POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

Integration of a Precision Scalable Multiplier in the AHA CGRA Framework

Advisors
Prof. Mario Roberto Casu
Dr. Edward Manca

Candidate
Boris Assenov Alexiev

Academic Year 2025-2026

Summary

Modern neural networks continue to expand in parameter count and number of layers, pushing the limits of training and inference hardware. Since convolutional, fully connected, and attention layers are dominated by multiply–accumulate (MAC) operations, the throughput and energy cost of multiplication largely determine overall system efficiency. Two techniques are especially effective: reducing numeric precision via quantization of weights and activations, and exploiting parallelism in HW architectures to perform many multiplications concurrently with reduced precision of operands.

This thesis explores the latter technique, with the integration of a reconfigurable multiplier into the Agile Hardware Approach (AHA) design flow, targeting coarse-grained reconfigurable arrays (CGRAs). The integrated unit allows for precision-scalability, allowing a more efficient hardware utilization, as the bit-width required by the workload varies. The goal is to accelerate multiplication-intensive kernels, typical of neural network layers, by executing more than one operation in parallel with the same HW unit, when the bit-width is reduced.

Several architectural variants are examined: accumulation performed inside the processing element (PE) versus in an external register file; splitting outputs into low- and high-bit paths to improve data reuse and routing; and input reordering to enable operand decomposition and independent sub-word parallelism. For each option, the thesis outlines microarchitectural trade-offs, placement within the PE datapath, and expected impacts on area, frequency, and utilization.

Although some variants are not presently feasible within the AHA framework, they reveal promising directions and design patterns for future exploration. The current AHA mapping methodology limits multiple micro-operations per instruction within a PE. To mitigate this constraint, the work proposes practical compiler-flow modifications. In particular, targeted changes to rewrite rules and mapping files allow the toolchain to recognize scalable-precision multiply operators and emit the required control sequences without overhauling the broader infrastructure. These adjustments preserve compatibility with the existing AHA workflow while enabling precision-scalable operations.

An experimental implementation validates the approach: custom Verilog for the PE, paired with updated rewrite rules and mapping files, integrates cleanly into

the AHA flow. RTL simulations confirm functional correctness of reduced-precision multiplication and indicate favorable throughput and efficiency trends, suggesting benefits for future NN accelerators based on this approach, even though several variants remain impractical under current constraints. Collectively, the results provide a path for introducing reconfigurable, precision-scalable arithmetic into CGRAs and identify existing tradeoffs for subsequent work.

Acknowledgements

I would like to express my gratitude to the research group led by Professor Mario Roberto Casu, of which I was a member, for their assistance with my research. I am especially grateful to Dr. Edward Manca, who always found time for me.

Contents

Chapter 1

Introduction

1.1 Context

Multiplication and accumulation operations are at the core of nearly all modern computational workloads, particularly in machine learning and neural networks. As network sizes grow, the efficiency of these multiplications directly impacts both performance and energy consumption. One of the best ways to improve the efficiency is to quantize these weights and activations and to parallelize the operations. Conventional hardware accelerators, such as GPUs and ASICs provide high throughput, but often lack flexibility to experiment with reduced-precision operations. Coarse-Grained Reconfigurable Arrays (CGRAs) offer a balance between flexibility and performance, enabling reconfigurations while maintaining energy efficiency, higher than that of general-purpose processors, making them perfect for employing these new accelerator architectures.
However, one of the main challenges in CGRA-based acceleration lies in optimizing arithmetic units, particularly multipliers, to support variable precision without significant hardware overhead. Reduced-precision arithmetic can significantly decrease computation time, data transfer, and power consumption, while maintaining sufficient accuracy for neural network inference. Therefore, integrating precision-scalable multipliers and efficient accumulation mechanisms within CGRAs is an important step toward high-performance and low-power neural network accelerators.

1.2 Goals of the study

The main objective of this study is to evaluate methods for implementing reduced-precision arithmetic and reconfigurable architectures within the Agile Hardware Approach (AHA) framework for Coarse-Grained Reconfigurable Arrays (CGRAs).

The specific goals of the study are as follows.

- Analysis of precision-scalable multiplier architectures
- Investigation of implementation strategies
- Experimental verification of the AHA-generated PE
- Identification of opportunities for future improvements

1.3 Overview

1.3.1 Chapter 1 – Introduction

The first chapter focuses on the growing demand for flexible, high-performance hardware accelerators capable of supporting computational workloads such as neural networks. It presents the goals of the study-namely exploring how reduced-precision arithmetic can be efficiently implemented within the Agile Hardware (AHA) framework for CGRAs.

1.3.2 Chapter 2 – CGRAs

This chapter provides a detailed background on CGRA architectures. It describes their structure, topologies, and the role of Processing Elements (PEs) and Arithmetic Logic Units (ALUs) within them. Then it compares their characteristics with ASICs and multicore processors. The second half introduces the Stanford Agile Hardware Approach (AHA) methodology and its Onyx CGRA implementation. Key components such as domain-specific languages, the MetaMapper framework, rewrite rules, and PE hardware generation are explained. The chapter concludes by examining the application of CGRAs to neural network acceleration and the potential for efficiency gains through quantization.

1.3.3 Chapter 3 – Reduced-Precision Reconfigurable Multiplier

Chapter 3 introduces the concept of precision-scalable arithmetic and explores several multiplier architectures suitable for reconfigurable systems. The Baugh-Wooley

and its derivative, the STAR multiplier, are analyzed in detail. The chapter further discusses the integration of multiply–accumulate (MAC) operations and the implications of using reduced precision for computation throughput, hardware utilization, and energy efficiency. These architectures provide the theoretical and structural foundation for the implementation work in Chapter 4.

1.3.4 Chapter 4 – ST and SA Operations in AHA

This chapter investigates how STAR-derived reduced-precision operations (ST and SA) can be integrated into the AHA CGRA framework despite its mapping and design flow constraints. Several implementation strategies are evaluated, including placing the accumulation register inside or outside the PE, dividing outputs into low/high bit sections, and reordering inputs for operand decomposition. The chapter also proposes workarounds to AHA's limitations through targeted modifications. Experimental implementations demonstrate that reduced-precision arithmetic can be effectively embedded in AHA without major changes. Simulation results confirm functional correctness and indicate potential performance improvements for neural network applications.

1.3.5 Chapter 5 - Conclusions and future work

The final chapter summarizes the findings and evaluates the feasibility of introducing reduced-precision operations into the AHA CGRA. It concludes that precision-scalable arithmetic units can enhance computational efficiency but there are still problems that need to be solved. Future work suggestions include extending the study to other CGRAs and improving the workarounds of implementing the precision-scalable multiplier into AHA, despite the limitations.

Chapter 2

CGRAs

2.1 CGRA Structure and Topologies

Modern Coarse-Grained Reconfigurable Architectures (CGRAs) are programmable accelerators composed of an array of Processing Elements (PEs), memories, and a configurable interconnect network. Unlike fine-grained FPGAs, which operate at the bit level, CGRAs execute word-level operations and therefore achieve higher energy efficiency and clock frequency while still retaining flexibility. Their structure enables spatial unroll of compute kernels, where operations are mapped onto PEs and data flows through programmable switch-boxes. These CGRAs, however, come at a cost - in order to benefit from their functionalities, a custom compiler is needed, as every CGRA is different.

2.2 PEs and ALU within a CGRA

The processing elements (PEs) can be considered the most critical components of a CGRA, as they perform the actual computations within the architecture. A typical PE includes an arithmetic–logic unit (ALU), an output register, and several multiplexers that determine the ALU's inputs and the PE's output. Some designs also incorporate a feedback register, allowing the ALU's previous result to be reused in the next operation. The configuration of a PE is defined by the selection signals for these multiplexers and the control signals that set the ALU's operating mode. Figure 2.1 shows a simple Processing element. Its main features include a selectable input, where in this case there are A and B. They are provided via two multiplexers, selecting from connections from neighboring PEs or from an internal register. The selection of said multiplexers is done via the configuration signal provided to the PE. These two operands are then provided to an Arithmetic and Logic unit - the part which does the actual computation. Multiple different ALU architectures exist, but in most cases the ALU is stateless - it only does a combinatorial function without

inner clocking. The operation done by the ALU is determined by the configuration signal given to the PE.

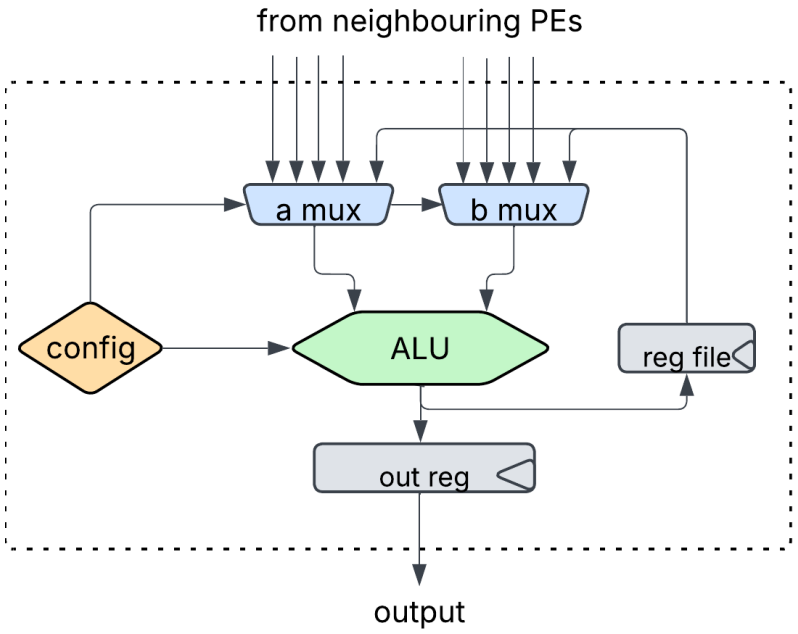

Figure 2.1: An example of a PE

The arithmetic and logic unit is the core element of the PE. It can be more basic - perform only simple operations like addition, subtraction, bitwise logic operations, etc. In some advanced CGRAs the ALU can have a lot of features - for example floating point operations, large multiplications etc. These two extremes are something like a reduced instruction set computer (RISC) and a complex instruction set computer (CISC). It is important to note that a CGRA with more simple PEs can have a lot more of them in the same area compared to one with more sophisticated PEs. This distinction can also be thought of as how "coarse grained" the array is.

2.3 FPGA to CGRA Evolution

Field-Programmable Gate Arrays (FPGAs) were among the first forms of reconfigurable computing, allowing hardware behavior to be defined after fabrication. Originally introduced as a safer, lower-cost way to prototype ASICs, they rely on fine-grained reconfigurability implemented through large numbers of LUT-based logic elements. Although FPGA prototypes typically run one to two orders of magnitude slower than their ASIC counterparts, they remain essential design and verification tools. After some time FPGAs began to be viewed as compute devices in their own right, with interest in general-purpose acceleration. However, several

limitations quickly became apparent

- Long compilation times.
- Poor mapping of certain arithmetic operations (e.g., multipliers).
- Relatively low clock frequencies.

Many of these challenges remain today and have caused the exploration of alternative reconfigurable architectures that offer higher performance and efficiency. One such architecture, the subject of this thesis, is the CGRA.

Increasing the granularity of the elements that are reconfigured, allows larger, more specialized computational units to be built. In this way we can improve overall performance, particularly the maximum clock frequency, as there is much less delay due to the long paths in a FPGA. These paths include multiple multiplexers, LUTs, etc. leading to reduced timing margins, and higher power consumption for equivalent operations.

Coarser-grained elements also require far less configuration data, making reconfiguration significantly faster and enabling effective time-sharing across multiple contexts. In a CGRA, this facilitates parallel execution, as processing elements (PEs) can be configured to operate on different parts of a workload simultaneously, parallelizing the load. Moreover, coarse-grained units allow operations that map inefficiently to traditional FPGAs—such as integer multiplications—to be integrated directly and implemented more efficiently.

However, all these advantages come with a cost, the need for a good mapping onto the CGRA in order to use its functionalities efficiently.

2.4 CGRA compared to other architectures

The CGRA has been compared to a basic FPGA in the previous section. In this one a comparison is made to other architectures used in similar contexts.

2.4.1 ASIC

A CGRA can be thought of being somewhere in the middle between an Aplication-specific IC and a microprocessor from a reconfigurability point of view. It seeks to combine the advantages of both ends of the spectrum: unlike an ASIC, a CGRA is not fully hard-coded for a single task, yet it is more specialized and efficient than a general-purpose processor. While an ASIC may achieve higher performance for a very specific application, a CGRA retains flexibility, allowing the same hardware to support multiple tasks. The main advantages of an ASIC over a CGRA are summarized.

- Better performance: Custom-designed circuits are optimized for a specific application, achieving the highest possible speed and throughput.
- Lower power consumption: Efficiency is maximized because only the required logic is implemented
- Minimal area overhead: Only essential circuitry is included, resulting in smaller silicon footprint
- High reliability: Optimized and verified for a single purpose, reducing potential sources of error or variability.

CGRAs though have some important advantages, which are summarized.

- Flexibility: Unlike ASICs, CGRAs are not hard-coded for a single task
- Faster development time: CGRAs can be deployed without the long design and fabrication
- Adaptability : useful when workloads evolve over time
- Lower development risk : Errors in design or changing requirements are supported

2.4.2 Multi-core processors

At the other end of this spectrum are multi-core processor solutions. In some cases they can outperform CGRAs in several ways. They benefit from high clock speeds, large caches, and advanced vector units, which accelerate dense linear algebra operations. Additionally, their software ecosystems and dynamic task scheduling allow CPUs and GPUs to efficiently handle mixed or irregular tasks that CGRAs may struggle with. However for highly parallelizeable tasks, such as NN workloads or especially multiplications, a CGRA can outperform a multi-core processor with a fraction of the cost. However, this makes the efficient mapping and optimized hardware of a CGRA even more important and is the reason a well-designed CGRA methodology is essential.

