delayed version of the write data. The main benefit of our choice is that it allows "repeat" border handling (see Section 4.5) and the cost is not as big as it might seem. Usually, adding an extra output to the block requires larger multiplexers in SB to support more that one output, but RS block is about 7 times larger than a compute tile, so it will occupy several(probably between 2 and 3, since $sqrt(7) = 2.6$) vertical locations on the chip layout grid and will have several SB nearby. Each of these SB is already provisioned to support one logic block output, so we are not adding any extra hardware. The number of SB is defined by physical design constraints and it affects the number of output ports each controller can drive. We assume that the SRAM has a height equal 3 and there are 3 SB associated with it to match the 3 bus outputs from basic RS block design.

Since the data comes from sensor in raster order, it will be stored in RS is the same row-major order. We will accumulate 128b/16b = 8 consecutive pixels in the "aggregator" and write them to an SRAM address. Thus all pixels in an SRAM word will belong to the same row and have the same y coordinate. However, the output of the RS block should be a column of pixels, these pixels would have different y coordinate and come from different SRAM locations. When we perform a read, each word will have 8 consecutive pixels from that y coordinate which means it can be used to create next 8 outputs. To avoid reading the same data multiple times, to save access energy and use the memory bandwidth efficiently, we store each world read from SRAM into a *transpose buffer*. The *transpose buffer* forms a double buffered[1] two dimensional shift register with 8×2 pixels in each copy. After one copy is full, we shift its content 8 times to generate the top 2 pixels of the output column, meanwhile the data is loaded into the other copy. The final pixel on the output comes from *last row*, which is just a delayed version of the incoming data.

With 2 rows stored in SRAM two reads will serve eight stencil columns and there will be one SRAM write for every eight pixels. This means that we only utilized 3/8 of the available memory bandwidth - we will need the rest later for better support of narrow images and bigger stencils.

To keep track of read/write operations to the SRAM and make all the elements work together each block has some amount of control logic. This logic will keep track of the data and will generate signals for data flow control (`ready, valid`). It will also provide the ability to "chain" several basic blocks together. The RS block is dominated by SRAM and the overhead from all the control logic is about 15% of the SRAM area.

4.2.3 RS block operation

In this section we describe the operation of the RS block in more detail. Its state is determined by the content of all the memories (SRAM, TB) and value of read and write pointers. We will also use the following signals:

- `wr_valid` - indicates that the pixel presented to the RS is valid and should be stored
- `rd_valid` - indicates that the output of the RS is valid. This output is a column of pixels.
- `stencil_valid` - indicates that there is enough data columns produced by the RS to form a full stencil and this can be used to enable kernel computation. Note that this signal becomes the `wr_valid` signal for the output LB after accounting for the propagation delay through the kernel which is determined by the number of pipeline registers enabled on the critical path during "CGRA wire pipelining" step (see Section 3.2.3)

[1]Instead of doubling the transpose buffer, we can save one location in the second set of registers by timing the second SRAM read such that the data comes exactly in the cycle when the last column on of the current transpose buffer is driven out. However this doesn't lead to noticeable saving, because logic in the memory tiles is dominated by the SRAM. Additionally, most of the tiles in the CGRA are computation ones, so the net effect would be small. Double buffering make it easier to understand the operation and it doesn't rely on the strict timing requirements between kernels.

Since our CGRA implements a statically scheduled pipelined, for any LB in the system, all these signals are determined based on the pixel coordinate of the data written from image sensor and the internal cycle delay of the blocks. Thus, this discussion will use x, y image coordinates as a pixel id. We will use function of this id, $A(x, y)$, to indicate where that pixel is stored as it moves through the RS unit. The two image rows in RS are stored one after another so the mapping between pixel id and memory address is simply: $A(x, y) = (y\%2) * (W/8) + (x/8)$, where W is the image width in pixels.

At the start of operation all elements of the RS block are empty. The RS controller keeps track of the number or pixels written to and read from it. We will describe the state of the RS using write and read pointers which indicate the memory address where the incoming data will be stored and the address of the "oldest" (which has the smallest Y coordinate) output pixels. Note that due to static schedule of all the kernels, these pointers are tightly coupled, in fact the controller keeps just one value for the memory address because the input pixels always replaces the oldest one. Using two pointers is convenient abstraction that allows us to view memory reads and writes as independent processes which advance as soon as they can. Additionally, we'll denote the values of these pointers using pixel id: (x_w, y_w) and (x_r, y_r) , rather than the memory address because it specifies additional state information: how many pixels are currently stored, is RS in boundary region, etc. The actual SRAM address is calculated using $A(x, y)$ function.

When the `wr_valid` becomes one the input pixel is written to the aggregator at location $x_w\%8$ ($x_w mod 8$). When $x_w\%8 == 7$ the aggregator contains a full 128bit memory word which is then written to the SRAM at the $A(x_w, y_w)$ location and the write pointer is incremented. This operation continues until write pointer reaches $(W - 8, 1)$ which points to the last SRAM block on the 2nd row. During this phase, the LB is being filled - the data is only written to it, and nothing is read from LB.

Near the end of 2nd row, RS will prepare for the normal operation (which we call *active mode*) by prefetching the data. It will read two blocks at $A(0, 0)$ and $A(0, 1)$ from the SRAM and store them in the *next* section of transposed buffer (which is double buffered). This is done to free up the $A(0, 0)$ location so that in can be reuse by the next row of pixels which will be written to LB next. There two read commands easily complete before the write pointer reaches the end of the second line - $(W, 1)$. During the cycle where the final data block of the second row is written to the memory, *next transposed buffer* is swapped with the *transposed buffer* and RS enters the active mode.

In active mode (Fig. 4.6), every time a pixel is written to the *aggregator*, it is also written to the *last row* which is combined with the data from $x_w\%8$ column in the TB to form the valid column of pixels at the output. Thus in this mode `rd_valid` will be high. Every time the write pointer $x_w\%8 = 7$ *next transposed buffer* is swapped with the *transposed buffer* and when $x_w\%8 = 0$ we prefetch from SRAM at $A(x_w\%8+1, y_w\%2)$ and when $x_w\%8 = 1$ we prefetch $A(x_w\%8+1, (y_w+1)\%2)$ to *next transposed buffer*.

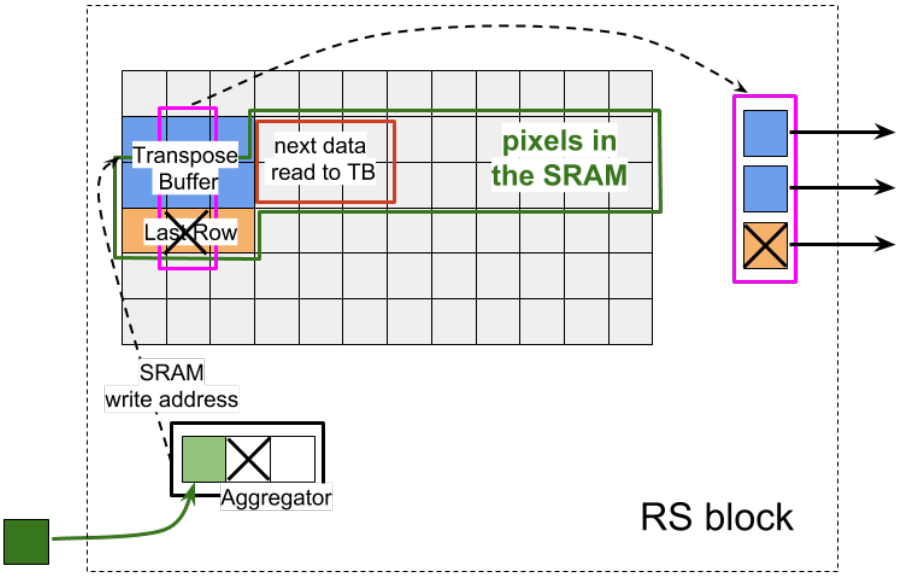

Figure 4.6: Operation of the RS block in active mode. TB is prefetched with the data from the previous rows, so every write to the RS causes a valid column on the output with 0 delay.

If the stencil width is N, then it will take that many cycles to fill the shift registers with the first valid stencil. After that every new output from RS will generate a valid stencil, thus `stencil_valid` would be low for the first $N-1$ cycles when `rd_valid` is high at the start of every image row, afterward it will be high every time `rd_valid` is high. Since `rd_valid` indicates the valid column of pixels at the beginning of shift registers, it also signals that the data has to propagate through the shift registers and through the compute kernel, so we can also think of it as a clock enable for that kernel.

Overall, during the first 2 rows `wr_valid=1, stencil_valid=rd_valid=0` after that for every row `wr_valid`=1 during W pixels, `rd_valid`=1 also during W cycles and `stencil_valid`=1 during $W-(N-1)$ cycles. The `rd_valid` signal will be just a 1 cycle delayed version of `wr_valid` and `stencil_valid` is a 1 cycle delayed `wr_valid` with first N high cycles set to low. The delayed version of `stencil_valid` will become a `wr_valid` to the output LB (Fig. 4.7).

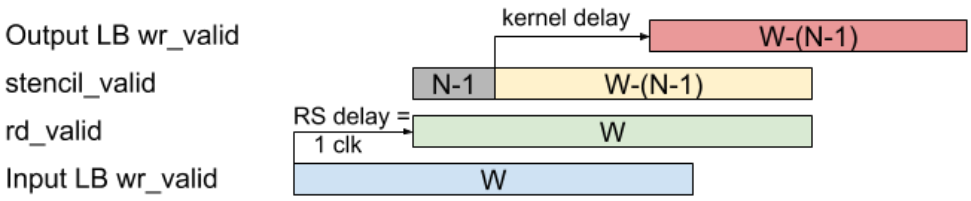

Figure 4.7: Timing of the RS control signals in active mode. The state of RS and the kernel is determined by the writes into the input LB.

The computation kernel is pipelined and just like with the shift registers it needs an enable signal to propagate the data. In our CGRA, we assume that this enable is implemented as a clock gate that controls clock distribution to a group of tiles, similar to a regional clock buffer in modern FPGAs[122]. As long as there are pixels coming out of the LB, the clock will be enabled and the will be moving through the shift registers and through the kernel. If there is a bubble in the incoming stream the clock will be shut off to stop the computation which will later be resumed. At the end of the image frame the clock will have to remain on for a few extra cycles to flush the pipeline. Thus the kernel enable will look like `rd_valid` with a few extra cycles at the end of the frame.

4.3 Creating larger RS units

Our basic RS block can hold 2 image rows with 4K 16b pixels thus supports stencils up to 3 pixels high but it uses only 3/8 of the SRAM blocks' bandwidth. For larger stencils we will use several of such blocks which together will act as one large block, and thus the control logic will have to be aware of this mode. Each basic block will be assigned two rows from the stencil: block 0 will hold rows 0 & 1, block 1 - rows 2 & 3 and so on. In this scheme we have to solve the problem of circular row eviction. It arises from the raster ordered of operations and the fact that the new data that comes to the RS has to replace the oldest pixel (see Fig. 4.8) so order of image rows that form the stencil rotates as we slide the stencil window down. In the case of single RS block, this problem was solved with double buffered transpose buffer and the fact that we could correctly fill each line just by accessing appropriate address in the SRAM. With more than one RS block we lost the ability to access *any* line thus we can't view SRAM as a crossbar between pixel rows.

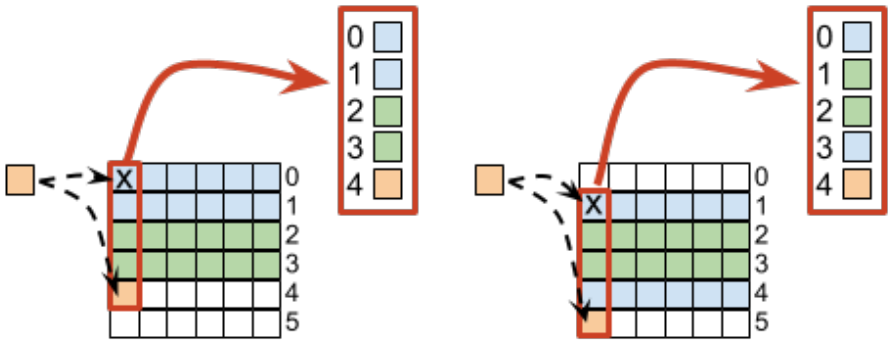

Figure 4.8: Circular shift in row mapping as stencil move to next row.

There are two ways we can solve this problem depending on how we choose to assign rows in the stencil to the RS blocks: *statically* or *dynamically*. With the static method RS block 0 *always* produces row 0 and 1 of the stencil *independent* of the output pixel's Y coordinate. With *dynamic* assignments the RS blocks are assigned specific rows in the image and thus the stencil rows will come from different RS blocks, depending on the Y coordinate of the output pixel. For example, when we process the first row of stencils, row 0 & 1 come from RS block 0 and row 2 & 3 come from RS block 1, while during second row of stencils, RS block 0 supplies row 0 & 3 and RS block 1 supplies row 1 & 2 (see Fig. 4.8).

We refer to the static row assignment as the *daisy-chain* scheme and the dynamic assignment as the *rotating* scheme. These two methods are complementary to each other and our implementation can use either one. Daisy-chaining supports stencils of any size but it adds between 10% and 30% to the energy of the LB, depending on the height of the stencil, while the rotating scheme is almost free from the energy point, but it is limited by the number of wire tracks and doesn't support large stencils.

4.3.1 Daisy-chain scheme

The daisy-chain scheme is shown in Fig. 4.9. It involves several RS blocks connected together in a daisy chain. This scheme solves the "circle shift" problem by essentially implementing a row shift register across all SRAM blocks. Every time a new pixel is written into a RS block *i*, the pixel evicted from block *i* is written into the block *i-1*. By the time we reach the end of the row on the input data, a full row of pixels would be copied from *each* RS block to the one before it in the chain. Thus the data in the RS block shifts down as the stencil window shifts down and RS block 0 *always* produces rows 0 & 1 of the stencil, independent of the output pixel's location. This scheme doesn't require any resources outside RS blocks or any new ports, it just chains the RS units together.

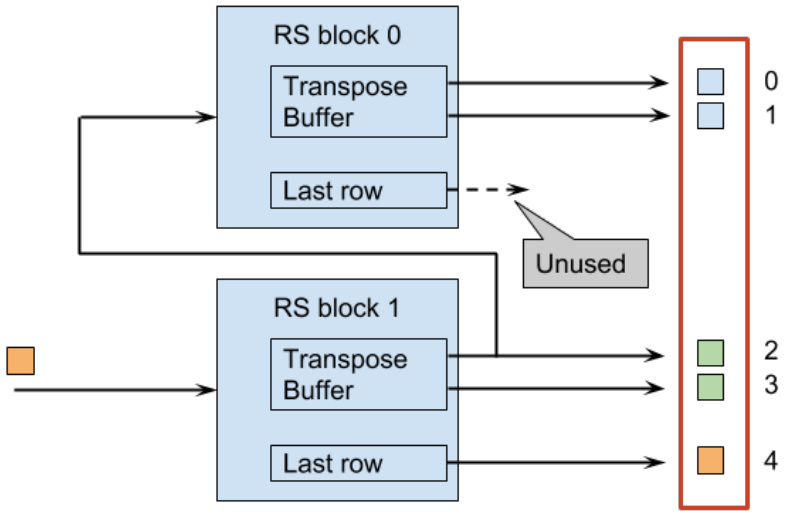

Figure 4.9: RS virtualization with static row assignment (daisy-chain).

All RS blocks in this scheme operate the same way as before with the write data to "older" RS units coming from the output of the unit below it. The main difference is how the control signals to the kernel are created and how each RS block is configured. The control of the kernel is set by the last unit on the chain, since it will have valid data last. RS blocks need to be programmed for an image of a different number of rows since rows of the image will be "stripped" from the data before the last RS unit sees it. At the end of each frame, all RS blocks are full and they "forget" this data by resetting the pointers and control signals before moving to the next frame.

The operation of this scheme relies on effectively 0 read delays in the RS block. When the first 2 rows are written in the first block, each time the next pixel will come, there will be a valid data in the top 2 positions of the output column (data which came from TB). The top pixel on the output of the first RS block is evicted and simultaneously written to the second RS (Fig. 4.10). The writes to all RS blocks will occur at the same time when LB as a whole is in active mode (after full $N-1$ rows were written). Until then, the `wr_valid` signal for every RS block will be tracking `rd_valid` of the previous block. In this scheme, all the RS blocks in the chain (except the fist) will have to have identical `rd_valid`, which would lead to a timing problem if we just tie the `wr_valid` to the `rd_valid` of the previous block. To avoid this issue, we **generate** the correct signals internally inside each RS block in the chain, base on the `wr_en` from the previous kernel which is connected to the `wr_valid` input of *every* RS block. All the RS blocks are programmed with their position in the chain during CGRA configuration, thus they all know by how many cycles `wr_en` has to be delayed to become `wr_valid`.

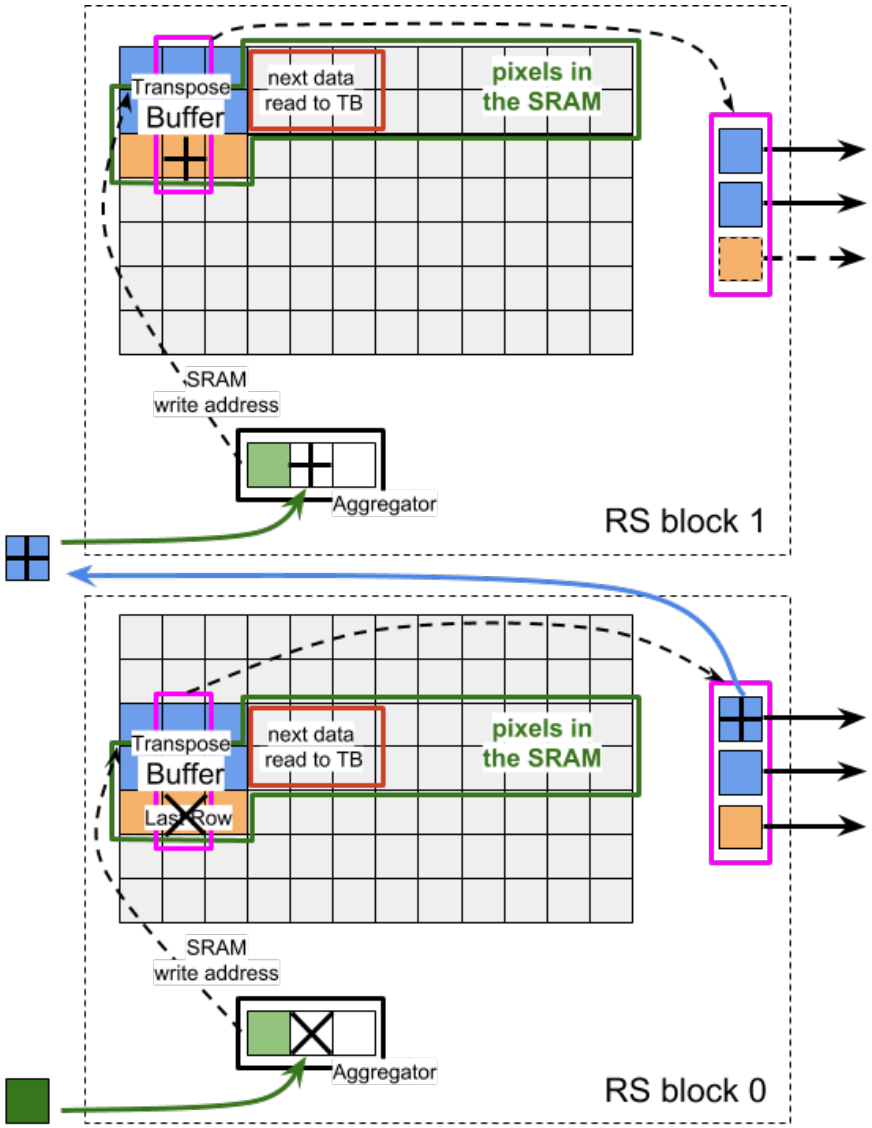

Figure 4.10: Operation of RS blocks in daisy chain is unchanged. In active mode data is written simultaneous in all block and relies on 0 delay at the output.

The down side of the daisy chain scheme is that it increases the energy cost of a RS unit. If an RS unit is supported $N-1$ rows, then it would be doing 1 write and $N-1$ reads to SRAM every 8 cycles, or $N/8$ SRAM access per pixel on average. With daisy chaining and 2 line per unit, we have $(N-1)/2$ blocks doing $3/8$ access per pixel on average, or $3(N-1)/16$ total. This means that breaking the RS into two line blocks increased the RS energy cost by $(N-3)/16$ accesses per pixel. We can model the relative energy increase of a daisy chain as a function of N and the number of operations in a kernel

- O as: $Relativecost = (3(N-1)/16 * E_{SRAM} + O * E_{operation})/(N/8 * E_{SRAM} + O * E_{operation})$. Using the energy numbers from Tab. 6.2 in Section 6.2.2, and assuming that 10% of operations are multiplications, 90% are "addition" and each operation needs one "communication", the cost of operation is: $E_{operation} = (0.9 * 1 + 0.1 * 13 + 1.6) = 3.8$, while the cost of a memory fetch is: $E_{SRAM} = 47$ the cost of an add. As we can see from Fig. 4.11, the relative energy per operation increase with this method is quite high for small kernel sizes. We need about 80 operations to make the overhead small: 1.8% for stencil height 5, 3.4% for height 7 and 4.9% for height 9.

