Department of Physics and Astronomy

University of Heidelberg

Master thesis in Physics

submitted in April 2021 by

Hendrik Walter Heinz Borras

born in Itzehoe, Germany

Exploring Structured Sparsity within Data-Flow Architecture on Reconfigurable Hardware

This Master thesis has been carried out by Hendrik Walter Heinz Borras at the

Institute of Computer Engineering at Heidelberg University (ZITI)

under the supervision of

Prof. Dr.-Ing. Ulrich Brüning

and

Prof. Dr. Holger Fröning

Abstract

Exploring Structured Sparsity within Data-Flow Architecture on Reconfigurable Hardware

Deep Neural Networks (DNNs) are gaining crucial importance for modern computing and devices. Memory and compute requirements of state of the art DNNs are increasing steadily and the requirements of many DNNs impose serious challenges for running on so-called edge devices. These include devices such as sensors in the automotive industry or smartphones with slow internet connection. To improve the performance of DNNs on such low-power devices, model compression has been proposed and evaluated as a possible solution. For Field Programmable Gate Arrays (FPGAs), FINN is one of the most widely used frameworks for deploying compressed DNN models on low-power hardware. The compiler framework provides tools for building data-flow implementations of Convolutional Neural Networks (CNNs) with quantized weights and activations.

In the central part of this work two methods are proposed for column pruning in FINN, enabling further compression of CNN based models. The two methods vary in their pruning granularity. The coarse-grain method only prunes blocks of columns, while the fine-grained method is able to prune single columns. Both methods are evaluated with the VGG-like example network of FINN, that was trained on the CIFAR10 dataset. It is demonstrated that significant throughput improvements can be accomplished, while keeping the loss in accuracy acceptable. In order to run these experiments, an optimization procedure to automatically maximize model throughput on a given FPGA is developed and evaluated. Additionally, a special contribution to the FINN framework is developed for enabling parallel transformations, thus significantly speeding up the synthesis time for large CNNs.

All algorithms developed within this work are published as open source code on GitHub. Some of them are now integrated into the FINN framework by XILINX, enabling many users to profit from the performance improvements accomplished in this work.

Zusammenfassung

Untersuchung von strukturierter Dünnbesetzung in Datenflussarchitekturen auf rekonfigurierbarer Hardware

Tiefe Neuronale Netzwerke (DNNs, engl.: Deep Neural Network) gewinnen stetig an Bedeutung in Bereichen des maschinellen Lernens und der modernen Automatisierungstechnik. Gleichzeitig steigen die Speicher- und Rechenanforderungen aktueller DNNs. Dies stellt eine ernsthafte Herausforderung für den Betrieb dieser Netzwerke auf sogenannten „edge devices“ dar. Dazu gehören Geräte wie Sensoren im Automobilbereich oder Smartphones mit langsamer Internetverbindung. Um die Effizienz von DNNs auf solchen Geräten mit niedriger Leistung zu verbessern, wurde die Kompression von DNNs als mögliche Lösung vorgeschlagen und evaluiert. Für rekonfigurierbare Logikbausteine (FPGAs, engl: Field Programmable Gate Arrays) ist FINN eines der am weitesten verbreiteten Frameworks für den Einsatz von komprimierten DNN-Modellen auf stromsparender Hardware. Das Compiler-Framework bietet Werkzeuge zur Erstellung von Datenfluss-Implementierungen von faltendes neuronalen Netzwerken (CNNs, engl.: Convolutional Neural Networks) mit quantisierten Gewichten und Aktivierungen.
Im zentralen Teil dieser Arbeit werden zwei Methoden für Column Pruningïn FINN vorgeschlagen, welche eine weitere Kompression von CNN-basierten Modellen ermöglichen. Die beiden Methoden variieren in ihrer Granularität. Die grobkörnige Methode kann lediglich Blöcke an Spalten (Columns) prunen, wohingegen die feingranulare Methode einzelne Spalten prunen kann. Beide Methoden werden mit dem VGG-ähnlichen Beispielnetz von FINN evaluiert. Es wird gezeigt, dass signifikante Durchsatzverbesserungen erreicht werden können, während der Verlust an Genauigkeit akzeptabel bleibt. Um diese Experimente durchführen zu können, wird ein Optimierungsverfahren zur automatischen Maximierung des Modelldurchsatzes auf einem gegebenen FPGA entwickelt und evaluiert. Zusätzlich wird eine Erweiterung zum FINN-Framework entwickelt, um parallele Transformationen zu ermöglichen und damit die Synthesezeit für große CNNs deutlich zu beschleunigen.
Alle hier entwickelten Algorithmen sind als Open Source Code auf GitHub veröffentlicht und für jedermann zugänglich. Einige dieser Algorithmen sind schon von dem Unternehmen XILINX in FINN integriert. Dies ermöglicht es den vielen Nutzern bereits jetzt von den Steigerungen in der Leistungsfähigkeit zu profitieren.

Contents

1 Introduction

Over the last decade Deep Neural Networks (DNNs) have become the most accepted and utilized approach for many machine learning tasks. DNNs are capable of achieving state of the art performance in many applications, such as robotics [Len16], speech processing [Hin+12] and most prominently in the field of visual object recognition [KSH12]. In recent years, Zhang et al. [Zha+16] have shown that DNNs are capable of fitting almost any training data for a given task, making them a possible choice for many classification problems. This success and the high learning performance are attributed to the over-parametrization of models compared to the number data points fitted during training. Although unintuitive, together with Stochastic Gradient Descent (SGD) this leads very good results with low generalization error [LL18].

DNNs are being adapted to more and more complex tasks and the number of parameters present in the deployed models increases to fulfill the required over-parametrization. This is generally acceptable during training time, where massive parallelization on steadily improving Graphics Processing Units (GPUs) can be employed [Yin+18]. However, at the same time the requirements also increase for devices using these models for inference. In some cases, this can be solved by online inference in the cloud, thus not demanding any significant resources on the end device. But when DNNs are operated on so-called edge devices, such as smartphones and sensors in the automotive industry, the constraint of a generally low-power budget becomes problematic. In many cases additional demands, such as low latency and high throughput, must be met, imposing further constraints.

This issue is often combatted by employing lossy compression for the parameters of a given DNN, because in theory, a DNN contains unused or unimportant information due to the previously mentioned over-parametrization. One approach for this compression is to quantize the activations and weights within a DNN. One device class for such an application are Field Programmable Gate Arrays (FPGAs), since they are highly configurable to adapt to the changing demands of DNNs, while being available as low-power devices. For FPGAs one of the leading frameworks for deploying quantized DNNs and Convolutional Neural Networks (CNNs) is FINN ([Umu+17], [Blo+18]). FINN was first published in 2017 and is now an open-source project[1] with a quickly growing user and development community, while also receiving active development from the company XILINX[2].

Quantization approaches on Central Processing Units (CPUs) or GPUs are often limited to a minimum bit-width of eight bits. FINN however is able to utilize

[1] https://github.com/Xilinx/finn
[2] https://www.xilinx.com/

extremely low bit-widths, as low as one bit, while producing optimized hardware designs for any given bit-width, because the employed FPGAs are highly flexible and can be reconfigured to resemble almost arbitrary circuits. Through the use of extreme quantization FINN is able to directly map DNNs into hardware, creating on-chip data flow architectures. This is in strong contrast to the more generalizable loop-back architectures employed by many neural network accelerators, such as GPUs or Googles Tensor Processing Unit (TPU). Here activations are usually kept on-chip, while weights are loaded from external memory. FINN is able to deliver low latency and high throughput inference by allowing DNNs to stay completely on-chip. The framework is thus a good fit for applications on edge devices.