2.5 Agile Hardware Approach Methodology

This section focuses on a specific CGRA development methodology used for this thesis. Some of the figures are taken from the AHA paper[3].

2.5.1 AHA Overview

The slowdown of Moore's Law and the end of Dennard scaling have shifted performance and energy improvements away from general-purpose scaling and toward hardware specialization[3]. Modern Systems-on-Chip (SoCs) increasingly integrate numerous domain-specific accelerators to meet the demands of workloads such as machine learning and image processing. However, these application domains evolve rapidly, and sustaining high performance requires that the application, accelerator architecture, and compiler are developed in sync rather than in isolation. AHA employs domain-specific languages (DSLs) to specify the accelerator architecture. These DSLs generate both the Register-Transfer Level hardware description and all required compiler artifacts, enabling consistent evolution of the hardware-software stack. This integrated methodology significantly accelerates design iteration and supports flexible architectural exploration.

Using the AHA framework, several CGRA chips including *Amber* and *Onyx* have been successfully designed and fabricated. These accelerators demonstrate high efficiency across dense image processing kernels, neural network inference workloads and others.

2.5.2 Existing Approach to Accelerator Design

The traditional way to design hardware accelerators typically follows a waterfall-like design approach. Engineers begin by analyzing the target application to identify computational needs, then design a custom accelerator optimized for them, and finally adapt the compiler and software stack to support the new hardware. While this process can produce highly efficient solutions, it is time-consuming and rigid. Any change in the application or hardware often necessitates manual redesign and testing across multiple components, making it difficult to keep up with rapidly changing workloads, especially in areas like machine learning and high-performance computing. Figure 2.2 shows a typical waterfall-like design approach.

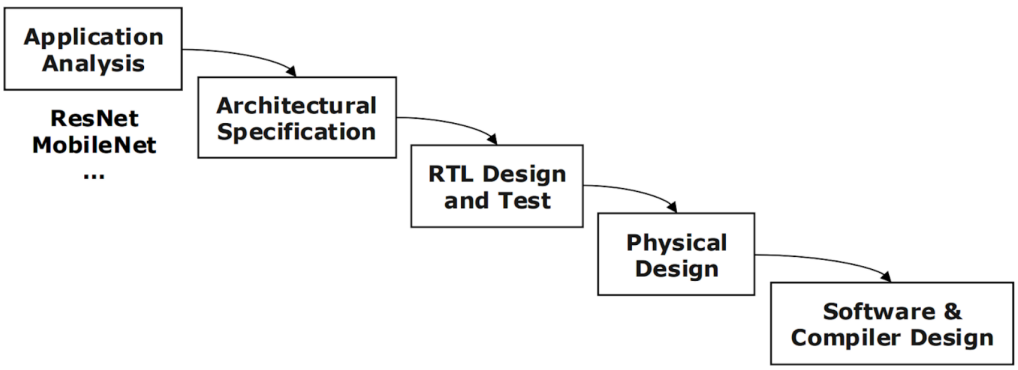

Figure 2.2: Waterfall development approach

2.5.3 The AHA Agile Approach

Agile design is an iterative approach to hardware and software development that enables rapid prototyping and continuous feedback. Unlike traditional waterfall methods, it allows designers to quickly explore, modify, and optimize systems throughout the development process. These features make it the reason it is the way AHA works. The agile approach is especially suited to CGRA development, because in this case a software compiler needs to evolve alongside the hardware of the CGRA. In this way a good compiler that takes advantage of the hardware functions is easier to maintain and improve. Figure 2.3 shows the iterative process. This feedback loop accelerates design-space exploration and ensures that the hardware and software remain synchronized as the architecture evolves[3]. Unlike traditional design flows, AHA promotes incremental changes to the architecture and automatic regeneration of the corresponding tools.

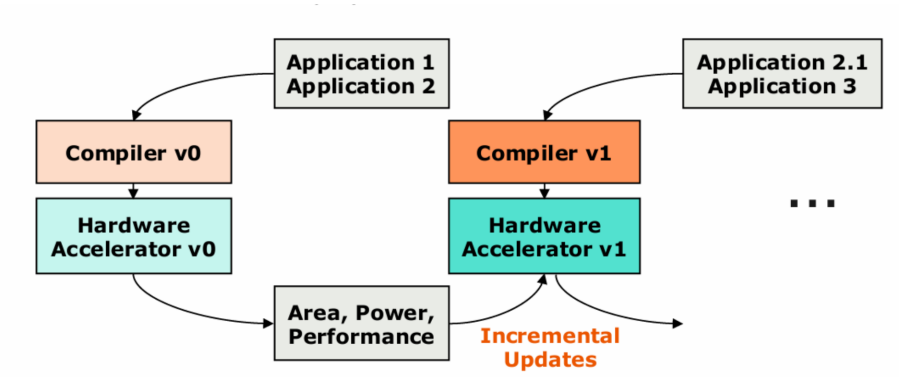

Figure 2.3: Agile development approach in CGRA

2.5.4 Domain specific languages in AHA

One of the main advantages of AHA is its “single source of truth" methodology. When a modification is done to the CGRA it only needs to be done in a single place. There are three different domain specific languages used in AHA, each for its own respective components of the CGRA. The processing elements are written in PEak, memories in Lake, and interconnect in Canal. The compiler for each DSL generates both the hardware Register-Transfer Level (RTL) and the collateral necessary for mapping applications onto the hardware, ensuring continuous end-to-end system functionality. These component DSLs are built upon magma, a lower-level hardware generation DSL embedded in Python, which serves as their host language[3]. In this thesis only the PEAK language was explored in more detail, as it is the dsl responsible for PEs. The other two dsls are only briefly mentioned.

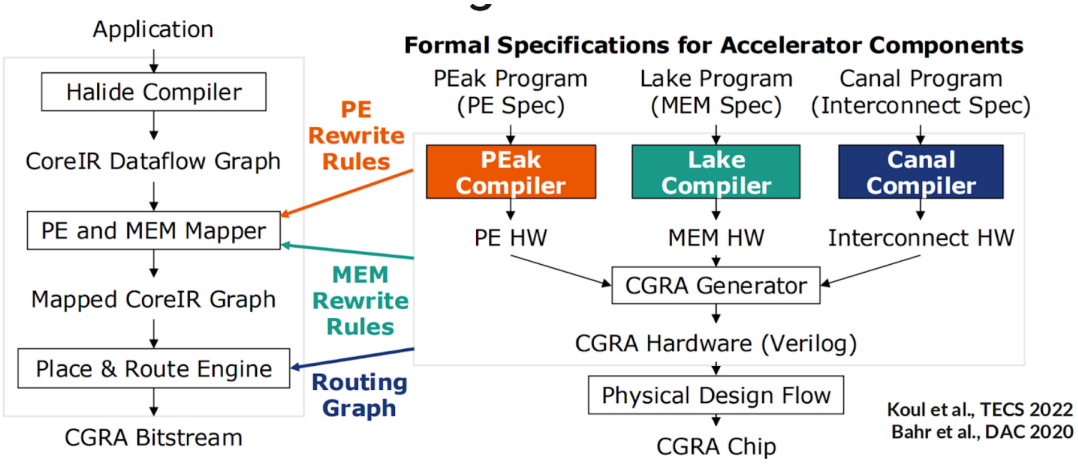

Figure 2.4: DSLs use in AHA

Figure 2.4 is taken from [3] and shows how the DSL specifications are used within the AHA flow.

PEAK

A PEak design file defines a PE's instruction set architecture (ISA), declares its state, and describes the semantics of each instruction as a function from inputs and current state to outputs and next state. PEak programs can be compiled into a functional model, RTL hardware, or a formal SMT model, which is used to automatically generate compiler rewrite rules(explained later in detail) targeting the PE.

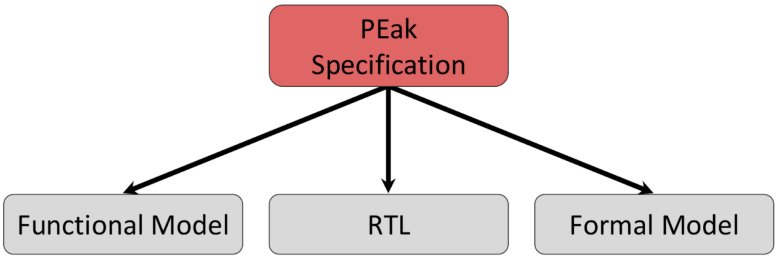

Figure 2.5: PEak

A Lake memory specification is based on the streaming memory abstraction used in the application compiler. It combines storage, address generation, and control logic for efficiency, and defines the physical unit's type (registers or SRAM), depth, width, and ports[3].

Canal specifies the CGRA interconnect as a directed graph, supporting heterogeneous PEs and memory cores with varying numbers of inputs and outputs. Like

PEak and Lake, it generates both hardware and compiler collateral, including the routing graph for the place-and-route tool and the configuration bitstream for the hardware[3].

2.5.5 PE Hardware generation

PEak uses Magma to compile PE specifications into RTL Verilog. PEak extends Magma's sequential circuit syntax with rich types that describe ISAs via Magma's type protocol, allowing new types to be interpreted as Magma's built-in primitives[3]. The rtl generated by the aha garnet script is mostly comprised of multiplexers, adders and bitwise operations. Lowering a PEak specification to Magma is straightforward and preserves the designer's functional intent. The ___call___ method defines the state transition executed on every clock edge. PEak encourages high-level specifications, leaving low-level details such as resource sharing, clock gating, and data gating to optimization passes in the compiler. The Magma intermediate representation, CoreIR, is SMT-based, enabling formal equivalence checking across compiler passes. As explained later a possible direction worth exploring is to try to exchange the generated rtl with a custom architecture, in order to use a more optimal solution. This could be useful in cases where the functionality of the PE is not straight-forward and decomposing it into simple, synthesizable operations can be done in many different ways. For example the multiplier in section **??** can be realized in many different ways. As PEAK is only a formal specification, it is not possible to differentiate between these different possible architectures at this stage(as their difference is only at synthesis level and not logical).

2.5.6 Meta Mapper, rewrite rules

The mapping stage in the application compiler relies on rewrite rules -Figure2.6 that describe how subsets of CoreIR dataflow graphs correspond to hardware PEs[3]. The PEak compiler transforms the ___call___ method into a normal form where each name is assigned at most once, there is a single return statement at the end, subcomponents are instantiated only once, and all `if` blocks are eliminated. This process begins with a standard SSA pass, during which return statements are replaced with assignments to new names. Next, the bodies of `if` blocks are inlined, and ternary expressions are inserted to select the final value for each assigned name. The return value is then expressed at the end of the function using a nested ternary expression covering all possible return paths. Boolean operators are replaced with their bitwise equivalents, and ternary expressions are converted to calls to the `ite` method (e.g., `x if c else y` becomes `c.ite(x, y)`).

Once the function is in this normalized form, applying ___call___ to abstract SMT variables—analogous to its application on concrete Python variables produces a symbolic execution of the circuit. This symbolic execution can then be used to

generate rewrite rules for a CoreIR IR node using a quantified SMT query:

$$\exists inst\ \forall inputs : IRNode(inputs) == PE(inst, inputs)$$

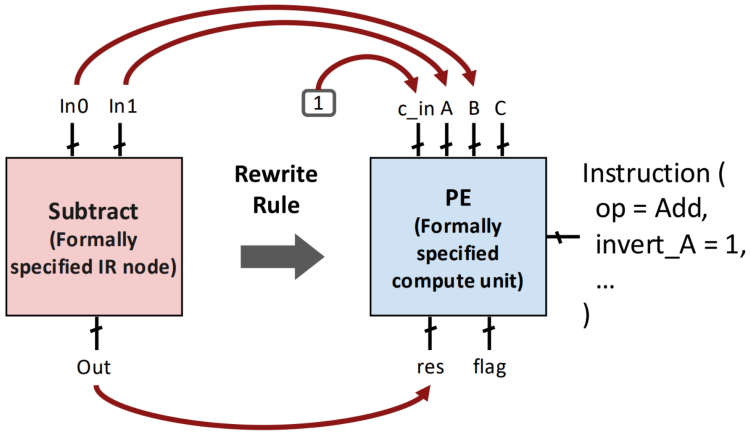

Figure 2.6: overview of the AHA rewrite rules

More simply said, a rewrite rule is the configuration needed for a PE to act in such a way that is performs the needed operation defined in the rule. Having these solved rules, the MetaMapper has all it needs to map a specific application onto the CGRA. The script transforms the functionality (for example state transitions) into configured PEs (see figure 2.7).

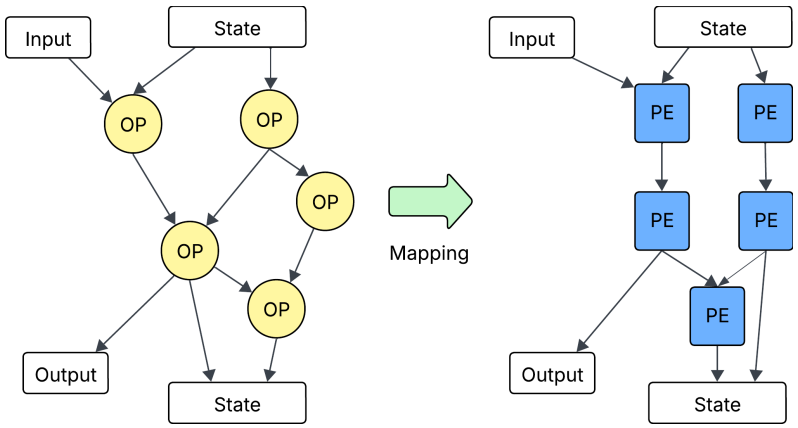

Figure 2.7: AHA mapping

In AHA these resulting configurations are put into fifos that give the appropriate instructions to each PE, mux and other configs as the application executes.

2.5.7 Design flow of the AHA approach

The steps for modifying an iteration of the designed CGRA to the next version are the summarized in figure 2.8. Here they are listed in the order they were done during the experimentation from the subsequent chapters.