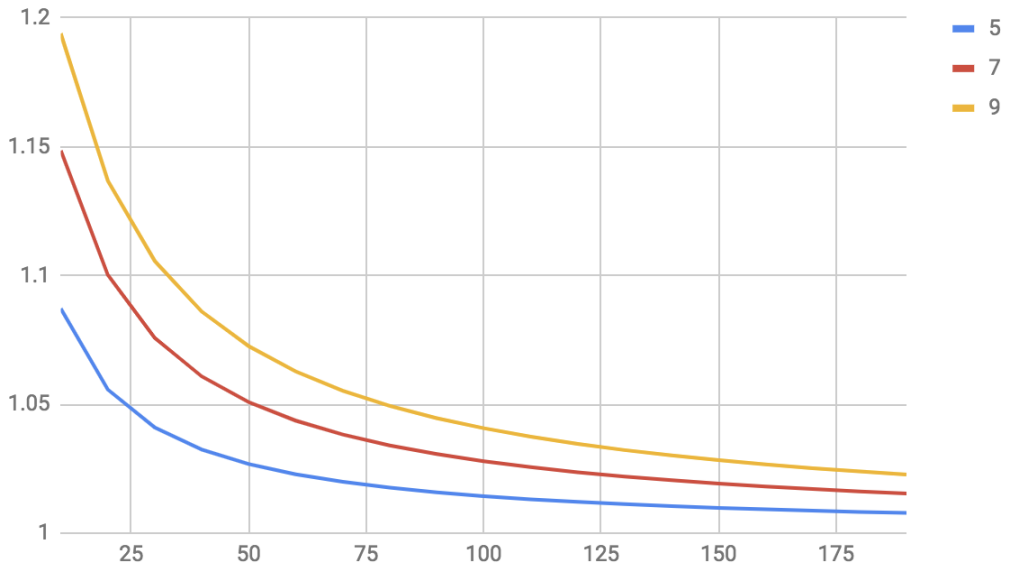

Figure 4.11: Relative energy cost of the daisy chain method for kernels with stencil height 5,7,9 depending on the total number of operations.

4.3.2 Rotating scheme

To eliminate the energy overhead when creating larger RS blocks, we need to remove the additional SRAM writes and use "rotating" scheme (Fig. 4.12). The main difference from daisy chain is that each pixel is written to SRAM only once. Different rows of the input image data are mapped to different RS units. This means that the input is connected to all RS blocks, instead of just the last one and that `wr_valid` for each RS block is generated separately and internally, based on the `wr_en` signal from the previous kernel and Y coordinate of the incoming pixel. The data is now written only to **one** of the blocks but the data coming *out* of the RS blocks will change order at the end of each line, because the RS block holding the top row of pixels for the output stencil changes depending on Y. Thus the output form RS blocks needs to be connected to a crossbar to ensure that the data remains correct when it reaches the compute units.

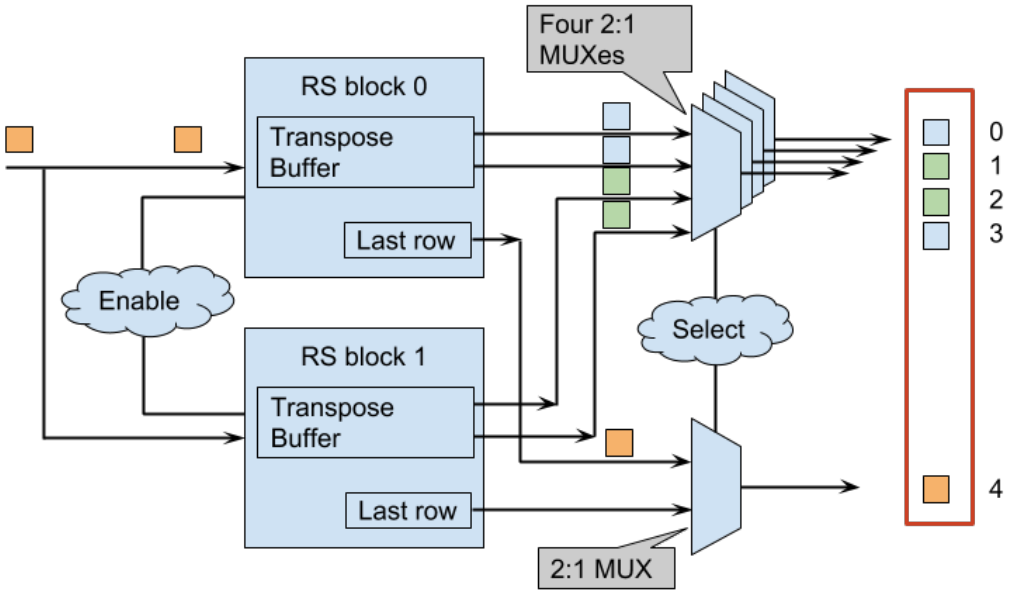

Figure 4.12: LB for larger stencil with rotating scheme.

To make the "rotate" scheme work, we will need to use blocks of the CGRA to build the crossbars. These crossbars can be efficiently implemented using the multiplexers in the SB, which are part of the routing resources described is Chapter 3. As described in Section 3.1.3, each multiplexer in SB can select between three input from different dimensions and an output of the logic block. We can use this switches to make a crossbar, as long as the number of vertical tracks is larger than the number of LB outputs. We allocate track i to always have the top row of the stencil; tack $i+1$ has 2nd row, etc. For each new row, we change the multiplexer select to connect the "right" memory to the correct vertical track based on position of the output pixel. So we can use the wire tracks to bring the output of different RS blocks here. A simplified diagram for this approach is shown on Fig. 4.13.

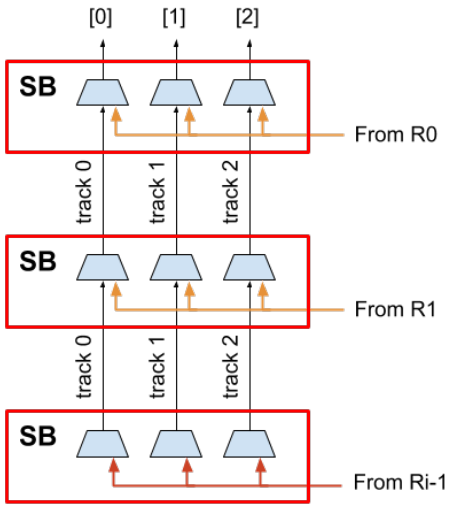

Figure 4.13: Implementation of "rotating" scheme using SB resources results in a hierarchical cross bar for outputs and requires dedicated wire track for each output.

In this rotation scheme we had to make `select` of the SB multiplexers dynamic, rather than static. This is a fairly easy and inexpensive change, but it's a **major** difference from traditional FPGAs. It is also different from dynamic fine grain reconfiguration technique proposed by Trimberger et. al. [106] and later used in Tabula FPGAs[116]: instead of reconfiguring every element on the chip multiple times per cycle, we are *infrequently* reconfiguring only a *small portion* of our CGRA. The reconfiguration is done only at the end of an image row, thus it will happen only *once in a few thousand cycles*. Because this is a rare event that affect a small portion of the chip, we expect the energy cost of such reconfiguration to be negligible in our CGRA.

This scheme is limited by the number of wire tracks and by timing. Since every pixel in the output column can come from any RS block, we have to allocate *different* tracks for all the outputs and use vertical wire tracks to bring the data from all RS blocks, implementing hierarchical crossbar (see Fig. 4.13). The limitation on timing occurs because the LB output doesn't come directly from a register inside RS unit - now there several additional SB multiplexers on the path which implement crossbar. The number of these additional multiplexers varies with the position of the output pixels because it depend on which RS block is providing the data. If the worst case critical path through the crossbar fits within a period of the target clock frequency, we can simply add the pipeline registers after the crossbar, but the number of stages in the cross bar has to be small, hence limiting the number of RS blocks in this LB. We can remove this limitation by pipelining the hierarchical cross bar, just like in case with the regular connections (see Section 3.2.3), but we have to deal with the situation when the number of registers along the path changes based on which RS block is providing the data, thus resulting in a variable pipeline delay on the output of LB. This

problem can be solved by adding a programmable number of output registers inside RS block so we can adjust it for each RS unit individually, depending on which block is supplying the data.

Our implementation of the rotate scheme has a crossbar at the data origin - inside LB and we use SB blocks of the RS units that constitute LB. An alternative approach would be to place the cross bar at the receiving end - inside kernel, near each PE which is connected to LB, by using multiplexers inside CB. We didn't use it because it would require signals from all RS blocks to be routed to every tile that performs operation using data from the input stencil, creating routing congestion. It would also require multiple crossbar implementations per kernel, rather than just one.

4.4 Supporting different image widths

Memory subsystem design is about efficiency optimization while balancing capacity and bandwidth requirement. For our basic RS configuration we were mostly driven by area and energy efficiency assuming a 4K image width and relatively small kernel height. As a result, our design fully used the capacity of SRAM, archiving full area efficiency, and needed at most only 3/8 of the memory bandwidth. However for smaller images, the utilization quickly drops while large images won't fit.

For images smaller than 4K wide, we can take advantage of unused bandwidth and either map several RS blocks to the SRAM or increase the number of lines per RS. With 8 pixels/cycle total SRAM bandwidth and 1 pixel written to the block every cycle, there are $8 - 1 = 7$ pixels/cycle of available read bandwidth and we could theoretically store up to 7 lines supporting stencils with height 8 and image width 16KB/7 lines/2B per pixels = 1170 pixels/line. However, increasing the number of line per buffer wouldn't solve the area efficiency problem, because kernels often use smaller stencils, like 3x3 or 5x5 and we would have to find a way to split this block. Instead we prefer a different approach that can map several small RS blocks to one SRAM. The exact number of RS units depends on the maximum port bandwidth of the memory block.

Given the SRAM bandwidth of 8 pixels/cycle, we can map two RS blocks with two lines of 2K pixels each, which will use 6/8 of the bandwidth. We also have to increase the number of ports and the routing resources to accommodate extra inputs and outputs of two independent RS macros. As discussed previously, due to the large area of SRAM macro, each RS block occupies 3 grid locations and has 3 SBs which can drive 3 outputs, but two logical RS blocks will have 6 outputs. To support the extra outputs we would have to make each multiplexer inside SB (see Section 3.1.3) in the memory tile bigger and change it from 4:1 to 5:1, so that it can select between 2 outputs and 3 inputs to the tile. We also have to account for additional CBs and extra control logic in RS units. All these would increase the area of a memory tile, but since most of the CGRA tiles are computational, the total chip area increase would be relatively small.

To implement two RS controllers per SRAM we can replicate most of the control logic from Fig. 4.5 which gives us fully independent blocks. However, if we plan to use them in the same imaging

application, this is not optimal. Instead, we can take advantage of the static scheduling and the fact that two RS will advance in a lock step with some known cycle delay. This means that instead of replicating the aggregator, transpose buffer and last row buffer, we just double the interface to them. The hardware would look the same as before - transpose buffer and aggregator would still have the same number of registers as before, but now they both would have double the bandwidth (see Fig. 4.14). Aggregator will get two pixels (which come from different kernels) or 32 bits written to it per cycle, and TB will produce two columns of data instead of one. The data from the two kernels would be interleaved in the SRAM instead of mapped to different regions as would be the case with replicated RS controllers.

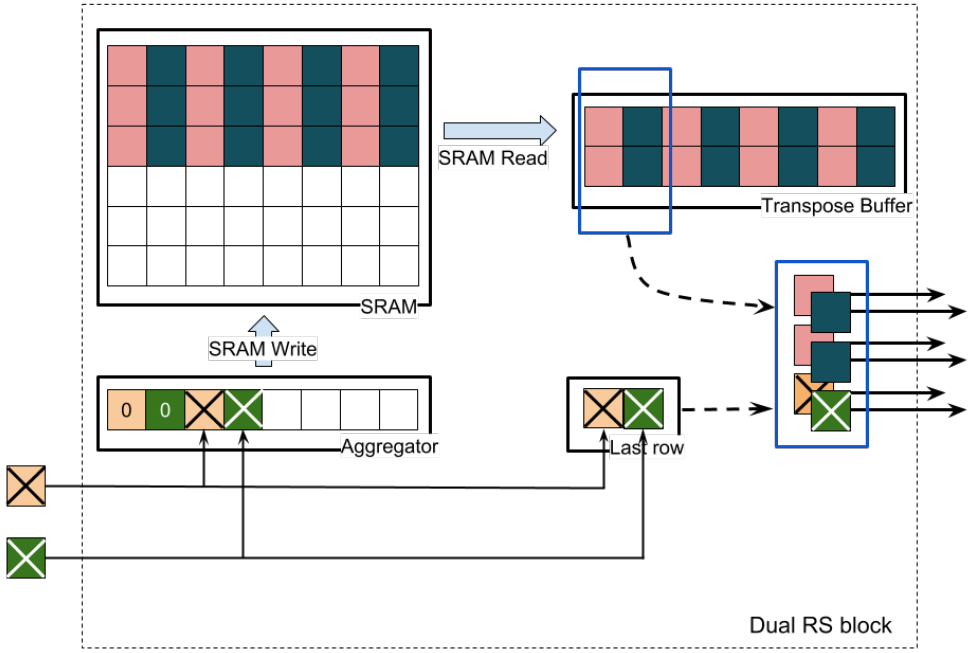

Figure 4.14: Two logical RS units can be implemented using the same hardware by doubling bandwidth to the buffers and interleaving the data.

Each small RS block will hold 2 lines of 2K image if we split one SRAM in two. There is still some amount of unused memory capacity for older video formats with 1080pxl per line, but now the area efficiency in RS is about 50% instead of 25%. To make it better we would have to use smaller SRAM modules. However in this case, we won't be able to support 4K image lines natively and would have to find a way to map larger image size.

If the image width is between 4K and 8K, we can map one line per RS and use rotating or daisy chaining scheme to get required amount of storage. For dimensions larger than 8K, we need more than one SRAM to fit a full line. The easiest way to handle this is *striding*, or dividing image

vertically into chunks (called *strides*) that fit into memory. This method doesn't require changes to the RS controller, all we need is to add multiplexer at the output to choose the correct data based on the X coordinate. The RS block outputs a single column of pixels, and if we select the correct data *before* it enters shift registers, we don't need to worry about making strides overlap. Selection of the right RS source of the data can be done using SBs, just like we did with rotating scheme for chaining (Fig. 4.12).

4.5 Boundary support

Since the stencils operations require a support region, the output image becomes smaller than the input data. For example, with $N \times N$ stencil and $W \times H$ image, the result is $W-(N-1) \times H-(N-1)$ which correspond to the range of valid positions of the stencils' center point (aka centroid) in the input image coordinates. In some applications (like CNN), shrinking image size is an undesirable property and in order to avoid it, the input data is extended or padded with synthetic values, such that its dimensions become $W+(N-1) \times H+(N-1)$. Alternatively, we can think of a valid centroid positions being the full input image dimensions and the portion of the stencil that fall outside of the input image (when pixel's coordinate is negative for example) being filled with synthetic data (see Fig. 4.15).

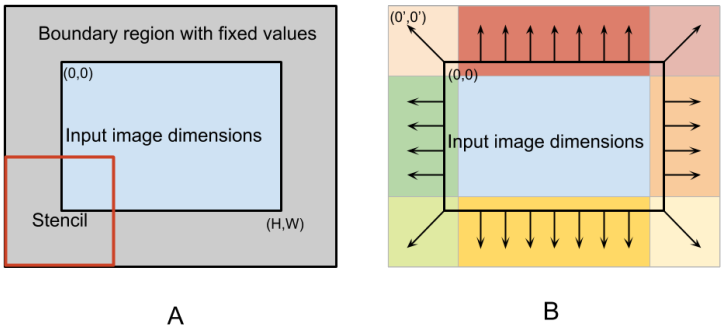

Figure 4.15: Boundary region and fill with constant(A), repeat edge (B).

The three common types of boundary handling which define what is used for the synthetic data are:

- Fill - uses some predefined constant, like 0 everywhere in the boundary region (Fig. 4.15 A)
- Repeat edge - use the closest valid pixel data from the input (Fig. 4.15 B). For example at the position (-X, Y) it will put data from coordinate (0, Y).
- Mirror - repeats the image to cover the entire coordinate space, such that the value at (-X, -Y) position is the same as (Width-X, Height-Y)

Fill with 0 is often used in Vision and CNN, while repeat edge and mirror methods are more common is imaging applications. Mirror is useful for applications that process data in frequency domain, but it's a poor fit for raster order steaming model dictated by the image sensors, because it requires the entire image to be stored before processing. For this reason, we *don't* support this mode. In imaging, it's not a big limitation because it affects only a tiny area of the image and other boundary modes can be used instead with minor loss in image quality.

In order to support the boundary region, a line buffer implementation has to keep track of the centroid coordinates which will identify the portion of the stencil that has to be filled with synthetic data. These coordinates identify the part of the stencil that has synthetic data. The missing values come either from the some other place in the stencil (repeat edge) or from a constant (fill). RS block will generate the correct column of the data and feed it into the shift register which creates the entire stencil. Boundary data can be generated in two ways depending on the choice of flow control.

Without the boundary support, if the stencil dimensions are $N \times N$ and the data written into line buffer has width $= W$ and height $= H$, then the LB will produce $H - (N - 1)$ rows during each row, `rd_valid` will be high for W cycles and `stencil_valid` is high during the $W - (N - 1)$ cycles for each row. So the number of valid LB outputs (stencil columns) is $(H - N + 1) * W$ and the number of valid stencils is $(H - N + 1) * (W - N + 1)$ - both numbers are less that the number of valid inputs (`wr_valid`) which is $H * W$. The situation changes if we want to generate a boundary region. Now the number of valid stencils has to be the same as number of writes to the line buffer: $H * W$. This requires H rows of pixel columns from the RS block which is equal to the number of incoming rows, but for each row we now need to produce $W + (N - 1)$ valid columns (`rd_valid=1`) in order to propagate the data through shift register. Out of these data columns only W contain the real data, the rest will have to be generated by the block, based on selected boundary handling scheme. Because we need to push $W + (N - 1)$ columns into a shift register, there should be at least $N - 1$ "dead" cycles between each row, where the data is not being written into the RS units. The advantage of this method is that the structure of *data preparation* is just a shift register - the same as without boundary support (see Fig. 4.16 A).

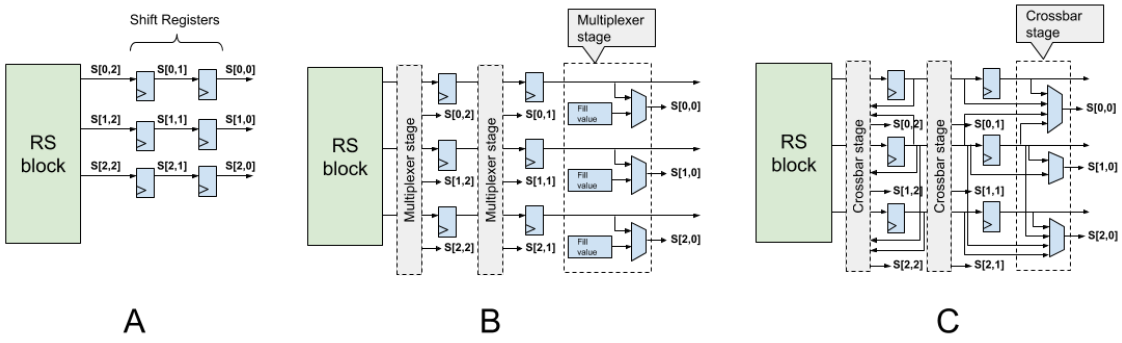

Figure 4.16: Design options to generate 3×3 stencil values S[y,x] with the boundary support: A - boundary values generated by RS units pushed through a shift register, requires N-1 "dead" cycles, B - "fill" boundary implemented in kernel, no "dead" cycles, C - "repeat edge" boundary implemented in kernel, no "dead" cycles.

An alternative approach to having these dead cycles, would be support boundary regions inside a computation kernel, rather than in a RS block. This requires a change of the *data preparation* stage (see Fig. 2.2) to a more complicated structure that includes a shift register to store previous outputs of the RS block and a *multiplexer* or *crossbar* stages to generate the final stencil data - S[y,x]. In case of the "fill" boundary (Fig. 4.16 B) we need a *multiplexer stage* after every row. This stage selects between a constant "fill value" and the content of the shift register to create each pixel in the stencil. It is a very simple and scalable structure because the hardware for each S[y,x] is identical (except at the center where the MUX is not needed) and doesn't depend on the size of the stencil. For the "repeat edge" boundary (Fig. 4.16 C) the final stencil values come from the *crossbar* stages which must select the correct source for the value at each location from many possible places in the shift register that might have it. The complexity of this structure depends on the stencil size and the pixel position within the stencil. At the center, the data always comes from the shift register. The worst is at the corners where the cross bar must select between one of the $1 + (N - 1)/2$ values on the same row of the shift register, $(N - 1)/2$ values in the same column of the shift register and the value at the center of the stencil.

A crossbar would be very expensive, due to large number of inputs to the multiplexers and challenging timing. Implementing it using logic blocks inside each tile would result in much larger total area of the implementation, so we would have to rely on the routing resources and dynamically manage them - just like in "rotating" scheme (see Section 4.3.2). The problem is that the total complexity of the crossbar stages depends on the stencil size quadraticaly instead of linear dependence in the "rotating" scheme, requiring much more wire tracks. For example, we could use large mux-es in CBs for the crossbar, but than means that multiple data signals, instead of just one, would have to be brought to each CB than uses this signals. Clearly, this would make some tracks unavailable

for general routing, resulting in larger number of tracks (thus making total chip area bigger) and the routing tool would have to be more sophisticated.

A multiplexer stage (Fig. 4.16 B) is much simpler and doesn't require more bus routing tracks - it only needs small storage to keep the "fill" value and multiplexer dynamically controlled depending on the X coordinate of the stencil. This structure resembles SB modification we added for pipelined wires (see Fig. 3.5), but those registers can't be loaded with value. More importantly, adding a select signal to each SB is challenging because this signal is generated based on X position and would have to be routed to the SB, which means it would require its own CB. Instead of using pipelined wire registers in the routing, we will rely on storage element for "embedding constants" feature (see Sect. 3.2.1 and Sect. 5.1.2). These registers are designed to be "loadable" with the value and each tile already has 1bit CBs which we can use to route select signal. Now the difference between no boundary' case and fill boundary' is the need to bring `select` signal to some PE tiles, which will be used by a PE to take the data either from the input or from local register. Such change **doesn't** require any router modifications - all the functionally will be hidden inside the tile. The select `signal` is using overprovisioned 1 bit routing network, so it's unlikely to have a negative effect on the number of tracks required and the cost of this implementation is energy for configuration switch, which is small because it only happens for a few pixels on each row. There is no need to wait for propagation delay through the shift registers, since the "fill" constants are stored inside each PE that needs them. These constants never change during the life of an application and they are loaded once during chip configuration, before any computation is performed.