In this work different avenues are explored to improve the FINN framework. The following contributions are made:

- **Parallel synthesis**
  In order to speed up the synthesis time of networks deployed with FINN, a parallelization algorithm is developed. It separates long running synthesis tasks into different processes, enabling them to run concurrently. This allows the end-to-end build times of CNNs to improve by up to a factor of 2.5. The contributed implementation became part of the official FINN framework since release *0.3b*.

- **Automatic tuning of performance parameters**
  While conducting work on pruning, the requirement for maximizing throughput performance for a given network on a given FPGA arose. FINN provides multiple parameters for adjusting the performance per layer of a given CNN model, which results in a trade-off between resource usage and throughput. Two algorithms are developed which are capable of automatically adjusting the performance parameters of a given network to maximize throughput and resource utilization. Both algorithms are shown to properly adapt to different bit-widths for weights and activations. They are used in the ensuing work on pruning in FINN to evaluate the developed approaches for their maximum throughput performance on the Ultra96V2 FPGA.

- **Pruning in FINN**
  Two different methods for pruning within the data-flow architecture of FINN are developed. Both are based on column pruning (see definition Chapter 2.1 and visualized example in Figure 2.1a), which has been shown to achieve significant resource savings for DNNs, [Han+15] and [GYC16]. This approach to pruning has previously been studied within this group [Sch+20]. In an earlier study by XILINX so-called filter pruning without hardware modifications has been explored [Far+18]. In this work however the hardware is explicitly modified to improve performance. The presented methods are distinguished by the granularity with which they can prune a given weight tensor. The coarse-grain method prunes blocks of columns. The size of these blocks is defined by a performance parameter of FINN. In contrast, the fine-grained method is able to

prune single columns, but the selection of the sparsity settings is limited. Both methods are implemented as changes to the low-level code of FINN. Corresponding to each method, different training approaches are explored for the pruned networks. These are implemented to run with *Brevitas*, which is the training frontend used by FINN. An evaluation takes place afterwards for both methods in terms of achievable throughput and inference accuracy. As CNN a VGG-like network from the examples provided by FINN is used. The abbreviation VGG stems from the name of the group inventing this specific network layout. The network is trained on the CIFAR10 data set [KNH]. CIFAR10 is a data set commonly used for bench-marking machine learning algorithms for visual image recognition. These evaluation experiments are conducted for different sparsity settings, as well as for varying bit-widths for weights and activations.

The thesis is structured following the sequence of the developments produced during this work. In Chapter 2 necessary background information and defenitions are given. Then, the approach to the parallelized synthesis is explored in Chapter 3. Here, the implementation strategy is explained and performance improvements are evaluated.

In Chapter 4, the automatic tuning of performance parameters is discussed. Before implementing the optimization algorithm, different aspects of how the performance parameters impact the final circuit design are explored. This knowledge is employed for an informed decision process during the design of the optimization algorithm. To verify the capabilities of the optimization process, the resulting algorithm is tested on networks with different bit-width settings.

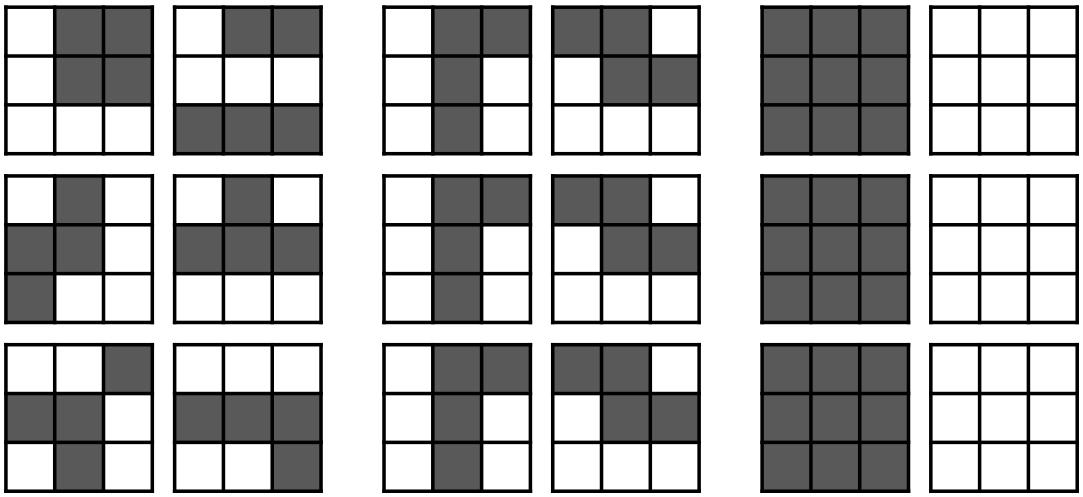
The main contribution of this work is described in Chapter 5. Here, two different approaches for pruning in FINN are explored. After implementation, they are assessed for their performance in terms of throughput and accuracy in different network bit-widths and sparsity settings.

Chapters 6 and 7 evaluate the results accomplished in this work, possible improvements are discussed and an overall conclusion is drawn.

2 Background

2.1 Sparsity and pruning

Sparsity describes how many elements of a tensor are zero and is generally measured as a percent value. For the compression of CNNs, sparsity is applied to the weight tensors of a given convolutional layer. Sparsity can be structured or unstructured. Unstructured sparsity means that values in a given tensor are randomly zero without any pattern, whereas structured sparsity exhibits some pattern. Examples of this are



(a) Weight pruning (b) Column pruning (c) Channel pruning

Figure 2.1: Illustration of unstructured (Figure 2.1a) and two structured forms of sparsity (Figure 2.1b and 2.1c) for a convolution tensor. The large squares represent the kernels, and the corresponding horizontal and vertical dimensions represent the number of input feature and output feature maps, respectively. The individual figure name describes the method by which the sparsity was introduced.

given in Figure 2.1. While both, unstructured and structured sparsity are possible, structured sparsity is often favored for CNN applications, because in general it maps better to parallel processors, such as GPUs. Furthermore, the improvements of selecting a fitting sparsity structure are two-fold: on one side, it becomes possible to reduce the amount of memory required by a weight tensor by not storing the zero values and using an indexing map to pass over large sections of zero values. On the other side, calculations for zero values can now be skipped, requiring less energy and time in the process.

The process by which weight tensors are made sparse is commonly called pruning. If the tensor contained no sparsity beforehand, then the percentage of values pruned corresponds to the sparsity percentage. The wording is adapted from the way in which decision trees in machine learning can be scaled down, similar to how trees in a garden can be trimmed with a pruning saw. In this work column pruning (Figure 2.1b) is employed to prune the convolution tensors in FINN.

2.2 FINN

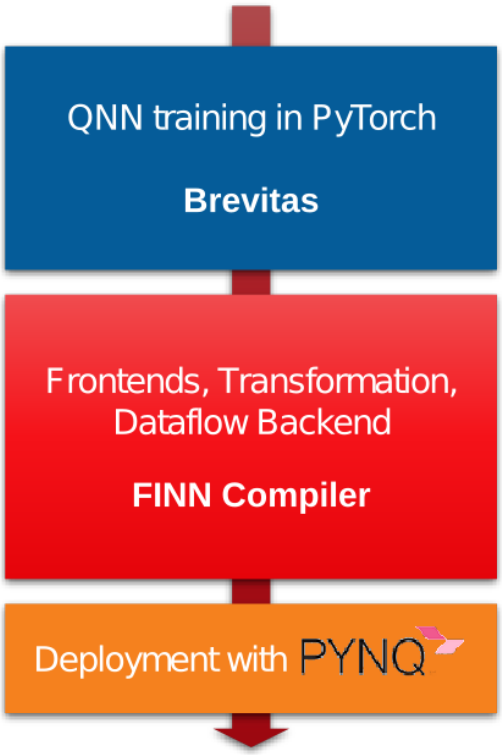


Figure 2.2: Schematic high-level description of the FINN tool-flow. Adapted from the FINN documentation[1].