- Modifying the PEAK formal specification
- Generation of the HDL of the CGRA with garnet
- Defining a formal operation-it should take advantage of the new functionality
- Running the rewrite rule script to generate the rule
- Optional - modifying the Halide app
- Mapping the app onto the CGRA
- Place and route
- Testing - hardware simulation with a generated testbench

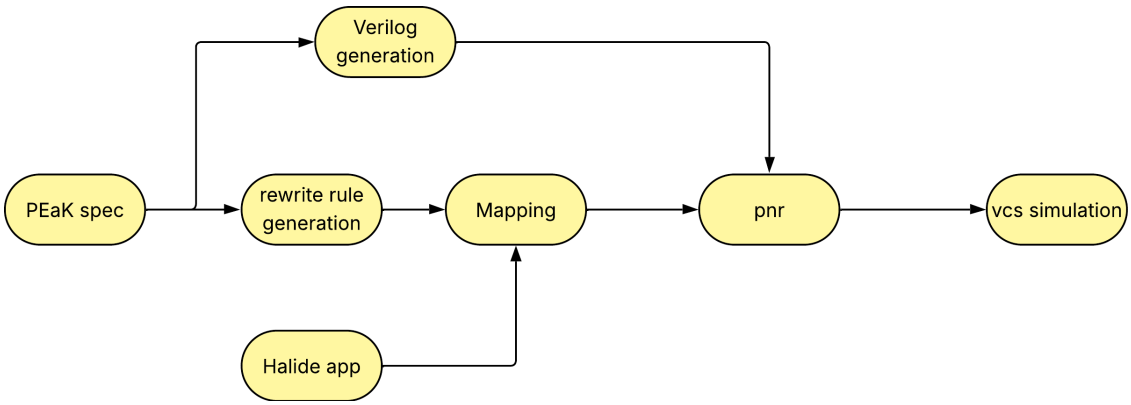

Figure 2.8: overview of the AHA design flow

The magma circuits are tested with the fault Python package using the function call syntax. Designers directly reuse functional tests for the hardware description as well, and fault generates a test bench for the design using a hardware simulator.[3]

2.5.8 Halide use within AHA

Writing high-performance code on modern machines requires not just optimizing inner loops, but reorganizing computations globally to exploit parallelism, especially in image processing pipelines where intermediate stages do little work relative to memory costs [4]. The requirements of image processing are somewhat similar to

those of NN computation as they both require a high number of operations. Traditional programming models confuse the algorithm with scheduling decisions, making high-performance code complex and hard to maintain. Halide is a programming language that separates the algorithm from its schedule, allowing programmers to explore different computation organizations while the compiler generates efficient loop nests for CPUs, GPUs, and specialized processors. This approach is the reason it is used in AHA to program applications.

2.6 CGRA for Neural Networks

2.6.1 NN Types and Layers

Neural networks consist of different layers, but the convolutional, fully connected, and attention ones are important to this work as they can benefit from more efficient multiply–accumulate (MAC) operations. The high density of MACs makes the choice of underlying hardware and its ability to efficiently execute these operations critical for overall performance.

Fully connected

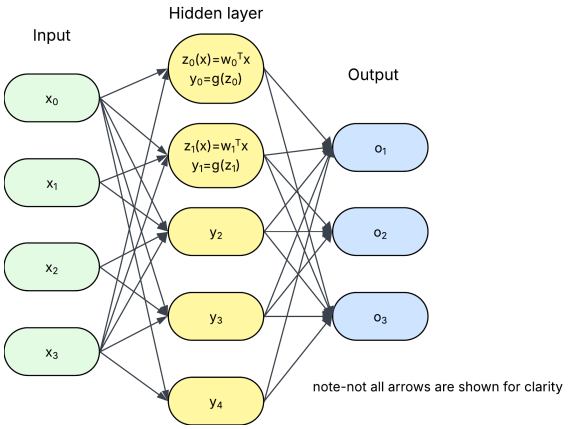

Figure 2.9: Fully-connected layer in NN

In fully-connected layers each node is connected to all nodes from the previous layer. Each connection has an associated weight, and computing the output involves multiplying each input by its corresponding weight. These products are then summed (accumulated) to produce the node's activation. The process of computing these layers includes a high number of multiply–accumulate operations.

Convolutional

Some neural networks (for example computer vision) often contain too many parameters to train efficiently. Large numbers of nodes and consequently weights in the hidden layers demand large amounts of training data. This is the reason for creating convolutional layers in NNs. Drawing an analogy from neurophysiology, visual neurons in deep layers respond to:

- Simple patterns within small receptive fields
- Complex patterns within larger receptive fields, independent of precise position

To exploit this, Convolutional Neural Network layes use a specialized connectivity structure for more efficient pattern recognition

- Full neuron-to-neuron connections are replaced with convolutions
- Pooling layers further limit connectivity to reduce parameters and improve efficiency

One output pixel in a convolutional layer is computed as the dot product of an image window and a filter, which operates only on a subset of pixels (local connectivity). Each filter in a CNN is a 3D tensor composed of 2D kernels, and its weights are shared among all neurons in the layer to reduce the amount of training data required. Convolution filter weights are often normalized so that their sum equals 1, helping to prevent overflows during computation.

A convolution operation is essentially a series of multiply–accumulate (MAC) operations, where each filter weight is multiplied by the corresponding input pixel value. The results of these multiplications are then summed to produce a single output value for the convolution.

Attention layer

Attention layers compute weighted interactions between elements of an input sequence, allowing a model to focus on the most relevant information. The core computation is the scaled dot-product attention, which forms three projections - queries (Q), keys (K), and values (V)and computes:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.$$

This operation is also dominated by matrix multiplications (for QK^T and the subsequent multiplication with V). Because attention evaluates global interactions across the entire sequence, it is more computationally intensive than convolution or fully connected layers, making hardware efficiency especially important.

2.6.2 NN Layer Mappings

Mapping neural network layers onto a CGRA benefits greatly from efficient multiplication and accumulation units.Coarse-grained reconfigurable architectures can exploit spatial parallelism and pipelining to accelerate these operations, enabling better energy efficiency and throughput compared to general-purpose multicore processors.

2.6.3 Enhancing the efficiency of neural network models

One of the ways to improve the efficiency of NN layers that heavily use multiplication and accumulation is to quantize the weights and activations. When relying on fixed power budget this technique can improve the performance of a NN by allowing it to have more layers/parameters for the same power consumption in calculations.

Quantization in NNs

Quantization maps input values from a large (often continuous) set to a smaller (finite) set, using rounding or truncation. While many recent neural network quantization methods are connected to classical approaches, NNs introduce unique challenges and opportunities: inference and training are computationally intensive, and models are often heavily over-parameterized, allowing high reduction of bit precision without hurting accuracy. Neural networks also work well with extreme discretization, meaning a quantized model can differ significantly from the original yet still generalize well. This flexibility, combined with the layered structure of NNs, motivates mixed-precision quantization strategies, where different layers can use different precisions based on their impact on the loss of information. These factors distinguish modern NN quantization from traditional quantization approaches focused on minimizing error in the signal itself. To quantize neural network weights and activations into a finite set of discrete values, we define a quantization function that maps floating-point numbers to a lower-precision range. An example quantization function is:

$$Q(r) = \text{Int}\left(\frac{r}{S}\right) - Z,$$

where Q is the quantizer, r is a real-valued activation or weight, S is a real-valued scaling factor, and Z is an integer zero-point. The function $\text{Int}(\cdot)$ applies rounding, producing an integer. This yields a mapping from real values to uniformly spaced quantization levels, a process known as *uniform quantization*. Real values can be approximately reconstructed through a dequantization operation:

$$\tilde{r} = S\Big(Q(r) + Z\Big).$$

Because rounding introduces error, the recovered value $\tilde{r}$ does not exactly match the original r, leading to quantization-induced inaccuracy.

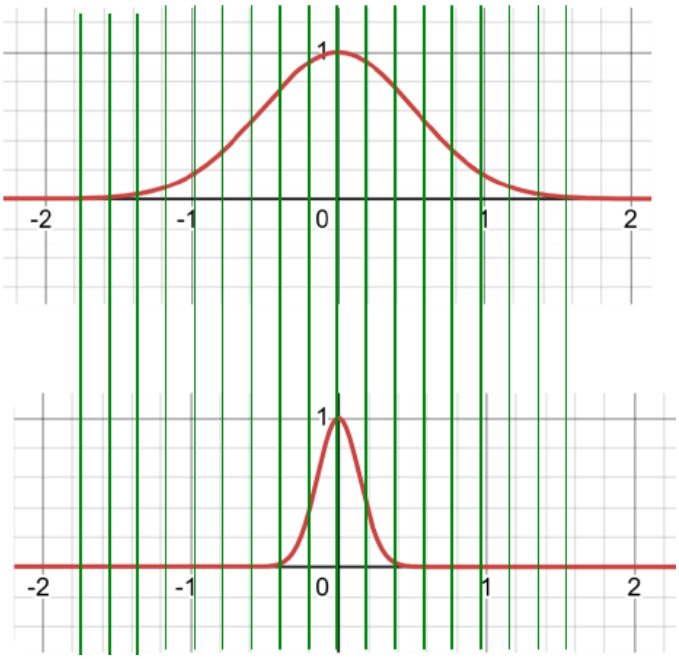

Figure 2.10: Quantization vs spread of values

Figure 2.10 shows how important it is to choose a good quantization in order to have as little loss of information in NN layers as possible.

Different granularities of quantization can be applied to different channels and layers of a neural network. A key distinction among quantization methods is the granularity at which the clipping range for weights is computed.

Layerwise Quantization

In this approach, the clipping range is determined from all parameters within an entire layer, and a single range is applied to all convolution kernels. While easy to implement, it may be suboptimal in accuracy when kernels exhibit very different value distributions. Narrow-range kernels may lose significant information because the chosen resolution is driven by a wider-range kernel.

Channelwise Quantization

A more refined strategy computes a separate clipping range for each channel. Although computationally more expensive, this approach provides better per-channel resolution and reduces information loss.

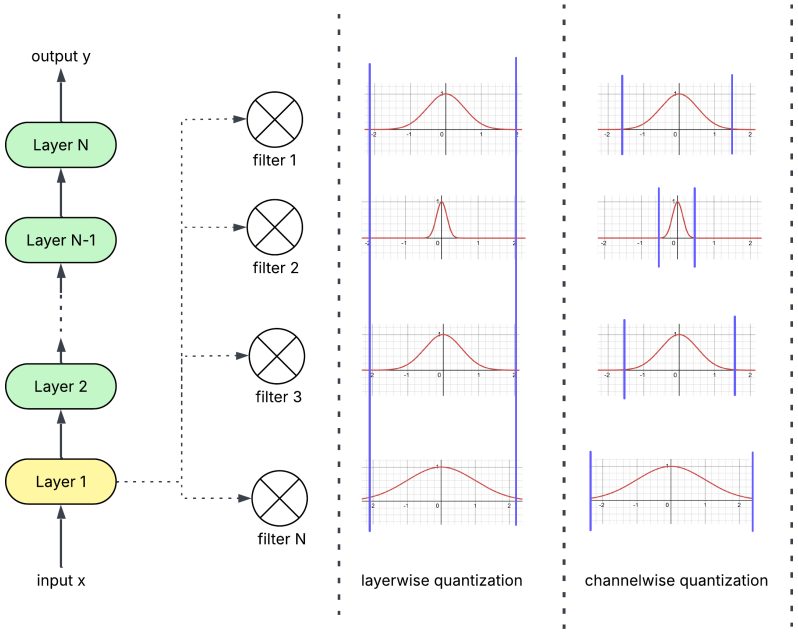

Figure 2.11: Layerwise vs channelwise Quantization

Figure 2.11 shows the advantage of channelwise quantization. The different spreads of weights/activations are all well represented and a lower loss of data is happening.

Chapter 3

Reduced precision reconfigurable multiplier

Chapter 3 focuses on the inner workings of a reconfigurable reduced precision multiplier. It starts by introducing a simple BW multiplier in section 3.1 and shows what has been modified to support different summation modes in the reconfigurable architecture.

3.1 Precision-Scalable MAC

3.1.1 Baugh-Wooley multiplier

A standard BW multiplier is a grid of 1by1 bit multipliers (essentially AND gates) where each block computes the partial product (PP) between a different pair of bits of the two 16-bit input operands using an AND gate. Then, by using a Full Adder (FA), the product is summed together with 2 other contributions - the input sum Si and the carry Ci bits coming from the previous row of PPs, and it provides the output sum So and carry Co bits to the blocks of the next row of the PPM. The sixteen So bits exiting from the right-most column of the PPM represent the least significant part of the multiplier's output. The most significant part is instead obtained by adding through a 16-bit Ripple-Carry Adder (RCA), the So and Co output bits exiting from the last row of the PPM. The concatenation of these two produce the 32 bit result.

In order to support signed 2s complement multiplication, the grid has two types of cells. The inner ones have an AND between A and B and the outer ones have a NAND. Th this way the second variant inverts the partial product. In Figure 3.1 the two types of cells are shown.

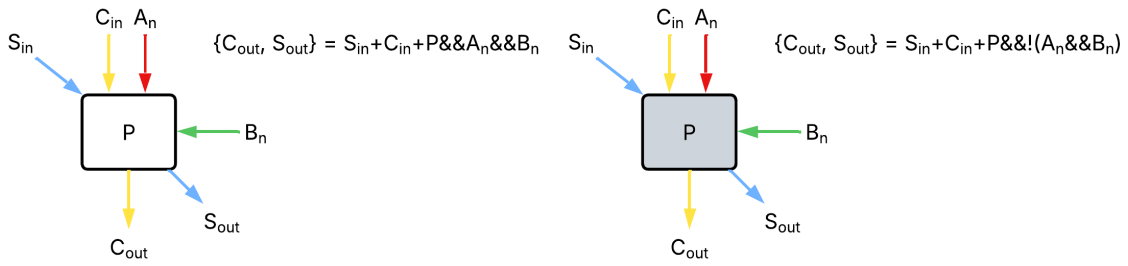

Figure 3.1: The building blocks of the Baugh-Wooley multiplier.

