Optimizing Memory Mapping for Dataflow Inference Accelerators

Efficient Memory Utilization on FPGAs

by

M. I. Kroes

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

at the Delft University of Technology,
to be defended publicly on Tuesday March 24, 2020 at 14:00 PM.

Abstract

Convolutional Neural Network (CNN) inference has gained a significant amount of traction for performing tasks like speech recognition and image classification. To improve the accuracy with which these tasks can be performed, CNNs are typically designed to be deep, encompassing a large number of neural network layers. As a result, the computational intensity and storage requirements increase dramatically, necessitating hardware acceleration to reduce the execution latency. Field-Programmable Gate Arrays (FPGAs) in particular are well-suited for hardware acceleration of CNN inference, since the underlying hardware can be tailored to rapidly and efficiently perform the required operations. To this end, Xilinx introduced the FINN (Fast, Scalable Quantized Neural Network Inference on FPGAs) framework to leverage FPGAs for Neural Network (NN) inference. The FINN end-to-end deep learning framework converts high-level descriptions of CNN models into fast and scalable FPGA inference accelerator designs that are based on a custom dataflow architecture. In this dataflow architecture the input data are streamed in a feed forward fashion through a pipeline of per-layer dedicated compute units, that each have on-chip access to the associated NN parameters. In order to keep the compute units occupied, specific throughput requirements have to be satisfied by the memory subsystem. These throughput requirements directly dictate the shapes of the on-chip buffers that contain the NN parameter values. Especially for accelerators that exploit a high degree of parallelism, these memory shapes map poorly to the available on-chip memory resources of FPGA devices. As a result, these resources are typically underutilized, which leads an On-Chip Memory (OCM) deficiency, and limits the amount of parallelism that can be exploited. In this thesis, a methodology is proposed that improves the mapping efficiency of NN parameter buffers to the embedded Block RAM (BRAM) resources on FPGAs, without negatively impacting the accelerator throughput. To accomplish this, an architecture is proposed where the memory subsystem and compute units are decoupled, and operate as a producer-consumer system. Within this architecture, the memory subsystem functions at a higher clock frequency relative to the compute units, which enables the memory subsystem to match the consumption rate of the compute units when multiple NN parameter buffers are clustered within the same BRAM instance. Furthermore, a genetic algorithm is used to find optimal group arrangements for these clusters such that the mapping efficiency is improved, and the throughput requirements are still met. The proposed methodology has been applied to a number of CNN accelerators, and demonstrates BRAM reductions of up to 30%. The observed BRAM reductions enable existing FINN accelerator designs to be ported to smaller FPGA devices while maintaining the computational throughput.

Preface

In this thesis a methodology is presented that improves the mapping efficiency of buffers to the available block RAM resources on FPGA dataflow inference accelerators. Although this methodology was developed for specific deep learning inference purposes, the techniques that were employed in this work are universal, and can be applied to improve the memory utilization efficiency of any FPGA design that has predictable access patterns to its memory subsystem.

The majority of the work described in this thesis has been performed during my employment at the Xilinx Research Labs in Ireland under supervision of Dr. Lucian Petrica. The research team in Ireland consists of a group of researchers who are investigating the potential of leveraging FPGAs for heterogeneous hardware acceleration of applications to counteract the limitations imposed by Moore's Law. One of the fruits that came forth out of these efforts is FINN, where FPGA accelerated inference of quantized neural networks is explored. FINN targets a wide variety of FPGA devices, and aims to fully exploit the fabric such that the most optimal computational throughput is achieved for a targeted device. However, soon after FINN came to realization, it was discovered that the most prominent obstruction in the way of achieving this type of scalability was the shortage of on-chip memory. The work described in this thesis marks the culmination of my efforts to mitigate this problem.

Acknowledgements

There are several persons I would like to thank for making all of this work possible. In the first place I would like to express my deep gratitude to Sorin Cotofana, my supervisor at TU Delft, for putting me in contact with the right people, for his words of wisdom and for being a great personal and academic mentor. Equally, I would like to express my gratitude to Lucian for his guidance, commitment to bring this project to a good end, and willingness to share his extensive knowledge of computer architecture with me. I would also like to thank Michaela Blott, my team manager, for making this work possible and for providing me with valuable feedback together with Giulio Gambardella and Yaman Umuroglu. Finally, I thank my relatives and parents in particular for their unconditional support, and being there for me through all the hardships.

M. I. Kroes
Delft, March 2020

Contents

Abstract i
Preface ii
List of Tables v
List of Figures vi

Chapter 1: Introduction 1
- **1.1 Research Questions** 4
- **1.2 Contributions** 4
- **1.3 Thesis Outline** 5

Chapter 2: Background Theory 6
- **2.1 FPGA Infrastructure** 6
  - 2.1.1 Memory Resources 7
  - 2.1.2 The AMBA Standard 9
  - 2.1.3 Multi-SLR Devices 9
  - 2.1.4 High-Level Synthesis 10
- **2.2 FINN** 12
  - 2.2.1 Neural Networks 12
  - 2.2.2 FPGA Inference Accelerators 14
  - 2.2.3 The FINN Framework 16
  - 2.2.4 FINN Architecture 19
  - 2.2.5 The BRAM Bottleneck 19
  - 2.2.6 Accelerator Designs 23
- **2.3 Conclusion** 26

Chapter 3: Decoupled Architecture 27
- **3.1 Architectural Overview** 27
- **3.2 Asynchronous FIFO** 29
- **3.3 Stream Generator** 29
- **3.4 Hardware Costs** 30
- **3.5 External Memory Access** 31
- **3.6 Conclusion** 32

Chapter 4: Memory Packing 33
- **4.1 The Bin Packing Problem** 33
  - 4.1.1 Asymptotic Worst Case Ratio 33
  - 4.1.2 Deterministic Heuristics 34
- **4.2 State-of-the-Art Algorithms** 35
- **4.3 Hardware Constraints** 36
  - 4.3.1 Problem Definition 36

4.4 Metaheuristics **37**
4.4.1 Simulated Annealing. 37
4.4.2 Genetic Algorithm 37
4.5 Proposed Algorithms. **38**
4.5.1 The Next-Fit Dynamic Heuristic 38
4.5.2 Simulated Annealing Approach 40
4.5.3 Genetic Algorithm Approach 41
4.6 Conclusion **44**

Chapter 5: Evaluation **45**
5.1 Memory Packing Algorithm **45**
5.1.1 Test Methodology 46
5.1.2 Experiment Setup 46
5.1.3 Convergence Time 46
5.1.4 Mapping Efficiency. 48
5.2 Hardware Implementations **49**
5.2.1 Timing Closure 49
5.2.2 Synthesis Results 51
5.2.3 Memory Subsystem Frequency Scaling 53
5.3 Conclusion **54**

Chapter 6: Conclusions **55**
6.1 Research Questions **55**
6.2 Outcome. **56**
6.3 Contributions **57**
6.4 Future Work. **58**

Chapter A: Submitted Conference Paper (GECCO'20) **59**

List of Tables

2.1 BRAM: Supported Aspect Ratios [18] 8
2.2 Accelerator Specifications 23

4.1 Worst-Case Ratios for a Number of Classical BPP Heuristics [53, 54] 34

5.1 Baseline Dataflow Accelerators 47
5.2 SA and GA Hyperparameters 48
5.3 Memory Packing Comparison for the different Heuristics 49
5.4 Mapping Efficiency Increase (GA-NFD) 50
5.5 Timing Closure for a Number of Accelerator - FPGA Configurations 51
5.6 Packing Results 53

List of Figures

1.1 FINN Dataflow Architecture 2
1.2 Mapping Efficiency Decreases with Increased Parallelism 3

2.1 Overview of the Basic Building Blocks of an FPGA [16] 7
2.2 Xilinx Virtex Ultrascale+ 2.5D Planar Die Package [24] 10
2.3 Interconnect for SLR Crossings 11
2.4 Vivado HLS Design Flow 11
2.5 ANN Representation 13
2.6 A Typical CNN Topology [36] 13
2.7 Quantization vs. Top-5 Accuracy Trade-Off on the ImageNet Dataset [11] . . 15
2.8 Different Types of Inference Accelerator Architectures 16
2.9 The FINN End-to-End Tool Flow 17
2.10 Architecture of the MVTU Primitive 18
2.11 Schematic overview of the FINN Architecture 20
2.12 Terminology and Dimensions in Relation to Partitions 21
2.13 Packing Strategies 22
2.14 Network Topologies of the Accelerators 24

2.15 Residual Block Types 25
2.16 ResNet-50 Floorplan 25

3.1 Proposed FINN Architecture 28
3.2 Architectural overview of the Memory Subsystems 29
3.3 Asynchronous FIFO [47] 30

4.1 Illustration of the Bin Packing Problem 34
4.2 Packing Solutions Obtained by Applying Various BPP Heuristics 35
4.3 Bin Placement Checks 39
4.4 Item Per Gene Chromosome Encoding 41
4.5 Bin Per Gene Chromosome Encoding 43

5.1 VU13P: Reset Routing 52
5.2 Impact of the Memory Subsystem Operating Frequency on Power Consumption and Framerate 54

1 Introduction

Convolutional Neural Networks (CNNs) have demonstrated state of the art performance on image classification. In image classification the goal is to match input images to a corresponding class (i.e. the type of animal or vehicle). For CNNs, the accuracy with which the correct classes can be inferred strongly correlates with the amount of incorporated layers. For this reason, CNNs that are capable of classifying large datasets (i.e. the 1000 class ImageNet dataset [1]) with high accuracy, typically consist out of a large amount of layers [2]. Each of these layers are comprised of a number of neurons, and perform convolutions by sliding filters across the input feature maps. An input feature map can be the image presented at the input of the network, or the output generated by neurons from the preceding layer. Understandably, this process referred to as neural network inference, is very computationally intensive to perform for deep CNNs.

To enable fast image classification (i.e. for real-time applications), the execution of CNN inference has been accelerated by means of parallel execution on Graphical Processing Units (GPUs) [3, 4, 5], and Field Programmable Gate Arrays (FPGAs) [6, 7]. Moreover, traditionally the numerous multiplication and addition operations that are involved in CNN inference have been performed using floating point arithmetic. Research has demonstrated that reducing the bit precision of the CNN parameters (i.e. the filter values) dramatically reduces the storage and computational requirements at a minor penalty in terms of accuracy [8]. Evidently, the change from floating point representations of these parameters to quantized integer and fixed-point representations allows for more efficient implementations of neural network accelerators on FPGAs.

Indeed, FPGAs as a platform have demonstrated to be capable of reducing the required energy for neural network inference [9]. FINN is an end-to-end flow developed by Xilinx that exploits the computational benefits that are obtained from the quantization of neural networks, and aims to deliver low-latency, high-throughput and low-power FPGA inference accelerator designs [10, 11]. As input, FINN takes in the the bit precision of the parameters and the topology of a particular neural network consisting of the amount and types of operations per layer, and the sequence of the layers. Subsequently, it processes these specifications and reduces them to a sequence of elementary operations such as matrix-vector products and activation operations that can be implemented in hardware and mapped to the FPGA fabric.

FINN generates neural network inference accelerators based on a custom dataflow architecture (also referred to as a streaming architecture). As shown in Figure 1.1, in this architecture the input data are streamed in from the host processor, and the operations corresponding to a particular CNN layer are performed by dedicated compute units that are located in specific sections of the FPGA. Additionally, all of the compute units and parameter buffers that are required to implement the functionality of the CNN are instantiated on

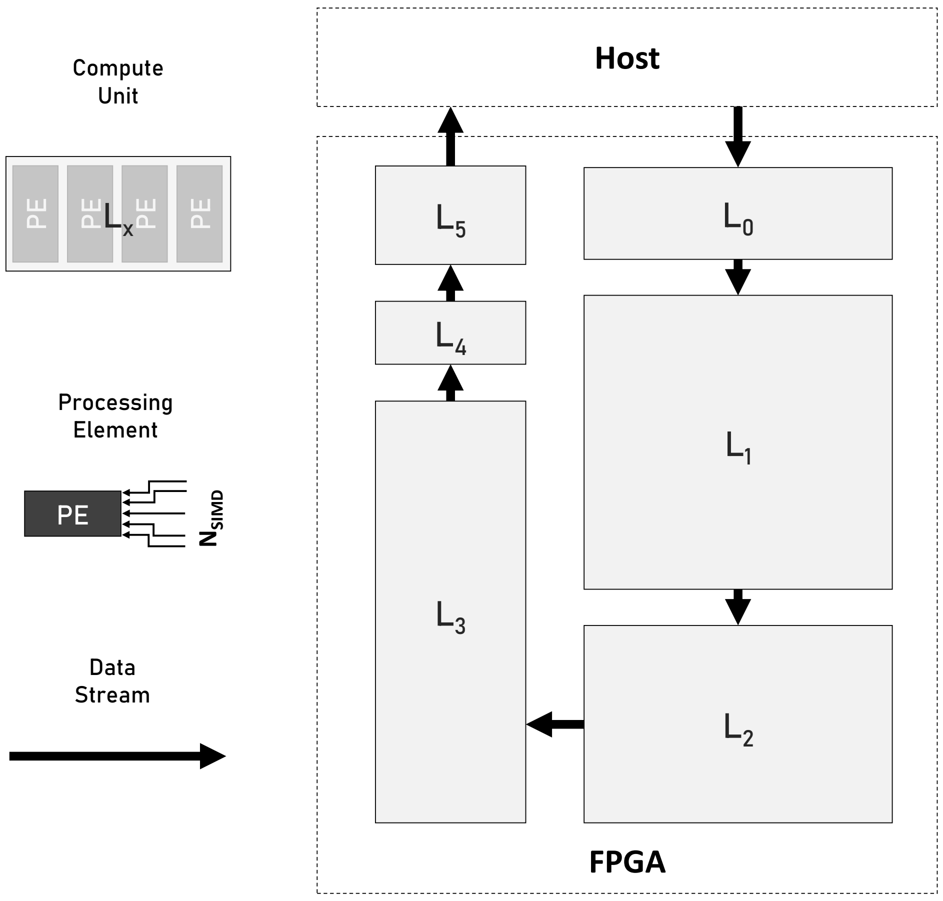


Figure 1.1: FINN Dataflow Architecture

the FPGA fabric, and connected in a pipelined fashion. In the most optimal case there is sufficient availability of resources on the targeted FPGA device, such that all of the operations that need to be performed by the compute units can be scheduled in parallel. This would give each compute unit a latency of one clock cycle, and would allow new input data to be classified per clock cycle. Realistically, in order to fit the accelerator design on a target device, the extent to which the compute units can be spatially unrolled is limited, and thus the execution of the various layers needs to occur by means of time-multiplexing. This means that certain operations have to be scheduled sequentially across multiple cycles; effectively sharing resources across time. To attain the best throughput for an accelerator design in terms of Frames Per Second (FPS), the individual compute units need to be latency matched. This is realized by modulating the amount of operations each compute unit will execute in parallel; governed by the amount of processing elements (N_{PE}) contained in each compute unit, and the amount of Single Instruction, Multiple Data (SIMD) lanes per PE (N_{SIMD}).

Naturally, to keep the compute units occupied, the parameter and input values need to be present at each clock cycle. This puts a specific constraint on the required throughput that the memory subsystem must provide. Since the available Block RAM (BRAM) modules on FPGAs have fixed shapes (i.e. 18-bit wide and 1024 addresses deep on Xilinx FPGAs), and a

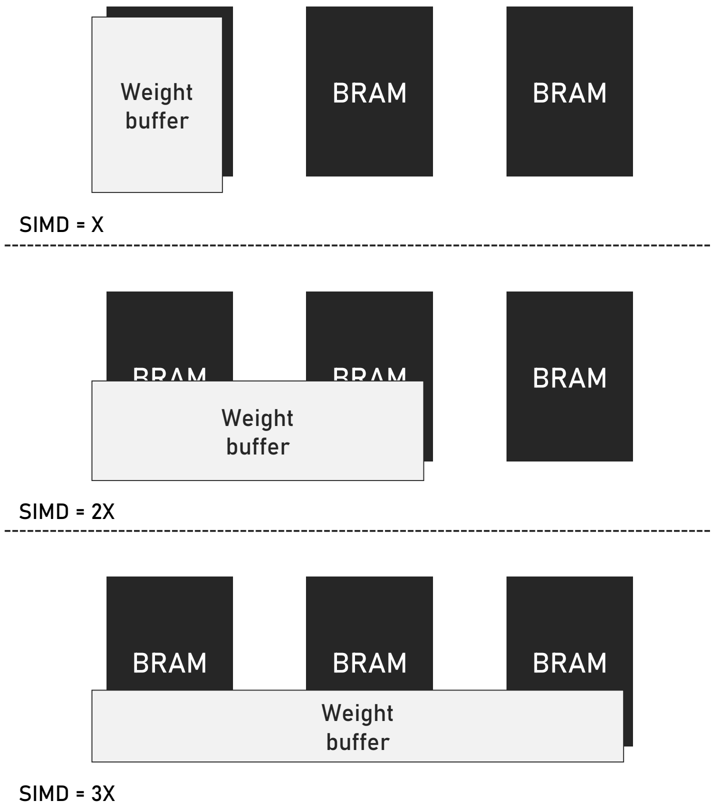


Figure 1.2: Mapping Efficiency Decreases with Increased Parallelism

limited amount of ports (thus supporting only a limited amount of read operations per clock cycle), it is sometimes necessary to partition the data across multiple RAM modules to meet the throughput requirement; even if the modules are not used to full capacity. As illustrated in Figure 1.2, especially for large, high performance accelerators that exploit a high degree of parallelism the mapping of weight buffers (i.e. the logical memory arrays containing the CNN filter values) to BRAM modules can lead to severe underutilization. This underutilization of an already scarce resource leads to the problem where FINN typically runs out of BRAM to store the parameter data before any of the other resources. For this reason, improving the BRAM utilization enables greater scalability, since this reduces the BRAM requirement for targeted devices. Although the parameter data for CNNs consist out of the various filter and threshold data (for activation of the neurons), the work in this thesis exclusively targets the memory utilization optimization of the filter data, since these dominate in terms of memory utilization.

The work described in this thesis aims to improve the On-Chip Memory (OCM) utilization by applying two techniques. Recognizing that the compute units are typically considerably more complex than the embedded BRAM and read-out logic, a new architecture is proposed where the memory subsystem is decoupled from the compute units and operated in a separate, higher frequency clock domain. This technique allows multiple weight buffers that require simultaneous access to be clustered within the same BRAM instance, while reducing or even completely eliminating the associated throughput penalties; depending on the attained operating frequencies of the two subsystems.

Finally, given the opportunity to cluster multiple weight buffers within the same BRAM instance, and the fact that the varying computational requirements of different CNN layers enforce different shapes on these buffers, the remaining problem is how we can form clusters of weight buffers such that the overall BRAM count is minimized. This problem is referred to as the memory packing problem [12], and is redefined and framed in this thesis to be a variant of the classical bin packing problem with custom constraints. The bin packing problem is a combinatorial optimization problem where items of varying sizes are grouped into a minimal amount of bins with fixed capacities [13]. To algorithmically identify the optimal configuration of clusters, a custom bin packing heuristic is proposed and embedded in simulated annealing and genetic algorithm metaheuristics. The algorithms converge to a solution within a couple of seconds for all designs that were evaluated within this work, although the genetic algorithm finds, in general, slightly better results. Furthermore, the identified packing solutions have been implemented in hardware where BRAM reductions of up to 30% are demonstrated. Especially within the context of Design Space Exploration (DSE), rapid convergence of the algorithm is an important factor in the required amount of time to find an optimal accelerator design; since memory packing needs to be performed for each design iteration.

1.1. Research Questions

The main objective of the research described in this thesis is to improve the utilization efficiency of BRAM in FINN style neural network inference accelerators. While striving to achieve this objective the following questions were central to the research process:

1. *"How can the BRAM utilization efficiency be improved without negatively impacting the system throughput?"*
2. *"What is the hardware cost to realize this approach?"*

These questions are addressed in a recurrent fashion throughout the different stages of the research described in the thesis, and are answered in the conclusion.

1.2. Contributions

The main contributions of the research described in this thesis are:

- a memory-efficient dataflow architecture for neural network inference accelerators on FPGAs
- the formulation of the memory packing problem as a variant of the bin packing problem
- a new heuristic for rapidly solving the memory packing problem
- an analysis of the impact of different memory packing strategies on: the maximum attainable operating frequencies, and BRAM utilization of neural network inference dataflow accelerators on FPGA

1.3. Thesis Outline

This thesis is structured as follows. In Chapter 2 relevant background theory is provided regarding neural networks, FINN, the architectures of the accelerators that were used as demonstrators, and the underlying FPGA technologies. The proposed architecture that enables us to cluster multiple buffers within a single BRAM module, without reducing the throughput of the accelerator is introduced in Chapter 3. The final part of the methodology is covered in Chapter 4, where we introduce the bin packing problem and reformulate this problem into the memory packing problem. In Chapter 5, the impact of the developed methodology is evaluated by applying it to a number of accelerator designs, targeting several FPGA devices. Finally, the thesis is concluded in Chapter 6, where the research questions and outcome of the pursued research are discussed, and pointers for future work are provided.

2
Background Theory

In this chapter relevant concepts that are required to understand the proposed methodology are presented. First, the overall hardware structure and available memory resources of Xilinx FPGAs are discussed. Subsequently, a brief introduction on neural networks and deep learning is provided. After establishing this foundation, the connection with the FINN framework is made, and the process of mapping neural network models to FPGA neural network inference accelerators is elaborated. Finally, the chapter is concluded with a comprehensive formulation of the On-Chip Memory (OCM) bottleneck, and introduction of the accelerator designs to be utilized for the validation and evaluation of the proposed methodology.

2.1. FPGA Infrastructure

Field-Programmable Gate Arrays (FPGAs) are devices that can be reprogrammed to change their functionality after production. A structural overview of the organization of a typical FPGA is provided in Figure 2.1. FPGAs primarily consist out of programmable Logic Blocks, which are commonly referred to as Configurable Logic Blocks (CLB), and programmable interconnect. These CLBs contain Look-Up Tables (LUTs), which are arrays that can be programmed to provide the desired output for a corresponding input. As such, these elements can be utilized to implement combinatorial logic or storage elements.

To satisfy the requirements of performance critical applications, such as Digital Signal processing (DSP), FPGA vendors also integrate hard-wired, embedded DSP units and Block RAM (BRAM) modules for fast multiplication and greater amounts of storage, respectively. The Input/Output Blocks (IOBs) enable fast communication between external devices (connected to the pins of the FPGA) and programmable logic (CLBs, DSPs, etc.) through the programmable interconnect.

Undeniably, FPGAs have undergone many transformations over the past decades [14]. These changes can be traced back to an increase in computational requirements for applications in different fields. However, with Moore's Law coming to a halt, the demand for heterogeneous computing solutions has risen [15]. The highly customizable and adaptable nature of FPGAs makes these devices exceptionally well suited to satiate these demands. Especially in the turbulent field of deep learning, FPGAs are capable of exploiting cutting edge developments by means of their reconfigurability. However, in order to extract the full potential of a particular device it is necessary to understand its underlying architecture and limitations. In this section we delve into the characteristics of the available memory resources, the AMBA standard and multi-die FPGAs, and touch briefly upon the topic of high level synthesis.

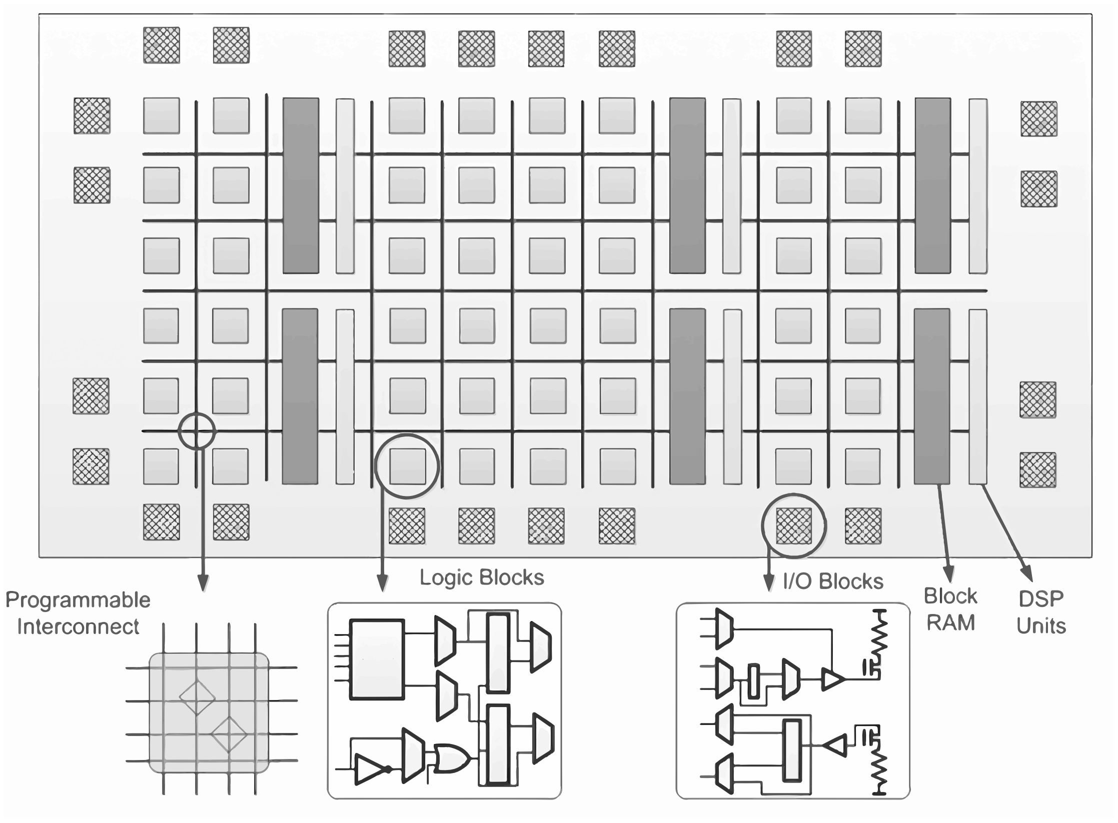


Figure 2.1: Overview of the Basic Building Blocks of an FPGA [16]

2.1.1. Memory Resources

In this section, the details of the memory resources that are available on Xilinx 7 series and Ultrascale+ FPGAs are presented. The increasing need for on-chip memory storage brought some changes to the architecture of FPGA devices. Generally, the trend has been to integrate larger, higher density memory resources to meet these demands. For efficient memory mapping it is vital to map smaller logical memories to smaller primitives as this prevents underutilization. Conversely, mapping large logical memories to small primitives results in routing congestion and resource depletion.

Distributed RAM

The smallest physical RAM that is present on Xilinx FPGAs is the distributed RAM located in the CLBs. This resource is also commonly known as LUTRAM and, as the name suggests, utilizes the memory cells in LUTs for data storage. More specifically, each CLB is subdivided in slices (containing the LUTs) that can be subdivided in two types: SLICEM and SLICEL. Exclusively the LUTs in SLICEM slices can be used as RAMs, since the LUTs in SLICEL slices lack a data input port, and are therefore only configurable at bitstream initialization [17]. It is, nonetheless, possible to use all LUTs as ROMs.

LUTRAM can be configured to produce RAMs that have specific dimensions and throughput properties. By default, each 6-input LUT can be used as a 1-bit × 64 addresses single port RAM. However, it is also possible to create dual and quad port RAMs out of multiple LUTs by sharing the write address ports of the LUTs. Additionally, as long as only one read/write per cycle is requested from the RAM, LUTs can be configured as Simple Dual Port (SDP) RAMs to efficiently implement wide memory arrays. For example, the 4 LUTs in a SLICEM can be combined to build a 3-bit × 64 addresses SDP RAM, as opposed to the 6 LUTs required to build the true dual port equivalent.