The FINN framework was first published at the FPGA'17 conference [Umu+17] and an extension was later published in 2018 [Blo+18]. FINN aims to build an end-to-end deep-learning framework to train and build high throughput, ultra-low latency DNN compute architectures on FPGAs. The project has received major structural changes since the original publications. This was primarily triggered by the decision to move FINN to a fully open-source development model in 2019[2].

[1] https://github.com/Xilinx/finn/blob/af783db8dc2a1d2e95bd569d39464b935520b6d2/docs/img/finn-stack.png
[2] https://xilinx.github.io/finn//2019/10/02/rebuilding-finn-for-open-source.html

Nowadays FINN is compartmentalized into multiple projects and repositories, as visualized in Figure 2.2. These are as follows:

- **Brevitas**
  Extension library to PyTorch, which enables the training of quantized DNNs. A more detailed description will be given in Chapter 2.3. In the context of FINN *Brevitas* acts as a frontend to the FINN compiler and produces the network artifacts, which FINN uses to build and define DNN models.

- **FINN Compiler**
  This is the central part of FINN. The compiler combines multiple repositories and is used to facilitate the build process of a given DNN. Its functionality is explained in more detail in the following text.

- **Deployment with PYNQ**
  While PYNQ[3] is not directly part of the FINN effort, it facilitates additional infrastructure on the FPGAs supported by FINN. PYNQ provides *Python* libraries for interfacing the FPGA on XILINX ZYNQ and ALVEO devices. The open-source project is maintained by XILINX and provides pre-build SD card images for multiple ZYNQ boards as well as documentation for adapting the project to other boards. PYNQ enables the programmatic creation of drivers around the DNN accelerators, which FINN builds. These drivers can be built automatically by FINN and are used for throughput and verification tests on the target hardware.

As a key concept FINN aims to enable an end-to-end tool-flow. The idea is that users can define and train their custom DNN architecture in *Brevitas*, then use the FINN compiler to build a custom accelerator for a supported FPGA. Afterwards, FINN can then build a custom driver to run the accelerator remotely on the targeted FPGA. The end-to-end tool-flow of FINN currently supports multiple devices from the XILINX ZYNQ and ALVEO product family. A complete list for release version *0.5b* is given in Table 2.1. Alternatively, FINN can also build a single IP-block, which can be integrated into existing projects.

Product family	Devices				
Zynq	Ultra96	Pynq-Z1	Pynq-Z2	ZCU102	ZCU104
Alveo	U50	U200	U250	U280	

Table 2.1: XILINX FPGAs supported by FINN version *0.5b*.

The FINN compiler is built upon multiple key technologies. *Python* is used as the central programming language. The ONNX[4] format enables importing networks from *Brevitas* and is used as an intermediate network representation while building

[3] https://github.com/Xilinx/Pynq/
[4] https://onnx.ai/

an accelerator. For generating Register-Transfer Level (RTL) designs *Vivado* High Level Synthesis (HLS)[5] is used, which is included in the *Vivado Design Suite*[6]. With *Vivado* HLS it is possible to build RTL designs by describing them in *C++*. *Vivado* HLS additionally provides compiler *pragmas* to enable the user to issue RTL specific commands to the HLS compiler.

The FINN compiler is split into multiple repositories to enable easier and decoupled development. The central repository is the FINN repository itself[7], here most of the *Python* code for building an accelerator is located. This repository is also used when working with the FINN compiler in practice, as it can be used to launch a docker container which will setup a complete working environment that includes commits known to work well from the other repositories of the project. FINN uses the finn-base repository[8] for interacting with ONNX. This repository provides a wrapper for ONNX models along with basic transformations for interacting with the model. These functionalities can be used independent of the main FINN project to enable other acceleration frameworks. As mentioned before, FINN makes use of HLS to build the RTL design of a given accelerator. The finn-hlslib[9] is home to this HLS code. Here DNN specific operations are defined as templated *C++* functions. FINN uses these functions to create individual IP-blocks for each layer in a DNN.

In summary, FINN is a deep-learning and inference framework for quantized neural networks, defined in distinct projects. *Brevitas* is used as the training frontend. The FINN compiler represents the backend in which a trained model is built into an accelerator, that can be run on different FPGAs.

2.3 Brevitas

Brevitas[10] is designed as an extension library to the DNN training framework *PyTorch*[11]. It is developed as an open-source project and is maintained by XILINX [Pap21]. *Brevitas* is mainly designed as a tool for research with quantized DNNs. For this purpose, *Brevitas* provides quantized drop-in replacements for *PyTorch* functions. *Brevitas* supports mixed precisions for activations, weights and biases. The implementation of these quantizations is kept flexible to enable easier research, but comes with the trade-off in terms of increased training-time.

Because *Brevitas* is designed as a frontend for working with quantized networks, it does not provide any increased performance or memory savings during inference of quantized networks. Instead, increased performance during inference can be provided, by using one of the multiple backends to which *Brevitas* can export models. It should be noted that not all backends support all features of *Brevitas*, thus care

[5] https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html
[6] https://www.xilinx.com/products/design-tools/vivado.html
[7] https://github.com/Xilinx/finn
[8] https://github.com/Xilinx/finn-base
[9] https://github.com/Xilinx/finn-hlslib/
[10] https://github.com/Xilinx/brevitas
[11] https://pytorch.org/

should be taken when choosing which backend to use for a certain quantization approach. The available backends are:

- **FINN**, as mentioned in Chapter 2.2, FINN utilizes ONNX files exported by *Brevitas* to enable fast inference of quantized DNNs on FPGAs.
- **onnxruntime**[12], which enables fast inference of DNNs on processors and is build around ONNX.
- **PyTorch**[13], also has support for quantization since version *1.3*, however it is not as extensive as the one provided by *Brevitas*.
- **Apache TVM**[14] is a compiler framework for machine learning models and targets different CPUs, GPUs and accelerators.
- **XILINX Deep Learning Processor Unit**[15], which is similar to FINN and can map quantized DNNs onto FPGAs. However, in contrast to FINN it implements a loop-back instead of a data-flow architecture and is limited to an 8-bit fixed-point quantization.

In conclusion, with *Brevitas* it is possible to explore different quantization approaches, while using the underlying structure of *PyTorch*. Models trained with *Brevitas* can then be exported to different backends for fast inference.

[12] https://www.onnxruntime.ai/
[13] https://pytorch.org/blog/introduction-to-quantization-on-pytorch/
[14] https://tvm.apache.org/
[15] https://www.xilinx.com/products/intellectual-property/dpu.html

3 Parallel synthesis

Beginning of April 2020, when the work on this master thesis began, the rebuild of FINN had just started [Umu19]. The first release only provided very basic functionality for the framework, meaning that at this point in time the implementation was focused on completeness and correctness rather than performance. In this chapter a basic investigation into speeding up the framework is conducted. The results of this study are not only speeding up the experiments in this work, but are also contributed to the FINN framework, bringing their benefits to all users.

At its core FINN consists of a series of transformations applied to a neural network model. These transformations fulfill varying tasks, such as simplifying the network or replacing standard layers with their HLS counter parts. In general, these transformations divide into two categories:

1. Transformations, that affect the whole network graph.
2. Transformations, that only modify individual nodes in the network, without changing connections between nodes.

The expression "node" here explicitly references a node in the representation of a model, which FINN uses internally. To be precise, FINN uses the ONNX[1] format to keep an intermediate representation of the model. Nodes are often equivalent to layers in a neural network, but in some cases they can also fulfill other roles. As an example, a node can also represent a buffering First In - First Out (FIFO) queue on the FPGA, which has no equivalent in the original architecture of the neural network.