The topology of partial product blocks forms a grid where one operand is given on the rows and the other in the columns. The carry propagates down and the sum on the diagonals (right-down) of the array. The output result is formed from Sum out bits of the outer-most blocks.

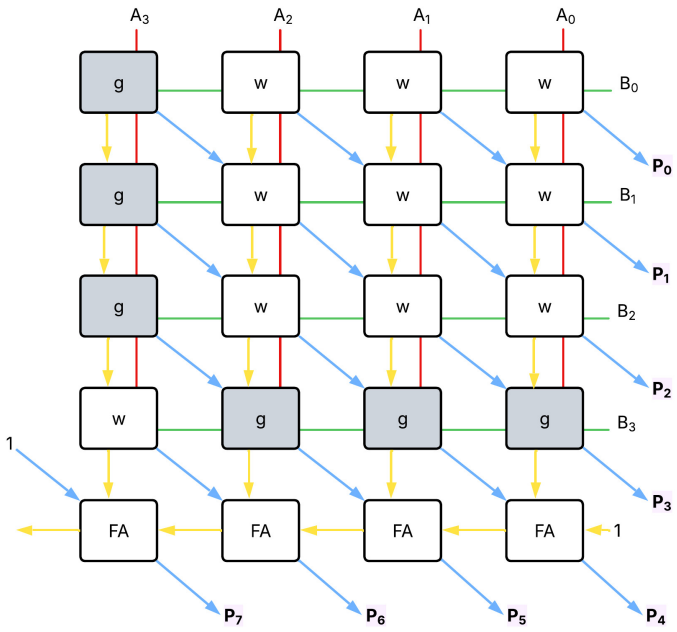

Figure 3.2: An example of a 4-bit Baugh-Wooley multiplier

As it is visible from the figure 3.2, the grid formed by the cells has white ones (A&&B) everywhere except the left and bottom sides, without the left-most bottom corner. The red lines form the connections of the A operand and the green - of the B operand. The Sum propagates on the blue arrows and the carry on the yellow one.

Then there is a row of full adders on the bottom to sum the resulting sum out and carry out vectors. One is added in the carry of this full adder to support 2s complement multiplication. The carry out of the last FA is not considered as the

largest possible value obtained by multiplying two 4-bit numbers will never exceed 8 bits. This multiplier forms the basis to further improvements explained in the next section. Each variant later on will transform a reconfigurable grid into versions of this multiplier - including the inverting (gray) blocks and logic 1 injection in order to support 2s complement signed multiplication.

3.1.2 st, sa operations

The concept of Sum together(st) and sum apart(sa), also called sum separate(ss) explains the different types of parallelization for doing a multiple number of reduced precision multiplications simultaneously. The difference between the two general schemes is in the formation of the output result. Different parts of the output vector form the relevant result of the operation in different modes. Here an example of a 4by4 grid is shown, but the same concept is extended to larger arrays.

Sum Together (ST)

The objective of the ST mode is to perform several reduced-precision multiplications instead of one high-resolution multiplication. The resulting products are directly accumulated inside the multiplier. Its principle is (for a simple 4-bit full precision example) depicted in Figure 3.3. This configuration saves the necessity of using additional adders to sum up these products, as it uses the multiplier array cells to implicitly perform the addition. As a result, ST mode saves significant register activity and MAC output bandwidth.

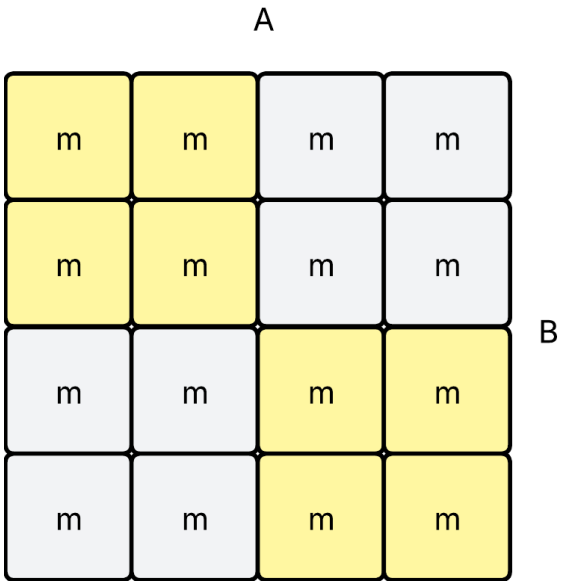

Figure 3.3: depiction of a simple ST mode multiplier

As we can see in Figure 3.3, in this sum-together variant only the two 2x2 yellow squares are "active". This means that the non-active ones need to only "pass through" the information to their neighbors, without adding their partial product contribution to the sum.

As the sum and carry propagate down and right, the results of the two smaller multiplications(from the partial products of the active regions) will end up in the same portions of the output vector result and be added together. For larger arrays (for example 16 by 16) this mode can have variants - 4 times 4by4 bits, 2 times 8x8 bits or even 8 times 2 by 2 bit multiplications. Each mode has to form a power of 2 total multiplications in the variant of the multiplier array used in this case.

Sum Separate (SS)

The sum-separate (or sometimes called sum-apart) configuration of the multiplier enables the computation of $16/m$ products independently at m-bit precision with m = 2,4,8,16. The multiplier array shown in figure 3.4 acts similarly to the previous one, however the results of the operations propagate to the output without getting added together. They essentially end up in different slices of the output vector. In this example the multiplication is split into two 4by4 ones creating the 8bit results. In this mode too, the array can be configured as different smaller-precision multiplications, for example, in the 16 by 16 array.

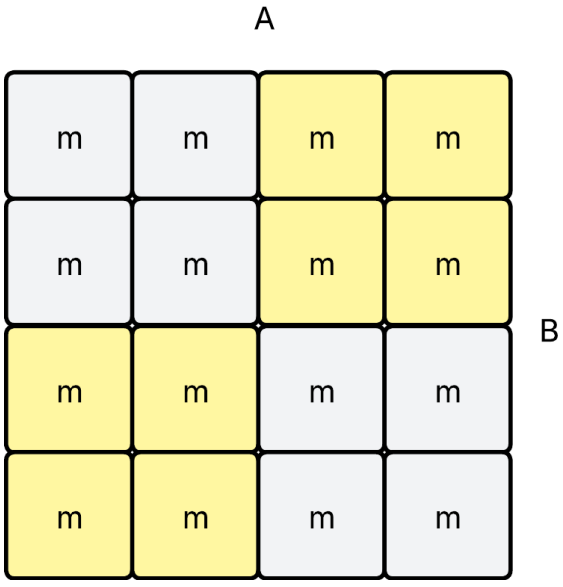

Figure 3.4: depiction of a simple SA mode multiplier

Comparison between the modes
It is important to note that different parts of the input operands end up in the

same multiplication in SA and ST modes. As it's visible from figure 3.5 that in Sum Apart (or Sum Separate) the same portions of the two operands end up in the same multiplications, compared to Sum Together, where the low bits of A are multiplied by the high bits of B. In the example below there are 16 bits operands and they are split into 4 times 4 by 4 multiplications.

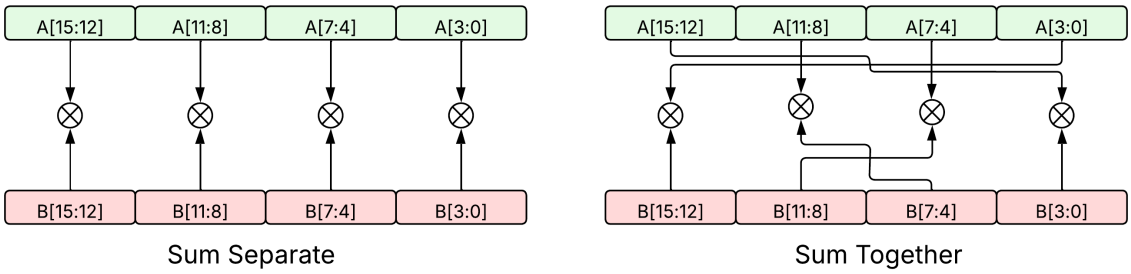

Figure 3.5: Figure SS vs ST operand splitting

The same type of "mirrored" bit slicing of the operands also happens in other larger variants of the multiplier and in other configurations (in 8x8st and 8x8 sa modes).

The next step of improvement is to make the multiplier reconfigurable so that it can support both types of modes with different variants. What needs to be done is to have additional control signals for each partial product in the grid. In this way the whole grid becomes reconfigurable. Another thing to consider is that for supporting 2s complement multiplication, small multipliers need to be able to 'switch' from the white to the gray variant similarly to figure 3.1 so that the bottom and left sides become invertible.

3.2 STAR multiplier

The STAR (Sum Together/Apart Reconfigurable) architecture, developed at Politecnico di Torino[1], is a reconfigurable multiplier array capable of operating in two distinct modes, Sum-Together (ST) and Sum-Apart (SA), depending on its configuration. This flexibility makes STAR suitable for integration in a variety of applications, such as within the multiply and accumulate (MAC) units and hardware accelerators.

The ability to dynamically reconfigure application-specific accelerators allows for more efficient utilization of hardware resources, enabling the same hardware blocks to be shared across multiple computational tasks. For example, as illustrated in Figure 3.6, a STAR-based MAC unit can be used inside a single hardware accelerator to support both 2D and Depth-Wise (DW) convolution operations.

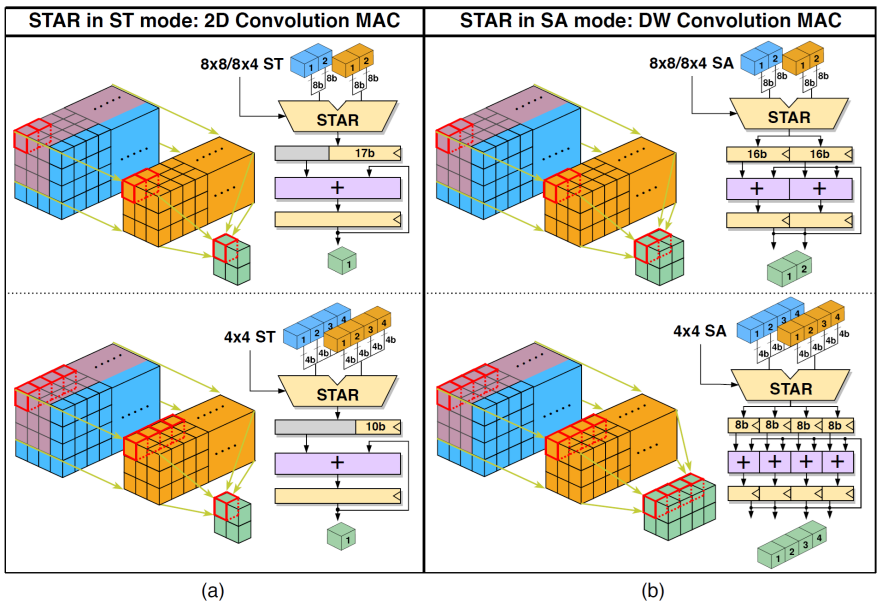

Figure 3.6: STAR-enabled reconfigurable MAC for 2D/DW Convolution.[1]

In **2D convolution**, the STAR multiplier operates in **ST mode**, performing multiply–accumulate (MAC) operations between low-precision input values (light blue) and weights (orange) on a channel-by-channel basis (Fig. 3.6). The resulting partial products are forwarded to an external accumulator, which combines them until all input elements have been processed and the corresponding output value (green) is produced.

In **depthwise (DW) convolution**, the STAR multiplier is configured in **SA mode**, enabling several low-precision multiplications to run in parallel without internal accumulation, using features and weights from separate channels (Fig. 3.6). In this configuration, the external accumulator is adapted to the target precision and maintains independent partial sums for $N = 2$ or $N = 4$ output elements.

The STAR architecture can also use the **ST mode** to accelerate **fully connected (FC)** layers, extending its applicability across a wide range of neural network workloads. An example illustrating the modes of operation for a 16×16 STAR multiplier is shown in Fig. 3.7.

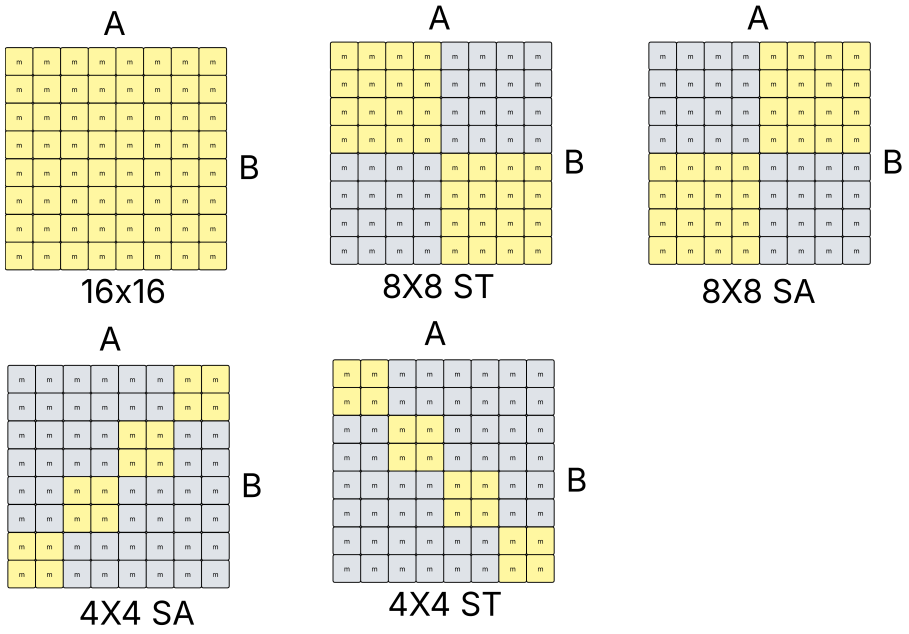

Figure 3.7: STAR modes of operation

According to the selected operating mode, some partial products (PPs) become active (yellow squares) and contribute to generating the valid output bits (yellow), whereas other PPs are inactive (gray squares) and do not contribute to the final 32-bit result.