Block RAM

BRAM is embedded memory that is located outside of the CLBs. Compared to LUTRAM, Block RAM is used to implemented larger memories. The dimensions of the BRAM primitives in Xilinx FPGAs are 18-bit × 1024 addresses for the RAMB18 primitive and 36-bit × 1024 addresses for RAMB36 (essentially two cascaded RAMB18s). BRAM modules support a range of modes and aspect ratios to improve the mapping efficiency of logical memories.

To start with the aspect ratios, the width of the address and data bus of a BRAM can be configured to effectively change the dimensions of the memory array. The supported aspect ratios for the BRAM primitives are listed in Table 2.1. As can be seen, the primitives essentially support trading in some data bus bits for additional address bits.

Notable within the scope of the work in this thesis are the SDP modes. Since the poorly mapping weight buffers have shapes that can be characterized as wide and shallow, the SDP modes are a way to internally improve the mapping efficiency of those buffers. The data buses of the two read and write ports are combined, which results in a memory array that is twice as wide and shallow as the default configuration. Ultimately, an equivalent improvement in mapping efficiency can be obtained by simply utilizing the True Dual-Port (TDP) mode of the BRAM by packing two memories on top of each other. In general, since these memories have a greater storage capacity compared to LUTRAM, it is more likely that these resources are underutilized.

Table 2.1: BRAM: Supported Aspect Ratios [18]

(a) RAMB18

Data$_w$	Addresses
1	16384
2	8192
4	4096
9	2048
18	1024
36 (SDP)	512

(b) RAMB36

Data$_w$	Addresses
1	32768
2	16384
4	8192
9	4096
18	2048
36	1024
72 (SDP)	512

UltraRAM

The last memory resource is UltraRAM (URAM). URAM is the most dense memory resource on FPGAs, and was integrated in the latest families of Xilinx FPGAs to meet the ever increasing demand of on-chip memory. URAM modules have a fixed size of 72-bit × 4096 addresses and do not support any of the previously mentioned aspect ratios nor SDP modes.

Another limitation is that URAM cannot be initialized through the bitstream [19]. This makes this type of memory the least flexible and most difficult to integrate in FPGA designs.

Regardless, this memory resource offers vast amounts of storage capacity on the Virtex Ultrascale+ family of devices and, when efficiently used in a design, can reduce the on-chip memory shortage problems to a great extent. Furthermore, balancing out the utilization of the various memory resources is imperative to ease timing closure when working with multi-die FPGA design. However, the 4× increase in both width and depth relative to the RAMB18 primitive, combined with the lack of aspect ratio modes, causes the likelihood of underutilizing this resource to rise significantly.

2.1.2. The AMBA Standard

Aside from having sufficient on-chip memory that provides data at the required throughput, it is vital to have fast and scalable interconnect at your disposal to transfer that data to the compute units. The ARM Advanced Microcontroller Bus Architecture (AMBA) is a popular interconnect standard developed by ARM for the transfer of data between different hardware components [20]. Part of this standard is the popular and widely adopted interface known as Advanced eXtensible Interface (AXI).

The fourth generation of this interface, AXI4, is used extensively on Xilinx FPGAs to communicate with the various hardware blocks [21]. The AXI interface is a synchronous point-to-point connection between master and slave devices. Communication is established based on a combination of two handshaking signals. The master always initiates the communication, and data are transmitted on positive clock edges when the recipient asserts the READY signal, and the sender asserts the VALID signal. AXI4 supports a maximum burst size of 256 beats, and allows register slices to be placed on AXI4 interconnect to improve timing. Since AXI4 is a memory mapped interface, data transfers between the endpoints occur at specific address ranges.

Also part of the AMBA 4 standard is the AXI4-Lite interface. Similar to AXI4, this interface is also point-to-point and memory mapped. The difference is that the Lite variant does not support burst mode, which reduces the resource overhead for this interface when compared to the full AXI4 variant.

The last interface out of the series, AXI4-Stream, is not memory mapped but still point-to-point. Since AXI4-Stream has no addressing logic, there is less resource overhead. Additionally, the interface has an unlimited burst length. Beats can be transmitted on each rising clock edge as long as TREADY and TVALID are both asserted.

2.1.3. Multi-SLR Devices

The trend of developing chips with higher compute capabilities, as well as integrating greater amounts of memory, leads to an increase in die size [22], with some processor designs nearing the reticle limit [23]. Reliably manufacturing large chips is challenging and requires extensive R&D efforts. For this reason, building large monolithic dies rapidly becomes an uneconomical venture.

A strategy that can be employed in order to circumvent these difficulties, is to create 2.5D System-in-Package devices. Here the large design is partitioned across multiple chiplets interconnected through an interposer. Xilinx developed the Virtex Ultrascale+ devices using this design strategy, which they refer to as Stacked Silicon Ínterconnect (SSI) Technology. VU13P, for example, consists out of four stacked FPGA dies that are called Super Logic Regions (SLRs). This is illustrated in Figure 2.2. The wires that connect the SLRs through the interposer are called Super Long Lines (SLL).

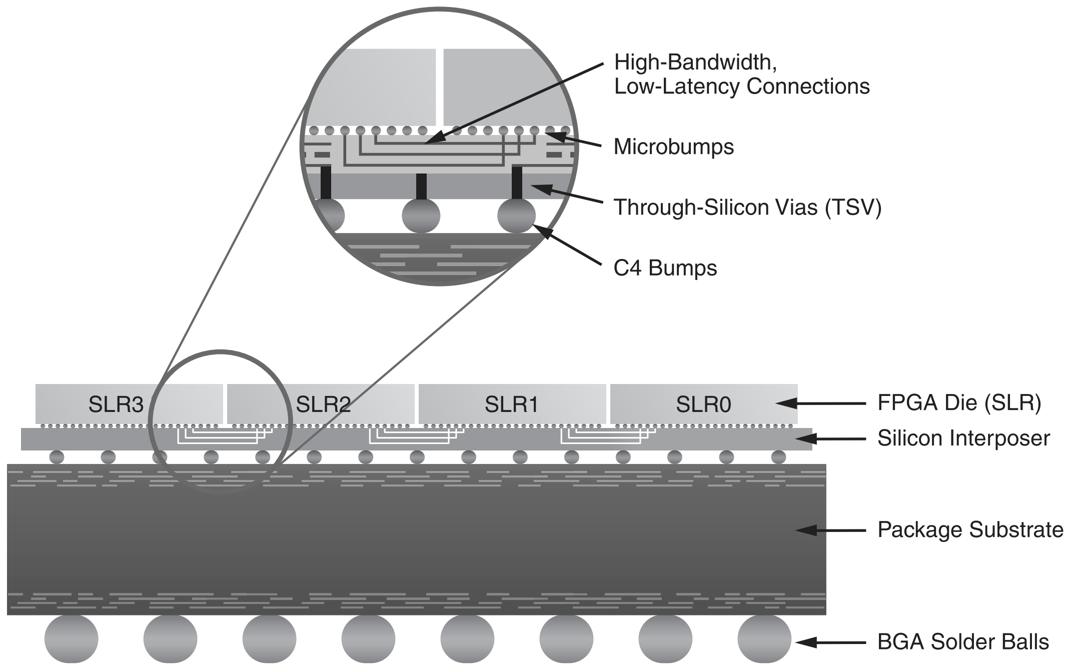


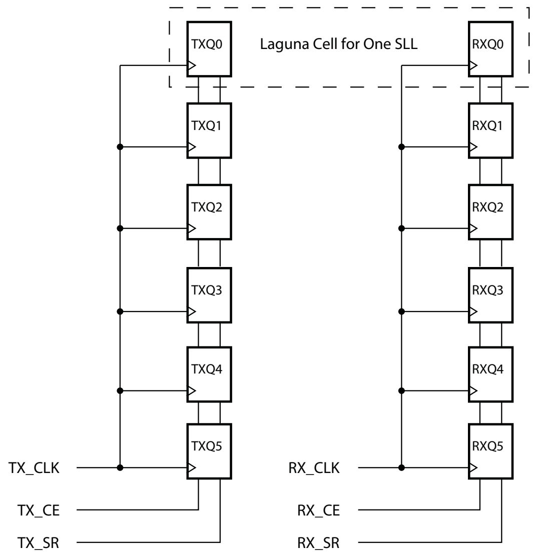
Figure 2.2: Xilinx Virtex Ultrascale+ 2.5D Planar Die Package [24]

While the SSI technology enables the design of large FPGA devices, these multi-die devices have inherent drawbacks compared to their monolithic counterparts. The most prominent problem is controlling clock skew and latency variance (due to potential process corner differences between dies) when transmitting data across dies [25]. Additionally, while SLLs might be low-latency compared to long PCB traces, communication across these lines is still an order of magnitude slower than on-chip wires. The amount of connections between SLRs is also rather limited, and must be taken into account during design.

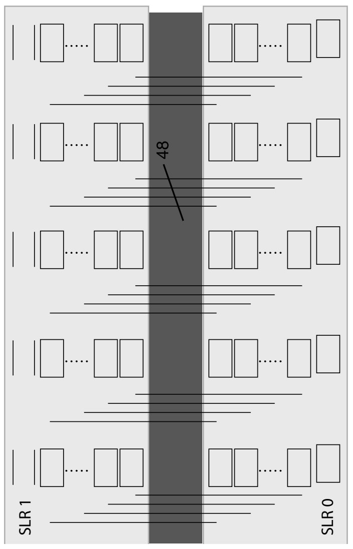
Inter-SLR communication is handled by the Laguna tiles. On Ultrascale+ devices each Laguna tile consists out of four Laguna sites. These sites each contain six full-duplex connections to SLLs that can optionally be routed through flip-flops for synchronous communications. A Laguna site and SLL connections to adjacent SLRs are shown in Figure 2.3. In the VU13P there are 3840 of such Laguna sites on each side of an SLR crossing junction. This shortage of SLL connections combined with the incurred latency for off chip communication, imply that it is essential to keep SLR crossings to a minimum if we want to maximize the operating frequency [26].

2.1.4. High-Level Synthesis

High-Level Synthesis (HLS) is a design process that allows hardware designers to create hardware through high-level language descriptions. The concept here is that a high-level language can abstract away many of the tedious details associated with the traditional HDL development flow [28]. In theory, HLS should therefore allow hardware designers to be more productive, and function as a bridge to enable software developers to design hardware to accelerate the computationally intense parts of their code. Additionally, functional verification



(a) Laguna Site [27]



(b) SLL Routing [24]

Figure 2.3: Interconnect for SLR Crossings

of the hardware can also be realized more easily at a higher level of abstraction.

Vivado HLS is a high-level synthesis tool developed by Xilinx that translates C/C++ descriptions of hardware into Register-Transfer Level (RTL) descriptions [29]. C variables, for example, are synthesized as wires or registers, and arrays as larger memory primitives like LUTRAM, BRAM or URAM. Hardware blocks are created with functions, and the function arguments of the function are inferred as ports. Verification of the hardware block can be done by creating a C program that calls the function representing the hardware block. After verification, the designed hardware block can be packaged and exported to Vivado for system integration. A typical design flow for Vivado HLS is displayed in Figure 2.4.

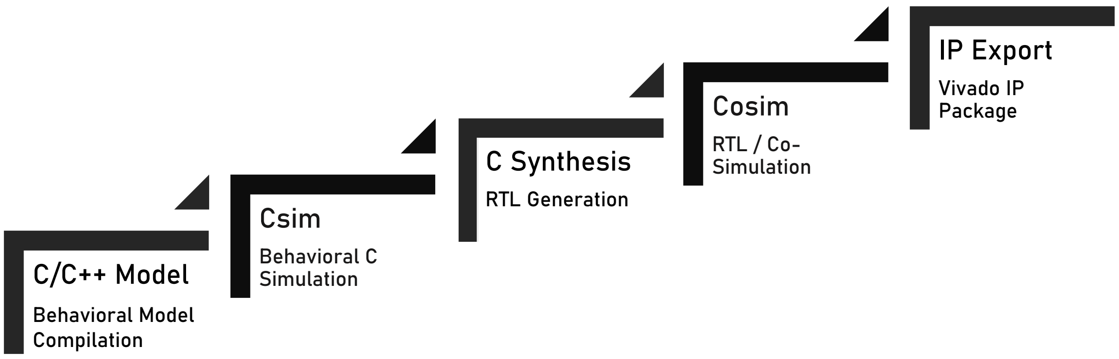


Figure 2.4: Vivado HLS Design Flow

By default, Vivado HLS does a best effort to find a good schedule and bind resources for the described hardware based on the target device and its speed grade, but it is possible to override this behavior. Performance optimizations and resource utilization management, for example, are realized by passing directives to the compiler. Variables can be remapped to different hardware resources, and latency constraints can be specified for the operations. In practice it is often necessary to optimize the design with directives to ensure that the optimal resource is selected.

Describing hardware with a high level language is beneficial in several areas, including address logic generation and the exploration of different architectures. The manual effort is handled by Vivado HLS on the background, and only the relevant parameters are exposed to the end user. However, when comparing the hardware generated through HLS with handcrafted RTL designs on performance and resource utilization, the results are often less optimal.

2.2. FINN

In this section the FINN end-to-end deep learning framework, the identified BRAM bottleneck, and neural network inference accelerator designs are detailed. Before we touch upon the main matter, it is important to first go through the fundamentals of neural networks to gain a better understanding of the involved concepts and terminology.

2.2.1. Neural Networks

Artificial Neural Networks (ANNs) are computational structures that are modeled after the human brain, and have as goal to enable a machine to learn to perform a certain task [30]. The fundamental processing elements within these networks, through which the input is propagated, are correspondingly named neurons. These neurons are grouped into a hierarchical structure of layers. As depicted in Figure 2.5, a neuron takes as input the sum of outputs of neurons from the preceding layer, which are connected through its synapses; following the model of the perceptron as introduced by Rosenblatt [31]. The value of this sum is compared to a certain threshold value, and depending on the outcome of this comparison, might cause the neuron to activate (i.e. produce an output signal). The value that is produced at the output of a neuron is determined by the employed activation function. This activation function can be linear or non-linear [32]; commonly employed in FINN accelerators are the Rectified Linear Unit (ReLU) [33] and unit step functions. The key parameters of the neuron are the weights w_n with which the inputs x_n to the synapses of the neuron are scaled, the activation function used for producing an output value, and the threshold value that determines whether the neuron should fire.

The value of these parameters are derived for all of the neurons within the network during a learning process that is referred to as training. Training can be supervised (where examples with answers are provided), unsupervised (where the goal is to find patterns without answers), or hybrid which is referred to as semi-supervised learning [34]. In this thesis we limit our scope to supervised learning, where the network learns to classify input data out of a number of known classes. When the values of the parameters are derived, the network should be capable of inferring the class of new data from outside the dataset with a certain accuracy. This accuracy is typically listed in terms of top-5 and top-1 accuracy, which corresponds to the case where the correct class is within the five highest or absolute highest probability of the prediction, respectively.

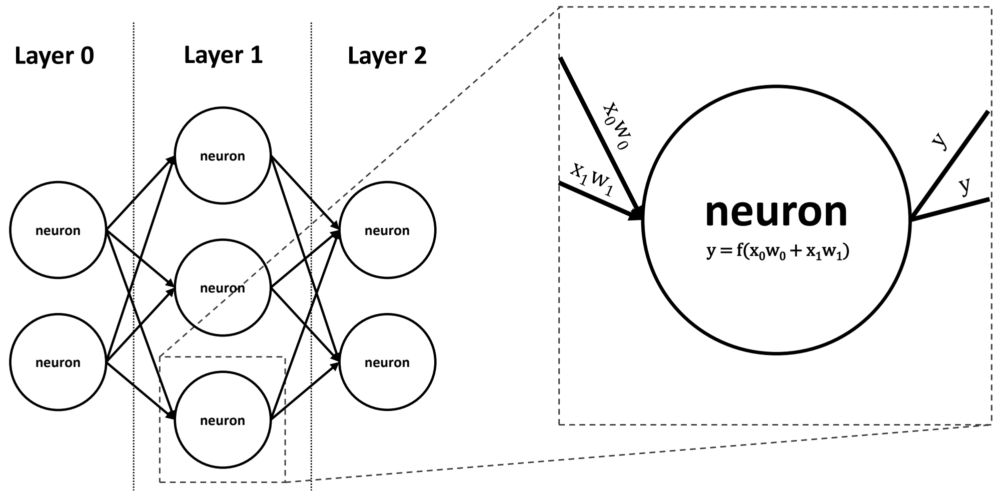


Figure 2.5: ANN Representation

Convolutional Neural Networks

For the work that is conducted within this thesis, the main goal has been to improve the memory utilization of inference accelerators that are targeted at image classification. The neural networks that are typically used for this purpose belong to a class of networks referred to as Convolutional Neural Networks (CNNs) [35]. A typical CNN topology is shown in Figure 2.6. These networks deviate from the ANNs that were described previously in a number of ways.

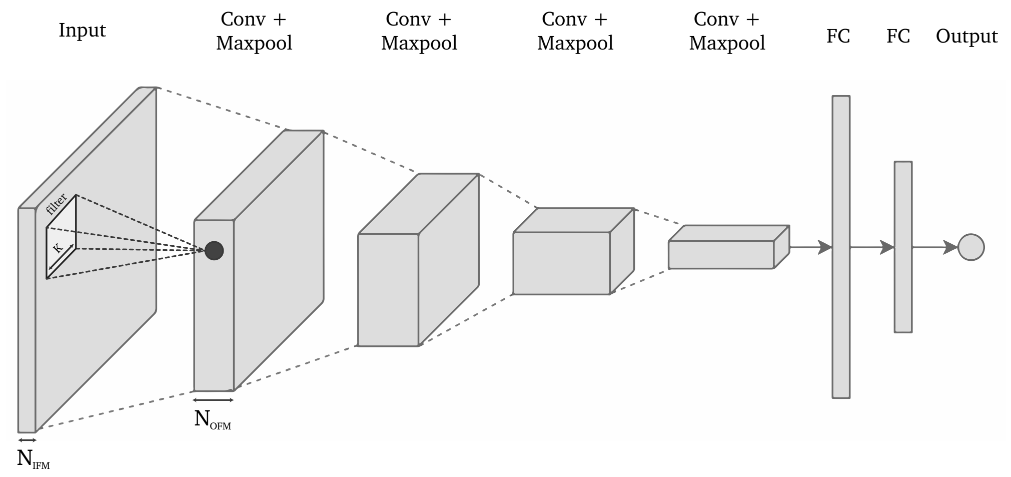


Figure 2.6: A Typical CNN Topology [36]

In the first place, CNNs contain so called convolutional layers where for each layer the convolution of a number of filters with an Input Feature Map (IFM) is performed. More specifically, each filter is a $K \times K \times N_{IFM}$ volume that slides across an input feature map containing N_{IFM} channels (i.e. for the first CNN layer the input feature map is typically an

RGB image containing three channels). Each convolution produces an output feature map, where each neuron contributes a single feature to an output feature map. The part of the filter that overlaps with the input feature map is referred to as the convolution window. The corresponding output value for the feature that is centered in the window is calculated by taking the inner-product of the filter and the convolution window. For symmetry purposes, K is typically chosen to be an odd integer. In contrast to ANNs, neurons in CNNs are only connected to the $K \times K \times N_{IFM}$ neurons that are within its convolution window. The exception are the neurons in the Fully Connected (FC) layers. These layers are included at the end of CNN topologies to produce the classification results [37], and are structurally similar to the layers in ANNs as depicted in Figure 2.5.

Finally, CNNs typically use a form of sub-sampling to reduce the size of the network. This sub-sampling is mostly implemented in the form of a max or mean pooling layer. In such layers a window is moved over the output of the preceding layer and, respectively, the maximum or mean values are selected.

Deep Learning

Since neural network layers are interconnected in such a way that each subsequent layer is capable of extracting a higher degree of abstraction, the accuracy with which predictions can be made typically increases with the amount of incorporated layers. This notion gave rise to the development of Deep Neural Networks (DNN) that were trained to accomplish more complex tasks [38]. The word “deep” refers to the fact that DNNs typically consist out of a large amount of layers.

It is evident that for large neural networks with many connected neurons the memory requirement for storing all of the weights and threshold values becomes large. Likewise, the computational intensity is considerable for performing convolutions and activation function evaluations for all of the neurons in the network.

Quantized Neural Networks

Following from the high computational requirements of large neural networks, extensive research has been pursued to improve the performance and reduce the footprint of neural network inference [39]. One of the explored avenues is quantization of the neural network parameters. Research has demonstrated that for large networks, it is possible to trade in a small amount of accuracy, for a significant reduction in terms of storage and computational requirements [8]. For FPGAs in particular, interesting trade-offs can be made between resource utilization and classification accuracy [11]. When reducing the parameter precision all the way down to binary representation, it is even possible to replace the costly multiplication and addition operations with simple XNOR-popcount operations [10, 40].

In Figure 2.7 a number of accelerators with different quantization configurations are compared on top-5 accuracy for classification on the ImageNet dataset and LUT utilization. From these data it can be seen that there is only a 4% accuracy decrease at a 250× reduction in LUT utilization when reducing the parameter precision from 32-bit floating point down to 8-bit integer.

2.2.2. FPGA Inference Accelerators

There are many different ways in which inference accelerators are implemented on FPGAs in the literature. Most common are overlay architectures and their variants [41, 42, 43]. Conversely, the accelerators that are used as demonstrators in this thesis are based on a

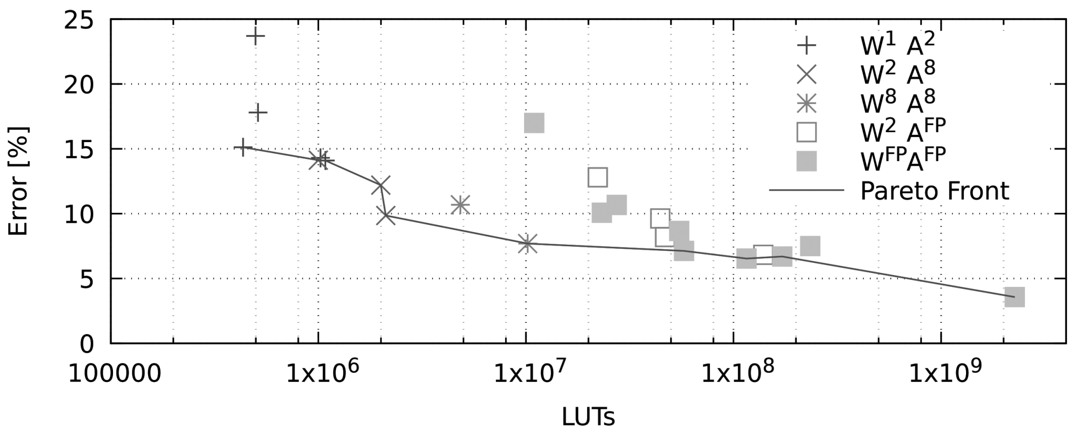


Figure 2.7: Quantization vs. Top-5 Accuracy Trade-Off on the ImageNet Dataset [11]

different architecture, which is referred to as a streaming architecture. FINN is based on this architecture [10]. A schematic representation of overlay architectures and streaming architectures is provided in Figure 2.8.

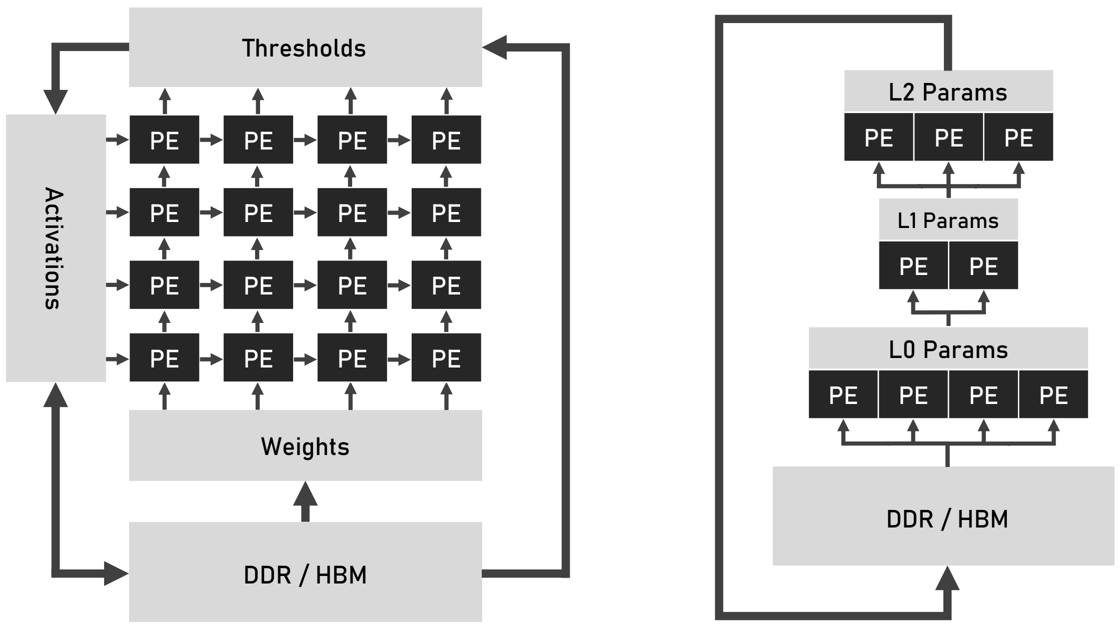
Overlay Architectures

The underlying concept of overlay architectures, is that the entire FPGA fabric is dedicated to execute a subset of the layers in a particular neural network. On one extreme end are overlay architectures that process neural networks on a layer-by-layer basis. An example of such an architecture is described in the work of Zhang et. al. [41]. Here, a systolic array of Processing Elements (PEs) is instantiated on the FPGA fabric, and is commonly shared for the computation of the operations corresponding to different layers. Once the execution of the operations pertaining to a single layer have been completed, the produced output data are stored in buffers, and the NN parameters of the subsequent layer are loaded on chip from external DDR or High Bandwidth Memory (HBM).

Lin et. al. mention that this “one-size-fits-all” approach leads to underutilization of FPGA resources, and report significantly higher computational throughput using a layer clustering approach [42]. In such an architecture, resources are shared between layers that have similar hardware requirements, and are placed in parallel next to layers that have complementary hardware requirements. Sharma et. al. take this approach one step further and divide the entire neural network into slices [43]. These slices consist out of a number of compute units, where each compute unit performs the functions of a particular layer. The authors attempt to fit as many compute units corresponding to subsequent layers as possible on the FPGA to promote data reuse, which ultimately improves performance and memory utilization efficiency.

Streaming Architectures

An alternative to overlay architectures are streaming architectures. In streaming architectures, per layer dedicated compute units are generated for each layer in the network. These compute units are composed out of multiple PEs that each perform the calculations for one neuron, or multiple neurons if the execution of a particular layer is time-multiplexed over several cycles. Finally, the compute units are scaled in terms of the number of PEs that are instantiated in



(a) Overlay Architectures **(b)** Streaming Architectures

Figure 2.8: Different Types of Inference Accelerator Architectures

parallel, which happens according to the computational requirements for the corresponding layer, and the available hardware resources of the FPGA, such that all compute units can be placed on the device. This approach is similar to that of Sharma et. al., but the difference lies in the fact that in streaming architectures, all of the dedicated per layer compute units and parameter values are placed on-chip, such that the data never have to leave the chip. Off-chip data transfers only take place when the input is streamed into the accelerator, and classification results are streamed out of the accelerator (typically to and from a host system). Furthermore, the compute units are pipelined and latency matched such that the computational throughput is maximized. This latency matching is required, since execution starts as soon as data are presented at the input of the compute units. In practice, this implies that the more computationally intense layers require more parallel processing power to prevent those layers from stalling the pipeline.

The immediate advantage of streaming architectures lies in the fact that their latency can be significantly lower compared to overlay architectures, as there are less data transfers involved in moving data between layers; the parameter values and output data are kept on-chip. Furthermore, the latency matching of layers based on their respective computational requirements leads to designs that maximally utilize the available resources, such that the highest possible performance can be obtained for the targeted device. Regardless, fitting the compute units and parameter data for all network layers on chip remains a challenge.