Most transformations within FINN fall into the first of these two categories. However, some of the most time intensive ones fall into the second class. These are in particular transformations which synthesize individual layers into IP-blocks or compile them for the *C++* simulator. The *C++* simulator is one of multiple ways to verify the correctness of networks generated with FINN. Most of the execution time in these transformations is generally spent on running external programs and as such not within the *Python* interpreter. For example, the transformation *HLSSynthIP* synthesizes each layer of a neural network into an IP-block. Here, the external command line tool *vivado_hls* is called for each node to generate an IP-block for the given layer. The utilized tool *vivado_hls* is part of the *Vivado Design Suite*[2]. While used by FINN, this tool is distinctly external to the project. The mentioned *HLSSynthIP* transformation then changes the nodes on an individual level, the layout of the network and the relation of the nodes to each other does not change.

[1] https://onnx.ai/
[2] https://www.xilinx.com/products/design-tools/vivado.html

As such, transformations of the second category are a prime example for a perfectly parallel problem. Each node can be viewed individually as its own independent process, which has no communication requirements during execution. Therefore, these processes can be executed in parallel and it should be possible to significantly speed-up a given transformation on a modern multi-core system. Additionally, the observed improvements should scale better with larger networks, because larger networks provide more layers to be synthesized in parallel, thus allowing for larger degrees of parallelization.

Although multiple transformations fit in the previous classification, the work focused on the *HLSSynthIP* transformation. This is the main task investigated for the work presented and the resulting implementation was subsequently included in the pull request to the FINN GitHub project[3].

3.1 Implementation

The implementation utilizes *Python's* module for working with parallel processes, aptly called *multiprocessing*[4]. Initially, the transformations were first adapted individually, but in the interest of code reuse and better readability, the parallel transformation itself is implemented as a class called *NodeLocalTransformation*. From this class further transformations can then be sub-classed. At its core, *NodeLocalTransformation* implements an *apply* member function which is called by a FINN model to apply a given transformation to itself. The class also forces sub-classing transformations to implement a member function called *applyNodeLocal*. Additionally, *applyNodeLocal* must accept a node from the model. The function can then perform a transformation on this single node. The *applyNodeLocal* implementation is then used by the *apply* function to realize the parallelization in practice. When a *NodeLocalTransformation* is applied to a model, it first spawns a pool of worker processes. The number of worker processes can be set either by passing the keyword argument *num_workers* or by setting the environment variable *NUM_DEFAULT_WORKERS*. This allows FINN to freely set the amount of parallelism for a given transformation call or for a project as a whole. The individual nodes of the model are then extracted and passed together with the *applyNodeLocal* function to the worker pool. The worker pool executes the node specific transformations in parallel and at its own ordering. After the node level transformations have completed, the model is reassembled with the modified nodes and passed back to the user.

While this implementation works surprisingly well, it exhibits some fundamental limitations. In particular, the synthesis times between different layers vary a lot. The synthesis time is primarily influenced by how many resources of the FPGA are used by a given layer. While the resource requirements vary between layer types, most resources are occupied by the Matrix-Vector-Threshold Units (MVTUs),

[3] https://github.com/Xilinx/finn/pull/78

[4] https://docs.python.org/3.6/library/multiprocessing.html

which perform most processing in a neural network. The resource demands of these layers depend primarily on how many processing elements are instantiated by FINN and how large these elements are. Additionally, the size of the weight matrix is a significant factor directly related to memory use.

Since these settings vary from one layer to another the synthesis time also varies significantly between layers. As a consequence of this variation, the scaling of the observed speed-up with the number of workers is not optimal. Figure 3.1 displays these time variations for a simple example. Here, a small fully-connected network from the FINN examples library is used. This network consists of three fully-connected layers, each containing 256 neurons, and an output layer with 10 neurons. Additionally, as a FINN requirement, the network also contains an output marker. This layer is used to connect to auxiliary hardware of the FPGA to transfer the resulting data. A total of five layers can be synthesized in parallel for this network. Figure

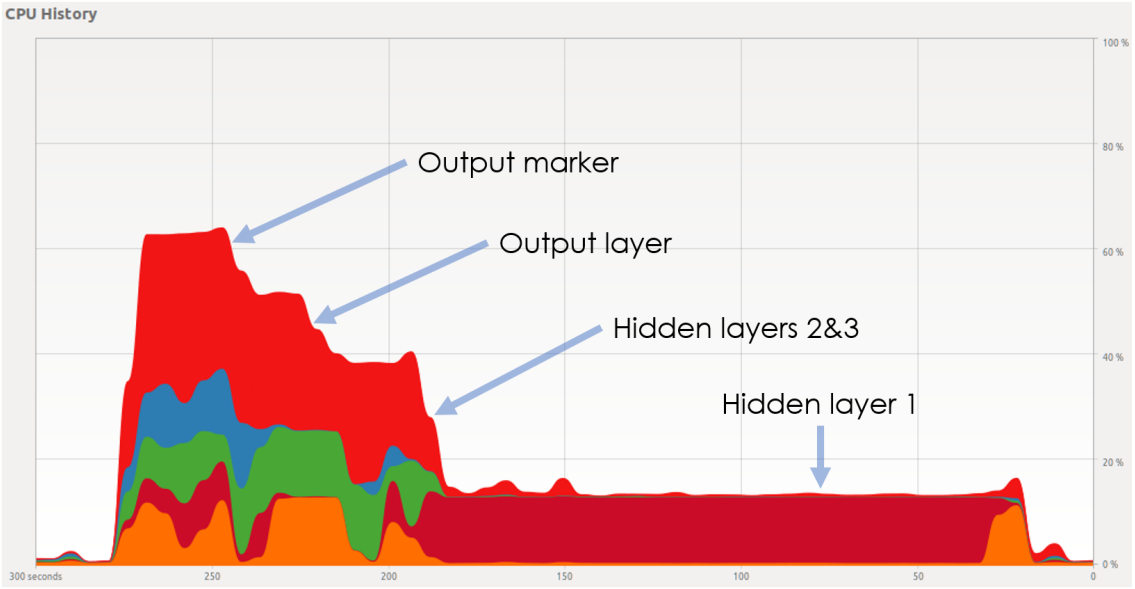


Figure 3.1: CPU utilization for the per-layer synthesis of a small network in FINN, visualized by the Ubuntu System Monitor, x-axis: Time in seconds, y-axis: CPU utilization of the test system in percent of total available CPU time.

3.1 shows the CPU utilization for this synthesis on an eight-core system. As expected, the transformation starts all five synthesis processes almost simultaneously, using up to 62% of the systems processing power. Whenever a layer completes its synthesis, the CPU utilization drops by approximately 12.5%, until only the final layer with the highest resource requirement is left running. This layer finally runs for approximately two thirds of the total synthesis time without any parallelization. This highlights how detrimental this imbalance of resource requirements can be for extrapolating potential time improvements. Nonetheless, the speed-up gained in this example is still about a factor of two and the results are encouraging to explore

this approach for parallelization further.

The final *Python* code of this implementation, as developed within this work, can be found in pull request 78 to the FINN repository on GitHub[5].