In table 3.1 are the modes and the output function that should be calculated

CONFIG	STAR output
16x16	O[31:0] = A[15:0]*B[15:0]
4x4st	O[21:12] = A[3:0]*B[15:12] + A[7:4]*B[11:8] + + A[11:8]*B[7:4] + A[15:12]*B[3:0]
8x8st	O[24:8] = A[7:0]*B[15:8] + A[15:8]*B[7:0]
4x4sa	O[31:24] = A[15:12]*B[15:12] O[15:8] = A[7:4]*B[7:4] O[23:16] = A[11:8]*B[11:8] O[7:0] = A[3:0]*B[3:0]
8x8sa	O[31:16] = A[15:8]*B[15:8] O[15:0] = A[7:0]*B[7:0]

Table 3.1: STAR modes of operation

3.2.1 Star architectures

There are a few architectures of the STAR multiplier that can achieve the desired functionality. For example a naive implementation would be to have separate multipliers for each mode and then have a multiplexer that decides on the appropriate output of the whole device. Here the SWP architecture is explored further.

3.2.2 SWP architecture in 32 bits

In order to support these multiplication modes, the grid needs to have a few control signals distributed along its elements. Most of them are a modified version of the modules of the Baugh-Wooley multiplier (section 3.1.1) with an added enable signal to the A and B operand. This signal can turn off said elements by providing them with a 0. Then they would only pass through what was added to the total sum without adding the components for which they are responsible in the grid. In the figure below are shown these modified elements.

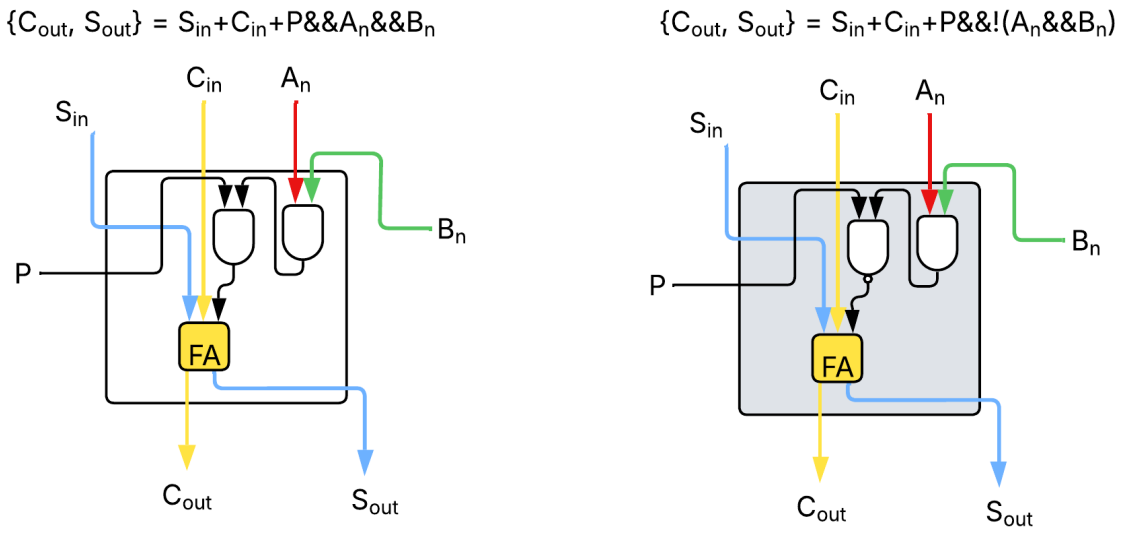

Figure 3.8: the white and gray element

With P=1 these elements behave like the ones of the Baugh-Wooley array. The P signal is decoded from the configuration and given to the elements in groups as seen later. In this way the normal array is transformed into the configurations from fig.3.8.

The red element's function is to invert the partial product's sum coming from the A and B operands such that it "transforms" a white block into a gray one and vice versa. In this way it can “create” the needed left row and bottom column of inverting blocks to support 2s complement multiplication. In the red blocks, the input of their internal full adders (FAs) can be forced to logic 1, independent of

the actual value of the partial product (PP), by setting the control signals ($P = 0$ and $I = 1$). This capability can be leveraged to insert the 1s required by the BW algorithm in place of the S_i inputs of the PPM blocks in the top row and left-most column, reducing resource usage and shortening the propagation path for these inputs.

Nonetheless, the red blocks do not cover all the positions where logic 1s are needed. To handle the remaining positions, blue blocks (details in Fig.**??**) and green blocks (see Fig.**??**) include an OR gate that can force the FA input to logic 1 when $I = 1$.

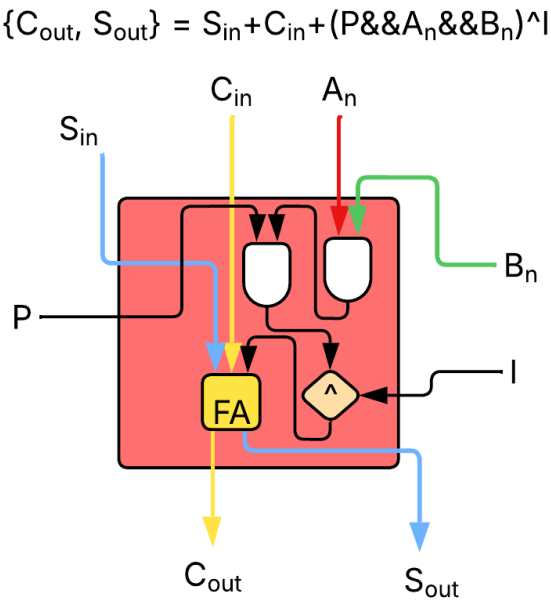

Figure 3.9: the inverting block

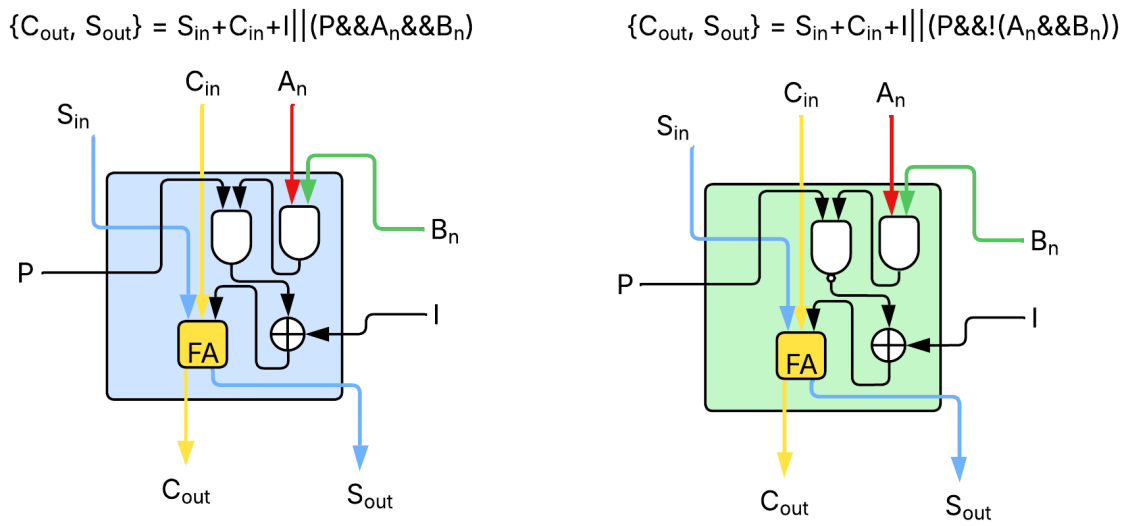

Figure 3.10: the blocks that "inject" a 1. Note: the plus sign block is an or gate

Each block of the Partial Product Matrix (PPM) receives specific binary inputs for the signals P and I, as indicated by the letters inside each block (see Figure 3.11). When a block contains a single letter, it receives only a logic value for P, since white and gray blocks do not have an I input [1]. When a block contains two letters, it receives logic values for both I and P. For example, a label such as `c/d` indicates that $I = c$ and $P = d$.

For **SA (Sum-Apart)** operations, the carry chains connecting the most significant bit (MSB) of one sub-word to the least significant bit of the next must be interrupted. This is achieved by inserting AND gates at specific points in the circuit to block carry propagation when the control signal $M = 0$, as shown in Figure 3.11. These AND gates also modify the carry path of the 16-bit Ripple Carry Adder (RCA), effectively splitting it into two independent 8-bit RCAs when necessary. The positions of these AND gates are marked with "X" symbols in Figure 3.11, and each diagonal of "X" symbols corresponds to a control signal applied to the M inputs of the AND gates along that diagonal.

Similar to P and I, the M signals are derived from the CONFIG decoding logic, where the letters `m` and `d` in Figure 3.11 represent the logic values associated with the green and violet "X" symbols, respectively. Additionally, carry propagation enable signals, also indicated by crosses in the diagram, are used to selectively block carry propagation, thereby splitting the results during SA mode operation. This mechanism allows the array to support all five operational modes of the STAR architecture.

For clarity, the specific control signal values are not shown here, but they can be derived directly from the configuration logic. A more detailed implementation is discussed in the following section[1].

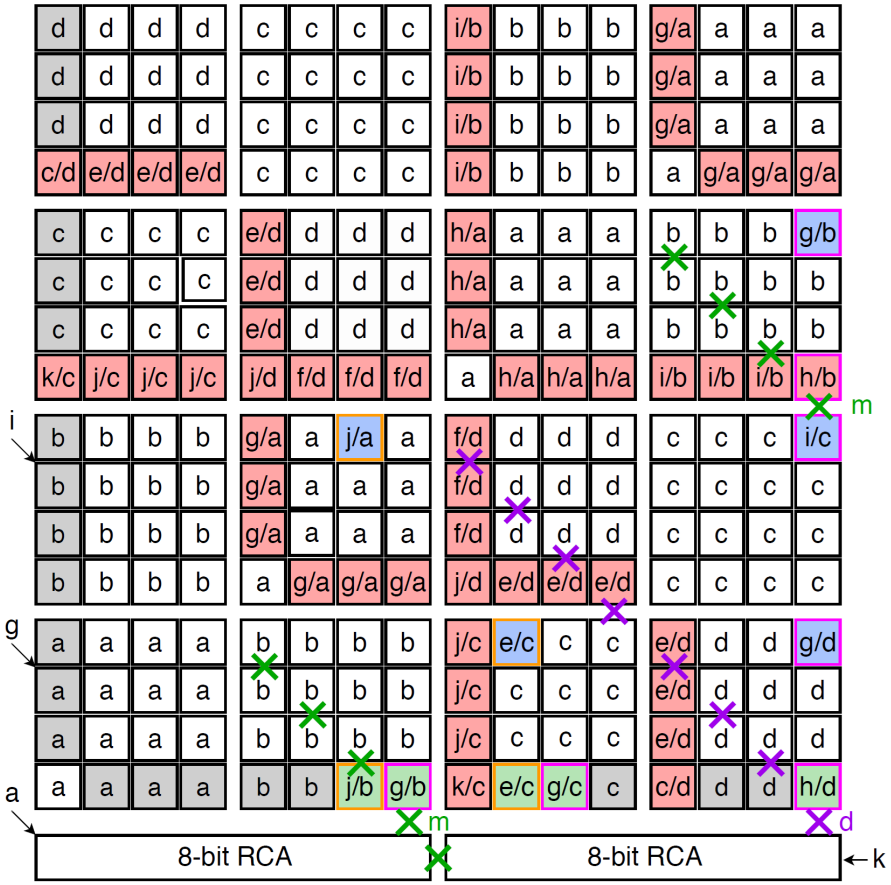

Figure 3.11: STAR SWP (BW) PPM with two 8-bit RCAs.[1]

3.2.3 52 bit STAR with MAC

A more complex variant of the STAR multiplier[2] is to have it accumulate the result over some cycles and then a larger vector could be read from it. This is useful when a higher number of accumulations need to be performed and can reduce output bandwidth. The general idea is to have the config not only control the mode of the array of partial products, but also control the way the output is added and stored within the larger 52 bit register.

