Implementing Convolutional Neural Networks in a Cluster of Interconnected FPGAs Using Vivado HLS

Evangelos Mageiropoulos

Thesis submitted in partial fulfillment of the requirements for the

Master of Science degree in Computer Science and Engineering

University of Crete
School of Sciences and Engineering
Computer Science Department
Voutes University Campus, 700 13 Heraklion, Crete, Greece

Thesis Advisor: Prof. *Manolis Katevenis*

Thesis Co-Advisor: Dr. *Nikolaos Chrysos*

This work has been performed at and supported by the Computer Architecture and VLSI Systems (CARV) Laboratory, Institute of Computer Science (ICS), Foundation for Research and Technology - Hellas (FORTH).

Abstract

Convolutional Neural Networks (CNNs) are extensively used to augment our everyday experience of the world by automatically labeling and categorizing digital data, such as images, voice records, and video, thus helping in web search and in comprehension of data available in the digital world. In this thesis, we explore the possibility to map complete CNNs in clusters of multiple interconnected computing devices, such as ASICs or FPGAs. We seek to define a scalable architecture where a cluster of FPGAs (a segment of the ExaNeSt-based HPC prototype) works concurrently on user streams of inference requests. We base our work on existing tools that simplify the mapping of arbitrary networks, such as using Keras to define the Convolutional Neural Network and hls4ml, a tool developed at CERN, that implements a convolutional neural network into RTL using the Vivado High Level Synthesis (HLS) framework. We introduce a number of code and directive-based optimizations in order to achieve speedups in excess of 700x for the individual kernels, and manage to map all parameters inside FPGA BRAMs. Furthermore, we split and redesign the network in order to minimize the data transfers and balance the work across FPGAs. Finally, we design custom RTL blocks which we integrate with HLS directives in order to use an HPC network for inter-FPGA communication. Our final implementation of the SqueezeNet CNN network which needs 800 million operations per inference task, in 5 FPGAs, offers a throughput of 303 image classifications per second (CPS), and a total inference latency of 24 ms, one order of magnitude smaller than typical user Service Level Aggreements (SLAs).

Υλοποίηση Συνελικτικών Νευρωνικών Δικτύων σε μία Δομή Διασυνδεδεμένων FPGA με τη Χρήση Vivado HLS

Περίληψη

Τα Συνελικτικά Νευρωνικά Δίκτυα χρησιμοποιούνται ευρέως για να βελτιώσουν την καθημερινή μας εμπειρία με τον κόσμο, κατηγοριοποιώντας αυτόματα ψηφιακά δεδομένα, όπως εικόνες, ηχητικές καταγραφές και βίντεο, βοηθώντας έτσι στις διαδικτυακές αναζητήσεις και την κατανόηση των δεδομένων που είναι διαθέσιμα στον ψηφιακό κόσμο. Σε αυτή την εργασία, εξερευνούμε τη δυνατότητα να κατανείμουμε ολόκληρα Συνελικτικά Νευρωνικά Δίκτυα σε ομάδες πολλαπλών διασυνδεδεμένων συστημάτων υπολογισμού, όπως ASICs και FPGAs. Επιδιώκουμε να ορίσουμε μια κλιμακούμενη αρχιτεκτονική όπου μια ομάδα από FPGAs (τμήμα του προτύπου HPC που βασίζεται στο ExaNeSt) δουλεύει ταυτόχρονα σε ροές αιτημάτων χρηστών για κατηγοριοποιήσεις. Βασίζουμε την εργασία μας σε προϋπάρχοντα εργαλεία που απλοποιούν την κατανομή διαφόρων δικτύων, όπως το Keras για να ορίσουμε το Συνελικτικό Νευρωνικό Δίκτυο και το hls4ml, ένα εργαλείο που έχει αναπτυχθεί στο CERN, που υλοποιεί ένα νευρωνικό δίκτυο σε RTL χρησιμοποιώντας τη δομή λογισμικού Vivado High Level Synthesis (HLS). Συστήνουμε ένα σύνολο από βελτιστοποιήσεις στον κώδικα και βασιζόμενες σε οδηγίες, ώστε να πετύχουμε επιτάχυνση άνω του 700x σε αυτούσιους υπολογιστικούς πυρήνες, και καταφέρνουμε να κατανείμουμε όλες τις παραμέτρους σε BRAMs των FPGA. Επιπλέον, διαχωρίζουμε και επανασχεδιάζουμε το δίκτυο, ώστε να ελαχιστοποιήσουμε τις μεταφορές δεδομένων και εξισορροπούμε την εργασία στο σύνολο των FPGAs. Εν τέλει, σχεδιάζουμε ιδιόχειρα κομμάτια RTL, στα οποία ενσωματώνουμε οδηγίες του HLS, ώστε να χρησιμοποιήσουμε ένα δίκτυο HPC για επικοινωνία μεταξύ των FPGAs. Η τελική μας υλοποίηση του Συνελικτικού Νευρωνικού Δικτύου SqueezeNet που απαιτεί 800 εκατομμύρια πράξεις ανά εργασία κατηγοριοποίησης, σε 5 FPGAs, προσφέρει διεκπεραιωτική ικανότητα 303 κατηγοριοποιήσεων εικόνων ανά δευτερόλεπτο και συνολική καθυστέρηση κατηγοριοποίησης 24 χιλιοστά του δευτερολέπτου, μια τάξη μεγέθους μικρότερη από τυπικές Συμφωνίες Επιπέδου Υπηρεσίας.

University of Crete
Computer Science Department

Implementing Convolutional Neural Networks in a Cluster of Interconnected FPGAs Using Vivado HLS

Thesis submitted by
Evangelos Mageiropoulos
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author:

Evangelos Mageiropoulos

Digitally signed by Evangelos Mageiropoulos
Date: 2020.11.11 14:08:53 +02'00'

Evangelos Mageiropoulos

Committee approvals:

EMMANOUIL KATEVENIS

Digitally signed by EMMANOUIL KATEVENIS
Date: 2020.11.11 14:49:55 +02'00'

Prof. Manolis Katevenis
Professor, Thesis Supervisor

Digitally signed by Apostolos Dollas
Date: 2020.11.11 15:11:21 +02'00'

Prof. Apostolos Dollas
Professor, Committee Member

Polyvios Pratikakis

Digitally signed by Polyvios Pratikakis
Date: 2020.11.11 13:18:32 +02'00'

Prof. Polyvios Pratikakis
Assistant Professor, Committee Member

Departmental approval:

Polyvios Pratikakis

Digitally signed by Polyvios Pratikakis
Date: 2020.11.11 13:19:03 +02'00'

Prof. Polyvios Pratikakis
Assistant Professor, Director of Graduate Studies

Heraklion, October 2020

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, the scientific field of artificial intelligence has experienced large-scale improvements, thanks to the wide adoption of convolutional neural networks. Pioneered by Lecun et al in 1990 [2], with the foundation laid one decade earlier [3], convolutional neural networks initially did not see widespread adoption, since the hardware of that era was too limited to match their memory and processing power requirements and large datasets for training purposes were not available. After the major breakthrough of AlexNet [4] in 2012 however, where a GPU-accelerated approach achieved a top-5 error rate of 17% in the ImageNet LSVRC-2010 contest, the field of convolutional neural networks has become a major part of big data analysis and image classification, with large-scale implementation efforts from Google [5], Microsoft [6], etc. Nowadays, CNNs are used to augment our everyday experience of the world by automatically labeling and categorizing digital data, such as images, voice records, and video thus helping in web search, and machine-human comprehension of the digital world.

Convolutional neural networks consist of a set of interconnected computational layers, each of which manipulates its input with a set of predetermined parameters. Most convolutional neural networks consist of convolutional, pooling and activation layers, together with an input and output layer. Their architecture excels in the field of image classification, where they can provide high quality categorization results, while skipping the need for domain knowledge, by automatically extracting features from pre-classified data sets and using these features to classify new inputs. However, in order to do so, CNNs rely on high-capacity computational units, spending over one billion computing operations per classification task. On the other hand, the control-flow of the programs implementing the CNNs is quite simple and allows for parallelism opportunities. As such, CNNs are very good candidates for acceleration in custom (non General-Purpose) hardware, such as GPUs and FPGAs. We have witnessed a wealth of scientific research [7] that demonstrates methods to implement CNN computation kernels on custom hardware, resulting in designs much faster and/or more efficient than general-purpose

CPU approaches. This thesis studies the optimized implementation of convolutional neural networks (CNNs) in clusters of FPGAs.

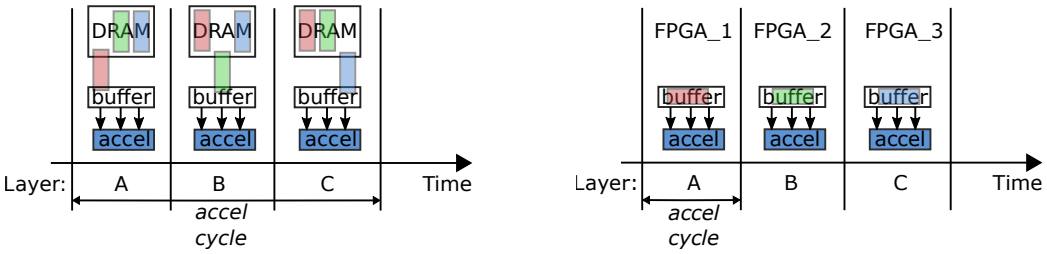
1.1 Accelerating CNNs using special hardware

In the past years, general purpose graphics processors have been used for this purpose. GPUs indeed can provide high-throughput for workloads similar to that of CNNs. They typically consist of multiple compute clusters, each capable of performing the same instruction on wide sets of input and output operands, in the form of single instruction multiple data (SIMD). Modest GPUs today achieve thousands of billion floating point instructions per second, i.e. TFLOPS. Moreover, their architecture has been adapted to better support neural network workloads, by increasing their off-chip memory bandwidth or by incorporating special units, such as tensor units [8]. However, GPUs are power-hungry devices that are not fully tailored for CNNs workloads, thus leaving open the question whether other accelerator units can do perform better than GPUs.

Recent efforts from the industry and the academia have started examining FPGAs for CNN acceleration. FPGAs consist of re-configurable blocks, logic, interconnect, block RAM, and I/O. Functionally, FPGAs contain many thousands of units that can be interconnected to operate in parallel on multiple data that reside on local distributed memories; thus, FPGAs can provide the same parallelism for spatial computation as GPUs [9]. However, FPGAs are not general-purpose, and do not rely on software instructions in order to compute. Their advantage over GPUs is their lower power consumption and that one can fully customize their data path, implementing the intended algorithm in hardware speed. Additionally, FPGAs can implement calculations with arbitrary data sizes, allowing for fine-tuned and optimal resource utilization for any given scenario in that regard and can implement highly-parallel hardware when applicable. However, compared to GPUs, their development cycle is much longer due to the nature of hardware programming and virtually no memory abstraction and are not suitable for processing sparse data algorithms, while floating-point operations consume much of the FPGA real estate.

1.1.1 Time-shared versus per-layer hardware units

The dominant implementation of CNN accelerators on FPGAs relies on an generic hardware unit that is *time shared among the different layers of the CNN*; in this paradigm, each layer typically fetches the parameters (weights and biases) from DRAM, in order to compute its output [10], as shown in Figure 1.1a(a). This resembles the processing steps also followed by many GPUs that the weights in DRAM and re-use the same single-instruction-multiple-data engines for different layers of the the CNN. In this approach, the execution of one layer is split into one fetch-from-DRAM phase and one computation phase. The bottleneck of on-chip memories is further escalated, as efficient hardware accelerator units typically



(a) Implementation in a single FPGA (or GPU) with parameters kept in DRAM

(b) Implementation of a CNN in multiple FPGAs with parameters kept in BRAMs

Figure 1.1: A comparison of an implementation that store the parameters in global DRAM with one that stores them on-chip (e.g. in FPGA BRAMs.) The figure on the left re-uses the same hardware for every layer, and stores the per-layer parameters (weights) in DRAMs. The figure on the right, uses different hardware for every layer; each layer thus can have its parameters stored close to the computational units, e.g. in FPGA BRAMs. The latter solution allows to pipeline inference requests, through the hardware units.

need to *partition their data over multiple SRAM blocks or registers* in order to allow multiple hardware units to operate on them in parallel (see loop unrolling). Thus, even if not memory-capacity bounded, many FPGA designs are often limited by the number of memory blocks that are available inside the FPGA (memory-bandwidth limited). Effectively, today, most single-FPGA accelerators for CNNs are forced to work on a subset of the weights at a time, thus also needing to fetch the parameters/weights from DRAM. These solutions typically seek to minimize the size of weights as well as to overlap the DRAM transfer time with computation. In addition, they use batching, i.e. process multiple images on the same layer before moving to the next layer, in order to amortize the cost of memory fetch operations.

Reusing a hardware engine (e.g. FPGA) to compute the output of multiple layers of a CNN is a valid assumption made in many designs [11]. Although these designs typically use one accelerator per inference engine, it is still possible to scale-out the infrastructure by using multiple instances of the same accelerator (e.g. FPGA or GPU) working on different inputs (inference tasks).

On the other hand, a CNN inference task can be promptly mapped to a *feed-forward (data-flow-like) compound hardware engine, with distinct hardware units, possibly spanning multiple FPGAs, responsible for different layers of the network*, as shown in Figure 1.1a(b). In this approach, one can *fully pipeline* the available hardware, serving inference requests with higher throughput, without the need to perform extensive batching, which induces delays and is not always possible, as in the case of video processing. As each stage of this hardware engine is responsible for one layer, one can also in principle *keep the corresponding weights inside the distributed on-chip memories next to their processing units*[1]. Additionally, per-layer

[1]Even a single FPGA can in principle implement a full CNN network with distinct units for

partitioning opens up the possibility to *use alternative implementations and/or accelerator technologies* (CPU, FPGA-kernels, GPUs) for different layers, i.e. a non-uniform, highly-optimized acceleration environment.

1.2 Motivation for Multi-FPGA CNNs

In this thesis, we seek to define and demonstrate an architecture where a cluster of FPGAs work concurrently on user-streams of inference requests, in which one can *scale-out the computation capacity* by using more hardware resources, i.e. more nodes of the FPGA cluster. In particular, we explore the possibility to *map the functions/layers of a CNN onto a set of discrete hardware resources that work in tandem, obtaining a data-flow compound engine for every CNN, with simple control-flow (left to right).* In this thesis, these hardware units of the CNN will be distributed across a group of nodes in a cluster of FPGAs, but in principle, they can also be located inside a big ASIC. This design option might seem "expensive" ("throw hardware to the problem"), but has the following potential benefits:

- **Scalability:** Compared to a single-FPGA implementation, a multi-FPGA environment allows for a *scale-out design of a CNN* with processing capacity (million-operations-per-second) that is not limited by the resources of a single FPGA.

- **Pipelining and computation-overlap**: In our approach, every hardware unit is responsible for one or more neighbor layers of the CNN, and outputs a vector, in batches of cells, expected by the next layers. After finishing one task, a hardware unit can begin processing the next request/image, thus improving throughput (*pipelining*). Furthermore, the *computation of neighboring layers working on the same request/image may overlap*, if the next layer can process the output batches of the former one, as they become available. Such computation-computation overlap (also known as *dataflow*) reduces the inference latency and further improves the throughput.

- **Weights in on-chip buffers:** One byproduct of this solution, is that we can obtain designs with all weights kept in local BRAMs, thus minimizing the power needed to bring the weights inside the computing device. We still need to communicate the output feature maps between adjacent stages when these stages reside crossing devices, but this is much cheaper (2-3 orders of magnitudes less size). This concept is demonstrated in figure 1.1b, where the *accel cycle* only consists of the data processing stage, without the need for parameter fetch. An efficient enough communication implementation will allow for each kernel to start processing before the data transmission completes.

every layer; however, typically, a single FPGA cannot hold the parameters of a full network and does not have enough resources to efficiently implement all of its layers.

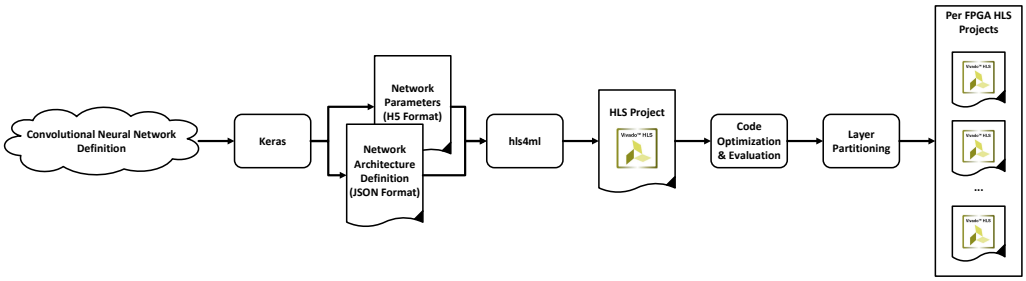


Figure 1.2: The workflow of our thesis

- **Good fit with available platform:** We envision that this solution will fit well to the ExaNeSt QFDB prototype [12], which consists of several tens of FPGAs, interconnected with high-speed network links.

- **Optimize computation to the extreme:** In a multi-stage physical design, the memory resources per FPGA is reduced, thus making it possible to use more parallelism per stage.

- **No under-utilization of resources:** One can tailor different designs for different (groups of) layers, thus manage resource utilization in a fine-grain manner, and potentially further improving the design efficiency in terms of performance and energy consumption. In this perspective, our work may serve as a demonstrator for future efforts to map CNNs in heterogeneous platforms.

- Biological neural networks, which inspired the design of CNNs and DNNs in the first place, appear to follow this paradigm: separate layers of (physical) neurons process every image in an ordered fashion, with each layer responsible to extract a different set of features, such as lines, edges, orientation, shapes, human faces, etc. [13]

1.3 Thesis Workflow

Our work *relies and builds upon existing frameworks in order to simplify the current and future efforts to port CNNs in FPGA clusters.* After examining a number of available tools, such as DnnWeaver and CHaiDNN, we decided to use Keras and hls4ml. The former allows for defining and training arbitrary convolutional neural network and the latter converts it into a Vivado HLS project in C++ for implementation into a single FPGA. With this combination of tools, one can transform any arbitrary CNN into a generic HLS program that can be used to program any FPGA supporting HLS for image classification, without assuming any specific FPGA model or platform.

Figure 1.2 demonstrates the steps of the workflow followed by this thesis. We applied this workflow on *SqueezeNet*, a convolutional neural network with a straightforward and repeatable architecture and a size that can easily be managed. We first used Keras and hls4ml to automatically create a first HLS project for SqueezeNet. We then rearranged the architecture of the network building blocks (the *Fire modules*) in order to minimize their output size, assuming that *each building block will reside entirely in a single FPGA*. A significant segment of our work has been to *improve the resulting HLS code* by evaluating its efficiency and bottlenecks, and applying a series of code modifications and HLS directives, on a per-layer and per-block (*Fire module*) basis, in order to improve their execution speed when implemented in Ultrascale+ FPGAs. We finally partitioned the network into sets of building blocks, with each set assigned to a different FPGA, by taking into consideration the FPGA real estate required for each set, while striving to equally load-balance the work across FPGAs and minimize their communication.

In this regard, this thesis shows how to:

1. efficiently split the output of hls4ml into multiple FPGAs,

2. optimize the output of the hls4ml by modifying the HLS code in order to gain significant speedups,

3. starting from the output of automatic tools, come up with highly-optimized HLS designs for convolutional and pooling layers, suitable for implementation in multiple FPGAs,

4. to tailor the hls4ml designs for the ExaNeSt testbed.

The output of this workflow, i.e. the network of FPGAs that implements SqueezeNet or any other network in future work, can be viewed in an abstract form in figure 1.3. Here, each FPGA implements a predefined set of network layers or building blocks, together with queues and data transfer engines that handle its input and output. Flow control logic at the input handles rate mismatches between adjacent stages, using a FIFO buffer to store incoming tasks (from the previous layer) before the FPGA logic is ready to process them. As discussed next, the system can be controlled by the user through a server interface which can queue tasks to the input of the first FPGA and process them in the classification pipeline.

1.4 Demonstrating a high-throughput inference appliance

Inference may be used to label images while the user navigates in the web, in augmented reality environments, in self-driving cars, i.e., wherever systems are asked to automatically and promptly label (segments/figures in) images or video.

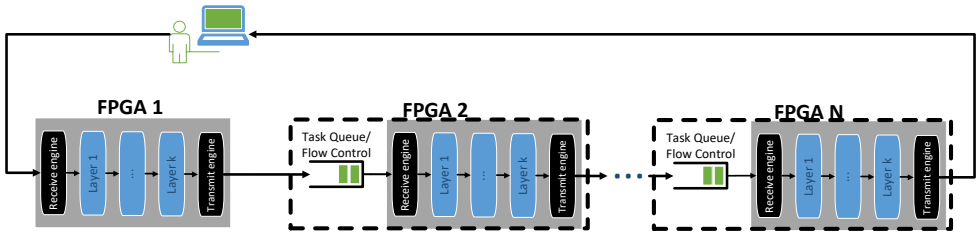


Figure 1.3: Proposed CNN partitioning in multiple FPGAs

In order to guide this thesis, we will adopt the *requirements of an inference appliance for the web*. Inference tasks in the web and in commercial data centers are *typically user-facing, requiring low latency (below 250ms)*, to keep the end-user satisfied, and *high throughput* to serve many users in parallel [14] .

The scheme proposed in this thesis can be used to build a machine that serves such user-facing inference tasks, as shown in figure 1.4. As shown in the figure, we need an input queue to accept the user-initiated tasks and push them into the inference engine pipeline when it becomes available[2].

The *total latency of a task request* is equal to the *queuing time* (inside the queue) plus the latency (service time) of a request inside the inference . Our approach can be used to establish a **Service Level Agreement (SLA)**, since the pipeline has a strictly bounded total latency, equal to the sum of the latencies of its stages. This primitive, together with an efficient implementation of the server-side buffer queue, allows for a maximum per-task latency to be defined, under predetermined workload scenarios, i.e. when the queue backlog is zero or bounded by some upper limit. The multi-FPGA pipelined implementation, provides higher processing capacity, thus in principle allows to meet the SLA under a wider range of load conditions.

Performance target: To become meaningful in this environment, the inference engine that we build must *feature an end-to-end latency L least one order of magnitude smaller than the typical SLA 250ms, (below 30 ms)*. On the other hand, *throughput should scale generously with the HW units deployed*, well exceeding of course the reciprocal of latency, 1 task per L time units, thanks to the pipelining effect.

It is worth noting that these performance targets steered a significant piece of our workflow while we implemented SqueezeNet for a cluster of FPGAs in chapters 4, 5 and 6. In particular, it affected the split of the network and the optimization of the HLS kernels in synthesized firmware for the FPGAs, providing answers

[2]Note that the queueing functionality can be implemented at the server, thus offloading this work from the inference engine and alleviating the need to store queued tasks in expensive FPGA hardware.

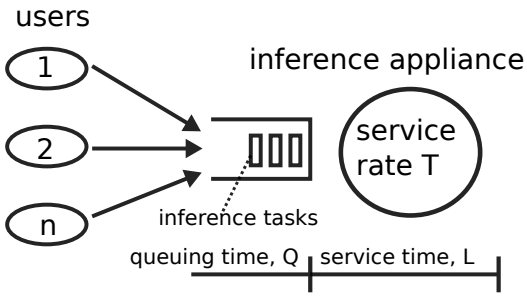


Figure 1.4: An inference appliance that supports user-level inference (web) requests, e.g. in a datacenter.

to questions what optimization are good enough, and which segments must be improved before moving to the next kernel/step in our workflow.
Inference appliance results: In this thesis, we implement all the designs needed for the *SqueezeNet CNN requiring 778M operations per inference task*. The appliance's performance numbers are the listed below:

- total inference latency $L = 24$ms,
- offered throughput is 1 inference task (classification operation) per 3ms.

To achieve these results, we use *five (5) FPGAs*. In our solution, the cluster process nearly $24/3 = 8$ tasks concurrently in its pipeline. As the SqueezeNet network has in total 57 layers, on average, each layer consumes 0.04 ms on average.

1.5 Contributions

Throughout the duration of this thesis, we dealt with a plethora of subjects, ranging from evaluating CNN-to-hardware frameworks, to defining an optimization methodology for CNN kernels on HLS and transferring task data between the FPGAs. The contributions of this thesis are the following:

- We introduce an efficient and flexible method to partition any convolutional neural network into an environment of interconnected FPGAs. We furthermore demonstrate this method by *partitioning the SqueezeNet CNN*. Our work is based on existing tools, *simplifying the path from network definition to HLS*. Our implementation is fully hardware-driven, i.e. no calculations are carried in general-purpose processors, allowing for a more uniform design that also takes full advantage of the parallelism opportunities of the FPGAs.
- Our implementation is tailored for the ExaNeSt multi-FPGA prototype [12], introducing a methodology to utilize its available programmable logic as a

CNN inference engine. Although our design targets the UltraScale+ MPSoC xczu9eg-ffvc900-2-e [15], its modularity and uniformity allows it to be *easily adapted for other FPGAs, e.g. without integrated CPUs.*

- In the path from network definition to optimized HLS, as well as when splitting the work in FPGAs, we strive to **1.** minimize the latency per layer (or meet a user SLA), **2.** minimize the deployment cost (number of FPGAs, **3.** balance the work per stage (FPGA) to avoid rate-mismatch congestion, **4.** minimize the communication across stages, and 5. meet the FPGA resource requirements. Our split of SqueezeNet into five (5) FPGAs achieves a reasonable balance between these objectives: *we minimize the number of FPGAs* (to simplify the flow and the administrative cost of the deployment for the purposes of this thesis), but *do NOT trade performance*: we utilize the best-performing kernel configuration for each layer.

- In our endeavor to successfully partition the network, we redefined the architecture of the *Fire module*, the core block of SqueezeNet that contains three convolutional layers; our definition shifts the layer ordering, so that the module's output size is the minimum amongst its adjacent layers in the network topology.

- We evaluated and optimized the Keras-to-HLS hls4ml framework for our purposes *by modifying its core functions.* In particular, we modified its `conv2d` function that implements convolutional network layers to minimize its memory footprint and allow for straightforward optimizations using Vivado HLS directives. We furthermore introduced our own max and average pooling and softmax functions that use a simpler code base.

- We carried out an extensive sequence of tests to evaluate *the impact each HLS optimization directive (`UNROLL`, `PIPELINE` etc.) had on the core functions of our design*, both in terms of latency and resource utilization. For this, we isolated the most compute-intensive convolutional layer of our network and compared the results of different optimization directives. The primitives of this part of our work also contribute as a *standalone evaluation of the effectiveness of HLS directives.*

- Our tests on the convolutional function of `hls4ml` included implementations with both 16-bit and 8-bit parameter sizes. These experiments thereby offer an insight on how Vivado HLS handles implementation with different data sizes, regarding both latency and resource utilization metrics.

- We *accelerated the convolutional and pooling functions of SqueezeNet* using the appropriate sets of Vivado HLS directives, by identifying parallelism opportunities of the design. We furthermore *implemented task-level pipeline between the network layers bundled in the Fire modules*, which became possible by adding intermediate data manipulation functions and allowed for

top-module data access through AXI-stream interface. This resulted to two discrete implementation strategies with different speedup and resource utilization metrics for each layer set; therefore, each of them can be selected for the hardware implementation of different layer sets, as seen fit.

- In our work, we achieved a *speedup factor ranging from 129x up to 713x* per computational module compared to a baseline approach. In particular, we achieved a pipeline initiation interval of 655436 clock cycles, which, at a clock period of 5 ns, translates to *one classification result per 3.3 ms*.

- In total, we dealt with and optimized the 57 network layers of SqueezeNet, namely: 26 convolutional layers accompanied by ReLU activation layers, 4 pooling layers (3 max pooling and 1 global average pooling) and the output softmax layer. Our design used a total of 7 *Fire modules*, two of which, in addition to 3 convolutional layers, contained a max pooling layer.

- We implemented a *primitive for inter-FPGA network transfers*, including receive block notifications, that leverages the ExaNeSt network interface primitives (HPC interconnect) and combines them with HLS and custom RTL blocks.

- Throughout our endeavor, we dealt with *undocumented issues related to Vivado HLS itself*, particularly regarding unexpected behavior of directives and problems with the synthesis procedure; we subsequently proposed appropriate course of action to overcome them and achieve the expected results.

- Our design allowed for the SqueezeNet parameters to be stored in local BRAMs; therefore there was no issue for DRAM-accelerator data transfers that could cause additional latency to the inference process.

- Although we did not develop a method for overlapping inter-FPGA data transfer time and processing time (i.e. *computation-communication overlap*), this was not crucial in our setup because our results show that the data transfer latency is small compared to the computation time. Nevertheless, we approached this behavior for kernels implemented in the same FPGA using the `DATAFLOW` HLS directive, with which we also achieved *partial computation overlap* between adjacent network layers. The implementation results showed that the total latency of the network is lower than the sum of the individual layer latencies.

- During the course of my thesis, albeit not documented in this report, I modified an Omnet++ simulator to emulate the ExaNeSt QFDB-Mezzanine-Blade topology and playback MPI traces captured from execution of scientific applications (DPSNN, GADGET, LAMMPS, RegCM). I then performed extensive performance evaluation tests to analyze the performance of congestion management scheme under adverse synthetic and MPI-trace based simulations. This work resulted in a paper presented at NOCS 2018 [16].