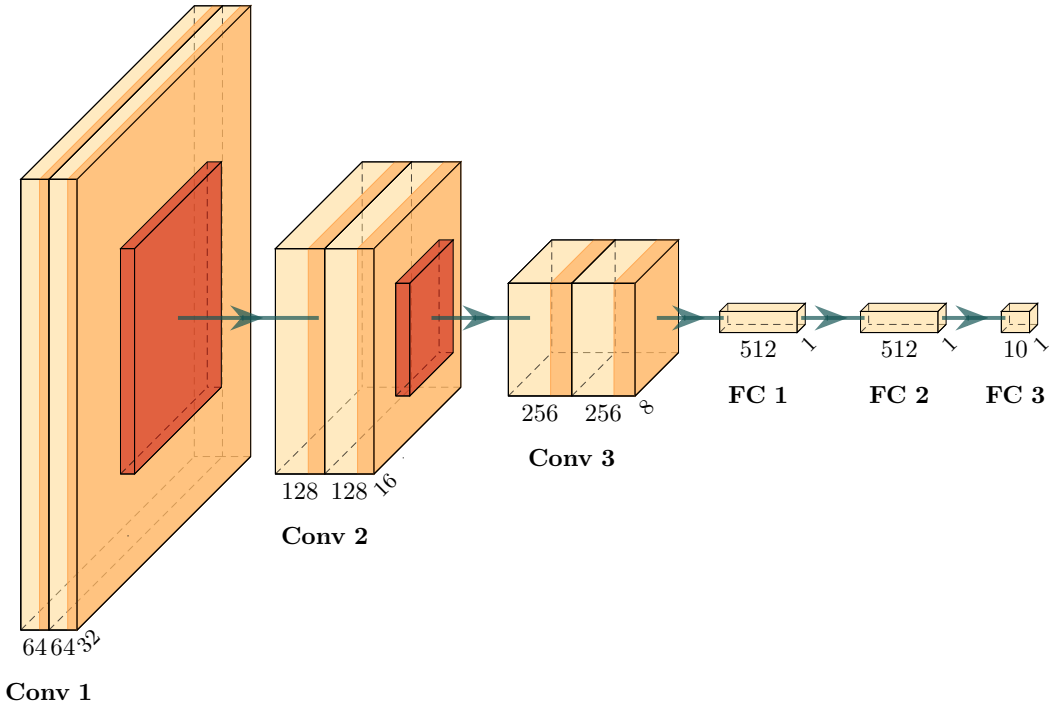


Figure 3.2: Architecture of the FINN CNN example, visualization created with PlotNeuralNet [Iqb18].

3.2 Performance improvements

In general, the performance improvements achieved by running transformations in parallel vary from one FINN release to another. Primarily because FINN is still in an early development state and there are plenty of feature changes or additions. The implementation for parallelization developed in this work and described above was integrated into FINN in release *0.3b*, published in May 2020. The fast evolution of FINN is mirrored in the fast increase of version numbers. In the interest of demonstrating the long-lasting impact of this contribution, all following benchmarks are conducted with the most recent release version of FINN. At the point of the final experiments of this work, this is *0.5b*, published in December 2020. The ensuing study is primarily working with the convolutional example network of FINN and, the speed-ups achieved with this network are highlighted in this thesis. In general,

[5] https://github.com/Xilinx/finn/pull/78

the convolutional network can be described as VGG-like, and is loosely based on the network architecture proposed by [SZ15]. The architecture is visualized in Figure 3.2 with each convolutional layer using 3x3 kernels, and each pooling layer being configured as a maximum pooling layer with 2x2 kernels. Images from the CIFAR10 dataset [KNH] are used as input. The size is given by 32 x 32 with each pixel containing three channels of 8-bit color information.

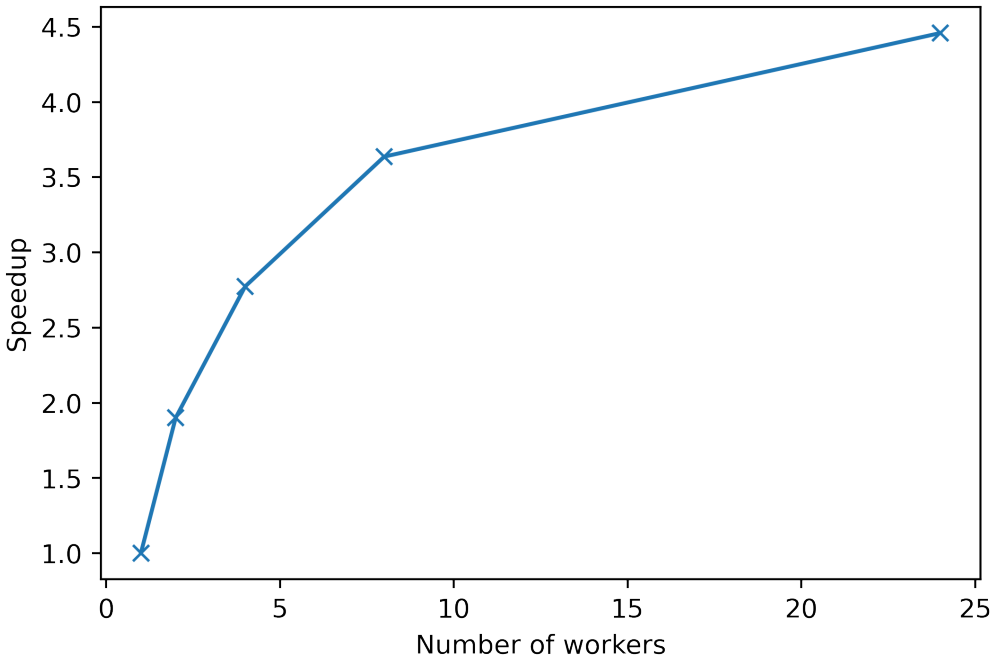


Figure 3.3: Speed-up for the HLS per-layer synthesis (*HLSSynthIP*) for varying numbers of workers, x-axis: Number of workers running in parallel, y-axis: speed-up as defined in equation 3.1.

As highlighted previously the primary interest of this contribution is the parallelization of the *HLSSynthIP* transformation. To quantify the performance gain, the speed-up compared to the serial implementation is measured, where the speed-up (S) is defined as:

$$S = \frac{T_s}{T_p}, \tag{3.1}$$

with T_s the wall time for the serial implementation and T_p the wall time for a given parallel implementation. The number of workers is then varied in multiples of two up to the maximum number of 24 cores on the test system. The results are shown in Figure 3.3. At first the speed-up is almost linear, but quickly begins to saturate. This behavior can be expected because of the previously described imbalance in synthesis time between different layers. The largest speed-up can be achieved when all cores of the system are in use with 24 workers and a value of $S_{max} = 4.46$ is accomplished. The initially almost linear increase in speed-up also means that even

small synthesis systems, such as local workstations can immensely profit from the speed-up through parallelization. This is very useful while developing for or with FINN. Consequently, also the rest of this work profited a lot from the speed-ups achieved here.

To determine the impact of parallelization in a broader context, measurements for the end-to-end performance of FINN are made. Of particular interest is again the convolutional network employed before. Figure 3.4 shows the overall wall time required, when moving from the initial *Brevitas* model to the finished bit-file, which can be run on a FPGA. This in particular highlights that the *HLSSynthIP* transfor-