2.2.3. The FINN Framework

FINN is an end-to-end framework that takes advantage of parameter quantization to generate fast, scalable and efficient FPGA dataflow inference accelerators. In Figure 2.9 an overview of the tool flow is shown.

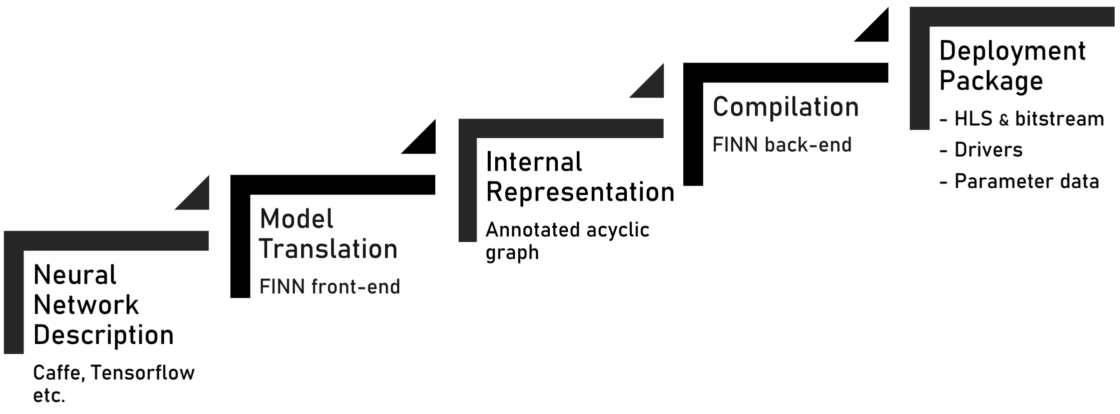


Figure 2.9: The FINN End-to-End Tool Flow

The description of a neural network is provided at the input, and is translated into an Intermediate Representation (IR) by the FINN frontend. The FINN compiler then does iterative passes encompassing: the application of transformations to the IR, and carrying out performance analyses to ensure that the provided throughput constraint can be met. When a good solution has been found, the compiler translates the IR into a sequence of elementary operations that can be mapped to hardware primitives from the `finn-hls` library. As final step, FINN generates the deployment package for a series of supported FPGA devices. The elementary operations that are supported by the `finn-hls` library are matrix-vector products (including activation), convolution input generation and pooling.

Matrix-Vector-Threshold Unit

The Matrix-Vector-Threshold Unit (MVTU) is the computational unit that is responsible for performing the matrix-vector products involved in the convolutional and fully connected layers. As mentioned before, the execution of the layers is time-multiplexed on the FPGA. In Figure 2.10 a schematic overview of the architecture of the MVTU, and its corresponding execution scheme are displayed.

The data in the input FIFO of the MVTU correspond to the data in the convolution window of the neurons. When the MVTU is activated (i.e. once the input FIFO contains data), each PE reads from the same stream, but applies a different filter to the data, and contains a different threshold value. Functionally, this correlates to the generation of features corresponding to neurons that operate on the same convolution window, but are located in different channels. Since the amount of operations that can be scheduled in parallel is limited, the generation of features for the entire output feature map is time-multiplexed over several cycles. It can be seen from the execution scheme that there are two parameters that control this latency: the synaptic fold S_F and the neuron fold N_F.

The synaptic fold controls the amount of multiplication and addition operations that are performed in parallel by each PE. The exact relation between the synaptic fold and the amount of SIMD lanes per PE is described by Equation (2.1a). As can be seen in the execution scheme in Figure 2.10, each PE processes its convolution window (i.e. the lightly shaded volume in the input feature map) in six steps or ‘folds’; the current fold is depicted as a solid, white cuboid in the input feature map. Therefore, the synaptic fold in the given example is six, which implies that the generation of a single feature takes six cycles.

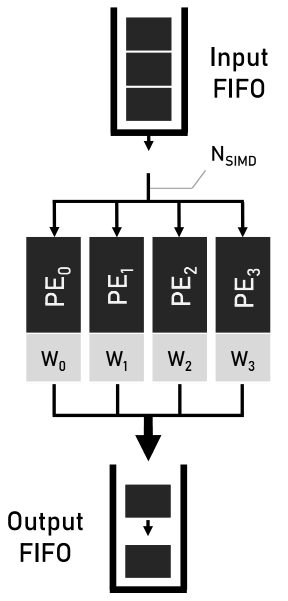
$$S_F = \frac{K^2 N_{IFM}}{N_{SIMD}} \tag{2.1a}$$

$$N_F = \frac{N_{OFM}}{N_{PE}} \tag{2.1b}$$

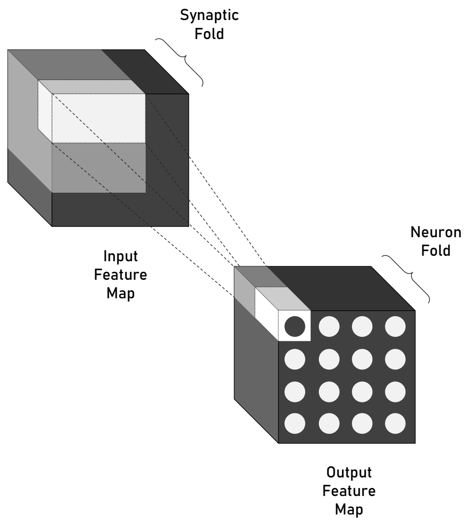
$$N_\ell = N_{OFD}^2 S_F N_F \tag{2.1c}$$

The amount of PEs that are instantiated in parallel is controlled by the neuron fold. In the given example there are four PEs instantiated in parallel, that each generate a feature corresponding to a different channel. The features that are being generated in the current fold are indicated by the solid white cuboid in the output feature map. From the execution scheme we can deduce that the neuron fold in this example is two. The overall latency in clock cycles of the MVTU for a particular layer can then be calculated by Equation (2.1c). Here, K is the filter size, N_{IFM} and N_{OFM} the amount of channels in the input and output feature map, and N_{OFD}^2 signifies the amount of features per channel in the output feature map; assuming the output feature map is square.

It can also be seen from the schematic overview that the weight buffers (i.e. the memory arrays containing the filter values for the layer) are stationary and partitioned across all PEs; where each PE requires parallel access to a different filter. This partitioning is necessary to guarantee that all of the filter values can be accessed in a single clock cycle. When the computation of the features for all channels has finished (i.e. corresponding to those neurons that share the same convolution window), the data are written to the output FIFO; activating the MVTU of the subsequent layer downstream.



(a) Schematic Overview



(b) Execution Scheme

Figure 2.10: Architecture of the MVTU Primitive

Sliding Window Unit

The Sliding Window Unit (SWU) converts and sequences the input feature map into a stream. This stream can then be forwarded to the MVTU in order to perform convolutions. In essence, the SWU sequences the input data such that a convolution can be performed by simply taking the inner product of the data in the input stream, and the filter values.

Pooling Unit

The pooling unit is responsible for implementing the maxpool sub-sampling functionality. Conforming to the FINN architecture, the pooling unit is implemented in dataflow style with line buffers. When the line buffer is filled sufficiently such that it is possible to perform the maxpool operation, the maximum value in the window is selected and forwarded.

2.2.4. FINN Architecture

The building blocks from the `finn-hls` library can be used to build custom inference accelerators. Figure 2.11 contains a schematic overview of an inference accelerator created with the FINN tool flow. As previously mentioned, the accelerator designs created by FINN are based on a streaming architecture. Starting from the top, the input images are streamed in from an external host. To accomplish this, the memory on host side is accessed through Direct Memory Access (DMA) over an AXI interface. Subsequently, the words that were loaded from the host are converted into a FIFO stream.

From this point forward, all operations are executed in a dataflow environment. This implies that as soon as the FIFO contains data, the functional unit that reads from this stream will be activated. In the example in Figure 2.11, the Sliding Window Unit starts converting the input data into a format that can be readily processed by the MVTU (which is also connected through a FIFO stream). As soon as the SWU has finished preparing the input data, the MVTU is activated, and commences with the generation of the rows of the output feature map. Once the line buffers in the Maxpool (MAX) unit are filled with a sufficient number of rows (such that the window of the maxpool unit can be filled), the MAX unit is activated and sub-samples the output feature map. At the very end of the pipeline, the data are passed to a Fully Connected (FC) layer where the image is classified, and after which the classification data can be sent back to the host. Since there are no convolutions in fully connected layers, no SWU is necessary.

2.2.5. The BRAM Bottleneck

The underlying problem that causes the BRAM bottleneck is a mismatch between the dimensions of the weight buffers and BRAM modules. What follows in this section is a deep investigation into the root cause of the BRAM bottleneck. In the process, the necessity of the proposed methodology is motivated, and the associated terminology is introduced.

Memory Shapes

To fully understand why the weight buffers map poorly to BRAM, we need to revisit the topology of CNNs. Convolutional layers perform convolutions by sliding a filter over the input feature map. In a typical CNN, as illustrated in Figure 2.6, we start with an image and attempt to extract meaningful features by applying various filters. Evidently, the amount of produced feature maps increases as we go deeper into the network.

Notice that the shapes of the cuboids reveal much about the computational requirements

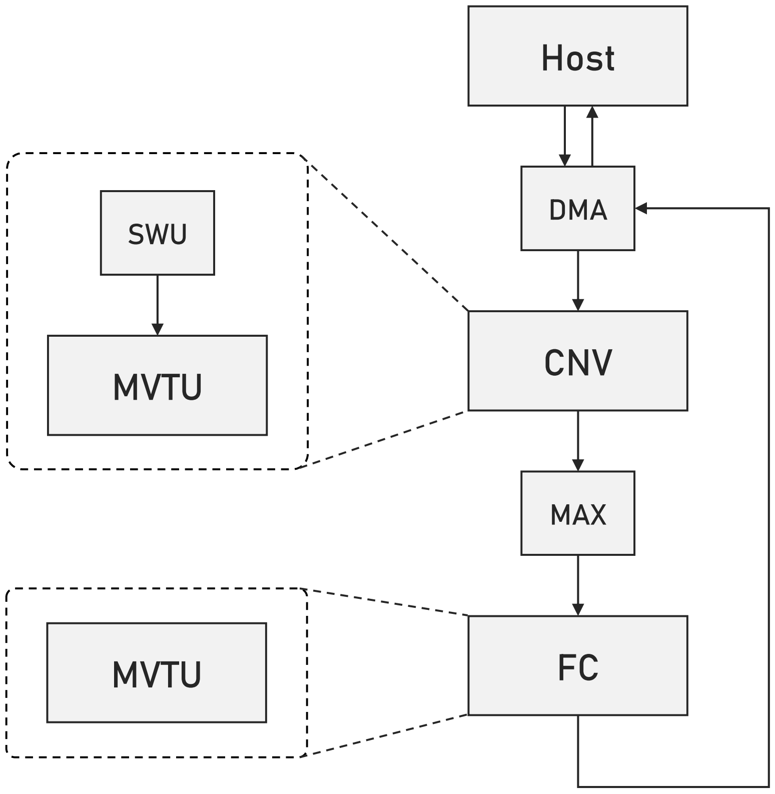


Figure 2.11: Schematic overview of the FINN Architecture

of the corresponding layers. The greater the frontal area of the cuboid corresponding to the preceding layer is, the higher the degree of data reuse for that layer will be. This follows logically when we consider that the same filter values will be moved across more data points. Now, the depth of the cuboid multiplied by the depth of the one in front of it is directly proportional to the memory footprint of the corresponding layer. Consider that different filters are required for processing each channel of an output feature, and that the depth of the filter equals the amount of channels of an input feature map. Finally, the volumes of these two cuboids reflect the computational requirement for the layer, since the amount of convolutions that have to be performed is equal to the amount of channels in the output feature map and the amount features inside each channel, and the complexity of each convolution grows with the amount of channels in the input feature map. The memory footprint of the weight buffers for a particular layer M_ℓ in terms of bits is as formulated in Equation (2.2a), and the computational intensity of a particular layer in terms of multiplications and additions is given in Equation (2.2b). The newly introduced variable W_P is the bit precision of the filter values.

$$M_\ell = K^2 N_{IFM} N_{OFM} W_P \tag{2.2a}$$

$$O_\ell = 2K^2 N_{OFD}^2 N_{OFM} N_{IFM} \tag{2.2b}$$

Owing to the fact that in FINN-style accelerators the execution of each layer is time-multiplexed or folded by a certain factor, only O_P operations are executed per cycle. To execute these operations within one cycle, parallel access to $O_P \times W_P$ bits of data is required. From this we can immediately derive that the width M_W and depth M_D of the weight buffers, are governed by Equations (2.3a) and (2.3b). The buffers are partitioned into multiple memories across PEs to ensure that the throughput constraints are met. These memories as depicted in Figure 2.12 are referred to as partitions from this point on.

$$M_W = O_P W_P = N_{PE} N_{SIMD} W_P \tag{2.3a}$$

$$M_D = \frac{M_\ell}{M_W} = \frac{K^2 N_{IFM} N_{OFM}}{N_{PE} N_{SIMD}} \tag{2.3b}$$

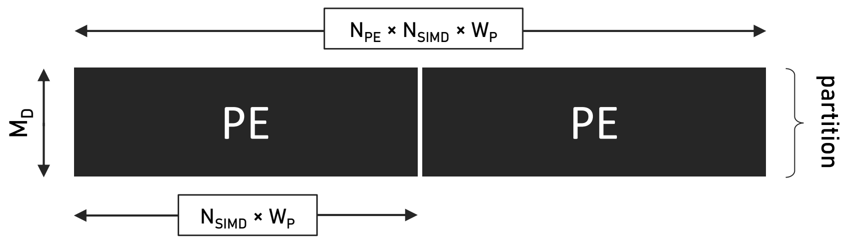


Figure 2.12: Terminology and Dimensions in Relation to Partitions

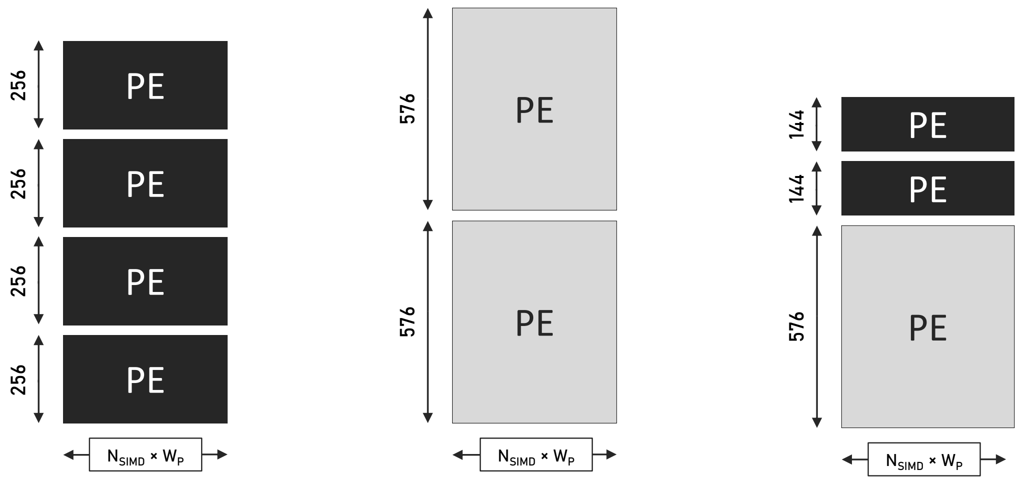
Taxonomy

Clearly, when we increase the amount of parallel accesses, M_W increases and M_D decreases correspondingly. Due to the different computational requirements of the individual layers, it is necessarily to modulate O_P to match their latencies. In a typical CNN one would therefore encounter partitions with the following shapes:

- **‘Wide’ and ‘shallow’**: corresponding to layers that are computationally intense and exploit a significant amount of parallelism.
- **‘Narrow’ and ‘deep’**: corresponding to layers that are less computationally intense and are time-multiplexed to save resources.

It is these wide and shallow buffers that have low mapping mapping efficiencies, and are the main contributors to underutilization problems of BRAM resources. The mapping efficiency of a partition is as defined in Equation (2.4). The numerator in this equation denotes the memory footprint for a particular layer M_ℓ, while C_{BRAM} stands for the ‘BRAM bits’ we actually allocate to store the corresponding weight buffers on chip (i.e. the storage capacity of a BRAM module multiplied by the amount of synthesized BRAM modules).

$$\eta_{BRAM} = \frac{N_{PE} N_{SIMD} W_P M_D}{C_{BRAM}} \tag{2.4}$$



(a) Intra-Layer ($K = 1$) **(b)** Intra-Layer II ($K = 3$) **(c)** Inter-Layer

Figure 2.13: Packing Strategies

Mapping Efficiency Optimizations

The central objective in this thesis is to improve the mapping efficiency of weight buffers to BRAMs. A selection of possible mapping optimization strategies are displayed in Figure 2.13. In Figure 2.13a, the simplest optimization strategy is shown. Here we pack multiple poorly mapping partitions on top of each other to fill up the empty space and improve utilization. However, an important aspect to keep in mind is the K^2 factor in Equation (2.3b). Typically, filters are centered around features for symmetry; popular dimensions are 3×3, 5×5 and 7×7. Resultant from this, are buffers that have depths (M_D) equal to odd multiples of powers of two and potentially map poorly to BRAM, which is an 18-bit × 1024 addresses deep array. In Figure 2.13b we can see that simply packing multiple of these buffers on top of each other might not immediately improve their mapping efficiencies, since each individual partition does not evenly divide the depth of the BRAM.

Due to latency matching, it might be possible to combine partitions from layers with different folding factors, such that their combined depths approaches the depth of a BRAM primitive, or an integer multiple thereof. We define the case where we combine partitions from the same layer as intra-layer packing. Conversely, when we combine partitions from different layers as depicted in Figure 2.13c, we denominate it as inter-layer packing.

Regardless, since each BRAM module contains only two ports, we can not access all partitions in parallel if we cluster more than two parameters in a single module. As such, the computational throughput of the accelerator would be degraded if the strategies as displayed in Figure 2.13a or 2.13c were directly applied in the architecture as discussed in the previous sections. Consequently, a new architecture is proposed in Chapter 3 that allows the BRAM to operate at a higher frequency, which allows each BRAM to serve more than two requests on the same data port within one compute cycle. Considering that this strategy allows us to pack more than two partitions without violating the throughput constraints, and we have the possibility to combine partitions from the same or different layers, there is now a potentially vast amount of packing configurations to explore. This very problem is what we refer to as the memory packing problem, which is addressed in Chapter 4.

Table 2.2: Accelerator Specifications

Accelerator	FPS	kLUTs	F_{clk} (MHz)	Efficiency	Dataset	Device
CNV-W1A1	632	25.7	100	69.3%	CIFAR-10	Z-7020
CNV-W1A2	632	37.7	100	69.3%	CIFAR-10	Z-7020
CNV-W2A2	206	32.9	100	79.9%	CIFAR-10	Z-7020
RN50-W1A2	2015	787	205	52.9%	ImageNet	VU13P

2.2.6. Accelerator Designs

The proposed methodology has been implemented, validated and evaluated on several inference accelerators. The specifications and implementation details of the accelerators and targeted FPGA devices are provided in this section.

BNN-PYNQ

The first class of accelerators are targeted at low-power embedded platforms like the PYNQ-Z1/Z2 and Ultra96 that are based on, respectively, the Zynq-7020 and Ultrascale+ ZU3EG SoCs. The BNN-PYNQ suite [44] contains three relatively simple CNNs that support CIFAR-10 [45] and SVHN [46] classification. The topologies of these networks can be found in Figure 2.14. Table 2.2 contains the throughput for each accelerator design in terms of Frames Per Second (FPS) as evaluated on the respective datasets, the amount of required LUTs to implement the accelerator designs, the attained clock frequency, and the mapping efficiency of exclusively the weight buffers to BRAM resources as defined in Equation (2.4).

As can be seen in Figure 2.14, the three CNV accelerators that are included in this work all share the exact same topolgies, but differ in parameter precision, and thus, the extent to which the accelerator designs are spatially unfolded. CNV-W1A1 and CNV-W1A2 share the same folding factors S_F and N_F, but have different values for the precision of the activations (i.e. the produced features). Since exclusively the optimization of weight buffers to BRAM mapping is targeted in this work, CNV-W1A2 was omitted from the evaluation. More interestingly, the CNV-W2A2 accelerator shares the same topology with the two aforementioned accelerators, but contains ternary (2-bit) weights as opposed to binary weights. As such, in order to fit the CNV-W2A2 accelerator design on the Z-7020 FPGA, the folding factor had to be increased. Looking at Table 2.2, it can be seen that the increase in the folding factor is directly reflected in the mapping efficiency for the weight buffers. Since the amount of exploited parallelism is decreased relative to the CNV-W1A1 and CNV-W1A2 designs, the weight buffers are less shallow, which results in less underutilization of BRAM resources.

ResNet-50

The quantized ResNet-50 accelerator was designed for high-throughput and high-accuracy image classification on the ImageNet dataset. This accelerator targets the Alveo U250 board that is based on the Virtex Ultrascale+ VU13P FPGA.

The ResNet-50 network was developed by He et. al. from the Microsoft research team. This team decisively won the Large Scale Visual Recognition Challenge of 2015 [2]. He et. al. introduced the concept of residual learning to solve the vanishing gradient problem, which significantly improved the accuracy of deep neural networks. The topology of the Resnet-50 accelerator used in this work is laid out in Figure 2.14. The network consists out

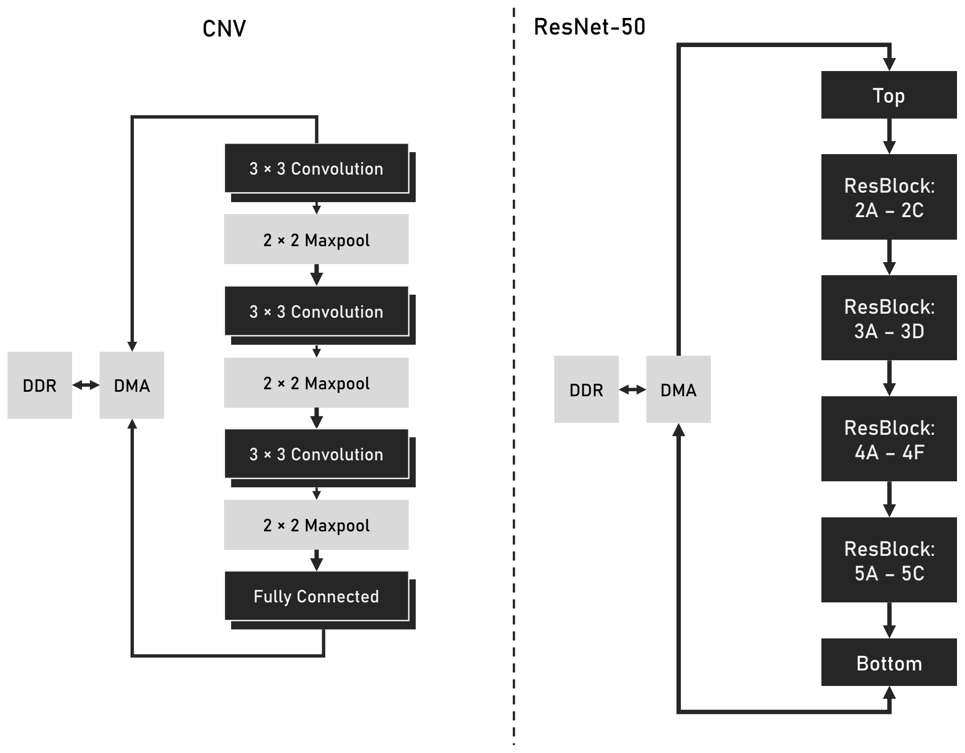
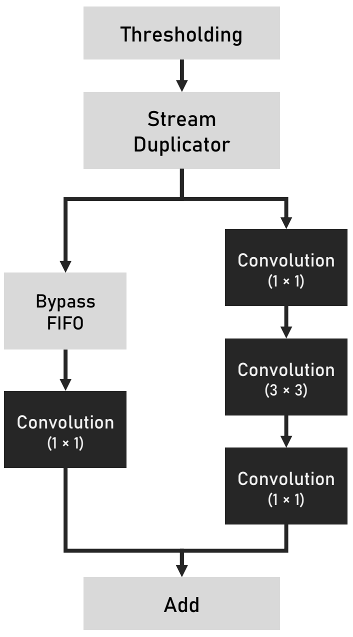


Figure 2.14: Network Topologies of the Accelerators

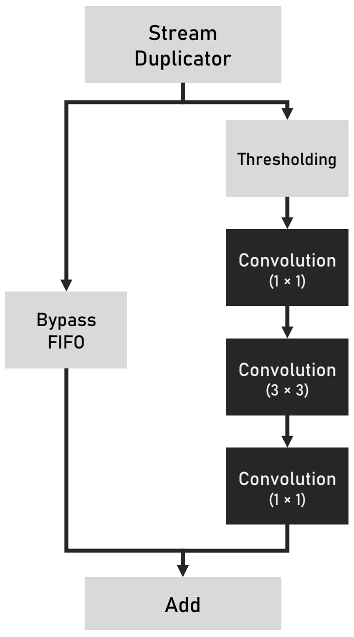
of a sequence of 16 residual blocks, with fully connected layers at the very top and bottom. Each residual block consists out of a number of convolutional layers and a bypass path. The exact configuration differs per type of residual block, which are displayed in Figure 2.15. The main difference is that the Type-A blocks (i.e. ResBlock: *A) have 1 × 1 (referring to the dimensions of the filter) convolutional layers on the bypass path, while the Type-B blocks (ResBlock: *B-F) simply forward the input.

In total, the entire quantized ResNet-50 network consists out of 52 convolutional layers and 2 fully connected layers. As expected, it is quite challenging to fit all of the layers on a single device; especially, when we consider the poor memory utilization figures as displayed in Table 2.2. This design was mapped to the largest commercially available device: the VU13P. Since this is a multi-die FPGA, there are additional floorplanning difficulties associated with this design.

The floorplan for the ResNet-50 design is given in Figure 2.16. Here the aim is to distribute the residual blocks across the SLRs such that the resource utilization is spread evenly, and the amount of SLR crossings are minimized. This balancing of resources primarily consists out of balancing the BRAM requirements to implement the weight buffers. The Res2x blocks have a small memory footprint, and have their weight buffers mapped to LUTRAM. The residual blocks with the largest storage requirements are the Res5x blocks and, for this reason, are each placed on separate SLRs. In total, the ResNet-50 design has 12 SLR crossings.



(a) Type-A



(b) Type-B

Figure 2.15: Residual Block Types

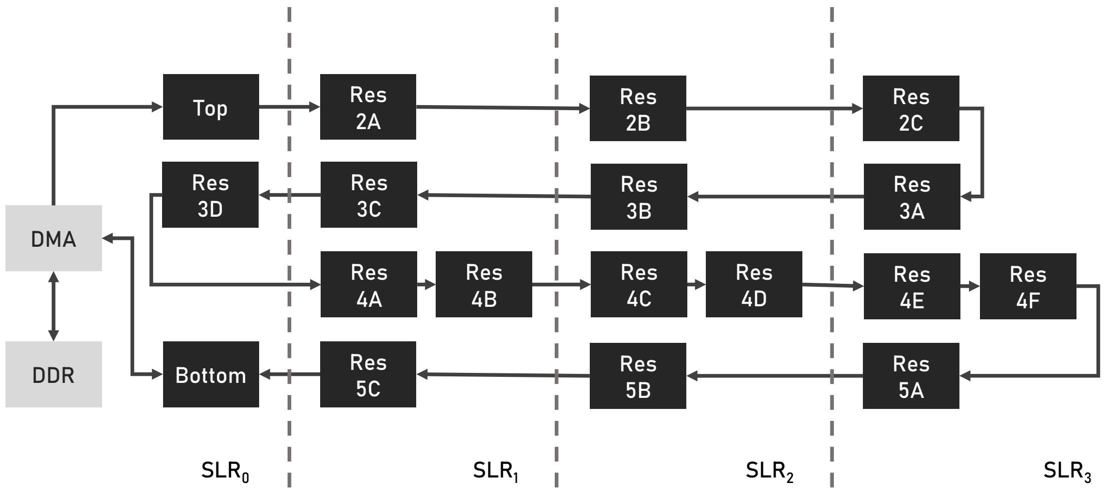


Figure 2.16: ResNet-50 Floorplan

2.3. Conclusion

In the first part of this chapter we covered the infrastructure of Xilinx FPGAs, and found that these FPGAs consist primarily out of programmable interconnect and logic elements (LUTs). Additionally, modern FPGAs incorporate hard-wired, embedded devices for vast amounts of storage and fast arithmetic. The AMBA standard is used for communication between different hardware blocks, and the largest Xilinx FPGAs consist out of multiple dies that are connected through a silicon interposer.