1.6 Thesis milestones

In this thesis, we have set an ambitious goal, which gave us the incentive to study many interesting problems and encounter fundamental implications. A significant portion of this thesis was spend on theoretical and technical discussions on how to partition a CNN in multiple FPGAs in an efficient and scalable manner, how will FPGAs communicate efficiently using ExaNeSt and HLS primitives, what are the possible applications of the solution, etc. But, of course, we did not have enough time to work equally on everything. In order to actually demonstrate the main principles in a practical output, we allotted the total time we had available for this work into a multitude of steps, which we can summarize with the following **milestones**:

1. *Studying* the architecture of convolutional neural networks under the scope of hardware implementation,
2. *Researching* available resources/frameworks/CNNs to base our work on,
3. *Creating* communication primitives using the ExaNet infrastructure as a proof-of-concept and for further installments of our work,
4. *Evaluating, modifying* and *optimizing* hls4ml code on Vivado HLS per-layer, per-module and for the complete SqueezeNet network – a feat that consumed the bulk of our time, since we encountered multiple issues regarding both the initial hls4ml code and the Vivado HLS tool that demanded our attention.

We note here that, starting this thesis, there was no clear recipe for all choices we had to make, and many of these choices depended on results we did not have from the beginning. Throughout the course of the thesis, we continued questioning some of our initial decisions, e.g., on communication primitives for inter-FPGA communication overlap. We present interesting conclusions and future work items originating from these discussions in Chapter 7.

1.7 Remainder of this thesis

In Chapter 2, we introduce existing scientific research related to the scope of our approach. Additionally, we present the ExaNeSt prototype that we used as a target device for this dissertation, as well as the hls4ml and Vivado HLS tools that we use to generate, synthesize and optimize our design on the ExaNest FPGAs. In Chapter 3, we define basic principles for partitioning a convolutional neural network into multiple FPGAs, present alternative communication strategies, allowing partial or full overlap between communication and computation, and also present and restructure the SqueezeNet convolutional neural network that we have selected as a use case for our thesis. In Chapter 4, we present our work on optimizing individual hls4ml functions that implement CNN layers using directives provided

by Vivado HLS. We then apply these optimizations on SqueezeNet's *Fire* module in chapter 5. In Chapter 6, we introduce the optimizations on all SqueezeNet modules, that we then partition into multiple FPGAs and gather the latency and resource utilization results. Finally, in Chapter 7, we summarize the key points of our work and discuss future work items that can further improve, evolve and utilize the outputs and the lessons learned in this thesis.

Chapter 2

Related Work & Background

2.1 Related Work

The topic of FPGA implementations of convolutional neural networks is being actively researched in the present time. In this section, we present related work that focuses on three of the many aspects of this topic; namely the *usage of fixed-point data types with arbitrary size* to leverage the flexibility FPGAs provide in that regard, SqueezeNet implementations *using a single FPGA module* and implementations *on a multi-FPGA environment*.

2.1.1 Minimizing memory overheads

In an FPGA environment, where memory is scarce, efforts to beat the memory wall are highly important, especially in the realm of CNN implementations where ample memory is typically required to store the neural network parameters. The work in [17] introduces an efficient method to approximate floating-point parameters using dynamic fixed-point arithmetic, with minimal loss in accuracy. Ristretto, the framework presented, automates the process of converting floating-point model parameters to fixed-point approximations. Pitsis et al. [18] utilize advanced methods to successfully compress the weights of convolutional neural networks, namely weight pruning and clustering; their implementation achieves **1.23x** faster throughput and consumes **5x** less energy compared to a GPU counterpart, with only **0.6%** classification error. The majority of FPGA implementations use fixed-point data representations, as mentioned in [10]. Our approach uses a fixed-point representation for the network parameters; future implementations can explore the benefits of employing more sophisticated weight compression techniques.

2.1.2 CPU implementation

In this work, we include the results of implementing and evaluating SqueezeNet on an Intel Core-i5 3570 CPU, using the Keras framework. For 10000 images, the

average measured latency is **18 ms** (**180 seconds** total latency). For a single image, the inference latency is higher, reaching **50 ms**. Comparatively, our approach can complete such task in **33 seconds** ($3.3ms_{tput} * 10000 + 24ms_{lat}$), resulting in a **5.45x** speedup over the CPU implementation. This evaluation is presented in detail in Subsection 6.6.3.

2.1.3 Single-FPGA implementation

Most single-FPGA approaches store the network parameters in shared memory space (e.g. DRAM) and transfer them to local buffers inside the FPGA when needed (the *time-shared paradigm* discussed in Subsection 1.1.1). This is the case for the implementations we refer to in this segment. ZynqNet [19] shows the implementation of a SqueezeNet-based convolutional neural network optimized to run on a single Zynq SoC, using Vivado HLS. It is pointed however, that an issue with Vivado HLS was encountered, that resulted in pipelined regions to unnecessarily flush their data, causing a slowdown by a factor of **6.2x** on the expected design latency. This approach therefore achieves a latency of **1955 ms per image**, i.e. a throughput of **0.51 CPS**. We also encountered a number of issues caused by Vivado HLS, that we document and propose solutions to in this work. It should be noted that this work also presents the NetScope tool [1] for visualizing convolutional neural networks. Netscope assisted us in various steps of our work by visualising the SqueezeNet CNN. Our implementation reaches a throughput of **303 CPS**, i.e. **594x** speedup over ZynqNet.

EdgeNet [20] also presents a single-FPGA implementation of SqueezeNet on the Cyclone V FPGA. Similarly to our NewFire module (presented in Section 3.6), the developers of EdgeNet identify a module with minimal output size that they call "Computation Block". The Computation Block contains two expand, one pooling and one squeeze layer and one instance of it is implemented in the FPGA using Vivado HLS. The module is executed multiple times in sequence for each inference task. Since there is no batching in this implementation, the hardware is fully engaged for each inference task and therefore cannot accept new data before the previous task has finished processing. Therefore, EdgeNet achieves a throughput of **9 CPS**. Comparatively, our multi-FPGA implementation offers a **33.6x** speedup over EdgeNet.

Huang et al. [21] demonstrate an implementation of SqueezeNet on a combination of a Xilinx VC709 evaluation board (that contains a Virtex-7 FPGA) and an Intel Core-i7 7700k CPU. This heterogeneous approach uses the FPGA for calculating the convolutional layers, while the CPU calculates the pooling layers. This work achieves parallelism on both the input and output channels of each layer; in our approach, we achieved parallelism only on the output channels of each layer, since Vivado HLS identified data dependencies in the hls4ml code. However, due to our deeply pipelined design, we achieve a faster throughput compared to the

[1] The Netscope tool is available at https://dgschwend.github.io/netscope/quickstart.html

aforementioned approach, which demonstrates a throughput of **274 CPS**. This is indicative of the high scalability opportunities our approach offers.

2.1.4 Multi-FPGA implementations

The work by Zhang et al. [22] uses a dynamic programming algorithm to accommodate AlexNet and VGG on a multi-FPGA pipeline. Here, a method for optimized mapping of the network layers in different FPGAs is proposed, considering both the size of each hardware module and the inter-FPGA data transfer latency. Different solutions are demonstrated for each CNN, with a varying number of Virtex-7 FPGAs in the pipeline (*from one to four*). The image latency fluctuates between **30 and 213.6 ms** for the different solutions. Our work achieves a competitive **24 ms** latency, albeit for a different CNN (i.e. SqueezeNet). Future endeavors can evaluate implementation of AlexNet using the optimization primitives our work defines.

Layer partitioning in multiple FPGAs is explored in [23, 24]. In particular, [23] demonstrates a throughput of **1173 CPS** on SqueezeNet, using 6 Virtex-7 FPGAs. Both works perform an analysis of each layer operational and data requirements in order to efficiently partition it accross different FPGAs. This result achieves a speedup of **3.9x** over our approach, signifying the importance to explore intra-layer partitioning and thus pursue higher parallelism factors. Moreover that approach achieves **195.5 CPS/FPGA**, whereas our work reaches **60.6 CPS/FPGA** (for a five-FPGA implementation). It should be noted however, that the Virtex-7 FPGA model used in that work offers more programmable logic real estate compared to the Zynq UltraScale+ we use for our approach (e.g. there is more than double the BRAM availablle in the Virtex). Referring to the results we present in Section 6.6.1, our approach could be accommodated in less than five Virtex-7 FPGAs, which would improve the CPS/FPGA metric.

2.2 Vivado HLS

The Vivado HLS is a programming suite from Xilinx that allows the hardware designer to define a hardware module or architecture using `C` or `C++`, therefore minimizing development cycle and increacing production throughput. When the code development is completed, the user can *synthesize* the code to have it converted to RTL. HLS provides a method to define testbenches to test the code validity and a design analysis tool that provides a graphical representation of the module execution flow in steps, allowing the developer to gain insight on how the hardware implementation behaves and optimize it by adjusting its code appropriately.

2.2.1 HLS Directives

Vivado HLS introduces code directives that the developer can use in order to optimize the code, by requesting for the tool to synthesize a part of the code in a particular way. The directives are therefore a very important aspect of developing on HLS, since they provide a link between the high-level code definition and its hardware implementation; this allows the developer to define code optimizations, as well as constraints and interfaces. Here we will present the directives we used for our design, by describing their purpose and their parameters.

- `PIPELINE`: The `PIPELINE` directive applies to functions, loops and loop nests inside the code; it allows for consecutive function calls and iterations of the loop body to begin execution before the previous function call or loop iteration has finished processing, essentially pipelining the function or loop body. In order for the pipeline to behave as expected, the developer must take into consideration inter-loop dependencies and take appropriate action to resolve them. The directive uses the following parameters:
  - `II` (Initiation Interval): This parameter defines the interval, in clock cycles, after which the next iteration begins processing. The default value is 1, indicating that the loop body accepts new input every clock cycle. If HLS fails to meet the requested initiation interval, it issues a warning and implements a pipeline with the lowest initiation interval possible.
  - `enable_flushing`: This parameter applies only to pipelined functions; when the input data valid signal turns inactive, the pipeline will flush its data to its output and then stop; otherwise it will stall with unprocessed data still in its pipeline stages.
  - `rewind`: In the case of loop pipelining, under normal circumstances the pipeline will cause a bubble between the end of a loop iteration and the beginning of a new one. The `rewind` parameter allows the next loop iteration to begin before the previous has finished, removing the bubble from the pipeline.
- `UNROLL`: By default, HLS synthesizes loops into a hardware module that implements the loop body and adds logic that executes it as many times as the loop iteration count. Although the produced design has minimal resource requirements, it will be quite slow, as it is executed sequentially. The `UNROLL` directive unrolls the loop body, which translates to multiple hardware block implementations with concurrent execution; it thus introduces parallelism to the design. In order for the unrolling to be successful, inter-loop dependencies, as well as hardware constraints must be taken into consideration: For example, when the loop body reads data from a memory, the memory module must allow for multiple instances of the body to read from it in parallel;

otherwise, the parallelism benefits will be limited. The following parameters allow for the fine tuning of the directive:

- `factor`: This numeric parameter defines the unroll factor of the directive. If a value is not specified as an unroll factor, HLS assumes full unrolling of the loop. However, this behavior is not supported in loops with unknown bounds at synthesis time and therefore an integer value must be defined as unroll factor. The `factor` value does not need to be an exact divisor of the loop iteration count; HLS performs appropriate exit checks to ensure design correctness.
- `skip_exit_check`: This parameter applies to unrolling with `factor` value set and unknown loop bounds at synthesis time. Under normal circumstances, HLS would implement logic to check for loop exit, as it would not be aware if the loop iteration count is a multiple of the `factor` value. However, If the developer knows that this is the case, they can enable the `skip_exit_check` parameter and instruct HLS to not implement exit check logic and minimize latency and resource utilization.
- `region`: This parameter changes which loops get unrolled. If the directive is declared in a loop nest with `region` disabled, it will unroll the loop in which it is declared and will keep the inner loops rolled; if `region` is enabled, the directive functionality is reversed: it will keep the loop in which it is declared rolled and will unroll all the inner loops.

The `UNROLL` directive can also be declared in a function body. In this case, all the loops inside the function are unrolled according to the directive parameters.

- `ARRAY_PARTITION`: HLS implements arrays into memory elements in the synthesized design, with one array being assigned to one memory module. The `ARRAY_PARTITION` directive can be used to instruct HLS to split the array into multiple memory modules, or even registers. This method allows for multiple concurrent accesses to the same array and can be used in conjunction with the `UNROLL` directive to satisfy the requirement for parallel memory access of the generated hardware modules. The directive has the following parameters:
  - `variable`: Defines which array variable the directive refers to.
  - `factor`: This parameter accepts an integer value that defines the number of memory modules the original array will be partitioned into. If a partitioning factor is not defined, the array will be fully partitioned.
  - `dim` (dimension): If an array is multidimensional, this parameter defines which dimension the directive applies to. If 0 is used, the directive applies to the whole array.

  - **type** (`block`, `cyclic` or `complete`): This defines how the data of the original array will be assigned into the generated memory modules. Given an array **A** of size n and a factor f, when `block` is used, the first n/f elements will be assigned to the first memory module, the next n/f elements will be assigned to the second memory module, etc. The last elements will be assigned to the f-th module. The `cyclic` type assigns the first array element to the first module, the second element to the second module, the f-th element to the f-th module, the $f+1$-th element to the first module again, etc. When `complete` is used, the dimension is totally partitioned into memory modules and the `factor` parameter is ignored. If the array is unidimensional or 0 is assigned to `dim`, the array is converted into registers, each of which contains one array element.

- `INLINE`: By default, HLS implements subfunctions into hardware blocks separate from the higher-level functions. The `INLINE` directive transfers the subfunction code into the higher-level function and implements a single hardware block for them. This allows for area optimizations to take place; however, the implemented logic of an inlined function cannot be shared between two or more higher-level functions: Each function call will generate its own logic to implement the function, resulting in higher resource optimization. The directive can be controlled with the following parameters:

  - `region`: By default, the function to be inlined is the function that contains the `INLINE` directive. If `region` is defined, the directive instead applies only to the functions called from this function and is by default non-recursive.
  - `recursive`: This changes the behavior of the directive to inline all the functions called from the directive declaration onward.
  - `off`: Used in particular functions and prevents them from being inlined by the previous parameters. Additionally, HLS might automatically inline small functions; this parameter disallows it.

- `DATAFLOW`: This directive is used to introduce task-level pipelining to a design. It allows for a series of functions with producer-consumer relations to work on different inputs, minimizing the time the design needs before it can accept new data, as shown in figure 2.1. HLS identifies and examines data channels between the functions and implements a series of FIFOs or ping-pong buffers for the data transactions.

  The `DATAFLOW` directive can be applied to a body of a function or a loop that contains a series of function calls. In order for a function or loop to be used with the `DATAFLOW` directive, it must meet the following criteria:

  1. The variables used for data transfers must be non-static scalar or array

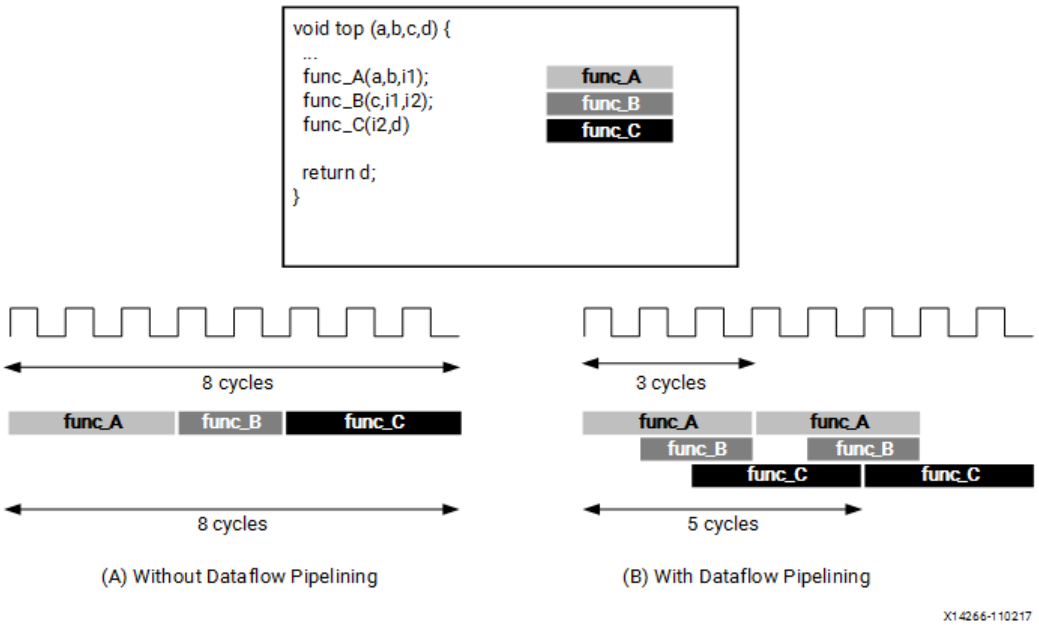


Figure 2.1: Task-level pipelining with the `DATAFLOW` directive. Image form Vivado HLS User Guide ([1]

variables, or static stream variables and must be defined locally (inside the function body).

2. Any non-scalar variables can have exactly one reading and one writing process (single producer-consumer model).
3. The variables used for inter-function data transfer must be first written to and then read from.
4. The arguments of the functions in the dataflow region must be first read from and then written to (if applicable).
5. The function return types must be void.
6. No loop-carried dependences between the pipelined functions are allowed.
7. Feedback loops between the pipelined functions are not allowed.

The directive does not include any configuration parameters. The variables that carry data from one function to another are implemented as FIFOs if they are scalar and as ping-pong buffers by default if they are arrays, to allow for random access. The user can dictate HLS to convert arrays to FIFOs, using the `STREAM` directive shown below, or use the HLS configuration flag `config_dataflow -default_channel fifo` to automatically convert all arrays in the dataflow region into FIFOs.

- `STREAM`: This directive allows for arrays used in a dataflow region and are accessed in a sequential manner to be implemented as FIFOs, as a more efficient communications mechanism in comparison to ping-pong buffers. The directive can be configured with the following parameters:
  - `variable`: Sets the name of the array to be configured by the directive.
  - `depth`: Indicates the depth of the FIFO to be created. By default, the FIFO has the same depth as the array it implements. In the case that large FIFO depths are not required, e.g. when the initiation interval of every function in the dataflow region is 1, the FIFO depth can be reduced and thus minimize resource usage.
  - `dim` (dimension): For multidimensional arrays, it specifies the array dimension to be converted into a FIFO.
  - `off`: Overrides the `config_dataflow -default_channel fifo` HLS flag and disallows a particular array to be automatically converted into a FIFO, but instead into a ping-pong buffer.
- `INTERFACE`: This directive is used to define the interface that will implement the top-level function parameters of the module. HLS allows for the parameters to be implemented into a plethora of interfaces, most notably using AXI4 and AXI4-Lite, AXI-Stream protocols, simple handshake and data-valid protocols, standard memory and bus interfaces, or no protocols at all. Each option can be further configured as needed.

HLS allows for two different methods of applying directives into the code. The first method uses the `#pragma` notation and the developer can simply write it as a part of the code in the source file. For example, an `ARRAY_PARTITION` directive on the bidimensional array `foo` with factor 5 on its second dimension and `cyclic` type would be declared as

```
#pragma HLS array_partition variable=foo cyclic factor=5 dim=2
```

The second method to define directives is with the *solution system* the HLS provides. The developer can create multiple different solutions for a single project, each of which can contain different project configurations and directive sets. These settings are stored in tcl files and are handled by HLS. This method allows the developer to use multiple directives in the same project and compare their impact on the synthesized result.

2.2.2 Arbitrary precision variable formats

FPGAs, compared to general purpose CPUs and GPUs, have the advantage of allowing arbitrary precision operations, that speed up the design and minimize the parameter size. Vivado HLS allows for the declaration of arbitrary precision

variables by using custom variable types that can be loaded using the appropriate header file.

In this thesis, we use the `ap_fixed` variable format, that provides a fixed-decimal-point representation of variables, with predefined bit-widths for the integer and fractional parts. The format declaration is included in the `ap_fixed.h` header. A variable declared as `ap_fixed<w,i> var` (where `w` > `i`) is implemented by Vivado HLS a memory element with a bit-width of `w` and an integer part of `i`. Therefore, the fractional part of `var` is `w` − `i`.

We additionally use the `ap_uint` variable format, used to declare unsigned integers of arbitrary bit size. This format is included in the `ap_int.h` header file and can be used as `ap_uint<w> var`; here, `var` is implemented as an unsigned integer of bit-width `w`.

2.3 The hls4ml framework

hls4ml, developed by J. Duarte et al. [25] and based on the RFnOC framework by E.J. Kreinar [26], is a framework that accepts a convolutional neural network definition from the Keras API as a pair of `.json` and `.h5` files (for architecture and parameter definitions) and automatically generates a Vivado HLS project with the network pipeline written in C++. hls4ml uses a set of function definitions, each of which implements common layers of a convolutional neural network. Listing 2.1 shows the steps to export the definition of a network model from keras to a pair of `.json` and `.h5` files.

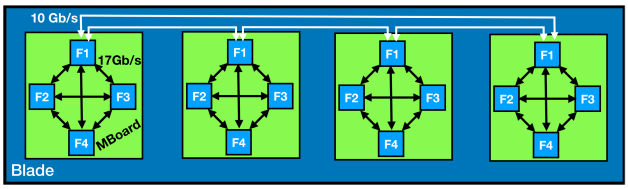
Listing 2.1: Steps to export a neural network definition from Keras

```
1 model_json = model.to_json(indent=4)
2 model_file = open(''model_name.json'', ''w'')
3 model_file.write(model_json)
4 model_file.close()
5 model.save_weights(''model_weights.h5'')
```

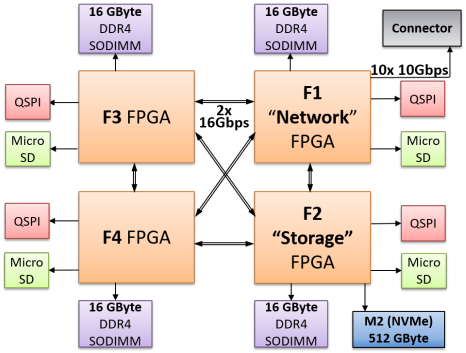
The top-level function of the generated project includes the calls to the appropriate functions and defines the arrays for inter-function data transfers. The parameters of each function are stored in separate header files that are also included in the top-level function. We have selected hls4ml as the codebase for our work, because its output (network architecture and parameters) can be easily distributed across multiple FPGAs and it uses a simple and straightforward approach for the function definitions.

2.4 ExaNeSt HPC prototype and FPGA resources

Our work for this thesis is built upon the interconnected QFDB prototype of the ExaNeSt project [12, 27, 28]. A high-level view of the prototype interconnect used



(a) Mezzanine board



(b) QFDB block diagram

Figure 2.2: Block diagram of the QFDB and the mezzanine board of the ExaNeSt prototype accommodating 4 QFDBs

in this study is depicted in figure 2.2a. This currently consists of 12 mezzanines each one carrying four (4) Quad FPGA Daughter Boards (QFDBs) for a total of 12x4x4 FPGAs or 12x4x4x4 ARMv8 cores. The QFDBs, depicted through green boxes, are connected in a 3D Torus topology.

As shown in figure 2.2b, each QFDB provides four (4) interconnected FPGAs, and one SSD, within a small footprint (120 mm x 130 mm). The FPGAs are Xilinx Zynq Ultrascale+ devices (ZCU9EG), featuring four (4) ARM-A53, 16-GByte DDR4. Each FPGA additionally contains, 600K reconfigurable logic cells (LUTs), 2520 DSP slices, and 32 Mbits of internal memory. There are two GTH transceivers (16 Gb/s each) for each FPGA pair, offering a total bandwidth of up to 32 Gbps.

The top right FPGA, referred to as the "Network FPGA", provides connectivity to the external world through ten (10) GTH links. The bottom right FPGA, named the "Storage FPGA", provides connectivity to the NVMe memory through PS-GTR transceivers implementing a 4xPCIe Gen2.0 channel. Finally, each FPGA can boot from an attached NOR flash, accessible through QSPI. Our platform supports two networks, namely, Exanet and 10G Ethernet.

ExaNet is a custom packet-based hierarchical interconnect realized over high speed serial links, developed within the ExaNeSt project [27]. Within each QFDB, there is an all-to-all connectivity, both for ExaNet and Ethernet traffic, shown using black arrows among F1, F2, F3, and F4.

2.4.1 FPGA: Targeted clock speed and resource planning

For reference, Table 2.1 shows the available resources in the MPSoc platform. The network interface of ExaNeSt consumes as little as 5% of these resources, thus offering nearly the entire FPGA for acceleration.

BRAM	DSP	FF	LUT
1824	2520	548160	274180

Table 2.1: Available resources in Xilinx czu9eg-ffvc900-2-e MPSoC

In all our synthesis tests, we the targeted clock period of the accelerators at 5ns, i.e. 200 MHz. We use this frequency to approximate the frequency of the ExaNeSt system. Designs in this Ultrascale+ FPGA may run faster than that, i.e. 300-400MHz. We leave additional frequency optimizations as a future work item.

Chapter 3

Partitioning SqueezeNet into Multiple FPGAs

When splitting a CNN into multiple FPGAs that work in a pipeline fashion, multiple objectives become relevant. In this chapter, we discuss objectives relevant when splittig a CNN into multiple FPGAs (Section 3.1). We also discuss computation-communication overlap (Section 3.2) and present the communication primitives we have implemented for HLS kernels leveraging blocks the ExaNeSt network interface (Section 3.3). Then, we discuss potential issues with unbalanced designs, when different layers have different throughput (initiation interval) (Section 3.4).

Then, we present the SqueezeNet CNN that we use in our work (Section 3.5) and restructure its core module (the Fire module) to minimize communication (Section 3.6). Finally, in part 3.6.1, we discuss the representation of parameters that we selected for our SqueezeNet implementation (weights, biases, input/output).

3.1 Splitting objectives

In this thesis, we target to split the layers of a CNN into a set of interconnected FPGAs. Every layer will be accommodated in one FPGA, i.e. we do not examine multi-FPGA implementation of individual layers.

A straightforward approach to split a CNN across multiple FPGAs would be to *assign one network layer per FPGA*. However one can easily see that it is not an optimal solution, because it will require a large amount of FPGAs, with no guarantee that the hardware of each layer will utilize all the FPGA resources, due to parallelism limitations. Additionally, this method can result in highly unbalanced designs, since there is no guarantee that the processing rates of layers will be uniform. Finally, there is no forethought regarding the network transfer latency with this approach, as every layer output will have to travel between FPGAs causing delays in the pipeline.

Another alternative would be to attempt to *include as many layers as possible*

inside one FPGA and move to the next one when the first exhausts all its resources. This approach, however, also has some disadvantages: It still doesn't account for the size of inter-FPGA data transfers and their impact to network latency, i.e. the network layer subset to be accommodated in FPGA a might have an output that will cause delays when transferred to FPGA $a+1$, but those delays might be alleviated if the last layer of the FPGA a subset is instead accommodated FPGA $a+1$. Moreover, a design with very high FPGA utilization might pass synthesis in HLS but fail in Vivado due to limited real estate. Therefore, this approach is undesirable in the sense of having to rearrange the layer subsets, reoptimize and resynthesize each design multiple times.

Our main goal is to *improve the throughput of the inference engine in a scale-out manner*, by utilizing multiple FPGAs. Striving to meet this target, we set the following objectives:

- The hardware implementation of each layer should be as fast as possible, accepting a new task as fast as possible. In order to achieve this objective, in the next two chapters, we synthesize and optimize every layer separately, in order to make it work as fast as possible.

- The hardware, including both the computation engines and communication engines, as well as the parameters storage, should fit in the available FPGA resources. For the communication segment, we allocate 5-10% of the FPGA resources. An additional sub-objective is to uniformize the utilization of FPGAs, which we try to meet by grouping adjacent layers inside the same FPGA.

- The communication between FPGAs should be minimized. As the communication across FPGAs will be realized by serial links, it is essential to architect the network so that the layers that communicate heavily (with regards to bytes per inference task) are allocated in the same FPGA. In this chapter, we discuss how we minimized the communication by restructuring the SqueezNet network.

- When possible (and needed) we should overlap communication with computation. For this, special hardware engines are needed. Later in this chapter, we discuss a first implementation of the communication engine utilizing primitives from the ExaNeSt network interface, and coupling them with HLS primitives.

- Ideally, the FPGAs of the network should have uniform latencies, in order to eliminate bottlenecks, queueing latencies, and resource under-utilization. In order to meet this goal, we will try to group adjacent layer when possible inside the FPGA. In this chapter, we also discuss how can one deal with non-uniform latency.

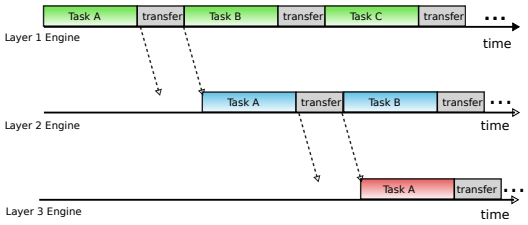


Figure 3.1: Timeline of inference pipeline: baseline case

3.2 Computation-Communication Overlap

Our multi-FPGA pipeline consists of stages of computations, where every FPGA completes a number of adjacent layers of a CNN for an inference task (input/image). One FPGA cannot start processing an image task before its upstream FPGA (layer) has produced the needed output, and this output has become available. Thus, a simple mode of operation may be as follows: every FPGA produces its entire output and then transfers it to the next FPGA so that the inference can progress. Note that this operation does not preclude pipelining: after passing its output, the first FPGA can start processing the next image, while the next one continues the processing of the first one. In this section, we discuss ways to optimize the timing of computation and communication segments of one inference task happening across different layers.