Boundary handling is more important for computer vision application than imaging, because they use more kernels and smaller images. For example CNNs often have tens of layers resulting in large effective stencil size and they use about 224×224 pixels[57] images, compared to about 3000×4000 pixels for 12 Mpx camera. Since CNN typically use fill with 0 and because this method doesn't require routing tool changes or extra hardware or clock rate changes, we use "fill with constant" as a default boundary handling for all applications.

In case of "repeat edge" we rely on the extra clock cycles to populate the shift registers for every image row. Our CGRA uses 800MHz clock and all operations in application kernel are implemented on dedicated logic blocks without hardware time multiplexing, thus our chip can process 800 Mpxl/sec compared to 12Mplx/frame * 30frame/sec = 360Mplx/sec, so there is more than enough spare clock cycles available for each row. Beside higher processing clock frequency the extra cycles needed to fill the shift register can come from "dark reference pixels"[95] which are often added to each row of pixels in imaging sensors. These pixels are only used to estimate the amount of noise and don't contribute to the final result. To support "repeat edge", the RS block would have to output a valid column *every* time a pixel is written to it. This column depends on X and Y position of the centroid with values populated according to boundary rules (Fig. 4.16 B). The logic is entirely inside RS block and consists of additional control states and a few extra registers to hold previous

values. Since the value of *all* pixels in the output column is affected by the boundary handling, the only way to keep the logic inside RS block is to make `last row` the output of this block. The advantage of such approach is that the number of tiles in the implementation doesn't change based on the boundary behavior specified by the application.

4.6 Summary

Imaging applications follow line buffer abstraction and any chip targeting this domain has to be able to efficiently implement this abstraction. Our CGRA uses the same architecture as custom ASICs, it consists of shift registers and bulk storage centered around single port SRAM with wide interface. The challenge is to make it flexible while keeping the efficiency. Our approach is to "chain" bulk storage modules and to reuse other block of the design as much as possible for various functionalities in memory subsystem: pipelined wires as shift registers, muxes in SB and CB for boundary condition and as part of chaining. In case of CGRA, memory subsystem also implements synchronization between kernels in the dataflow architecture: the computation is performed as soon as there is enough data in the input LB to produce a full stencil and enough space in the output LB to store the result.

Chapter 5

Compute block: Processing Element

A kernel in image processing applications performs data transformation through computations. It can be represented as a DAG of simple operation on pixel data (see Section 1.1.2). In our CGRA these operations are performed by processing element (PE) tiles in the CGRA. This chapter describes a PE hardware generator which we built to explore this design space. We evaluate various configurations and functionalities of the PE, starting with the basic (minimum) set of operations required to implement a kernel and moving to a more complicated and full featured design. We develop a microarchitecture of the PE that allows it to support additional features with small area cost by maximally reusing existing compute resources. Later in the chapter we employ the same strategy to further improve the PE design by increasing the computational density for several patterns commonly found in ISP programs: dot product, sum-of-absolute differences and reduction with addition.

We explore the area cost of various PE configurations using Genesis 2[97] hardware generator framework. Genesis 2 uses Perl as metaprogramming language for industry standard System Verilog. Compared to a standard design flow it raises the level of abstraction by adding support for highly parameterizable hardware objects which can inspect other object in the design and modify the generated logic according to those parameters. This introspection is done during the first stage of the Genesis 2 flow - elaboration. Here the source code is treated like a Perl script that generates a set of Verilog files with some collateral (like .xml description of all the parameters). After elaboration, the constructed Verilog files follow the tradition design flow. Overall the process is similar to the way #marco and #define key words are supported during precompile stage in C/C++ language. Thus, Genesis 2 is different from other methods like High Level Synthesis[74] in that is doesn't require a new compiler and is easily compatible with existing tools and previously developed modules.

In order to make the architecture programmable its compute resources have to be able to implement any operations found in application kernel. To achieve this goal, we have chosen to build the Processing Element around a configurable Arithmetic Logic Unit (ALU) - similar to the found in CPU, GPU and other programmable in time architectures. An ALU can perform operation from the limited set, which is often called Instruction Set Architecture (ISA). The operation is selected by a code word which is called an instruction in the programmable in time machines and a configuration in programmable in space machines to highlight that it's not expected to change. This fundamental difference between the architectures doesn't change the microarchitecture of the ALU very much: nearly the same design can be used in both ways.

5.1 Simple 2:1

This section covers the simplest design for the PE. First we define an Instruction Set Architecture (ISA) which describe the minimum set of operations that our compute element has to support in order to support kernels from benchmark applications. Than we present the core compute (ALU) design that implements this ISA and a full PE microarchitecture. Finally we analyze the design in the context of entire compute tile.

5.1.1 List of operations (ISA) and compute element

The minimum requirement for the processing element is to implement basic set of operations (ISA) that constitute compute kernels (see Tab. 5.1). This set covers "compute" operations we have previously described in Abstract IR (see Tab. 2.1), it has all the native integer operation found in C++ (except divide) and includes several extra operations from DPDA. From DPDA, we have excluded reduction operations that take variable number of pixels (stencil reduction operations), like SUM(), MAX(), etc. We rely on technology mapping phase (see Section 3.3) to convert this stencil operations to a mini graph of two input operations. For example $MAX(x[0], x[1], ..., x[N-1])$ function over N pixels will be expressed as a chain of $N-1$ $MAX(a, b)$ operations, alternatively is can be converted into a binary tree of $MAX(a, b)$. Both options are supported as they use dual input version of $MAX()$.

Operation	**Semantic**	**Inputs**	**Result 16b**	**Result 1b (p)**	**HW name**
Minimum Set					
Addition	a+b+d	a,b - int16 or u_int16 d - 1bit	res = a + b + d	(a+b+d) >= 2^16	ADD
Subtraction	a-b	a,b - int16 or u_int16	res = a + ∼b + 1	N/A	SUB
Greater or equal	a>=b	a,b - int16 or u_int16	N/A	a>=b	GTE_MAX
Less or equal	a<=b	a,b - int16 or u_int16	N/A	a<=b	LTE_MIN
Equal	a===b	a,b - int16 or u_int16	N/A	a===b	EQ
Select	d?a:b	a,b - int16 or u_int16 d - 1bit	d?a:b	N/A	SEL
Shift right	a >> b[3:0]	a - int16 ot u_int16 b - u_int16	a >> b[3:0]	N/A	RSHFT
Shift left	a << b[3:0]	a - int16 or u_int16 b - uint16	a << b[3:0]	N/A	LSHFT
Multiply	a*b	a,b - int16 or u_int16	(a*b)[15:0]	N/A	MULT_0
Multiply middle	a*b >> 8	a,b - int16 or u_int16	(a*b)[23:8]	N/A	MULT_1
Multiply high	a*b >> 16	a,b - int16 or u_int16	(a*b)[31:16]	N/A	MULT_2
Or	a|b	a,b - int16 or u_int16	a|b	N/A	OR
And	a&b	a,b - int16 or u_int16	a&b	N/A	AND
Xor	a ^ b	a,b - int16 or u_int16	a ^ b	N/A	XOR
Extra Operations					
Absolute	$\|a\|$	a - u_int16	(a<0) ? (0-a) : a	a[15]	ABS
Maximum	MAX(a,b)	a,b - int16 or u_int16	(a>=b)? a :b	N/A	GTE_MAX
Minimum	MIN(a,b)	a,b - int16 or u_int16	(a<=b)? a :b	N/A	LTE_MIN
Leaky ReLu	ReLu(x,W)	x, W - int16	(a<0) ? (a*W)[31:16] : a	a[15]	RELU

Table 5.1: List of compute operations (ISA).

Beside the minimum set which is common between C++ and DPDA operations, Tab. 5.1 lists a few extra operation that exist only in DPDA. These extra operations can be expressed by using several operations from the minimum set, for example ABS(), can be defined using three other operations: SEL, GTE, SUB. These operations and commonly used by ISPs and the cost of supporting this operations directly in hardware is negligible compared to the benefit they provide of saving in the number of PE elements.

The hardware block that implements operations listed in Tab. 5.1 is shown in Fig. 5.1. It contains six main components each implementing a operation of a certain type: select, addition, compare, logic, shift or multiply. Only one of those blocks is driving it's result to the outside. This selection is controlled by `op_code` - configuration parameter. It's worth noting that the design of this block doesn't make any assumption about the kind of machine it will be used in - programmable in time or in space.

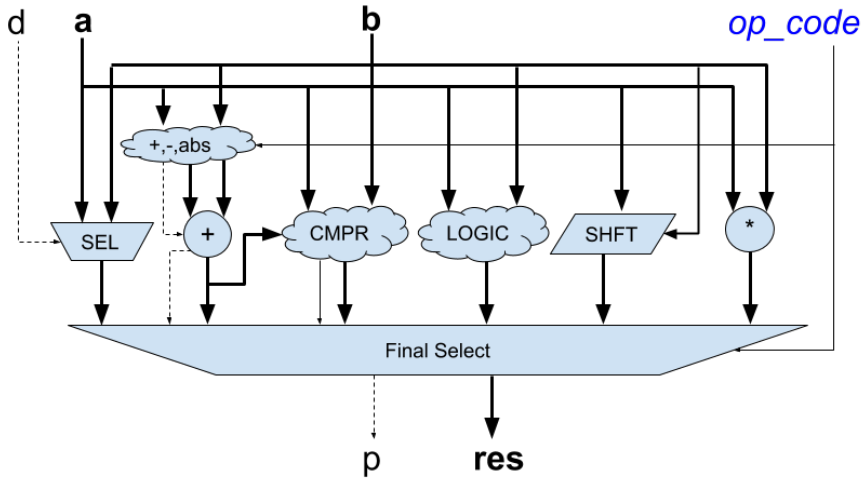

Figure 5.1: ALU block diagram - based on configuration word the result of one of the operation is selected as an output.

All the operations listed in Tab. 5.1 are 16 bit - they take 16 bit data inputs. We choose 16 bit data precision because it's the most common in our data set. Other data precision math can be expressed using operations on 16 bit data, although the efficiency suffers compared to the native implementation. Any lower data width (e.g. 1 bit and 8 bit) non saturating arithmetic can be implemented on 16 bit block directly without any extra step, but in order to efficiently support higher precision (32bit) we need to keep `carry` bit in addition to the result.

5.1.2 Processing element

If we look at the case of 1 bit data operations (or simply binary operations), we can quickly see that the design on Fig. 5.1 becomes extremely inefficient, even though it can do them. Out of 6 modules, we only need one - logic and even that we'll be used partially - in just only 1/16 of logic sub block is required. This is the reason why FPGAs are using look up tables - LUTs rather than ALUs and are much more efficient for logic operations. In image processing, binary operations do occur often enough to be concerned with their implementation.

Beside compute kernels, binary operations are also used to implement control signals required for synchronization so we need to handle them more efficiently. For this reason, in each PE block we include a 3:1 LUT. This configuration takes advantage of the fact that only a few operations in Tab. 4 use 1 bit output p (mostly comparisons) and 1 bit input d, thus we can share the i/o ports between LUT and compute core blocks and still have a high chance that the same PE can perform both 16 bit and 1 bit operations simultaneously. The 3 input look configuration can implement more than one dependent binary operations - as long as the result is 1 bit. The size of this LUT configuration is also quite small compared to the compute core, such that it doesn't affect the area much even when it's not used.

In real kernels many operation use an immediate value as one of the operands. It could be a true constant required by algorithm (like 0 or 1) or a tap value, like a filter coefficient which might change between different invocations of the application but is held constant while the application is running. Rather than sending such constant on the expensive communication and routing network, it's better to keep them locally in each PE using dedicated storage(Fig. 5.2) and save interconnect bandwidth.

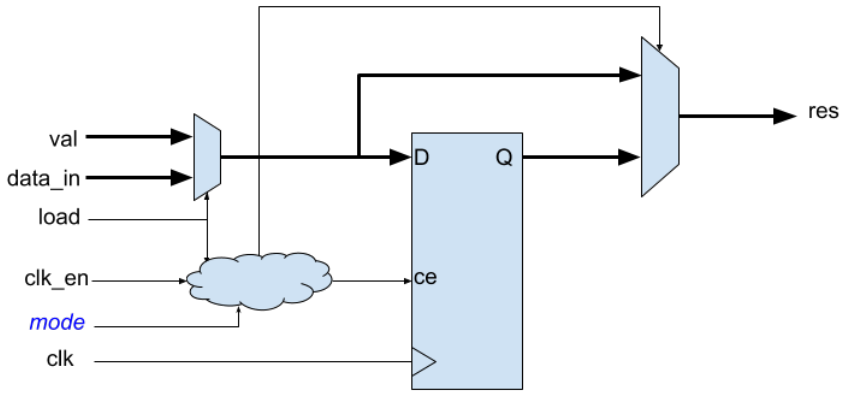

Figure 5.2: Storage element block diagram.

The storage element is connected to the general data routing (`data_in`, `clk_en`) and to configuration bus (`val, load`) . It can act as an extra pipeline stage, as previously discussed(see Section

3.2.3), alternatively it can function as a data delay element, source of constant value or bypassed. The constant is loaded once by the configuration bus before the application starts and then is used as an immediate value in the operation performed by PE. The functionality of this module is summarized in Tab. 5.2.

Mode	ce	res	Note
REG_CONST	0	val	Immediate value (Constant)
REG_BYPASS	na	data_in	Bypass
REG_VALID	clk_en	Q	Flip-Flop with clock en
REG_DELAY	1	Q	Delay by 1 cycle

Table 5.2: Storage element modes of operation.

Fig. 5.3 shows the block diagram of the resulting design for PE block. It has ALU compute core (see Fig. 5.1) with for 16 bit operations, 3:1 LUT for binary operation and optional input/output storage. The block has four 1 bit inputs: `d,e,f,clk_en` and two 16 bit inputs: `a,b` producing one 1 bit - `p` and one 16 bit - `res` results. The behavior is controlled by five signals: `sel_p, sel_d, sel_a, sel_b, op_code` which are grouped into one configuration word accessible through a config bus.

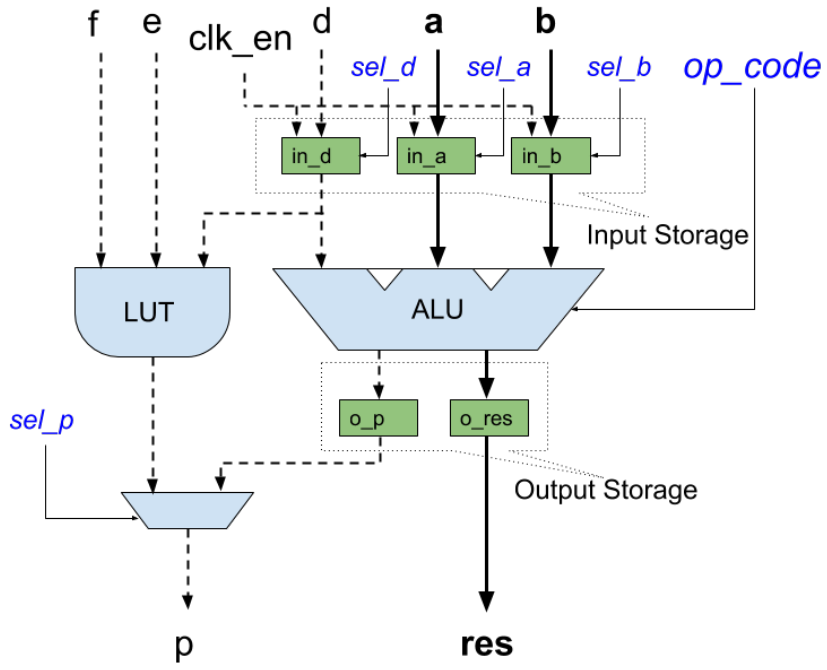

Figure 5.3: PE block diagram consisting of ALU, LUT and storage elements.

The area analysis of the PE block (Fig. 5.4) shows that support for the extra operations listed

in Tab. 5.1 is negligible - around 3% of the total

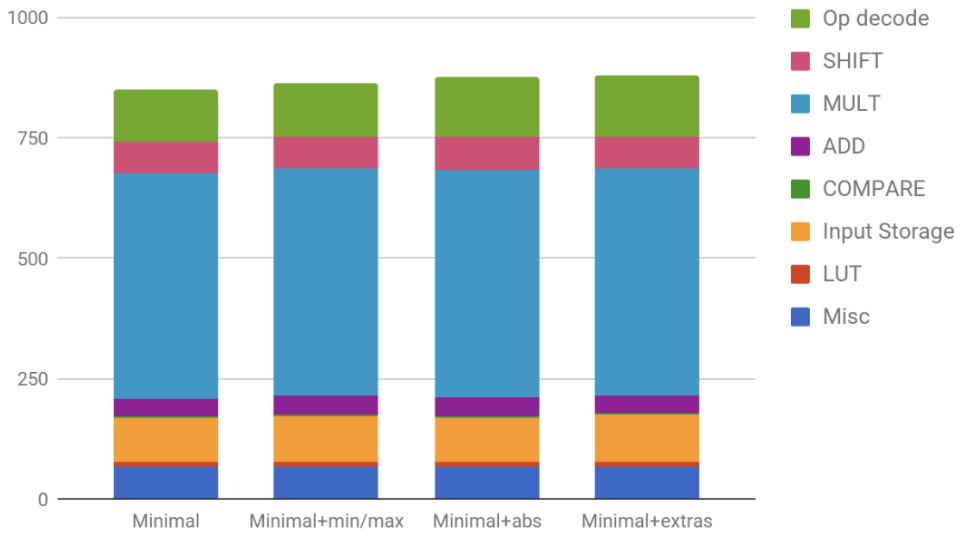

Figure 5.4: Simple 2:1 PE area breakdown.

It's worth noting that even that the area utilization of a PE is vastly different for various operations (see Tab. 5.3). The area shown in this table only includes parts of the PE that are involved when a certain operation is configured, including selected unit and a certain amount of overhead from various support elements like input storage, internal multiplexers, etc. This overhead is different for 1 bit and 16 bit operations. The first group is implemented by LUT, and only includes configuration word plus internal mux, while the second group included overhead of ALU plus the area of storage elements - 1 bit operation don't need those because constants can be directly embedded in LUT function. As a result the overhead for 1 bit operation is around 7.5% and around 30% for 16 bit buses, which leads to only 8.7% utilization in case of LUT and 37.7% in case of ADD.

LUT	**ADD/CMPR**	**MULT**	**SHIFT**	**LOGIC**
8.7	37.7	86.6	40.5	33.0

Table 5.3: PE area utilization by operation.

5.1.3 Tile with 2:1 PE

When analyzing PE design trade off, it's important to keep in mind the relative impact of changing PE's area on the total design. This impacts follows Amdahl law[5] according to the portion of the area occupied by PE. As previously discussed, the CGRA architecture consists of PE and routing/interconnect resources organized in an island style. The basic building block of the design is called a *tile* - it is repeated many time, occupying most of the chip. For the 2:1 PE a tile consists

of PE, SB and two CBs. A simplified diagram with only 16 bit bus routing elements is shown in Fig. 5.5. Besides bus routing, each tile also has 1bit routing resources which include a SB and one CB. Another "hidden" element omitted from the diagram is the storage required to hold the configuration.

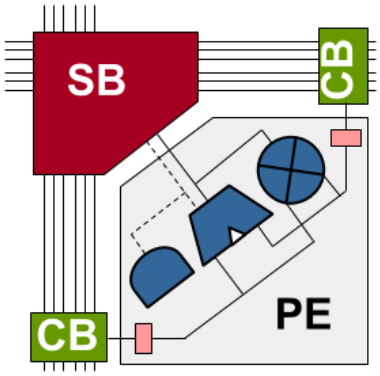

Figure 5.5: Tile diagram for the 2:1 PE based on island style organization.

The design of routing resources (SB, CB, number of tracks) is discussed in Chapter 3. Using those results to build a tile gives the area breakdown shown in Fig. 5.6. As we can see, multiplexers for flexible routing (16 bit SB, 1 bit SB and all CBs) take 68.7% of tile area while 2:1 PE is only 31.3%. This means we can explore more complex PE design in order to increase the compute density, thus reducing the number of tile required to implement a design. Making the PE more complex amortizes the high cost for general routing across several operation performed by the new PE. Any modest size increase over the current PE design will only cause a small overall area increase.

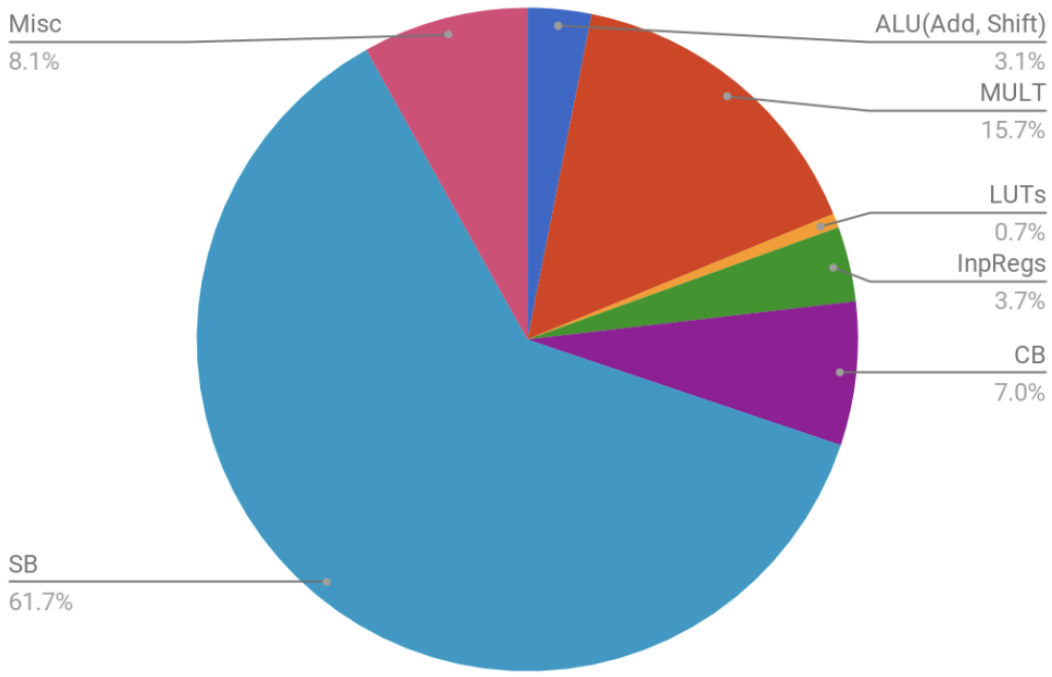

Figure 5.6: Tile area breakdown for 2:1 PE design. SB and CB dominate.