In the second part, the FINN framework was introduced, which aims to generate fast and scalable neural network inference accelerators for FPGAs. The accelerators that are generated by FINN can be built with a set of elementary hardware blocks from the `finn-hls` library. We explored different inference accelerator architectures, and found that FINN generates accelerators based on a streaming architecture, which have a lower associated latency compared to overlay architectures. The drawback of streaming architectures, is that it can be difficult to fit all of the hardware on the FPGA. Relating to this, we investigated the BRAM bottleneck, and found that these resources are easily depleted when the degree of exploited parallelism is increased. Finally, the accelerator designs that are to be utilized for validation and evaluation have been introduced.

In the next chapter, we propose a new architecture that enables more efficient utilization of the BRAM resources.

3
Decoupled Architecture

In this chapter the first part of the methodology is discussed. Here a new architecture is proposed in which the memory subsystem is operated in a separate, higher frequency clock domain. This allows us to scale up the operating frequency of the memory, and pack more buffers within the BRAM modules. The aim of this approach is to allow for more flexible allocation of memory, such that the BRAM resources can be utilized more efficiently.

This chapter is structured as follows. First an overview of the proposed architecture is provided. Subsequently, the concept of asynchronous FIFOs is clarified. Lastly, the operations of the involved hardware, and the access to the external memory are addressed.

3.1. Architectural Overview

In the original FINN architecture, all of the weight buffers were assumed to fit on chip, and were integrated as buffers in the compute units. As previously mentioned, this assumption combined with the limited amount of ports, and specific dimensions of the BRAM modules leads to a memory bottleneck. An overview of the proposed architecture is shown in Figure 3.1. Compared to the original architecture in Figure 2.11, it can be seen that the MVTU now reads from an additional stream, the weight stream, which contains the filter values for each of the PEs. This streaming interface allows us to store the filter values in different locations, and essentially separates or 'decouples' the memory subsystem from the MVTU. For example, in Figure 3.1 we can see that the filter values for the CNV layer are streamed from the on-chip memory, while the filter values for the FC layer are streamed from the external memory. From this point forward, the original FINN architecture is referred to as the *integrated weights architecture*, and the proposed architecture is denominated as the *decoupled weights architecture*.

Figure 3.2 displays the differences between the integrated weights and decoupled weights architectures in more detail. It can be seen that in the integrated weights architecture the weight buffers are directly accessed by the PEs, and thus we have as requirement that both elements are clocked at the same frequency. By contrast, we do not have such constraints in the decoupled weights architecture. Additionally, we can see that in the decoupled weights architecture, the stream generator is responsible for fetching the various partitions of the weight buffers, concatenating them and then streaming them to the MVTU. Now the MVTU requires a stream splitter to dissect the streamed data, and forward them to the designated PE.

Since the memory subsystem may now reside in a different clock domain, there is a need for synchronization. This is handled by an asynchronous AXI Stream FIFO [47]. The stream generator in the memory clock domain fills up the FIFO until it is full, and refills the FIFO

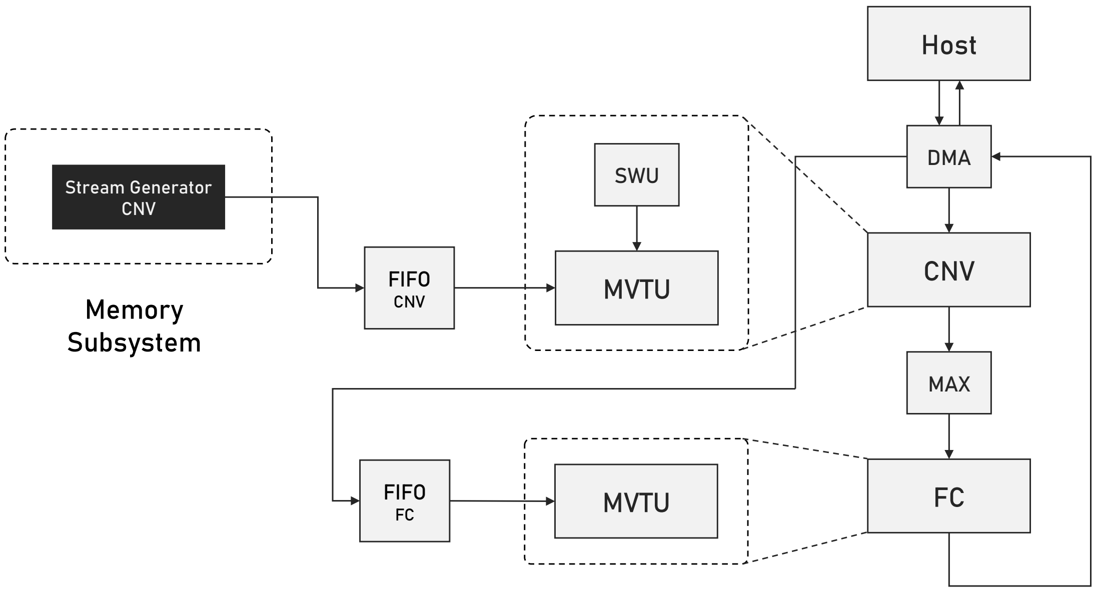
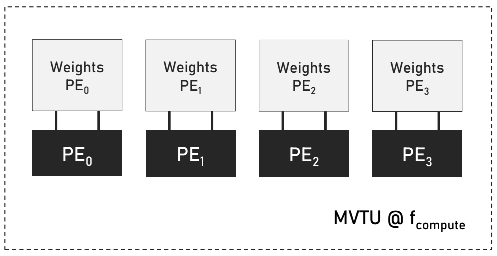


Figure 3.1: Proposed FINN Architecture

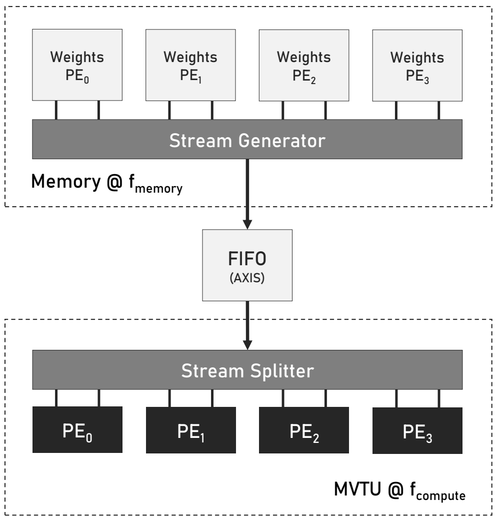
as parameter values are requested and read out by the MVTU. This essentially creates a producer-consumer system where the stream generator is the producer, and the MVTU the consumer. If the memory subsystem is clocked at a higher frequency than the MVTUs, this structure allows for a latency of greater than one memory clock cycle, without stalling the pipeline. The amount of memory partitions that can be packed in a single BRAM without incurring throughput penalties H_B, is now dictated by the ratio of the memory clock F_m, and compute clock F_c, multiplied with the amount of ports of the memory primitive N_p, as defined in Equation (3.1). From this point forward, $\max\{H_B\}$ is referred to as the maximum bin height.

$$H_B \leq \max\{H_B\} = \left\lfloor N_p \frac{F_m}{F_c} \right\rfloor \tag{3.1}$$

As can be seen in Equation (3.1), in order to increase the maximum bin height by one, the operating frequency of the memory subsystem must be increased by F_c/N_p. Regardless, due to the producer-consumer mechanism, the computational throughput decreases gracefully if the maximum attainable value for F_m does not fully satisfy Equation (3.1). To illustrate this notion, consider the case where we have an accelerator design in which the MVTU operates at 200 MHz, and we try to pack four partitions in BRAM. Additionally, suppose that it is at most possible to meet timing closure at 395 MHz for the memory subsystem. Then the expected computational throughput will only be slightly impaired compared to the case where Equation (3.1) would have been satisfied (i.e. for $F_m = 400$ MHz). As such, the decoupled weights architecture can offer a fine-grained trade-off between computational throughput and BRAM utilization.



(a) Integrated Weights



(b) Decoupled Weights

Figure 3.2: Architectural overview of the Memory Subsystems

3.2. Asynchronous FIFO

Moving data from one clock domain to another domain that appears asynchronous to it, leads to data hazards. In such multi-clock systems, the data are sampled at irregular intervals, which leads to a phenomenon that is referred to as metastability. More specifically, the flip-flops in digital circuits have specific hold and setup time constraints. If the sampled data are changed within this time window, the values of the data at the output of the flip-flop are unpredictable. In general, when dealing with multiple clock signals, some form of synchronization is required at the boundaries where signals cross between clock domains.

Metastability can be avoided by transferring data through asynchronous FIFOs. These storage elements can be filled up to a certain capacity (referred to as the FIFO depth), and can be read out sequentially when filled. An asynchronous FIFO as supported on Xilinx FPGAs is depicted in Figure 3.3. The read and write operations are performed in their respective clock domains. The synchronization of data that are transmitted between these clock domains is accomplished through a number of auxiliary signals. The signals ‘full’ and ‘empty’ respectively flag whether there is space left to store data, or that there are data available to be read out. In order to know whether the FIFO is full or empty, the read and write pointers have to be synchronized and compared. This synchronization implies that a certain amount of latency is incurred in the comparison of these pointers. A write operation, for example, affects the empty signal in the read clock domain several clock cycles later. The same is true for read operations and the full signal in the write clock domain. In such scenarios, the ‘programmable’ or ‘almost’ full and empty signals are necessary to anticipate whether the respective limits are neared; which are triggered when a predefined fill level is reached.

3.3. Stream Generator

The stream generator is responsible for fetching all the weight values for a specific layer. In each memory clock cycle it reads out the required data for all PEs in a particular layer, and

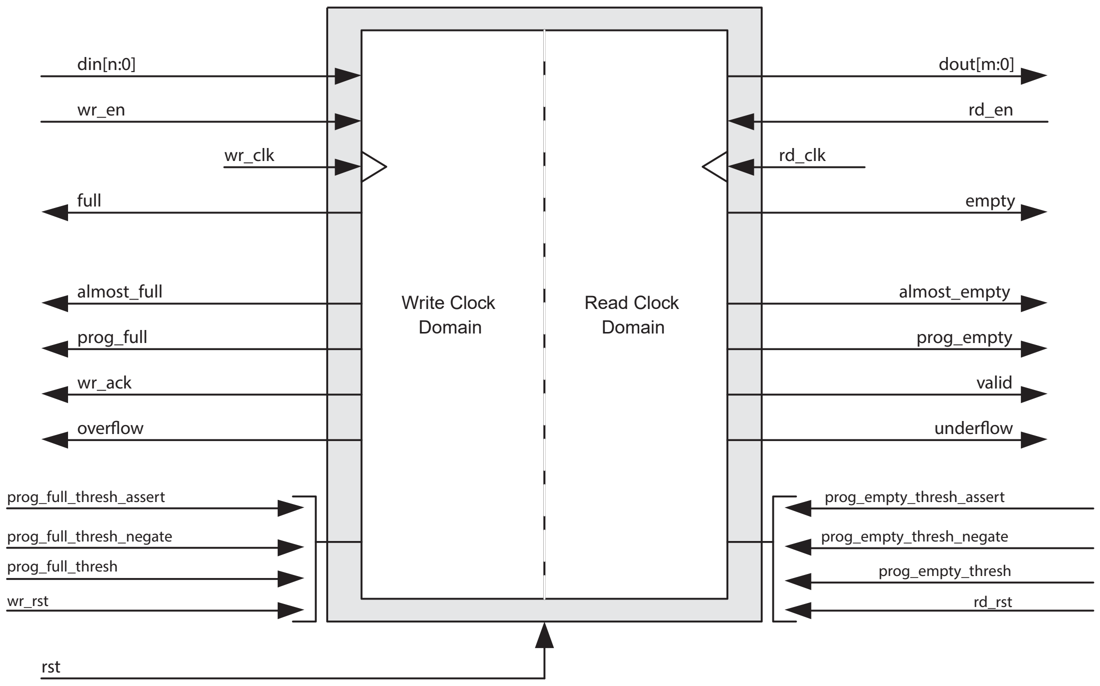


Figure 3.3: Asynchronous FIFO [47]

concatenates the partial words into an S_W wide word. The value of S_W is given in Equation (3.2). Here N_{PE} is the amount of PEs that are instantiated in parallel for a particular layer, N_{SIMD} is the amount of SIMD lanes per PE, and W_P is the precision of the weight values. This wide word therefore contains all the required data for one cycle of execution of the corresponding MVTU. After these data are concatenated, the word is streamed through an AXI Stream interface, and written into a FIFO.

$$S_W = N_{PE}\, N_{SIMD}\, W_P \tag{3.2}$$

The transactions between the MVTU and stream generator are arbitrated by a set of handshaking signals. When the FIFO is not full, parameter values are written into the FIFO, and TVALID is asserted. From the MVTU side, processing can commence by asserting TREADY when the FIFO is not empty, and TVALID is asserted. Due to this mechanism, it is important that the stream generator streams words fast enough to keep the MVTU occupied.

3.4. Hardware Costs

Efficiently transferring data between clock domains is essential to maintain throughput. As previously mentioned, asynchronous FIFOs are used as synchronizer structures between clock domains. In principle, the FIFO should simply be deep enough to cover for the latency between read out from the MVTU side, and refill from the stream generator side. For the accelerator designs that were included in this work, the asynchronous FIFOs were most efficiently implemented with a FIFO depth of 32. This FIFO depth maps well to the LUTRAM resources in SDP configuration.

The hardware cost for applying this methodology is largely due to the required LUTRAM to implement the FIFOs. As can be seen in Equation (3.3), this cost C_{LUT} is linearly proportional to the width of the stream. For this reason, the memory packing strategy needs to be considered on a case by case basis, to determine whether or not the LUTRAM expenditure is worth the savings in terms of BRAM.

$$C_{LUT} \approx \frac{2S_W}{3} \tag{3.3}$$

3.5. External Memory Access

As shown in Figure 3.1, the decoupled weights architecture also allows us to store the weight values in external memory. Unfortunately, FPGA devices typically have relatively low bandwidth to the external memory compared to alternative platforms that are typically used for neural network inference (i.e. GPUs). This makes the options to store data in external memory without affecting the throughput of the accelerator rather limited.

On the Zynq-7000 devices, the fastest access to the external memory is through a 64-bit wide AXI4 interface with a maximum burst length of 256. The Alveo U250 has access to four banks of 64 bits each, resulting in an aggregate interface width of 256 bits. The externally stored weight values, once loaded, can then be streamed to the MVTU after performing a data width conversion from the interface width to S_W bits (i.e. the stream width). Naturally, this cannot be done in a single cycle if S_W is greater than the interface width. The most suitable layers for storage in external memory are typically the final fully connected layers for three reasons:

1. In this latency matched dataflow architecture the fully connected layers have the lowest bandwidth requirements, and thus should not stall the pipeline when accessing data externally.
2. Fully connected layers have many connections between neurons, and therefore many weight values to be stored. This increases the likelihood that the memory footprint of the weight values becomes too large to fit in the on-chip memory.
3. The fully connected layers have the lowest amount of data reuse, which implies that the weight values can be loaded and consumed sequentially, as required by the MVTU.

Because of the reasons listed above, the bottom fully connected layer in the ResNet-50 design is streamed from DDR. The memory footprint of the weight values for the evaluated BNN-PYNQ networks is in all cases small enough to fit in the on-chip memory.

3.6. Conclusion

In this chapter the decoupled weights architecture was proposed, in which the memory subsystem is decoupled from the compute units. First, we looked at an overview of the decoupled weights architecture, and established that this architecture allows for more flexible memory resource allocation. The maximum amount of buffers that can be clustered within the same BRAM instance was expressed as a function of the operating frequencies of the memory subsystem and compute units, and we found that the decoupled weights architecture can provide a fine-grained trade-off between BRAM savings and computational throughput. Asynchronous FIFOs were identified as a suitable method to avoid metastability when working with multiple clock domains, and we characterized the associated LUT overhead for the hardware implementation of the proposed architecture. Lastly, guidelines to mitigate throughput penalties when storing data in external memory were proposed.

In the next chapter, two new memory packing algorithms are proposed that are capable of quickly identifying ideal cluster configurations.

4 Memory Packing

In this chapter two new algorithms to solve the memory packing problem are proposed. The concept of memory packing entails algorithmically clustering multiple buffers into BRAM such that the BRAM cost is reduced, and no penalty in terms of computational throughput is incurred. In this thesis we consider memory packing to be a Bin Packing Problem (BPP) with a number of unique hardware implementation related constraints. For this reason, a brief introduction on the classical BPP is provided, after which simple heuristics for solving this problem are discussed. Subsequently, more advanced state-of-the-art algorithms that were proposed in previous work are presented. Next, a new definition of the memory packing problem is provided, after which the imposed hardware implementation constraints are discussed. Finally, genetic algorithms and simulated annealing are introduced and hybridized with a novel heuristic to rapidly solve the memory packing problem.

4.1. The Bin Packing Problem

The bin packing problem is the problem of trying to pack a set of objects of varying sizes into bins with fixed capacities, with as goal to utilize as few as possible bins to pack all the objects. This problem is illustrated in Figure 4.1. Since this problem is NP-hard [13], good heuristics are needed to find acceptable solutions in a reasonable amount of time. The memory packing problem can be considered to be a BPP if we consider the various logical memories corresponding to the weight buffers to be the items or objects, and the physical BRAM instances to be the bins into which we have to fit the objects.

The primary factors that make the memory packing problem different from the classical bin packing problem, is that the bins in this case (the BRAM instances) have a limited amount of ports, and can therefore contain only a limited amount of objects if we want to maintain the same throughput. This constrained version of the BPP is referred to as a BPP with cardinality constraints [48]. There is also the fact that the bins can have variable widths, and therefore variable capacities, depending on the weight buffer partitions that are mapped to the corresponding BRAM instances. Due to these differences, the efficacy of classical heuristics, e.g., first-fit, next-fit, etc., are not sufficient for solving this particular constrained problem [49, 50, 51], so alternative strategies were explored.

4.1.1. Asymptotic Worst Case Ratio

In the literature, bin packing heuristics are usually compared in terms of their theoretical worst-case performance ratio. This performance ratio is defined as follows: let L be any list of items to pack $\{a_0, \dots, a_n\}$, $A(L)$ the amount of bins the heuristic requires to pack the items, and $C^{OPT}(L)$ the amount of bins that are minimally required to pack the items, then the

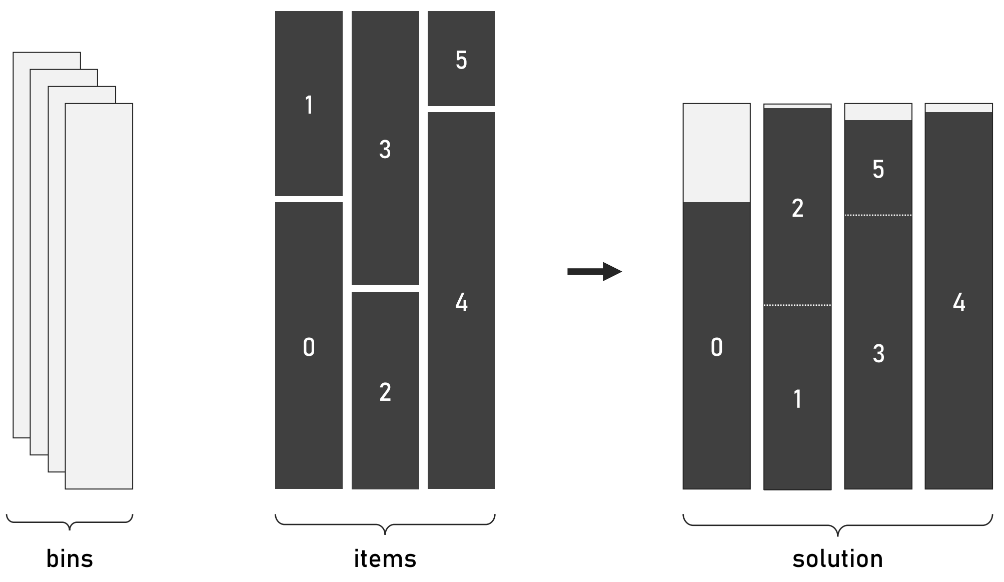


Figure 4.1: Illustration of the Bin Packing Problem

asymptotic worst case performance ratio is given by Equation (4.1) [52]. Thus, R_A^∞ gives a tight upper bound for the packing solutions found by a particular heuristic, compared to the optimal solution for any combination of items.

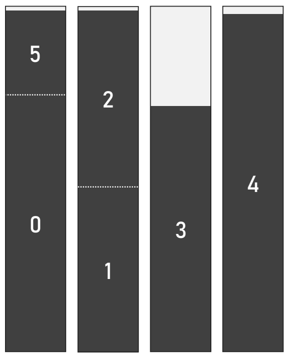
$$R_A^\infty = \limsup_{k\to\infty} \max_L \left\{ \frac{A(L)}{C^{OPT}(L)} \,\middle|\, C^{OPT}(L) = k \right\} \tag{4.1}$$

4.1.2. Deterministic Heuristics

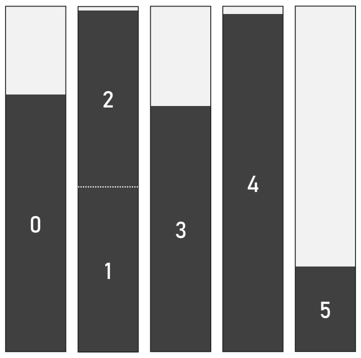
The classical deterministic approximations to the unconstrained BPP are rather simple and fast, but offer sub-optimal results. Figure 4.2 illustrates how various classical heuristics are approaching the BPP. The first-fit heuristic places each item from the list into the first bin that has sufficient capacity left to store it. As can be seen in Table 4.1, this heuristic has a worst-case ratio of $\frac{17}{10}$ and has time complexity $\mathcal{O}(n \log n)$.

Table 4.1: Worst-Case Ratios for a Number of Classical BPP Heuristics [53, 54]

Heuristic	R_A^∞	Time Complexity
Next-Fit	2	$\mathcal{O}(n)$
First-Fit	$\frac{17}{10}$	$\mathcal{O}(n \log n)$
Next-Fit Decreasing	1.691	$\mathcal{O}(n \log n)$
First-Fit Decreasing	$\frac{11}{9}$	$\mathcal{O}(n \log n)$



(a) First-fit



(b) Next-fit

$$L = \{a_0, a_1, a_2, a_3, a_4, a_5\}$$

Figure 4.2: Packing Solutions Obtained by Applying Various BPP Heuristics

Next-fit works similarly to first-fit, but in contrast to first-fit, bins are not revisited. When items do not fit into a bin, that bin is closed, and the item is stored in a new bin. Knowing this, the higher worst-case performance ratio relative to first-fit is unsurprising. Regardless, by not revisiting closed bins, the complexity of this heuristic is reduced.

The last two entries in the table refer to the case where the list of items that are to be packed, are sorted in descending order with regards to size. Intuitively, this is what one would typically do when packing items in boxes as well. And indeed, from Table 4.1 we can deduce that the order of the list has a significant impact on the quality of the obtained packing solutions.

4.2. State-of-the-Art Algorithms

Within the body of literature there also exist approaches that aim to solve the cardinality constrained BPP. Karchmer and Rose apply a a branch and bound algorithm to solve the problem [12]. They pose the constraint that partitions can only be clustered together when they have non-overlapping memory access patterns. This methodology is however not very effective in streaming architectures that typically have many overlapping memory accesses. Furthermore, the authors report a high worst-case time complexity of $\mathcal{O}(m^n)$ for the algorithm; with m being the amount of bins, and n the amount of partitions that are to be packed.

Vasiljevic and Chow propose a simulated annealing algorithm [55]. They essentially solve a two-dimensional bin-packing problem, since they also consider width-wise packing opportunities. Width-wise packing refers to the fact that buffers are placed next to each other (i.e. words from different buffers are concatenated into a long word, which are then stored in a single address of the RAM). The authors report fast convergence on all of the designs that were included in their evaluation. However, the evaluated designs contain a comparatively small amount of buffers. The authors perform an extensive throughput analysis for the various packing configurations, but do not compensate for the incurred penalties. The simulated annealing algorithm as proposed in the paper explores the search space with random movements of buffers between bins; referred to as buffer swaps. This buffer swap method, which is denoted SA-S within this thesis, was replicated based on the information provided in the paper, and used as a baseline in the evaluation of the proposed memory packing algorithms.

4.3. Hardware Constraints

We arrived at the conclusion that the memory packing problem we aim to solve has unique constraints that requires a different approach from the classical bin packing problem. Some of these constraints are hard, while others are referred to as soft constraints. The difference between hard and soft constraints lies in the feasibility of a solution. A solution that violates one or more soft constraints is deemed *undesirable*, while violating a hard constraint immediately renders the packing solution *unfeasible*. In this section the imposed hardware related constraints are defined and classified.

4.3.1. Problem Definition

Let us first reformulate the classical bin packing problem into our specific memory packing problem.

> **Definition 1** *"The memory packing problem is a bin packing problem where the items correspond to buffers, and the bins correspond to the physical memories into which the buffers are to be packed. Bins have variable storage capacities and an upper bound to the amount of discrete partitions they can contain."*

Aside from the cardinality constraints, a fundamental difference with the classical bin packing problem is that we can have bins of variable sizes. Ultimately, the items that are to be packed are the weight buffers, and one or more of their dimensions might exceed the capacity of a BRAM primitive. BRAM also supports aspect ratios that can alter its dimensions, depending on the width of the buffers that are mapped to it. Note that it is instead possible to slice the buffers into smaller segments and keep the bin capacity constant. However, this approach significantly increases the problem size.

Now that we have defined the memory packing problem, we can impose the following constraints:

1. A bin can contain no more partitions than as specified in Equation (3.1) **(hard)**
2. Ceteris paribus, homogeneous bin configurations have precedence **(soft)**
3. Ceteris paribus, shallow bin configurations have precedence **(soft)**

The first constraint is self-explanatory and requires no further elaboration. The second constraint, however, pertains to the concept of intra and inter-layer packing configurations as defined in section 2.2.5. While inter-layer packing configurations offer greater potential for memory reductions, they do have drawbacks associated with them. In particular, the routing becomes more complicated when we pack buffers belonging to layers that are located in completely different sections of the FPGA. This problem is exacerbated on multi-SLR devices, where we should avoid memory accesses across different dies to improve timing closure.

Knowing this, we give precedence to bin configurations that combine partitions from as few as possible different layers.

Similarly, from a hardware perspective, the LUT cost increases proportionally with respect to the amount of addresses spanned by a particular packing configuration. The address decoding becomes more complex, and it is in general a waste of resources to clock the RAM higher, and spend additional LUTs on CDC logic when a packing configuration does not yield any benefits. For this reason, it is necessary to actively discriminate between constructive actions that improve the mapping efficiency, and those that are simply non-deteriorative.