A simple pipelined architecture is shown in figure 3.1. Here we can see that after Layer a engine has completed processing its data, it has a new output vector ready, and initiates a transaction to transfer this vector to the engine for Layer $a+1$. While the output from a is transferred to $a+1$, a stalls, because there is no computation-communication overlap. In addition, while a is producing (or even transferring) its output for $a+1$, the latter layer stalls, because it waits for a to complete (and transfer) its full output, i.e., there is no computation-computation (or communication-computation) overlap between adjacent layers.

An optimized architecture would allow for each engine to start processing data before the previous engine (layer) has completed its computation and transfer phases, as shown in figure 3.2. This hides the data transfer overhead and improves the throughput and total latency of the design. However, in order to achieve that, the transmitter engine needs to implement a communication engine that can work independently from the computation engine—this may require duplicate output arrays to store/transmit data). Moreover, the receiver engine should be able to accept data in a particular pattern that allows it to begin processing without waiting for the whole output from the previous stage to be ready and transmitted. This issue is therefore tied to the architecture of the design.

In our implementation of SqueezeNet, transfer times were much smaller compared to computation times (as seen in figure 6.6 of the final implemented design) and therefore we did not make an attempt to modify the design architecture to overlap computation with communication. However, by instrumenting the code

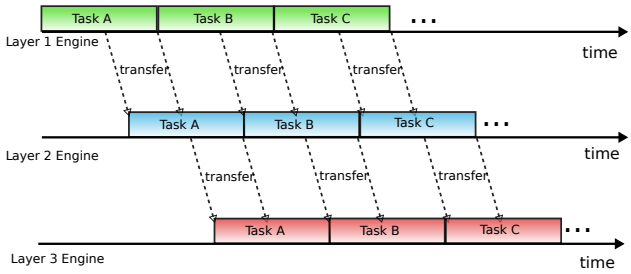


Figure 3.2: Timeline of inference pipeline: full overlap between computation and communication, partial overlap between computation of adjacent layers

and using the `DATAFLOW` directive, we allowed Vivado HLS to create streaming channels between adjacent layers implemented inside the FPGA. This method effectively *overlapped the computations between different layers implemented inside the same FPGA*. One may observe this in the final latency results in table 6.3: the *total latency of SqueezeNet is less than the latency sum of the layers it consists of.* We leave similar DATAFLOW operation across FPGAs as future work item.

3.3 Inter-FPGA communication using the ExaNeSt NI

HLS typically uses AXI stream to communicate between blocks. Although AXI stream may be suitable for intra-FPGA or even die-to-die communication, it is not suitable for a cluster interconnect. First, it is not reliable, as it does not have end-to-end acknowledgements; second, it does not provide completion notification at the receiver; third, there are no quality-of-service functions that are required in a shared, cluster network; finally, it cannot be used over the high-speed-serial links that are used to transport cluster-network packets –although the latter constraint can be overcome using an adapter that converts AXI stream to network packets, and vice versa, as we discuss in the future work section One of the target in the ExaNeSt project that created the ExaNeSt FPGA platfrom was to use a single unified network for both inter-process communication (e.g. MPI) and storage traffic, in order to decrease the cost and increase the utilization of the interconnect. For the aforementioned reasons, in this thesis we set out to use the ExaNeSt network interface primitives for communication between accelerators.

In this section, we show how we utilized the NI channels for accelerator-to-accelerator communication, without invoking the main ARM processors and the corresponding systems software–before this thesis, these channels were utilized only by user-level processes running on ARM cores. The IPs of the network interface have multiple channels, which can be configured by writing descriptors inside them. The general idea is that an accelerator is assigned a set of channels and issues network commands without any processor or operating system intervention.

We utilize three primitives of the ExaNeSt NI for inter-FPGA (direct accelerator-to-accelerator) communication. The ExaNeSt *packetizer (Exanetizer)*, suitable for

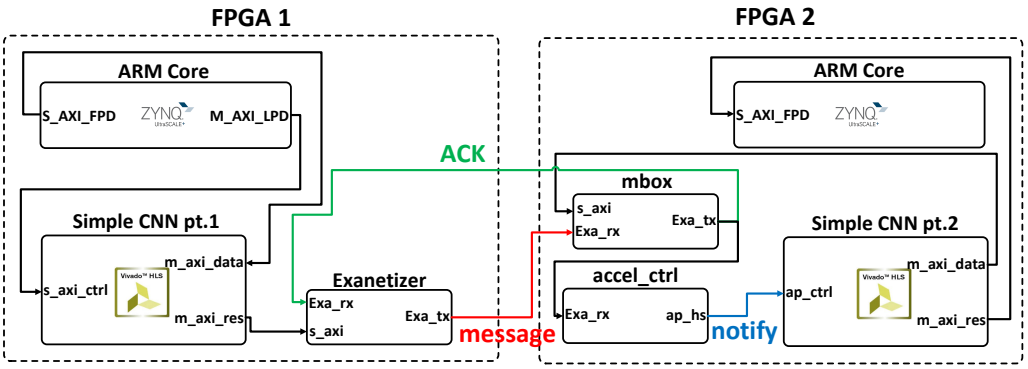


Figure 3.3: Our data transfer engine that utilizes the packetizer and mailbox

sending short (up to 256 B) messages; the *ExaNeSt hardware RDMA engine*, which can send 16KB messages; and the *mailbox (buffer) primitive* which can accept messages and keep them before the engine processes them. As shown in figure 3.4(a), we assume that flow control between adjacent stages prevents buffer overflows. Additionally, as we discuss in Section 3.6, we also strive to reduce the output of layers so that it fits to as few (16KB) messages as possible.

Both the packetizer and the hardware RDMA wait for an acknowledgement from the receiving end-point. Although these two NI primitives differ, their usage scenarios are quite similar: **1)** a hardware block (such as an accelerator) or the main processor writes a descriptor to one of their available channels—the case of the packetizer, the descriptor also contains the message payload, whereas with RDMA, the descriptor only carries the read address of the message payload. **2)** the NI primitive issues the transfer; **3)** the NI primitive waits for an end-point acknowledgement before the channel used becomes available again. To improve throughput, multiple NI channels can be used in parallel[1].

Figure 3.3 shows our implementation of the our accelerator communication engine that utilizes the ExaNeSt NI IPs and special new hardware. To demonstrate its operation, we implemented a simple hls4ml-generated convolutional neural network that we partitioned in two accelerator modules, each module placed in a distinct FPGA. FPGA 1 contains the first module (first part of the network) that we have configured to use an AXI4-Lite-encapsulated `ap_ctrl_hs` interface for parameter initialization, needed by the HLS module itself, and AXI4-Full (memory-mapped) for data reads and writes with external IPs, such as the main DRAM, and in our case, also the ExaNeSt NI blocks:

1. The user interacts through a program running on the ARM core of the Zynq UltraScale+ FPGA. Through this program, the user sends the initialization

[1]At this point, I would like to thank **Michalis Giannoudis** at FORTH that provided a modified version of the packetizer and mailbox tailored to our use case.

parameters to the accelerator module (memory address of input data, coordinates of the receiver FPGA and signal to initialize the process), using the AXI4-Lite interface of the module.

2. The module reads the input data (image) from the appropriate memory address through an AXI4-Full interface and starts processing.

3. When the data process completes, the module splits its output into four (4) 128-bit registers and prepares a special 128-bit word containing the destination address of the receiving FIFO of the second FPGA, as shown in listing 3.1.

4. A special block in the accelerator uses AXI4-Full interface to write these words inside the address space corresponing to the allocated packetizer channel.

5. The packetizer then creates an ExaNeSt packet and forwards it through the ExaNeSt interconnect, over high-speed serial links.

Listing 3.1: The trigger initialization function

```
1 ap_uint<128> set_packetizer_trigger(ap_uint<42> mailbox-addr,
     ap_uint<22> FPGA-coordinate, ap_uint<14> size-in-bytes){
2  #pragma HLS INLINE
3  ap_uint<128> trigger = 0;
4  trigger.range(77,36) = mailbox-address;
5  trigger.range(35,14) = FPGA-coordinate;
6  trigger.range(13,0)  = size-in-bytes;
7  return trigger;
8 }
```

On the receiving end (in FPGA 2), the accelerator module uses a simple handshake interface (`ap_ctrl_hs`) for status reporting and for accepting the initiation signal. We have developed an `accel_ctrl` module to control the accelerator; the procedure has as follows[2]:

1. `accel_ctrl` monitors the ExaNet output of the mailbox. This output will send the acknowledgement back to the packetizer once a new message arrives successfully. When the module see new data on this output, it knows that a new task arrived (*arrival notification*).

2. The module then waits for the accelerator to report idle status and then sends the signal to initiate data processing.

[2]The executables running on the ARM cores of the FPGAs, as well as the workflow to integrate our Vivado HLS accelerator into the QFDB infrastracture are heavily based on work by **Georgios Ieronymakis** at FORTH, to whom we express our gratitude.

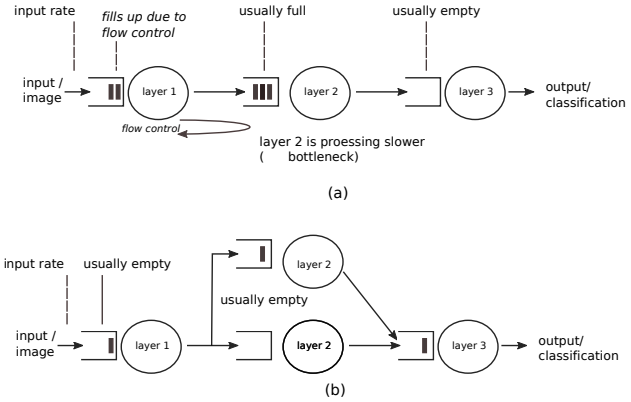


Figure 3.4: (a) Rate of pipeline stages and potential bottlenecks; (b) matching the rates using multiple stages

3. The accelerator initiates an AXI read from the local mailbox adddress (AXI4-Full) and begins processing the data.

4. When the second FPGA finishes its processing, the accelerator transfers the output data to a predefined memory address of the DRAM (AXI4-Full) accessible by a program running on the ARM core of the FPGA.

3.4 Stage processing bottlenecks in unbalanced designs

In this section we discuss the rate mismatch issue across the pipeline stages. This will manifest when the engines realizing some layers process tasks faster than other engines do. A slow layer cannot feed a faster one, therefore under-utilizing its capabilities. In addition, a slow layer cannot process enough the output of a faster one. As CNN layers process tasks in sequence, the overall throughput of the network will be dominated by the throughput of its slowest component.

As shown in figure 3.4(a), this can lead to under-utilization of some stages, as well as in-network backlogs and large in-network delays, hence also to large delay variations, which is unwanted for user-facing tasks, where we typically need to minimize the tail latency. These effects are analogous to network congestion in lossless interconnection networks.

There are multiple ways to deal with these issues. One is to slow down (admission control) the input rate to the network so as to avoid backlogs.

Although this can be a practical solution at operation time, there are other options which can increase the overall throughput at design time. One possibility is shown in 3.4(b). Here we show one can replicate a slow layer engine (space expansion) so that the compound capacity of the replicas can match the capacity of faster layers.

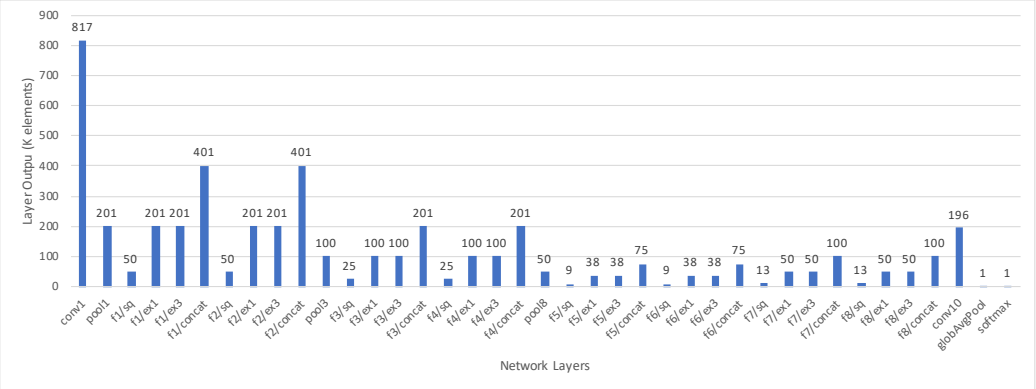


Figure 3.5: Output size of each network layer, in number of array elements (words)

Output size of each network layer, in thousands of elements (words). In our work, each word can be either 8 or 16 bit. *f*, *sq*, *ex1* and *ex3* stand for fire module and squeeze, expand1x1 and expand3x3 layers, respectively.

3.5 Original SqueezeNet network

SqueezeNet, developed by Iandola et al. [29], is a convolutional network designed to achieve classification accuracy similar to AlexNet, while requiring 50x less parameters, i.e. minimizing its memory footprint. This makes ideal for embedded and FPGA applications, where storage resources are scarce, according to its developers.

The core element of SqueezeNet is the *Fire* module, which contains one convolutional layer with 1x1 filters (called *squeeze*) that outputs its result to two concurrent covolutional layers, one with 1x1 filters and one with 3x3 (called *expand1x1* and *expand3x3*, respectively). There are 8 *Fire* modules in the SqueezeNet pipeline. Each convolutional layer is followed by a ReLU activation layer. SqueezeNet also includes two max pooling layers and in its pipeline and uses a combination of a convolutional and a global average pooling layer in lieu of a fully-connected layer. The final layer of the network is a softmax layer that serves as a probabilistic function for the classification of the input image. SqueezeNet also contains a dropout layer used for training; hence, we have omitted it from our work. In total, there are 57 layers in the SqueezeNet pipeline.

There are two versions of SqueezeNet – 1 and 1.1. The latter requires 2.4x less computations compared to the original, without sacrificing accuracy (described in the official SqueezeNet Github repository [3]). The SqueezeNet v. 1.1 architecture is demonstrated in figure 3.8a. As we further discuss in section 6.6, SqueezeNet performs a total of 778 million operations throughout its layers. We selected SqueezeNet for our work for two main reasons: first, its small architecture made it manageable for our work and allows us to demonstrate the feasibility of our method

[3]SqueezeNet Github repository: https://github.com/forresti/SqueezeNet

in a reasonable time frame; a large CNN would require large-scale testing and intense implementation effort, without extracting optimization primitives for the most part of the procedure. Second, the *Fire* module of SqueezeNet incorporates convolutional layers that execute concurrently, which adds another dimension for experimentation, compared to CNNs with purely sequential architectures.

SqueezeNet is originally implemented on the Caffe framework. Since hls4ml does not support Caffe implementations, we used a Keras implementation [30] as a starting point for this thesis.

3.6 Modifying SqueezeNet to reduce communication

In this section, we restructure the SqueezeNet network in order to minimize the inter-layer commmunication. In figure 3.5, we depict the output size of each layer of SqueezeNet, grouped by fire modules, in 1000s of elements, ignoring activation layers that have the same input and output size. It is clear that the output of each squeeze layer is a local minimum of its adjacent layers.

Under this observation, we can group the network layers in triads, with the input of each group feeding the two expand layers, which on their turn feed the squeeze layer through the concatenation function and the output of the squeeze layer becoming the output of the group. With this procedure, we essentially *modify the fire module definition, by shifting the layers it contains*, as shown in figure 3.6: The squeeze layer of the original fire module a is assigned to the new fire module a, whereas the expand layers of the original a are assigned to the new $a + 1$.

This method yields a module with minimized output, all while keeping the original SqueezeNet idea of a repeated layer-encapsulating module mostly intact. It additionally allows for a synthesis approach that is granular enough, by optimizing one fire module and then applying the optimization methodology to others. For clarity, we will from now on refer to the new configuration as the *NewFire module* and to the original SqueezeNet definition as the *original Fire module*, to avoid any confusion.

Note that we can place more than one fire module in each FPGA, up to as much as the available resources allow, while still making sure that the network overhead stays at a minimum.

Initial and final layers: The SqueezeNet layers that precede the fire modules, as well as those that follow them, will be treated individually and will be referred to as *initial* and *final* layers, accordingly. The two pooling layers between the original fire modules will now become part of the NewFire modules, creating the *NewFire module with pooling*.

In figure 3.7, we can view the SqueezeNet layers as parts of the original fire modules, as well as the new ones. Fire 3 and Fire 7 of the new modules are fire modules with pooling. This methodology results in a total of 7 NewFire modules (including the ones with pooling) plus the initial and final layers. We decided to assign the initial layers in one separate module (**Initial module**) and split

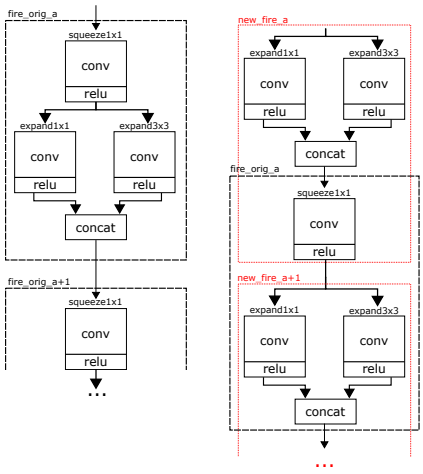


Figure 3.6: The original Fire module and its new implementation (NewFire module)

the final layers into two modules (**Final 1 and Final 2 modules**), based on the high resource utilization of the last convolutional layer of SqueezeNet that we observed in Section 4.1.4. The Final 1 module contains two expand layers and Final 2 module contains the final convolutional layer, the global average pooling layer and the softmax layer. With this partitioning, SqueezeNet consists of ten (10) modules, which we will optimize separately in chapter 6.

In figure 3.8b, we can observe SqueezeNet as partitioned with our proposal. In the following section, we will present the process of creating test projects on Vivado HLS for our designs, optimizing them, and preparing the glue logic needed to connect them.

3.6.1 8-bit & 16-bit parameter encoding

In this work, we fixed-point representation to encode the parameters for SqueezeNet, with relatively low accuracy (8 or 16 bit) in order to easier *fit the parameters inside FPGA BRAMs*. We test both 8-bit and 16-bit encoding, and we select different widths in different layers: in most cases, we select 8-bit, in order to minimize the BRAM utilization, but in some layers, we use 16-bit in order to maintain the accuracy of the computation. In addition, we use a 32-bit floating-point representation in the softmax (final) layer, which is responsible for the classification probabilities using function`expf()`, which accepts only floats, as discussed in Section 4.4.

For the 8-bit case, we use the fixed-point HLS representation `ap_fixed<8,4>` (4-bit integer 4-bit fractional) and for 16-bit we use HLS `ap_fixed<16,6>` (6-bit integer, 10-bit fractional). Determining an optimal size for the integer and fractional part of the numeric representation is outside the scope of our work. In our tests for different integer/fractional sizes, both the 8-bit and 16-bit data sizes yielded the same results with our original setup, in terms of latency and resource

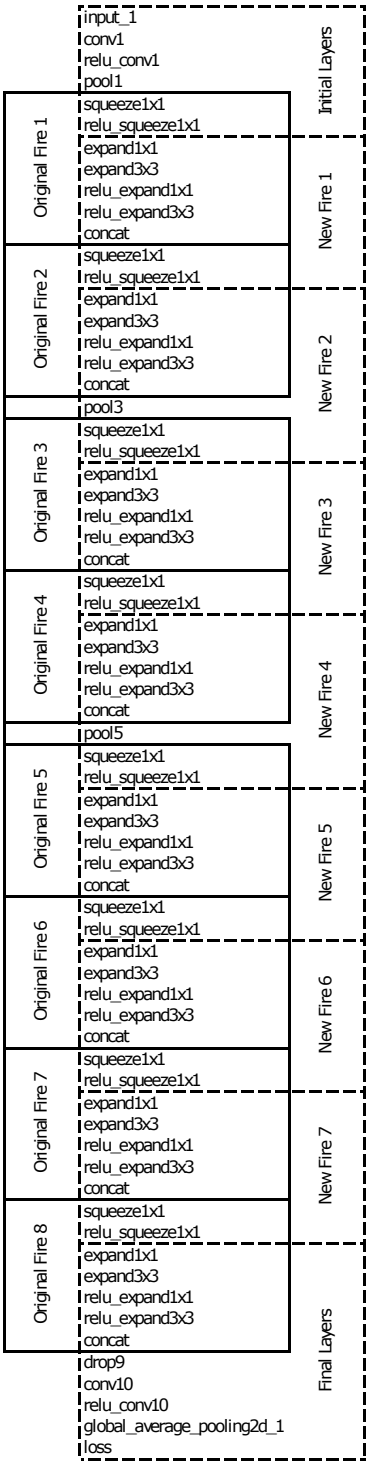
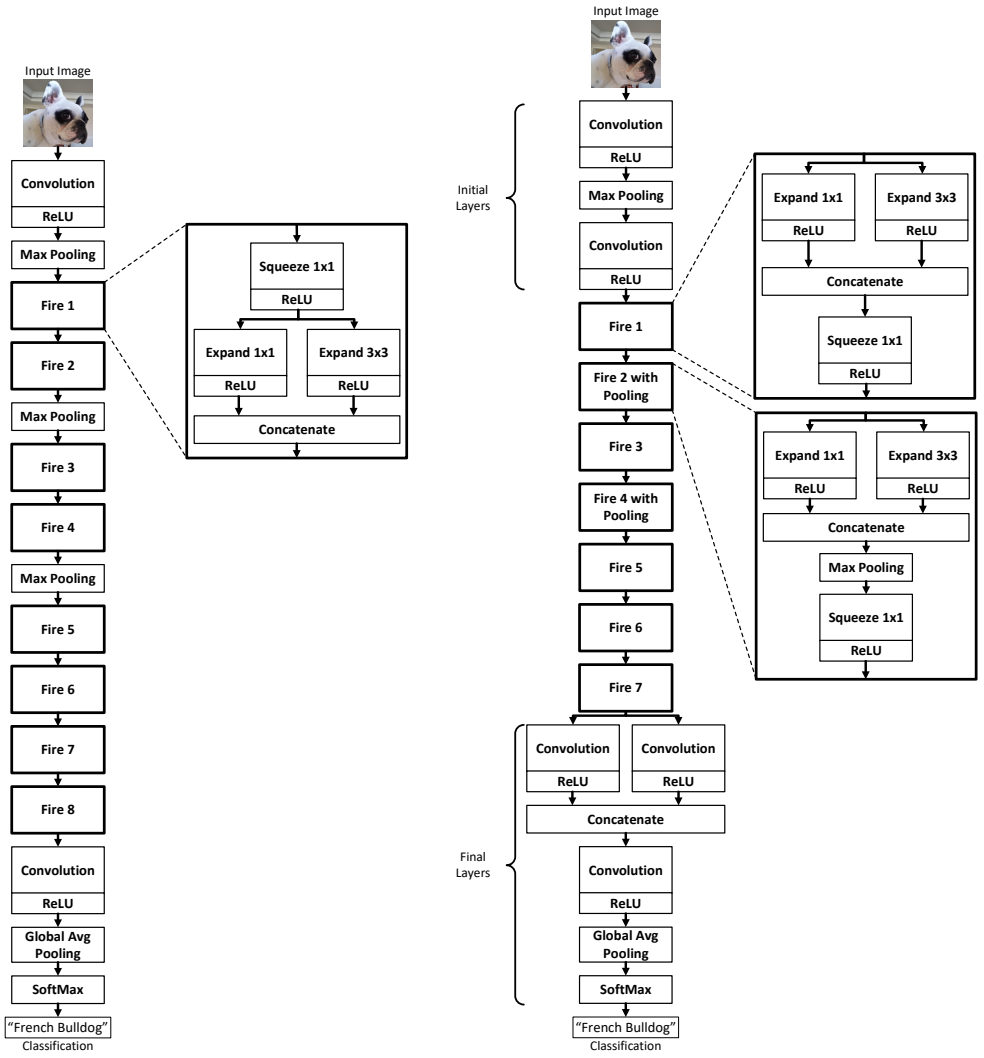


Figure 3.7: Assignment of SqueezeNet layers to Original & New Fire modules

utilization.



(a) Original SqueezeNet (b) Our proposal to partition SqueezeNet

Figure 3.8: Original SqueezeNet architecture and our proposal to minimize per-module output data sizes

Chapter 4

Convolution, Pool and Softmax Kernels

In this chapter, we focus on the core functions (kernels) of `hls4ml` that implement the individual layers of SqueezeNet, namely the **1)** *convolutional* (Section 4.1), **2)** *max pooling* (Section 4.2) **3)** *global average pooling* (Section 4.3) and **4)** *softmax* functions (Section 4.4). We then discuss the experiment results in Section 4.5.

Our evaluation includes both latency result and resource utilization comparisons for the different synthesis results on the Ultrascale+ MPSoC, described in Section 2.4.1. We perform multiple sets of experiments using different optimization directives and modify the code when needed (for both 8-bit and 16-bit data sizes in the convolutional function and 8-bit data sizes for the max pooling, global average pooling and softmax functions), in an attempt to define an optimization strategy that we then apply to our SqueezeNet implementation.

These will prove to be crucial for implementing SqueezeNet with latency less than 30 ms: starting from an non-optimized code for the convolutional function with latency of 1.5 sec, in this chapter we reduce it by a factor of 600x, bringing it down to 2.5 ms, as summarized in Section 4.5.

4.1 The convolutional kernel

The convolutional kernels are implemented by the `conv2d` core function of `hls4ml`. In this section, we present `conv2d` in its original form and point out its constraints and limitations. Subsequently, we introduce and evaluate improvements to the original design and discuss their impact on the synthesized module.

4.1.1 Original `conv2d` function from `hls4ml`

`conv2d` in its original form is defined in the header file `nnet_conv2d.h`, at the *nnet_utils* directory of `hls4ml`. The function accepts the input data, weights and biases and returns the results in unidimensional arrays, which are declared as

parameters in the function identifier. The array sizes for each `conv2d` call are defined in the header file `parameters.h` and are passed to the function as a type-name struct of type `conv2d_config`, using a template declaration. The identifier of `conv2d` is shown in Appendix A.1. There we can see that the template not only accepts the configuration struct from `parameters.h` in `CONFIG_T`, but it also allows for different type definitions of the input and output data arrays per function call. Additionally, we can see that the array sizes are set as products of the input and output dimensions and number and size of the filters.

After receiving the input data and configuration settings, the `mult` and `acc` arrays are declared as shown in Listing 4.1. These are used to implement the multiply-accumulate functionality of the module, with `mult` storing every multiplication between each data element and weight, which then get accumulated and stored in `acc`.

Listing 4.1: Definitions of arrays `mult` and `acc`

```
1 typename accum_t mult[out_height * out_width * n_filt * n_chan *
    filt_height * filt_width];
2    typename accum_t acc[out_height * out_width * n_filt];
```

The multiplication part of the convolution is a set of 6 nested loops that traverse the *input height, width and depth*, as well as each convolutional *filter, channel, filter height and filter width*, as shown in Appendix A.4.

As we mentioned before, the input and weights arrays are unidimensional, however, as they contain flattened three-dimensional objects, the array accesses are not linear; the correct indices are calculated and stored into the variables `index_data` and `index_weight` for every loop iteration. The same applies to the `mult` array, the index of which is stored in the `index_mult` variable. The code checks whether an element in the `mult` array might correspond to an out-of-bounds convolutional step, in respect to the stride height, width and padding of the convolution and, if it is true, sets the element value to zero. Otherwise, it stores the product of the input data element and weight.

The function then proceeds to initialize the `acc` array with values from the `biases` array through three nested loops, shown in Appendix A.3. The array accesses are non-sequential, as `acc` follows the same principle as `data` and `weights`: It stores a flattened three-dimensional object, therefore the position of each element is calculated on every loop iteration.

After this initialization step, the function proceeds to the accumulation part of the convolution. This part also consists of 6 nested loops and for each iteration the indices of `mult` and `acc` are calculated and the value of `mult` is added to that of `acc`, as shown in Appendix A.2. The values of `acc` are then cast to `res_T` type, which is defined in the function template and saved to the output array `res`.

In addition to the function definition, the original code contains the following HLS optimization directives declared as pragmas, in order to increase parallelism opportunities:

1. The `mult`, `acc` and `biases` arrays are fully partitioned with `ARRAY_PARTITION`. Notably, the `res` array also gets fully partitioned through a directive on the top module of `hls4ml`, however the `weights` array does not get partitioned.

2. The function uses the `FUNCTION_INSTATIATE` directive with `weights` and `biases` as parameters, to have the synthesized design contain multiple optimized instances of `conv2d`.

3. In order for the function to be able to accept new input on every clock cycle, a `PIPELINE` directive is declared.

4. The function implements a form of zero-weight pruning, by limiting the number of multiplication operations with the `ALLOCATION` directive. The maximum number of multiplications to be implemented is determined by the function shown in Appendix A.5, which is not synthesized; its output value is then passed as an argument to the directive. Note that the `reuse_factor` variable, which is passed from the configuration, can be used to further limit the total amount of multiplication operations to be implemented, in the form of a resource reuse factor, as its name suggests.