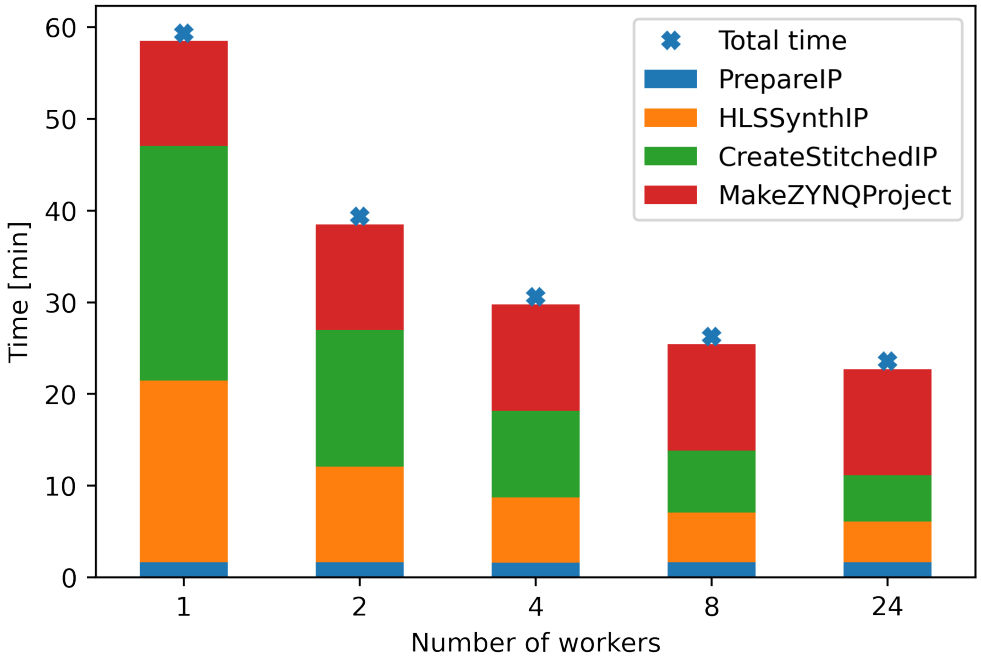


Figure 3.4: Build-time for the convolutional network. Long running transformations are given as stacked bars, x-axis: Number of workers running in parallel, y-axis: Total wall time for the end-to-end build.

mation is not the only long running transformation. Most notably *MakeZYNQProject* and *CreateStitchedIP* are also among the three most time intensive transformations. *MakeZYNQProject* creates the final *Vivado* project and is fully serial without many options for parallelization. *CreateStitchedIP* stitches the individual IP-blocks of different layers together. Since version 0.4b *CreateStitchedIP* also offers support for parallelization. In contrast to the implementation developed in this master thesis the parallelization is realized through *Vivados* own support for multi-core CPUs, because this transformation only involves one single call to *Vivado* and is executed on the whole model, not on a per-node basis. When employed together the parallelization of the *HLSSynthIP* and the *CreateStitchedIP* realize a maximum speed-up

of $S_{max} = 2.5$ when using all 24 cores of the test system, reducing the build-time from 59 minutes for the serial implementation down to 24 minutes. From private communication with Yaman Umuroğlu this speed-up is reported to be even larger for very large networks, such as ResNet-50 [He+15a]. Here the total build-time goes down from about a week to less than 24 hours, resulting in a speed-up of about seven times.

Taking it at face value: the early developments in this work laid the ground work for parallel transformations in FINN, which invoked major positive impact on the project and its community until this day.

4 Automatic tuning of performance parameters

After a close look at some of the internal functionalities of FINN, the further work is focused on trying to understand how FINN handles performance parameters. This chapter concentrates on how to influence and maximize throughput performance with FINN. Different internal parameters are explored and finally an optimization strategy is presented, developed and evaluated.

4.1 Motivation

In this work some of the potential limits of what FINN is able to deliver in terms of throughput on embedded devices is explored. Therefore, most of the experiments are run at the limits of what a given FPGA can deliver within its resource budget. This becomes particularly interesting when evaluating different approaches for sparsity for FINN in Chapter 5.

In order to achieve this goal of near maximum performance, an optimization algorithm is designed which can tune the available performance parameters within FINN to maximize the resource usage for a given FPGA. The algorithm is based on the estimate of resource utilization and latency for HLS layers in FINN. These estimates are a central part of [Blo+18] and [Umu+17], in which these estimates are used to meet a specific performance goal. However, in this work they are used to optimize towards a given resource goal. While these two goals are similar, the resulting algorithms differ significantly.

Board (Chip)	CLBs [K]	BRAM [Mb]	DSP slices	Max. clock [MHz]
PYNQ-Z1 (Z-7020)	85	4.9	220	200
Ultra96V2 (ZU3EG)	154	7.6	360	300

Table 4.1: Resource key figures for FPGA boards installed in the embedded lab of the Computing Systems Group at the University of Heidelberg. Further explanations are included in the text. Information taken from the Z-7020 [Xil19] and ZU3EG [Xil21b] product briefs.

In general, the computing resources on embedded FPGAs for edge applications are often limited, in comparison to their larger equivalents designed for data centers. Nonetheless, the overall amount still varies notably between different embedded devices. FINN currently supports five different devices from the XILINX Zynq

product line and another four from the Alveo product line[1]. For example, in the embedded lab of the Computing Systems Group of the University of Heidelberg, the PYNQ-Z1 and the Ultra96V2 evaluation boards are installed. Both of these boards are compatible with FINN. However, as shown in Table 4.1, these boards have largely varying amounts of resources available. Of major interest are often the number of processing units, divided into Digital Signal Processors (DSPs) and Configurable Logic Blocks (CLBs), along with the amount of Block Random-Access Memory (BRAM).

4.2 Pre-design experiments and insights

Different parameters and settings of FINN are explored before designing the optimization algorithm. This made it easier to understand how to adjust the performance parameters of different layers in FINN. Thus providing valuable input on how to design the final algorithm.

4.2.1 Available performance parameters

Multiple parameters are available within FINN to influence the performance of a synthesized neural network. In general, the performance of a given implementation is often characterized in terms of throughput, measured in images per second, and latency, measured in seconds or cycles when the clock frequency is fixed. In this work the throughput is used as the main performance indicator.

FINN currently provides two parameters for convolutional and fully-connected layers to influence performance. These are the Single Instruction Multiple Data (SIMD) and the Processing Element (PE) parameters and they are part of the HLS implementation of FINN. The functionality of these two parameters is shown schematically in Figure 4.1. The PE parameter defines how many computation elements are instantiated within a layer. These can work in parallel to each other and each of them produces one output element at a time. The SIMD parameter on the other hand determines how many input elements can be handled by one PE in parallel. Each of these parameters can individually linearly scale the amount of parallelism present in the MVTU. With the combination of both parameters quadratic scaling can be achieved.

However, when used for a specific network both parameters are subject to constraints limiting the range in which they can be set. These limits are given as follows for the SIMD parameter:

1. $\text{SIMD} \leq \text{CH}_{\text{IN}}$
2. $\text{CH}_{\text{IN}} \bmod \text{SIMD} = 0$

[1] https://github.com/Xilinx/finn-base/blob/0d362205b21e48f28e1bfabe6d10c4ecade6bef6/src/finn/util/basic.py

3. SIMD $> \mathrm{CH_{IN}}/1024$,

with SIMD denoting the current setting of the SIMD parameter as an integer and $\mathrm{CH_{IN}}$ the number of input channels. The first limitation is simply a resource related one: The MVTU cannot handle larger input widths than there are input elements to process. The second requirement is imposed by FINN itself, requiring that the SIMD parameter cleanly divides the number of input parameters. The last limitation is imposed by the *Vivado* HLS synthesis: for each MVTU an input buffer is constructed. The size of this buffer is defined as $\mathrm{CH_{IN}}/\mathrm{SIMD}$, however *Vivado* limits the maximum size of these buffers to 1024, thus resulting in a lower limit for the SIMD parameter.

The constraints for the PE parameter are in general similar, but exist in relation to the number of output channels ($\mathrm{CH_{OUT}}$):

1. $\mathrm{PE} \leq \mathrm{CH_{OUT}}$
2. $\mathrm{CH_{OUT}} \bmod \mathrm{PE} = 0$.

It is worth to note, that because no internal buffer exists for the output channels, there is no lower constraint imposed by *Vivado*.