4.4. Metaheuristics

In this section the genetic algorithm and simulated annealing metaheuristics are introduced. These metaheuristics form the basis of the proposed algorithms, and serve as tools to perform a guided random search through the vast amount of potential packing solutions.

4.4.1. Simulated Annealing

Simulated annealing was introduced by Kirkpatrick et. al. as a simple optimization strategy [56]. It is rather similar to general hill climbing algorithms, but its distinguishing feature is that the algorithm occasionally jumps between hills to prevent getting stuck in a local optimum. This jumping of hills is modeled by random thermal motion that forces the algorithm to sometimes perform bad actions. By default, the algorithm accepts an action if it leads to a solution that optimizes a certain cost function. If the action leads to a worse solution, that action might still be accepted with a certain probability $P_A(T)$, as described in Equation (4.2). This probability approaches one for high temperatures, and decays exponentially as the temperature decreases. As a result, the algorithm frequently jumps between hills at the start of the annealing process, and selects a hill to climb in the final phase.

$$P_A(T) = e^{\frac{-\Delta E}{T}} \tag{4.2}$$

4.4.2. Genetic Algorithm

Genetic algorithms, as introduced by Holland, are optimization algorithms that belong to a class of algorithms known as evolutionary algorithms [57]. These algorithms operate on a set of individuals, referred to as a population, that are subjected to genetic manipulations that are inspired by natural selection. Individuals represent candidate solutions with a range of properties that are encoded into chromosomes. The individuals compete against each other in order to procreate and evolve. The quality or fitness of an individual is determined by a cost function that is to be minimized. The strategy is to improve the fitness of the population across multiple generations by applying mutation, crossover (procreation), and selection. This entails having the fittest individuals pass on their genes in the form of creating offspring, and having mutants develop positive traits. In order to avoid getting trapped in a local optimum, genetic algorithms promote genetic diversity by means of random mutation and easing the selective pressure.

4.5. Proposed Algorithms

In this thesis, a new heuristic named *Next-Fit Dynamic* (NFD) is introduced that improves upon previous works [58, 59, 60]. This heuristic enables fast exploration of the search space, aims to deliver hardware optimal packing solutions, and can be embedded in genetic algorithms and simulated annealing approaches. This section covers the details of the algorithms that incorporate this heuristic.

4.5.1. The Next-Fit Dynamic Heuristic

Next-fit dynamic is based on the simplest bin packing heuristic next-fit (i.e. it has time complexity $\mathcal{O}(n)$), but explicitly takes the cardinality constraints and variable bin size into account. As can be seen in Algorithm 1, out of a particular packing solution, those bins that map poorly to BRAM are marked for recombination. The mapping efficiency to BRAM is calculated as defined in Equation (2.4). If the mapping efficiency of a bin is below the given threshold, the bin is marked for repackaging. Repackaging implies that the bin is emptied, and the buffers that it contained are added to a list of items that are to be packed.

Subsequently, next-fit dynamic attempts to optimize the placement of the buffers belonging to these bins. As previously mentioned, the order of the items in the list strongly impacts the efficacy of bin packing heuristics like next-fit and first-fit. As such, the order of the list is shuffled before packaging, which allows for the exploration of new packing configurations. Since next-fit dynamic is based on next-fit, the packing list is traversed linearly. This means that the first item in the list is placed into a new bin. If the next item does not 'fit' in the already opened bin, that bin is closed and the item is placed in a new bin.

Dynamic Bin Capacity

Since we have bins of variable capacity, we need to redefine the fullness of bins. In next-fit dynamic the fullness of a bin is determined by three factors:

- the cardinality constraints (i.e. max bin length)
- the current width of the bin
- the current height of the bin

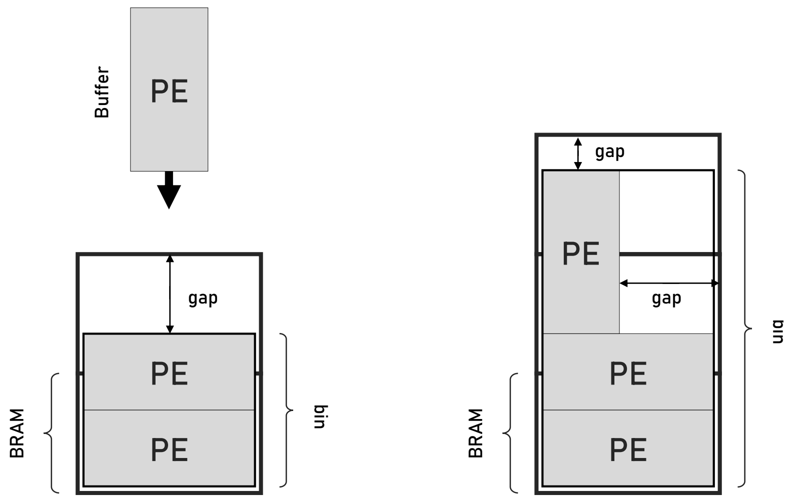
If the opened bin already contains an amount of buffers equal to the maximum bin length (i.e. H_B as defined in Equation (3.1)), the bin is closed, and the buffer under consideration is placed in a new bin.

The two remaining points are related to the mapping efficiencies of the bins. As can be seen in Figure 4.3b, the mapping efficiency of a bin is determined by the empty space that would be left if the bin were to be mapped to BRAM. Since the width of a bin is equal to the maximum width of all buffers that are packed inside it, the bin's mapping efficiency typically diminishes if we group buffers with different widths. For this reason, the bin is closed if the width of the buffer that is to be packed, is different from the width of the opened bin.

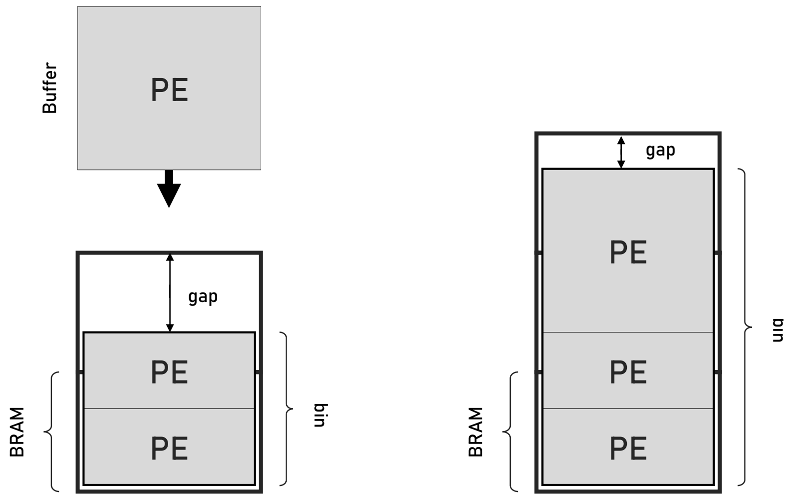
The height of a bin is equal to the sum of the heights of all the buffers that are packed inside. As shown in Figure 4.3b, a buffer can be added to a bin if it improves the mapping efficiency of that bin; which implies that the gap as indicated in the figure is minimized. More specifically, if the sum of the bin height and the height of the buffer under consideration better approaches an integer multiple of the depth of the memory resource (i.e. 1024 addresses for

BRAM) than the current bin height, the mapping efficiency is improved, and the buffer can be placed in the bin.

To avoid getting stuck in local optima there are small admission probabilities $P_{adm,w}$ and $P_{adm,h}$ that cause the heuristic to occasionally permit the exploration of solutions that do not immediately improve the mapping efficiency. This strategy enables fast exploration of large search spaces, and gives more control over bin compositions; i.e. not unnecessarily packing buffers if it won't lead to BRAM savings. Moreover, this additional control also enables restrictions like intra-layer packing to potentially obtain less routing congestion on the FPGA.



(a) Buffer Width



(b) Buffer Height

Figure 4.3: Bin Placement Checks

Algorithm 1: Next-Fit Dynamic (NFD) Heuristic

```
Input: list of packed bins
Output: list of repackaged bins
sublist = calculateMapEfficiency(list, threshold);
shuffle(sublist);
for buffer in sublist do
    if bin height == 0 then
        bin ← buffer;
        update(bin width, bin height);
    else
        calculate(new bin height);
        gap = calculateGap(BRAM height, bin height);
        new gap = calculateGap(BRAM height, new bin height);
        if length bin < max bin length AND
        ((new gap < gap OR rnd() < $P_{adm,h}$) AND
        (bin width == buffer width OR rnd() < $P_{adm,w}$)) then
            bin ← buffer;
            update(bin width, bin height);
        else
            list ← bin;
            reset(bin, bin width, bin height);
            bin ← buffer;
if length bin > 0 then
    list ← bin;
```

4.5.2. Simulated Annealing Approach

In this section, the simulated annealing approach that incorporates the next-fit dynamic heuristic (denoted SA-NFD) is presented. The general flow of the simulated annealing approach is as described in Algorithm 2. First a random, yet feasible solution is generated that adheres to the aforementioned constraints. Then, the BRAM cost for this solution is calculated according to the developed model for BRAM resource utilization, which is listed in Algorithm 3. This model takes the different aspect ratios and SDP modes into account.

Finally, the annealing optimization process commences as described before. Next-fit dynamic is used to “perturb” the candidate solution. If the perturbation was beneficial, the candidate solution is immediately accepted. Otherwise, the acceptance probability P_A is calculated according to the current temperature, and the acceptance of the bad move might be reconsidered.

The efficacy of this algorithm is largely determined by selecting an appropriate annealing schedule for a given problem. If the temperature is reduced too fast, the algorithm converges quickly, but might get stuck in a local minimum. Conversely, reducing the temperature too slowly also causes the algorithm to converge slowly, and causes oscillations because of the high probability of accepting good and bad moves alike.

Algorithm 2: Simulated Annealing

```
   Input: list of partitions, max bin height
   Output: BRAM cost, list of packed bins
 1 initilize(solution, T);
 2 cost = costFunction(solution);
 3 while not converged do
 4     T = calculateTemperature();
 5     candidate = perturb(solution);
 6     new cost = costFunction(candidate);
 7     $P_A$ = probability(cost, new cost, T);
 8     if new cost < cost OR rnd() < $P_A$ then
 9         solution = candidate;
```

4.5.3. Genetic Algorithm Approach

Many of the approaches in the literature use genetic algorithms to solve the unconstrained bin packing problem [58, 60]. However, the heuristics that are mentioned in the literature are not directly compatible with the imposed hardware related constraints. In this section the details of the proposed genetic algorithm are be provided.

The Naïve Approach

A straightforward approach to solve the memory packing problem with genetic algorithms is with the "item per gene" representation. As depicted in Figure 4.4, the genes represent the buffer partitions, and the value of each gene decides in which bin the corresponding partition is to be packed. The search space can be explored by randomly changing the value of some of the partitions, which effectively places the partitions into different bins, or by means of recombination where genes from two parents are combined.

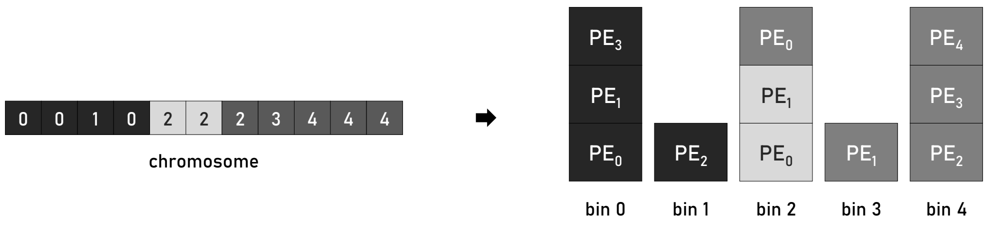


Figure 4.4: Item Per Gene Chromosome Encoding

As Falkenauer demonstrated, this chromosome encoding scheme was found to be highly inefficient [59]. The transmission of genes from parents with high fitness to their offspring is mostly meaningless, since the quality of a gene is determined by the ensemble in which it is grouped; this context is lost when individually transmitted. Furthermore, in the case of cardinality constrained bin packing, this method had a high probability of generating unfeasible solutions.

Algorithm 3: BRAM Cost Model

```
Input: list of packed bins
Output: BRAM cost
for bin in list do
    if bin width == 1 then
        // "narrow" 1-bit x 16384 adr.  aspect ratio
        new BRAM height = 16 * BRAM height;
        new BRAM width = BRAM width / 16;
    else if bin width == 2 then
        // "narrow" 2-bit x 8192 adr.  aspect ratio
        new BRAM height = 8 * BRAM height;
        new BRAM width = BRAM width / 8;
    else if bin width <= 4 then
        // "narrow" 4-bit x 4096 adr.  aspect ratio
        new BRAM height = 4 * BRAM height;
        new BRAM width = BRAM width / 4;
    else if bin width <= 9 then
        // "narrow" 9-bit x 2048 adr.  aspect ratio
        new BRAM height = 2 * BRAM height;
        new BRAM width = BRAM width / 2;
    else
        // "regular" 18-bit x 1024 adr.  aspect ratio
        new BRAM height = BRAM height;
        new BRAM width = BRAM width;
    if length bin == 1 AND bin height <= 512 then
        // "wide" 36-bit x 512 adr.  aspect ratio (SDP)
        new BRAM height = BRAM height / 2;
        new BRAM width = 2 * BRAM width;
    BRAM cost += ceil(bin height / new BRAM height) * ceil(bin width / new BRAM width);
```

Grouping Genetic Algorithm

The chromosome encoding scheme that is employed in this work is referred to as the so called group based, or simply "bin per gene" chromosome representation as introduced by Falkenauer and Delchambre [58]. This time, as illustrated in Figure 4.5, a gene represents a bin that contains as value a list of buffers that are to be packed together. The bin per gene representation enables targeted manipulation of genes to improve the quality of a solution.

The genetic operators used in the algorithm are: mutation, and tournament selection where the best solution is picked out of a randomly selected batch of solutions. The selection process is repeated until the new generation has the same population count as the preceding generation. The pseudocode for the genetic algorithm is listed in Algorithm 4.

Mutation: The mutation operator is the driving factor in the process of exploring the search space. Two exploration methods are employed in this thesis. For what is from here on referred to as GA-S, the buffer swap method that was introduced in [55] is used for mutation. GA-S serves as an ablation test in the evaluation of the proposed algorithms, where we attempt to determine the impact of the next-fit dynamic heuristic. The proposed algorithm, GA-NFD, incorporates the next-fit dynamic heuristic as mutation operator. Here a number of genes are

Algorithm 4: Genetic Algorithm

```
   Input: list of partitions, max bin height
   Output: BRAM cost, list of packed bins
1  initialize(population);
2  while not converged do
3      for individual in population do
4          if rnd() $< P_{mut}$ then
5              mutate(individual);
6          calculateFitness(individual);
7      while new population count < population count do
8          new individual = tourSelect(population, tour size);
9          new population $\leftarrow$ new individual
10     population = new population;
```

selected, after which the contents of the corresponding bins are emptied, and marked for recombination with the next-fit dynamic heuristic.

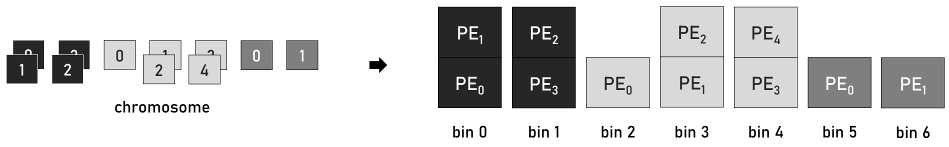


Figure 4.5: Bin Per Gene Chromosome Encoding

Selection: In order to distinguish good solutions from bad solutions, we need to select based on a quantifiable metric. After exploration through mutation, the best solutions of a particular generation are selected through tournament selection. As can be seen in Algorithm 4, each tournament has a single victor that makes it into the next generation.

The factor that determines which individual (solution) wins the tournament, is the fitness of that particular individual. In this work a multi-objective cost function is employed, where a weighted sum between BRAM cost and layer count per bin is computed. Packing solutions that result in the lowest BRAM cost, and do so with bin configurations that contain buffers from as few as possible different layers, are more likely to make it into the next generation. As time progresses, only the solutions that best meet these criteria remain.

Hyperparameters

Similar to the simulated annealing algorithm, the performance of the genetic algorithm depends on the chosen hyperparameter values. The population size effectively controls how many candidate solutions are still being considered at any given generation. Incrementing this value generally leads to higher quality results at the cost of slower convergence.

The tournament size (N_t) determines how many candidate solutions are selected per tournament to directly compete against each other. If this value is low, the selection process is essentially random. For high values, the solutions with the poorest quality have low chances of survival. While the latter may lead to faster convergence, it reduces the genetic diversity, and increases the likelihood of getting stuck in a local optimum.

Finally, the probabilities P_{mut}, $P_{adm,w}$ and $P_{adm,h}$ determine how the search space is explored. Increasing P_{mut} causes the mutation operator to be invoked for a larger portion of the population, which leads to a greater amount of explorations. Increasing this value too high, however, eventually leads to the deterioration of good solutions. Likewise, increasing $P_{adm,w}$ and $P_{adm,h}$ causes the algorithm to explore more packing configurations, even if these do not improve the mapping efficiency.

4.6. Conclusion

In this chapter two new algorithms to rapidly solve the memory packing problem were proposed. We introduced the bin packing problem, and deduced that the memory packing problem is an instance of the bin packing problem with custom constraints. We then investigated several state of the art algorithms, of which the most promising one was based on a simulated annealing approach. Subsequently, we formally defined the memory packing problem, and composed a list of hardware implementation related constraints. At the end, a new heuristic was proposed, and was embedded in simulated annealing and genetic algorithm metaheuristics.

In the next chapter, the algorithms that were introduced in this chapter, and the architecture that was proposed in Chapter 3 are evaluated.

5 Evaluation

The methodology as described in the previous chapters was applied and validated on a range of accelerator designs that were built upon the FINN framework. What follows in this chapter is an analysis of the merit of the decoupled architecture and memory packing algorithms for a range of real-world use cases. First, the developed algorithms are compared in terms of runtime and obtained quality of results. Next, a selection of the solutions are implemented in hardware, and evaluated in terms of BRAM reductions and hardware costs. Finally, we study how the computational throughput and power consumption are impacted by the attained frequency of the memory subsystem in the decoupled weights architecture. In this chapter, the BRAM count is expressed in terms of RAMB18 primitives, unless otherwise specified.

5.1. Memory Packing Algorithm

The memory packing algorithms are evaluated on several CNN-based object detection and classification accelerators that were selected from previous work, and are listed in Table 5.1. The table indicates the source publication for each accelerator, and also the shapes and number of weight buffers of each accelerator, which serve as input for the packing algorithms.

Small Image Classifiers

The CNV CNNs belong to the BNN-PYNQ suite of object classification accelerators. As mentioned in Chapter 2, the BNN-PYNQ suite consists of FINN-style FPGA accelerators, and target embedded FPGA devices such as the Zynq-7020. CNV-W1A1 utilizes binary quantization while CNV-W2A2 utilizes ternary (2-bit) quantization [61]. Both CNNs are trained on the CIFAR-10 [45] dataset, and are able to distinguish between ten classes of common objects (e.g. birds, cars, dogs, etc.).

Mid-Size Image Classifiers

DoReFaNet and ReBNet are medium-size CNNs trained for object classification on the 1000-class ImageNet dataset. These CNNs are both quantized versions of AlexNet [1], a popular image classification CNN topology. The accelerators use binary weights, and consist of five convolutional layers and three fully-connected layers. However, DoReFaNet and ReBNet differ in the folding factors utilized for their implementation, and therefore in the shapes of their weight buffers, and as such are treated separately in the evaluation. DoReFaNet was first binarized in [62] and implemented in FPGA in [10]. ReBNet was described and implemented in FPGA in [63] where it is denoted 'Arch3'.

Large Image Classifiers
As mentioned in Chapter 2, ResNet-50 [2] is a high-accuracy classification CNN designed for high-accuracy image classification on the ImageNet dataset. The accelerator has been implemented according to the design principles of FINN accelerators [10], uses binarized weights, and targets the largest commercially available Xilinx FPGA, the Alveo U250. Larger ResNet variants are also included in the evaluation — ResNet-101 and ResNet-152 – which are approximately two and three times deeper than ResNet-50, respectively, but share the overall structure.

Object Detectors
Tincy-YOLO was first published in [11] and is a binarized-weight variant of YOLO [64], a popular object detection CNN. The design consists exclusively out of convolutional layers, six of which utilize binary weights, while two utilize 8-bit weights.

5.1.1. Test Methodology

Packing Algorithm Comparison: First, the Genetic Algorithm (GA) and Simulated Annealing (SA) packing algorithms are compared with and without Next-Fit Dynamic, in terms of wall-clock time to convergence and quality of results, for each of the accelerators under evaluation. For all algorithms a cardinality constraint of a maximum of four weight buffers per physical BRAM is imposed. The reported time to convergence is defined as the amount of time it takes each algorithm to attain a packing result that is within 1% of the discovered minimum. The reported BRAM count is defined as the amount of RAMB18 primitives that are required to implement the weight buffers. For each convergence experiment, ten different initial random seeds are evaluated.

Mapping Efficiency Increase: The mapping efficiency of the weight buffers to BRAM is calculated for each of the CNN accelerators (according to Equation (2.4)), targeting a maximum bin height of four, and utilizing both inter-layer (unconstrained) and intra-layer packing strategies. In this set of experiments, the GA with the NFD heuristic is used for optimization.

5.1.2. Experiment Setup

The GA and SA packing algorithms are implemented in Python code utilizing the DEAP (Distributed Evolutionary Algorithms in Python) evolutionary computation library (version 1.3.0) [65]. The packing algorithms are executed in single-thread mode on a server equipped with Intel Xeon Silver 4110 CPUs, 128 GB of system RAM, and SSD storage. The run-time is measured using Python's time package.

5.1.3. Convergence Time

In this section the performance of the proposed heuristic is evaluated. As baseline, the SA and GA that incorporate Next-Fit Dynamic (SA-NFD and GA-NFD), are compared against SA and GA implementations that use the buffer swap methodology (SA-S and GA-S). Both versions of the GA and SA were applied to solve the memory packing problem for the accelerator designs as listed in Table 5.1. The corresponding hyperparameter settings can be found in Table 5.2.

The runtime comparison results are displayed in Table 5.3 for all accelerator designs, with the best results highlighted in bold where a clear winner could be distinguished. It has to be mentioned that for the GA implementations (i.e. GA-S and GA-NFD) the minimum BRAM

Table 5.1: Baseline Dataflow Accelerators

(a) Small Image Classifiers

Accelerator:	**CNV-W1A1 [10]**	**CNV-W2A2 [10]**
Memory Shapes	$16 \times (32, 144, 1)$	$8 \times (16, 576, 2)$
$N_{PE} \times (N_{SIMD}, M_D, W_P)$	$16 \times (32, 288, 1)$	$8 \times (16, 1152, 2)$
	$4 \times (32, 2304, 1)$	$4 \times (1, 8192, 2)$
	$4 \times (1, 8192, 1)$	$4 \times (8, 9216, 2)$
	$1 \times (32, 18432, 1)$	$3 \times (2, 65536, 2)$
	$1 \times (4, 32768, 1)$	$1 \times (8, 73728, 2)$
	$1 \times (8, 32768, 1)$	
Total Buffers:	43	28

(b) Object Detectors

Accelerator:	**Tincy-YOLO [11]**
Memory Shapes	$16 \times (32, 144, 1)$
$N_{PE} \times (N_{SIMD}, M_D, W_P)$	$25 \times (8, 320, 1)$
	$16 \times (32, 144, 1)$
	$80 \times (32, 2304, 1)$
Total Buffers:	137

(c) Mid-Size Image Classifiers

Accelerator:	**DoReFaNet [11]**	**ReBNet [63]**
Memory Shapes	$136 \times (45, 72, 1)$	$64 \times (54, 256, 1)$
$N_{PE} \times (N_{SIMD}, M_D, W_P)$	$64 \times (34, 108, 1)$	$64 \times (25, 384, 1)$
	$32 \times (64, 108, 1)$	$64 \times (36, 384, 1)$
	$68 \times (3, 144, 1)$	$64 \times (32, 576, 1)$
	$8 \times (8, 64000, 1)$	$128 \times (64, 1152, 1)$
	$4 \times (64, 65536, 1)$	$40 \times (50, 2048, 1)$
	$8 \times (64, 73728, 1)$	$128 \times (64, 2048, 1)$
Total Buffers:	320	552

(d) Large Image Classifiers

Accelerator:	**RN50-W1A2**	**RN101-W1A2**	**RN152-W1A2**
Memory Shapes	$368 \times (32, 256, 1)$	$1456 \times (32, 256, 1)$	$2288 \times (32, 256, 1)$
$N_{PE} \times (N_{SIMD}, M_D, W_P)$	$32 \times (64, 256, 1)$	$32 \times (64, 256, 1)$	$32 \times (64, 256, 1)$
	$192 \times (64, 288, 1)$	$736 \times (64, 288, 1)$	$1152 \times (64, 288, 1)$
	$176 \times (32, 1024, 1)$	$176 \times (32, 1024, 1)$	$176 \times (32, 1024, 1)$
	$32 \times (64, 1024, 1)$	$32 \times (64, 1024, 1)$	$32 \times (64, 1024, 1)$
	$96 \times (64, 1152, 1)$	$96 \times (64, 1152, 1)$	$96 \times (64, 1152, 1)$
Total Buffers:	896	2528	3776

count of all candidate solutions in a particular generation is tracked.

All the algorithms are capable of quickly solving the memory packing problem for the smaller CNV accelerator designs. However, as the problem size is increased (e.g. Tincy-YOLO, DoReFaNet and ResNets) the NFD versions of the algorithms are capable of solving the

Table 5.2: SA and GA Hyperparameters

	GA					SA	
Accelerator	N_p	N_t	$P_{adm,w}$	$P_{adm,h}$	P_{mut}	T_0	R_c
CNV-W1A1	50	5	0	0.1	0.3	30	1
CNV-W2A2	50	5	0	0.1	0.3	30	2
Tincy-YOLO	75	5	0	0.2	0.4	30	1
DoReFaNet	50	5	0.1	0.3	0.4	30	1
ReBNet Arch3	75	5	1	0.2	0.4	30	1
RN50-W1A2	75	5	0	0.1	0.4	40	0.004
RN101-W1A2	75	5	0	0.1	0.4	40	0.004
RN152-W1A2	75	5	0	0.1	0.4	40	0.004

memory packing problem considerably faster, and with higher quality of results. For the ResNets in particular, the NFD algorithms are capable of finding solutions that require up to 8% less BRAM to implement, and reduce the required runtime by a factor of more than 200× compared to SA-S. GA-S provides poor quality of results especially for larger networks and is also slower than all other algorithms. In general, GA-NFD achieves the best quality of results, while SA-NFD is the fastest.

The outlier here is the ReBNet Arch3 accelerator design. This design contains weight buffers with a large variety in widths (SIMD lanes), which causes difficulty for NFD as it is forced to pack together buffers with misaligning widths. In order to compete on the metrics as presented in Table 5.3 (i.e. BRAM cost and runtime), the hardware constraints had to be relaxed significantly, as is reflected by the high admission probabilities — $P_{adm,w}$ and $P_{adm,h}$ — as listed in Table 5.2. Nevertheless, the NFD-based algorithms (especially SA-NFD) arrive at a packing solution significantly faster than buffer swap based GA and SA.

To emphasize the differences in quality of results, aside from potentially greater BRAM reductions, the NFD algorithms also provide packing solutions with more ideal bin configurations from a hardware design perspective. The reason for this is that the heuristic typically only packs buffers into bins when it improves the mapping efficiency of these bins. As a consequence, the NFD algorithms typically provide packing solutions that contain, on average, bins of lower height, which results in a lower LUT overhead.