5.2 Increasing compute density 3:1

The area of a CGRA based on simple 2:1 PE design is dominated by the routing and since the functional unit is only performing a single operation, this leads to a rather small area efficiency. The area efficiency for an application is calculated as an average chip area required to perform one operation:

$$AreaEff = (TotalMemoryTilesArea + C * TileArea)/N$$

It includes total area of all line buffers ($TotalMemoryTilesArea$) and kernel implementations ($C*TileArea$). Here C - is the total number of compute tiles used to implement a program with N operations. This is a simplified formula - it assumes that any tile (compute or memory) which doesn't use its logic block is either used by some other application or doesn't exist. In reality, there could be tiles that are involved in routing or simply an overhead due to chip granularity - basically the tiles, which exist on a chip but are idle because the application needs slightly smaller resources than the the chip has. To account for these details an application has to be placed and route on a chip. Later (see Chapter 6), we'll be using a more precise method.

With the simple 2:1 PE $C \approx N$ and the portion of area per operation due to compute is simply the total area of the tile. One way to improve this metric is to reduce the number of tiles, C, by making

a more complicated PE capable of performing several operation, thus increasing the computational density. Of course, a more complex PE is likely to use more area as well, so we have to be careful and make sure the extra functionality we put in is actually use. Traditionally this is done by analyzing the target applications in order to find common patterns - a sequences of operations that often occur. One of the classic example is Fused Multiply Add[78](A*B+C) which is common in dot product computations, like convolution. Not surprisingly, we also find this sequence to be common in our benchmarks. Besides multiply-add we also find two additional patterns: Sum of Absolute Difference or SAD (|A-B|+C) which is commonly used as L1 norm for image registration and Add-Add or Sum of 3 (or more) elements which is often used in *reduction* stages of the applications. The complete list of new operations is given in Table 5.4.

Operation	**Semantic**	**Inputs**	**Result 16b**	**Result 1b (p)**	**HW name**
Modified Operations					
SUM	a+b+c+d	a,b - int16 or u_int16 d - 1 bit	res = a + b + c + d	(a+b)>=2^16	ADD
Subtraction-Addition	a-b+c	a,b - int16 or u_int16	res = a + ∼b + c + 1	N/A	SUB
Multiply	a*b+c	a,b - int16 or u_int16	(a*b+c)[15:0]	N/A	MULT_0
Multiply middle	(a*b+c) >> 8	a,b - int16 or u_int16	(a*b+c)[23:8]	N/A	MULT_1
Multiply high	(a*b+c) >> 16	a,b - int16 or u_int16	(a*b+c)[31:16]	N/A	MULT_2
Add Absolute difference	|a-b|+c	a - u_int16	((a-b)<0) ? (b-a+c) : (a-b+c)	sign(a-b)	ABS
New Functionality					
Counter	cnt[t+1] = cnt[t] + inc	a - inc b - cnt[t] c - tc_val clk_en	res = cnt[t] (b) b_next = tc ? 0 : (b + a)	tc = (b == tc_val)	CNTR_OP

Table 5.4: Modified list of operations for 3:1 PE.

Since we use the same `ADDER` block for addition and subtraction, beside A+B+C we also got A-B+C "for free". It should be noted that all the extra pattern use 3 inputs and 1 output (just like FMA) and this extra input requires an additional CB in the tile (Fig. 5.7) and that the new *patterns* are a superset of previous instructions: ADD, SUB, MULT and ABS. In fact we can still get the same functionality as 2:1 design by setting the `c` input to constant 0.

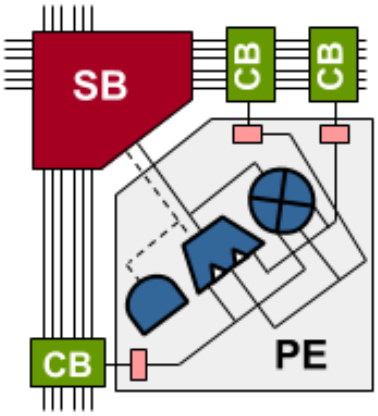

Figure 5.7: Tile diagram for the 3:1 PE block with on more CB.

With the addition of the third input, we have also added another input register and can now store 3 immediate values. This extra storage enables us to add new functionality, CNTR_OP, and allows PE to act as counter. In this mode, the value on the output is incremented by a certain amount (`inc`) on every clock cycle until it reaches terminal count value (`tc_val`) at that point the counter is reset to 0 and terminal count (`tc`) signal is sent. Note that this requires three storage registers: `cntr, inc, tc_val`. One of these registers (`cntr`) has to be updated every cycle the counter is enabled. Each input storage block already has an ability to load the new "constant" from the configuration bus, we are just reusing this mechanism to store the next value of the counter. We can also think of it as pipeline register that stores a new value every cycle.

Normally, a counter operation would require a feedback connection on the general routing network, thus breaking our DAG assumption about the structure of compute kernels. Having a counter implemented as atomic instruction avoids this problem because the feedback connection is now hidden in the element and from the compute kernel point of view it looks just like any usual operation, like addition.

The main use of the counter operation is to enable kernels to keep track of the position within image - X, Y coordinates of the current pixel. It also can be used as an "accumulator" for histograms or some other statistics, like in auto focus/auto exposure/auto white balance (AAA) algorithms which every digital camera runs.

Taking the original 2:1 design as a base line, the area of the 3:1 PE has increased by 18%, out of which 10% were due to MAD unit which now includes adder and extra input storage associated with additional input. The counter support didn't cause any noticeable area increase. The area breakdown for this configuration is shown in Fig. 5.8.

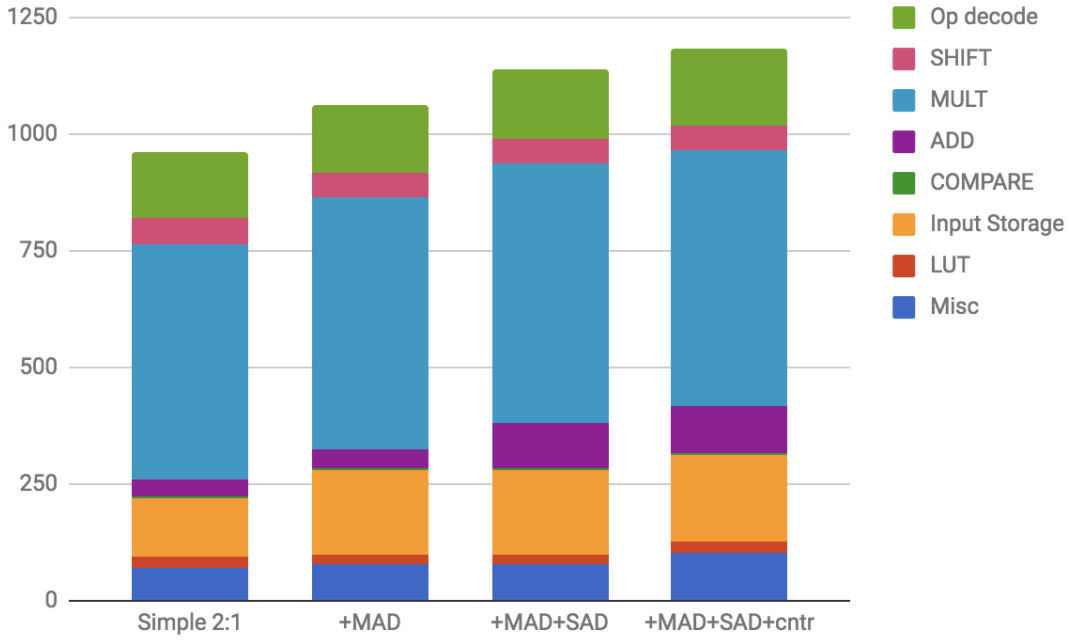

Figure 5.8: 3:1 PE area breakdown.

Since the baseline PE was only 26.7% of the entire tile area, the 18% gain from using new 3:1 PE, leads to just 6% overall area increase. The benefit is that some dependent operations now can be done in one tile. For example, image registration (SAD) using 8x8 patch used to take 191 tiles: 8*8 subtracts + 8*8 abs + (8*8-1) additions vs 64 tiles with the new design: 8*8 SAD(abs), or in case of 5x5 convolution it was 49 tiles: 5*5mult + (5*5-1) additions vs 25 tiles: 5*5 MAD. Of cause not every application will see such a dramatic reduction, but overall benefit in tile count reduction far outweighs 6% area increase. The new tile area breakdown (Fig. 5.9) shows that interconnect is now taking 64.9% and PE is 35.1%.

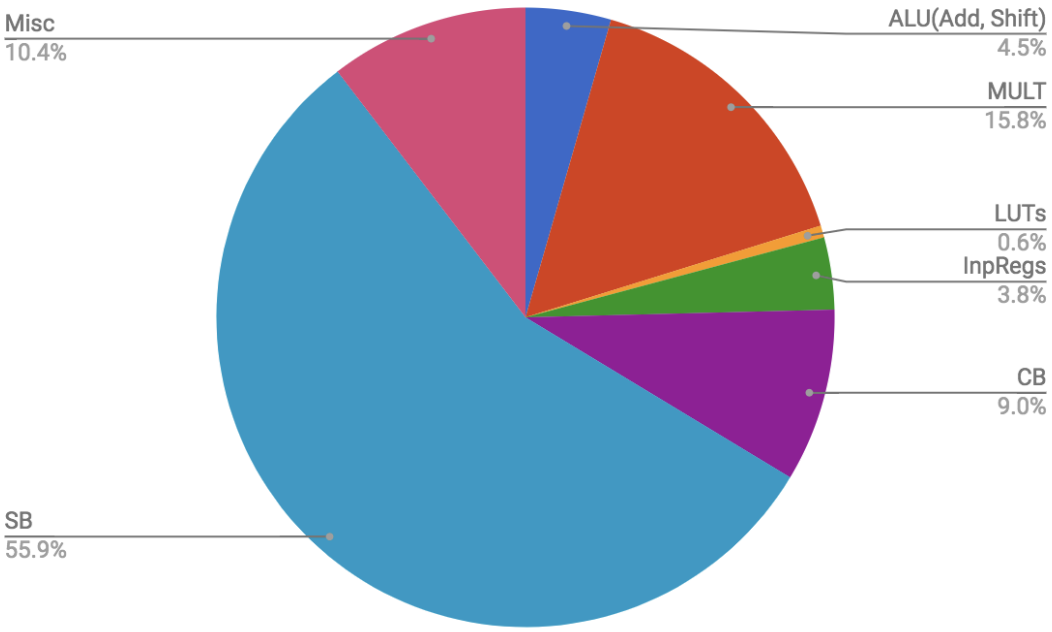

Figure 5.9: Tile area breakdown for 3:1 PE design. SB and CB still dominate.

5.3 Double precision support

We built the PE to perform the most common 16 bit operations, but our applications occasionally use more complicated 32 bit operations and division. Support for such operations increase the precision and range of computation and broaden the number of applications that can be run on or CGRA.

Higher precision operations can already be done using a series of existing 16 bit instructions, but such implementations are fairly inefficient due to more complicated control and the need to add extra steps in order to access the required data bits, for instance, in case of 32 bit shift. Instead we can add some extra logic to the PE design in order efficiently map these complex operations to existing hardware and minimize the number of tiles required.

Our goal is to expand ISA to cover 32 bit operations and use the same definitions as in Tab. 5.1 and Tab. 5.4, except change the width of input and ports: int16 $->$ int32 and u_int16 $->$ u_int32. Since most of the operations in imaging applications are 16 bit or less, we want to reuse existing compute resources instead of adding new blocks, however we do need to add some extra logic.

The hardware complexity of most arithmetic and logic operations scale proportionally to number of bit, expect for multiplication that scales quadraticaly. Division is another special case that requires a long chain of simple operations or a dedicated unit. Next we'll describe how each type of operations is implemented in our generator and will present area cost analysis for these features.

5.3.1 Operations with linear scaling

Most operations (add, sub, compare, shift, logic) scale linearly in complexity with the increase of data width. For them we can combine two of existing PE and view it as one double precision PE - see Fig. 5.10.

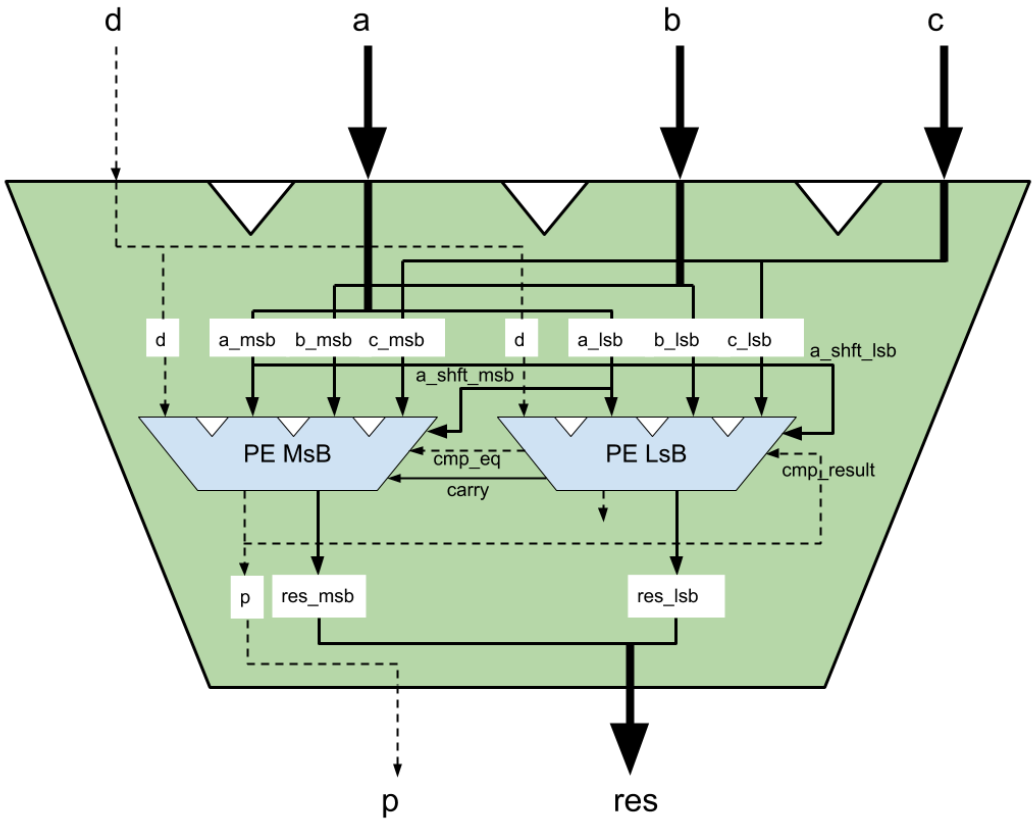

Figure 5.10: Combining two PEs for 32 bit precision operations (except MULT) can be viewed as one PE supporting 32 bit.

Notice a few extra signals, which we had to add: `carry` - is the carry signal from the adder of lsb portion into the adder of msb portion. It is used in any operation that needs 32 bit adder (addition, subtraction, abs/sad, compare) `cmp_eq` - is used in equality checks, it notifies PE_MsB that lsb portions matched `cmpr_result` - use in comparison operations, it notifies PE_LsB of the comparison result `a_shift_msb`, `a_shift_lsb` - are use for shifts, because an PE_MsB needs access to lower bits of the operand for left shift operation and PE_LsB needs access to upper bits of the operand for right shift. These signals have a special purpose in implementation of various double precision operations. They are not reusable and are part of the critical path during these operation, so such signals are connected directly between two neighboring PE blocks and don't use flexible

routing resources (SB, CB, tracks).

Adder based operation (add, sub, compare) use two 16 bit adder blocks in PE MSB and PE LSB which are connected with `carry` signal to compute double precision result. In case of compare (GTE_MAX, LTE_MIN), the final result (1 bit - `p` signal) is calculated in PE_MsB and we pass it to PE_LsB to select the correct lower bits of MAXMIN operation, which is performed simultaneously. For EQ operation, PE MSB needs the comparison result of lower bits which computed in PE LsB and passed on `cmp_eq` signal.

Logic operations don't require any additional signals. For them, each bit in results depends only on corresponding bits in the operands.

Shift support requires the following operations in PE_MsB and PE_LsB respectively:

```
left_shift_MsB  = {a_msb, a_lsb} << b
right_shift_Msb = a_msb >> b

left_shift_LsB  = a_lsb << b
right_shift_Lsb =  {a_msb, a_lsb} >> b
```

Notice that this "naive" implementation use one 32 bit shifter and one 16 bit shifter. We can do better by using the same "reverse" trick as in the simple PE case:

```
left_shift_MsB  = {a_msb, a_lsb} << b
right_shift_MsB = reverse(reverse({0, a_msb}) << b)

left_shift_LsB  = reverse(reverse({0, a_lsb}) >> b)
right_shift_Lsb =  {a_msb, a_lsb} >> b
```

which requires just one 32 bit shifter. Alternatively, we can do:

```
left_shift_MsB  = (b < 16) ? (a_msb << b) | (a_lsb >> (16-b)):
                             a_lsb << (b-16)
right_shift_Msb = a_msb >> b

left_shift_LsB  = a_lsb << b
right_shift_Lsb = (b < 16) ? (a_lsb >> b) | (a_msb << (16-b)):
                             a_msb >> (b-16)
```

This method uses two 16 bit shifters and some OR gates. The last two approaches give about the same area result and are better than the "naive" method.

5.3.2 Support for Division and Modulo

To support double precision operations we combined resources of the two PEs into one larger block, now we can use this block to better support `division` and `modulo` operations. There are several

ways to implement `Y/X`:

- approximate with right shift:

  $Y/X = Y >> log2(round\_base(X,\ 2)) = Y >> 16 - cnt\_lead\_zero(X))$

- Direct hardware, for example by using Newton-Raphson or with multiply by reciprocal which is taken from lookup table(LUT):

  $Y/X = Y * (1/X) = Y * LUT(X)$

- Iterative methods similar to long division

The first method is an approximation and doesn't give the exact number, the second requires large (proportional to N^2, where N is number of bits) memory for LUT or dedicated hardware unit which could be expensive, so we focus on the iterative approach which can be expressed though simple math for which already have hardware in PE. In particular let's consider restoring division algorithm[115]:

```
P := N
D := D << n             // P and D need twice the word width of N and Q
for i := n - 1..0 do    // For example 31..0 for 32 bits
  P := 2 * P - D        // Trial subtraction from shifted value
  if P >= 0 then
    q(i) := 1           // Result-bit 1
  else
    q(i) := 0           // Result-bit 0
    P := P + D          // New partial remainder is (restored) shifted
                        // value
  end
end

// Where: N = Numerator, D = Denominator, n = #bits,
//        P = Partial remainder, q(i) = bit #i of quotient
```

It takes N=16 steps resolving one bit of the result per step and in the end yields both the result and remainder, thus performing Division and Modulo operations simultaneously. The core of the algorithm can be rewritten as:

```
Diff = 2*P - D
if Diff > 0 then
  q(i) = 1
```

```
  P = Diff
Else
  q(i) = 0
  P = 2*P
end
```

From this we can see that each step of the iteration requires 3 inputs: P, Q, D and produce 2 outputs: P, Q. Computationally it performs 1 shift (for *2), 1 subtract, 1 compare , 1 multiplex and 1 set bit operations. Using 3:1 PE computations of step each would require 7 tiles if set bit is implemented by shift, select, or:

```
two_p = P << 1
Diff  = two_p - D
sel   = Diff > 0
P     = sel ? Diff : two\_p
Q     = Q << 1
mask  = sel ? 1 : 0
Q     = Q | mask
```

One of the limiting factor is the single output of a 2:1 PE. The double precision PE (Fig. 5.10) doesn't have this limitation, with total 4 inputs and 2 outputs there is enough bandwidth for a division. If fact we can do better by taking advantage the following properties of the chosen algorithm. First, the number of inputs and outputs doesn't depend on the number of steps performed - it is always 3 and two. Shift right by a fixed amount is very cheap to implement in hardware, so is setting a bit to 1 or 0 in fact the most expensive operation at each iteration is subtract. In the double precision PE (Fig. 5.10), we have two ALUs and can do two subtractions, thus with some extra logic and connections between PE MsB and PE LsB we can perform two steps of iteration and division (or modulo) implementation would take 16 tiles vs 7*16=112 tiles. The new signals and important connections are shown on Fig. 5.11

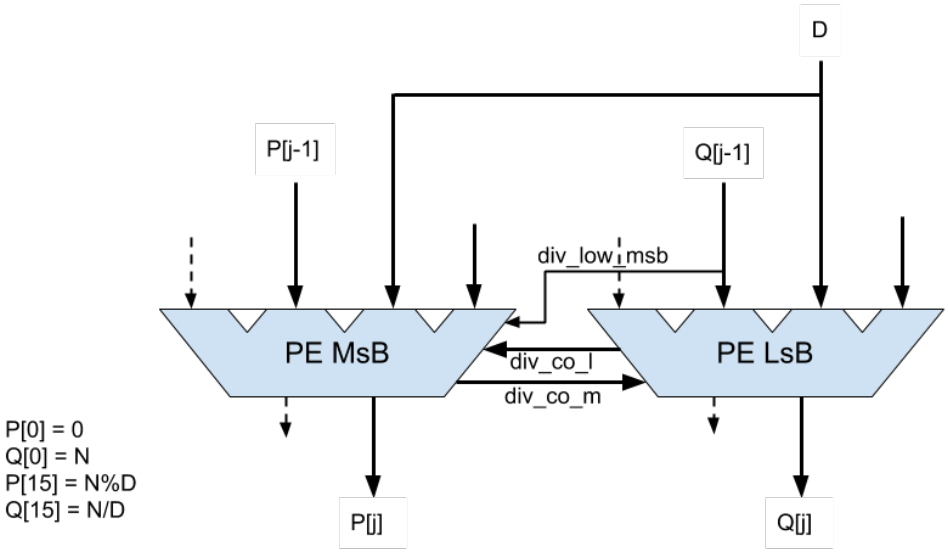

Figure 5.11: Implementing two steps of iterative division using two adjacent PEs with a few dedicated connections.