While the SIMD and PE parameter are the primary settings to manipulate the performance and resource utilization of a layer, there are also other network settings

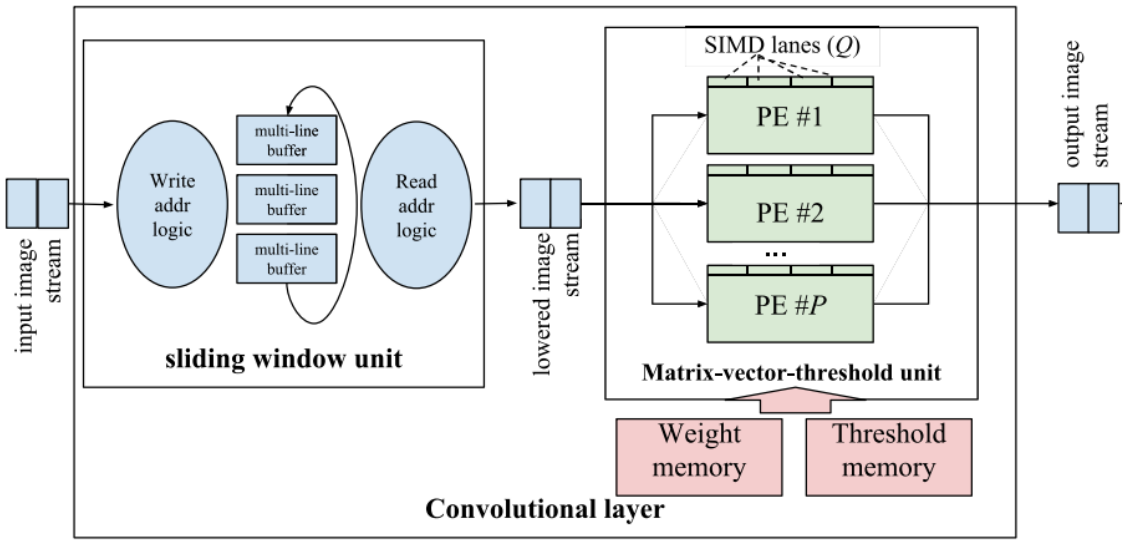


Figure 4.1: Schematic HLS implementation of a convolutional layer in FINN, that contains the so-called sliding window unit and the the Matrix-Vector-Threshold Unit. In contrast, fully-connected layers only contain the Matrix-Vector-Threshold Unit, with no sliding window unit in font. Adapted from [Umu20].

influencing the throughput more indirectly. These are for example the size of the FIFO queues between layers and the frequency setting for the FPGA clock. While those two parameters are also investigated within this work, they are not part of the final optimization algorithm. A more detailed look into those aspects can be found in the discussion about possible improvements, in Section 4.5.

4.2.2 Default settings of FINN

Since FINN comes with multiple network and quantization examples, there are also some default settings available in terms of SIMD and PE parameters. For the binarized convolutional network these settings are shown in Table 4.2. These defaults are designed to run on the PYNQ-Z1 board, which is the smallest FPGA supported by FINN. The implemented algorithm will be mostly demonstrated using binary quantization for the weights and activations, although the algorithm of course works for any quantization setting as shown at the very end of Section 4.4.

Layer group	Conv 1		Conv 2		Conv 3		Fully connected		
SIMD	3	32	32	32	32	32	4	8	1
PE	16	32	16	16	4	1	1	1	5

Table 4.2: Default settings for the SIMD and PE parameters for the convolutional network included in FINN. As highlighted in Figure 3.2 each convolutional block contains two layers, resulting in two sets of parameters per convolutional block. In addition, the network contains three fully-connected layers, resulting in one set of parameters for each fully-connected layer.

4.2.3 Impact on resource usage

While it is clear, how the throughput should behave in dependence on the SIMD and PE parameters, the same could not be inferred about the resource utilization. To understand this, different combinations of parameter settings for the first layer of the convolutional network are tested. In particular, the HLS testbench present in finn-hlslib[2] is employed. Here the test parameters are set to resemble those of the first convolutional layer. To evaluate the resource usage the testbench with the convolutional layer is then synthesized and the resource usages in terms of utilized Lookup Tables (LUTs) and Flip-Flops (FFs) are recorded. Both LUTs and FFs count towards the logic resource budget of a given FPGA in a similar manner as they are both implemented with CLBs on the device.

For visualization the results are plotted once in terms of adjusting the PE parameter (Figures 4.2 & 4.3) and a second time by adjusting the SIMD parameter (Figures

[2] https://github.com/Xilinx/finn-hlslib/blob/5c45a41a755bca534ba737e1b54ec9bbddf42a41/tb/conv3_tb.cpp

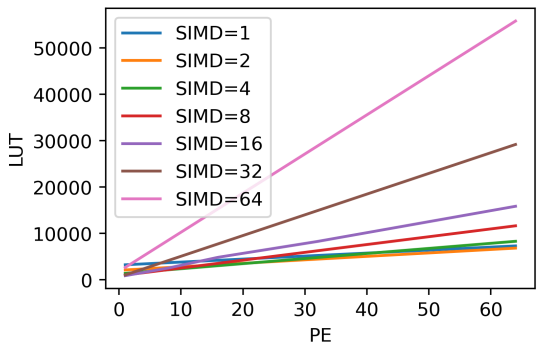


Figure 4.2: LUT utilization after synthesis for varying SIMD and PE parameter, x-axis: PE parameter, y-axis: Number of LUTs instantiated.

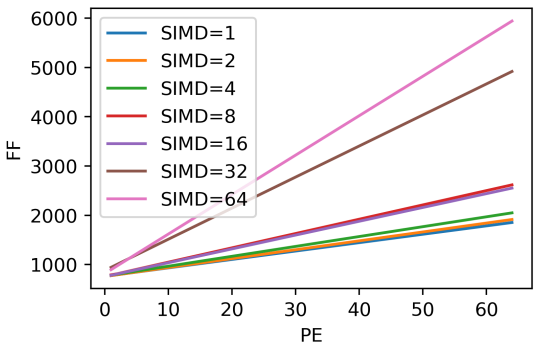


Figure 4.3: FF utilization after synthesis for varying SIMD and PE parameter, x-axis: PE parameter, y-axis: Number of FFs instantiated.

4.4 & 4.5). Note, that both sets of plots contain essentially the same data, but are drawn in different ways to highlight how each parameter influences the resource utilization.

When investigating the PE parameter in Figures 4.2 & 4.3, a linear scaling in both LUTs and FFs is observed. When looking into each line individually it becomes apparent that the SIMD parameter is only responsible for the slope of each curve and its offset.

On the other hand, when investigating the SIMD parameter in Figures 4.4 & 4.5, the linear relation is not as clear anymore. For the FF measurements a monotonic increase can still be observed, though the linearity is broken up at multiple points. This is in contrast to the LUT measurements, where the linear relation is still mostly true for large PE values (greater 16). For smaller values however, an initial decrease followed by an almost linear increase can be observed.

This demonstrates that during optimization of the performance parameters, one can expect that the LUT and FF utilization is mostly in a linear relationship to the SIMD and PE parameter with the PE parameter being more stable and the LUT and FF utilization increasing with increasing values for SIMD and PE.

4.3 Algorithm description

The primary goal of the implemented algorithm is to optimize a given network for the highest possible throughput while staying within a given LUT budget. In a pipelined system, such as a neural network, the throughput is limited by the slowest element in the pipeline, or reformulated for FINN, the primary goal is to keep the maximum layer latency as low as possible. The SIMD and PE parameter are used

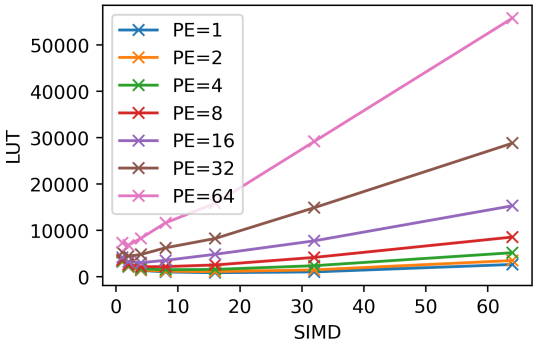


Figure 4.4: LUT utilization after synthesis for varying SIMD and PE parameter, x-axis: SIMD parameter, y-axis: Number of LUTs instantiated.