5.1.4. Mapping Efficiency

In this section, the impact of the two different memory packing strategies (intra- and inter-layer packing) on the mapping efficiency of weight buffers to BRAM is evaluated. The results are displayed in Table 5.4, and were obtained by utilizing the GA-NFD algorithm, which achieved the best overall packing performance in terms of quality of results in Table 5.3. As mentioned in Chapter 2, the term “intra” refers to the fact that exclusively buffers corresponding to the same neural network layer are packed together, while in inter-layer packing configurations no such constraints are imposed. The results are presented in terms of the amount of required BRAM resources to implement the weight buffers, the resulting mapping efficiency as dictated by Equation (2.4), and the reduction in memory footprint Δ_{BRAM}.

While the smaller accelerators benefit from the GA-NFD memory packing, the largest impact is observed for the ResNet accelerators, which are configured for high throughput,

Table 5.3: Memory Packing Comparison for the different Heuristics

(a) Buffer Swap

	Buffer Swap			
Accelerator	t_{SA-S} (s)	t_{GA-S} (s)	N_{BRAM}^{SA-S}	N_{BRAM}^{GA-S}
CNV-W1A1	0.1	0.2	96	96
CNV-W2A2	0.1	0.1	188	190
Tincy-YOLO	1.8	1.7	420	428
DoReFaNet	1.0	1.6	3823	3826
ReBNet Arch3	40.1	57.5	**2301**	2313
RN50-W1A2	239	290	1404	1472
RN101-W1A2	615	935	2775	3055
RN152-W1A2	1024	1354	3864	4422

(b) Next-Fit Dynamic

	Next-Fit Dynamic			
Accelerator	t_{SA-NFD} (s)	t_{GA-NFD} (s)	N_{BRAM}^{SA-NFD}	N_{BRAM}^{GA-NFD}
CNV-W1A1	0.1	0.1	97	96
CNV-W2A2	0.1	0.1	190	188
Tincy-YOLO	**0.1**	0.2	430	420
DoReFaNet	**0.1**	0.2	3849	**3794**
ReBNet Arch3	**2.2**	28.9	2483	2352
RN50-W1A2	**0.8**	1.7	**1368**	1374
RN101-W1A2	**0.9**	3.3	**2616**	**2616**
RN152-W1A2	**1.5**	4.9	3586	**3584**

and therefore have a low initial memory mapping efficiency. It is worth mentioning that the added constraint of intra-layer mapping does not significantly degrade the achievable efficiency; in most cases the intra-layer efficiency is within 5% of the inter-layer efficiency.

5.2. Hardware Implementations

A selection of the accelerators as listed in Table 5.1 have been implemented in hardware based on the decoupled weights architecture as laid out in Chapter 3. The specifications and implementation details of the original accelerator designs based on the integrated weights architecture can be found in Section 2.2.6. In this section, the synthesis results for the decoupled weights accelerator designs are presented, and the impact of the operating frequency of the memory subsystem on the power consumption and computational throughput of the accelerator is examined.

5.2.1. Timing Closure

The efficacy of the memory packing methodology strongly depends on the available headroom to increase the memory subsystem's operating frequency F_m, relative to that of the compute units F_c. The obtained results show that the ability to increase the frequency strongly depends on the target platform, and how constrained the device is for resources. The implementation results for attempting to increase F_m for several accelerators are presented in Table 5.5.

Table 5.4: Mapping Efficiency Increase (GA-NFD)

Accelerator	BRAM	Efficiency	Δ_{BRAM}
CNV-W1A1	120	69.3%	
CNV-W1A1-Intra	100	82.3%	1.20×
CNV-W1A1-Inter	96	86.6%	1.25×
CNV-W2A2	208	79.9%	
CNV-W2A2-Intra	192	86.6%	1.08×
CNV-W2A2-Inter	188	88.4%	1.11×
Tincy-YOLO	578	63.6%	
Tincy-YOLO-Intra	456	80.7%	1.27×
Tincy-YOLO-Inter	420	87.6%	1.38×
DoReFaNet	4116	78.8%	
DoReFaNet-Intra	3797	85.4%	1.08×
DoReFaNet-Inter	3794	85.5%	1.08×
ReBNet	2880	64.1%	
ReBNet-Intra	2363	78.1%	1.22×
ReBNet-Inter	2352	78.4%	1.22×
RN50-W1A2	2064	57.9%	
RN50-W1A2-Intra	1440	82.9%	1.43×
RN50-W1A2-Inter	1374	86.9%	1.50×
RN101-W1A2	4240	52.4%	
RN101-W1A2-Intra	2748	80.9%	1.54×
RN101-W1A2-Inter	2616	84.9%	1.62×
RN152-W1A2	5904	50.9%	
RN152-W1A2-Intra	3758	80.0%	1.57×
RN152-W1A2-Inter	3584	83.9%	1.65×

To find the maximum attainable clock frequencies, all accelerator designs are converted to the decoupled weights architecture, and implemented in Vivado (version 2019.1). F_c is kept constant compared to the original implementation, while for F_m an initial estimate of 2 $\times F_c$ is chosen; initially aiming for a ratio of 2× between the two clock domains. If the design fails in timing closure, the ratio between the clock domains is decreased by 0.5. If the design meets timing closure, and has not failed to meet timing closure before, the experiment is repeated with a higher ratio. The highest attainable clock frequencies are recorded for the highest ratio at which the design met timing closure.

What can be observed from the results as listed in Table 5.5 is that there is a larger headroom on designs that are relatively constrained for resources. The Z-7020 is rather constrained for LUTs, and struggles to meet timing closure past 100 MHz for the CNV-W2A2 design. By contrast, it is relatively easy to meet timing closure with F_m at 200 MHz, yielding a ratio of 2×.

Mapping the same designs to the ZU3EG, that has significantly more resources available on-chip, demonstrates that it is difficult to maintain a ratio of 2× for devices that are not constrained for resources. The compute units are almost capable of reaching 300 MHz, but unsurprisingly, meeting timing closure at 600 MHz for the memory subsystem proves to be impossible.

Determining the maximum clock frequencies for ResNet-50 is not as straightforward.

Table 5.5: Timing Closure for a Number of Accelerator - FPGA Configurations

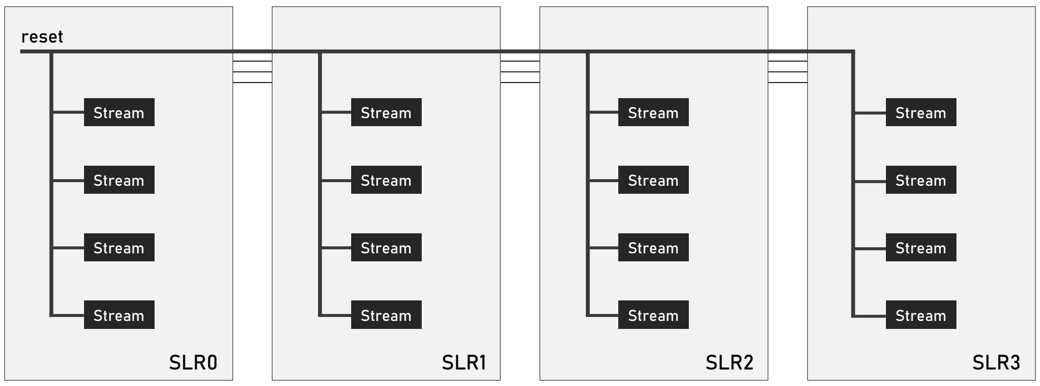
Device	Accelerator	F_c (MHz)	F_m (MHz)	F_m / F_c
Z-7020	CNV-W1A1	100	250	2.5×
Z-7020	CNV-W2A2	100	200	2×
ZU3EG	CNV-W1A1	299	449	1.5×
ZU3EG	CNV-W2A2	299	449	1.5×
VU13P	RN50-W1A2	180	360	2×

While the compute units were rather constrained at 205 MHz, the overhead of the stream generators and FIFOs introduced additional routing congestion, which negatively impacted the maximum attainable frequency. Moreover, meeting timing closure at 360 MHz for the memory subsystem on this multi-SLR device was not trivial. Initially, the operating frequency of the memory subsystem was bounded at 225 MHz. The critical path was, as depicted in Figure 5.1a, from the reset pin to the stream generators on remote SLRs. The reset was routed through all SLRs, and crossed multiple Laguna sites in the process, and thus introduced a considerable amount of delay. Additionally, the reset is a signal with a high fan-out, which significantly increases the net delay. As depicted in Figure 5.1b, to reduce the inter-SLR path delay, the reset is pipelined on each SLR. To reduce the skew within each SLR, the reset is distributed evenly by routing it through clock buffers (BUFGs). The ResNet-50 design demonstrates that in order to fully leverage the potential of this methodology on multi-SLR devices, more extensive design considerations might be required.

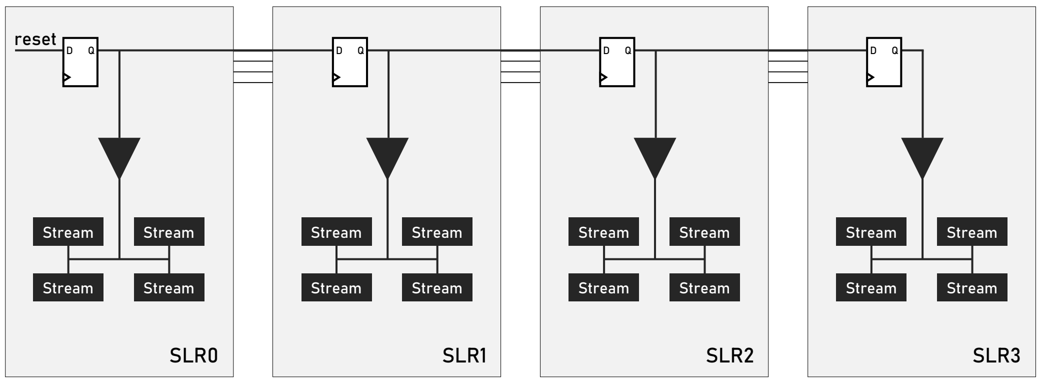
5.2.2. Synthesis Results

After deriving the cardinality constraints from the maximum synthesis frequencies for the accelerator designs on each targeted device, the memory packing algorithm was used to generate efficient packing solutions. All of the identified bin configurations are translated to stream generators with asynchronous FIFOs and corresponding CDC logic. Both inter- and intra-layer configurations were explored, and are compared on attained accelerator throughput in terms of FPS, LUT utilization, and BRAM utilization for implementing the weight buffers in Table 5.6.

A point of attention is that the BRAM utilization and efficiency data in Table 5.6 might deviate from the data as displayed in Table 5.4. These deviations are caused by a difference in resource binding; some weight buffers that were assumed to be mapped to LUTRAM are instead mapped to BRAM in the hardware implementation. Furthermore, for the small CNV networks, the weight buffers corresponding to layers that do not require the frequency overhead (i.e. none of the buffers of a particular layer are grouped in bins with $H_B > 2$) are implemented as integrated weights rather than streamed weights to save LUTs. However, this optimization was not applied to ResNet-50 due to the complexity of this design. Instead, all of the weight buffers are implemented in a streaming fashion. Finally, to improve timing closure for the ResNet-50 design, the memory packing problem was solved individually for each SLR. This implies that only weight buffers belonging to residual blocks that are placed on the same SLR, as indicated in the floorplan displayed in Figure 2.16, can be grouped together.



(a) High Reset Skew and Net Delay Routing



(b) Pipelined Minimal Reset Skew Routing

Figure 5.1: VU13P: Reset Routing

As can be seen in Table 5.6, the impact of the memory packing methodology varies greatly across the different accelerator designs. For CNV-W1A1 a reduction in memory footprint of 1.25× is observed at a 17% increase in LUTs in inter-layer configuration. The intra-layer configuration yields slightly less savings and introduces a similar LUT overhead. Regardless, these BRAM reductions enable the possibility to port the CNV-W1A1 accelerator design to the Zynq-7012S, at the same computational throughput. To emphasize the importance of this result, the fact that the original design can be ported to smaller devices implies that it is possible to significantly reduce the financial cost of, for example, edge computing solutions.

The CNV-W2A2 design does not benefit significantly from the memory packing strategy. The amount of parallelism that is exploited in this design is notably lower compared to the binary variant (CNV-W1A1), which was necessary to make the design fit on the Zynq-7020. Hence, the weight buffers already mapped reasonably well to the BRAM resources. Consequently, the BRAM reduction is insufficient to fit the design on smaller devices.

As expected, the ResNet-50 design benefits the most from the memory packing methodology in terms of BRAM savings. Since this accelerator design exploits a substantial amount of parallelism, many of the BRAM resources are underutilized in the original implementation. After applying the intra- and inter-layer packing strategy, the required amount of BRAM

Table 5.6: Packing Results

Accelerator	FPS	kLUTs	RAMB18s	F_c (MHz)	F_m (MHz)	Efficiency
CNV-W1A1	632	25.7	120	100	100	69.3%
CNV-W1A1-Intra	632	29.7	100	100	200	82.3%
CNV-W1A1-Inter	632	30.0	96	100	200	86.6%
CNV-W2A2	206	32.9	208	100	100	79.9%
CNV-W2A2-Intra	206	34.9	192	100	200	86.6%
CNV-W2A2-Inter	206	34.8	188	100	200	88.4%
RN50-W1A2	2015	787	2320	205	205	52.9%
RN50-W1A2-Intra	1468	968	1632	180	360	75.3%
RN50-W1A2-Inter	1468	968	1588	180	360	77.3%

resources to implement the weight buffers was reduced by respectively 1.45× and 1.46×, at an increase in LUT overhead of 23%. Similar to the CNV accelerators, there is no difference in maximum attained frequency between the intra- and inter-layer packing strategies. Remarkably, the throughput of the accelerator was reduced by 1.37×. While lower throughput was expected, due to the added LUT overhead and the impact of this overhead on the operating frequency of the compute units, the reduction in clock frequency does not account for the 27% loss in computational throughput. This problem could potentially be caused by an insufficient FIFO depth (which was chosen to be 32 deep), but further investigation is required to ascertain the exact cause for this discrepancy.

5.2.3. Memory Subsystem Frequency Scaling

One of the aforementioned merits of the decoupled weights architecture is the fact that the design trade-off between the computational throughput and BRAM utilization can be made with a significantly higher degree of granularity. In this section, the impact of the maximum attainable frequency of the memory subsystem (F_m) on the computational throughput, and power consumption of the accelerator is investigated.

For this experiment, the inter-layer ResNet-50 accelerator design as listed in Table 5.6 is utilized. To reiterate, this design targets the Alveo U250, is implemented based on the decoupled weights architecture, and is packed with a cardinality constraint of four (i.e. we require $F_m/F_c = 2\times$ to prevent throughput penalties). The synthetic benchmark from the Xilinx ResNet50-PYNQ repository [66] was used to measure the power and throughput of the accelerator. In this benchmark, the accelerator repeatedly classifies 1000 images for 100 times in an unbroken sequence (i.e. without data transfers between the host and the accelerator). The batch size is deliberately chosen to be large in order to measure the maximum attainable throughput (framerate), and the associated power draw more closely. To obtain the power readings, the supply voltage rail of the FPGA is monitored during inference through the Power Management Bus (PMBus), and the mean value over the entire inference process is reported as the final (average) FPGA power draw. Lastly, F_m is increased from 180 MHz to 360 MHz in steps of 5 MHz, while F_c is kept constant at 180 MHz.

The measurements displayed in Figure 5.2 clearly demonstrate that the throughput increases gradually as the operating frequency of the memory subsystem is incremented. Equally, a gradual rise in power consumption is observed; the power consumption increases by 25% (10W) between the two endpoints. These results indicate that interesting trade-offs could be made with respect to throughput and BRAM utilization. If some degradation in computational throughput is tolerated, several memory packing strategies can still be applied, even if it is not possible to meet timing closure at the desired frequency.

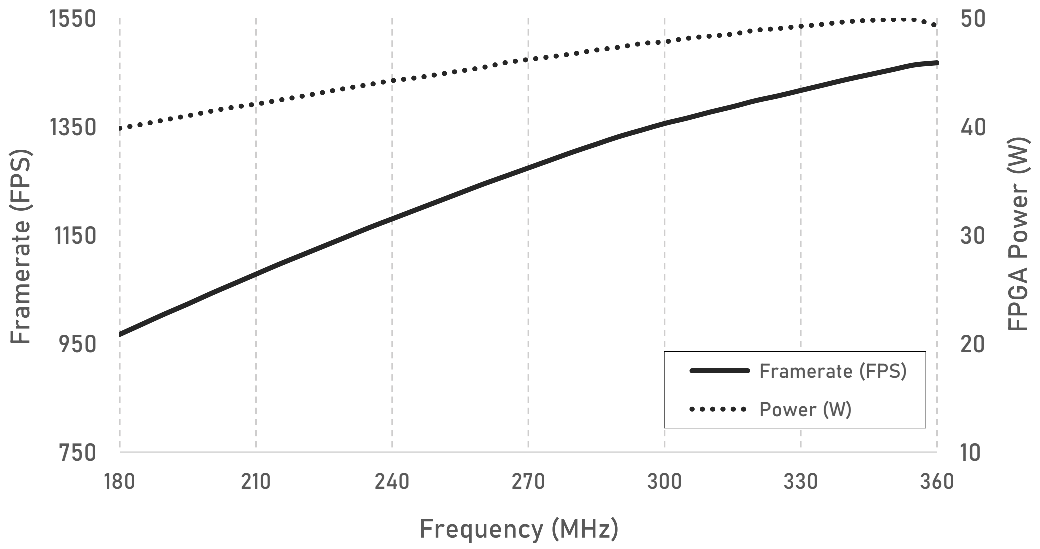


Figure 5.2: Impact of the Memory Subsystem Operating Frequency on Power Consumption and Framerate

5.3. Conclusion

We evaluated the two parts of the proposed methodology in this chapter. The algorithms based on the next-fit dynamic heuristic demonstrated to be capable of solving the memory packing problem significantly faster than the state of the art simulated annealing algorithm, and typically also found better solutions; resulting in a greater amount of BRAM savings.

After experimentally deriving the maximum attainable clock frequencies for several accelerators, the identified solutions were implemented in hardware based on the decoupled weights architecture, and were compared against the original implementations. For all designs a trade-off between additional LUT expenditure and BRAM savings was observed. It strongly varies per case whether this trade-off is advantageous. Nevertheless, we found that the CNV-W1A1 accelerator design can be ported to smaller devices without any loss in throughput. Additionally, we investigated the impact of the frequency of the memory subsystem on the computational throughput, and found that the applied techniques offer a great amount of flexibility, even if the desired clock frequency can not be attained.

In the next chapter the research questions are answered, and recommendations for future work are provided.

6 Conclusions

Central to the conducted work in this thesis was a set of two research questions: *"How can the BRAM utilization efficiency of FPGA neural network inference accelerators be improved without negatively impacting the system throughput?"* and *"What is the hardware cost to realize this approach?"*. To answer these questions, relevant literature was explored in the field of FPGA technology and neural network inference accelerator architectures in Chapter 2, after which the fundamental cause of the BRAM bottleneck was uncovered. As an outcome of this research, a new memory-efficient architecture for dataflow neural network inference accelerators was proposed in Chapter 3, and its associated cost in terms of hardware was characterized. Furthermore, in Chapter 4 two algorithms were proposed to reduce the BRAM utilization by intelligently clustering weight buffers. These algorithms were evaluated in conjunction with the proposed architecture in Chapter 5. In this final chapter, the posed research questions are answered, and pointers for future work are provided.

6.1. Research Questions

How can the BRAM utilization efficiency be improved without negatively impacting the system throughput?

In dataflow inference accelerator architectures the BRAM utilization inefficiencies are caused by a mismatch between the shapes of the weight buffers, which are dictated by the throughput requirements of the corresponding neural network layers, and the physical dimensions of the BRAM modules on FPGA devices.

Opportunities to reshape the weight buffers, in order to improve their mapping efficiencies to BRAM, exist in decoupling the memory subsystem from the compute units. Here the memory subsystem is placed in a higher frequency clock domain relative to the compute units. With this approach, multiple parameters that require simultaneous access can be clustered within a single BRAM instance, while preventing degraded computational throughput of the accelerator. The obtained results indicate that it is readily possible to increase the clock frequency of the memory subsystem on small embedded devices. However, it can potentially be more challenging to meet timing closure on 2.5D planar die packages. Regardless, for accelerator designs that are constrained for resources (i.e. the designs that would benefit the most from memory reductions), it proved to be possible to attain operation frequencies for the memory subsystems that were a factor of 2 - 2.5× higher than that of the compute units. This allows four to five buffers to be clustered within the same physical 2-port BRAM, without degrading the computational throughput.

The possibility to cluster more buffers within the same BRAM instance gave rise to the following problem: "How can the buffers be clustered such that the overall BRAM cost is reduced?". Algorithmically finding the optimal configuration of clusters, such that the amount of utilized BRAM modules is minimized, can be considered to be a custom, constrained variant of the bin packing problem. Hybridized metaheuristics like simulated annealing and genetic algorithms that incorporate traditional bin packing heuristics, demonstrate to be capable of quickly converging to optimal packing solutions. The applied strategies yield BRAM reductions of up to 30%. Furthermore, no loss in computational throughput is observed on embedded devices, where the the memory subsystem's operating frequency is twice that of the compute units.

Moreover, all of the cluster configurations were generated within 5 seconds, which allows the memory packing algorithm to be integrated within design space exploration tools for CNN inference accelerators. To elaborate on the last statement, design space exploration tools often have to perform numerous evaluations to identify Pareto optimal designs [67]. Since the memory packing algorithm would have to be invoked for each of the candidate solutions, this algorithm must be fast in order to be feasible for integration into the toolchain.

What is the hardware cost to realize this approach?
The cost to implement the decoupled weights architecture, as proposed in this thesis, is dominated by the hardware cost associated with implementing the CDC synchronization logic and asynchronous FIFOs. On its turn, the hardware cost for implementing the CDC logic and FIFOs depends on: the amount of neural network layers that are packed and streamed across different clock domains, and the width of these streams. In order to justify the LUT overhead associated with the application of the memory packing strategy to a particular layer, the savings have to be considered on a case-by-case basis.

Aside from the additional CDC logic, certain memory packing strategies might introduce additional routing congestion. Within this work both inter- and intra-layer packing configurations were evaluated on monolithic, as well as, multi-die FPGA devices. The intra-layer memory packing strategy enforces the restriction that only weight buffers from the same neural network layer can be clustered together, while the inter-layer memory packing strategy considers all packing configurations, such that the BRAM cost can be reduced as much as possible. From the obtained results no significant impact on the attained clock frequency was observed between the two packing strategies. However, the impact of the additional routing congestion should be evaluated per design, and as of such, no general statement can be made.

6.2. Outcome

The proposed methodology has been applied to several neural network inference accelerators, and has demonstrated to be capable of significantly reducing the BRAM cost. The CNV-W1A1 accelerator design can be ported to the Zynq-7012S that contains considerably less LUT and BRAM resources than the Zynq-7020. The alternative strategy to fit this design would be to increase the folding factors (as defined in section 2.2.3), and thus exploit less parallelism. This strategy is clearly sub-optimal as it reduces the throughput of the accelerator. Evidently, the ability to port accelerator designs to smaller devices does not solely increase scalability, but also essentially reduces the monetary cost for a particular computing solution.

The obtained BRAM savings did not immediately yield benefits for all of the cases that were included in the evaluation. The CNV-W2A2 accelerator design already utilizes the BRAM resources efficiently, which results in the fact that even after applying memory packing, the design can not be ported to smaller devices.

The situation for the ResNet-50 accelerator design is more complicated. The operating frequency is negatively impacted by the incurred LUT overhead, and thus, a regression in computational throughput is observed. The identified candidate devices to port the design to are the Alveo U200 and U280. However, since these devices have different SLR configurations (i.e. both devices are composed of three SLRs, but contain different amounts of LUTs and BRAM per SLR), and thus require different floorplans, it is arduous to predict the impact on the computational throughput. Nonetheless, the resource utilization estimates generated by FINN, and the obtained synthesis results from the design as implemented on the U250, suggest that there is a possibility to port the design to the U280. Similarly, due to the BRAM utilization reduction, there exists an opportunity to increase the bit precision of the weight values to two bits on the U250, which enables inference at a higher accuracy.

Aside from modifying existing accelerator designs, an alternative strategy would be to address the problem at the source. When FINN generates the accelerator designs, the proposed memory packing algorithm can provide more accurate estimates for the required amount of BRAM to implement the design on FPGA. Highly parallel accelerator designs that would otherwise be deemed unfeasible, can then potentially still be implemented; effectively increasing the performance of the accelerator designs created by FINN. Since the proposed memory packing algorithm converges fast (even for large multi-die accelerator designs), the algorithm can be integrated within the FINN framework.

6.3. Contributions

In this thesis, research was pursued to improve the memory utilization efficiency of neural network inference accelerators on FPGAs. The main contributions of this research are:

- a memory-efficient dataflow architecture for neural network inference accelerators on FPGAs that demonstrates BRAM reductions of up to 30% compared to the integrated weights architecture.
- the formulation of the memory packing problem as a variant of the bin packing problem with custom hardware related constraints.
- a new heuristic that is embedded in simulated annealing and genetic algorithm approaches. Both approaches rapidly solve the memory packing problem, and are over 200× faster than a state-of-the-art simulated annealing algorithm for large accelerator designs.
- an analysis of the impact of different memory packing strategies on the maximum attainable operating frequencies, and BRAM utilization of FPGA dataflow accelerators.

6.4. Future Work

The memory packing algorithm and decoupled weights architecture that are proposed in this thesis enable increased memory resource utilization efficiency in modern FPGAs. The applied techniques in this approach are universal, and can be applied to any digital circuit where parameter values are read out in a predictable manner. Regardless, the obtained results suggest that the methodology should be applied with care, since there is a trade-off between BRAM savings and a LUT overhead.

Improvements can be made on this front by performing multi-objective optimization that explicitly takes throughput, BRAM and LUT utilization of the design into consideration during the evolution process. As demonstrated on the ResNet-50 accelerator designs, reducing the BRAM count while disregarding the additional LUT expenditure might adversely impact the performance of the design. Since these are strongly conflicting objectives, more advanced multi-objective optimization strategies for evolutionary algorithms should be adopted to obtain a set of Pareto optimal design points [68, 69, 70].

Furthermore, while the proposed methodology targeted the optimization of the utilization efficiency for BRAM, the methodology can be extended to URAM as well. These memory resources are larger than BRAM modules, and combined with the fact that these resources do not support the various aspect ratios, are more likely to be underutilized. For some of the designs that were involved in the evaluation, the BRAM utilization efficiency was approaching the theoretical maximum in both inter- and intra-layer configurations with an imposed cardinality constraint of four. Since URAM is both wider and deeper than BRAM, the same packing strategies would still leave a significant portion of these modules underutilized. Moreover, aside from packing buffers on top of each other, width-wise packing where separate words are stored next to each other at the same address, should also be considered to compensate for the lack of aspect ratios and added width (72-bit vs 18-bit).

A
Submitted Conference Paper (GECCO'20)

Evolutionary Bin Packing for Memory-Efficient Dataflow Inference Acceleration on FPGA

Mairin Kroes
TU Delft
m.i.kroes@student.tudelft.nl

Lucian Petrica
Xilinx
lucianp@xilinx.com

Sorin Cotofana
TU Delft

Michaela Blott
Xilinx

ABSTRACT

Convolutional neural network (CNN) dataflow inference accelerators implemented in Field Programmable Gate Arrays (FPGAs) have demonstrated increased energy efficiency and lower latency compared to CNN execution on CPUs or GPUs. However, the complex shapes of CNN parameter memories do not typically map well to FPGA on-chip memories (OCM), which results in poor OCM utilization and ultimately limits the size and types of CNNs which can be effectively accelerated on FPGAs. In this work, we present a design methodology that improves the mapping efficiency of CNN parameters to FPGA OCM. We frame the mapping as a bin packing problem and determine that traditional bin packing algorithms are not well suited to solve the problem within FPGA- and CNN-specific constraints. We hybridize genetic algorithms and simulated annealing with traditional bin packing heuristics to create flexible mappers capable of grouping parameter memories such that each group optimally fits FPGA on-chip memories. We evaluate these algorithms on a variety of FPGA inference accelerators. Our hybrid mappers converge to optimal solutions in a matter of seconds for all CNN use-cases, achieve a reduction of up to 65% in OCM required for deep CNNs, and are up to 200× faster than current state-of-the-art simulated annealing approaches.