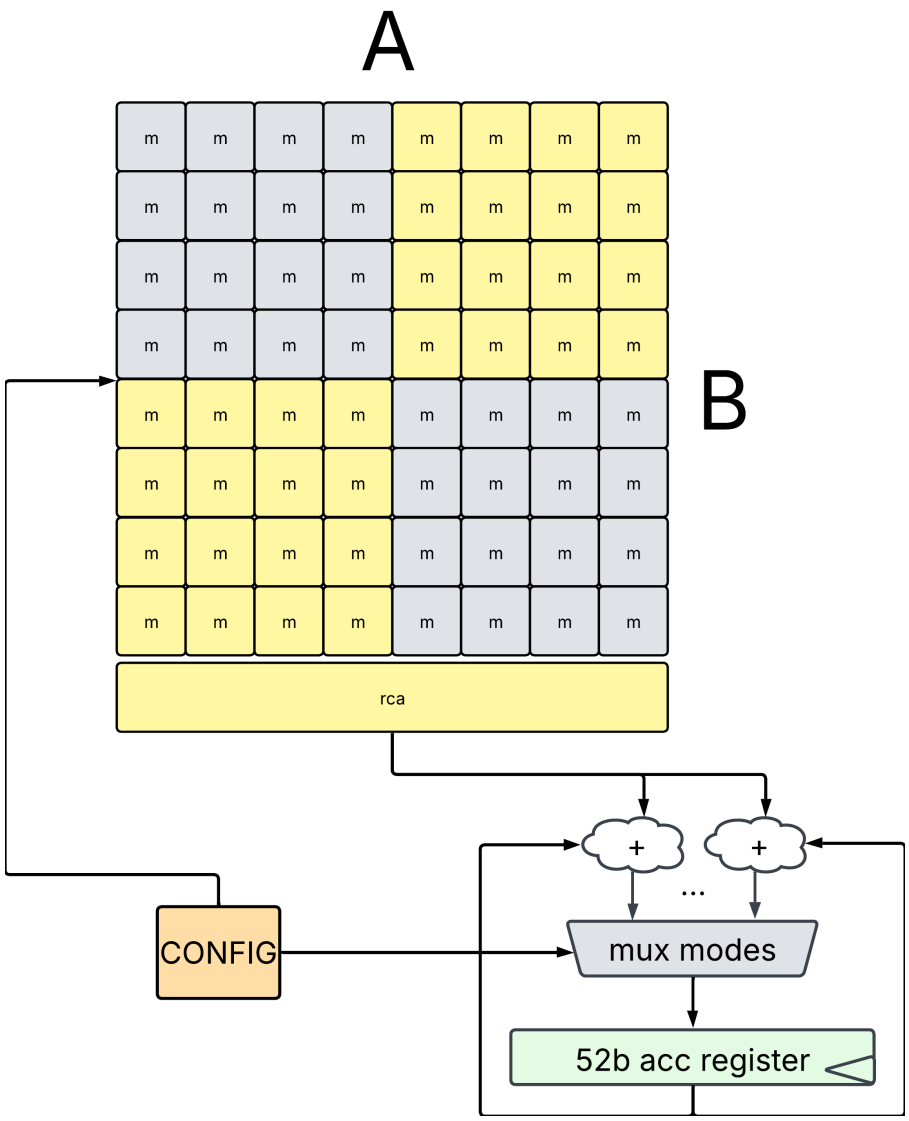

Figure 3.12: STAR with internal accumulation

The variant of the star multiplier with an internal accumulation register has a wider output of 52 bits instead of 32 bits. The idea is to use specific bit slices of the result for each different mode. They are spaced in such a way that a normal adder could be used to sum the 52-bit STAR output with the 52-bit register value and each result will be summed with its proper value in the reg. There is enough spacing in order to support at least up to some number of accumulations in the worst case (maximum value of the operands). This spacing (gray area in the figures below) is added after the calculation from the SWP array by spacing out the bits of the result. The spacing is controlled by the opcode. For different modes not all bits of the result will be considered. The bit sections, maximum value of the result

and guaranteed cycles for which there is no overflow for each mode are summarized in the next subsections.

16x16st mode

Largest value from summation is

$$16\{1\} * 16\{1\} = (2^{16} - 1)^2 \approx 2^{32}$$

The result is stored in $[47 : 0]$

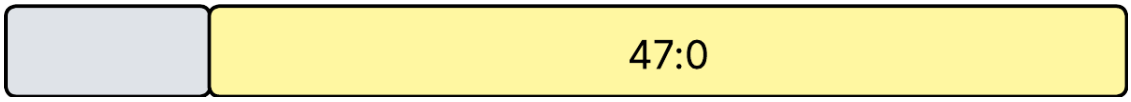

Figure 3.13: 16x16st mode

Therefore a safe approximation in the worst case for a maximum number of accumulations without overflow is $2^{48}/2^{32} = 65536$

8x8st mode

Largest value from an individual summation is

$$2 * 8\{1\} * 8\{1\} = 2(2^8 - 1)^2 \approx 2^{17}$$

The result is stored in $[36 : 0]$

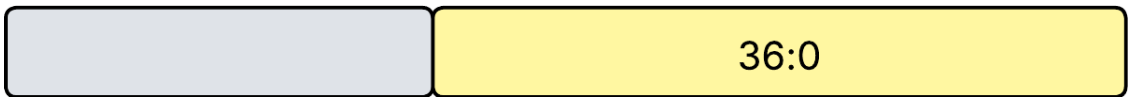

Figure 3.14: 8x8st mode

A worst case for the maximum number of summations is $2^{37}/2^{17} = 1048576$

4x4st mode

Largest value from an individual summation is

$$4 * 4\{1\} * 4\{1\} = 4(2^4 - 1)^2 \approx 2^{10}$$

The result is stored in $[23 : 0]$

Figure 3.15: 4x4st mode

Worst case maximum summations of 4x4st operations is $2^{24}/2^{10} = 16384$

8x8sa mode

Largest value from an individual summation

$$8\{1\} * 8\{1\} = (2^8 - 1)^2 \approx 2^{16}$$

for 2 entries in the result vector $[46 : 26]$ and $[20 : 0]$

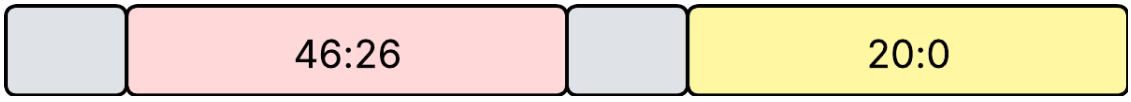

Figure 3.16: 8x8sa mode

In this case a safe approximation for maximum number of summations $2^{21}/2^{16} = 32$

4x4sa mode

Largest value from an individual summation

$$4\{1\} * 4\{1\} = (2^4 - 1)^2 \approx 2^8$$

for 4 entries in the result vector

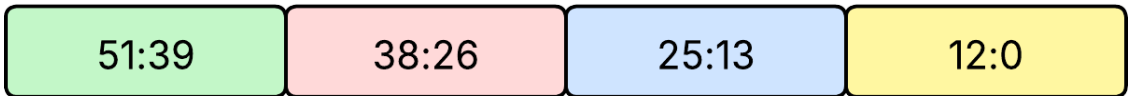

Figure 3.17: 4x4sa mode

Safe approximation for maximum number of summations: $2^{13}/2^8 = 32$

Chapter 4

ST, SA operations in AHA

This section discusses possible implementations of a configurable reduced precision multiplier within a PE of an AHA CGRA. Some variants are promising and are in theory good solutions, even if their implementation success was limited. The chapter focuses on comparison between those which could work and the other non-optimal solutions. In the last section an experimental implementation is explored, which could be improved further in future works.

4.1 Overview of implementation strategies

4.1.1 Increased bitwidth

An approach that initially seems straightforward is to increase the bus bitwidths in order to fit the star result. It could be done with 32 bits for the multiplication or 52 bits for the multiplication and accumulation variant of the configurable multiplier. The downside of this approach is that the whole interconnect will have to be also scaled up by a large factor - from 16 to 32 or to 52. This increases the overhead, power consumption, area of the CGRA and is therefore very limiting in terms of performance. If for example the total area of the chip is fixed, adding all these extra interconnect wires, multiplexers and logic within the PE (to support also 32 or 52 bit operands) will equate to less PEs within the chip. The same is valid for a fixed power consumption - it leads to lower number of PEs when wider interconnects are used. The increased bit-flips lead to a greater power consumption, because of the higher number of mos-capacitor charges and discharges within gates (of flip-flops) changing their state at each clock cycle. After all, the whole reason for using reduced precision is to be able to launch NN calculations on a platform where power and area are limited. This will therefore diminish the advantages of the whole STAR integration. Another more “real-world” reason this approach was not feasible in the Stanford AHA Amber CGRA framework is that the operands,

interconnect, memory and other similar signal's widths are hard-coded in their current implementation. This makes it very hard to manage making this approach work even just as a proof of concept. A next step-forward is to include an internal register used for accumulation and then output the data only when the MAC operation is finished. This is discussed in the next subsection.

4.1.2 Internal accumulation reg

The way a multiply and accumulate operation is used in the context of a NN is to perform a high number of small multiplications either for matrix operations or for convolutions. In both cases, the STAR multiplier can speed up the process by doing some number of these operations in parallel. If then the results are accumulated again, the result would be an accumulation over a larger number of operands. In the ST case there would be only a single result and in the SA case there will be some separate accumulations happening independently. This means that we don't really need the ability to do separate star operations and we can have a register that accumulates the result. Then we can provide this result to the PE output. In order to be able to initialize the internal register or reset it for any reason, the implementation also requires an operation to clear the sum in the register. The total width of the reg is 52 bits to facilitate the different operations of the star multiplier. As it is 52 bits instead of 16, it can be achieved by either having 4 by 16 bit buses used as one or by changing one bus to 52 bits for one input and one output.

wide output

A wide output can be done by changing the interconnect width as described in the previous subsection, but as we have seen this approach is not feasible in AHA. Nevertheless, it could be a working solution in another CGRA architecture.

output in 16b parts

The other way, which leverages onto the existing widths, is to crop the result in 4 separate parts and output them with different opcodes. This is no longer a limiting factor to the throughput, as is the case with subsection 4.1.3, because now we only need to do this high number of opcodes at a single point in time - at the output of the mac result, opposed to doing it on every cycle of the calculation.

Location of the accumulation register

This implementation strategy can be done in a few different ways, depending on the placement of the accumulation register. Its location impacts the type of rewrite rules we can have, as they look at the PE from outside (as a whole unit). The

location also impacts the sizes of the inputs/outputs of the PEs, which is seen in the later subsections.

Inside the alu

In the original Amber CGRA from Stanford the arithmetic and logic unit of the PE is purely combinatorial. This means that it doesn't have an internal state - therefore, no clock and no reset. To achieve the internal state we have to pass these signals to the alu. The ALU will have to get a structure similar to the one in figure 4.1.

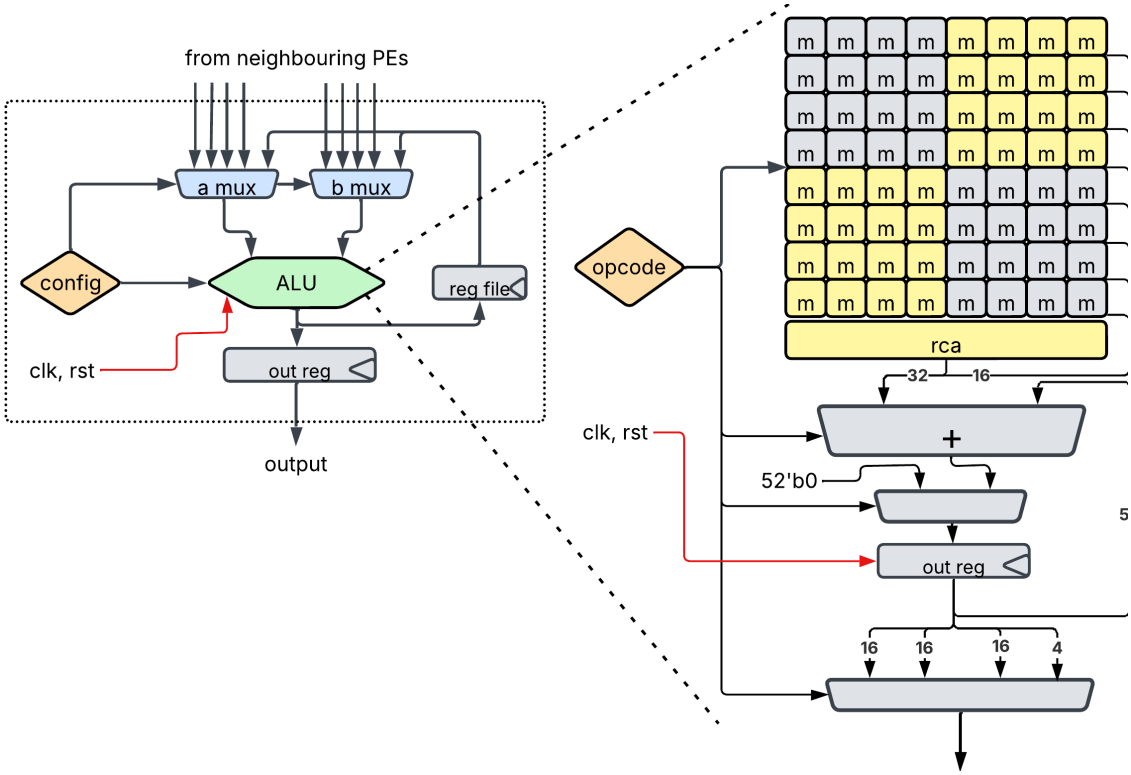

Figure 4.1: accumulation reg inside the ALU

Note that the multiplexer before the register has a second input of 0 to clear the accumulation when it is needed. This solution is elegant, but it turns out to be unusable in AHA, because in the way the rewrite rule generation works it can not operate on the internal state of the ALU and is unable to map the STAR functionality onto the new opcodes.

Inside the PE, but outside the alu

This possibility is closest to the original lassen PE, as it takes advantage of the register file within the PE. The main difference is that it requires wider signals on the interface between the alu and the actual register. To fit the star multiplier functionality with accumulation means to have 52 bit input and output of the alu, along with the two 16 bit input operands. The 52 bit input is the registered value and the 52 bit output is the register input. Then there would need to be the 16 bit indexing of the register (similar to the last subsection) used for sending the accumulation value through the 16-bit interconnect in chunks. A possible implementation is reported in the figure below.