This was a brief presentation of the original code for the convolutional module, together with the optimizations it carried. In the following sections, we will present the tests we carried to assess its efficiency and synthesizability, its limitations and the changes and code improvements we made to further optimize it and shape it to better fit the needs of our design.

4.1.2 The problem with the initialization of large arrays

We perform the first test of `conv2d` by using the parameters of the last convolution of SqueezeNet, *conv10*, because of its large size and number of parameters in comparison to the other convolutional layers of the network. In detail, this convolution contains 512000 weights and 1000 biases (shown in 4.2), for a total of 513 thousand parameters, and a total of 100.4 million macc operations. This allows us to observe how the code would implement the most demanding part of our network and what course we should follow to adapt it to our particular needs. We use a 16-bit fixed-point format for this test, with 12 bits for the integer part and 4 for the decimal.

Listing 4.2: Weight initialization

```
1 weights_default_t w88[512000] = { <512000 initialization values> }
2 biases_default_t b88[1000] = { <1000 initialization values> }
```

In our initial attempt to synthesize the convolutional module, we found out that HLS would not accept the syntax of the `ALLOCATION` directive as declared in the original code by `hls4ml`. This is caused by the directive having the maximum number of multiplication operations passed to it as a variable, not as an integer. From the Github page of the project, it seems that HLS accepts this syntax when

using the tcl environment[1]. However, since we are using the GUI environment of the program, we disabled the directive altogether. We also determined that it would be best to remove all the original directives from the code, to get an unoptimized design as a baseline result, before adding them back in.

Throughout our initial synthesis attempts, the procedure would halt after displaying "Starting code transformations..." in the terminal and would stay there, even after allowing it to run for more than a day. Since this behavior was abnormal, we began removing parts of the `conv2d` code, to determine what was causing the issue. We pinpointed the problem to the HLS process that converted floating point values to `ap_fixed` ones, used in our case to convert the weight values to the appropriate format. As a matter of fact, when we removed the initialization values from the source code, the synthesis proceeded without problems. The biases array initialization, due to its much smaller size, did not cause any delays. In order to overcome this issue, we considered two options: First, we attempted to keep the weights and biases arrays uninitialized and observe how HLS behaved. The synthesis results showed that HLS detected that the array elements were all zero and were not updated throughout the design run, removed all the multiplication operations and set their outputs to zero and thus produced an incorrect, trimmed down design.

To overcome this problem, *we set the weights and biases arrays as input parameters* on the top module, essentially removing them as memory modules from the HLS design and creating interfaces to access them. This allowed the synthesis to complete and produced a correct design. The drawback of this approach was that *the design reports would not include the memory utilization of the weights and biases arrays* and the design itself would not be self-contained, as its memory modules had to be implemented outside of the HLS module. Our solution to this is to create empty weights and biases arrays inside the design and add two interfaces as top-level parameters, together with a parameter called `init_weights_biases`. Inside the function, before the multiplication part, we created for-loops that would initialize the weights and biases arrays whenever the value of `init_weights_biases` was 1. This method not only allows us to have a correctly-synthesized design, it also augments the functionality of the convolutional module, as it provided a way of updating its weights and biases values at runtime. In order for the module to sustain the weight and bias values between runs, we declared the arrays as `static`.

4.1.3 Merging Multiplication and Accumulation

After applying the changes on the array initializations, the design synthesized successfully, with a latency of about 907 million clock cycles, as stated in the HLS synthesis report. Since the design is synthesized with a clock period of 5 ns, the latency of the generated module was 4 seconds in its unoptimized state.

[1] https://github.com/hls-fpga-machine-learning/hls4ml/issues/2

However, the main issue of this design is its memory utilization, which required 65850 BRAM_18K modules. This translates to a 3600% memory utilization, since the FPGA itself has 1824 BRAM modules available. The large memory utilization is caused by the `mult` array, which consumed 65535 BRAM modules. By its definition, `mult` has a size of 86.5 million elements in this configuration, therefore its size has to be minimized.

After analyzing the code, we determined that we could merge the multiplication with the accumulation part, essentially rendering the `mult` array redundant. Therefore, we change `conv2d` by merging the two convolutional parts into a single set of 6 nested loops and moving the initialization of the `acc` array to the beginning of the function. Additionally, we *remove* the last part of the function used to cast the output to a different type, since it serves no purpose in our case. We subsequently set the `acc` array as the function output. These changes are reflected in Appendix A.6.

This modification resulted in a synthesized design with latency of 314 million clock cycles and a memory utilization of 34%, with only 627 out of 1824 BRAMs being used. The removal of the `mult` array is reflected in the vast difference of BRAM utilization between the original and the modified kernel and the merging of two distinct procedures into one resulted in a speedup of about 2.9x. In the following subsections, we evaluate and compare different design optimizations with the use of HLS directives.

4.1.4 Optimizing the `conv2d` kernel

The modified `conv2d` function now contains a single loop nest, which we will now focus on optimizing with HLS directives. In order to introduce parallelism, we will apply the `UNROLL` directive that will create multiple hardware blocks with copies of the loop body, together with the `ARRAY_PARTITION` directive that will allow the array memory modules to be accessed by the hardware blocks in parallel. An obvious starting point would be to apply the `UNROLL` directive to all six loops, in an attempt to produce a fully-parallelized design and observe the results. However, this is not feasible in our case, since the code has data dependencies between different loop iterations that restrict parallelism opportunities. For example, different iterations of the `cc` loop will access the same elements of `acc` for write operations, which will hinder the tasks from executing in parallel. Additionally, the `oh`, `ow`, `fh` and `fw` loops have non-trivial access patterns on the `data` array, making it harder to correctly partition it for multiple concurrent accesses. Therefore, we focus on the `ff` loop that has independent and straightforward data accesses on its iterations, making it ideal for unrolling and parallelization.

4.1.4.1 Using multidimensional arrays

We added the `UNROLL` directive on both the `InitFilt` and `ff` loops with an unroll factor of 1000 to match their iterations and applied the `ARRAY_PARTITION` on the

`weights` and `acc` arrays using the `cyclic` partitioning scheme to allow for concurrent array accesses. We additionally partitioned the `biases` array using the `complete` partitioning scheme. Although we expected the latency of the resulting design to be much lower compared to the unoptimized implementation, its latency exceeded 500 million clock cycles. By synthesizing parts of the code step-by-step, we determined that the problem was caused by Vivado HLS not being able to successfully infer the array access patterns and therefore creating an inefficient implementation. In order to make the access pattern obvious to HLS, we subsequently converted the arrays to multidimensional. This resulted in the `data` and `acc` arrays to become three-dimensional and the `weights` array four-dimensional. The `biases` array remained unidimensional.

In our next attempt to synthesize the design, Vivado HLS printed the following error (here trimmed down):

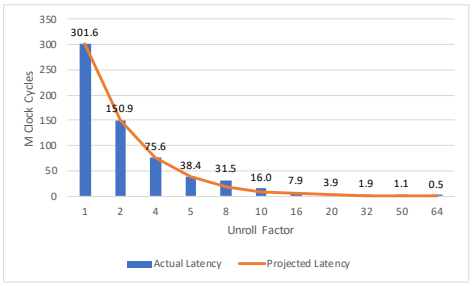
```
Stop unrolling loop 'ConvFilt' in function 'conv2d' because it may
 cause large runtime and excessive memory usage due to increase in
 code size.  Please avoid unrolling the loop or form sub-functions
                    for code in the loop body.
```

After disabling different loops in the nest to narrow the problem, we concluded that the issue was caused by the `ConvChan` loop; its iteration count was 512 and HLS could not implement an instance of it in each of the 1000 modules the `UNROLL` directive would create. We therefore rearranged the loops by moving the `ConvFilt` loop to the innermost position of the loop nest, since it did not affect the code. This allowed the synthesis procedure to complete without any problems.

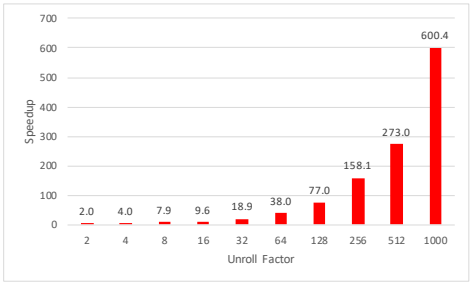
4.1.4.2 Optimizations using the `UNROLL` directive

In our first set of tests, we used a 16-bit datapath and observed the results of the `UNROLL` directive with unroll factor values of 2, 4, 8, 16, 32, 64, 128, 256, 512 and 1000 in both the initialization and convolutional loop nests. In figure 4.1a, we can see the latency results in millions of clock cycles, compared against projected latency estimates, i.e. the baseline latency divided by the unroll factor. Although the actual latency is somewhat slower than the projected, we nevertheless observe substantial performance improvements compared to the baseline approach, with a speedup ranging from 2 to 600.4, as shown in figure 4.1b.

In figure 4.2, we can observe the resources allocated for each experiment for BRAMs, DSPs, Flip Flops and LUTs, as reported by Vivado HLS. DSP usage, shown in 4.2b, matches the unroll factor of each test; this comes to show that HLS utilizes one DSP per multiplication module for 16-bit operation. In contrast, from figure 4.2a, we can see that the BRAM utilization does not grow in accordance to the unroll factor, but stays the same between different iterations. We attribute this to the array partitioning scheme which essentially improves throughput by distributing data between different block rams, leaving some underutilized. However, this was a non-issue for our implementation, since BRAM utilization did not



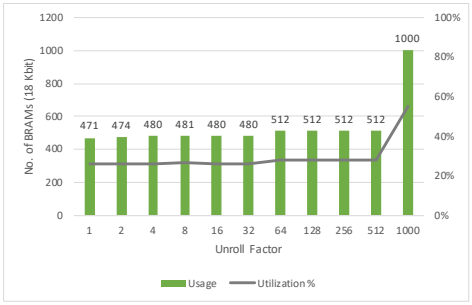
(a) Real and projected latency



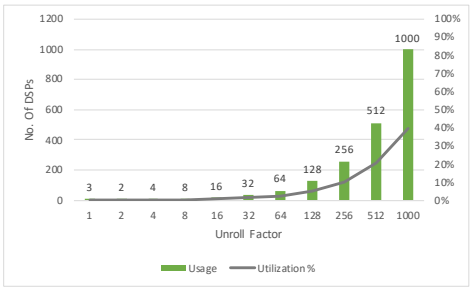
(b) Speedup compared to baseline implementation

Figure 4.1: Latency and speedup metrics of `conv2d` with 16-bit configuration, for different unroll factors

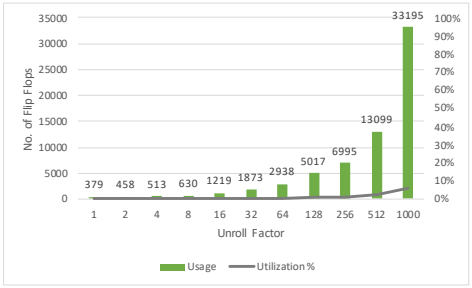
reach percentages higher than 55% across the tests. Additionally, Flip Flop and LUT utilization percentages were low even with the `ff` loop fully unrolled.



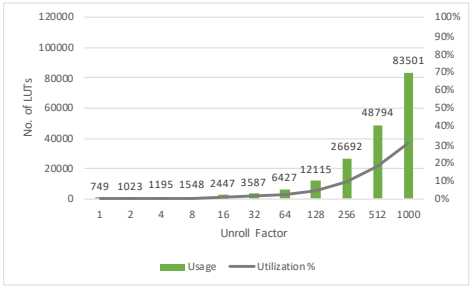
(a) BRAM usage and utilization



(b) DSP usage and utilization



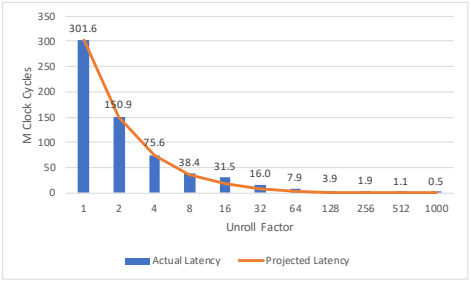
(c) Flip Flop usage and utilization



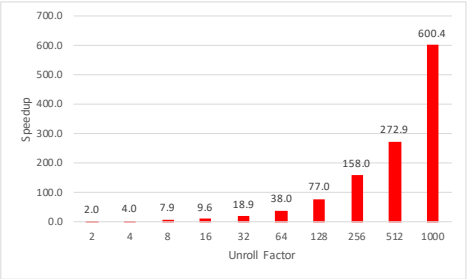
(d) LUT usage and utilization

Figure 4.2: Resource usage and utilization percentage of `conv2d` with 16-bit configuration, for different unroll factors

Although a 16-bit implementation showed reasonably moderate resource usage, we proceeded to further minimize the module footprint by creating an *8-bit datapath* using fixed-point data representations (4-bit integer and 4-bit decimal) and performing the same tests as before. For this test, we expect an obvious decrease



(a) Real and expected latency



(b) Speedup compared to baseline implementation

Figure 4.3: Latency and speedup metrics of `conv2d` with 8-bit configuration, for different unroll factors

in memory usage; however it will be interesting to see whether there will be any deviations in other resource utilization and latency. In figure 4.3, we can observe the latency results for different unroll factors applied on the module, together with the speedup they provide in comparison to a baseline design. The latency results of this configuration are largely identical to the ones with a 16-bit datapath. This is also reflected in the speedup in figure 4.3b and projected latency values.

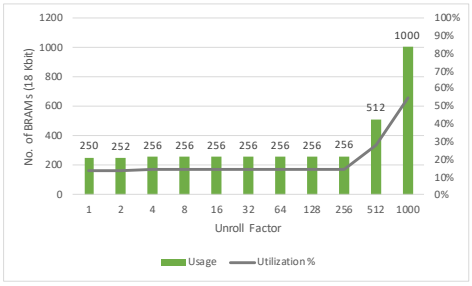
In figure 4.4, we can see the resource utilization metrics for the 8-bit design. As expected, the BRAM usage in figure 4.4a is almost halved in comparison to the 16-bit datapath, shown in figure 4.2a, with the exception of the last two tests, with unroll factors of 512 and 1000. This comes as a result of increased BRAM utilization, not for storage purposes, but in order to increase the data access throughput. An interesting observation is that HLS does not utilize any DSPs for the multiplications on this datapath; instead, they are implemented using LUTs, hence the higher utilization shown in figure 4.4c compared to figure 4.2d.

Impact of the `UNROLL` directive As we observed in the latency results, the `UNROLL` directive largely succeeded in increasing the design throughput by a factor of 600.4x in both the 16-bit and 8-bit configurations. Additionally, all the tests demonstrated reasonable resource utilization metrics, indicating that the design for the most memory and computation-intensive convolution of SqueezeNet could be easily accommodated in one FPGA.

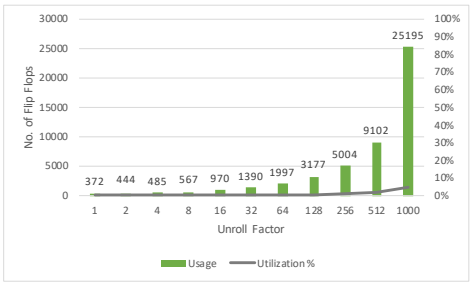
4.1.4.3 Optimizations using `UNROLL` with `PIPELINE`

In the following experiments, we will introduce the `PIPELINE` directive to the loop body. We expect it to further improve the design latency, by allowing the multiple hardware blocks created by the `UNROLL` directive to be able to accept new input every clock cycle. As with the previous experiment sets, we will run these tests for different unroll factors, both for 16-bit and 8-bit datapaths and will monitor the latency and resource utilization estimates as reported by HLS.

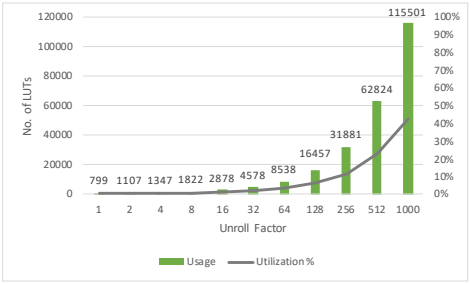
In figure 4.5, we can see the latency of the module with different unroll factors.



(a) BRAM usage and utilization



(b) Flip Flop usage and utilization

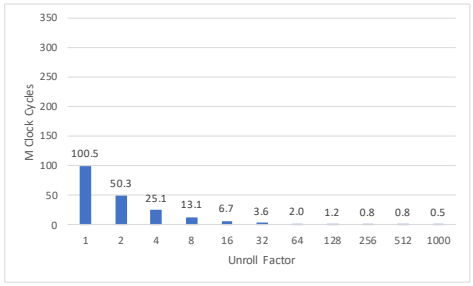


(c) LUT usage and utilization

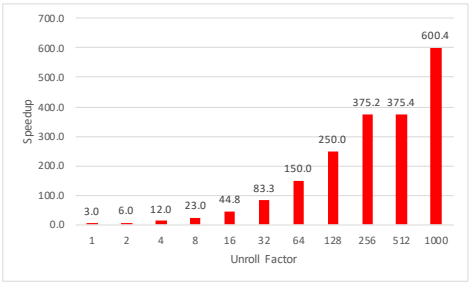
Figure 4.4: Resource usage and utilization percentage of `conv2d` with 8-bit configuration, for different unroll factors

Figure 4.5c provides an interesting insight: The introduction of the `PIPELINE` directive can provide a speedup of up to 2.7x compared to a non-pipelined version, and the performance of a pipelined design largely exceeds that of non-pipelined designs with an unroll factor value 4 times higher. As the unroll factor increases, the improvement of the directive diminishes; at full unroll, there is no improvement whatsoever and the design latency matches its non-pipelined version. This phenomenon can be explained by the fact that the more hardware blocks are generated, the fewer loop iterations will be carried by each one. At the fully unrolled test, HLS informed that *the `PIPELINE` directive was removed because the loop was fully unrolled.* This means that HLS disregarded the iterations of the enclosing loops as means of engaging the inner-loop pipeline. We could not revert this behavior even after enabling the loop rewinding option of the `PIPELINE` directive. The results of that configuration therefore match the previous one without pipelining.

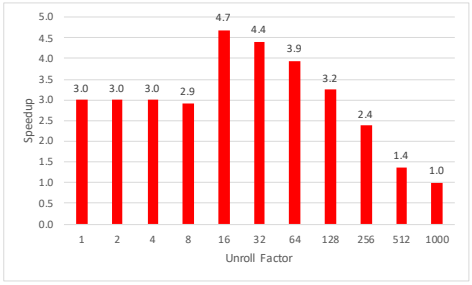
The resource utilization results of the tests with a 16-bit configuration and pipeline are shown in figure 4.6. The BRAM usage (figure 4.6a) has remained the same compared to the non-pipelined implementation; however the DSP usage has changed dramatically, especially for unroll factor values between 32 and 256; here we can see that the configurations allocate much more DSPs and the observation of one DSP per module no longer stands. The DSP utilization percentage stays reasonably low, nonetheless. We can also observe discrepancies in LUT (figure 4.6b) and Flip Flop (figure 4.6c) utilizations, albeit not as extreme.



(a) Module latency



(b) Speedup compared to baseline implementation



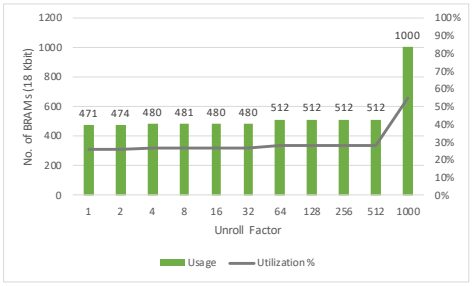
(c) Speedup compared to unrolling without pipeline

Figure 4.5: Latency and speedup metrics of `conv2d` with 16-bit configuration and pipeline, for different unroll factors

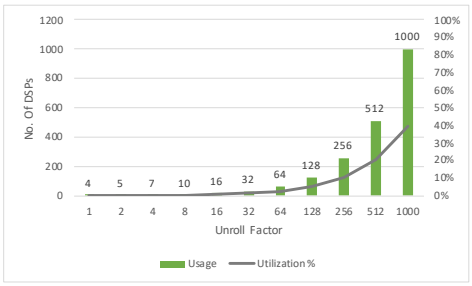
The latency and utilization results of the pipelined implementation demonstrate that the directive greatly improves the speed of the design, with little impact on the resource utilization (the resource with the highest utilization percentages is the BRAM, which has the same utilization metrics between the pipelined and non-pipelined design). We will therefore use the directive in conjunction with `UNROLL`, for loops with partial unroll factor.

We will now proceed with the test results of the 8-bit datapath with pipeline. The latency metrics shown in figure 4.7 are almost identical to their 16-bit counterparts, which is the expected behavior. Additionally, in figure 4.8 which demonstrates the resource utilization metrics, we can see observe the higher, yet expected usage of Flip Flops and LUTs. The DSP usage is almost non-existent, similarly to the non-pipelined configuration.

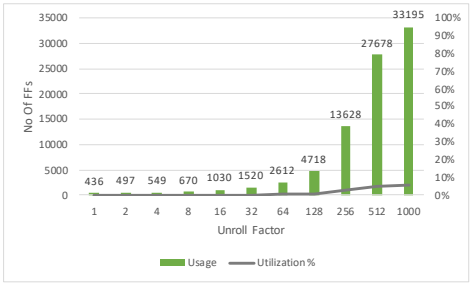
Impact of `UNROLL` with `PIPELINE`: The previous tests demonstrated that the `PIPELINE` directive can greatly improve the speed of the design, at a relatively small increase in resource utilization. Interestingly enough, it allows for speedups higher than non-pipelined configurations with higher unroll factor, resulting in an efficient and optimized design. However, it becomes useless in fully-unrolled loops, since Vivado HLS removes it at synthesis time, even though there might be outer loops that still drive the pipelined modules, as we demonstrated. For the rest of



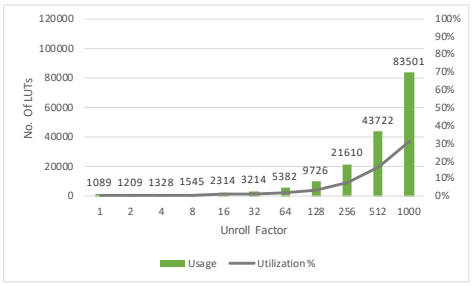
(a) BRAM usage and utilization



(b) DSP usage and utilization



(c) Flip Flop usage and utilization



(d) LUT usage and utilization

Figure 4.6: Resource usage and utilization percentage of `conv2d` with 16-bit configuration and pipeline, for different unroll factors

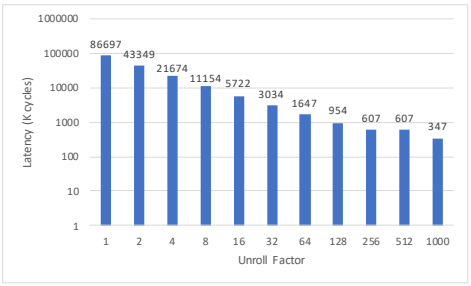
the functions implementations, we use *an 8-bit fixed-point representation with 4 integer and 4 decimal bits.*

4.2 The max pooling kernel

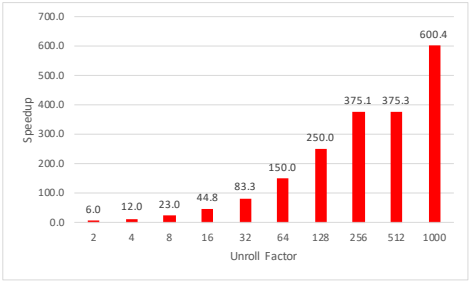
SqueezeNet uses three (3) max pooling layers in its architecture. In this section, we present **our own implementation** of the `maxpooling` function that we find better suited for our work than the original from hls4ml and optimize it. Subsequently, we demonstrate the optimization results in both latency and resource utilization metrics.

4.2.1 The `maxpooling` function definition

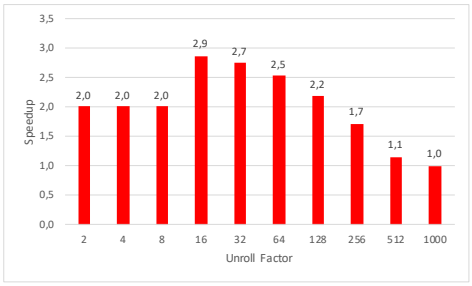
The max pooling functionality in `hls4ml` is originally implemented in the `pooling2d` function, located at the `pooling.h` header. This function declares the input and output unidimensional arrays as header arguments (shown in Listing 4.3) and accepts a configuration struct, in the same fashion as `conv2d`. The function then calls a number of smaller functions to calculate the result.



(a) Real and expected latency

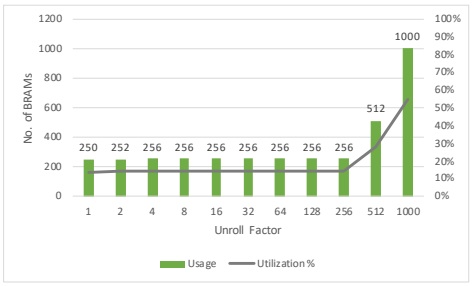


(b) Speedup compared to baseline implementation

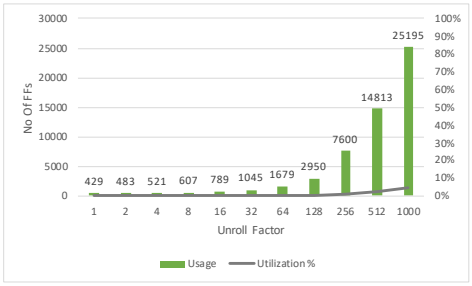


(c) Speedup compared to non-pipelined implementation

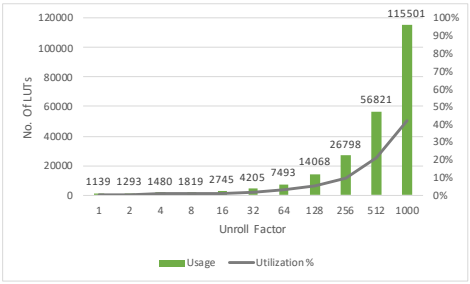
Figure 4.7: Latency and speedup metrics of `conv2d` with 8-bit configuration and pipeline, for different unroll factors



(a) BRAM usage and utilization



(b) Flip Flop usage and utilization



(c) LUT usage and utilization

Figure 4.8: Resource usage and utilization percentage of `conv2d` with 8-bit configuration and pipeline, for different unroll factors

Listing 4.3: The header of the original `pooling2d`

```
1 template<class data_T, typename CONFIG_T>
2 void pooling2d(data_T data[in_height * in_width * n_filt],
3 data_T res[out_height * out_width * n_filt])
```

Since our implementation of the convolutional function uses three-dimensional arrays instead of unidimensional (the original implementation of hls4ml) and the `maxpooling` function performs operations that rely tightly on the unidimensional form of its input/output arrays, we had to create our own `maxpooling` function. Thus, we can avoid any additional latency that would incur if we kept the original hls4ml implementation and convert three-dimensional arrays to unidimensional for the `maxpooling` function (and vice-versa).

Our implementation is shown in Appendix A.7. We took advantage of the fact that *every* max pooling function in SqueezeNet has a window size of **3x3** and a stride length of **2** to simplify the code. The three outermost loops iterate over the elements of the output array and the two innermost iterate over the pooling window and assign the max value to the appropriate element. In order to ensure the correctness of the new `maxpooling`, we executed it with random input data and compared the results with the output of a max pooling function we created in MATLAB. We then proceeded to synthesize the function and measure its latency and resource usage.

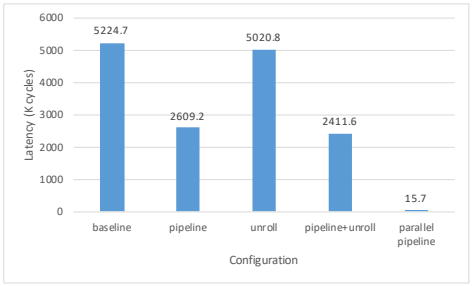


Figure 4.9: Latency of the max pooling function for different configurations

4.2.2 Optimizing the `maxpooling` function

For the `maxpooling` function, we evaluate three configurations: One without any optimizations, as a baseline result, one with a `PIPELINE` directive in the innermost `PoolWidth` loop, one with `UNROLL` applied to the `OutFilt` loop, and one combining the previous configurations, named *pipeline+unroll*. For the latter two configurations, we fully unroll the `OutFilt` loop, as there are no data dependencies to limit its parallelization opportunities and we fully partition the third dimension of the `data` and `res` arrays. We use the configuration of the *first* max pooling layer of SqueezeNet as a use case for our evaluation: it accepts an array with dimensions 113x113x64 and outputs an array with dimensions 56x56x64; it therefore performs 1.8 million operations.

The latency results of the four configurations is shown in figure 4.9, in thousands of clock cycles. The latency of the optimized designs is unexpectedly high; the *unroll* configuration is marginally faster than the baseline and almost twice as slow as the *pipeline* configuration - even though it should have parallelized the design by a factor of 64 (the number of the `OutFilt` loop iterations). Conversely, the *pipeline+unroll* configuration demonstrates different results compared to the *unroll* configuration, since there are loops that execute inside the body of each parallelized hardware module, namely the `PoolHeight` and `PoolWidth` loops.