5.3.3 Multiply

Multiplication is different than all other operations because its complexity scales quadraticaly with the number of bits. Thus to support double precision (32 bit) operation we need four blocks with single (16 bit) precision multipliers. We can derive this implementation using basic arithmetic rules:

$$\{a1,a0\}*\{b1,b0\}+\{c1,c0\} = a1b1<<32+((a1b0)+(a0b1+c1))<<16+(a0b0+c0)$$

Here, `a0,a1,b0,b1,c0,c1` are 16 bit numbers and `{a1,a0},{b1,b0},{c1,c0}` are 32 bit numbers composed of two 16 bit wide fields. Notice that this equation has four terms that match operations we have already defined in the PE: MULT and MULT-ADD, the only difference is that we have to keep all 32bit of results which latter will be summed to give the final answer. This final addition has to be done on 64bit numbers to support **signed** multiplication, but we can take advantage of the fact that first term (`a1*b1<<32`) has 32 lower bits set to zero, the other two terms (`a1b0<<16` and `(a0b1+c1)<<16`) would have the upper 16 bits set to the sign and the lower 16bit set to zero ,the last term (`a0b0+c0`) we only need 16 bits, because it always unsigned value whose lower 16 bits go to the result unmodified. Thus this addition of four 64bit numbers simplifies to one 32 bit addition of 4 numbers and one 16 bit addition of 3 numbers:

$$\begin{aligned}Result = {} & \{P0,32'b0\} + \{\{16\{sign(P1)\}, P1, 16'b0\} + \\ & \{\{16\{sign(P2)\}, P2, 16'b0\} + P3\end{aligned}$$

$$P0 = a1*b1$$

$$P1 = a1*b0$$

```
P2 = a0*b1+c1
P3 = a0*b0+c0

Result = {high[15:0] , mid[31:0] , P3[15:0]}
mid[33:0] = {P0[15:0], 16'b0} + P1 + P2 + {16'b0, P3[15:0]}
high = P0[31:16] + const(sign(P1), sign(P2)) + mid[33:32]
const(sign(P1), sign(P2)) = 16'b0, if P1>0 & P2>0
                          = FFFF, if P1<0 & P2>0 or P1>0 & P2<0
                          = FFFE, if P1<0 & P2<0
```

We can map this operation to four of the compute tiles (PEs) if we add some extra logic to implement the final addition - see Fig. 5.12.

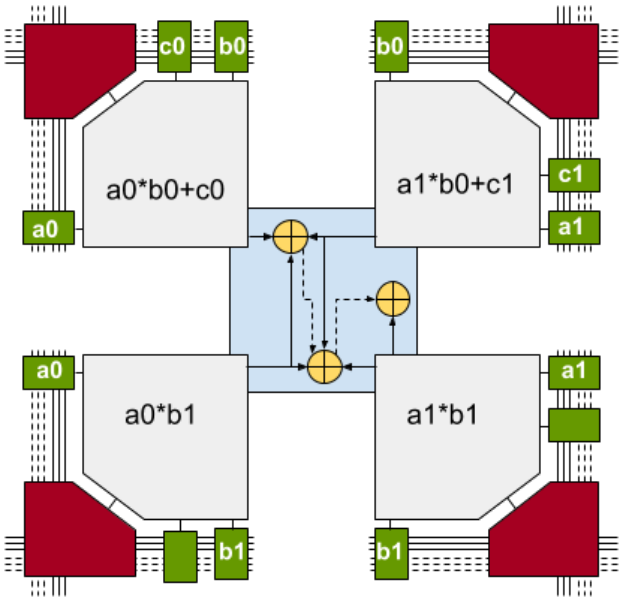

Figure 5.12: 32 bit precision multiplication using four PE tiles require extra block for the final addition.

Note that four tiles have 12 inputs and 4 outputs, while 32 bit MULT-ADD need 6 inputs and 2 outputs for a 32 bit result or 4 outputs if we want to preserve 64 bit precision (64 bit operations are typically not required by ISP implementations). Here each in/output is 16 bit. This allows us to implement a single multiplication operation which produces full 64 bit result, rather than have multiple versions which produce upper/middle/lower bits as in single precision case. We can still get the behavior of MULT_HI, MULT_MID and MULT_LO for 32 bit operation, by taking advantage of the fact that 64 bit result is on four output busses which can be independently routed, so we can simply route just the words we need.

5.3.4 Area cost analysis

In order to fairly evaluate various design features described in this chapter, we have to compare groups of four PEs, rather than individual blocks. The reason is that for design with support of double precision multiplication, the group of four PEs, including the partial product summation, or "reduction" is the smallest pattern that is replicated throughout the chip.

When working with larger blocks, the synthesis tools have more opportunities to "blend" blocks into each other and produces smaller area, so here we reimplemented the previous results for 2:1 PE and 3:1 PE replicated 4 times. Taking the area of simple 2:1 design as a base, the relative cost of various features is given in Tab. 5.5:

2:1	**3:1**	**+double**	**+div**	**+mult32**
1	1.218	1.259	1.313	1.514

Table 5.5: Relative area cost for quad PEs.

The results show that support for 32 bit(+double) precision of linearly scalable operations (add, subtract, compare, shift, logic) and iterative division is relatively cheap 4% and 5% respectively. Support for 32bit precision multiplication was the most expensive feature that costed 20%. The overall area increase due to 32 bit operations was 30% of the base design.

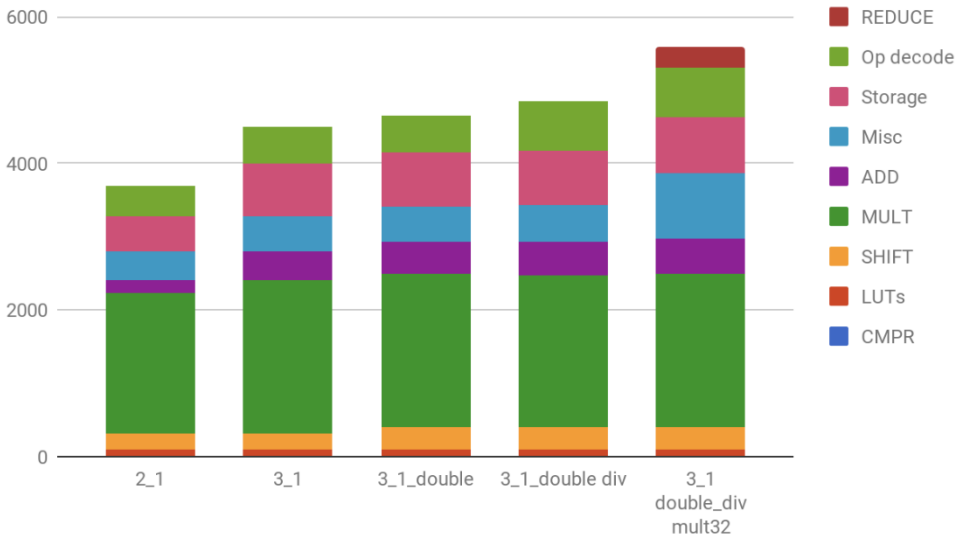

Figure 5.13: Area breakdown for quad PEs. 32 bit multiplication is the most expensive feature.

From the area breakdown (Fig. 5.13) we see that the majority of the PE area (44.7%) is still taken by the multiplier and that the most the area increase for double precision support of linearly scalable operations was due to the SHIFT block.

The last point is not surprising - as we discussed, both PE_LsB and PE_MsB have to perform two shifts (on upper and lower parts of the operand). Given that shifts are fairly rare compared to the other operations and that there are no applications which are dominated by them (unlike was the case with multiplication and filter/convolution, CNN applications) we could make heterogeneous blocks where double precision shifts (or any shifts) are supported only by some PE. This would add restrictions and complexity to placement tools and only save at most 3% of the area of the PE.

Finally, there is a significant increase of "misc" category for double precision multiplication. It is caused by multiplexing of the partial result which also was blended with parts of the final summation (aka REDUCE) logic. The biggest change of this design is the fact that most other output signals depends on all four PEs which makes timing harder and eventually leads to area increase.

5.4 Clustering and special patterns

In the previous section, to support double precision multiplication we had to add a block to sum the partial results. We can expand the functionality of this block by adding other operations using the same adder, such that it performs reduction function on the results from four PEs.

Most common reduction operation is addition (ADD) which appears in such important patterns as filter/convolution and image registration/sum of absolute differences. Previously, we added a support for this patterns to a PE, however such approach rely on the application developer to make sure that the result stays within 16bit of range and doesn't overflow. This is a rather big assumption because typically there are a lot of numbers that are summed or "reduced" into one higher precision number which is then analyzed or cast down to original 16 bit precision. Performing such "cast down" after every addition, can lead to less accurate results. This problem is avoided when we perform summation in the "reduction" block we added to every group of four PEs. This blocks operates on 32 bit precision numbers and it has access to four 16 bit output buses, rather than one. The resulting structure is a cluster of 4+1=5 processing elements (Fig. 5.14).

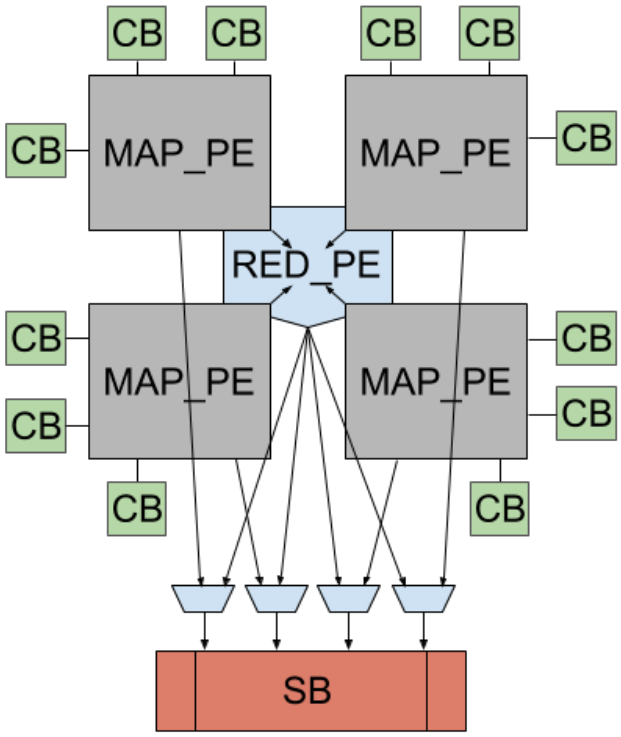

Figure 5.14: Cluster design using 3:1 PE and changing final added for 32 bit multiplication into reduction PE.

The PEs in the cluster are heterogeneous: 4 *map* PEs, which support all operations from Tab. 5.1 & Tab. 5.4 and the fifth block is *reduction* PE. The reduction PE is not connected to the input ports(CBs) directly, only through the map PEs and can do only a limited set of "reduction" operations (see Tab. 5.6) and a special operation to produce the final result of multiplication(MULT_PP_SUM).

Operation	**Semantic**	**Inputs**	**Result**
SUM	a+b+c+d	a,b,c, d - int16 or u_int16	32 bit
ABS_SUM	|a|+|b|+|c|+|d|	a,b,c, d - int16	32 bit
MIN	MIN(a,b,c,d)	a,b,c, d - int16 or u_int16	16 bit
MAX	MAX(a,b,c,d)	a,b,c, d - int16 or u_int16	16 bit
MULT_PP_SUM (unsigned)	{a,16'0} + {16'0,b} + {16'0,c} + {32'0,d}	a,b,c, d - 32 bit	48 bit
MULT_PP_SUM (signed)	{a,16'0} + {16'sign(b),b} + {16'sign(c),c} + {32'0,d}	a,b,c, d - 32 bit	48 bit

Table 5.6: Reduction operations implemented by reduce PE.

The PE cluster on Fig. 5.14 is fully compatible with all the operation supported by 3:1 design

described earlier (see Section 5.2). We can treat this cluster simply as four independent neighboring PEs. The advantage of the new structure (besides 32 bit multiplication support) is that it can do "accumulation" patterns (MAD, SAD operations) in 32 bit precision. Besides the increased precision, the reduction PE allows us to amortize the "accumulation" logic we have added to each PE for MULT-ADD, SAD and ADD-ADD operations. Now we can revert to using 2:1 configuration for map PEs without losing functionality. Fused addition operation (used in SAD and MULT-ADD) is usually part of a "reduction" over larger domain and typically they form a reduction chain which we turn into a reduction tree (See Fig. 5.15). To support the reduction domains larger than four points, we can added an additional 32 bit (dual ports) `chaining` input which goes only into reduction. This allows the compiler to build large reduction chains, just like before, except each node now is a small cluster that reduces 4 pairs of values instead of 1.

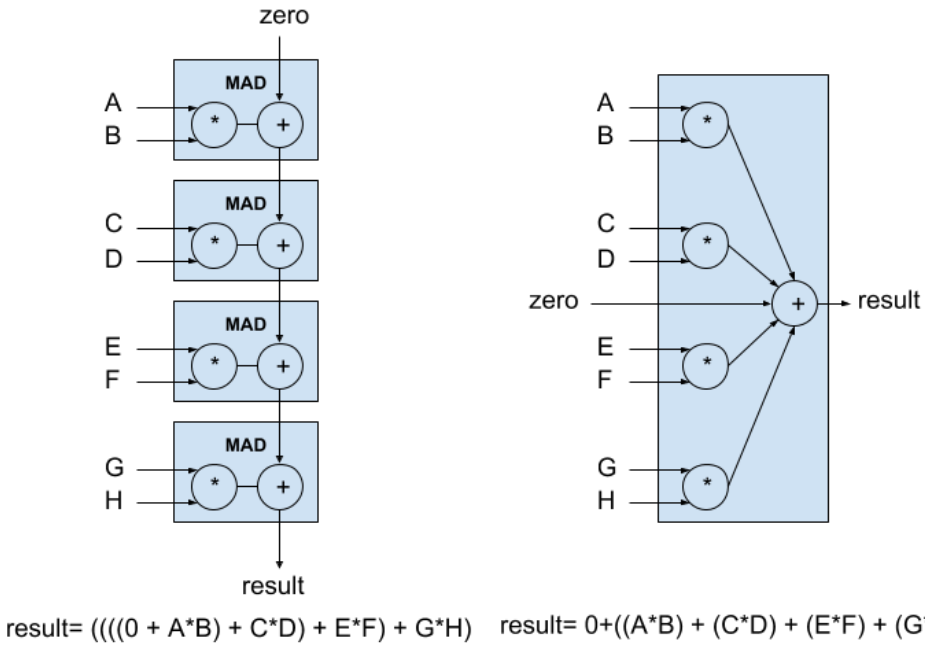

Figure 5.15: Reduction pattern implementations using 3:1 tiles (chain) vs quad tile (tree).

Compared to the original structure (Fig. 5.14), the optimized cluster(Fig. 5.16) with quad 2:1 PE design has a smaller area of the multiplier (compared to mult-add) and adder units. Another benefit is the reduction of I/O ports: 2*4+2=10 inputs vs 3*4 = 12. The downside is that now we can do 1 distinct reduction per group of four PEs instead of 4. This is usually not a problem because there are a lot PE on the chip and because the number of distinct reductions is much smaller than the number of operations in the kernel. Even in the kernels with a lot of additions (for example digital filters, sum-of-absolute-difference) those adds from just a few large reduction trees or chains,

thus the number of distinct reductions is still very small.

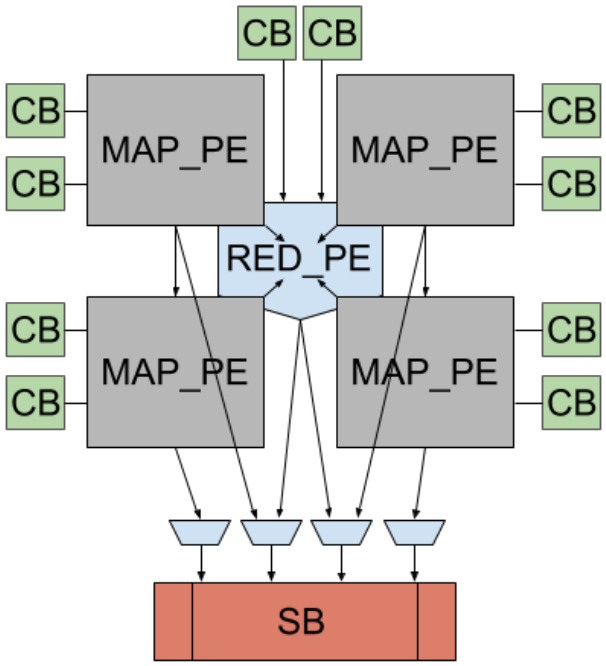

Figure 5.16: Optimized cluster design uses 2:1 PE and extra inputs to chain clusters rather than individual PEs.

Looking at the area breakdown for the PE (Fig. 5.17), we see that turning the final summation of the 32 bit multiplication into a Reduction PE was quite cheap 7% of the base design. Just as we expected, optimized cluster design using 2:1 map PEs noticeably reduced the area (by 20%) due to savings in the multiplier, adders and storage. Support of chaining in the reduction PE increased the area insignificantly - by 5% of the original design.

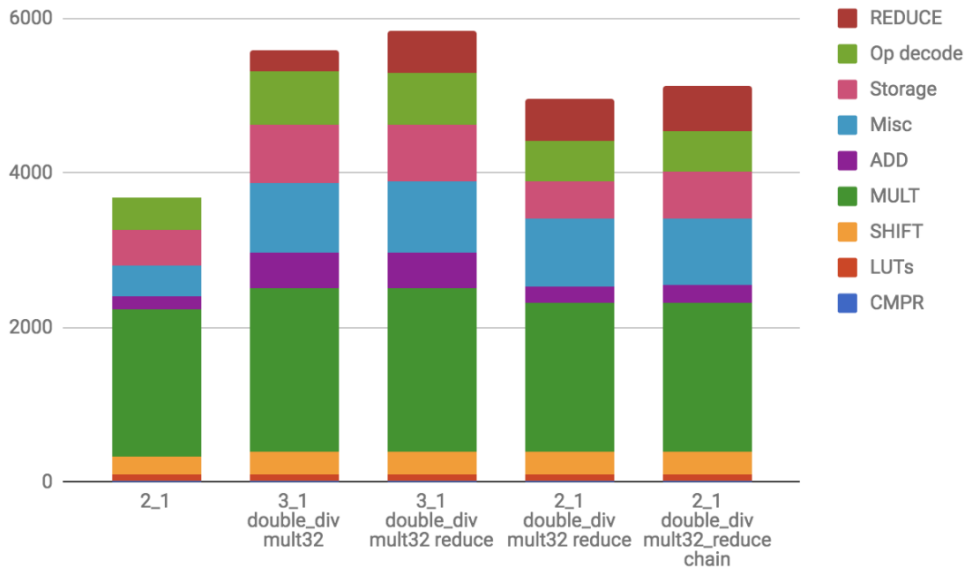

Figure 5.17: Area breakdown for quad PEs in different cluster configurations.

We have added a number of features to the original 2:1 PE making the design significantly more complex and increasing the area by almost 40% even after optimizing the cluster. This might seem like a large number, but it doesn't include the routing area. Because the tile area is dominated by the routing resources (see 2:1 and 3:1 tile analysis), when SB & CBs are factored in the overhead goes down significantly to only 14% (see Tab. 5.7).

Cluster configuration	PE area	Tile area
2_1	1.000	1.000
3_1	1.218	1.128
2_1 double div	1.145	1.068
3_1 double div	1.313	1.185
2_1 double div mult32	1.353	1.138
3_1 double div mult32	1.514	1.243
3_1 double div mult32 reduce	1.581	1.260
2_1 double div mult32 reduce	1.343	1.101
2_1 double div mult32 reduce chain	1.391	1.141

Table 5.7: Relative increase in PE and tile area depending on cluster configurations.

5.4.1 Efficiency considerations of clustering and special patterns

The main argument in favor of the cluster design is the same as for the 3:1 PE - to increase the compute density by adding support for the special patterns like: sum of absolute differences (SAD),

dot product and reduction with the summation:

$$SAD : S + \sum_{i=0}^{3} ABS(a[i] - b[i]) = S + ((ABS(a[0] - b[0]) + ABS(a[1] - b[1])) + (ABS(a[2] - b[2]) + ABS(a[3] - b[3])))$$

$$DOT : S + \sum_{i=0}^{3} a[i] * b[i] = S + ((a[0] * b[0] + a[1] * b[1]) + (a[2] * b[2] + a[3] * b[3]))$$

$$SUM : S + \sum_{i=0}^{3} a[i] + b[i] = S + (((a[0] + b[0]) + (a[1] + b[1])) + ((a[2] + b[2]) + (a[3] + b[3])))$$

With the basic 2:1 design, SAD requires 12 tiles, DOT and SUM - 8 tiles. All these operations map to a single cluster with 4 `map` tiles, so those 14% extra area can reduce the total die size by almost 2x or 3x. Compared to 3:1 design, area advantage of clustering is more modest: 1.243/1.141 = 1.089(see Tab. 5.7) or 9% since there is no reduction in the number of PE tiles and all three patterns require 4 tiles. Here we ignore the benefit of being able to do reduction with 32 bit precision in case of clustering. There is also a noticeable savings in energy.