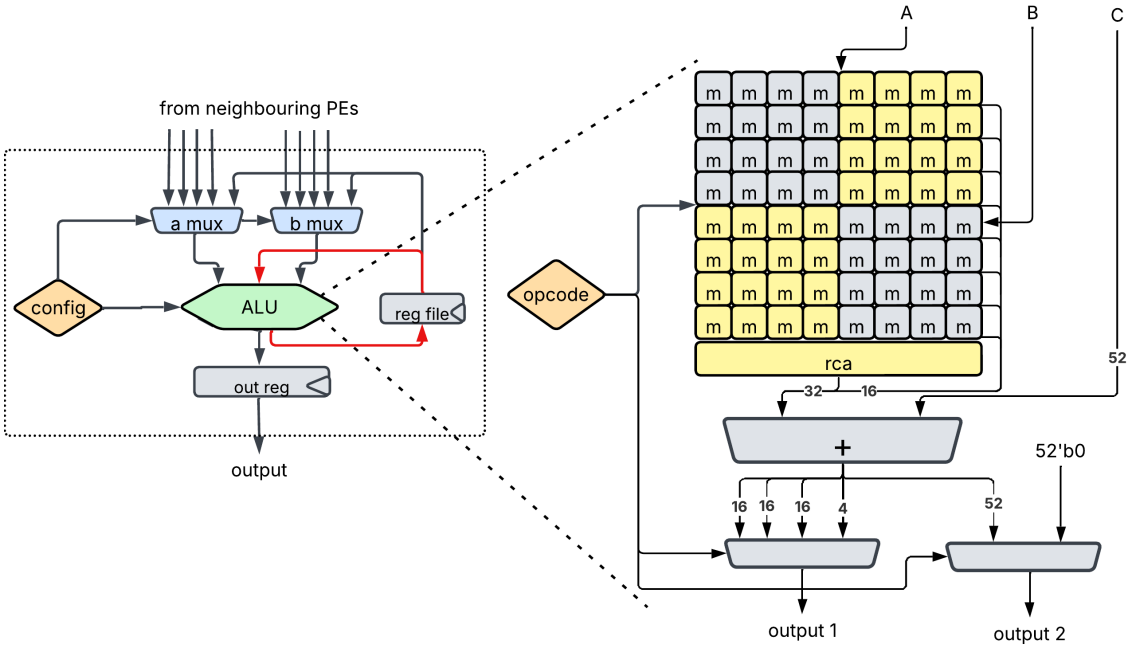

Figure 4.2: internal accumulation reg using the register file

Note that in this configuration, there is no internal state within the ALU. It has to provide the C input to the next cycle through one output and the result through another in 16b parts. Another way to do it is to have the C input go directly into a mux, so that the output 1 will be the current result, not the next one.

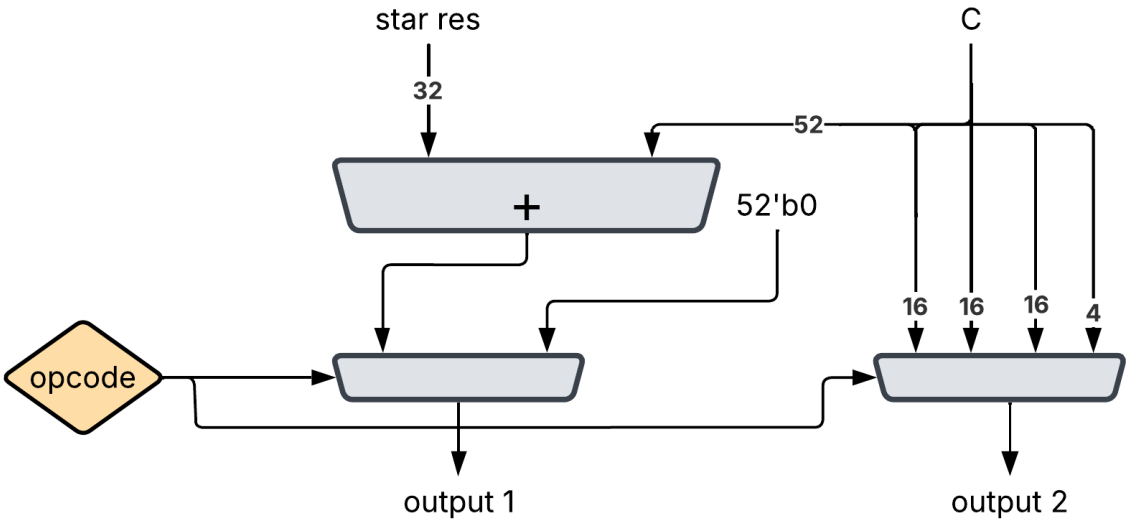

Figure 4.3: the two output assignments

Note that the second output is just slicing of the C input (the value of the register file). The problem with this otherwise elegant solution is that we need to have two outputs of the ALU, or in other words to have a single instruction - multiple operations. This type of ALU is not compatible with the AHA methodology for the reason it can map only a single operation to a single PE. What it does with this specific hardware is that it maps one operation to one PE and another one to a different PE. In other words, it doesn't "realise" that it can use a single PE with one opcode to do both operations. This solution, however, has potential to be used in another CGRA architecture. It combines the functionality and interconnect efficiency(in terms of overhead) of the other presented solutions and appears to be the best one in theory. The downside to this approach is that it spans more opcodes than the usual reconfigurable star multiplier. The added ones are related to sending the data from the register out of the PE and to resetting it.

Outside the PE

The third possibility is to have the register out of the PE. This solution will also be quite similar to the original AHA Amber implementation, however it would need extra elements within the grid. This is not straightforward to do in AHA, because it requires to have registers inside the grid, along with the memories. The memories can not work for this cycle-to-cycle accumulation, as the memories have 2 or 3 cycle write-read delay. Even if it is doable though, its greatest limitation would be the fact that each PE would require a 52 bit input and output to transfer the old and the new value of the accumulation register. This approach could be a good solution in another CGRA topology, for example one which has a register type of memory (1 cycle delay between write and read), and a CGRA framework

which has a different type of mapping. In AHA, there is no guarantee that a single PE would be used, instead it may map the multiply and accumulate operation on multiple PEs. Therefore this approach is also not feasible in AHA. Compared to the previous one it is also inferior, because it doesn't have the advantage of smaller width interconnections.

4.1.3 Low/high bits division of result

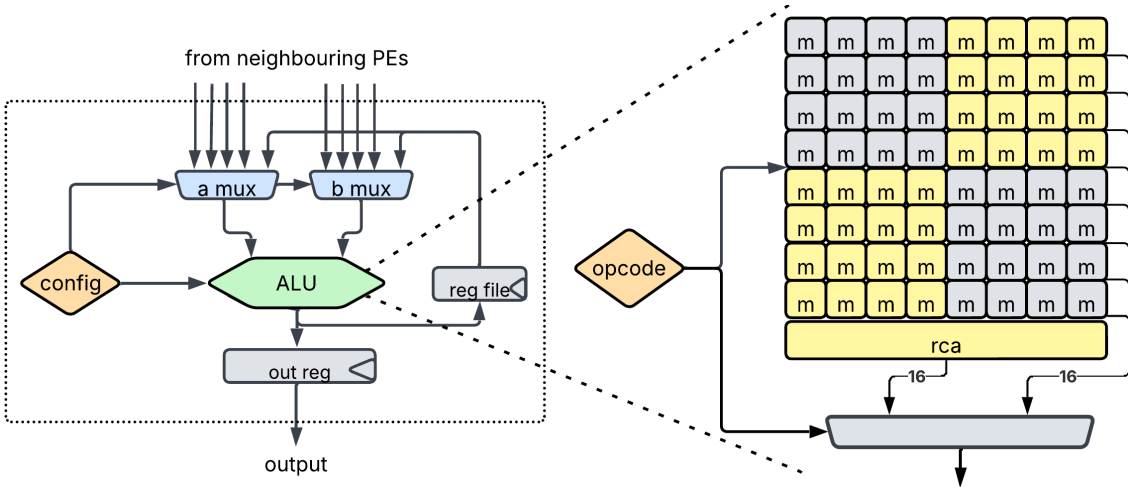

Figure 4.4: division of result

A simple way to fit into the Aha Garnet cgra is to implement the star multiplier in a way such that the output is calculated in two different operations with separate opcodes. One of them to have the output equal to the low 16 bits of the result and the other to have it to be the high 16 bits. This is similar to the way the multiplication is done inside the lassen alu - it has three opcodes for the low, middle and high 16 bit sections of the 32 bit result. This means that for applications where only one part is non-zero or needed at all the multiplication can be done in 1 clock cycle. For cases where the full result is needed the 16x16 multiplication can be done either in 2 clock cycles or by using two PEs and concatenating the results. For the STAR multiplier, however, this approach is very limited, because it will defeat the purpose of doing the multiplications together if it takes more than one clock cycle or PE. It could only make sense for the 4 bit operations. The 4x4st can fit within 16 bits and the approach could work. The 4x4sa fits within 32bits, but we still do 4 multiplications in 2 cycles, so it also makes sense. The 8 bit ones essentially do 2 multiplications in 2 cycles, therefore it's as good as using the normal multiplication. The 16x16 is exactly the same as the normal one in terms of efficiency.

4.1.4 Input reordering

A different approach is to limit the implementation to the smaller output modes of the STAR multiplier(4x4st and 8x8st). In this way we don't have to deal with the high number of bits in the output data-path, which was seen to be not feasible in AHA, at least in the ways that were tried. However, even limited by the smaller throughput there are some strategies to implement the larger multiplications as explained. As the mac operation is performed multiple times and in the use case of a neural network, multiple results are usually accumulated. A different approach to employing the STAR multiplier in a CGRA is to use the 4x4st or 8x8st modes for doing a larger multiplication than 4bit and 8bit respectively. By decomposing the large multiplication into a bunch of smaller ones, followed by shifting and adding, we can distribute the large one within some number of smaller ones. By switching the order of summation, we effectively do the same final operation, as summarized in the figure below.

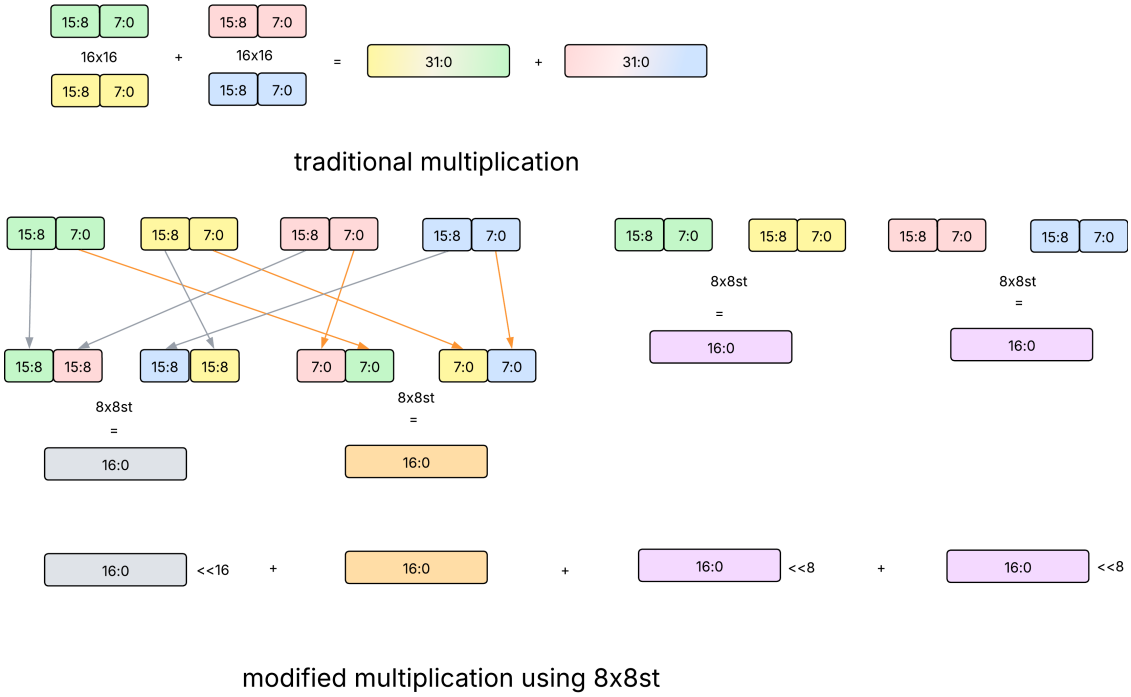

Figure 4.5: 8x8st reorder to do 16x16

In the example from the figure the case with 8x8st is shown. For 4x4st a similar reordering is done - by concatenating 4 bit operands together for multiplications that would be summed up either way. This is how the 4x4st can even be extended to perform 8-bit and 16-bit multiplications.

4x4st used to do 8 bit operations

An 8 by 8 bit multiplication is essentially four 4 by 4 bit multiplications added together with different amounts of shift-left. The lsb multiplied by lsb has no left shift, lsb multiplied by msb has 4 and msb by msb has to be shifted left by 8 bits. Summing them together results in the full 8 by 8 bit multiplication.

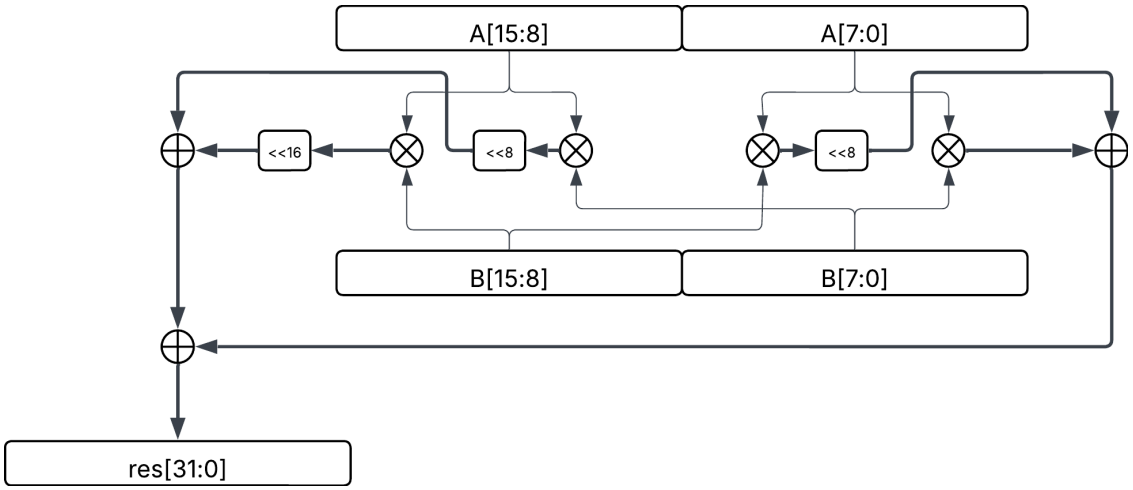

Figure 4.6: 16x16 done by 8 bit multiplications

When the star multiplier is used to perform a high number of MAC operations, as in a neural network, the four separate components of the 8x8 multiplication can be summed together in three groups - the 0, 4 and 8 bit shift-left, before adding them together. As an example - four 8x8 ones summed together can be done with four 4x4st operations.