In order to determine the cause of this result, we used the Analysis Perspective of Vivado HLS. As shown in 4.10, the tool has created multiple hardware blocks (indicated by the `PoolHeight` bars), however it has scheduled them to execute sequentially. Since there is no data or other kind of dependency to explain this behavior, we attribute it to the position of the `OutFilt` loop: because it is not the innermost loop, HLS has trouble parallelizing it, in a similar way it could not unroll the `ConvFilt` loop of `conv2d` when it was not the innermost loop. In this case, however, we could not rearrange the loop order, as it would affect the functionality of the module.

In order to overcome this issue, we decided to use the `PIPELINE` directive in a different way: we declared it in the `OutWidth` loop and set its initiation interval parameter to 1. This would instruct Vivado HLS to attempt to execute one loop iteration every clock cycle, by optimizing the loop body as much as possible, to meet

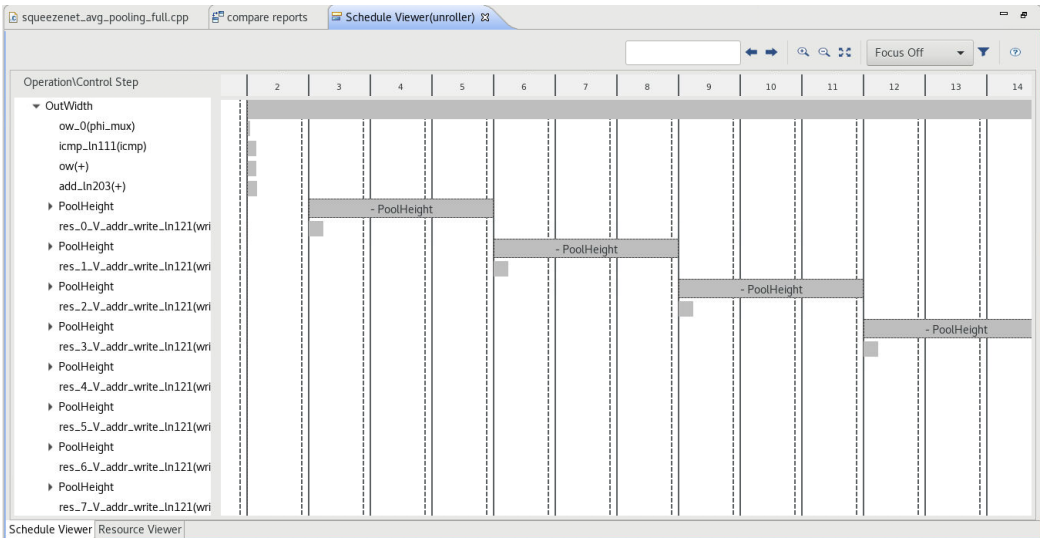


Figure 4.10: Analysis Perspective demonstrating parallelism issues in the `maxpooling` function using the `UNROLL` directive with 64x unroll factor. The `PoolHeight` modules execute sequentially, even though they can execute in parallel

the requested initiation interval. We kept the `data` and `res` arrays partitioned in their third dimension to allow for parallelization. HLS issued a warning that it could not meet the initiation interval, which was a non-issue for our case. This design resulted in a latency o **15.7 thousand clock cycles** as shown in figure 4.9 (the *parallel pipeline* configuration), which translated to a speedup factor of **320x** compared to the configuration that used the `UNROLL` directive and of **333x** compared to the baseline implementation. This result, which is much faster that the projected 64x of the `unroll` configuration can be explained because HLS not only parallelized the `OutFilt` loop, but also flattened and pipelined the `PoolHeight` and `PoolWidth` loops. Analysis Perspective furthermore demonstrated regular parallel scheduling of the operational steps of the block. Figure 4.11 shows a part of the scheduling, where data are read in parallel from the `data` array.

All the `maxpooling` configurations consume minimal amounts of resources. No BRAM modules are used, since all input and output data are accessed from outside of the module. The *pipeline* configuration uses one DSP module and the *pipeline+unroll* configuration uses 64 to calculate the indices of the shifting window, which is consistent with the unroll factor. No other configurations use DSP; in *parallel pipeline*, the window indices are calculated using LUTs. The Flip Flop and LUT usage metrics are shown in figure 4.12. The highest Flip Flop usage is observed with the *unroll* configuration and amounts to less than 1% resource utilization; the *parallel pipeline*, the latest configuration has the highest LUT usage, which translates to around 5.5% of the total FPGA utilization.

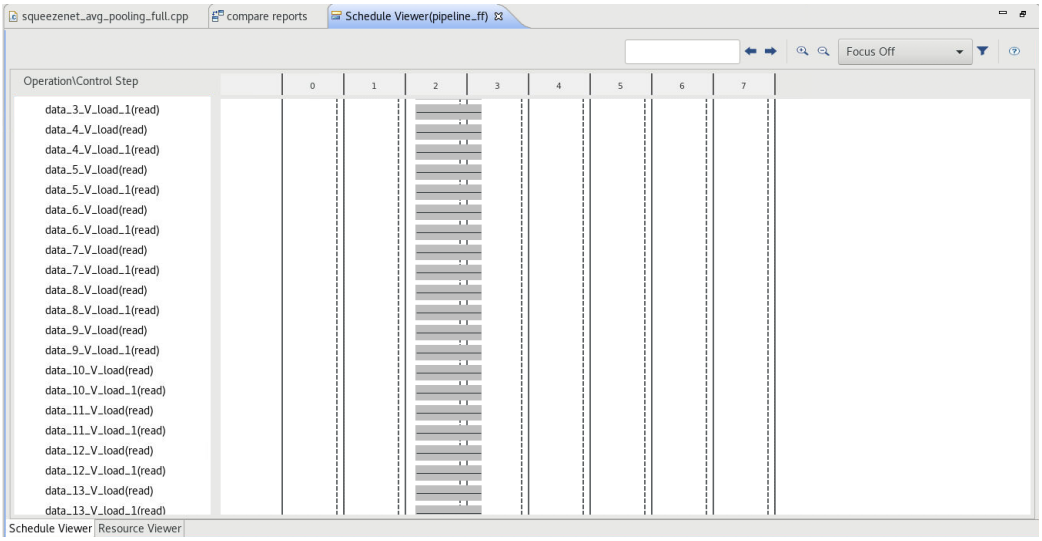
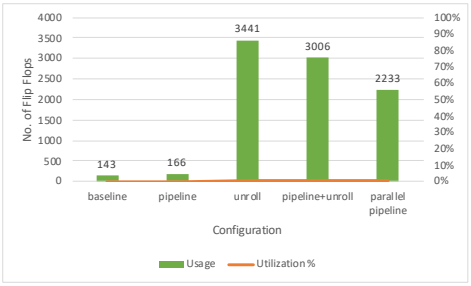
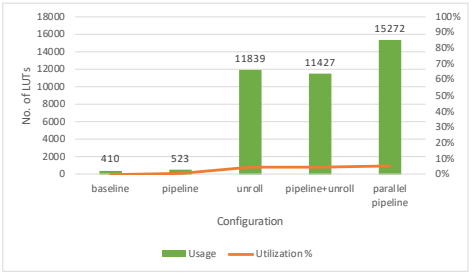


Figure 4.11: Analysis Perspective demonstrating parallel data read operations in the `maxpooling` function.



(a) Flip Flop usage



(b) LUT usage

Figure 4.12: Resource usage of the different `maxpooling` configurations

4.2.3 Lessons learned optimizing `maxpooling`

The previous experiments revealed three important factors to be considered during an optimization process with Vivado HLS:

1. When optimizing a loop nest, it would be preferable to unroll/parallelize the *innermost* loop. Vivado HLS might fail to generate a parallel design, even though it allocated the resources for it, or might not be able to synthesize the design at all.

2. The `UNROLL` directive uses what might be called a "relaxed" approach to parallelism. Throughout the experiments with `maxpooling`, it generated the appropriate hardware (as indicated by the resource usage metrics), but did not parallelize it and did not return any message indicating that it failed.

3. The `PIPELINE` directive might be better suited for fully parallelizing a loop than `UNROLL`, since it not only parallelized the appropriate code portions, but also optimized them by pipelining their body. The `UNROLL` directive might therefore be more appropriate for scenarios where controlled, partial unrolling is needed.

4.3 The global average pooling kernel

In this section, we present *our implementation* and optimization of the global average pooling layer. SqueezeNet uses one (1) instance of this layer in its pipeline, as a second-to-last layer.

4.3.1 The `globalAvgPooling` function definition

hls4ml **does not** have an implementation for the global average layer. Therefore, we use two distinct implementations of our own to compare regarding their optimization potential.

The principle of a global average pooling layer is simple: given an input array **A[a,b,c]**, the layer calculates the average value of the elements in dimensions **a,b**, resulting in a vector of length **c**. Therefore, by definition, the algorithm allows for parallelism in the third dimension of the input and output arrays. In SqueezeNet, the global average pooling layer accepts an array `data[14,14,1000]` and outputs an array `res[1000]`. Since the elements of both the input and output arrays use the 8-bit fixed-point `ap_fixed<8,4>` format, and they are added 196 times, we use a 16-bit fixed-point format (12-bit integer and 4-bit fractional parts) to store the sums.

Our two implementations differ in the manner they handle the division of the averaging operation. The first implementation, shown in Appendix A.8 uses the same loop nest to accumulate the values of the input array, perform the division and store its result to the output array. The second, shown in Appendix A.9, uses

a temporary array to store the intermediate accumulated values and then divides each one on a distinct loop.

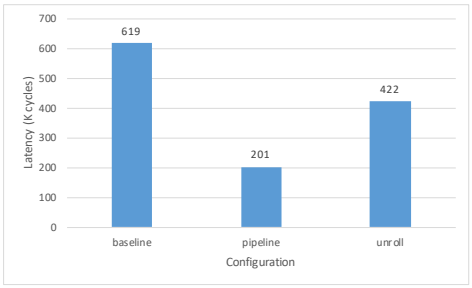
4.3.2 Optimizing the `globalAvgPooling` function

For the first `globalAvgPooling` function implementation (with a single loop nest), we examine a baseline implementation and set optimized configurations: one using the `PIPELINE` directive in the `InWidth` loop and one using `UNROLL` to parallelize the iterations of `OutFilt` loop, with the `data` and `res` arrays partitioned on their third dimension. In order to use the `PIPELINE` directive to enforce parallelism, as we showed in the `maxpooling` function, we apply the directive on the `globalAvgPooling` function itself, since the `OutFilt` loop was the outermost loop of the nest. However, Vivado HLS failed to synthesize the design and produced the same `Stop unrolling` error it did in part 4.1.4.1. This error remained even when we changed the initiation interval value from 1 to 196 (i.e. the iteration count of the `InHeight` and `InWidth` loops). The test results are shown in figure 4.13. Since the design operates on 16-bit variables, it allocates DSPs, but no BRAMs are used (the `temp` instances are stored in Flip Flops). However, although figure 4.13b shows that the *unroll configuration* allocates exactly 1000 DSPs, indicating that (together with the increased usage of Flip Flops and LUTs) the design implements hardware for an unroll factor of 1000, the latency results fail to reflect that; the results using the `PIPELINE` directive are better, with lower latency and resource usage. This comes to further verify our observations that Vivado HLS favors parallelism on the innermost loop and that the `UNROLL` directive might not be effective in some designs.

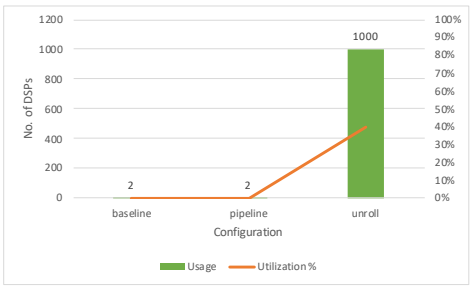
Since the first implementation of `globalAvgPooling` could not be successfully optimized by Vivado HLS, we proceed to evaluate the second implementation (with the division calculate in a distinct loop). Here, the `OutFilt` loop is in the innermost position of the loop nest and we expect that the parallel configurations will show the expected results.

For this set of tests, we repeated the previous ones with `PIPELINE` and `UNROLL` on both `OutFilt` and `OutRes`, but also with `PIPELINE` on `InWidth`. The last test, labelled *parallel pipeline* in the result figures executed successfully as expected, in contrast to the previous implementation. Additionally, although we used `PIPELINE` to parallelize the loop nest, we kept the `UNROLL` directive on the `OutRes` loop.

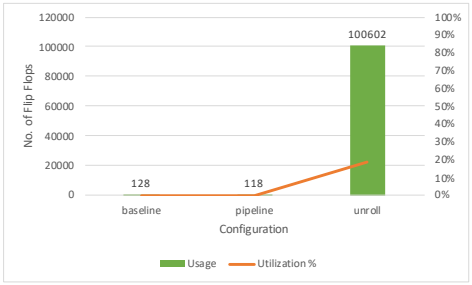
Figure 4.14 shows the results of the tests. Only the *baseline* and *pipeline* configurations use a single BRAM module to store the `temp` array; the subsequent test configurations implement it using Flip Flops, as it is fully partitioned. The latency results, shown in figure 4.14a demonstrate the expected behavior of the *unroll* showing a speedup of **987x** compared to the baseline evaluation, and *parallel pipeline* further achieving a speedup of **1975x**, almost reaching the fastest possible latency of 196 clock cycles. However, both parallel implementations exhibit high LUT utilization of around **44%**.



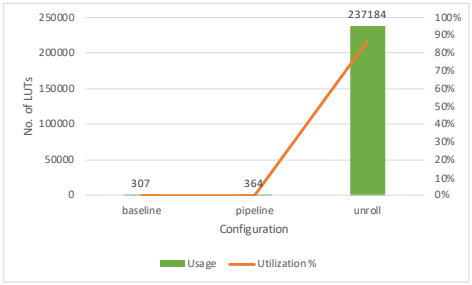
(a) Latency results



(b) DSP usage

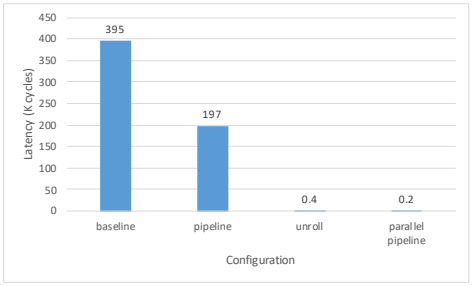


(c) Flip Flop usage

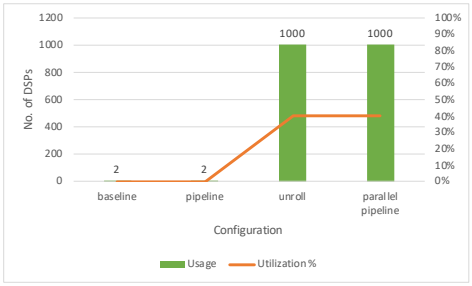


(d) LUT usage

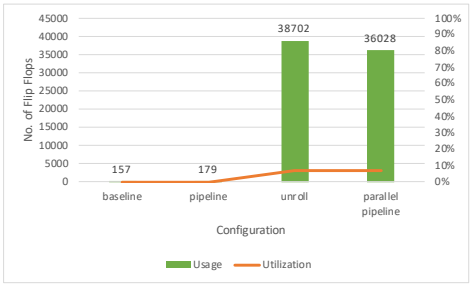
Figure 4.13: Latency and resource usage for different configurations of `globalAvgPooling`



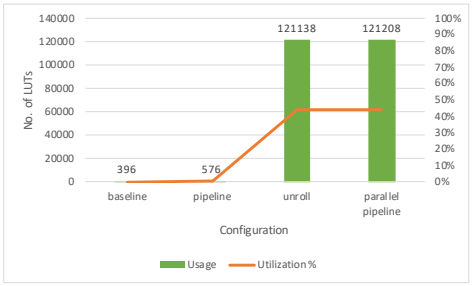
(a) Latency results



(b) DSP usage



(c) Flip Flop usage



(d) LUT usage

Figure 4.14: Latency and resource usage for different configurations of `globalAvgPooling` with array

4.3.3 Lessons learned optimizing `globalAvgPooling`

The *global average pooling* function, although simple to implement, needed to be defined in a particular way for Vivado HLS to be able to optimize it. The optimization remarks we made on `conv2d` and `maxpooling` were further observed here. Additionally, although the function showed great opportunity for parallelism with large speedup factors compared to the unoptimized latency result, it consumed large amounts of available hardware, which should be considered when implemented alongside other network layers.

4.4 The softmax kernel

In this section, we discuss our implementation and optimization of the softmax layer. SqueezeNet uses it as the final layer of its pipeline, to convert the values generated by the network into probabilities for the classification results.

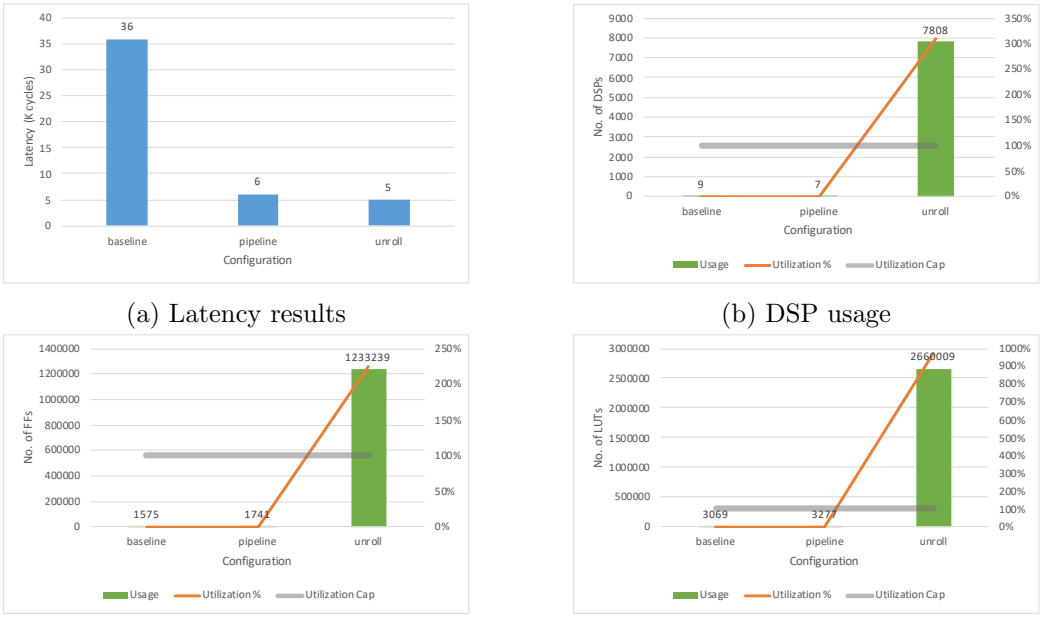
4.4.1 The `softmax` function definition

As with the `globalAvgPooling` function, hls4ml does not provide an implementation for the `softmax` function. Therefore, we present our own. The functionality of a *softmax* layer involves calculating the exponent of the input and dividing it by the sum of the input exponents to produce the output. In SqueezeNet, both input and ouput are unidimensional arrays with **1000** elements. In order to implement the exponential function, Vivado HLS allows the use of `exp()` and `expf()` functions, included in the `hls_math.h` library. However, these functions only accept `double` and `float` variable types, respectively. Hence, although the input of our `softmax` function is `ap_fixed<8,4>`, its output is `float`. Our `softmax` implementation is shown in Appendix A.10. The `InputExp` loop calculates the exponent of each input element, using the `to_float()` method to convert it from `ap_fixed` to `float`. The loop is also used to calculate the sum of the exponents, by which each element is divided and stored in the output array.

4.4.2 Optimizing the `softmax` function

Our testing of `softmax` involved a *baseline* configuration and a *pipeline* and *unroll*, where we used the `PIPELINE` and `UNROLL` directives on both loops respectively, to speed up the design. We also set up a *parallel pipeline* configuration, where we used `PIPELINE` on the function itself to instruct Vivado HLS to parallelize the design and minimize its initiation interval. However, we observed a strange behavior on the last configuration: during synthesis, Vivado HLS reported the following issue (repeated 1000 times):

```
INTERNAL-INFO: never seen llvm instruction 'fexp'(507)
```



(a) Latency results

(b) DSP usage

(c) Flip Flop usage

(d) LUT usage

Figure 4.15: Latency and resource usage for different configurations of `softmax`

We could not find any documentation for this message, as it was not present in any Vivado HLS reference manual, nor discussed in the Xilinx forums. Because of the message header (`INTERNAL-INFO`), it might be a **bug** of the 2019.1 version of Vivado HLS that we use. We observed long synthesis times and the process halted at the interface declaration stage; we stopped synthesis after it had halted for more than a day.

The results of our tests with the *baseline*, *pipeline* and *unroll* configurations are shown in figure 4.15. The *baseline* and *pipeline* configurations consumend two BRAM modules to store the `exp_data` array; the *unroll* configuration did not require any memory modules, since the array was fully partitioned. As expected, the *pipeline* configuration has a latency of around 6000 clock cycles: around 5000 for the `InputExp` loop and 1000 for the `OutExp` loop, each of which had 1000 iterations. This resulted in a speedup factor of **6x** compared to the *baseline* configuration, with only moderately larger resource usage. Conversely, the *unroll* configuration reached a latency of 94 clock cycles and a speedup factor of **383x** compared to *baseline*. However, the configuration used an exorbitant amount of DSPs, Flip Flops and LUTs in order to store the fully partitioned `exp_data` array (32 Flip Flops per element for 1000 elements) and perform the floating-point operations, as reported by Vivado HLS. The utilization was quite larger than the total available FPGA resources for both resource types, as shown in figures 4.15c and 4.15d.

4.4.3 Lessons learned optimizing `softmax`

Our experiments on optimizing the `softmax` function revealed that its algorithm presents great opportunities for parallelism; its total latency can be potentially drop below 100 clock cycles. However, this comes with unfeasibly large resource utilization requirements, which stem from the use of floating-point arithmetic. The `PIPELINE` directive on the other hand achieves a sixfold decrease in latency compared to the unoptimized design without allocating much more resources, which would make it the preferred optimization method.

4.5 Results overview

In this chapter, we presented our experiments and testing towards optimizing the different layer types of SqueezeNet. We evaluated the impact of different directives, namely the `UNROLL` for parallelism and `PIPELINE` to both pipeline and parallelize each design. We pointed out on which instances it would be preferred to use one over the other and observed situations where Vivado HLS struggled to produce the appropriate synthesized design, or failed to complete the synthesis process altogether.

In summary, we achieved the following results: we managed to reduce the latency of the convolutional kernel from 1.5 seconds to 2.5 ms, reaching a speedup of **600x**; for the max pooling kernel, we achieved a latency drop from 26 ms to 78 μs and **333x** speedup. For the global average pooling kernel, we reduced the latency from 3 ms to 995 ns and achieved a speedup factor of **1987x**. Finally, we dropped the latency of the softmax kernel from 180 μs to 470 ns, which translates to **383x** speedup.

In the next chapter, we will use the derived optimization methods to optimize the *NewFire modules* of SqueezeNet. We will also introduce task-level pipelining (dataflow) inside the FPGA that implements each module to further optimize the design.

Chapter 5

Optimizing the Fire Modules

In chapter 3, we modified the core block of SqueezeNet, the Fire module, and defined two versions of the New Fire module:

1. **NewFire** that contains one (1) "squeeze" and two (2) "expand" convolutional layers. These three (3) layers use the convolutional kernel.

2. **NewFirePool** that also incorporates a max pooling layer (`maxpooling` kernel).

As shown in figure 3.8b, SqueezeNet employs five (5) instances of the NewFire and two (2) of NewFirePool. It additionally includes a set of layers in its beginning (**Initial layers**) and its ending (**Final layers**). We will use these modules as building blocks to realize SqueezeNet in our FPGA cluster: **in this thesis, we do not split individual modules across multiple FPGAs**.

In this chapter, we try to build efficient single-FPGA implementations of fire modules, applying the techniques of Chapter 4. In particular, we use the parameters of *fire 1 instance* to optimize NewFire logic and *fire 2 instance* to optimize NewFirePool logic. In the next chapter (6) we will apply these techniques to all seven (7) instances of fire modules in SqueezeNet.

As described in Section 5.4, in this chapter we will obtain **optimized designs with 900K cycles (4.5ms) latency, compared with the 119M cycles in the unoptimized ones, featuring an initiation interval of 3.3ms**.

5.1 Overview of NewFire module

The NewFire module was first outlined in Figure 3.6. It consists of layers `expand1x1` and `expand3x3` (which logically run in parallel), followed by a `concatenate3d` function and a `squeeze` function[1]. The hls4ml code of NewFire modules succeeded

[1]Listing in Appendix A.11 shows the head of `fire` function. In hls4ml, the NewFire module uses one definition of the `conv2d` function, which is called three times for its three main layers. We instead opted for three separate definitions, in order to optimize each layer separately.

Layer	# Mops	# Params
expand1x1	6.6	1088
expand3x3	58	9280
squeeze	12.9	2064
Total	**77.5**	**12432**

Table 5.1: The number of operations and parameters per layer in New Fire 1

every `conv2d` function with a RELU one. We decided to *merge the convolutional and ReLU functions together*, simplifying the code.
Intra-fire communication buffers: NewFire implements the buffers (arrays) to connect its layers. In particular, the the outputs of `expand1x1` and `expand3x3` are inputs to `concatenate3d` using arrays **concat_1 concat_2** respectively; the output of the concatenation function is input to `squeeze` using array **concat**.

The parameters and the number of MOPs of the NewFire 1 on a per-layer basis are shown in table 5.1. Note that the operations include the distinct multiplications and additions of the convolutional part and the comparisons of the ReLU part.

5.2 Optimizing the NewFire 1 module

In this section, we present the following implementations of NewFire 1 module:

1. A *baseline* implementation, *without any optimizations* (Subsection 5.2.1),
2. an implementation that evaluates `UNROLL` and `PIPELINE` directives to achieve parallelism inside each layer (Subsection 5.2.2),
3. an implementation using the `DATAFLOW` directive in order to (partially) overlap the execution of different layers and to pipeline their execution, thus reducing the initiation interval (II) (Subsection 5.2.3).

In addition to the aforementioned implementations, we also tested an implementation with pipelining for the innermost loop of each NewFire 1 function, so that they begin a new iteration every clock cycle. This configuration achieved a speedup of **3x** compared to the baseline, without any significant raise in resource utilization. However, the important latency improvements are observed with loop unrolling and multiple engines of the aforementioned implementations and therefore we do not refer to this configuration in the following sections.

5.2.1 Baseline, non-optimized implementation

In this configuration, we implement the NewFire 1 module as-is, without any optimization directives. The results will be used as baseline comparison for the subsequent configurations.

Element	BRAM usage
`ex1_weights`	1
`ex3_weights`	6
`sq_weights`	1
`concat_1`	98
`concat_2`	98
`concat`	196
Total	**400**

Table 5.2: BRAM usage per memory element for the NewFire 1 baseline implementation

Without any improvements, the Newfire 1 module has a **latency of 120 million clock cycles (0.6sec)**. The utilization results are quite modest, except for the BRAM utilization that is unexpectedly high: it uses 400 BRAMs (22% utilization), 1332 Flip Flops (0.2% utilization) and 3849 LUTs (1.4% utilization). Table 5.2 shows the BRAM usage per memory element. The output arrays from the expand functions (`concat_1` and `concat_2`), as well as the output array from the concatenation function (`concat`) show the highest utilization.

5.2.2 Comparing `UNROLL` and `PIPELINE` directives for per-layer operation parallelism

In this part, we compare the `UNROLL` and `PIPELINE` directives as different ways to parallelize the innermost loop operations of each convolutional layer of the NewFire 1 module. In this way, we achieve a module latency of **902 thousand clock cycles** and a speedup factor of **129x** compared to the baseline approach.

Since the baseline implementation showed **high BRAM utilization** caused by the arrays involved with the concatenation function, **we remove it for this and subsequent implementations and have the expand kernels write to the `concat` array directly** (each expand function writes on a separate part of the array). In this way, we **both minimize the BRAM utilization and latency** of the NewFire 1 module. The diagram of this design is shown in figure 5.1.

5.2.2.1 Parallelism with `UNROLL`

For this configuration, we apply the `UNROLL` directive on all the innermost loops of each convolutional function (i.e. the initialization, convolutional and ReLU loops), while partitioning the output, weights and biases arrays to provide the throughput needed. We can observe the latency results in figure 5.3a and the BRAM, Flip Flop and LUT usage in figure 5.3b, for the *unroll* configuration.