ACM Reference Format:
Mairin Kroes, Lucian Petrica, Sorin Cotofana, and Michaela Blott. 2020. Evolutionary Bin Packing for Memory-Efficient Dataflow Inference Acceleration on FPGA. In *Proceedings of the Genetic and Evolutionary Computation Conference 2020 (GECCO '20).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have achieved state of the art performance on image classification, object detection, image semantic segmentation and other computer vision tasks and have become an important part of both data-center and embedded workloads. Modern high-accuracy CNNs are typically deep, i.e. they consist of a large number of convolutional layers, each trained through backpropagation. The large number of layers is a key enabler of CNN performance but creates difficulties for their implementation due to the large total number of parameters and the high latency of executing very deep CNNs which makes real-time inference difficult. To reduce inference latency, modern systems typically utilize parallel computing accelerators for CNN inference, either GPUs or Field Programmable Gate Arrays (FPGAs). To date, on FPGA, custom dataflow CNN inference accelerators have achieved the best combination of low latency, high throughput, and low power dissipation [1]. In the custom dataflow approach, each CNN layer is executed on a dedicated section of the FPGA and its parameters are stored in a dedicated part of the FPGA on-chip memory (OCM), such that the inference process can occur without data ever leaving the FPGA chip, eliminating the high latency and power dissipation associated with external memory reads and writes.

Of course, a key prerequisite for the custom dataflow approach is for the CNN parameters to fit in FPGA on-chip memory. While quantization and pruning techniques have been successful in reducing the overall size of the CNN parameter memories, one aspect of CNN accelerator design has not been approached in previous work: how to optimally map the diversely-shaped CNN parameter memories to FPGA OCM. In the case of several of the published CNN accelerators, the mapping efficiency is below 70%, i.e. for structural reasons 30% of the FPGA OCM bits cannot be utilized. This inefficiency is proportional to inference throughput in frames per second and also increases with the CNN depth.

In this paper we introduce a CNN accelerator memory subsystem construction methodology which enables increased memory mapping efficiency. We achieve this by co-locating multiple CNN parameter memories in a single bank of FPGA OCM, and taking advantage of the multi-port capability of the FPGA memories to minimize the impact to CNN inference performance. Given this design approach, the challenge becomes how to optimally pack CNN parameter memories into available physical memories to achieve the highest memory utilization efficiency within certain throughput constraints. Additionally, given the recent popularity of design space exploration (DSE) techniques for automatically discovering pareto-optimal CNN accelerator configurations [19, 22], any memory packing algorithm must be very fast to be able to run in the inner loop of a DSE process. Given these considerations, the contributions of this paper are as follows:

- We present a novel heuristic which hybridizes Genetic Algorithms and Simulated Annealing with traditional bin packing algorithms to achieve high-efficiency mapping of CNN parameter memories to FPGA OCM
- We apply the proposed algorithms on a number of CNN accelerators from previously published work, as well as 3

GECCO '20, July 8–12, 2020, Cancun, Mexico
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

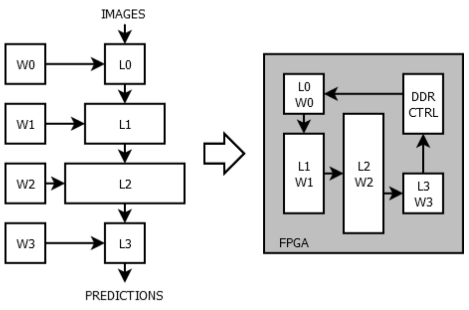


Figure 1: CNN Mapped to Dataflow Accelerator on FPGA

new accelerators, and demonstrate an increase of up to 33% in the mapping efficiency, or 65% reduction in required FPGA OCM size, achieved after running the optimization for under 5 seconds in most cases.

- We compare our proposed algorithms against the state-of-the-art simulated annealing based algorithm in the field and observe 8% increase in efficiency as well as over 200× increase in optimization speed.

The rest of the paper is structured as follows. Sections 2 and 3 provide background and state the problem. In section 4 we present our genetic algorithm and simulated annealing based mapping algorithms. We show in section 5 that our algorithms improve on previous work in this domain [7, 24] in both quality of results (i.e final mapping efficiency) and optimization speed for large dataflow CNN accelerators.

2 FPGA ACCELERATED CNN INFERENCE

2.1 Accelerator Architectures

We distinguish two typical approaches of accelerating CNN inference on FPGAs. The first approach leverages a GPU-like matrix of processing engines implemented in FPGA, where the corresponding scheduler determines how to best map the operations of the CNN onto the hardware architecture, typically resulting in a layer by layer compute pattern. The second approach, which is the target of our efforts, leverages feed forward dataflow implementations where the accelerator implements a pipeline of per-layer dedicated compute and associated on-chip parameter memories, as illustrated in Fig. 1. All layers of the neural network are in essence spatially unrolled. Benefits are typically lower latency and and higher throughput. However, as all weights remain in OCM, this becomes the primary bottleneck and limits the layer depth and type of CNN that can be deployed on a single device.

To alleviate this bottleneck, but also to help with the overall computational burden, many optimization techniques have been proposed, with quantization and pruning being two of the most popular schemes [10]. Quantization is a particularly effective optimization for neural network inference. On smaller image classification tasks such as MNIST, SVHN and CIFAR-10, heavily quantized CNNs can achieve significant memory reduction, directly proportional to the reduction in precision, with small loss in accuracy, even when reducing the precision to 1 or very few bits [3, 25]. Furthermore, novel quantization schemes such as [2], and new training and optimization techniques [18, 26] can potentially recoup the accuracy. Similarly, pruning can dramatically reduce the size of CNN parameter memory by removing all convolutional filters with *sensitivity* (sum of magnitude of all included weights) below a set threshold.

The progress on quantization and pruning has enabled the implementation of multiple dataflow accelerators and accelerator frameworks [1, 9, 23] for binarized and quantized CNNs in FPGA. Nevertheless, most dataflow accelerators described in previous work still target relatively small binarized CNNs which achieve acceptable accuracy on simple image and speech processing tasks (e.g. classification for MNIST, CIFAR10, SVHN datasets). Dataflow-style FPGA-accelerated binarized CNNs for the Imagenet [5] 1000-class classification problem have been developed utilizing the FINN [1] and ReBNet [9] accelerator frameworks, but have limited Top-1 accuracy compared to equivalent GPU and CPU inference solutions, in the range of 40-50%.

To date, achieving state of the art accuracy with dataflow accelerators in FPGA remains a challenge. While approaches such as utilizing higher weight precision, e.g. 2-bit ternary quantization [17] instead of binary, or deeper NNs such as ResNet-50 [11] have the potential to increase achievable accuracy, they also significantly increase the size of the required on-chip weight storage, making dataflow acceleration difficult.

2.2 Memory Efficiency vs. Throughput in Dataflow CNNs

For the remainder of this paper, unless otherwise indicated, we assume FPGA dataflow accelerators are constructed using the architectural principles of the FINN framework [23]. In FINN-style accelerators, convolutions are lowered to matrix multiplications which are computed by carefully scheduling data to multiply-accumulate (MAC) circuitry on the FPGA fabric. The computational throughput of each layer of the accelerator is controlled by several parallelism variables: the number of vector processing elements (PEs), denoted N_{PE}, the vector length of the PE, denoted N_{SIMD}, and the number of pixels processed in parallel, denoted N_{MMV}. The total number of MAC operations executing in parallel at any given time for any layer is equal the product $N_{PE} \times N_{SIMD} \times N_{MMV}$. To fully utilize the fabric computational resources (Look-up Tables - LUTs - and embedded DSP processors) and therefore maximize throughput, we must perform many MACs in parallel.

However, this approach forces specific shapes on the parameter memory, in order to achieve parameter readback at the same rate as the compute. Specifically, in each clock cycle and for each layer, the parameter memory must deliver $N_{PE} \times N_{SIMD}$ parameters to the MAC circuits (N_{MMV} is not relevant because pixels share parameters). As such, the memories storing the parameters must have a word width equal to the product $N_{PE} \times N_{SIMD} \times W$, where W is the bitwidth of each parameter. Therefore, as the parallelism (and inference throughput) increase, parameter memories must become wider, and because the total number of parameters for each layer is constant, the depth of the parameter memory must become smaller. In contrast, FPGA OCM consists of block RAM memories (BRAMs) which have a fixed narrow and deep aspect ratio, e.g. 18-bit wide 1024-deep in Xilinx FPGAs. Because of this

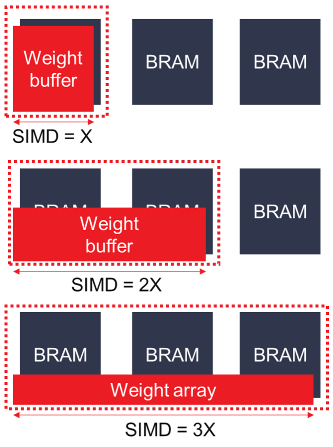


Figure 2: Efficiency decreases with increased parallelism

shape mismatch, CNN parameter memories map inefficiently to BRAM, and given the link between CNN parameter memory shapes and MAC parallelism, high computational throughput implies low BRAM efficiency, and vice-versa.

Figure 2 illustrates this effect. We start from an ideal case of the parameter memory (weight buffer) mapping perfectly to one BRAM. If N_{SIMD} increases by a factor of 2, the shape of the weight buffer must adjust accordingly, and now two adjacent BRAMs must be utilized to each store one half of the buffer. Because the depth has been reduced to half, the efficiency is now 50%, and can be reduced even further if we increase N_{SIMD} more.

In each case, as the parameter memory becomes wider to provide more parameters in each read cycle, it also becomes shallower and utilizes more BRAMs for implementation. We define the physical RAM mapping efficiency as in Equation 1, where D is the depth of the parameter memory, $\lceil\rceil$ denotes rounding up to nearest integer, and W_{BRAM} and D_{BRAM} denote the width and depth of one BRAM respectively, in bits. Here, the numerator indicates the bits required to be stored and the denominator indicates the total capacity of the BRAM required to actually implement the weight buffer, defined as the product of the width and depth of each physical RAM multiplied by the number of utilized RAMs. The efficiency scales inversely proportional to the exploited parallelism, an undesirable effect.

$$E = \frac{N_{PE} \cdot N_{SIMD} \cdot W \cdot D}{W_{BRAM} \cdot D_{BRAM} \cdot \left\lceil \frac{N_{PE} \cdot N_{SIMD} \cdot W}{W_{BRAM}} \right\rceil \cdot \left\lceil \frac{D}{D_{BRAM}} \right\rceil} \qquad (1)$$

Secondly, in the FINN dataflow approach, buffer depth D is proportional to the product of the convolutional kernel size K and the number of channels C. K is typically an odd number, most often 3 or 5 in modern CNN topologies, while C is typically a power of 2 but can be odd if e.g. pruning has been applied to the CNN parameters. Therefore D most of the time does not evenly divide the depth of a physical BRAM, leading to frequent under-fill of the allocated BRAMs.

2.3 DSE for CNN accelerators

It is typically the responsibility of a framework-specific resource allocator or design space exploration tool to set the correct value for each parallelism variable in each layer of the CNN dataflow accelerator to maximize overall throughput while remaining within the OCM capacity and LUT/DSP constraints of the target FPGA. Previous work in this area [19, 22] has demonstrated that extensive automated search in the design space can identify accelerator configurations better than human designers. As FPGA-accelerated CNNs become deeper and the total number of parallelism variables increases, we expect this trend to continue, as long as appropriate tools exist to quickly estimate the LUT, DSP and OCM requirements of an accelerator from a given set of values of the parallelism variables.

3 PROBLEM STATEMENT

Given the strict constraints on the shape of CNN parameter memories, and the fixed shape of FPGA BRAMs, solving the efficiency problem is a matter of finding a way to utilize the space left after one parameter memory has been mapped to a (set of) BRAMs. A straight-forward way to utilize this space is to map a second parameter memory (or more, if possible) in the empty space. In this approach, the efficiency maximizing problem is analogous to a bin packing problem: the problem of trying to pack a set of objects of varying sizes into bins of certain capacities, with a goal to utilize as few as possible bins to pack all the objects. Here we consider the various CNN parameter memories as the objects, and the physical BRAM instances (or combinations thereof) as the bins into which the objects should be packed. Since this problem is NP-hard, good heuristics are required to obtain acceptable solutions fast.

The primary factors that make our memory packing problem different from the classical bin packing problem, is that the bins in this case (the block RAM instances) have a limited number of ports which can be used to for parameter reads. For example, if using Xilinx FPGAs, memories have 2 ports, and if we pack 2 parameter memories in one BRAM, we can read one parameter in every clock cycle from each of the packed memories, and the original throughput of the accelerator is maintained. However, beyond 2 parameter memories per BRAM, access to the parameter memories is achieved through time multiplexing of ports, which implies the MAC unit of the accelerators will not be fed parameters in every cycle and the inference throughput suffers. Therefore, beyond simply filling the bin, a good algorithm must also minimize the number of items per bin in order to preserve the inference throughput. In practice, we desire to set an upper limit to the number of items per bin, a so-called cardinality constraint. Secondly, block RAMs can be combined in various ways such that the bins in our case can have variable widths and depths, and therefore have variable capacities.

These differences significantly deteriorate the efficacy of classical bin packing heuristics that are covered in the literature, since these heuristics build on the concept that an unlimited amount of small items can be used to fill up bins that are almost full. The cardinality constrained version of the bin packing problem was initially explored by Krause et. al. [14] and Kellerer and Pferschy [12]. Though despite taking the cardinality constraints explicitly into account, they obtain poor packing results and assume fixed bin sizes, which make these algorithms unsuitable for mapping CNN parameter memories to physical RAMs on FPGA.

Some of the highest performing bin packing algorithms explored in recent literature use genetic algorithms to solve the bin packing problem [6, 20]. In these works, genetic algorithms are combined with classical bin packing heuristics to deliver high quality packing results. More importantly, the proposed strategies utilize an efficient chromosome encoding scheme that was introduced by Falkenauer and Delchambre [7]. This scheme allows for better exploration of the search space. Since these implementations do not take cardinality constraints into account, some modifications are required before these strategies can be applied to the memory packing problem.

The specific problem of efficient mapping of logical buffers to block RAMs has also been approached in MPack [24], where a simulated annealing algorithm is utilized to discover a good mapping of multiple logical buffers in a single block RAM, but is only demonstrated on relatively small examples compared to modern inference accelerators.

4 PACKING CNN MEMORIES TO FPGA

Previously we established that solving the memory packing problem equates to solving a bin packing problem with a set of hardware constraints, and that genetic algorithms (GA) and simulated annealing (SA) are promising approaches to solve the bin packing problem within the stated constraints. Existing realizations of GA bin packers incorporate recombination techniques that yield good results when there is no upper limit on the amount of items that can be packed into a bin of fixed capacity. Conversely, Vasiljevic and Chow solve the memory packing problem with a simulated annealing approach that explores the search space with random movement of items between bins, referred to as buffer swaps, which can be inefficient for large numbers of bins. We improve on these approaches by introducing *next-fit dynamic*, which is a new heuristic that explicitly takes the custom memory packing related constraints into account and enables a faster and more efficient exploration of the search space. We then embed this heuristic into GA and SA.

4.1 Next-Fit Dynamic Heuristic

Next-fit dynamic (NFD) is a recombination technique that is based on the simplest bin packing heuristic next-fit, which has time complexity $O(n)$. As can be seen in Algorithm 1, the NFD heuristic takes as input a list of bins, where each bin contains one or more items. Out of this list we mark the bins that map poorly to BRAM, using an efficiency threshold, decompose the bins into their constituent buffers, and subsequently try to re-pack the buffers into new bins, dynamically adjusting the size of the bin currently being packed according to known BRAM composition rules, e.g. a bin can have widths multiple of W_{BRAM} bits and depths multiple of D_{BRAM}.

By design, NFD only adds an additional buffer into an already populated bin if the resulting bin composition leads to less BRAM space being wasted. We allow, however, small admission probabilities (P^{w}_{adm} and P^{h}_{adm}) that occasionally accept packing configurations that do not immediately improve the mapping efficiencies of the width and height of a bin respectively, to increase the exploration ability of the heuristic and its embedding optimization algorithm.

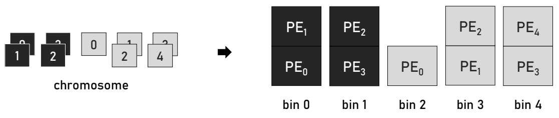


Figure 3: Bin Per Gene Chromosome Encoding

The NFD strategy enables us to explore large search spaces faster and gives us more control over bin compositions (i.e. not unnecessarily packing buffers if it won't lead to BRAM savings). Moreover, this additional control also enables us to add supplementary restrictions. One example of such a restriction is to exclusively explore bin packing configurations that contain buffers belonging to the same neural network layer (referred to as intra-layer packing), which reduces the average distance between parameter memories and their corresponding MAC circuits on FPGA after the resulting accelerator is implemented, maximizing the operating frequency of the inference accelerator.

Algorithm 1: Next-Fit Dynamic (NFD) Heuristic

```
Input: list of packed bins
Output: list of repackaged bins
sublist = calculateMapEfficiency(list, threshold);
shuffle(sublist);
for buffer in sublist do
    if bin height == 0 then
        bin ← buffer;
        update(bin width, bin height);
    else
        calculate(new bin height);
        gap = calculateGap(BRAM height, bin height);
        new gap = calculateGap(BRAM height, new bin height);
        if length bin < max bin height AND
        ((new gap < gap OR rnd() < $P_{adm,h}$) AND
        (bin width == buffer width OR rnd() < $P_{adm,w}$)) then
            bin ← buffer;
            update(bin width, bin height);
        else
            list ← bin;
            reset(bin, bin width, bin height);
            bin ← buffer;
if length bin > 0 then
    list ← bin;
```

4.2 Genetic Algorithm Bin Packing

In this work we employ a genetic algorithm that utilizes the so called "bin per gene" chromosome representation as illustrated in Figure 3. Here a bin refers to a group of CNN parameter memories that will be packed together, so each gene is a list of CNN parameter memories.

The genetic algorithm pseudocode is listed in Algorithm 2 and consists of repeating rounds of evolution. In each round, on a given population of bin packing solutions, we apply mutation with a probability P_{mut} for each individual in the population, and then perform fitness evaluation of the population. Using the fitness values, we perform tournament selection where we extract the best solution out of a randomly selected subset of solutions from the current population, and add it to a new population. The selection process is repeated until the new population has the same count as the preceding population, which it replaces and the next evolution round starts.

Algorithm 2: Genetic Algorithm

```
Input: list of partitions, max bin height
Output: BRAM cost, list of packed bins
1  initialize(population);
2  while not converged do
3      for individual in population do
4          if rnd() < $P_{mut}$ then
5              mutate(individual);
6          calculateFitness(individual);
7      while new population count < population count do
8          new individual = tourSelect(population, tour size);
9          new population ⟵ new individual
10     population = new population;
```

Mutation. The mutation operator is the driving factor in the process of exploring the search space. Two different operators are utilized in this work. The first method is the buffer swap method mentioned in [24]. Buffer swapping entails moving buffers to different bins, which changes the packing configuration and its associated BRAM cost. The second method is the next-fit dynamic recombination technique - we select a number of genes, unpack the corresponding bins and mark these memories for repackaging with NFD.

Fitness and Selection. The factor that determines which individual (solution) wins the tournament, is the fitness of that particular individual. In our work we employ a multi-objective fitness function where we compute a weighted sum between BRAM cost and the layer count per bin. Solutions that result in the lowest BRAM cost, and do so with bin configurations that contain buffers from as few as possible different layers are more likely to make it into the next generation. As time progresses, only the solutions that best meet these criteria will remain.

4.3 Simulated Annealing Bin Packing

Simulated Annealing is an optimization algorithm first introduced by Kirkpatrick et. al. in [13]. It is similar to general hill climbing algorithms, but its distinguishing feature is that it occasionally jumps between hills (i.e. makes large optimization steps) to prevent getting stuck in a local optimum. This escaping behaviour is modeled by random thermal motion that forces the algorithm to sometimes perform (locally) disadvantageous actions. By default, the algorithm accepts an action if it leads to a solution that optimizes a certain cost function. If the action leads to a worse solution, that action might still be accepted with a certain probability $P_A(T)$ as described in Equation 2. This probability approaches 1 for high temperatures and decays exponentially as the temperature decreases. As a result, the algorithm will frequently jump between hills at the start of the annealing process, and then selects a hill to climb in the final phase.

$$P_A(T) = e^{\frac{-\Delta E}{T}} \tag{2}$$

Our implementation of the simulated annealing memory packer follows the approach as described in [24], and the general flow is as described in Algorithm 3. We first generate a random, yet feasible memory packing solution that adheres to the cardinality constraint. Then we calculate the BRAM cost for this solution. Finally, the optimization process commences as described before. For the different versions of the SA either the simple buffer swap or next-fit dynamic are used to “perturb” the solution. If a perturbation was beneficial, the perturbed solution is immediately accepted. Otherwise, the acceptance probability P_A is calculated according to the current temperature, and the acceptance of the bad move might be reconsidered.

Algorithm 3: Simulated Annealing

```
Input: list of partitions, max bin height, $T_0$, $R_c$
Output: BRAM cost, list of packed bins
1  initilize(solution, T);
2  cost = costFunction(solution);
3  iter = 0;
4  while not converged do
5      T = calculateTemperature($T_0$,$R_c$,iter);
6      candidate = perturb(solution);
7      new cost = costFunction(candidate);
8      $P_A$ = probability(cost, new cost, T);
9      if new cost < cost OR rnd() < $P_A$ then
10         solution = candidate;
11     increment iter;
```

In all, we have defined three novel algorithms for solving the CNN parameter memory to FPGA OCM mapping problem: genetic algorithm using buffer swap and NFD as mutation operators, denoted GA-S and GA-NFD respectively, and simulated annealing using NFD as perturbation mechanism, denoted SA-NFD. The simulated annealing with buffer swap, denoted SA-S, has been published in [24] but not evaluated for systems of the size of modern CNN inference accelerators.

5 EVALUATION

5.1 CNN Use-Cases

We evaluate our buffer to BRAM mapping algorithms on several CNN-based object detection and classification accelerators selected from previous work and listed in Table 1. The table indicates the source publication for each accelerator and also the shapes and number of parameter memories of each accelerator, which serve as input for our buffer to BRAM packing algorithm.

Small Image Classifiers. CNV-WxAy CNNs belong to the BNN-Pynq[1] suite of object classification accelerators. They are FINN-style [23] FPGA accelerators and target embedded (relatively small)

[1] https://github.com/Xilinx/BNN-PYNQ

FPGA devices such as the Zynq-7020. CNV-W1A1 utilizes binary (1-bit) quantization [4] while CNV-W2A2 utilizes ternary (2-bit) quantization [17]. Both CNNs are trained on the CIFAR-10 [15] dataset and are able to distinguish between 10 classes of common objects (e.g. birds, cars, dogs, etc.).

Mid-Size Image Classifiers. DoReFaNet and ReBNet are medium-size CNNs trained for object classification on the 1000-class ImageNet [5] dataset. These CNNs are both quantized versions of AlexNet [16], a popular image classification CNN topology, use binary (1-bit) weights, and consist of 5 convolutional layers and 3 fully-connected layers. However, they differ in the folding factors utilized for their implementation and therefore in the shapes of their weight memories, and as such are treated separately in our evaluation. DoReFaNet was first binarized in [25] and implemented in FPGA in [23]. ReBNet was described and implemented in FPGA in [9] where it is denoted 'Arch3'.

Large Image Classifiers. ResNet-50 [11] is a high-accuracy classification CNN designed for high-accuracy image classification on the ImageNet dataset. To our knowledge, no ResNet-50 dataflow implementation currently exists, so we develop a folding solution (i.e. define values for the parallelism variables of each layer) according to the design principles of FINN accelerators [23], assuming binarized weights and aiming to fit within the LUT capacity of the largest commercially available Xilinx FPGA, the Alveo U250. We also implement larger ResNet variants: ResNet-101 and ResNet-152 which are approximately 2 and 3 times deeper than ResNet-50 respectively but share the overall structure.

Object Detectors. Tincy-YOLO was first published in [1] and is a binarized-weight variant of YOLO [21], a popular object detection CNN. It is a fully convolutional design consisting of 9 layers, 6 of which utilize binary weights while two utilize 8-bit weights.

5.2 Methodology

GA Fine-Tuning. We first analyze the effect of population size on the quality of results (packing efficiency) and convergence speed of GA-NFD to pack the ResNet-50, in order to derive guidelines with regard to the optimal population sizes. We evaluate a range of population sizes from 5 to 400, with each experiment repeated 5 times with different random seeds to reduce variability. For each experiment we run the optimization process for 7 minutes, which was empirically determined to ensure convergence for all population sizes under evaluation.

Packing Algorithm Comparison. We compare the GA and SA packing algorithms with and without NFD, in terms of wall-clock time to convergence and quality of results, for each of the accelerators under evaluation. For all algorithms we impose a cardinality constraint of a maximum of 4 parameter memories per physical BRAM. The reported time to convergence is defined as the amount of time it takes each algorithm to attain a packing result that is within 1% of the discovered minimum. For each convergence experiment, we evaluate 10 different initial random seeds.

Mapping Efficiency Increase. We calculate the efficiency of mapping parameter memories to FPGA OCM for each of the CNN accelerators, targeting a maximum bin height of 4 and utilizing both inter-layer (unconstrained) and intra-layer packing strategies. In this set of experiments, we utilize a single packing algorithm, to be selected from the comparisons described above.

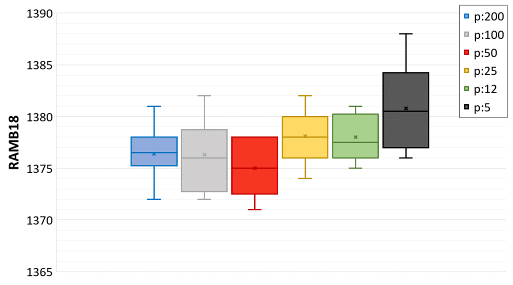


Figure 4: QoR comparison for different population sizes on ResNet-50 optimization (GA-NFD)

5.3 Experimental set-up

We implemented the GA and SA packing algorithms in Python code utilizing the DEAP evolutionary computation library [8] (version 1.3.0). We execute the packing algorithms in single-thread mode on a server equipped with Intel Xeon Silver 4110 CPUs, 128 GB of system RAM, and SSD storage. We measure time using Python's time package.

To enable us to check the BRAM counts of a packing solution in hardware, we implemented in Verilog HDL code a circuit representing a bin, (i.e. a set of assembled BRAMs with associated addressing logic for up to 6 co-located CNN parameter memories) and a Python-based post-processor which takes a packing solution and generates a Vivado 2019.1 project and Block Design consisting of bin instances configured according to the packing solution. We synthesize the resulting Vivado project and compare the post-synthesis BRAM counts to the software-estimated counts. We observe no difference in practice between these measurements.

6 RESULTS

6.1 Effect of GA Population Size

The run-time and QoR (Quality of Results) of genetic algorithms in general depends on the population size utilized. The population size essentially dictates how many candidate solutions are subject to selection and probabilistic mutation at any particular generation. In Figure 4 the results of solving the memory packing problem for ResNet-50 with GA-NFD at varying population sizes are displayed. As can be observed, the algorithm is able to find slightly better results as we scale the population size up to 50. Past this population size we observe a slight regression in performance, however the range of variation in final result after 7 minutes of optimization is very small. Overall, we conclude that population size does not greatly affect the QoR.