4x4st used to perform 16 bit multiplications

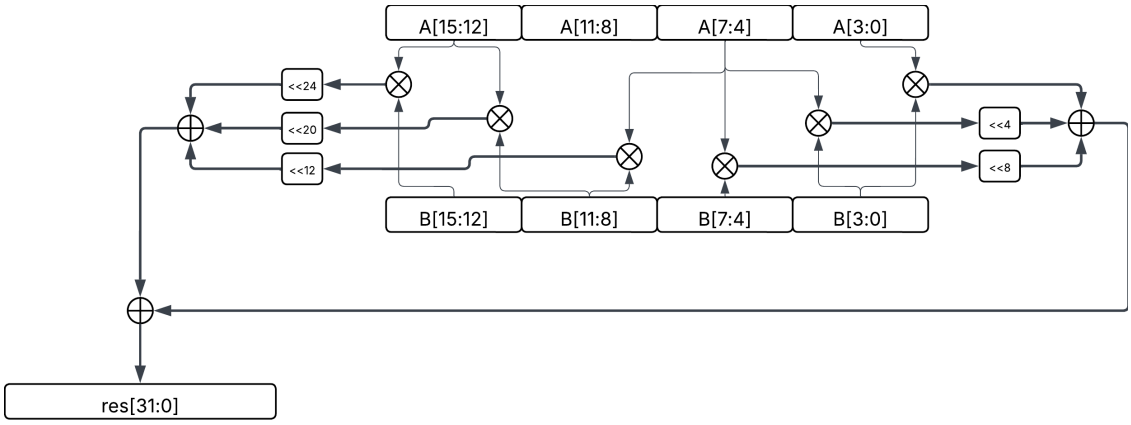

Figure 4.7: 16x16 done by 4 bit multiplications

The same can be done for performing a 16 by 16 bit multiplication. This time, however, the possible shift-left amounts are 0, 4, 8, 12, 16, 20, 24 and 28 bits. The total number of 4x4 multiplications needed to do a single 16x16 one is 16.

4.2 Experimental implementation

A workaround of the AHA limitations is to trick the mapping. It is possible to map the Halide app onto the CGRA as if the multiplication was a standard 16x16, but it is actually a reduced precision sum-together operation for example. The halide app also needs to be written in such a way that it has a normal multiplication in the places we want to implement the star operation. Then we have to do modifications in order to actually use the STAR operation. They can be one of the following:

4.2.1 Verilog generated file modification

The idea is to change the rtl file to do a STAR operation instead of multiplication. This is hard to achieve, because the generated rtl file is very big in size(it includes the whole PE). However there is a better way to do it - to trick the actual mapping.

4.2.2 Rewrite rule modification

In this variant, both multiplication and star reduced precision operations exist within the PE functionality. In order to trick the mapper to use it, the content of the rewrite rule json file need to be changed to actually setup the PE to perform a star operation instead of 16x16 multiplication. In this case we can simply substitute the opcode for the desired operation in place of the multiplication one. Here both

functionalities need to be a part of the formal PEaK specification and exist in the verilog file. The change has to be done between the rewrite rule generation and the application mapping step. The Figure below highlights when the change needs to be done within the design flow of the AHA process.

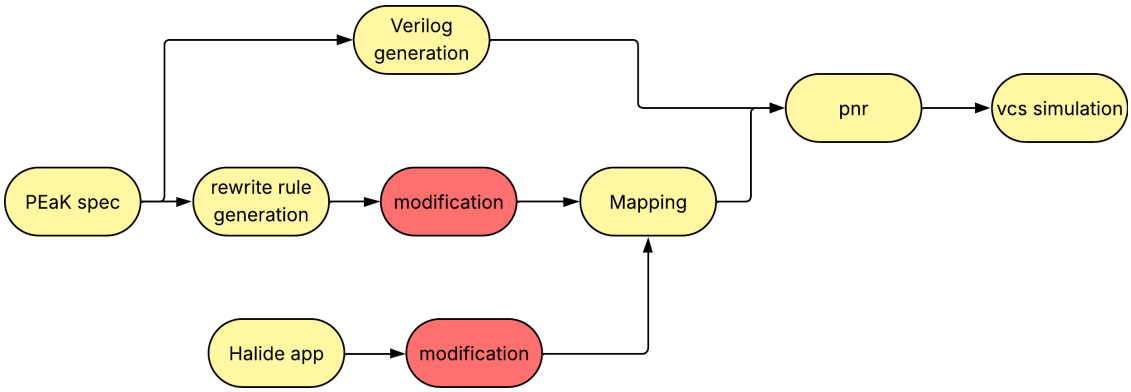

Figure 4.8: Rewrite rule modification

4.2.3 Modification of the mapping file

The simplest way to achieve star functionality within the AHA CGRA is to modify the mapped application file to use the star operation instead of another one. The modification has to be done only to operations that are not desired to be normal 16x16 multiplications. In the Halide app these operations need to be instead 16x16 instead of the star, in order for the AHA mapping to work. The figure below summarizes the modifications to the design flow.

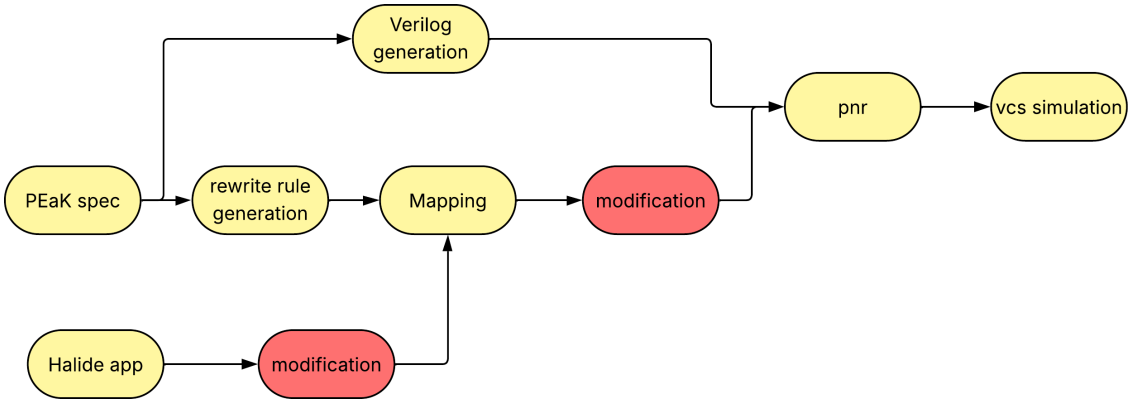

Figure 4.9: Mapping modification

Part of the formal PEaK specification of the PE (the part where the 16x16 and 4x4st are) is shown.

```
elif alu == ALU_t.STAR:
    mulstara = UData(SData8(a[0:8]) * SData8(b[8:16]))
    mulstarb = UData(SData8(a[8:16]) * SData8(b[0:8]))
    mulstar = mulstara + mulstarb
    res, C = mulstar, overflow(mulstara, mulstarb, mulstar)
    res_p = C
```

After this modification to the mapping, the PnR works correctly, but the testing, of course, fails, because the mapped CGRA is no longer doing the same operation as the Halide app. This is expected, because of the workaround. In the next subsection a simulation of the generated PE within a SystemVerilog testbench is done.

4.2.4 Simulation of the generated PE

The aha garnet script does the verilog generation, as explained in section 2.5.9. It generates a lot of muxes, multipliers and adders in order to fulfill the whole PE functionality. For example the part responsible for the 8x8st operation includes a 17 bit adder, where 16 of its bits are connected to the output of the alu and The MSB is connected to the ALU carry flag. The whole PE is instantiated inside a SystemVerilog testbench and is driven to perform the 8x8st operation. The result of the vcs simulation is a wave .fsdb file, it is displayed using Synopsys Verdi waveform viewer and the relevant parts are shown in figure 4.10.

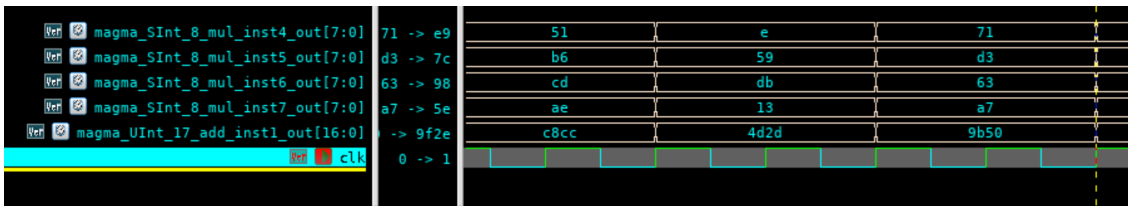

Figure 4.10: vcs Simulation of the PE

This result is somewhat promising, because it shows the correct behavior of the st operation. Further exploration of the possible next steps are described in Chapter 5.

Chapter 5

Conclusions and Future work

This thesis investigated the potential, design considerations, and practical limitations of integrating a configurable reduced-precision multiplier into the Stanford AHA Coarse-Grained Reconfigurable Array methodology. Although STAR multipliers offer significant advantages-particularly in energy efficiency and throughput for neural-network workloads-integrating these architectures into the AHA framework proved demanding. The analysis presented in Chapter 4 identified both the architectural possibilities and the system constraints that define what is realistically achievable within this specific CGRA environment.

5.1 Conclusions

A range of implementation strategies was examined, including widening datapaths, embedding accumulation registers inside or outside the Arithmetic Logic Unit (ALU), and various techniques for result partitioning and operand reordering. While many of these approaches are promising from a fundamental hardware perspective, the existing AHA flow and mapping methodology impose constraints on their direct implementation. This thesis doesn't prove an implementation within AHA is impossible, so it could still be done in theory, but the approaches tried all had significant downsides/constraints. The main conclusions are summarized.

Increased Bitwidth This approach introduces unacceptable overhead in area, power consumption, and routing complexity, thereby fundamentally undermining the goal of utilizing reduced precision. **Constraint within AHA Framework** The fixed-width infrastructure of the AHA framework makes widening interconnects (e.g., to 32 or 52 bits) impractical.

Internal Accumulation Registers Theoretically ideal because it preserves

the main benefit of precision scalability. Accumulating directly in the ALU/PE offers the cleanest architectural solution with minimal interconnect overhead. **Constraint within AHA Framework** The AHA rewrite-rule compiler cannot manage internal ALU state nor map multiple simultaneous operations to a single Processing Element.

External Accumulation and Extended I/O Constrained by fixed interconnect and distributed mapping. **Constraint within AHA Framework** Moving accumulation outside the PE requires 52-bit per-PE communication, conflicting with AHA's fixed 16-bit routing network. Additionally, the mapper may distribute MAC operations across multiple PEs, making consistent single-source accumulation impossible.

Low/High-Bit Slicing Limited usefulness; only viable for the smallest STAR modes and diminishes reduced-precision benefits. **Constraint within AHA Framework** Dividing a STAR result across multiple instructions provides minimal benefit beyond 4-bit or partial 8-bit modes. For larger modes, multi- cycle extraction eliminates the intended parallelism and speedup.

Operand Reordering (Shift-and-Add) Eliminates the key speedup offered by STAR multipliers. **Constraint within AHA Framework** Decomposing higher-precision multiplications into many small STAR operations followed by shift-and-add introduces significant overhead and yields no net throughput benefit.

5.2 Future improvements

The promising implementations of Section 4.1.2 have the potential to be realized in another CGRA architecture. The limitations of AHA will not hold for an appropriately chosen CGRA methodology, and these implementations can outperform normal multiplications in terms of operations per cycle. There are a few directions that could be taken for future improvements to the work, which are explained later in more detail.

- Analysis of other CGRA architectures
- Other implementation strategies
- Enhancement of the workaround strategy
- Using the desired rtl architecture of the multiplier

Other CGRA architectures

There are many other CGRA design methodologies which have to be explored. Some of them should be able to work with the requirements of the STAR operation mapping (multiple operations per instruction and/or internal state of the PE).

5.2.1 Other strategies

There still is the possibility of a better implementation of the STAR multiplier inside AHA, that was not covered by this thesis. It is very hard to say that something is impossible, so this direction of future work is still considered.

Workaround strategy

The workaround proposed in Section 4.2 to overcome AHA's limitations can be further refined. Among the investigated strategies, modifying the mapping file appears to be the most effective approach for enabling STAR-type operations within the existing AHA design flow. Future work could extend this method by also modifying the Halide application after mapping, ensuring that the generated testbench mirrors the behavior of the modified mapped design. Such an enhancement would restore AHA's testing capability, which is one of the framework's most valuable features.

rtl architecture of the multiplier

Another step of the implementation process that remained out of the scope of this thesis was the integration of the rtl design from section 3.11[1]. It is functionally the same as the generated rtl, but it has a more optimal architecture. The goal is to swap the optimal architecture's rtl with the one generated by AHA.

List of Figures

Bibliography

[1] E. Manca, L. Urbinati and M. R. Casu, "STAR: Sum-Together/Apart Reconfigurable Multipliers for Precision-Scalable ML Workloads," 2024 Design, Automation & Test in Europe Conference & Exhibition (DATE), Valencia, Spain, 2024, pp. 1-6, doi: 10.23919/DATE 58400.2024.10546734.

[2] Edward Manca, Luca Urbinati, Mario R Casu. An End-to-End Flow to Deploy and Accelerate TinyML Mixed-Precision Models on RISC-V MCUs. TechRxiv. June 19, 2025. doi: 10.36227/techrxiv.173161032.20267860/v3

[3] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, et al. 2023. AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers. ACM Trans. Embed. Comput. Syst. 22, 2, Article 35 (March 2023), 34 pages

[4] Alex Reinking, Gilbert Louis Bernstein, Jonathan Ragan-Kelley. 2022. Formal Semantics for the Halide Language. arXiv preprint arXiv:2210.15740 2022, 27 pages