In table 5.3, we can observe the latency of each loop iteration of the squeeze convolutional module, as reported by HLS for the *unroll* configuration. Each loop with a name beginning with a dash (-) is an outer loop and each loop with name

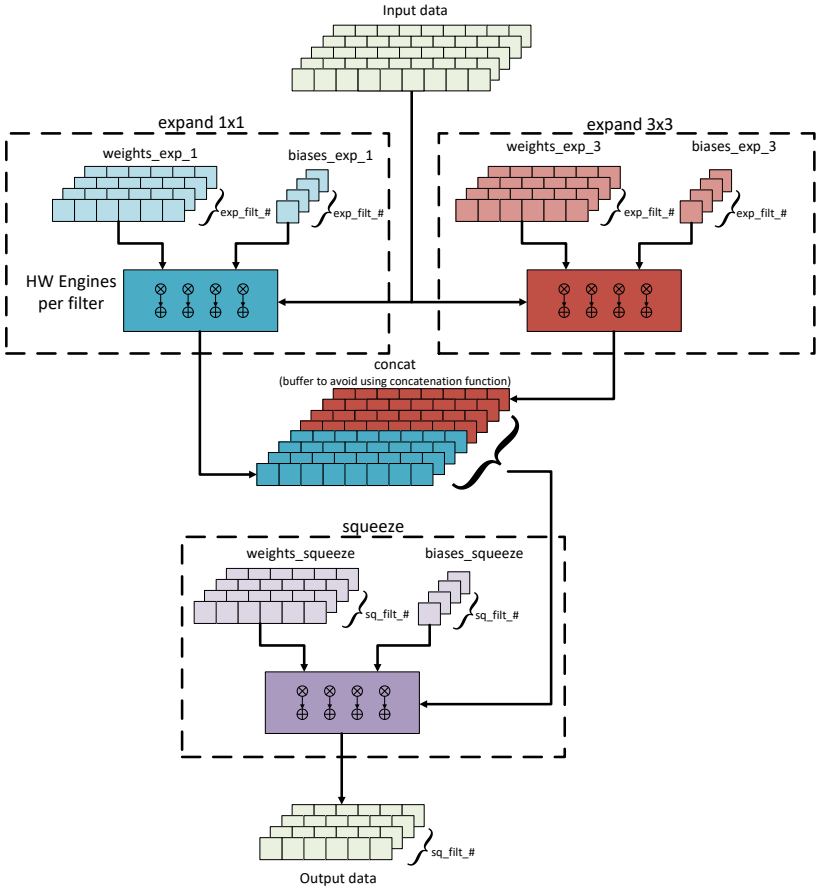


Figure 5.1: The new Fire module without the concatenation function

Loop Name	Total Latency	Iteration Latency	Initiation Interval	Trip Count
-InitOutHeight	3248	58	-	56
+InitOutWidth	56	1	-	56
-ConvOutHeight	1218672	21762	-	56
+ConvChan	21760	170	-	128
++ConvOutWidth	168	3	-	56
-ReluOutHeight	53424	954	-	56
+ReluOutWidth	952	17	-	56

Table 5.3: Latency analysis of the squeeze convolution of new Fire 1 module

beginning with a cross (+) is a nested loop; the number of crosses denote the nesting level of the loop. In this case some loops are missing: The innermost loops of the `Init`, `Conv` and `Relu` nests have been fully unrolled and are therefore not declared in the table and two of the loops in the `Conv` nest that traverse the height and width of the squeeze filters have a loop iteration count of 1 and are therefore removed by HLS.

The table contains the iteration latency, trip count and total latency of each loop, where $n_{iter} * n_{trip} = n_{total}$. However, the loops are not pipelined, hence the *initiation interval* column does not contain any values. Additionally, we can see that the iteration latency of the `ReluOutWidth` loop is 17. This is concerning, since the loop body contains the fully unrolled `ReluOutFilt` loop that only carries out a zero comparison and a memory write, therefore should be much faster than 17 clock cycles. We next use the *Schedule Viewer* of Vivado HLS in order to assess the issue.

In figure 5.2, we can see that the ReLU loop write accesses are performed in sequential fashion and not in parallel, even though we had applied both the `UNROLL` directive on a loop without inter-loop dependencies and the `ARRAY_PARTITION` directive on the appropriate array dimension[2].

5.2.2.2 Parallelism with `PIPELINE` directive

For this configuration we add the `PIPELINE` directive to the **second-innermost** loop of each loop nest, together with the `ARRAY_PARTITION` directive, to allow for parallel array access. In figure 5.3a, we can see the resulting latency difference between the previous *unroll* configuration and the current, *parallel pipeline* configuration. The latter configuration demonstrates a **3.5x** speedup in comparison to the previous one.

Table 5.4 presents the latency results of the loops in the squeeze module; HLS optimized the loops by flattening them and achieved an initiation interval of 1 clock

[2]This behavior is the same we observed in the previous chapter, where the `UNROLL` directive produced a sequential design, even though there were no restrictions for parallelism.

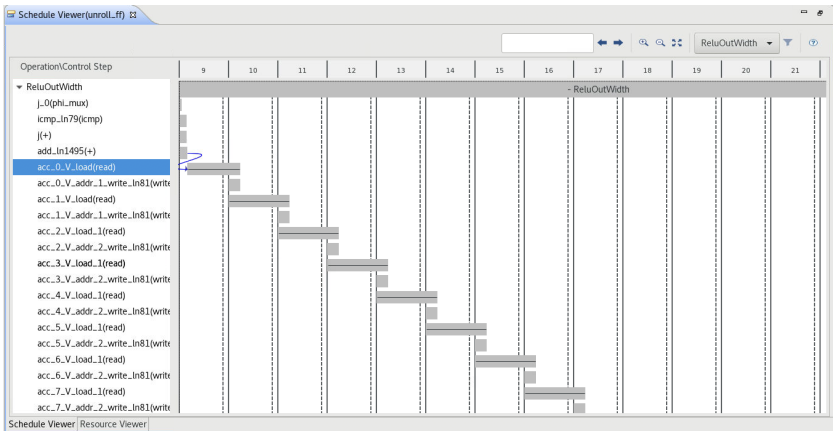
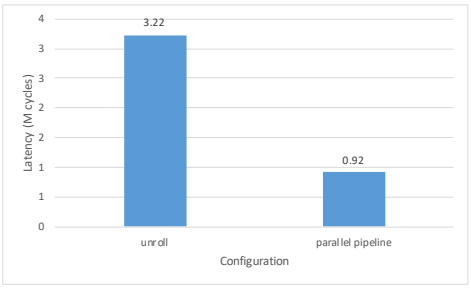
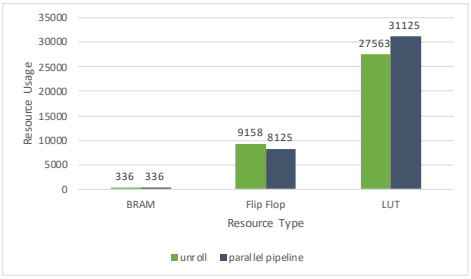


Figure 5.2: Schedule Viewer screenshot demonstrating sequential write accesses on the ReLU loop



(a) Latency results



(b) Resource usage

Figure 5.3: Latency and resource usage comparison of parallelizing the first New-Fire module using `UNROLL` and `PIPELINE`

cycle for the innermost loop, indicating that the `PIPELINE` directive accomplished its purpose successfully. Moreover, Schedule Viewer in figure 5.4 demonstrates that the innermost loop of the ReLU function is successfully parallelized.

Figure 5.3b presents a comparison of the resource usage between the unrolled and pipelined designs. The results show that the `PIPELINE` directive only has increased usage of LUTs; it uses the same number of BRAMs and less Flip Flops compared to `UNROLL`. These results, combined with the quite lower latency, highlight our use of the `PIPELINE` directive as the preferred method to introduce parallelism to the design.

We will use the methodology we presented here as the default for the rest of the new Fire modules, as it was proven much more successful in relation to latency optimization, compared to the previous one. However, we should note that the results of using `UNROLL` were not the ones we expected; HLS failed to deliver a truly parallelized design in regards to the ReLU loop nest, even though there were no dependency or hardware limitations, as was further proven by using the `PIPELINE`

Loop Name	Total Latency	Iteration Latency	Initiation Interval	Trip Count
-InitHeight_Width	3136	1	1	56
-ConvHeight_Chan_Width	401409	3	1	56
-ReluHeight_Width	3136	2	1	56

Table 5.4: Latency analysis of the squeeze convolution of new Fire 1 module, with parallelism through the `PIPELINE` directive

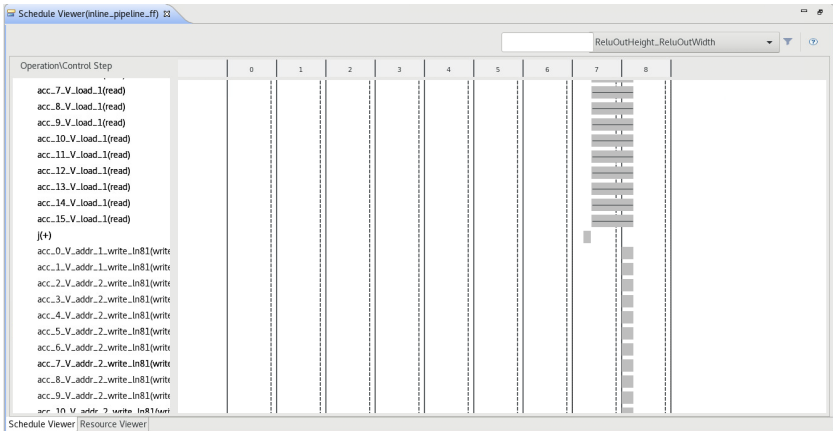


Figure 5.4: Schedule Viewer screenshot demonstrating parallel load and write accesses of ReLU, as a result of the `PIPELINE` directive

directive (that achieved the desired latency result).

5.2.3 Dataflow implementation: inter-kernel pipeline and concurrent kernel execution

In this section we introduce the `DATAFLOW` directive that allows for *computation-communication overlap as well as for pipelining across layers located in the same FPGA*, first discussed in section 3.2. Therefore, this configuration will:

1. decrease the total latency of the NewFire 1 module by overlapping the execution between the different layers and
2. increase the throughput (reduce the initiation interval) by introducing task-level pipeline.

As discussed in Subsection 2.2.1, the `DATAFLOW` directive has some strict requirements related to the structure and the data access pattern of the source code, in order to perform as expected. We therefore perform the following modifications to the code of the NewFire 1 module:

1. We *duplicate the input data* of the NewFire module into two distinctive arrays of the same size. We achieve this with the `duplicator` function shown in Listing 5.1, a simple nest of three loops that accepts input parameters through a configuration struct (in the same fashion as the other functions of hls4ml) and outputs to two arrays, `ex1_data` and `ex3_data`. These arrays are defined in the body of the updated `fire` function, as shown in Appendix A.12. Furthermore, we add the `PIPELINE` directive in the `DupChan` loop, in order to increase the throughput of `duplicator`.
2. We *reintroduce the concatenation function to the design*, since having the two expand functions write to the same array *would not allow for concurrent execution*. The function, now called `concat3d`, accepts the output of the expand functions (the `ex1_res` and `ex3_res` arrays) and merges them into the `concat` array, as shown in Appendix A.12. We use the same directive enhancements as before, i.e. adding the `PIPELINE` directive to the innermost loop of `concat3d` to increase its throughput. We also keep the `INLINE` directive at the top module of the design.
3. The `DATAFLOW` directive only performs computation-communication overlap between functions that communicate through data streams. We therefore instruct Vivado HLS to implement the `ex1_res` and `ex3_res` arrays as FIFOs by using the `STREAM` directive.
4. Since each convolutional function uses its output array for both reading and writing in the previous configurations, we instantiate a local array for each function to save the results to. We then take advantage of the sequential access patter the ReLU part has on the output array to stream the data out of each convolutional function.

5. We also convert the top-level data I/O ports to use the AXI-Stream interface.

These modifications are shown in figure 5.5

Listing 5.1: The `duplicator` function

```
1    template<class data_T, typename ex1_CONFIG_T>
2  void duplicator(data_T data[ex1_in_height][ex1_in_width][
       ex1_n_chan],
3         data_T ex1_data[ex1_in_height][ex1_in_width][ex1_n_chan],
4       data_T ex3_data[ex1_in_height][ex1_in_width][ex1_n_chan]){
5
6   DupHeight: for(int i = 0; i < ex1_in_height; i++){
7    DupWidth: for(int j = 0; j < ex1_in_width; j++){
8     DupChan: for(int k = 0; k < ex1_n_chan; k++){
9      ex1_data[i][j][k] = data[i][j][k];
10     ex3_data[i][j][k] = data[i][j][k];
11    }
12   }
13  }
14 }
```

In figure 5.6a, we can view the latency and throughput results of this configuration, compared to the previous, *parallel pipeline* configuration. Although the expand functions now execute in parallel, the total latency of the design (0.92 million clock cycles) is less than the latency sum of the `duplicator`, `expand3x3`, `concat3d` and `squeeze` functions, seen in table 5.5. This indicates that Vivado HLS successfully *overlapped the execution of the pipelined functions*, since the majority of the functions in the `DATAFLOW` regions *use FIFO channels to communicate*. Moreover, the metric that shows the impact of the `DATAFLOW` directive is the throughput of the synthesized design, reported in the synthesis report as its initiation interval: under this configuration it resulted to **655 thousand clock cycles**, which *matches the latency of the slowest function of the design (i.e. the `expand3x3`)* plus one clock cycle. Therefore, this configuration is both faster than all the previous we demonstrated, *both in total latency and throughput*.

The impact of the implementation of the `duplicator` and `concat3d` functions, as well as the additional memory required by the convolutional layers are demonstrated in figure 5.6b. The *dataflow* configuration consumes significantly more resources than the *parallel pipeline*; however, in regards to the total available resources in the FPGA, the BRAM utilization is **33.8%**, followed by the LUT utilization at **18.2%** and the Flip Flop utilization at only **2.1%**.

5.2.4 Lessons learned optimizing the NewFire 1 module

In this section, we presented the basic implementation of the New Fire module as a set of three convolutional and one concatenation function, and proceeded to

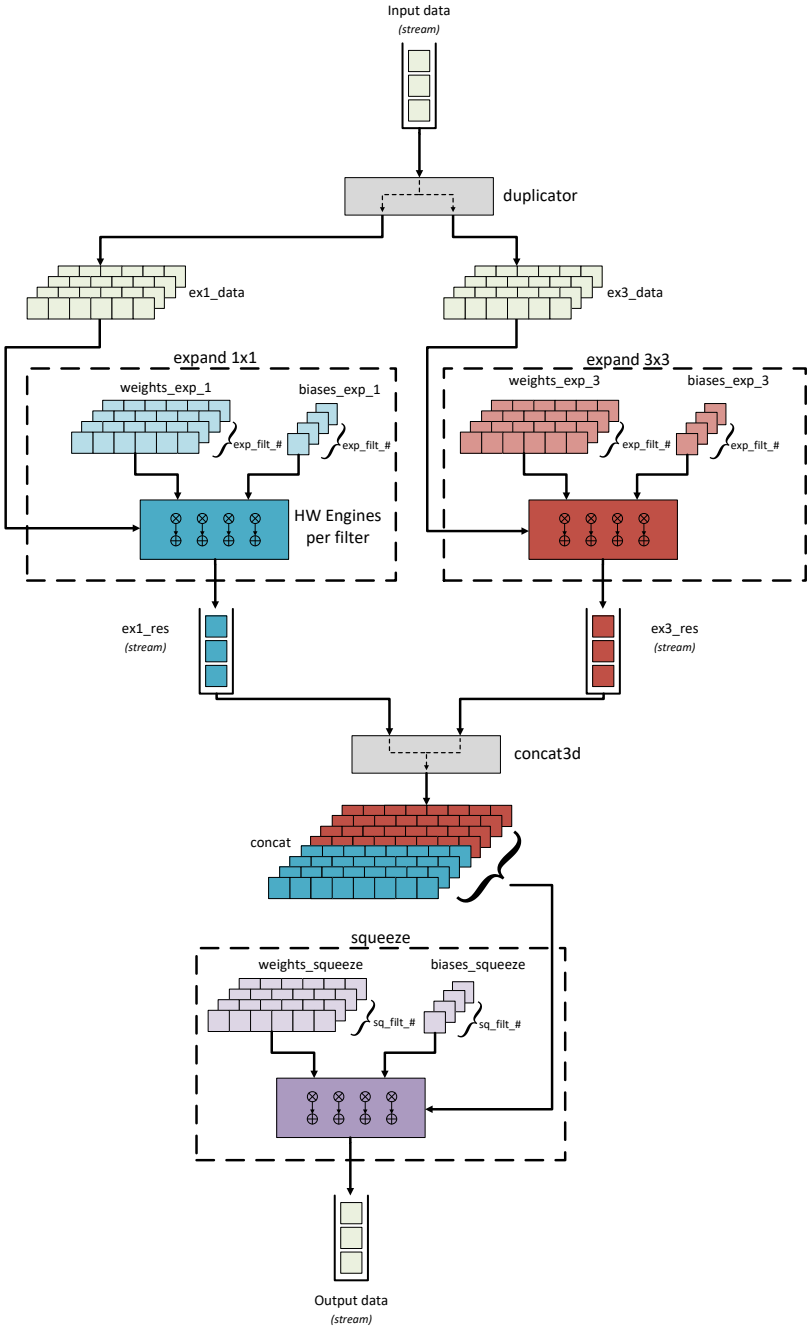
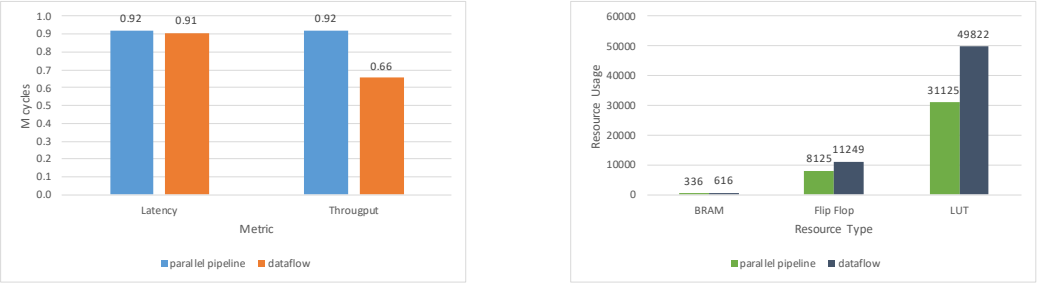


Figure 5.5: The new Fire module with the configuration for dataflow

Function Name	**Latency** (clock cycles)
`duplicator`	50178
`expand1x1`	254027
`expand3x3`	655435
`concat3d`	401410
`squeeze`	454732
Total	1815782

Table 5.5: Per function latency results for the first NewFire module with *dataflow* configuration



(a) Latency and throughput

(b) Resource usage

Figure 5.6: Latency, throughput and resource usage comparison between the *parallel pipeline* and *dataflow* configurations of the first NewFire module

Layer	# Mops	# Params
expand1x1	6.6	1088
expand3x3	58	9280
maxpooling	0.9	–
squeeze	3.2	4128
Total	**68.7**	**14496**

Table 5.6: The number of operations and parameters per layer of the *NewFirePool 2 module*

optimize its first instance by both using the *appropriate directives* and *modifying its code* to minimize resource utilization (by removing the concatenation function) and to make compatible with the `DATAFLOW` directive (by reintroducing the concatenation function, adding the `duplicator` function and converting the module inputs and outputs, as well as the convolutional function outputs to streaming interfaces). In the following section, we will present the optimization of the *NewFirePool module*.

5.3 Optimizing the NewFirePool 2 module

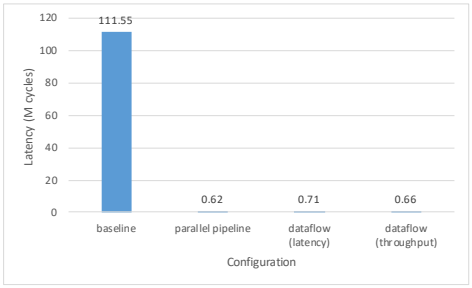
The NewFirePool introduces a pooling layer between the concatenated output of the two expand layers and the input of the squeeze layer. Since the tests of the previous section on the New Fire 1 module showed that the best latency results were achieved with the *parallel pipeline* and *dataflow* configurations, we will focus on them for the NewFirePool module. For our testing, we selected the second New Fire module in the SqueezeNet pipeline (NewFirePool 2 module). The parameters and number of operations of this module are shown in table 5.6.

5.3.1 Implementation results

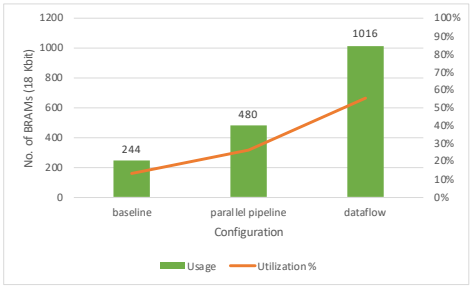
We tested three different configurations: the *baseline*, with **the concatenation function removed** and without any other optimizations, the *parallel pipeline*, where we used the `PIPELINE` directive to parallelize the operations of each convolutional function and the pooling function in the same way as the previous section and the `maxpooling` function as described in Section 4.2.

The test results are shown in figure 5.7. Contrary to the previous testing with the *NewFire module*, the *dataflow* configuration presents both slower latency and throughput results compared to the *parallel pipeline* configuration, which achieves a speedup factor of around **18.6x**, compared to *baseline*. Additionally, the *dataflow* configuration demonstrates high resource utilization, reaching almost **60%** of the total BRAM modules and **50%** of the total LUTs available in the FPGA.

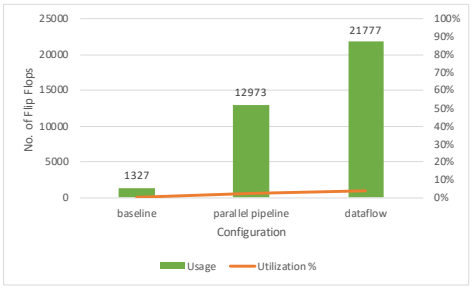
These latency metrics come as a result of the architectural constraints of the *dataflow* configuration. Table 5.7 shows a comparison of the latency results of the



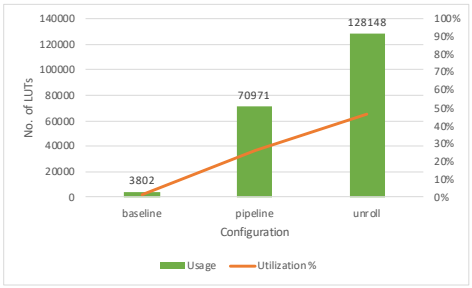
(a) Latency and throughput



(b) BRAM usage and utilization



(c) Flip Flop usage and utilization



(d) LUT usage and utilization

Figure 5.7: Latency, throughput and resource usage comparison between the different configurations of the *NewFirePool 2 module*

subfunctions of the NewFirePool 2 module between the two configurations. Even though the `maxpooling` function achieves the same latency for both configurations, the latency results of the convolutional functions are quite slower in the *dataflow* configuration. This happens because even though the convolutional part of the function is parallelized, the ReLU part is sequential, in order to export the output data using a streaming channel. In contrast, both the convolutional and ReLU parts are parallelized in the *parallel pipeline* configuration, radically improving the throughput of each convolutional function. Nevertheless, one can observe the impact of the `DATAFLOW` directive in regards to inter-function computation-communication overlap: the total design latency is around 705 thousand clock cycles, which is *less than the sum of the latencies of the convolutional and pooling functions* (which is around 785 thousand clock cycles).

We attempted to improve the *dataflow* configuration by parallelizing the ReLU parts of the expand functions and changing their output array implementation from streaming to the ping pong buffer. We expected a design with higher total latency, since Vivado HLS would not be able to overlap inter-function communication with computation (because the data are not transferred between functions in streaming channels); however, Vivado HLS failed to complete the synthesis process as it exhausted the host memory and got terminated, in the same fashion as in Subsection 5.2.3.

Function	***parallel pipeline***	***dataflow***
duplicator	-	50178
expand1x1	56456	254027
expand3x3	457864	655435
concat3d	-	401410
maxpooling	3922	3922
squeeze	101927	126235

Table 5.7: Subfunction latency result comparison of the *parallel pipeline* and *dataflow* configurations, in clock cycles

5.3.2 Lessons learned optimizing the NewFirePool 2 module

Examining the results of NewFirePool 2 module show that there is no optimization method that can be used under every scenario and deliver the best possible results. Therefore, we will consider both the *parallel pipeline* and *dataflow* configurations as feasible optimization approaches for the different parts of SqueezeNet. In the following chapter, we implement these optimizations in the separate New Fire modules of SqueezeNet (as well as the initial and final layers) and will map them to different FPGAs in the cluster.

5.4 Results Overview

In this chapter, we managed to synthesize the first two fire (multi-layer) modules of SqueezeNet for the targeted FPGA, **reducing the latency of NewFire 1 module from 0.6s to 4.5ms (3.2ms initiation interval); for NewFirePool 2, these numbers are 3.5ms and 3.2ms**. As we have seven (7) fire modules in SqueezeNet, these results give us confidence that we can achieve a network latency below 30ms, as targeted in Section 1.4.

The **maximum resource utilization for NewFire 1 is around 34% for BRAMs and 20% for LUTs; the corresponding numbers for NewFire-Pool 2 are 56% for BRAM and 47% for LUTs**[3]. In the next chapter, we will use the derived guidelines to implement all remaining modules in the SqueezeNet network, and distribute them into a small number of FPGAs.

[3]Higher than NewFire1 as NewFirePool includes the max pooling function.)

Chapter 6

Implementing SqueezeNet on multiple FPGAs

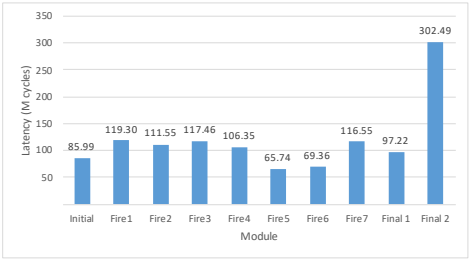
In this chapter, we synthesize *all modules of SqueezeNet* using the directives examined in previous chapters. SqueezeNet consists of seven (7) *New Fire modules*, together with a set of layers at the beginning of the network (the *Initial layers*), and a set at its end (the *Final layers*).

Then, in Section 6.6, we map the modules on a set of FPGAs. Our mapping strategy is to use as few FPGAs as possible to accommodate all modules of SqueezeNet, in their best-performing configuration. Our final solution achieves that with five FPGAs. We discuss our overall results in Subsection 6.6.1.

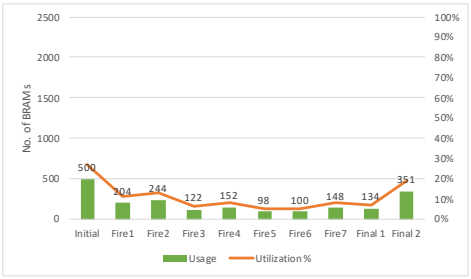
6.1 Baseline implementation

For the first set of tests, we synthesized the SqueezeNet parts without any optimizations, apart from the removal of the concatenation layers. For the global average pooling layer, we chose the second implementation described in Section 4.3, since it yielded the best results between the two implementations. The latency and resource utilization results are shown in figure 6.1. Only the *Final 2* module consumes 14 DSPs for the floating point calculations of the softmax layer; no other module uses any. Figure 6.1a shows that the *Final 2* module also has the longest latency (around 2.5x slower than the second-slowest, the *NewFire 1* module). This is expected, as it contains the largest convolutional layer with more than **200 million** operations. The resource utilization however is mostly balanced between the different modules, with only the *Initial* and *Final 2* modules having an increased utilization of BRAMs, which nevertheless was less that 30%.

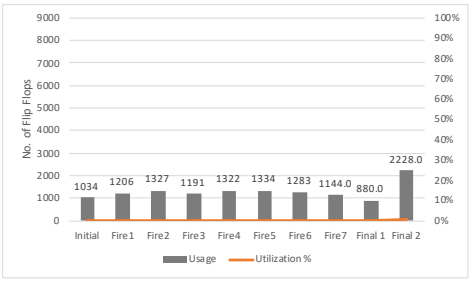
It should be noted that the resource utilization results for the *baseline* implementation **do not take into account the memory used for storing the input and output arrays of each module**. These memory modules are synthesized outside Vivado HLS and can either match the size of the array for a simple design with throughput equal to the total latency of the SqueezeNet modules, or



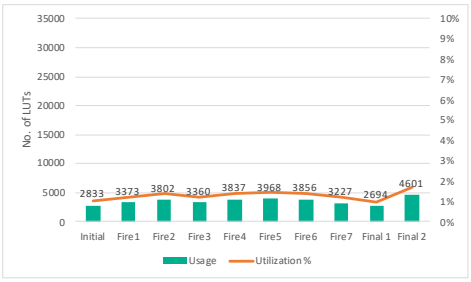
(a) Latency results



(b) BRAM usage and utilization



(c) Flip Flop usage and utilization



(d) LUT usage and utilization

Figure 6.1: Latency and resource utilization results for the *baseline* implementation of the modules of SqueezeNet

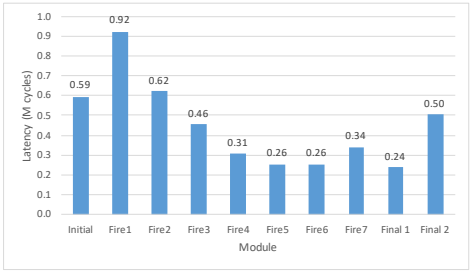
allocate twice the array size to implement a ping pong buffer that will allow for module-level pipeline.