To estimate the energy, we assumed that for all configurations tiles are placed to minimize the length of wires and that all the connections are done using just one wire track, even if the length is more than 1. We also assumed that all the inputs (a,b,S) originate directly at the tile that needs them, while in real applications this signals will be routed from I/O or other tiles that produced them so there will be some additional energy cost that depends on implementation of the program. For example, DOT pattern implementation for 2:1, 31: and cluster configurations is shown on Fig. 5.18, Fig. 5.19 and Fig. 5.20 respectively. Note, that all the wires have length=1, expect a single length=2 wire in case of 2:1. We assume that a pipeline register is enabled on a wire roughly after two operations in a chain (depending on the kind of operation and routing). Finally, we added energy of all elements involved in implementation using methodology from Chapter 6 to get the energy. For instance, in case of a DOT pattern and cluster configuration, this calculation is:

$$E = 4 * PE(MULT) + PE\_RED(ADD\_4) + 9 * CB + SB + 4 * wire + register$$

In this calculation we explicitly include 4*wire term to highlight the implementation of compute patterns in a cluster relies only on dedicated internal connections and bypasses the general routing resources like SB, CB and wire segments which are required in 2:1 and 3:1 configurations.

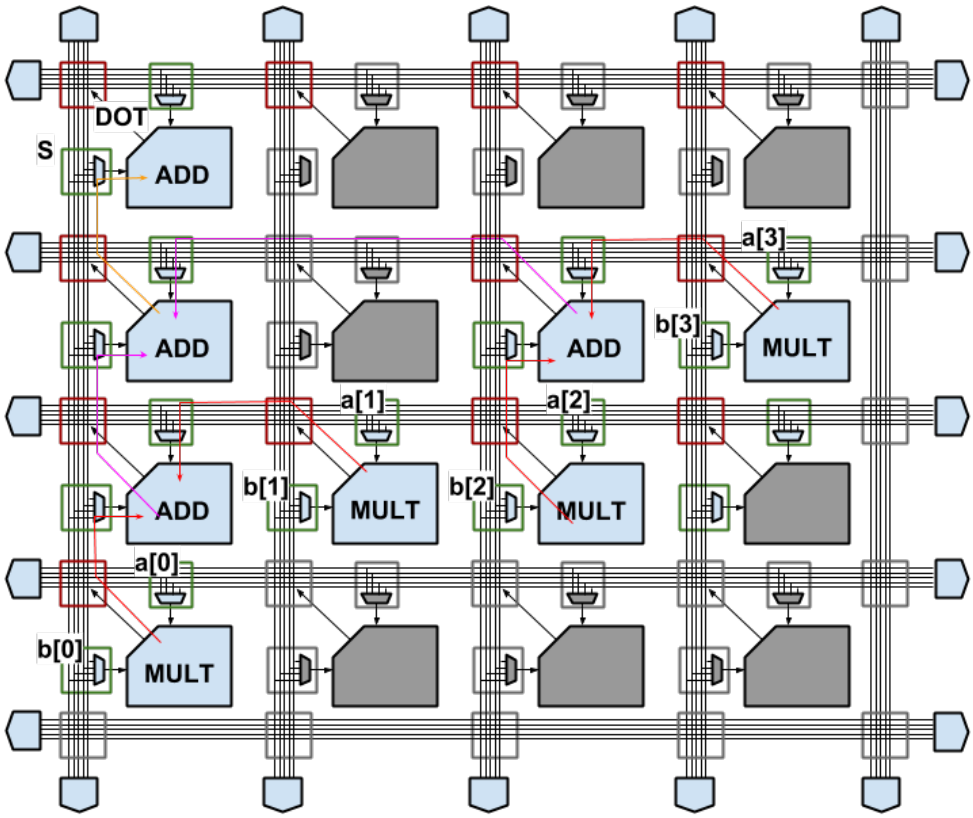

Figure 5.18: Implementation of DOT pattern with 2:1 configuration.

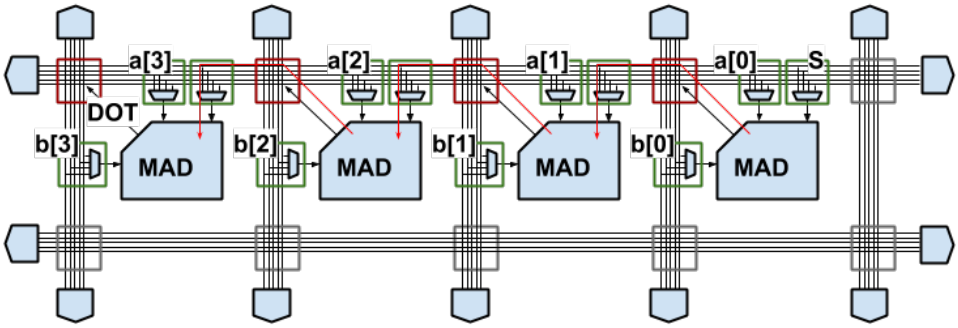

Figure 5.19: Implementation of DOT pattern with 3:1 configuration.

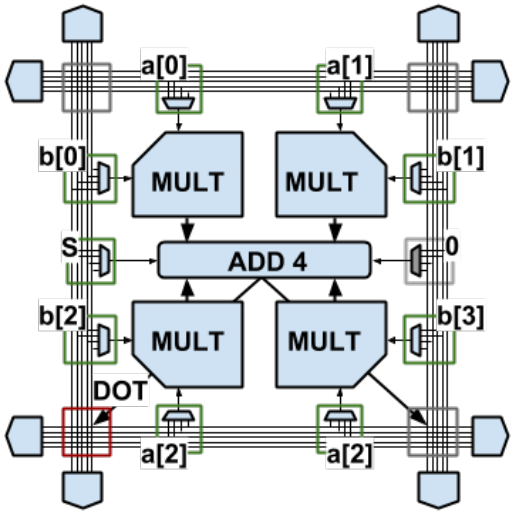

Figure 5.20: Implementation of DOT pattern with cluster configuration.

The results for area and energy improvements in case of DOT, SAD and SUM special pattern are given in Tab. 5.8. As we can see the energy improvement for SAD and SUM is roughly proportional to the decrease in the number of tiles. This is not surprising because these patterns use only cheap operations, each similar to the cost of an ADD and thus the energy is dominated by the cost of PE overhead and routing resource. In case of DOT product, the cost of MULT operation is more significant and it is constant (all configuration have to do 4 multiplications) so the relative improvement is smaller. We can also see that cluster are better than fused operations (3:1), although not by much: 9% in area and 15% in energy. The energy savings are large because 3:1 configuration does reduction in a `chain` which leads to a longer path with more elements involved and more pipeline registers compared to a "tree" style reduction in performed in a clusters(see Fig. 5.15).

	DOT	**SAD**	**SUM**
Area ratio			
3:1	1.61	2.41	1.61
Cluster	1.75	2.63	1.75
Energy ratio			
3:1	1.22	2.64	1.84
Cluster	1.45	3.09	2.04

Table 5.8: Area and energy improvement of compute patterns using 3:1 PE and cluster design over 2:1 PE.

To take advantage of special patterns and clustering, they have to be supported by the development tools. Implementation of each pattern results in a specific sequence of hardware elements which

have to be placed in a certain way. In hardware development these are often called macro. Place & Route tool has to be aware of the macros and has to treat them in a special way which means that macros have to be extracted by the end of the mapping step. This can be done by modifying either the technology mapper or the DSL compiler (see Section 3.3 Fig. 18). The first option would require isomorphic subgraph matching which is a complicated problem and second option would require a unique backend to make compiler aware of the macros supported by the target architecture. For our estimates, we modified implemented clustering during "CGRA Technology Mapping" phase (see Section 3.3) using simple greedy algorithm and breadth-first search through the kernels' DAG. This simple algorithm occasionally miss clustering opportunities which would've been recognized by a more sophisticated approach.

It should be noted that the support of special patterns through clustering is easier to implement with programming in space architectures. Since `REDUCE PE` requires 5 inputs, with programming in time we would have to modify the instructions to include addresses for the extra ports to the register file or add an additional slot in VLIW if we want to compute pattern in a single cycle. This would increase the energy of instruction fetch/decode which would require additional mitigation. In spatially programmable architecture, this inputs are cheap because they are implemented using direct connections to map PEs and there is NO increase in energy cost because configuration word (analog to instructions) doesn't change during execution.

Finally, the CGRA cluster is also quite different form LUT cluster in FPGA. In FPGA all logic elements in a cluster are identical and internal connections are flexible. Here some LUT inputs can optionally be connected to other LUTs outputs, and all the LUTs always can the connected to global routing. This can be explained by the desire to use all the LUTs in the cluster. Our CGRA cluster has two kinds of PE elements (Map and Reduce) and internal connections between them are fixed. We choose this approach because reduction PE has a limited functionality and is much smaller than map PEs. The mapper tool can always try to put the operation which produce values to be reduced in the same cluster and those values will go in the reduction PE.

5.5 Summary

To support a flexible compute resource in CGRA, we use Processing Element with configurable ALU - similar to the one used in programming in time architecture. These ALUs can implement any operation from the limit instruction set (ISA).

Using a hardware generator, we explored the cost of various additional instructions and features like increased compute density with 3 input PE design, support for divisions and 32 bit arithmetic. By carefully choosing the microarchitecture of the PE, we show that in some cases, a hardware required to support this features can be reused to implement even more complicated operations - like reduction, dot product and sum of absolute differences. These special operations provide significant area and energy saving for common patterns that can be found applications. They are

similar in functionality to dedicated complex instruction in the programming in time machines, but they add very small area and are built out of standard blocks. Thus, even if an application doesn't have any special patterns, there is little "wasted" logic blocks. We have also shown that the noticeable area increase of the PE due to extra features results in a relatively small change of the CGRA tile area because routing resources still dominate.

Chapter 6

Architecture evaluation

The previous chapters described how a flexible image processor can be built using CGRA. This chapter[1] will compare this approach to more traditional programming in time systems and estimates the cost of flexibility by comparing these results to the results for a fixed function ASIC implementation. It also considers the performance of current FPGAs, to see what advantage comes from the coarse grain nature of our CGRA. The next section describes the exact CGRA configuration and all the alternative approaches we consider: SIMD/VLIW, FPGA and ASIC.

6.1 Architecture options

The most interesting question is how a programmable in space image processing accelerator compares to a more common programmable in time architecture, so we are mostly interested in comparing our CGRA to SIMD/VLIW processor. In order to make a fair comparison on a hardware level, we'll be sharing as many components as we can between the two and apply a number of optimizations specific to each approach so make more realistic estimates. To find out the gain that comes from the coarse grain nature of our CGRA we will compare it to a traditional FPGA which our architecture is based on. Finally, we will also create a fixed function ASIC implementation to understand the cost of creating a programmable solution..

All implementations will have to meet at least a 4k @30fps performance requirement in order to be used in place of traditional ISP as a "real time" accelerator (see Imaging system architecture). To archive that, we allow the programmable architectures to use different number of resources - cores, tiles, etc., but the configuration and microarchitecture of this resource will have to remain the same across all benchmarks. In a similar fashion, we allow the area of fixed function implementation to vary based on application requirement.

[1] The majority of this chapter was previously published in 2016 MICRO paper[108].

6.1.1 CGRA configuration

The main parameters for the CGRA is the PE configuration and the number of tracks. The memory subsystem was optimized for images with 4K pixels width and used a previously described design (see Chapter 4.6). For this evaluation we have chosen the `2:1+extras` PE design which include basic set of OPs with a few extras like `min/max`, `abs`, etc. and *doesn't* have support for 32 bit operations or clustering. Furthermore, we will assume that PE and LUT functionality are mutually exclusive and only 1 OP can be executed by the tile. This choice allows us to use *exactly the same PE/ALU* in both programming in space and programming in time architectures.

The number of routing tracks for CGRA depends on the choice for PE but mostly it depends on the complexity of target applications as well as the router tool used. In our experiments we found that 12 tracks was sufficient for all our applications. Since the current VPR version doesn't support bus based routing, we turned all connections into 1 bit wires, after routing we used a custom script to parse the results and separate the signals back into 16 bit busses and 1 bit wires based on the annotated names. The consequence of this decision was that all the connections were mixed together and the total number of required tracks covers both 1 and 16 bit results. In order for the routing results to be valid we assume that 16 bit and 1 bit routing resources identical: both have 12 track with the same length distribution. This is a conservative approach because our largest benchmark application is dominated by 16 bit connections, thus the 1bit routing network will be overprovisioned. The cost of routing resources (tracks and mux-es) is proportional to the width of the signal, so the entire 1bit network will be much smaller compared to 16 bit one, thus this overprovisioning doesn't affect the results too much.

In a pure programming in space model, a PE doesn't change its operations during run time, thus an application would require as many PEs as the number of nodes in all kernel DAGs. To run different benchmarks on our CGRA, we will use different chip sizes based on applications' needs. The only change is the number of tiles, the design configuration including the number of tracks is the same across all applications. In reality, a chip would have a certain number of tiles fixed during manufacturing according to the largest application. If an application needs less resources, the remainder will either be used to run other tasks or power gate to save power - similar to how multiple cores are managed in modern CPU. For simplicity, we will use a square floor plan for PE arrays. This will inevitably result in some number of unused tiles which we will still count toward the total area of CGRA implementation for the particular application. Accounting for this area cost is fair because real chips have a notion of granularity - like a length of the vector in SIMD lane, or a "region" in FPGA. Typically the power gating and clock distribution is controlled on this granularity and it's only possible to shut off a group of elements rather than individual members of a large array.

6.1.2 SIMD: Programming in time

For programmable in time option, we used multi core SIMD/VLIW architecture shown on Fig. 6.1. Each core has three main parts: a scalar core that is responsible for control and instruction flow, a memory subsystem that hold the data acting as a line buffer and a vector core with several individual execution lanes.

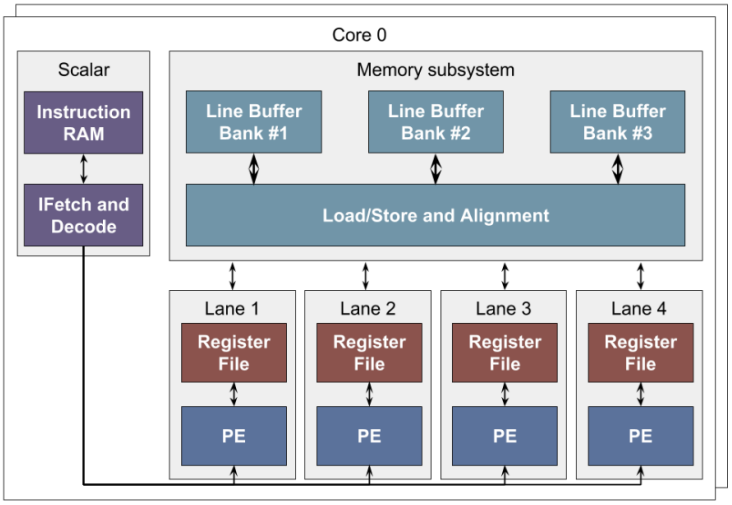

Figure 6.1: Programming in time architecture: multicore SIMD/VLIW.

Like most compute engines for highly data parallel applications, for the programmable in time option we use a SIMD architecture to amortize the instruction overhead over multiple data elements processed in parallel, and choose a SIMD width to make the resulting instruction overhead small. We parallelize over the image data, so if we represent the imaging application as a directed graph of operations, each SIMD lane processes this graph for its pixel. Like many others we use VLIW techniques to increase performance by enabling multiple ops to be processed each cycle, which also enables overlap of data movement (memory load/store) and computation.

The scalar core fetches an instruction from the instruction RAM, decodes it, and broadcasts control signals to multiple execution lanes (see Fig. 6.1). Each lane contains some local storage (register file), a processing element (PE), and a port to the memory subsystem. The register file is hierarchical and consists of a 16 registers and L0 spill SRAM that can hold additional 200 entries per lane. The lanes are fully independent and there are no interlane communication ports. This two level register organization was critical to optimize the energy dissipation of this approach.

A load-store unit fetches wide data from the system memory, splits it into the size for each lane, and loads it into the register file, or else it takes the data from the lanes, aggregates it and stores it into the memory. This unit contains a double wide register allowing data from two memory fetches to be concatenated together, and then a funnel shifter which can extract any contiguous vector from

this register to present the needed data to the right SIMD lane. This shifter is used to correctly align memory data for the SIMD engine.

We use a SIMD engine with 32 16-bit-wide independent lanes which is enough to reduce the instruction overhead, while not causing too much inefficiency with the block size. It has a conventional seven stage execution pipeline with variable length VLIW instructions. VLIW extension allows the overlap computation with data movement (load/store ops) and execution control (scalar core). Our functional units support 16-bit integer arithmetic (including multiply), logical and shift operations. It has two inputs and one output. The register file is initially sized to accommodate the largest active working set among all applications, while the instruction RAM is sized to accommodate the largest application.

We implement the line buffers as blocks in a large scratchpad memory. The access width of the scratchpad is equal to the SIMD vector width. We read the data that the kernel will operate on, one SIMD vector at a time, from the scratchpad memory into the load store unit. The load-store unit uses address calculated by the scalar core and funnels the incoming data to create multiple shifted vectors, which are then loaded in the register file, one at a time. For a convolution of width K, the loading of K shifted vectors triggers only two LB reads and K shifts in the load store unit. A $K \times N$ stencil gets stored in the register file as KN entries.

Our SIMD core is capable of processing entire input image, but the 32 PEs it contains doesn't have enough throughput to meet the real time requirement. The most obvious way to improve performance is to implement a multi-core SIMD system. We can divide the image into multiple "strips" and allow each core to process a strip. The downside of this approach is that the overlapping region at the strip boundaries must be read multiple times. However, with this approach, all cores follow the same schedule, and there are no inter-core dependencies. This makes the scheduling and implementation easier. The line buffer is divided into banks, with each bank storing data for a single core. The number of pixels stored in a bank is equal to the strip width. A core can access up to K (K=3 in Fig. 6.1) banks to its right, in order to support kernels with width greater than the strip width and to allow the kernel to access its neighbor's strip for data in the overlap region. Since the schedule for all cores is identical, the read and write addresses of all banks are identical and change in lockstep, avoiding any memory contention.

Our programming in time architecture is similar to GPU, but it is optimized to image processing in a few important ways. First it only supports 16 bit integer operations - no floating point. Second we take advantage of static scheduling which results in a simplified control for the core. Finally, the memory subsystem is optimized for line buffer with strided access. More information about the design choices and trade-offs for the SIMD/VLIW can be found in 2016 MICRO paper [108].

Although the memory subsystem of SIMD/VLIW architecture operates differently from CGRA, the capacity and bandwidth provided by it are the same. Since the control logic is small compared to SRAM storage we expect the cost of two approaches to be about the equal. The ALU blocks are

the same in both and the cost of configuration registers in CGRA tile would be roughly balanced by the cost of the control/scalar lane with instruction memory. Thus the difference between the two boils down to the cost of register file vs SB+CBs.

6.1.3 FPGA

For the FPGA comparison we'll be using the *k6-frac-N10-frac-chain-mem32K-40nm* architecture from VPR's distribution. It has 6-input, 2-output LUTs with carry chains for fast arithmetic organized into clusters of 10. Additionally it has hard DSP and SRAM blocks, all of which we use in our designs. The DSP module is 36 bit wide, but it can be split in two 18 bit multipliers, while SRAM has a capacity of 32kB, but it can be split into two blocks with 16kB each. This makes both of them similar to the multipliers and RS blocks used in GCRA. However the sample architecture supplied by VPR doesn't support pipelined wires. This was a very important feature for getting good area efficiency in our CGRA, so for fair comparison we'll estimate its effect on the baseline FPGA.

Overall the FPGA architecture we'll be using is similar to the modern commercial chips like Altera/Intel Stratix 10[50]. It uses the same implementation as our CGRA, but each compute block is implemented out of LUTs which area less efficient that PEs even for simple operations like add. On top of that, the routing is more expensive due to fine grain nature. The control logic for LB has to be build from general purpose LUTs, but for 4K images we expect 18k SRAM blocks to dominate. The difference between FPGA and CGRA will be in the fine vs coarse grain granularity of compute and routing.

6.1.4 ASIC

The ASIC is built using the same static pipeline model as our CGRA and implements the kernel DAGs in hardware using the same low level library elements (adder, multiplier, etc.). However each compute block is specialized to a particular operation and there is no programmable interconnect. Based on the CGRA tile area breakdown (see Fig. 5.6) we expect the fixed function implementation of compute data paths to be much more efficient especially for relatively cheap operations like add, shift and logical and. The difference in the memory subsystem will be small because LBs use large SRAMs which dominate over control logic. Our fixed function implementation is similar to the one described in Darkroom paper[48].

6.2 Methodology

For the evaluation, we selected a set of applications written in Darkroom DSL. Previous work[48] has shown that this code can be mapped to FPGA, CPU and ASIC. Our CGRA uses the DSL code directly without requiring the programmer to translate the applications into hardware specific

language, also we mostly rely on standard FPGA tools (VPR) which operated similarly to ASIC place & route. This allowed us to build an evaluation framework for implementations across different programmable architectures: SIMD/VLIW, CGRA, FPGA and compare them to ASIC in terms of area and energy efficiency.

6.2.1 Benchmark applications

To evaluate the architectures, we will use a set of five applications as a benchmark: Harris, Fast, ISP, FCAM and Stereo. Harris[44] and Fast[93] are corner detectors, where Fast uses a much simpler algorithm and hence has faster runtimes. ISP and FCAM are both complete image processing pipelines. ISP is a generic imaging pipeline that performs demosaicing, white balance, color correction, crosstalk correction, dead pixel suppression, and black level correction. FCAM is a more sophisticated version of this pipeline that does various levels of noise reduction, as well as higher quality (and more computationally intensive) versions of the other steps. Stereo[71] implements maximum similarity detection within a linear search distance. A more detail description for some of these applications can be found in Darkroom paper[48].

This benchmark set was chosen to stress different architectural components. For example, the Harris corner detector is dominated by line buffer accesses while Stereo is computation dominated. Fast contains many single bit operations and will benefits from an FPGA architecture, which can pack many such operations in a CLB. The large operation count of Stereo (two orders of magnitude larger than Harris) stresses the VPR mapper and the depth-first scheduler. Stereo also demands the highest amount of architectural resources like PEs, routing tracks and register file entries, and it lets us measure metrics at the upper utilization bound. ISP and FCAM are examples of "typical" applications that run on ISPs. Table 6.1 gives some statistics per each application used and lists their complexity. Here we define complexity as the sum of all operations weighted by bit width and normalized to 16 bits.