Generally we expect that genetic algorithm experiments utilizing larger population sizes will converge in a smaller number of iterations but those iterations will each be longer in duration than a corresponding iteration for a smaller population size. In this work

Table 1: Baseline dataflow accelerators

Accelerator:	**CNV-W1A1 [23]**	**CNV-W2A2 [23]**	**Tincy-YOLO [1]**	**DoReFaNet [1]**	**ReBNet [9]**	**RN50-W1A2**
Memory Shapes $N_{PE} \times (N_{SIMD}, D, W)$	$16 \times (32, 144, 1)$ $16 \times (32, 288, 1)$ $4 \times (32, 2304, 1)$ $4 \times (1, 8192, 1)$ $1 \times (32, 18432, 1)$ $1 \times (4, 32768, 1)$ $1 \times (8, 32768, 1)$	$8 \times (16, 576, 2)$ $8 \times (16, 1152, 2)$ $4 \times (1, 8192, 2)$ $4 \times (8, 9216, 2)$ $3 \times (2, 65536, 2)$ $1 \times (8, 73728, 2)$	$16 \times (32, 144, 1)$ $25 \times (8, 320, 1)$ $16 \times (32, 144, 1)$ $80 \times (32, 2304, 1)$	$136 \times (45, 72, 1)$ $64 \times (34, 108, 1)$ $32 \times (64, 108, 1)$ $68 \times (3, 144, 1)$ $8 \times (8, 64000, 1)$ $4 \times (64, 65536, 1)$ $8 \times (64, 73728, 1)$	$64 \times (54, 256, 1)$ $64 \times (25, 384, 1)$ $64 \times (36, 384, 1)$ $64 \times (32, 576, 1)$ $128 \times (64, 1152, 1)$ $40 \times (50, 2048, 1)$ $128 \times (64, 2048, 1)$	$368 \times (32, 256, 1)$ $32 \times (64, 256, 1)$ $192 \times (64, 288, 1)$ $176 \times (32, 1024, 1)$ $32 \times (64, 1024, 1)$ $96 \times (64, 1152, 1)$
Total Buffers:	43	28	137	320	552	896

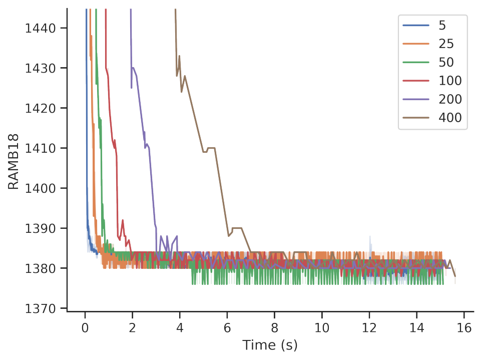


Figure 5: Convergence speed for different population sizes on ResNet-50 optimization

Table 2: SA and GA Hyperparameters

Accelerator	**GA**					**SA**	
	N_p	N_t	P^w_{adm}	P^h_{adm}	P_{mut}	T_0	R_c
CNV-W1A1	50	5	0	0.1	0.3	30	1
CNV-W2A2	50	5	0	0.1	0.3	30	2
Tincy-YOLO	75	5	0	0.2	0.4	30	1
DoReFaNet	50	5	0.1	0.3	0.4	30	1
ReBNet Arch3	75	5	1	0.2	0.4	30	1
RN50-W1A2	75	5	0	0.1	0.4	40	0.004
RN101-W1A2	75	5	0	0.1	0.4	40	0.004
RN152-W1A2	75	5	0	0.1	0.4	40	0.004

we are interested in optimizing wall-clock time to solution. To identify the population size that minimizes wall clock time, we pack the respective networks using increasingly larger population sizes and analyze the convergence curves. The results of the population size analysis for ResNet-50 and GA-NFD are illustrated in Fig. 5. The best compromise between rapid convergence (in wall-clock time) and quality of results is achieved at a relatively small population size of 50 while the largest population size experiment converged the slowest. This indicates that there is limited benefit from increasing population size to large values. For the experiments performed in this paper, population sizes of approximately 50 appear optimal.

6.2 Packing Algorithm Comparison

In this section the performance of the developed heuristic will be evaluated. As baseline we compare the SA and GA that incorporate next-fit dynamic against SA and GA implementations that use the buffer swap methodology. Both versions of the GA and SA were applied to solve the memory packing problem for the networks as listed in Table 1. We performed extensive hyperparameter tuning for all algorithms to ensure optimal quality of results. The corresponding hyperparameter settings can be found in Table 2 for simulated annealing and for the genetic algorithm.

The runtime comparison results can be found in Table 3 for all networks, with best results highlighted in bold where a clear winner could be distinguished. It has to be mentioned that for the GA the minimum BRAM count of all candidate solutions in a particular generation is tracked. All the algorithm are capable of quickly solving the packing problem for the smaller CNV networks. However, as we increase the problem size (e.g. Tincy-YOLO, DoReFaNet, ResNets) the NFD versions of the algorithms are capable of solving the packing problem much faster, and with higher quality of results. For the ResNets in particular, the NFD algorithms are capable of finding solutions that require up to 8% less BRAM to implement and reduce the required runtime by a factor of more than 200× compared to SA-S. GA-S provides poor QoR especially for larger networks and is also slower than all other algorithms. In general, GA-NFD achieves the best QoR while SA-NFD is the faster.

The outlier here is the ReBNet Arch3 accelerator design. This design contains memory partitions with a large variety in widths (SIMD lanes), which causes difficulty for NFD as it is forced to pack together parameter memories with misaligning widths. In order to compete on the metrics as presented in Table 3 (i.e. BRAM cost and runtime) the hardware constraints had to be relaxed significantly, as is reflected by the high admission probabilities — P^w_{adm} and P^h_{adm} — as listed in Table 2. Nevertheless, the NFD-based algorithms (especially SA-NFD) arrive at a packing solution significantly faster than buffer swap based GA and SA.

To emphasize the differences in QoR, aside from potentially greater BRAM reductions, the NFD algorithms also provide packing solutions with more ideal bin configurations from a hardware design perspective. The reason for this is that the heuristic typically only packs buffers in bins when it improves the mapping of these bins. As a consequence, the NFD algorithms typically provide packing solutions that contain, on average, bins of lower height, which results in lower throughput penalty.

Table 3: Memory packing comparison of SA and GA

	Buffer Swap				Next-Fit Dynamic			
Accelerator	t_{SA-S} (s)	t_{GA-S} (s)	N_{BRAM}^{SA-S}	N_{BRAM}^{GA-S}	t_{SA-NFD} (s)	t_{GA-NFD} (s)	N_{BRAM}^{SA-NFD}	N_{BRAM}^{GA-NFD}
CNV-W1A1	0.1	0.2	96	96	0.1	0.1	97	96
CNV-W2A2	0.1	0.1	188	190	0.1	0.1	190	188
Tincy-YOLO	1.8	1.7	420	428	**0.1**	0.2	430	420
DoReFaNet	1.0	1.6	3823	3826	**0.1**	0.2	3849	**3794**
ReBNet Arch3	40.1	57.5	**2301**	2313	**2.2**	28.9	2483	2352
RN50-W1A2	239	290	1404	1472	**0.8**	1.7	**1368**	1374
RN101-W1A2	615	935	2775	3055	**0.9**	3.3	**2616**	**2616**
RN152-W1A2	1024	1354	3864	4422	**1.5**	4.9	3586	**3584**

6.3 Achievable Efficiency Increase

Finally, we applied the memory packing methodology to the accelerators as listed in Table 1, utilizing GA-NFD which achieved the best overall packing performance in Table 3. The packing results of the accelerators in original and two different packed configurations are presented in Table 4.

As briefly mentioned before, the term "intra" refers to the fact that we only pack buffers corresponding to the same neural network layer together, while in inter-layer packing configurations we do not impose such constraints. The results are presented in terms of BRAM necessary to store the CNN parameters, the resulting mapping efficiency as dictated by Equation 1 and the reduction in memory footprint Δ_{BRAM}.

While the smaller accelerators benefit from the GA-NFD packing, the most benefit is achieved for the ResNet accelerators, which are configured for high throughput and therefore have a low initial memory mapping efficiency roughly around 50%. We also note that the added constraint of intra-layer mapping does not significantly degrade the achievable efficiency - in most cases the intra-layer efficiency is within 5% of the inter-layer efficiency.

7 DISCUSSION AND FUTURE WORK

The memory packing methodology presented in this work enables increased memory resource utilization efficiency in modern FPGAs. Our approach is general, i.e. can be utilized for any digital circuit utilizing large parameter memories that are read in predictable fashion at run-time, and is fast compared to the published state-of-the-art. In the specific context of dataflow NN inference accelerators, where memory resource availability is often a design bottleneck, our technique can enable a specific accelerator design to target smaller FPGA devices by becoming more efficient in its OCM usage. The rapid convergence of the NFD-based algorithms to a final packing solution enables their use within CNN accelerator design space exploration frameworks.

Beyond the CNN acceleration applications presented in this paper, we believe the algorithms presented have a general applicability to FPGA design optimization. Advances can be made by performing multi-objective optimization that takes throughput, memory and LUT utilization of the design into consideration during the evolution process. Thus, evolutionary optimization heuristics can serve to merge many traditional aspects of FPGA electronic design automation which are typically solved in isolation.

Table 4: Mapping Efficiency Increase (GA-NFD)

Accelerator	BRAM	Efficiency	$\Delta_{\mathbf{BRAM}}$
CNV-W1A1	120	69.3%	
CNV-W1A1-Intra	100	82.3%	1.20×
CNV-W1A1-Inter	96	86.6%	1.25×
CNV-W2A2	208	79.9%	
CNV-W2A2-Intra	192	86.6%	1.08×
CNV-W2A2-Inter	188	88.4%	1.11×
Tincy-YOLO	578	63.6%	
Tincy-YOLO-Intra	456	80.7%	1.27×
Tincy-YOLO-Inter	420	87.6%	1.38×
DoReFaNet	4116	78.8%	
DoReFaNet-Intra	3797	85.4%	1.08×
DoReFaNet-Inter	3794	85.5%	1.08×
ReBNet	2880	64.1%	
ReBNet-Intra	2363	78.1%	1.22×
ReBNet-Inter	2352	78.4%	1.22×
RN50-W1A2	2064	57.9%	
RN50-W1A2-Intra	1440	82.9%	1.43×
RN50-W1A2-Inter	1374	86.9%	1.50×
RN101-W1A2	4240	52.4%	
RN101-W1A2-Intra	2748	80.9%	1.54×
RN101-W1A2-Inter	2616	84.9%	1.62×
RN152-W1A2	5904	50.9%	
RN152-W1A2-Intra	3758	80.0%	1.57×
RN152-W1A2-Inter	3584	83.9%	1.65×

REFERENCES

[1] Michaela Blott, Thomas B. Preusser, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'Brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. 2018. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst.* 11, 3, Article 16 (Dec. 2018), 23 pages. https://doi.org/10.1145/3242897

[2] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.* 5918–5926.

[3] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, et al. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).

[4] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv preprint arXiv:1602.02830* (2016).

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.

[6] Emanuel Falkenauer. 1996. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics* 2, 1 (01 Jun 1996), 5–30. https://doi.org/10.1007/BF00226291

[7] E. Falkenauer and A. Delchambre. 1992. A genetic algorithm for bin packing and line balancing. In *Proceedings 1992 IEEE International Conference on Robotics and Automation*. 1186–1192 vol.2. https://doi.org/10.1109/ROBOT.1992.220088

[8] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research* 13, Jul (2012), 2171–2175.

[9] Mohammad Ghasemzadeh, Mohammad Samragh, and Farinaz Koushanfar. 2018. ReBNet: Residual binarized neural network. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 57–64.

[10] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv preprint arXiv:1512.03385* (2015).

[12] Hans Kellerer and Ulrich Pferschy. 1999. Cardinality constrained bin-packing problems. *Annals of Operations Research - Annals OR* 92 (01 1999), 335–348. https://doi.org/10.1023/A:1018947117526

[13] Kirkpatrick, S. and Gelatt, C. and Vecchi, M. 1983. Optimization by Simulated Annealing. *Science (New York, N.Y.)* 220 (06 1983), 671–80. https://doi.org/10.1126/science.220.4598.671

[14] K. L. Krause, V. Y. Shen, and H. D. Schwetman. 1973. A Task-scheduling Algorithm for a Multiprogramming Computer System. In *Proceedings of the Fourth ACM Symposium on Operating System Principles (SOSP '73)*. ACM, New York, NY, USA, 112–118. https://doi.org/10.1145/800009.808058

[15] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. *online: http://www. cs. toronto. edu/kriz/cifar. html* 55 (2014).

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

[17] Fengfu Li, Bo Zhang, and Bin Liu. 2016. Ternary weight networks. *arXiv preprint arXiv:1605.04711* (2016).

[18] Asit Mishra and Debbie Marr. 2017. Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy. *arXiv preprint arXiv:1711.05852* (2017).

[19] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. 2016. Design space exploration of FPGA-based Deep Convolutional Neural Networks. *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)* (2016), 575–580.

[20] Marcela Quiroz-Castellanos, Laura Cruz-Reyes, Jose Torres-Jimenez, Claudia Gómez S., Héctor J. Fraire Huacuja, and Adriana C.F. Alvim. 2015. A Grouping Genetic Algorithm with Controlled Gene Transmission for the Bin Packing Problem. *Comput. Oper. Res.* 55, C (March 2015), 52–64. https://doi.org/10.1016/j.cor.2014.10.010

[21] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 779–788.

[22] E. Reggiani, M. Rabozzi, A. M. Nestorov, A. Scolari, L. Stornaiuolo, and M. Santambrogio. 2019. Pareto Optimal Design Space Exploration for Accelerated CNN on FPGA. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 107–114. https://doi.org/10.1109/IPDPSW.2019.00028

[23] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 65–74. https://doi.org/10.1145/3020078.3021744

[24] J. Vasiljevic and P. Chow. 2014. Using buffer-to-BRAM mapping approaches to trade-off throughput vs. memory use. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. 1–8. https://doi.org/10.1109/FPL.2014.6927469

[25] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).

[26] Bohan Zhuang, Chunhua Shen, Mingkui Tan, Lingqiao Liu, and Ian Reid. 2018. Towards effective low-bitwidth convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 7920–7928.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. issn: 0001-0782. doi: 10.1145/3065386. url: https://doi.org/10.1145/3065386.

[2] K. He et. al. “Deep Residual Learning for Image Recognition”. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90.

[3] R. Raina, A. Madhavan, and A. Y. Ng. “Large-Scale Deep Unsupervised Learning Using Graphics Processors”. In: *Proceedings of the 26th Annual International Conference on Machine Learning*. ICML ’09. Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 873–880. isbn: 9781605585161. doi: 10.1145/1553374.1553486. url: https://doi.org/10.1145/1553374.1553486.

[4] S. Chetlur et. al. “cuDNN: Efficient Primitives for Deep Learning”. In: (Oct. 2014).

[5] M. Abadi et. al. “TensorFlow: A System for Large-scale Machine Learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. OSDI ’16. 2016, pp. 265–283. url: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[6] J. Misra and I. Saha. “Artificial Neural Networks in Hardware: A survey of Two Decades of Progress”. In: *Neurocomputing* 74.1 (2010). Artificial Brains, pp. 239–255. issn: 0925-2312. doi: https://doi.org/10.1016/j.neucom.2010.03.021. url: http://www.sciencedirect.com/science/article/pii/S092523121000216X.

[7] C. Zhang et. al. “Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks”. In: Feb. 2015, pp. 161–170. doi: 10.1145/2684746.2689060.

[8] S. Shin W. Sung and K. Hwang. “Resiliency of Deep Neural Networks under Quantization”. In: *CoRR* abs/1511.06488 (2015). arXiv: 1511.06488. url: http://arxiv.org/abs/1511.06488.

[9] S. Biookaghazadeh, M. Zhao, and F. Ren. “Are FPGAs Suitable for Edge Computing?” In: *USENIX Workshop on Hot Topics in Edge Computing (HotEdge 18)*. Boston, MA: USENIX Association, July 2018. url: https://www.usenix.org/conference/hotedge18/presentation/biookaghazadeh.

[10] Y. Umuroglu et. al. “FINN: A Framework for Fast, Scalable Binarized Neural Network Inference”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’17. Monterey, California, USA: ACM, 2017, pp. 65–74. isbn: 978-1-4503-4354-1. doi: 10.1145/3020078.3021744. url: http://doi.acm.org/10.1145/3020078.3021744.

[11] M. Blott et. al. “FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks”. In: *ACM Trans. Reconfigurable Technol. Syst.* 11.3 (Dec. 2018), 16:1–16:23. issn: 1936-7406. doi: 10.1145/3242897. url: http://doi.acm.org/10.1145/3242897.

[12] D. Karchmer and J. Rose. “Definition and Solution of the Memory Packing Problem for Field-Programmable Systems”. In: *Proceedings of the 1994 IEEE/ACM International Conference on Computer-Aided Design*. ICCAD ’94. San Jose, California, USA: IEEE Computer Society Press, 1994, pp. 20–26. isbn: 0897916905.

[13] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1979. isbn: 0716710447.

[14] P. H. W. Leong. “Recent Trends in FPGA Architectures and Applications”. In: *4th IEEE International Symposium on Electronic Design, Test and Applications (delta 2008)*. Jan. 2008, pp. 137–141. doi: 10.1109/DELTA.2008.14.

[15] T. N. Theis and H. -. P. Wong. “The End of Moore’s Law: A New Beginning for Information Technology”. In: *Computing in Science Engineering* 19.2 (Mar. 2017), pp. 41–50. issn: 1558-366X. doi: 10.1109/MCSE.2017.29.

[16] C. Kyrkou. *What Are FPGAs?* Ed. by Medium. Mar. 2017. url: https://medium.com/@ckyrkou.

[17] *7 Series FPGAs Configurable Logic Block*. UG474. v1.8. Xilinx. Sept. 2016.

[18] *7 Series FPGAs Memory Resources User Guide*. UG473. v1.14. Xilinx. June 2019.

[19] *UltraScale Architecture Memory Resources User Guide*. UG573. v1.10. Xilinx. Feb. 2019.

[20] C. Toole. *AMBA AXI and ACE Protocol Specification*. Ed. by Arm. Oct. 2019. url: https://static.docs.arm.com/ihi0022/g/IHI0022G_amba_axi_protocol_spec.pdf.

[21] *AXI Reference Guide*. UG761. v13.1. Xilinx. Mar. 2011.

[22] R. Radojcic. “More-than-Moore Technology Opportunities: 2.5D SiP”. In: *More-than-Moore 2.5D and 3D SiP Integration*. Cham: Springer International Publishing, 2017, pp. 5–67. isbn: 978-3-319-52548-8. doi: 10.1007/978-3-319-52548-8_2. url: https://doi.org/10.1007/978-3-319-52548-8_2.

[23] M. Walton. *Nvidia Tesla V100: First Volta GPU is one of the largest silicon chips ever*. Ed. by arstechnica. May 2017. url: https://arstechnica.com/gadgets/2017/05/nvidia-tesla-v100-gpu-details/.

[24] *Large FPGA Methodology Guide Including Stacked Silicon Interconnect (SSI) Technology*. UG872. v14.3. Xilinx. Oct. 2012.

[25] C. Ravishankar, H. Fraisse, and D. Gaitonde. “SAT Based Place-And-Route for High-Speed Designs on 2.5D FPGAs”. In: *2018 International Conference on Field-Programmable Technology (FPT)*. Dec. 2018, pp. 118–125. doi: 10.1109/FPT.2018.00027.

[26] N. Voss et al. “Memory Mapping for Multi-die FPGAs”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Apr. 2019, pp. 78–86. doi: 10.1109/FCCM.2019.00021.

[27] *UltraScale Architecture Configurable Logic Block User Guide*. UG574. v1.5. Xilinx. Feb. 2017.

[28] P. Coussy et. al. “An Introduction to High-Level Synthesis”. In: *IEEE Design Test of Computers* 26.4 (July 2009), pp. 8–17. issn: 1558-1918. doi: 10.1109/MDT.2009.69.

[29] *Vivado Design Suite User Guide: High-Level Synthesis*. UG902. v2019.1. Xilinx. July 2019.

[30] Warren Mcculloch and Walter Pitts. “A Logical Calculus of Ideas Immanent in Nervous Activity”. In: *Bulletin of Mathematical Biophysics* 5 (1943), pp. 127–147.

[31] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* (1958), pp. 65–386.

[32] H. Mahmood. *Activation Functions in Neural Networks*. Ed. by Towards Data Science. Dec. 2018. url: https://towardsdatascience.com/activation-functions-in-neural-networks-83ff7f46a6bd.

[33] D. Liu. *A Practical Guide to ReLU*. Ed. by Medium. Nov. 2017. url: https://medium.com/@danqing/a-practical-guide-to-relu-b83ca804f1f7.

[34] I. Salian. *SuperVize Me: What’s the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?* Ed. by NVIDIA. Aug. 2018. url: https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/.

[35] H. Wang, B. Raj, and E. P. Xing. “On the Origin of Deep Learning”. In: *CoRR* abs/1702.07800 (2017). arXiv: 1702.07800. url: http://arxiv.org/abs/1702.07800.

[36] A. Dertat. *Applied Deep Learning - Part 4: Convolutional Neural Networks*. Ed. by Medium. Nov. 2017. url: https://towardsdatascience.com/@ardendertat.

[37] A. Shawahna, S. M. Sait, and A. El-Maleh. “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review”. In: *IEEE Access* 7 (Jan. 2019), pp. 7823–7859. issn: 2169-3536. doi: 10.1109/ACCESS.2018.2890150.

[38] Y. LeCun et. al. “Backpropagation Applied to Handwritten Zip Code Recognition”. In: *Neural Computation* 1.4 (Dec. 1989), pp. 541–551. issn: 0899-7667. doi: 10.1162/neco.1989.1.4.541.

[39] S. Han, H. Mao, and W.J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. arXiv: 1510.00149 [cs.CV].

[40] M. Courbariaux and Y. Bengio. “BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1”. In: *CoRR* abs/1602.02830 (2016). arXiv: 1602.02830. url: http://arxiv.org/abs/1602.02830.

[41] C. Zhang et. al. “Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural networks”. In: *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. Nov. 2016, pp. 1–8. doi: 10.1145/2966986.2967011.

[42] X. Lin et. al. “LCP: A Layer Clusters Paralleling Mapping Method for Accelerating Inception and Residual Networks on FPGA”. In: *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. June 2018, pp. 1–6. doi: 10.1109/DAC.2018.8465777.

[43] H. Sharma et. al. “From High-Level Deep Neural Models to FPGAs”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Oct. 2016, pp. 1–12. doi: 10.1109/MICRO.2016.7783720.

[44] Xilinx. *BNN-PYNQ*. https://github.com/Xilinx/BNN-PYNQ. 2020.

[45] A. Krizhevsky, V. Nair, and G. Hinton. *The CIFAR-10 Dataset*. 2014. url: http://www.cs.toronto.edu/kriz/cifar.html.

[46] Y. Netzer et. al. “Reading Digits in Natural Images with Unsupervised Feature Learning”. In: *NIPS* (Jan. 2011).

[47] *FIFO Generator v13.1 LogiCORE IP Product Guide*. PG057. v13.1. Xilinx. Apr. 2017.

[48] L. Epstein. “Bin Packing Games with Selfish Items”. In: *Mathematical Foundations of Computer Science 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 8–21. isbn: 978-3-642-40313-2.

[49] K. L. Krause, V. Y. Shen, and H. D. Schwetman. “Analysis of Several Task-Scheduling Algorithms for a Model of Multiprogramming Computer Systems”. In: *J. ACM* 22.4 (Oct. 1975), pp. 522–550. issn: 0004-5411. doi: `10.1145/321906.321917`. url: `https://doi.org/10.1145/321906.321917`.

[50] H. Kellerer and U. Pferschy. “Cardinality Constrained Bin-Packing Problems”. In: *Annals of Operations Research* 92.0 (Jan. 1999), pp. 335–348. issn: 1572-9338. doi: `10.1023/A:1018947117526`. url: `https://doi.org/10.1023/A:1018947117526`.

[51] Gyorgy Dosa and Leah Epstein. *Online Bin Packing with Cardinality Constraints Revisited*. 2014. arXiv: `1404.1056 [cs.DS]`.

[52] A. van Vliet. “On the Asymptotic Worst Case Behavior of Harmonic Fit”. In: *Journal of Algorithms* 20.1 (1996), pp. 113–136. issn: 0196-6774. doi: `https://doi.org/10.1006/jagm.1996.0005`. url: `http://www.sciencedirect.com/science/article/pii/S019667749690005X`.

[53] P. Ongkunaruk. “Asymptotic Worst-Case Analyses for the Open Bin Packing Problem”. PhD thesis. Virginia Tech, Blacksburg, VA, USA, 2005. url: `http://hdl.handle.net/10919/30105`.

[54] M. Gen and R. Cheng. *Genetic Algorithms*. 1st. USA: John Wiley & Sons, Inc., 1999. isbn: 0471315311.

[55] J. Vasiljevic and P. Chow. “Using Buffer-to-BRAM Mapping Approaches to Trade-off Throughput vs. Memory Use”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8. doi: `10.1109/FPL.2014.6927469`.

[56] S. Kirkpatrick, C. Gelatt, and M. Vecchi. “Optimization by Simulated Annealing”. In: *Science (New York, N.Y.)* 220 (June 1983), pp. 671–80. doi: `10.1126/science.220.4598.671`.

[57] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Cambridge, MA, USA: MIT Press, 1992. isbn: 0262082136.

[58] E. Falkenauer and A. Delchambre. “A Genetic Algorithm for Bin Packing and Line Balancing”. In: *Proceedings 1992 IEEE International Conference on Robotics and Automation*. May 1992, 1186–1192 vol.2. doi: `10.1109/ROBOT.1992.220088`.

[59] E. Falkenauer. “A Hybrid Grouping Genetic Algorithm for Bin Packing”. In: *Journal of Heuristics* 2 (1996), pp. 5–30.

[60] M. Quiroz-Castellanos et. al. “A Grouping Genetic Algorithm with Controlled Gene Transmission for the Bin Packing Problem”. In: *Computers & Operations Research* 55 (2015), pp. 52–64. issn: 0305-0548. doi: `https://doi.org/10.1016/j.cor.2014.10.010`. url: `http://www.sciencedirect.com/science/article/pii/S0305054814002676`.

[61] F. Li, B. Zhang, and B. Liu. “Ternary Weight Networks”. In: *arXiv preprint arXiv:1605.04711* (2016).

[62] S. Zhou et. al. “DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients”. In: *arXiv preprint arXiv:1606.06160* (2016).

[63] M. Ghasemzadeh, M. Samragh, and F. Koushanfar. “ReBNet: Residual Binarized Neural Network”. In: *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2018, pp. 57–64.

[64] J. Redmon et. al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*. 2016, pp. 779–788.

[65] F. Fortin et. al. “DEAP: Evolutionary Algorithms Made Easy”. In: *Journal of Machine Learning Research* 13.Jul (2012), pp. 2171–2175.

[66] Xilinx. *ResNet50-PYNQ*. `https://github.com/Xilinx/ResNet50-PYNQ`. 2020.

[67] E. Reggiani et al. “Pareto Optimal Design Space Exploration for Accelerated CNN on FPGA”. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2019, pp. 107–114. doi: `10.1109/IPDPSW.2019.00028`.

[68] E. Zitzler, M. Laumanns, and L. Thiele. *SPEA2: Improving the Strength Pareto Evolutionary Algorithm*. Tech. rep. 2001.

[69] K. Deb et al. “A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II”. In: *IEEE Transactions on Evolutionary Computation* 6.2 (Apr. 2002), pp. 182–197. issn: 1941-0026. doi: `10.1109/4235.996017`.

[70] K. Deb and H. Jain. “An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints”. In: *IEEE Transactions on Evolutionary Computation* 18.4 (Aug. 2014), pp. 577–601. issn: 1941-0026. doi: `10.1109/TEVC.2013.2281535`.