6.2 Parallezing the layers with PIPELINE directive

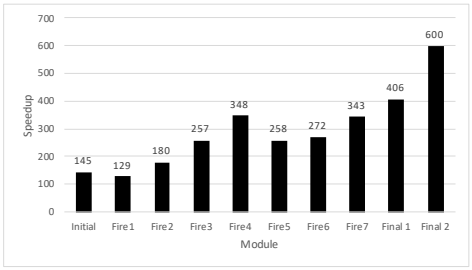
For this implementation we used the `PIPELINE` directive to parallelize the layers of SqueezeNet. For the convolutional layers, we parallelized all three loop nests (i.e. the initialization, convolutional and ReLU loop nests), as well as the max-pooling layers, with the same method we used in 4.2. However, we kept both the global average pooling and softmax layers sequential, with only pipelined loop iterations, since we observed high resource utilization in their parallel configurations (described in 4.3 and 4.4).

Figure 6.2a demonstrates the latency results of the SqueezeNet modules. For the `expand3x3` function of the *NewFire 5*, *NewFire 6*, *NewFire 7* and *Final 1* modules and the convolutional function of the *Final 2* module, Vivado HLS failed to achieve an initiation interval value of 1, with the following message (here truncated):

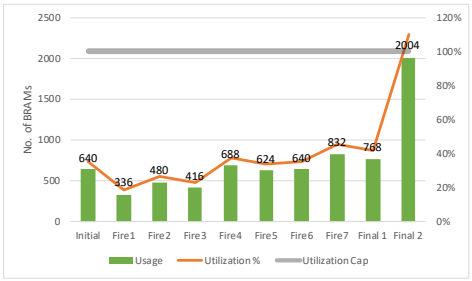
```
Unable to schedule 'load' operation on array 'data_V' due to limited
 memory ports.  Please consider using a memory core with more ports
                or partitioning the array 'data_V'.
```



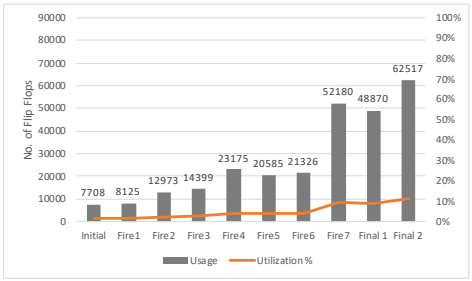
(a) Latency results



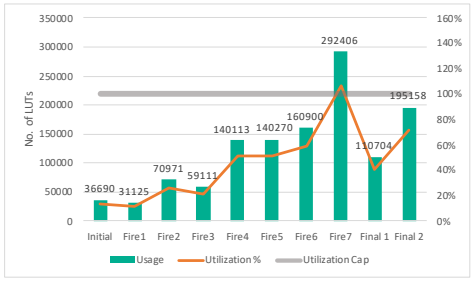
(b) Speedup compared to baseline



(c) BRAM usage and utilization



(d) Flip Flop usage and utilization



(e) LUT usage and utilization

Figure 6.2: Latency and resource utilization results for the *parallel pipeline* implementation of the modules of SqueezeNet

The `data_V` array is the input array of the convolutional functions, which would normally have only one element accessed in the pipelined loop and thus does not need to be partitioned. However, the *Analysis Perspective* of the convolutional loop nest revealed the following:

- Regarding the designs generated by Vivado HLS for the *NewFire 5*, *NewFire 6*, *NewFire 7* and *Final 1* modules, the BRAMs holding the `data` arrays for each design only drove **64** multiplication modules per access. Therefore the *New Fire 5* module required **three** `data` memory accesses and the others required **four**; since each BRAM has two ports for reading, the initiation interval value was raised to 2 clock cycles to accommodate the accesses for a total latency of 5 clock cycles per loop iteration.

- The `data` BRAM for the convolution in the *Final 2* module drives 200 multiplication units; therefore the design required an initiation interval of 3 clock cycles to accommodate **five** memory accesses and a total loop iteration latency of 6 clock cycles.

- This was not the case with the `expand1x1` functions in the aforementioned modules (except for the *Final 2* module that does not contain such function altogether): even though the parallelism factor matched that of the respective `expand3x3` function, the two ports of the `data` BRAMs for each module drove all the multiplication modules, thus achieving an initiation interval of 1 clock cycle and a latency of 4 clock cycles per loop iteration.

- Interestingly enough, Vivado HLS implemented a more "relaxed" scheduling method for the first convolutional function of the *Initial* module: it reached a loop iteration latency of 5 clock cycles; however, it still achieved an initiation interval value of 1 clock cycle and only required one `data` BRAM access per loop iteration.

- Since the array each `squeeze` function reads from is fully partitioned in its third dimension (in order for the expand/pooling functions that precede each `squeeze` to be parallelized), Vivado HLS implements a series of multiplexers that select the appropriate partition for each loop iteration to read from. This process allows for the total loop iteration latency to drop to 3 clock cycles.

In an attempt to remedy the multiple-data-read issue, we first fully partitioned the `data` array, to no avail. We also set its port to different interfaces by using the `INTERFACE` directive, but did not observe any changes. We lastly used the `config_schedule` command to disable its `relax_ii_for_timing`, which allows Vivado HLS to meet the requested clock period by increasing the initiation interval when needed; this however also failed to yield the desired result. Given the fact that Vivado HLS managed to produce the appropriate design for the `expand1x1` functions, we believe that this behavior might be an issue of the scheduling engine

of the tool; should this not be the case (i.e. there is an actual constraint that forces such scheduling method), Vivado HLS should report it with a more appropriate message.

Nevertheless, the speedup compared to the baseline implementation reached values as high as **600x** for the *Final 2* module, as shown in figure 6.2b. Even though the speedup value matches the one we documented in the `conv2d` optimization in Section 4.1, the speedup of the convolutional layer itself is **1000x**, since it is parallelized with the much more effective `PIPELINE` directive; the speedup factor is however lowered by the global average pooling function that is not parallelized and only achieves a speedup factor of **2x**. Modules placed later in the SqueezeNet pipeline achieve higher speedup than the earlier ones; this is attributed to their *higher parallelization factors*.

Most modules demonstrate moderate resource utilization. No module uses more than **10%** of the Flip Flops available in the FPGA, and less than half the available BRAM modules are used, with the **exception of *Final 2***, which requires more memory than is available in the FPGA. It is almost exclusively allocated by the convolutional function of the module, which requires **2000** BRAM modules: 1000 for the partitioned output array and 1000 for the partitioned weights array. Additionally, the *Fire 7* module requires **106%** of the total LUTs available. These are mostly required by the module's `squeeze` function, which implements 64 multiplexers to read from the partitioned array. Each of them requires **2693**, for a total of **172 K** LUTs.

Even though the parallel implementation using the `PIPELINE` directive resulted in an impressive drop in latency for the modules of SqueezeNet, the issue of high resource utilization rose. Additionally, only this design has the opportunity of module-level pipelining. These issues will be dealt with in the following implementations.

6.3 Dataflow implementation – Task-level pipelining

In this set of tests, we implement the `DATAFLOW` version of the modules that we described in Subsection 5.2.3. For the *Initial* and *Final 2* modules, we use a `memcopy` function that simply reads the data from the input AXI stream of the module and stores them into an array, which is then read by the first layer of the module. Vivado HLS failed to produce a synthesized design for the *Final 2* module; its process halted for multiple days before we canceled it. We subsequently use a design with limited parallelism (a combination of `unroll` with 500x factor and `PIPELINE` on the initialization and convolutional loops and only `PIPELINE` on the ReLU loop of the convolutional function), that Vivado HLS could successfully synthesize.

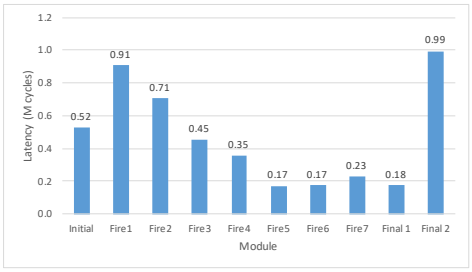
The results of this test set are shown in figure 6.3. Since the initiation interval of each module is different than its total latency, it is demonstrated separately in the speedup figures 6.3b and 6.2b: compared to the baseline implementation, the

speedup reaches a factor of as high as **559x** for latency and **714x** for initiation interval. Compared to *parallel pipeline*, the *dataflow* implementation is in average **1.56x** faster. However, for *Fire 2* and `Fire 4`, the *NewFirePool modules*, this configuration is marginally slower than the previous one. This happens because the *dataflow* configuration executes the ReLU part of each convolutional function sequentially. Compared to the respective expand functions, the `squeeze` function of each module executes much faster if fully parallelized. Therefore, the *parallel pipeline* configuration achieves a slightly lower total latency than the initiation interval of the *dataflow* configuration. However, the difference is **less than 32 K clock cycles** in both cases. Additionally, the *dataflow* configuration is slower than the *parallel pipeline* for the *Final 2* module, which is expected, since it is only parallelized with a factor of 500. Moreover, the design of *Final 2* uses a total of **989** DSPs to implement the multiply-accumulate functionality, with a **39%** utilization of the total available. Regarding the issue we observed with the previous configuration, where Vivado HLS failed to achieve initiation interval of one clock cycle for particular convolutional functions, under this configuration only the last convolution had the same problem. The `expand3x3` functions of the *NewFire 5*, *NewFire 6*, *NewFire 7* and *Final 1* modules achieved an initiation interval of one clock cycle.

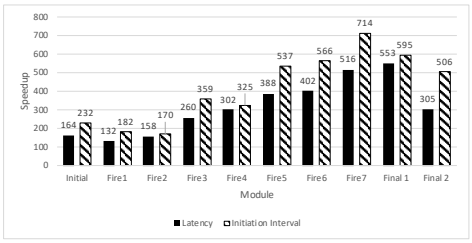
The BRAM utilization shown in figure 6.3d is consistently higher than with *parallel pipeline* in figure 6.2c. This happens because in the parallel implementation, the arrays for the module input/output are not accounted for; however the *dataflow* implementation fetches and data using AXI stream and stores them in local arrays. Additionally, the input data for the expand layers is duplicated, in order to satisfy the *one producer, one consumer* restriction of `DATAFLOW`. However, the *Final 2* module, that required 2004 BRAM modules under the previous configuration, now **only utilizes 1000**, since its parallelism factor (and thus the memory resource requirements) halved.

The main issue of the *dataflow* configuration regarding resource usage is the extremely high LUT utilization. *Fire 5* uses **97%** of the available LUTs and the following modules exceed the utilization cap by a large margin: *Final 1* requires more than **500%** of the LUTs available in the FPGA. The part that consistently required most of the total LUT utilization of each module is the *parameter initialization loops* we had implemented to set the weight and bias values. Vivado HLS most probably struggled to implement it as a part of the dataflow region, since it executes conditionally and accesses the weights and biases of each convolution. We moreover could not use the *Analysis Perspective* to evaluate it, as Vivado HLS would halt while loading it.

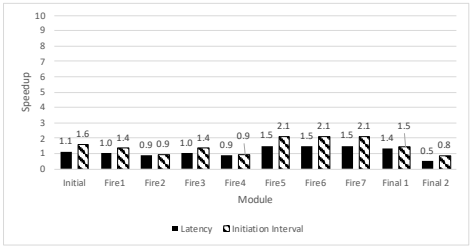
Although the implementation of task-level pipelining using the `DATAFLOW` directive improves the initiation interval (and thus the *throughput*) of the SqueezeNet modules, it comes at the cost of increased resource utilization. In the following section we will introduce an optimization to reduce the resource usage of the *dataflow* implementation, without any penalty in latency results.



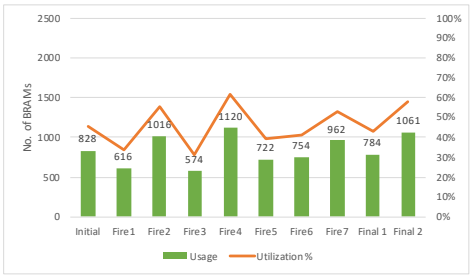
(a) Latency results



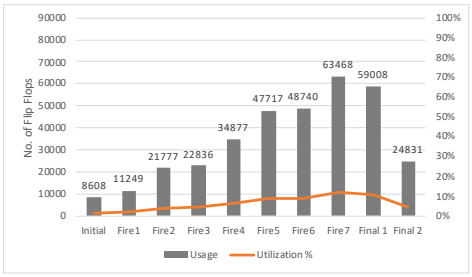
(b) Speedup compared to baseline



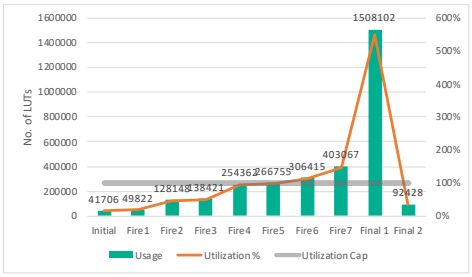
(c) Speedup compared to *parallel pipeline*



(d) BRAM usage and utilization



(e) Flip Flop usage and utilization



(f) LUT usage and utilization

Figure 6.3: Latency and resource utilization results for the *dataflow* implementation of the modules of SqueezeNet

6.4 Reducing resource utilization with dataflow

The *dataflow* implementation allows for both expand functions of each *New Fire* module to execute in parallel. However, the `expand1x1` function requires **9x** less operations than `expand3x3`, since it uses 1x1xn weights, compared to 3x3xn weights for `expand3x3`. It therefore completes execution much faster than its larger counterpart, which however does not benefit the module latency. In this set of experiments we will limit the parallelism of `expand1x1` in order to reach the approximate latency of `expand3x3` and will document the results.

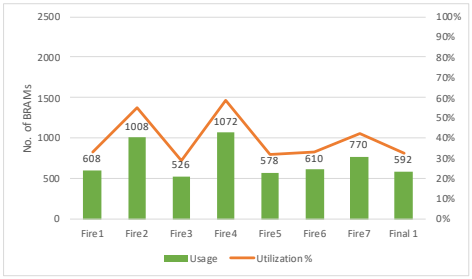
Figure 6.4 shows the utilization for each resource type and a comparison of the resource usage in relation to the previous *dataflow* configuration. We have not included the *Initial* and *Final 2* modules, since they do not contain expand layers ant are thus not optimized. Figure 6.4b shows that all modules have a drop in BRAM utilization, that accounts for both the arrays used locally for the convolution results and the weights arrays. Additionally, the Flip Flop usage is halved on average for all modules; however the Flip Flop utilization was already low, averaging around 7%. The most important result of this optimization is the significant drop in LUT utilization, shown in figure 6.4e: with the previous configuration, the *NewFire 6 NewFire 7* and *Final 1* modules required more than 100% of the available LUTs; this optimization allows the former two to be implemented in FPGAs with LUT utilization of **62%** and **91%**, respectively. Figure 6.4f furthermore shows that the LUT usage has dropped by **40%** on average.

This optimization successfully minimized the resource usage for each module. In the following implementation, we will further optimize the design by removing the initialization routines.

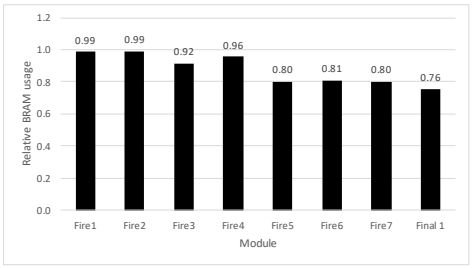
6.5 Removing the initialization loops

We introduced the weight and bias initialization loops in Subsection 4.1.2, as a method to overcome the issue Vivado HLS had with converting array initialization values into fixed-point. Although this mechanism allows for the design to modify the convolutional parameters at runtime, the results at Section 6.3 show that it causes issues in combination with the `DATAFLOW` directive. In this part we will synthesize the *optimized dataflow* configuration of the previous section without the parameter initialization code, with the weight and bias arrays declared as top-module parameters and will estimate their total resource usage based on the resources used by one instance of weight/bias module, multiplied by the number of instances each module implements.

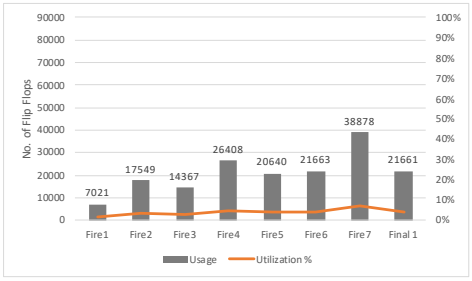
From previous experiments, each weight module consumes one BRAM on the FPGA, therefore an array partitioned with a factor of m requires m BRAM modules. Conversely, since the bias arrays are fully partitioned, they are allocated to Flip Flops and LUTs. In particular, Vivado HLS reports that each bias array element requires 16 Flip Flops and 1 LUT, thus a bias array of size n requires $16 * n$



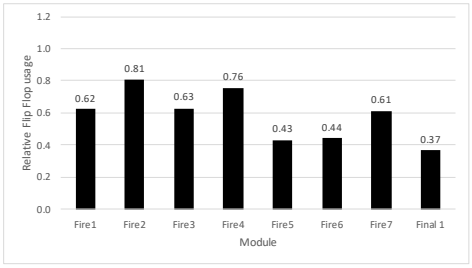
(a) BRAM usage and utilization



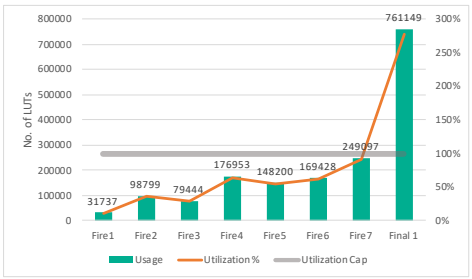
(b) Relative BRAM usage



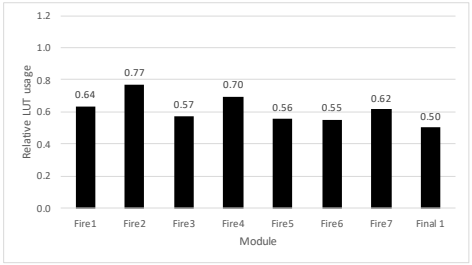
(c) Flip Flop usage and utilization



(d) Relative Flip Flop usage



(e) LUT usage and utilization



(f) Relative LUT usage

Figure 6.4: Resource utilization of the *optimized dataflow* configuration and comparison to the original *dataflow*

Module	Unroll factor			BRAMs	Flip Flops	LUTs
	conv1	conv2	conv3			
Initial	64	16	–	80	1280	80
Fire1	8	64	16	88	1408	88
Fire2	8	64	32	104	1664	104
Fire3	16	128	32	176	2816	176
Fire4	16	128	48	192	3072	192
Fire5	24	192	48	264	4224	264
Fire6	24	192	64	280	4480	280
Fire7	32	256	64	352	5632	352
Final1	32	256	–	288	4608	288
Final2	500	–	–	500	8000	500

Table 6.1: Unroll factors of the convolutional functions and resource usage for the weights/bias parameters of each SqueezeNet module

Flip Flops and n LUTs when fully partitioned. Using this information, we can estimate the parameter size and resource usage of each SqueezeNet module. Table 6.1 shows the unroll factor of each convolutional function of every SqueezeNet module (either `expand1x1`, `expand3x3`, `squeeze` or standard convoulutions) from the previous test set, together with the total BRAM, Flip Flop and LUT resources they require for the parameters with the particular unroll factor.

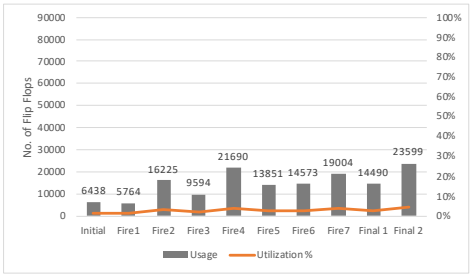
In figure 6.5, we demonstrate the resource of this *dataflow without initialization* configuration. The latency and initiation interval results had minimal deviations from the previous configuration and the BRAM utilization results were equal and therefore not shown in the figure. We observe a drop in both Flip Flop and LUT utilization across all modules compared to the *optimized dataflow* configuration. This is expected, since the Vivado HLS reports for the previous tests showed that a large percentage of LUTs was used to accommodate the parameter initialization process into the dataflow. Furthermore, this configuration resulted in modules that can be implemented into FPGAs, without exceeding the available resources.

6.6 SqueezeNet Assignment in FPGAs

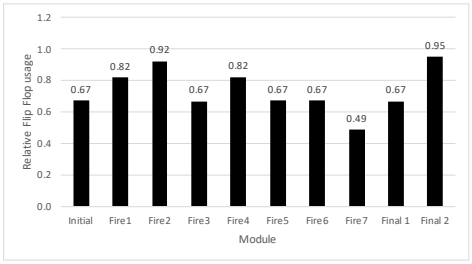
When partitioning SqueezeNet in FPGAs, we tried *to minimize the FPGA count*, 1) without *overstepping the FPGA utilization cap*, and 2) while using the *best-performing per-module optimizations* derived in previous sections. Thus, we tried to no in terms of speed..

Table 6.2 shows the total resource usage of every optimized SqueezeNet module, together with its output size in kbits. By using these metrics, we partition the network into **five FPGAs, with two modules per FPGA**. Essentially, our split tries to assign as many (contiguous) modules as possible per FPGA.

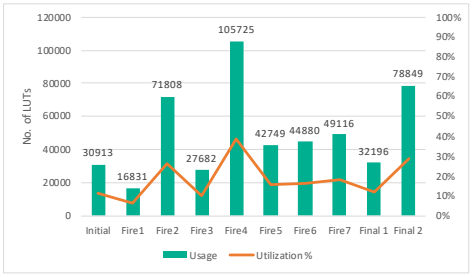
In table 6.3, we see the per layer latency of each module of SqueezeNet. For



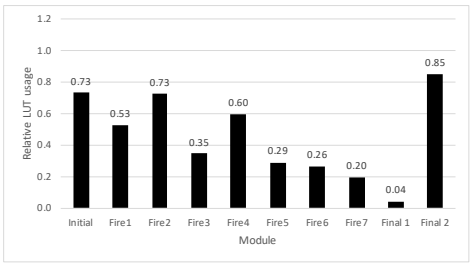
(a) Flip Flop usage and utilization



(b) Relative Flip Flop usage



(c) LUT usage and utilization



(d) Relative LUT usage

Figure 6.5: Resource utilization of the *dataflow without initialization* configuration and comparison to the *optimized dataflow*

Module	**Output size (Kbits)**	**BRAM**	**DSP**	**FF**	**LUT**
Initial	392	828	5	6438	30913
Fire1	392	608	0	5764	16831
Fire2	196	1008	0	16225	71808
Fire3	196	526	0	9594	27682
Fire4	73.5	1072	0	21690	105725
Fire5	73.5	578	1	13851	42749
Fire6	98	610	1	14573	44880
Fire7	98	770	0	19004	49116
Final1	392	592	0	14490	32916
Final2	7.8	1061	989	23599	78849

Table 6.2: Output size and resource usage of each SqueezeNet module

simplification's sake, we have not included the `concatenate` and `duplicator` functions. The *SqueezeNet total* latency at the bottom of the table is NOT the sum of latencies of the network layers because the total latency of each module is less than the latency sum of its comprising layers (due to `DATAFLOW` optimizing the communication between adjacent layers); effectively, the total network latency is lower than the latency sum of its layers. Using bold, table 6.3 depicts the latency of the *slowest layer per module*, which dictates the *initiation interval (II) of the module*. The table also depicts the operations each layer requires, in millions. In order to calculate these operations, we used the following formulas:

- For convolutional layers, we considered each multiply-accumulate (MAC) operation as two separate operations. We also included the comparisons of the ReLU each convolutional layer incorporates: $2*H_{out}*W_{out}*D_{out}*H_{filt}*W_{filt}*D_{in}+H_{out}*W_{out}*D_{out}$. H, W, D stand for *height*, *width* and *depth*, respectively.
- For max pooling layers: $H_{out}*W_{out}*D_{out}*H_{window}*Wwindow$.
- For global average pooling: $H_{in}*W_{in}*D_{in}+Dout$.
- For softmax: $2*D_{out}$

6.6.1 Squeezenet in five FPGAs: overall results

Figure 6.6 shows the final partitioning scheme of SqueezeNet in five FPGAs. The figure depicts the total resource utilization per FPGA and the inter-FPGA data transfer sizes and transfer times under a *10 Gbps* link with propagation latency of *500 ns*. The **total latency of a classification using SqueezeNet is L=23.8ms**, under zero load. In this metric, we don't include any latency for reading the input image or returning the classification results, as they might vary between implementations.

Our implementation achieves an initiation interval, II=655K clock cycles (i.e., the latency of the `expand3x3` functions in *NewFire 1* and *NewFire 2* modules). Running at 200MHz, **our implementation of SqueezeNet completes one new classification per 3.3ms, achieving a throughput T=303 image classifications per second (CPS)**.

Our optimized design **offers a total speedup factor of 340x compared to the baseline configuration**, coming from hls4ml. As the SqueezeNet network has 778M operations (see table 6.3), our inference engines offers a capacity of **238 GOPS**. In contrast, a single-core, *general-purpose CPU* with a clock frequency of 2 GHz and one instruction per clock cycle would achieve **2.57 CPS**, almost **119x slower** than our approach.

Regarding utilization, we observe a maximum of **90%** for BRAMs, **54%** for LUTs and **7%** for Flip Flops. Only the last FPGA utilized DSPs, reaching **39%** utilization. On average, there is a utilization of **84%** for BRAMs, **36%** for LUTs and **7%** for Flip Flops, across all five FPGAs.

Module	Layer	Latency (cc)	# Mops
Initial	conv	**370311**	44.1
	maxpooling	15683	1.8
	squeeze	254026	6.5
Fire1	expand1x1	627210	6.6
	expand3x3	**655434**	58
	squeeze	454731	12.9
Fire2	expand1x1	627210	6.6
	expand3x3	**655434**	58
	maxpooling	3922	0.9
	squeeze	126234	3.2
Fire3	expand1x1	307338	6.5
	expand3x3	**326938**	57.9
	squeeze	226587	12.9
Fire4	expand1x1	307338	6.5
	expand3x3	**326938**	57.9
	maxpooling	982	0.45
	squeeze	59790	4.8
Fire5	expand1x1	114484	3.7
	expand3x3	**122510**	32.6
	squeeze	84879	7.2
Fire6	expand1x1	114484	3.7
	expand3x3	**122510**	32.6
	squeeze	88015	9.7
Fire7	expand1x1	152106	6.5
	expand3x3	**163278**	57.9
	squeeze	113103	12.86
Final1	expand1x1	152106	6.5
	expand3x3	**163278**	57.9
Final2	conv	**598033**	200.9
	globAvgPooling	197061	0.2
	softmax	6030	0.002
SqueezeNet total		4676487	778

Table 6.3: Per layer latency and millions of operations of each SqueezeNet module (in clock cycles). Numbers in bold show the layer with the highest latency per module.

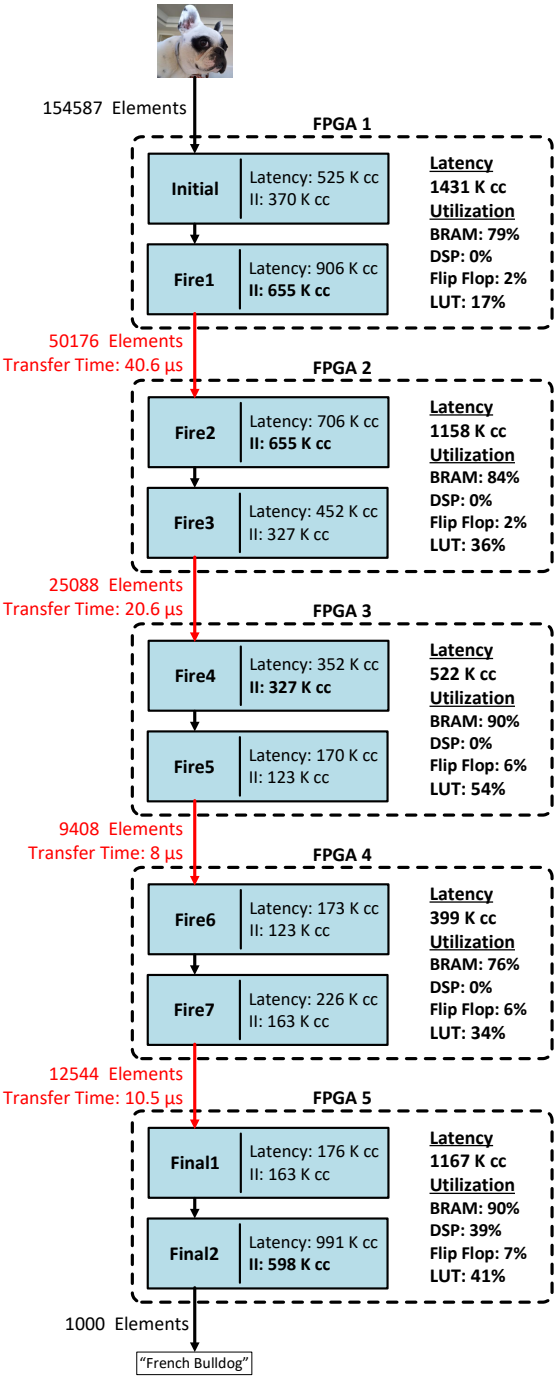


Figure 6.6: SqueezeNet modules assigned to FPGAs, with per-module latency and initiation interval (II) and per-FPGA latency and utilization. II values in bold indicate the modules with the highest initiation interval per FPGA (and thus the initiation interval of each FPGA).