	Stereo	**FCAM**	**ISP**	**Fast**	**Harris**
Line Buffer access, per pixel	213	62	26	7	8
LineBuff capacity (4k Pxl lines)	70	50	20	6	8
Shift Reg access	1491	402	129	52	22
LOGIC (1 bit OPs)	0	198	41	351	4
SHIFT	0	289	129	1	14
MUX	65	1285	198	1	1
ALU OPs (add, sub, abs, etc)	9683	3313	584	50	51
MULT	2	306	61	3	20
Div	1	4	0	0	0
Actual Ops	9750	5391	1013	406	90
Complexity (effective 16b Ops)	9750	5205	975	77	86
Perf target(4K@30fps), GOp/sec	2454	1357	255	102	23

Table 6.1: Application complexity and statistic.

6.2.2 Evaluation framework

To evaluate our CGRA to other solutions, we will compare the architectures using **energy efficiency**(pJ/operation) and **compute density**(operations/sec/mm2). This choice takes into account that all the solutions meet required performance level and reflects the fact that ISP are usually used in mobile and embedded systems that are powered from the battery and often have tight physical constraints. To derive these metrics, we ran the same benchmark applications on each architecture and tracked the activity of the hardware blocks during execution, e.g. RF, L0, PE and memory accesses for SIMD, and bus transitions for CGRA. We then combined this activity data with a table of the actual energy costs(Tab. 6.2) of the operations extracted from simulating block implementations.

We implemented the architectural components as Verilog models and placed and routed them using industry standard tools for the TSMC 40G(40nm) technology node, to determine performance and area. To estimate energy consumed, we simulated the routed netlists by applying uniformly random data to the data inputs while keeping the control signals constant at their expected values. This gave us toggle counts and, using the parasitics from the routed netlists, Synopsys' Primetime-PX tool estimated gate and wire energy. We compiled the memory instances used in the architectures with ARM's single port SRAM memory compiler and derived their performance, area, and access energy from the resulting compiler-generated datasheets.

Operation	Energy, pJ	Relative cost
Compute		
16b int ADD	0.03	1
16b int MULT	0.4	13
16b MUX-2	0.013	0.4
Memory		
16b register	0.03	1
16b 16-entry Reg File	0.12	4
16b 16 kB SRAM	1.4	47
16b DRAM [113]	128-240	4267-8000
Communication		
100um 1 bit wire per transaction [53]	0.012	0.4
100um 16 bit bus(activity factor 0.1-0.25)	0.019-0.048	0.6-1.6

Table 6.2: Energy cost of compute, memory and communication in 40nm.

For all benchmarks we used the same Darkroom DSL implementation across all platforms. The code was compiled to an intermediate representation (IR) which consisted of multiple kernels (filters) connected to produce a final image, plus descriptions of the DAG for each kernel (see Section 1.1.2). The DAG description consists of nodes doing simple operations like ADD, AND, MUX and MULT that can be scheduled on PEs. This IR was processed with architecture specific set of tools. On the SIMD, we scheduled each kernel's DAG in a depth-first-search manner(keeping any intra-kernel loops intact), and counted the number of clock cycles, register file accesses, instruction memory accesses, and load/stores needed to generate an output pixel. For CGRA, we used our VPR based flow (see Section 3.3) to map the completely unrolled DAGs onto a homogeneous array of CGRA tiles with configurable routing. We counted the number of tracks being used in each channel, the number of live buses in each CGRA tile, and the minimum number of CGRA tiles needed to fully route the DAG. In addition to predicting usage for computing energy, these metrics also enabled us to design the architectural components.

SIMD energy and compute density

For SIMD, we assume each IR instruction requires an instruction fetch i, plus one or more data fetches r from the register file, plus execution of an op in the PEs, and possibly a line buffer fetch L. We counted each of these events while running the scheduled code to get totals n_i, n_r, n_{op} and n_L respectively. Then, using the per-event energy found by simulation (see Tab. 6.3), and given time t_{app} for the app to run, we calculated energy E and compute density C as:

$$E = (n_i e_i + n_r e_r + n_{op} e_{op} + n_L e_L)/n_{op}$$
$$C = (n_{op}/t_{app})/(total\_area)$$

Our SIMD configuration has **32 lanes** and runs at **800MHz**. Given the component area breakdown (see Tab. 6.3), we can derive the total area for a single core as: **200,768 squm**.

Component	Area, squ	Energy, pJ	Notes
IRAM / Scalar core	28000	4	
Func Unit	42016	11.2	Energy is additional overhead over basic cost
Vector Reg File	44960	4	16x16b+ 16x1b per lane
Data Spill Ram	85792	48.3	200x16b entry per lane

Table 6.3: Area and energy parameters for SIMD.

CGRA energy and compute density

For CGRA, each IR instruction is mapped to a tile. The tiles are then placed and routed using VPR to minimize tile array size while ensuring complete connectivity. We obtained the number of live buses in every tile and the number of active and passive tiles from VPR logs (tiles that have been mapped to an IR operation are active tiles, while tiles that are used only for routing are passive tiles). We simulated tiles with different numbers of live buses and different PE mappings. We thereby got the energy of CGRA tile components for different activity levels. Using these energy numbers and VPR statistics, we derived the energy and compute density of each CGRA application in the same manner as with SIMD.

The key area and energy parameters for CGRA are presented in Tab. 6.4. Based of that, we can derive the total PE tile area as: PE+SB+CB = **4022squm** and the length of 1 unit wire track as: SQRT(tile area) = **63.4um**. In CRGA configurations, track length are often expressed in *units* which and unit 1 is the distance between neighboring tiles a regular CGRA array grid. We use tile area to compute the total area of the chip and to estimate the amount of leakage. The unit length is used to calculate the energy dissipated by wires for each application.

Component	Area, squ	Energy, pJ	Notes
PE	1327	0.4	Energy is additional overhead over basic cost of operation
SB	1516	0.02	MUX energy to drive 16b bus, doesn't include wire track
CB	545	0.04	Energy for one 16b connection box
Configuration	633	n/a	Static storage of configuration bits in each tile. ≈12B SRAM or latches

Table 6.4: Area and energy parameters for CGRA.

For simplicity, we used the same **800 MHz** for CGRA frequency as a SIMD clock. This target affected the number of registers inserted (see Fig. 3.9) and resulted in more performance than required by application. We could lower the frequency target to slightly increase the energy efficiency, but it would significantly lower the area efficiency, skewing the comparison results.

FPGA energy and compute density

For FPGA comparison we used the *k6-frac-N10-frac-chain-mem32K-40nm* architecture from VPR's distribution. This allowed us to use the same set of tools for placement and routing as we used for CGRA. Like CGRA, this is a 40nm part, and the model comes annotated with timing and energy information, providing a good comparison. For comparison, we assumed that FPGA supports pipeline wires and that all benchmarks run at **741 MHz** which was derived from the DSP block timing parameters of the VPR's reference architecture.

ASIC energy and compute density

For the ASIC comparison we translated each application's graph directly to Verilog and used standard synthesis and back-end tools to generate the physical implementation. Area, performance and energy numbers were extracted from this implementation. The ASIC designs used the same SRAM library as the other implementations, but could use memory sizes that were optimized for that particular application. These designs did not worry about being compatible with a range of applications.

6.3 Results

6.3.1 Energy efficiency: Energy per Op

Table 6.5 shows the energy per op for each architecture. To separate out structural overhead, versus the overhead caused by communication, we include a "peak" energy efficiency number in this table. The peak number gives the energy cost of a single local operation for each of the different architectures, i.e. a not-to-be-exceeded number that would happen only if the application consisted only of 16-bit adds with locally generated inputs. So the ASIC peak number is simply a 16-bit adder running at 1 GHz. Peak CGRA efficiency reflects one tile doing 16-bit adds at 800 MHz with data coming from adjacent units. SIMD peak efficiency is calculated for a single core with 32 lanes performing 16-bit adds at 800 MHz where two vectors are read while one is written to the register file. FPGA peak energy efficiency is modeled as a single CLB doing 16-bit add at 741 MHz counting the energy of CLB, SB and CB.

The peak numbers show the large energy overhead of creating a programmable system with flexible communication, even with locality. The lowest possible energy/op for all programmable platforms uses over 10x more energy than ASIC, which only contains the energy of the adder. The buses, mux-es, registers, etc. that are included to make the adder part of a PE all add overhead to its fundamental operation. While this overhead energy is not large, the add energy is very small, so the overhead energy dominates it.

Since the CGRA and SIMD machines use the same basic function unit design, their energy difference is caused by the cost of the register file relative to the connection box cost. This difference

	CGRA	SIMD	FPGA	ASIC
Harris	1.38	2.08	3.69	0.53
Fast	3.91	6.27	3.19	0.52
ISP	0.91	1.68	1.12	0.20
FCAM	1.04	1.41	1.42	0.15
Stereo	1.30	1.94	3.19	0.13
Average	1.71	2.68	2.35	0.31
Peak	0.38	0.76	1.35	0.03

Table 6.5: Energy per Op for the selected imaging applications on each architecture, pJ/Op

makes sense, as the SIMD register file explicitly does some amount of communication that the CGRA must do with longer with wires. Thus while the peak SIMD:CGRA ratio is 2:1, on average SIMD takes **1.6** times the energy of a CGRA implementation. The energy difference between an FPGA with pipelined wires and the CGRA is mostly caused by slightly longer wires in the FPGA and less efficient functional units, so FPGA takes 1.4 times more energy than CGRA.

While the line buffer energy is not a significant factor in the programmable solutions, it is important in ASIC implementations. For applications like FCAM and Stereo whose kernels have significant computation between line buffers, computation energy dominates even in the ASIC implementation and the gap between CGRA and ASIC is around **10x** in energy. However, smaller applications like Harris, Fast and ISP that are dominated by LB memory accesses have only around 3x gap in energy efficiency, since the line buffer energy in both implementations is similar.

Figure 6.2 breaks down energy consumption in SIMD and CGRA architectures. Since we use the same functional units in both designs, it is not surprising that the energy of the function units are nearly identical for the two architectures. ("func unit" for SIMD, "PE" for CGRA). The difference in total energy comes from the different cost of communication. In SIMD this is the combination of the register file and spill RAM energy and averages over 1 pJ/op, while the combined cost of the wires, switch box and connection box in the CGRA is around 0.5 pJ/op. Note that in both architectures the "overhead energy" is generally low.

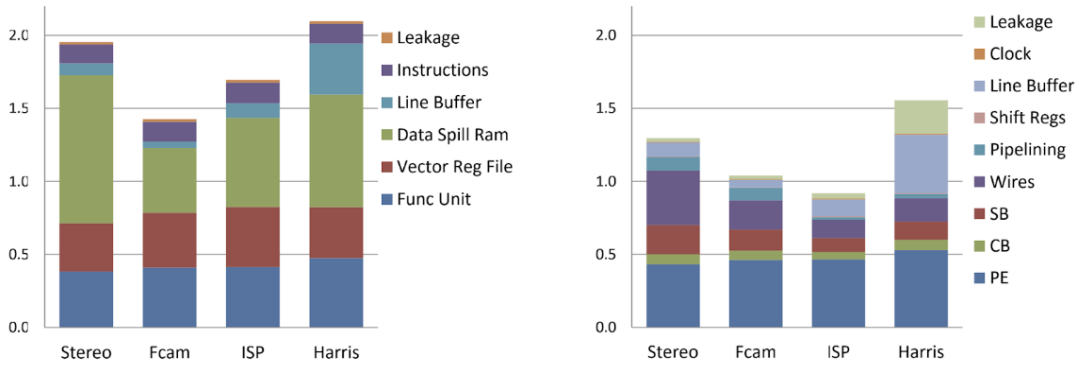

Figure 6.2: Energy breakdown in pJ/OP for the selected imaging applications (SIMD on the left, CGRA on the right).

6.3.2 Area efficiency: Area per Op/s

Table 6.6 tells the required area/GOPS for the different implementation approaches, with the peak line again giving the minimum possible mm^2 /GOPS. We use the same base configuration for calculating the peak compute density as energy for all the targets except for FPGAs, where we now include multipliers with the LUTs. When viewing these numbers, it is important to remember that the presence of the line buffers makes calculating compute density tricky for two reasons. First, since the LB area doesn't scale with application performance, lower performance (and hence smaller area) solutions have worse compute densities. So, since a single core SIMD machine has lower performance and lower area than the competing architectures - e.g. for FCAM it requires $2.6mm^2$ compared to CGRA's $25mm^2$ - to fairly compare their compute density, we used multiple SIMD cores to create solutions with matching performance.

	CGRA	**SIMD**	**FPGA**	**ASIC**
Harris	.013	.013	.039	.0046
Fast	.034	.046	.032	.0038
ISP	.007	.010	.016	.0011
Fcam	.006	.009	.017	.0006
Stereo	.006	.008	.024	.0004
Average	.013	.017	.026	.0021
Peak	.005	.008	.012	.00009

Table 6.6: Area efficiency for selected imaging applications on each architecture, in mm2/GOPS, including line buffer area.

The second issue is the size of LB that is included in the area cost. In an ASIC solution, you build the size LB that its single application requires. For a programmable solution, you provide sufficient resources to solve the entire range of problems it might see. Thus for most applications,

the programmable engine has surplus memory that it doesn't use. This extra memory further decreases the compute density. So, while the actual hardware contains a maximally-sized LB, Table 17 calculates a per-application area efficiency using only that portion of the LB that was required for each application, with the assumption that a user generally tries to put as much of the computation on the accelerator as will fit: it is unlikely that only Harris would use the accelerator, so when the accelerator is actually deployed in a real system, other applications would be using the part of the LB that that the Harris application didn't need.

To quantitatively address these issues, Table 6.7 shows the LB size needed and GOP rate for each application assuming 800MHz. Table 6.8 provides the area of an LB block in each of our programmable architectures. Together this data allows one to compute the area/GOPS overhead that is caused by the LB (for the size LB you choose to use). In addition, Table 6.7 gives the mm^2 per GOPS for the hardware excluding the LB, so adding the two terms together generates the data in Table 6.6.

	CGRA	**SIMD**	**FPGA**	**LB size, rows**	**Perf, GOPs**
Harris	0.007	0.008	0.022	8	69
Fast	0.029	0.041	0.018	6	62
ISP	0.005	0.009	0.012	20	780
Fcam	0.005	0.008	0.015	50	4164
Stereo	0.006	0.008	0.022	70	7800

Table 6.7: Area efficiency without line buffer, mm^2/GOPS.

CGRA	**SIMD**	**FPGA**	**Notes**
0.104	0.094	0.266	2 image rows per 16kB LB

Table 6.8: Size of the 16kB LB in mm^2.

For example, in Table 6.7 we see that Harris uses about one tenth of the LB that Stereo requires. CGRA compute hardware for Harris uses 0.007 mm^2/GOPS, so adding just the LB it needs (8 rows or 4 RS blocks) adds 0.42 mm^2/69 GOPS or 0.006 mm^2/GOPS yielding 0.013 mm^2/GOPS, which is the approximately the number in Table 6.6. Yet if you add the area for the full line buffer, the LB area increases by almost **10x**, and the area/GOPS increases to 0.6 mm^2/GOPS. This means for Harris the programmable solution is either around 3x or 12x the area of the ASIC solution depending on how you account for the LB area. Since the LB is only important for the simpler applications and depends on interpretation, the rest of this discussion will focus on the compute density excluding the LB.

Table 6.7 shows that Fast is an outlier since it is dominated by binary operations. Neither SIMD nor CGRA was optimized for this type of operation (even though they could be) so their compute density falls by about 5x. The table shows that CGRA and SIMD compute density are not far apart, being on average only about 40% different (excluding Fast), i.e. SIMD is taking about 40%

more resources than CGRA and thus has about 40% lower compute density. The reason for this difference can be seen in Table 6.9. Again, the area of the functional units is nearly the same in the two designs, but the area used for the programmable wires in the CGRA is less than the area required for the memory in the SIMD, even after the register file has been optimized.

SIMD lane		**CGRA tile**	
Func Unit	1313	PE	1327
Vector Reg File	1405	SB	1517
Data Spill Ram	2681	CB	545
Iram	875	Configuration	633

Table 6.9: Absolute area breakdown comparison for SIMD lane and CGRA tile, um^2.

Both SIMD and CGRA have much better compute density than regular FPGA (without pipelined wires), largely because of the difference in clock rate. Pipelining the CGRA wires and functional units was essential to reach 800 MHz. Adding wire pipelines to the FPGA greatly improved its area efficiency, but it is still 3.1 x worse if Fast is excluded (see FPGA in Table 6.7), because LUTs are less efficient than ALUs for addition and because it's hard macro blocks are not as well tuned to the application domain. The FPGA memory is dual ported and not as wide as our RS blocks, and the DSP block is a multi-precision 36-bit unit.

6.4 Discussion

As we saw in the results(Fig. 6.2 and Tab. 6.9), most of the difference in both energy and area efficiency between programming in space and programming in time approaches can be attributed to the SB+CB versus register file+spill ram, which suggest there is a difference in how big these modules have to be. We can traces the roots of this difference to a fundamental way the two machines operate. Each SIMD lane executes the entire application graph, so the compiler serializes this graph into a 1D stream of instructions and the register file(+spill ram) has to be sized to hold the worst number of temporary values (which correspond to active edges). In CGRA, on the other hand, the router can distribute connections on a 2D grid and avoid "hot" spots. Thus CGRA routing is sized according to the average requirement. This difference is fundamental to the parallelization strategies and mode of operations.

Additionally, SIMD architecture has to distribute instruction word to every lane. The instruction word gets quite wide in case of VLIW and would have to toggle on every clock cycle which reduces potential energy savings from having all the operands locally as opposed to bringing them from remote tiles through a series of routing resources which operate like a shift register in the CGRA.

Our result(Tab. 6.5 and Tab. 6.6) shows that CGRA is slight better than SIMD/VLIW and much better than FPGA, but the price of programmability is still quite high, especially for compute intensive applications. We can improve this in some case by increasing the compute density with more

complicated PE design and support for special application patterns as was discussed in Chapter 5.5. As we previously discussed (see Section 5.4), full support of the special patterns requires significant change to the tool flow. However, one can estimate the area efficiency, by just modifying the technology mapping (see Section 3.3). Once the design is mapped, the number of tiles that will be used in the design is known. Thus we can evaluate the area saving from increased compute density logic by comparing the number of PEs (same as tiles) in the architectural IR after the technology mapping.

We have modified the technology mapping step to recognize these special patterns: MAD, SUM, SAD and DOT (see Section 5.4). Additionally we want to consider the effect of using both 16 bit and 1 bit compute slots in a PE by allowing mapper to use both ALU and a LUT rather than selecting just one type of operation. Fig. 6.3 shows the relative area of these three configurations over the basic 2:1 design we used for evaluation (see Section 6.1.1). These results reflect both the change in total number of tiles and the area increase of individual tiles(see Tab. 5.7).

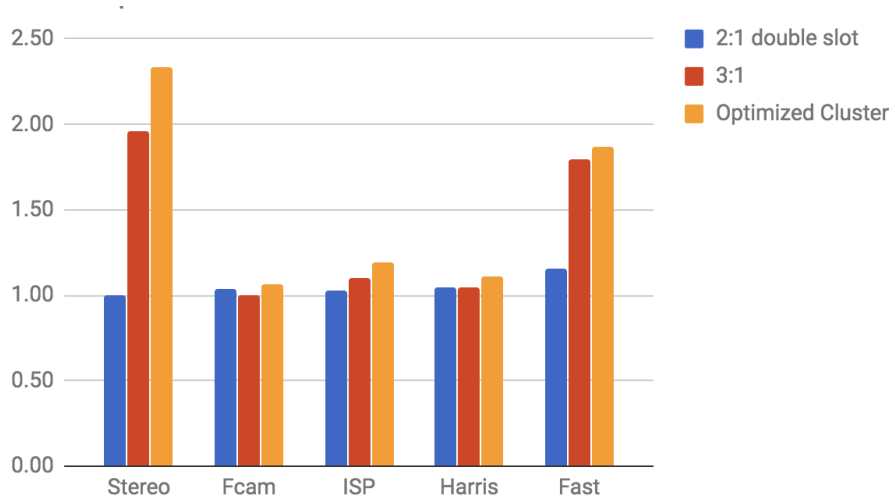

Figure 6.3: CGRA area efficiency improvements from increased compute density and clustering.

As we can see (Fig. 6.3), all three configurations do improve the area. Simply overlapping 1 bit and 16 bit operations has a small effect because the benchmark are dominated by ether the 16bit or 1 bit operations (see Tab. 6.1). Increasing compute density with 3:1 PE design or clustering gives a significant improvement to applications with a large percentage of special patterns, like Stereo' which is dominated by SAD. We can see the advantage of the clustering design over fused operations in 3:1 configuration. Due to smaller individual tile area, clustering yields area improvements across all benchmarks, while 3:1 barely breaks even on Fcam'. We observed a larger improvement of clustering on Stereo' which was due to the simple approach we used to map special patterns on 3:1 and clustering configurations. On a real application, our method works better if reduction is

done using a tree structure rather than a long chain, as a result the mapping to a 3:1 PE was not as efficient as mapping to a cluster which lead to larger number of tiles. We expect that a better algorithm will correct this and the area difference will be ≈10% as predicted by the difference in tile area (see Section 5.4.1).

Chapter 7

Conclusions

In this thesis, we showed how a flexible accelerator for imaging applications can be built using a CGRA architecture by leveraging the micro architecture of a fixed function ISP and the spatial programmability of FPGAs. Unlike most of the existing specialized processors which use programmability in time, CGRA operates more similar to ASIC implementations. It is programmable in space, building very large pipelines and using data flow architecture. We have demonstrated how a code written in DSL can be mapped to CGRA using existing DSL compilers and a Place & Route tool for FPGAs which are connected into unified flow with several custom scripts responsible for technology mapping and implementation of certain CGRA features not commonly found in FPGAs.

We have describes a micro-architecture and implementation of major CGRA subsystems that enable flexibility and build upon ideas from other architectures: FPGA for routing, ASIC for line buffers and CPU for compute. For each subsystem we demonstrated how various features required to make them flexible can be implemented in a programming in space setting. We have built a set of hardware generators to explore a design space for building compute subsystem starting with the simple design that only covers a minimal set of feature, to a design with increased compute density using fused operation and eventually arrived to a cluster configuration that supports higher precision operation and can efficiently implement a set of special patterns often using in the imaging domain.

Finally, we have created a framework to evaluate our CRGA and compare it other programmable architectures like FPGA and SIMD/VLIW. Using this framework with have showed that programming in space approach is more efficient than programming in time (about **1.5** times better in both energy and area efficiency). However, the cost of this efficiency gain is flexibility - CGRA uses pipeline parallelism and needs a large number of units operating in lock step to be efficient. Programming in time also relies on data parallelism but uses a relatively small number of parallel units, equal to the SIMD vector length, this leads to a smaller, lower performance implementation. To improve performance, several cores need to be used, but since each core executes an entire application, the *minimal* area of the chip is small. With programming in time the performance can be gradually increased in small chunks by adding more cores, which makes it more flexible compared to “all or

nothing" with the programming in space.

Using our evaluation framework, we compared CGRA to ASIC and determined that the cost of programmability was still high - about **10x** in both energy and area. We have showed that in some cases the efficiency of CGRA can be noticeably improved by increasing compute density of the PEs and taking advantage of the special compute patterns that exists in applications. With programming in space approach, support of these patterns is relatively cheap and easy. By reusing the hardware already present in the PE for larger precision math - the only cost of special patterns support in CGRA is small area increase. There is no penalty on energy because the configuration word doesn't change during operation, in fact the energy is decrease for the special patterns because they use direct connections within cluster instead of flexible routing resources.

Bibliography

[1] 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.

[2] Elias Ahmed and Jonathan Rose. The effect of lut and cluster size on deep-submicron fpga performance and density. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12(3):288–298, 2004.

[3] Altera. Stratix ii gx device handbook, volume 1, siigx5v1-4.4. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stx2gx/stxiigx_handbook.pdf`, 2007.

[4] Altera. Stratix v device handbook. `https://www.altera.com/documentation/sam1403479391092.html`, 2017.

[5] Gene M Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[6] Eduardo Angel. DIGIC processors explained. *Canon, Inc., http://www.learn.usa.canon.com/resources/articles/2012 /digic_processors.shtml*, January 2012.

[7] ARM. Neon programmers guide. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.den0018a/index.html`.

[8] Halil B Bakoglu. Circuits, interconnections, and packaging for vlsi. 1990.

[9] Jonathan T Barron. Convolutional color constancy. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 379–387, 2015.

[10] Vaughn Betz and Jonathan Rose. Cluster-based logic blocks for fpgas: Area-efficiency vs. input sharing and size. In *Custom Integrated Circuits Conference, 1997., Proceedings of the IEEE 1997*, pages 551–554. IEEE, 1997.

[11] Vaughn Betz and Jonathan Rose. Vpr: A new packing, placement and routing tool for fpga research. In *International Workshop on Field Programmable Logic and Applications*, pages 213–222. Springer, 1997.

[12] Vaughn Betz and Jonathan Rose. Fpga routing architecture: Segmentation and buffering to optimize speed and density. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 59–68. ACM, 1999.

[13] Vaughn Betz, Jonathan Rose, and Alexander Marquardt. *Architecture and CAD for deep-submicron FPGAs*, volume 497. Springer Science & Business Media, 2012.

[14] Olexa Bilaniuk, Ehsan Fazl-Ersi, Robert Laganiere, Christina Xu, Daniel Laroche, and Craig Moulder. Fast lbp face detection on low-power simd architectures. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 616–622, 2014.

[15] Gary Bradski and Adrian Kaehler. Opencv. *Dr. Dobbs journal of software tools*, 3, 2000.

[16] Matthew Brown and David G Lowe. Automatic panoramic image stitching using invariant features. *International journal of computer vision*, 74(1):59–73, 2007.

[17] John S Brunhaver. *Design and optimization of a stencil engine*. PhD thesis, Ph. D. dissertation, Stanford University, 2015.

[18] Robert Bushey, Hamed Tabkhi, and Gunar Schirner. Flexible function-level acceleration of embedded vision applications using the pipelined vision processor. In *Signals, Systems and Computers, 2013 Asilomar Conference on*, pages 1447–1452. IEEE, 2013.

[19] Fabio Campi, Mario Toma, Andrea Lodi, Andrea Cappelli, Roberto Canegallo, and Roberto Guerrieri. A vliw processor with reconfigurable instruction set for embedded applications. In *Solid-State Circuits Conference, 2003. Digest of Technical Papers. ISSCC. 2003 IEEE International*, pages 250–491. IEEE, 2003.

[20] Lukas Cavigelli and Luca Benini. Origami: A 803-gop/s/w convolutional network accelerator. *IEEE Transactions on Circuits and Systems for Video Technology*, 27(11):2461–2475, 2017.

[21] Yao-Wen Chang, DF Wong, and Chak-Kuen Wong. Universal switch modules for fpga design. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 1(1):80–101, 1996.

[22] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284, 2014.

[23] Don Cherepacha and David Lewis. Dp-fpga: An fpga architecture optimized for datapaths. *VLSI Design*, 4(4):329–343, 1996.

[24] Minsik Cho and David Z Pan. Boxrouter: a new global router based on box expansion and progressive ilp. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(12):2130–2143, 2007.

[25] Jason Clemons, Chih-Chi Cheng, Iuri Frosio, Daniel Johnson, and Stephen W Keckler. A patch memory system for image processing and computer vision. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 51. IEEE Press, 2016.

[26] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon dsp: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, (2):34–43, 2014.

[27] Peter Cumming. The ti omap platform approach to soc. In *Winning the SOC Revolution*, pages 97–118. Springer, 2003.

[28] Benoît Dupont de Dinechin, Renaud Ayrignac, Pierre-Edouard Beaucamps, Patrice Couvert, Benoit Ganne, Pierre Guironnet de Massas, François Jacquet, Samuel Jones, Nicolas Morey Chaisemartin, Frédéric Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*, pages 1–6, 2013.

[29] Paul E Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *ACM SIGGRAPH 2008 classes*, page 31. ACM, 2008.

[30] P. Desai. Choosing the right dsp for high-resolution imaging inmobile and wearable applications. `http://ip.cadence.com/uploads/899/TensilicaVisionP5WPFinal100515-pdf`, 2017.

[31] Keith Diefendorff. Pentium iii= pentium ii+ sse. *Microprocessor Report*, 13(3):1–6, 1999.

[32] Michael Ditty, John Montrym, Craig Wittenbrink, et al. Nvidia's tegra k1 system-on-chip. In *Hot Chips 26 Symposium (HCS), 2014 IEEE*, pages 1–26. IEEE, 2014.

[33] Wilm E Donath. Wire length distribution for placements of computer logic. *IBM Journal of Research and Development*, 25(3):152–155, 1981.

[34] Carl Ebeling, Darren C Cronquist, and Paul Franklin. Rapidreconfigurable pipelined datapath. In *International Workshop on Field Programmable Logic and Applications*, pages 126–135. Springer, 1996.

[35] Tarek El-Ghazawi, Esam El-Araby, Miaoqing Huang, Kris Gaj, Volodymyr Kindratenko, and Duncan Buell. The promise of high-performance reconfigurable computing. *Computer*, 41(2), 2008.

[36] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE, 2011.

[37] Bradly K Fawcett. Taking advantage of reconfigurable logic. In *ASIC Conference and Exhibit, 1994. Proceedings., Seventh Annual IEEE International*, pages 227–230. IEEE, 1994.

[38] Tom Feist. Vivado design suite. *White Paper*, 5, 2012.

[39] Antonio Gentile, Salvatore Vitabile, Lorenzo Verdoscia, and Filippo Sorbello. Image processing chain for digital still cameras based on the simpil architecture. 2005.

[40] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[41] 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.

[42] Tom R Halfhill. Tabulas time machine, rapidly reconfigurable chips will challenge conventional fpgas. *Microprocessor Report*, 2010.

[43] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.

[44] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, pages 10–5244. Citeseer, 1988.

[45] Reiner Hartenstein. Coarse grain reconfigurable architecture (embedded tutorial). In *Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 564–570. ACM, 2001.

[46] 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):192, 2016.

[47] GC Hawkes. Dsp: Designing for optimal results. high-performance dsp using virtex-4 fpgas. *Advanced Design Guide. Xilinx Inc*, 1, 2005.

[48] 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 Trans. Graph.*, 33(4):144–1, 2014.

[49] Intel. Understanding how the new intel hyperflex fpga architecture enables next generation high-performance systems. `https://www.altera.com/en_US/pdfs/literature/wp/wp-01231-understanding-how-hyperflex-architecture-enables-high-performance-systems.pdf`, 2017.

[50] Intel. Intel stratix 10 high-performance design handbook. `https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/stratix-10/s10_hp_hb.pdf`, 2018.

[51] Mircea Horea Ionica and David Gregg. The movidius myriad architecture's potential for scientific computing. *IEEE Micro*, 35(1):6–14, 2015.

[52] Syed MAH Jafri, Tuan Nguyen Gia, Sergei Dytckov, Masoud Daneshtalab, Ahmed Hemani, Juha Plosila, and Hannu Tenhunen. Neurocgra: A cgra with support for neural networks. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 506–511. IEEE, 2014.

[53] Stephen W Keckler, William J Dally, Brucek Khailany, Michael Garland, and David Glasco. Gpus and the future of parallel computing. *IEEE Micro*, (5):7–17, 2011.

[54] Ubaid R Khan, Henry L Owen, and Joseph LA Hughes. Fpga architectures for asic hardware emulators. In *ASIC Conference and Exhibit, 1993. Proceedings., Sixth Annual IEEE International*, pages 336–340. IEEE, 1993.

[55] Sami Khawam, Ioannis Nousias, Mark Milward, Ying Yi, Mark Muir, and Tughrul Arslan. The reconfigurable instruction cell array. *IEEE Transactions on very large scale integration (VLSI) systems*, 16(1):75–85, 2008.

[56] Satoru Komatsu, Mitsumaro Kimura, Akira Okawa, and Hideaki Miyashita. Milbeaut image signal processing LSI chip for mobile phones. *Fujitsu Sci. Tech. J*, 49(1):17–22, 2013.

[57] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[58] Ian Kuon and Jonathan Rose. Measuring the gap between fpgas and asics. *IEEE Transactions on computer-aided design of integrated circuits and systems*, 26(2):203–215, 2007.

[59] Ian Kuon, Russell Tessier, Jonathan Rose, et al. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

[60] Bernard S Landman and Roy L Russo. On a pin versus block relationship for partitions of logic graphs. *IEEE Transactions on computers*, 100(12):1469–1479, 1971.

[61] Chin Yang Lee. An algorithm for path connections and its applications. *IRE transactions on electronic computers*, (3):346–365, 1961.

[62] Charles E Leiserson and James B Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1-6):5–35, 1991.

[63] Guy Lemieux, Edmund Lee, Marvin Tom, and Anthony Yu. Directional and single-driver wires in fpga interconnect. In *Field-Programmable Technology, 2004. Proceedings. 2004 IEEE International Conference on*, pages 41–48. IEEE, 2004.

[64] Guy Lemieux and David Lewis. *Design of interconnection networks for programmable logic*, volume 22. Springer, 2004.

[65] Eric Lemoine and David Merceron. Run time reconfiguration of fpga for scanning genomic databases. In *FPGAs for Custom Computing Machines, 1995. Proceedings. IEEE Symposium on*, pages 90–98. IEEE, 1995.

[66] Marc Levoy and Pat Hanrahan. Light field rendering. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 31–42. ACM, 1996.

[67] David Lewis, Elias Ahmed, Gregg Baeckler, Vaughn Betz, Mark Bourgeault, David Cashman, David Galloway, Mike Hutton, Chris Lane, Andy Lee, et al. The stratix ii logic and routing architecture. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 14–20. ACM, 2005.

[68] David Lewis, Vaughn Betz, David Jefferson, Andy Lee, Chris Lane, Paul Leventis, Sandy Marquardt, Cameron McClintock, Bruce Pedersen, Giles Powell, et al. The stratixπ routing and logic architecture. In *Proceedings of the 2003 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, pages 12–20. ACM, 2003.

[69] David M Lewis et al. Analytical framework for switch block design. In *International Conference on Field Programmable Logic and Applications*, pages 122–131. Springer, 2002.

[70] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[71] Zheng Lu, Yu-Wing Tai, Fanbo Deng, Moshe Ben-Ezra, and Michael S Brown. A 3d imaging framework based on high-resolution photometric-stereo and low-resolution depth. *International journal of computer vision*, 102(1-3):18–32, 2013.

[72] Jason Luu, Jeffrey Goeders, Michael Wainberg, Andrew Somerville, Thien Yu, Konstantin Nasartschuk, Miad Nasr, Sen Wang, Tim Liu, Nooruddin Ahmed, et al. Vtr 7.0: Next generation architecture and cad system for fpgas. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 7(2):6, 2014.

[73] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Speed and area tradeoffs in cluster-based fpga architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 8(1):84–93, 2000.

[74] Grant Martin and Gary Smith. High-level synthesis: Past, present, and future. *IEEE Design & Test of Computers*, 26(4):18–25, 2009.

[75] M Imran Masud and Steven JE Wilton. A new switch block for segmented fpgas. In *International Workshop on Field Programmable Logic and Applications*, pages 274–281. Springer, 1999.

[76] Larry McMurchie and Carl Ebeling. Pathfinder: a negotiation-based performance-driven router for fpgas. In *Field-Programmable Gate Arrays, 1995. FPGA'95. Proceedings of the Third International ACM Symposium on*, pages 111–117. IEEE, 1995.

[77] Nick Mehta. Xilinx 7 series fpgas: the logical advantage. *Xilinx WP405*, 2012.

[78] Robert K. Montoye, Erdem Hokenek, and Stephen L. Runyon. Design of the ibm risc system/6000 floating-point execution unit. *IBM Journal of research and development*, 34(1):59–70, 1990.

[79] Junichi Nakamura. *Image sensors and signal processing for digital still cameras*. CRC press, 2016.

[80] Zoran Nikolić. Embedded vision in advanced driver assistance systems. In *Advances in embedded computer vision*, pages 45–69. Springer, 2014.

[81] C Nvidia. Nvidias next generation cuda compute architecture: Kepler gk110. *Whitepaper (2012)*, 2012.

[82] Andreas Olofsson, Roman Trogan, and Oleg Raikhman. A 25 gflops/watt software programmable floating point accelerator. In *High Performance Embedded Computing Computing Conference*, 2010.

[83] Angshuman Parashar, Michael Pellauer, Michael Adler, Bushra Ahsan, Neal Crago, Daniel Lustig, Vladimir Pavlov, Antonia Zhai, Mohit Gambhir, Aamer Jaleel, et al. Triggered instructions: a control paradigm for spatially-programmed architectures. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 142–153. ACM, 2013.

[84] Hyun Sang Park. Architectural analysis of a baseline isp pipeline. In *Theory and Applications of Smart Cameras*, pages 21–45. Springer, 2016.

[85] 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):26, 2017.

[86] II Quartus. Handbook version 10.1 volume 1: Design and synthesis. *Altera Corporation*, 130, 2010.

[87] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. 2012.

[88] Rajeev Ramanath, Wesley E Snyder, Youngjun Yoo, and Mark S Drew. Color image processing pipeline. *IEEE Signal Processing Magazine*, 22(1):34–43, 2005.

[89] James Reinders. Avx-512 instructions. *Intel Corporation*, 2013.

[90] Rahul Rithe, Priyanka Raina, Nathan Ickes, Srikanth V Tenneti, and Anantha P Chandrakasan. Reconfigurable processor for energy-efficient computational photography. *IEEE Journal of Solid-State Circuits*, 48(11):2908–2919, 2013.

[91] Jonathan Rose and Stephen Brown. Flexibility of interconnection structures for field-programmable gate arrays. *IEEE Journal of Solid-State Circuits*, 26(3):277–282, 1991.

[92] Azriel Rosenfeld. Picture processing by computer. *ACM Computing Surveys (CSUR)*, 1(3):147–176, 1969.

[93] Edward Rosten, Reid Porter, and Tom Drummond. Faster and better: A machine learning approach to corner detection. *IEEE transactions on pattern analysis and machine intelligence*, 32(1):105–119, 2010.

[94] Carl Sechen. *VLSI placement and global routing using simulated annealing*, volume 54. Springer Science & Business Media, 2012.

[95] On Semiconductor. Image sensor terminology. tnd6116/d. `https://www.onsemi.com/pub/Collateral/TND6116-D.PDF`, 2014.

[96] O. Shacham and M. Reynders. Pixel visual core: Image processing and machine learning on pixel 2. `https://blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/`, 2017.

[97] Ofer Shacham. *Chip multiprocessor generator: automatic generation of custom and heterogeneous compute platforms*. Stanford University, 2011.

[98] Yair Siegel. The path to embedded vision & ai using a low power vision dsp. In *Hot Chips 28 Symposium (HCS), 2016 IEEE*, pages 1–28. IEEE, 2016.

[99] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[100] Deshanand P Singh and Stephen D Brown. The case for registered routing switches in field programmable gate arrays. In *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, pages 161–169. ACM, 2001.

[101] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE transactions on computers*, (5):465–481, 2000.

[102] Gideon P Stein, Elchanan Rushinek, Gaby Hayun, and Amnon Shashua. A computer vision system on a chip: a case study from the automotive domain. In *Computer Vision and Pattern Recognition-Workshops, 2005. CVPR Workshops. IEEE Computer Society Conference on*, pages 130–130. IEEE, 2005.

[103] Eran Steinberg, Yury Prilutsky, Peter Corcoran, and Petronel Bigioi. Digital image processing using face detection information, August 11 2009. US Patent 7,574,016.

[104] Russell Tessier, Kenneth Pocek, and Andre DeHon. Reconfigurable computing architectures. *Proceedings of the IEEE*, 103(3):332–354, 2015.

[105] Russell George Tessier. *Fast place and route approaches for FPGAs*. PhD thesis, Massachusetts Institute of Technology, 1999.

[106] Steven Trimberger, Dean Carberry, Anders Johnson, and Jennifer Wong. A time-multiplexed fpga. In *Field-Programmable Custom Computing Machines, 1997. Proceedings., the 5th Annual IEEE Symposium on*, pages 22–28. IEEE, 1997.

[107] William Tsu, Kip Macy, Atul Joshi, Randy Huang, Norman Walker, Tony Tung, Omid Rowhani, Varghese George, John Wawrzynek, and André DeHon. Hsra: high-speed, hierarchical synchronous reconfigurable array. In *Proceedings of the 1999 ACM/SIGDA seventh international symposium on Field programmable gate arrays*, pages 125–134. ACM, 1999.

[108] Artem Vasilyev, Nikhil Bhagdikar, Ardavan Pedram, Stephen Richardson, Shahar Kvatinsky, and Mark Horowitz. Evaluating programmable architectures for imaging and vision applications. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.

[109] Arthur H Veen. Dataflow machine architecture. *ACM Computing Surveys (CSUR)*, 18(4):365–396, 1986.

[110] Ganesh K Venayagamoorthy and Venu Gopal Gudise. Fpga placement and routing using particle swarm optimization. 2004.

[111] Francisco-Javier Veredas, Michael Scheppler, Will Moffat, and Bingfeng Mei. Custom implementation of the coarse-grained reconfigurable adres architecture for multimedia purposes. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 106–111. IEEE, 2005.

[112] Paul Viola and Michael J Jones. Robust real-time face detection. *International journal of computer vision*, 57(2):137–154, 2004.

[113] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374. IEEE Computer Society, 2010.

[114] Wikipedia. Computational photography. `https://en.wikipedia.org/wiki/Computational_photography`.

[115] Wikipedia. Division algorithm. `https://en.wikipedia.org/wiki/Division_algorithm#Restoring_division`.

[116] Wikipedia. Tabula. `https://en.wikipedia.org/wiki/Tabula_(company)`, 2018.

[117] Steven Joseph Edward Wilton, Jonathan Rose, and Zvenko Vranesic. Architectures and algorithms for field-programmable gate arrays with embedded memory. *University of Toronto, Toronto, Ont., Canada*, 1997.

[118] David Witt. Omap4430 architecture and development. In *2009 IEEE Hot Chips 21 Symposium (HCS)*, pages 1–16. IEEE, 2009.

[119] Yu-Liang Wu and Malgorzata Marek-Sadowska. Orthogonal greedy coupling: a new optimization approach to 2-d fpga routing. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, pages 568–573. ACM, 1995.

[120] Xilinx. Virtex-5 fpga user guide. ug190 (v5.4). `https://www.xilinx.com/support/documentation/user_guides/ug190.pdf`, 2012.

[121] Xilinx. 7 series fpgas configurable logic block. ug474 (v1.8). `https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf`, 2016.

[122] Xilinx. 7 series fpgas clocking resources. ug472 (v1.14). `https://www.xilinx.com/support/documentation/user_guides/ug472_7Series_Clocking.pdf`, 2018.

[123] Vivado-HLS Xilinx. Vivado design suite user guide-high-level synthesis, 2014.

[124] Tianfan Xue, Michael Rubinstein, Ce Liu, and William T Freeman. A computational approach for obstruction-free photography. *ACM Transactions on Graphics (TOG)*, 34(4):79, 2015.

[125] Xuan Yang, Jing Pu, Blaine Burton Rister, Nikhil Bhagdikar, Stephen Richardson, Shahar Kvatinsky, Jonathan Ragan-Kelley, Ardavan Pedram, and Mark Horowitz. A systematic approach to blocking convolutional neural networks. *arXiv preprint arXiv:1606.04209*, 2016.

[126] Zhen Yang, Anthony Vannelli, and Shawki Areibi. An ilp based hierarchical global routing approach for vlsi asic design. *Optimization Letters*, 1(3):281–297, 2007.

[127] Andy Ye and Jonathan Rose. Using bus-based connections to improve field-programmable gate array density for implementing datapath circuits. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 3–13. ACM, 2005.