6.6.2 Power consumption estimation

Since the available timeframe for this thesis did not allow us to download and evaluate our design on the ExaNeSt prototype, the power consumption metrics are estimations based on previous research with similar primitives. Dr. Angelos Ioannou at FORTH, measured in [31] a power consumption of **12 Watts per FPGA** on the ExaNeSt prototype for a multiply-accumulate accelerator with larger hardware utilization than our design. We therefore inherit this metric as a *pessimistic approximation* of the projected power consumption of our design per FPGA (i.e. actual power consumption could possibly be lower than this metric).

The estimated total power consumption of our design (using five FPGAs) is **60 Watts**. Given the computational capacity of 238 GOPS, the *Energy Efficiency* of our design is **4 GOPS/W**. Furthermore, our design requires **2.1x** less power than [22] (that reports 126 Watts power consumption) and competes with [24] (reporting 54.4 Watts) and [23] (reporting 43.2 Watts for a 6-FPGA solution).

The aforementioned power metrics do not integrate any projections of what the power requirements regarding the part that prepares and transfers the images to our FPGA pipeline might be. This is because the power consumption of this part can vary based on the way the part is implemented;

We should note that the aforementioned power and throughput metrics **do not** include the part that precedes the FPGA pipeline and is responsible for preparing and transferring the images/tasks to the inference engine. This is because the power consumption of this part depends on *how its mechanism is implemented*. Such implementation is not a part of our work; however it is imperative that it is explored in future work both for the *completeness of the design* and *to gather accurate power consumption metrics*.

We are nevertheless confident that the image preparation/transfer part will not negatively affect the throughput of our design. Considering a typical 10 Gbps throughput of a commodity interface e.g. Ethernet, each image classification task of $227 * 227 * 3 = 155$ KBytes will be transferred in around **125μs**, i.e. *one order of magnitude faster than the throughput of the inference engine.*

6.6.3 Comparison with CPU implementation

In order to compare our result against that of a typical CPU implementation, we use an Intel Core i5-3570 CPU (quad-core running at 3.6 GHz) with 12 GB RAM running Windows 10 environment. The SqueezeNet model is provided by [30] and we use Keras version 2.1.1 over Tensorflow version 1.4. The dataset for our test is Tiny ImageNet [32] that provides 10000 images for inference. The Python script we use to carry out the experiment is included in Appendix A.13. In summary, we load all 10000 images in RAM and use the `perf_counter()` function to capture the time before and after the `model.predict()` call that carries out the inference task. By capturing the runtime of the prediction function, we seek to *only measure the inference latency of the CPU implementation* and exclude the preceding steps

of loading the images into the RAM and preparing them for the inference task. As such, the throughput/power consumption metrics we present in the following paragraph are comparable with those of our FPGA inference engine.

We measured the total latency for the inference task of 10000 images to be **180 seconds**. This translates to **18 ms** average inference latency and a throughput of **55.6 CPS**. Additionally, this result indicates that the CPU reached a computational capacity of **43 GFLOPS**. The CPU utilized all four cores throughout the inference process. In lieu of an appropriate method to accurately estimate the cumulative power consumed by the CPU, RAM and the motherboard, we resorted to using an online power estimator[1], which indicated that the CPU configuration would consume **162 Watts**. This translates to *Energy Efficiency* of **0.27 GFLOPS/W**.

Given the aforementioned metrics, our implementation is faster than the CPU approach by a factor of **5.45**, since it reaches a throughput of **303 CPS**. Additionally, our design consumes **2.7 times less power** than the CPU. Therefore, considering the energy efficiency of the two designs (i.e .GOPS/W), our approach outperforms the CPU implementation by a factor of **14.8**.

[1] https://outervision.com/power-supply-calculator

Chapter 7

Conclusions – Future Work

In this thesis, we defined an architecture where a cluster of FPGAs work concurrently on user-streams of inference requests, obtaining an data-flow like, pipelined inference engine in which one can *scale-out the computation capacity* by using more hardware resources. We demonstrated this method by implementing SqueezeNet using a pipelined array of five (5) Zynq UltraScale+ FPGAs. Our implementation achieved a throughput of **303 classifications per second (CPS)**, offering processing capacity of **238 GOPS**, and an **end-to-end inference latency of 24ms**. In this course, we modified the convolutional function of hls4ml, and we implemented our own max pooling, global average pooling and softmax functions. We performed a multitude of experiments, evaluating different sets of directives provided by Vivado HLS and we adapted them on the ten (10) SqueezeNet modules. Furthermore, we defined our own SqueezeNet modules *in order to minimize inter-module transfer size.* Our design is pipelined on a per-layer basis and demonstrates an overlap between the layer computation and inter-layer communication and data transfers inside the same FPGA, achieved through the use of `DATAFLOW` directives. The final design resides solely on *the programmable part of the Zynq UltraScale+, with its parameters stored on on-chip BRAMs.*

We aimed at a fully automated workflow using keras/hls4ml and Vivado HLS. In this path, *the hls4ml proved to be suboptimal and a significant portion of this work was to optimize its output (HLS code) using special HLS directives.* Although HLS is a powerful tool, solving difficult scheduling problems while synthesizing C++ code into RTL, in this thesis we had to overcome many issues, as we used it to heavily optimize large designs. Also, the front-end of the tool was not always very helpful: when trying aggressive optimizations, after many hours or days of wait, it was not always clear whether the synthesis had failed, stalled or was still in progress. Also some of the HLS reports were not clear and indicative of the problems experienced.

This work can be continued in many different ways. Below we list the most prominent of them:

1. A immediate step is to integrate all the building blocks we developed in this

thesis on the actual ExaNeSt platform. In this direction, we can develop a simple module that *converts the AXI-stream transmissions to ExaNeSt traffic* (and vice-versa) so that they can be routed between FPGAs using available hardware.

2. From that point onward, we can optimize the interconnect system by introducing flow control and buffering to minimize latency under high workloads, and also allow data-flow like operation (computation overlap) across FPGAs, something that we so far showed inside FPGA boundaries.

3. The hls4ml definitions of the convolutional layers limited parallelism to only one dimension of the output of each layer. One can explore different definitions of the convolutional function to allow for higher degrees of parallelism and achieve even higher throughput.

4. A different course of action is to *automate the conversion of a Keras neural network definition into an optimized Vivado HLS project*, by applying the optimization methods we used in our work into the hls4ml project generator. We believe that splitting a CNN into a number of FPGAs can also be automated to a certain degree.

5. The derived design modules have varying initiation interval values. We can configure them to match the initiation interval of the slowest module by limiting their parallelism, thus also saving FPGA resources. This will produce a more compact design with the same throughput results as the original, at the cost of higher end-to-end latency. In this direction, one can also explore different methods to optimize the design, by partitioning slow layers/modules on multiple FPGAs that can be pipelined. A differrent solution is to duplicate the hardware blocks that have high latency but cannot be parallelized. With this method, we can avoid congestion in the network, by having multiple slow layer instances working on different data inputs.

6. We can further elaborate on the many synthesis results we had, examining how performance correlates with utilization metrics.

7. In this work, we utilized five (5) FPGAs to implement SqueezeNet while meeting the performance targets set in Section 1.4. Larger networks may require more FPGAs to meet a similar target, thus for instance forcing to use as few layers per FPGA as possible. On the other hand, in certain configurations we may not need that many FPGAs, e.g. because of relaxed performance constraints. Limiting the per-layer parallelism (and accommodating more layers per FPGA) is a flexible knob that a user or automated tool can use in order to economize on FPGAs.

8. Last but not least, we can apply our methodology to other popular and possibly larger network architectures, e.g. GoogLeNet.

Bibliography

[1] Vivado Design Suite User Guide: High-Level Synthesis (UG902). Technical report, 2019.

[2] Yann LeCun, Bernhard E. Boser, John S. Denker, Donnie Henderson, R. E. Howard, Wayne E. Hubbard, and Lawrence D. Jackel. Handwritten Digit Recognition with a Back-Propagation Network. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 396–404. Morgan-Kaufmann, 1990.

[3] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[5] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A Domain-specific Architecture for Deep Neural Networks. *Commun. ACM*, 61(9):50–59, August 2018.

[6] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, and D. Burger. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38(2):8–20, March 2018.

[7] Xin Feng, Youni Jiang, Xuejiao Yang, Ming Du, and Xin Li. Computer vision algorithms and hardware implementations: A survey. *Integration*, 69:309–320, November 2019.

[8] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[9] Babak Falsafi, Bill Dally, Desh Singh, Derek Chiou, Joshua J. Yi, and Resit Sendag. FPGAs versus GPUs in Data centers. *IEEE Micro*, 37(1):60–72, January 2017. Conference Name: IEEE Micro.

[10] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. [DL] A Survey of FPGA-Based Neural Network Inference Accelerator. 9(4):26.

[11] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, and Francois Berry. Accelerating CNN inference on FPGAs: A Survey. *arXiv:1806.01683 [cs]*, May 2018. arXiv: 1806.01683.

[12] M. Katevenis, N. Chrysos, M. Marazakis, I. Mavroidis, F. Chaix, N. Kallimanis, J. Navaridas, J. Goodacre, P. Vicini, A. Biagioni, P. S. Paolucci, A. Lonardo, E. Pastorelli, F. Lo Cicero, R. Ammendola, P. Hopton, P. Coates, G. Taffoni, S. Cozzini, M. Kersten, Y. Zhang, J. Sahuquillo, S. Lechago, C. Pinto, B. Lietzow, D. Everett, and G. Perna. The ExaNeSt Project: Interconnects, Storage, and Packaging for Exascale Systems. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 60–67, August 2016.

[13] David Fitzpatrick. The functional organization of local circuits in visual cortex: insights from the study of tree shrew striate cortex. *Cerebral cortex*, 6(3):329–341, 1996. Publisher: Oxford University Press.

[14] Jeffrey Dean and Luiz Andre Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013. Publisher: ACM New York, NY, USA.

[15] Xilinx. *Zynq UltraScale+ MPSoC Technical Reference Manual UG1085*. 1.3 edition, 2016.

[16] Dimitris Giannopoulos, Nikos Chrysos, Evangelos Mageiropoulos, Giannis Vardas, Leandros Tzanakis, and Manolis Katevenis. Accurate Congestion Control for RDMA Transfers. In *2018 Twelfth IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, pages 1–8, October 2018. ISSN: 2474-3739.

[17] Philipp Gysel, Mohammad Motamedi, and Soheil Ghiasi. Hardware-oriented Approximation of Convolutional Neural Networks. *arXiv:1604.03168 [cs]*, October 2016. arXiv: 1604.03168.

[18] George Pitsis, Grigorios Tsagkatakis, Christos Kozanitis, Ioannis Kalomoiris, Aggelos Ioannou, Apostolos Dollas, Manolis Katevenis, and Panagiotis Tsakalides. *Efficient Convolutional Neural Network Weight Compression for Space Data Classification on Multi-fpga Platforms*. May 2019. Pages: 3921.

[19] David Gschwend. *ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network*. PhD thesis, ETH Zürich, Department of Information Technology and Electrical Engineering, August 2016.

[20] Kathirgamaraja Pradeep, Kamalakkannan Kamalavasan, and Natheesan Ratnasegar. EdgeNet: SqueezeNet like Convolution Neural Network on Embedded FPGA. December 2018.

[21] C. Huang, S. Ni, and G. Chen. A layer-based structured design of CNN on FPGA. In *2017 IEEE 12th International Conference on ASIC (ASICON)*, pages 1037–1040, October 2017.

[22] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-Efficient CNN Implementation on a Deeply Pipelined FPGA Cluster:. pages 326–331, August 2016.

[23] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing CNN Accelerator Efficiency Through Resource Partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 535–547, Toronto ON Canada, June 2017. ACM.

[24] Weiwen Jiang, Edwin H.-M. Sha, Xinyi Zhang, Lei Yang, Qingfeng Zhuge, Yiyu Shi, and Jingtong Hu. Achieving Super-Linear Speedup across Multi-FPGA for Real-Time DNN Inference. *CoRR*, abs/1907.08985, July 2019.

[25] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, and Zhenbin Wu. Fast inference of deep neural networks in FPGAs for particle physics. *Journal of Instrumentation*, 13(07):P07027–P07027, July 2018. arXiv: 1804.06913.

[26] EJ Kreinar. RFNoC Neural-Network Library using Vivado HLS. *Proceedings of the GNU Radio Conference*, page 17, 2017.

[27] Manolis Ploumidis, Nikolaos D Kallimanis, Marios Asiminakis, Nikos Chrysos, Pantelis Xirouchakis, Michalis Gianoudis, Leandros Tzanakis, Nikolaos Dimou, Antonis Psistakis, Panagiotis Peristerakis, and others. Software and Hardware co-design for low-power HPC platforms. In *International Conference on High Performance Computing*, pages 88–100. Springer, 2019.

[28] Fabien Chaix, Aggelos Ioannou, Nikolaos Kossifidis, Nikolaos Dimou, Giorgos Ieronymakis, Manolis Marazakis, Vassilis Papaefstathiou, Vassilis Flouris, Mihailis Ligerakis, Georgios Ailamakis, and others. Implementation and impact of an ultra-compact multi-FPGA board for large system prototyping. In *2019 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, pages 34–41. IEEE, 2019.

[29] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. *arXiv:1602.07360 [cs]*, February 2016. arXiv: 1602.07360.

[30] GitHub - rcmalli/keras-squeezenet: SqueezeNet implementation with Keras Framework.

[31] Aggelos Ioannou. UniLogic (Unified Logic): A scalable architecture for increased programmability in highly parallel reconfigurable systems. 2020. Publisher: Technical University of Crete.

[32] Tiny ImageNet Visual Recognition Challenge. Available at https://tiny-imagenet.herokuapp.com/.

Appendix A

Source code

This appendix includes source code referenced in the main part of this thesis.

Listing A.1: The header of `conv2d`

```
template<class data_T, class res_T, typename CONFIG_T>
void conv_2d(
    data_T data[in_height * in_width * n_chan],
    res_T res[out_height * out_width * n_filt],
    typename weight_t weights[filt_height * filt_width * n_chan *
        n_filt],
    typename bias_t biases[n_filt])
)
```

Listing A.2: The accumulation part of `conv2d`

```
AccumOutHeight: for(int oh = 0; oh < out_height; oh++) {
    AccumOutWidth: for(int ow = 0; ow < out_width; ow++) {
        AccumFilt: for(int ff = 0; ff < n_filt; ff++) {
            AccumChan: for(int cc = 0; cc < n_chan; cc++){
                AccumDotHeight: for(int fh = 0; fh < filt_height; fh
                    ++){
                    AccumDotWidth: for(int fw = 0; fw < filt_width;
                        fw++){
                        int index_mult = oh*out_width*n_filt*n_chan*
                            filt_height*filt_width
                                    + ow*n_filt*n_chan*filt_height
                                        *filt_width
                                    + ff*n_chan*filt_height*
                                        filt_width
                                    + cc*filt_height*filt_width
                                    + fh*filt_width
                                    + fw;
```

```
13                      int index_acc = oh*out_width*n_filt
14                                    + ow*n_filt
15                                    + ff;
16
17                      acc[index_acc] += mult[index_mult];
18
19                  }//end dot product filter width loop
20              }//end dot product filter height loop
21          }//end n channel loop
22      }//end n filter loop
23  }//end output width loop
24 }//end output height loop
```

Listing A.3: acc array initialization

```
1 for(int oh = 0; oh < out_height; oh++) {
2     for(int ow = 0; ow < out_width; ow++) {
3         for(int ff = 0; ff < n_filt; ff++) {
4             acc[oh*out_width*n_filt + ow*n_filt + ff]=biases[ff];
5         }
6     }
7 }
```

Listing A.4: The multiplication part of conv2d

```
1  ConvOutHeight: for(int oh = 0; oh < out_height; oh++) {
2      ConvOutWidth: for(int ow = 0; ow < out_width; ow++) {
3          ConvFilt: for(int ff = 0; ff < n_filt; ff++){
4              ConvChan: for(int cc = 0; cc < n_chan; cc++){
5                  ConvFiltHeight: for(int fh = 0; fh < filt_height; fh
                       ++){
6                      ConvFiltWidth: for(int fw = 0; fw < filt_width;
                           fw++){
7                          int index_mult = oh*out_width*n_filt*n_chan*
                               filt_height*filt_width
8                                         + ow*n_filt*n_chan*filt_height
                                              *filt_width
9                                         + ff*n_chan*filt_height*
                                              filt_width
10                                        + cc*filt_height*filt_width
11                                        + fh*filt_width
12                                        + fw;
13
14                         int index_weight = fh*filt_width*n_chan*n_filt
15                                          + fw*n_chan*n_filt
```

```
16                                      + cc*n_filt
17                                      + ff;
18                          if ((oh*stride_height+fh) < pad_top
19                          || (oh*stride_height+fh) >= (pad_top+in_height
                                )
20                          || (ow*stride_width+fw) < pad_left
21                          || (ow*stride_width+fw) >= (pad_left+in_width)
                                ) {
22                              mult[index_mult] = 0;
23                          } else {
24                              int index_data = (oh*stride_height+fh-
                                    pad_top)*in_width*n_chan
25                                          + (ow*stride_width+fw-
                                                pad_left)*n_chan
26                                          + cc;
27                              mult[index_mult] = data[index_data] *
                                    weights[index_weight];
28                          }
29                      }//end mult loop
30                  }//end channel loop
31              }//end filter width loop
32          }//end filter height loop
33      }//end output width loop
34  }//end output height loop
```

Listing A.5: The `compute_multiplier_limit` function

```
template<typename CONFIG_T>
    int compute_multiplier_limit_conv2d(
    typename weight_t weights[filt_height * filt_width * n_chan *
       n_filt]
)
{
    int n_mult = 0;
    for(int oh = 0; oh < out_height; oh++) {
        for(int ow = 0; ow < out_width; ow++) {
            for(int ff = 0; ff < n_filt; ff++){
                for(int cc = 0; cc < n_chan; cc++){
                    for(int fh = 0; fh < filt_height; fh++){
                        for(int fw = 0; fw < filt_width; fw++){
                                int index_weight = fh*filt_width*n_chan
                                   *n_filt
                                                + fw*n_chan*n_filt
                                                + cc*n_filt
                                                + ff;
                                if ((oh*stride_height+fh) < pad_top
                                || (oh*stride_height+fh) >= (pad_top+
                                   in_height)
                                || (ow*stride_width+fw) < pad_left
                                || (ow*stride_width+fw) >= (pad_left+
                                   in_width)) {
                                    continue;
                                } else {
                                    if (weights[index_weight] > 1e-20
                                        || weights[index_weight] < -1e
                                        -20) {
                                        n_mult++;
                                    }
                                }
                        }//end mult loop
                    }//end channel loop
                }//end filter width loop
            }//end filter height loop
        }//end output width loop
    }//end output height loop
    return ceil( float(n_mult) / float(reuse_factor) );
}
```

Listing A.6: conv2d with mult and acc merged

```
1  template<class data_T, class res_T, typename CONFIG_T>
2  void conv_2d(
3      data_T data[in_height*in_width*n_chan],
4      res_T acc[out_height*out_width*n_filt],
5      typename weight_t weights[filt_height * filt_width * n_chan *
           n_filt],
6      typename bias_t biases[n_filt])
7  {
8      typename accum_t mult;
9
10     InitHeight: for(int oh = 0; oh < out_height; oh++) {
11         InitWidth: for(int ow = 0; ow < out_width; ow++) {
12             InitFilt: for(int ff = 0; ff < n_filt; ff++) {
13                 acc[oh*out_width*n_filt + ow*n_filt + ff]=biases[ff];
14             }
15         }
16     }
17
18     ConvOutHeight: for(int oh = 0; oh < out_height; oh++) {
19         ConvOutWidth: for(int ow = 0; ow < out_width; ow++) {
20             ConvFilt: for(int ff = 0; ff < n_filt; ff++){
21                 ConvChan: for(int cc = 0; cc < n_chan; cc++){
22                     ConvFiltHeight: for(int fh = 0; fh < filt_height;
                           fh++){
23                         ConvFiltWidth: for(int fw = 0; fw < filt_width
                               ; fw++){
24
25                             int index_weight = fh*filt_width*n_chan*
                                   n_filt
26                                                 + fw*n_chan*n_filt
27                                                 + cc*n_filt
28                                                 + ff;
29
30                             if ((oh*stride_height+fh) < pad_top
31                                 || (oh*stride_height+fh) >= (pad_top+
                                       in_height)
32                                 || (ow*stride_width+fw) < pad_left
33                                 || (ow*stride_width+fw) >= (pad_left+
                                       in_width)) {
34                                     mult = 0;
35                                 } else {
```

```
36                                  int index_data = (oh*stride_height+
                                        fh-pad_top)*in_width*n_chan
37                                                  + (ow*stride_width+
                                                        fw-pad_left)*
                                                        n_chan
38                                                  + cc;
39                                  mult = data[index_data] * weights[
                                        index_weight];
40                              }
41                          int index_acc = oh*out_width*n_filt + ow*
                                n_filt + ff;
42                          acc[index_acc] += mult;
43                      }//end mult loop
44                  }//end channel loop
45              }//end filter width loop
46          }//end filter height loop
47      }//end output width loop
48  }//end output height loop
49 }//end conv2d
```

Listing A.7: The maxpooling function

```
1
2  void maxpooling(ap_fixed<8,4> data[113][113][64],
3       ap_fixed<8,4> res[56][56][64]){
4
5   OutHeight: for(int oh = 0; oh < out_height; oh++){
6    OutWidth: for(int ow = 0; ow < out_width; ow++){
7     OutFilt: for(int ff = 0; ff < n_filt; ff++){
8      ap_fixed<8,4> elem = 0;
9      PoolHeight:for(int ph = 0; ph < 3; ph++){
10      PoolWidth:for(int pw = 0; pw < 3; pw++){
11       if((oh*2+ph) < 113 && (ow*2+pw<113)){
12        if(data[oh*2 + ph][ow*2 + pw][ff] > elem) elem = data[oh*2 +
              ph][ow*2 + pw][ff];
13       }
14      }
15     }
16     res[oh][ow][ff] = elem;
17    }
18   }
19  }
20 }
```

Listing A.8: The `globalAvgPooling` function

```
1
2  void globalAvgPooling(ap_fixed<8,4> data[14][14][1000],
3       ap_fixed<8,4> res[1000]){
4
5   OutFilt: for(int ff = 0; ff < 1000; ff++){
6    ap_fixed<16,12> temp;
7    InHeight: for(int ih = 0; ih < 14; ih++){
8     InWidth: for(int iw = 0; iw < 14; iw++){
9      if(ih == 0 && iw == 0) temp = data[ih][iw][ff];
10     else temp += data[ih][iw][ff];
11    }
12   }
13   res[ff] = temp/196;
14  }
15 }
```

Listing A.9: The `globalAvgPooling` function with array

```
1
2  void globalAvgPooling(ap_fixed<8,4> data[14][14][1000],
3              ap_fixed<8,4> res[1000]){
4
5   ap_fixed<16,12> temp[1000];
6
7   InHeight: for(int ih = 0; ih < 14; ih++){
8    InWidth: for(int iw = 0; iw < 14; iw++){
9     OutFilt: for(int ff = 0; ff < 1000; ff++){
10     if(ih == 0 && iw == 0) temp[ff] = data[ih][iw][ff];
11     else temp[ff] = data[ih][iw][ff] + temp[ff];
12    }
13   }
14  }
15
16  OutRes: for(int ff = 0; ff < 1000; ff++){
17   res[ff] = temp[ff] / 196;
18  }
19 }
```

Listing A.10: The `softmax` function

```
1
2  void softmax(ap_fixed<8,4> data[1000],
3     float res[1000]){
4
5   float exp_data[1000], sum_exp = 0;
6
7   InputExp: for(int ie = 0; ie < 1000; ie++){
8    exp_data[ie] = expf(data[ie].to_float());
9    sum_exp += exp_data[ie];
10  }
11
12  OutExp: for(int oe = 0; oe < 1000; oe++){
13   res[oe] = exp_data[oe] / sum_exp;
14  }
15 }
```

Listing A.11: The head of the `fire` function

```
1  template<class sq_data_T, class sq_res_T, typename sq_CONFIG_T,
2    class ex1_data_T, class ex1_res_T, typename ex1_CONFIG_T,
3    class ex3_data_T, class ex3_res_T, typename ex3_CONFIG_T,
4    typename concat_CONFIG_T>
5  void fire(ex1_data_T data[ex1_in_height][ex1_in_width][ex1_n_chan
       ],
6     typename sq_weight_t sq_weights[sq_filt_height][sq_filt_width][
          sq_n_chan][sq_n_filt],
7     typename sq_bias_t sq_biases[sq_n_filt],
8     typename ex1_weight_t ex1_weights[ex1_filt_height][
          ex1_filt_width][ex1_n_chan][ex1_n_filt],
9     typename ex1_bias_t ex1_biases[ex1_n_filt],
10    typename ex3_weight_t ex3_weights[ex3_filt_height][
          ex3_filt_width][ex3_n_chan][ex3_n_filt],
11    typename ex3_bias_t ex3_biases[ex3_n_filt],
12    sq_res_T res[sq_out_height][sq_out_width][sq_n_filt]){
13
14   ex1_res_T concat_1[concat_n_elem1_0][concat_n_elem1_1][
         concat_n_elem1_2];
15   ex3_res_T concat_2[concat_n_elem2_0][concat_n_elem2_1][
         concat_n_elem2_2];
16   ex1_res_T concat[concat_n_elem1_0][concat_n_elem1_1][
         concat_n_elem1_2 + concat_n_elem2_2];
17   expand1x1<ex1_data_T, ex1_res_T, ex1_CONFIG_T>(data, concat_1,
         ex1_weights, ex1_biases);
18   expand3x3<ex3_data_T, ex3_res_T, ex3_CONFIG_T>(data, concat_2,
         ex3_weights, ex3_biases);
19   concatenate3d<ex1_data_T, ex3_data_T, ex1_res_T, concat_CONFIG_T>(
         concat_1, concat_2, concat);
20   squeeze1x1<sq_data_T, sq_res_T, sq_CONFIG_T>(concat, res,
         sq_weights, sq_biases);
```

Listing A.12: The updated `fire` function

```
1  template<class sq_data_T, class sq_res_T, typename sq_CONFIG_T,
2      class ex1_data_T, class ex1_res_T, typename ex1_CONFIG_T,
3      class ex3_data_T, class ex3_res_T, typename ex3_CONFIG_T,
4      typename concat_CONFIG_T>
5   void fire(ex1_data_T data[ex1_in_height][ex1_in_width]
6       [ex1_n_chan],
7   typename sq_weight_t sq_weights[sq_filt_height]
8    [sq_filt_width][sq_n_chan][sq_n_filt],
9   typename sq_bias_t sq_biases[sq_n_filt],
10  typename ex1_weight_t ex1_weights[ex1_filt_height]
11  [ex1_filt_width][ex1_n_chan][ex1_n_filt],
12  typename ex1_bias_t ex1_biases[ex1_n_filt],
13  typename ex3_weight_t ex3_weights[ex3_filt_height],
14  [ex3_filt_width][ex3_n_chan][ex3_n_filt],
15  typename ex3_bias_t ex3_biases[ex3_n_filt],
16  sq_res_T res[sq_out_height][sq_out_width]
17  [sq_n_filt]){
18
19   ex1_data_T ex1_data[ex1_in_height]
20   [ex1_in_width][ex1_n_chan];
21   ex3_data_T ex3_data[ex1_in_height]
22   [ex1_in_width][ex1_n_chan];
23   ex1_res_T ex1_res[ex1_out_height]
24   [ex1_out_width][ex1_n_filt];
25   ex3_res_T ex3_res[ex3_out_height]
26   [ex3_out_width][ex3_n_filt];
27
28   duplicator<ex1_data_T, ex1_CONFIG_T>(data, ex1_data, ex3_data);
29   ex1_res_T concat[concat_n_elem1_0]
30   [concat_n_elem1_1][concat_n_elem1_2 + concat_n_elem2_2];
31   expand1x1<ex1_data_T, ex1_res_T, ex1_CONFIG_T>(ex1_data, ex1_res,
        ex1_weights, ex1_biases);
32   expand3x3<ex3_data_T, ex3_res_T, ex3_CONFIG_T>(ex3_data, ex3_res,
        ex3_weights, ex3_biases);
33   concat3d<ex1_data_T, ex1_CONFIG_T>(ex1_res, ex3_res, concat);
34   squeeze1x1<sq_data_T, sq_res_T, sq_CONFIG_T>(concat, res,
        sq_weights, sq_biases);
35  }
```

Listing A.13: Python script to determine CPU latency of SqueezeNet in Keras (average of 10000 image classifications)

```
1
2  import numpy as np
3  import os
4  import time
5  from keras_squeezenet import SqueezeNet
6  from keras.applications.imagenet_utils import preprocess_input,
       decode_predictions
7  from keras.preprocessing import image
8
9  model = SqueezeNet() # Using predefined SqueezeNet model
10
11 folder_path = '.' # Run on folder containing 10000 images
12
13 img_width, img_height = 227, 227
14
15 images = []
16 for img in os.listdir(folder_path):
17     img = os.path.join(folder_path, img)
18     img = image.load_img(img, target_size=(img_width, img_height))
19     img = image.img_to_array(img)
20     img = np.expand_dims(img, axis=0)
21     images.append(img)
22
23 images = np.vstack(images)
24
25 # Execute inference task, measuring execution time
26 start = time.perf_counter()
27 preds = model.predict(images)
28 end = time.perf_counter();
29
30 print('Image latency: ', (end-start)/10000)
```