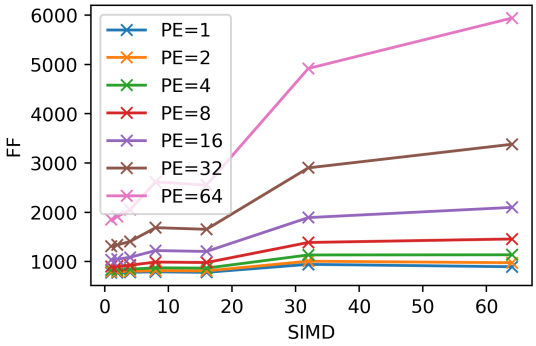


Figure 4.5: FF utilization after synthesis for varying SIMD and PE parameter, x-axis: SIMD parameter, y-axis: Number of FFs instantiated.

as optimization parameters. When looking at different pruning methods in Chapter 5, it also becomes apparent that the investigated pruning methods are linked to the SIMD and PE parameters of FINN. In particular, one approach is expected to achieve higher network prediction accuracy for large SIMD parameters, while the other method is likely to accomplish better prediction accuracy for large PE parameters. However, the default values in FINN follow a more balanced approach, where the SIMD and PE parameter are kept approximately the same. Consequently, a secondary goal for the algorithm is defined, which is to be able to either prioritize large SIMD or PE parameters or to keep both approximately the same.

The algorithm is then designed around the idea to iteratively increase the performance parameters of the specific layer that currently had the highest latency in a network until a previously set LUT budget is met. The algorithm can be described with the pseudo-code in Algorithm 1.

This algorithm is able to fulfill all primary goals defined above. In particular it maximizes the throughput for a given network by reducing the maximum latency in line 2. The secondary goal of flexibly maximizing either the PE or SIMD parameter is implemented in line 3 to 21, while the idea to maximize resource usage and further reduce the latency of the overall network is reflected in lines 8, 14 and 17 to 21. It should be noted that in the final implementation the increase of SIMD and PE parameters in line 3 to 21 is always done in factors of two. This is done because it generally fulfills the modulo constraints for both parameters, which are mentioned in Chapter 4.2.1. Of particular interest is also the choice to first increase the SIMD parameter before increasing the PE parameter in the balanced case. This is primarily motivated by how FINN maintainer Yaman Umuroglu described that

these parameters are normally handled during their internal tuning[3].

Running this algorithm for a given LUT budget generally yields good results, as described in Section 4.4. While the resource budget of a given FPGA is known

[3] https://gitter.im/xilinx-finn/community?at=5fc5226fba0b7a0fc53b54c8

Algorithm 1: SIMD and PE parameter optimization

Data: LUT budget, parameter priority, model to optimize
Result: Optimized SIMD and PE parameters

```
1  Calculate latency for all layers in model
2  Select layer with highest latency
3  try:
4      if parameter priority == SIMD then
5          if SIMD is at maximum then
6              Increase PE of layer
7          else
8              Increase SIMD of layer
9          end
10     else if parameter priority == SIMD then
11         if PE is at maximum then
12             Increase SIMD of layer
13         else
14             Increase PE of layer
15         end
16     else if parameter priority == balanced then
17         if SIMD <= PE then
18             Increase SIMD of layer
19         else
20             Increase PE of layer
21         end
22 if Layer cannot be sped up any further then
23     Select next slowest layer
24     Repeat from line 3
25 end
26 Calculate LUT utilization of all layers
27 if LUT utilization is within LUT budget then
28     Repeat from line 1
29 else
30     Discard the most recent speed-up
31     End algorithm here
32 end
```

precisely, the resource estimates used here suffer from imprecisions. This is expected since synthesis tools for FPGAs can often employ optimization methods which are hard to capture with an analytical model. To work around this imprecision the LUT budget itself is also optimized. This guarantees close to maximum resource usage. The algorithm employed for this is shown as pseudo code in Algorithm 2. The final implementation in *Python* as it is used for the rest of this work can be found in the GitHub repository for this thesis[4] (see also Chapter 9).

Algorithm 2: LUT Budget optimization

```
Data: Initial LUT budget
Result: Final LUT budget
budget = Initial LUT budget;
running = True;
settings = get optimal SIMD/PE for budget;
initalSuccess = test if settings synthesize;
success = initalSuccess;
while running do
    if initalSuccess == success then
        previousBudget = budget;
        if success == True then
            budget = budget + 0.1 * Initial LUT budget;
        else
            budget = budget- 0.1 * Initial LUT budget;
        end
    else
        if success == True then
            return budget;
        else
            return previousBudget;
        end
    end
    settings = get optimal SIMD/PE for budget;
    success = test if settings synthesize;
end
```

[4] https://github.com/HenniOVP/MA_ZITI/tree/main/simd-pe-tuning and https://github.com/HenniOVP/MA_ZITI/blob/main/simd-pe-tuning/cnv_varied_bit_and_pruning_parallel_testing-0.4b_dev.ipynb

4.4 Results

All following tests are run on the Ultra96V2 board, whose resource budget can be found in Table 4.1.

To confirm that Algorithm 1 works as expected and fulfills its primary goal it is first verified that the output in terms of per-layer latency corresponds to expectations. At the start of the optimization low SIMD and PE settings are set as a starting point, much lower than the defaults of FINN. The optimization increased those parameters to fit the LUT budget of the Ultra96V2 board. The resulting latency for each layer is expected to obey to the following relation:

$$T_{\text{start}} \leq T_{\text{default}} \leq T_{\text{end}}, \tag{4.1}$$

with T_{start} being the latency for the initial parameter settings, T_{default} the latency for the default settings by FINN and T_{end} the latency for the final SIMD and PE settings at the end of the optimization.

Additionally, the highest latencies present at the end of the optimization should all be relatively close to each other in value, because the algorithm is supposed to bring individually high latencies down to a common maximum for most latency values.

The measurement result is shown in Figure 4.6. For almost all data points the expected relation according to equation 4.1 is revealed, confirming that the algorithm fulfills its primary goal of reducing the maximum latency and utilizing more of the available resources than managed by the FINN defaults. The only data point for which this is not fulfilled is the *ConvolutionInputGenerator_5*, here $T_{\text{start}} \leq T_{\text{end}}$ is fulfilled, but $T_{\text{default}} \leq T_{\text{end}}$ is not. The reason for this is, that each *ConvolutionInputGenerator* shares its SIMD parameter with the following *StreamingFCLayer_Batch*, because they form one combined convolutional layer. Furthermore, only the *StreamingFCLayer_Batch* exposes both the SIMD and PE parameter, while the *ConvolutionInputGenerator* on the other hand only makes use of the SIMD parameter. When looking at the *StreamingFCLayer_Batch_5* layer following the *ConvolutionInputGenerator_5* layer it becomes apparent that the latency of the *ConvolutionInputGenerator* layer likely was reduced further than required, because the *StreamingFCLayer_Batch* needed significant reductions in latency from its starting point. The low latency of the *ConvolutionInputGenerator_5* layer is then a side effect, because the two layers sharing one common SIMD setting. This unexpected behavior is not detrimental to the network performance, because no increased latency is introduced.

Along with the first expectation, also the second expectation of the latency maxima being close together at the end of the optimization is also fulfilled. This is somewhat in contrast to the default settings of FINN, where the latency maxima are distributed over a wider range.

After demonstrating that the algorithm can accomplish its primary goal, it is tested if it can also correctly prioritize between the SIMD and PE parameter. The

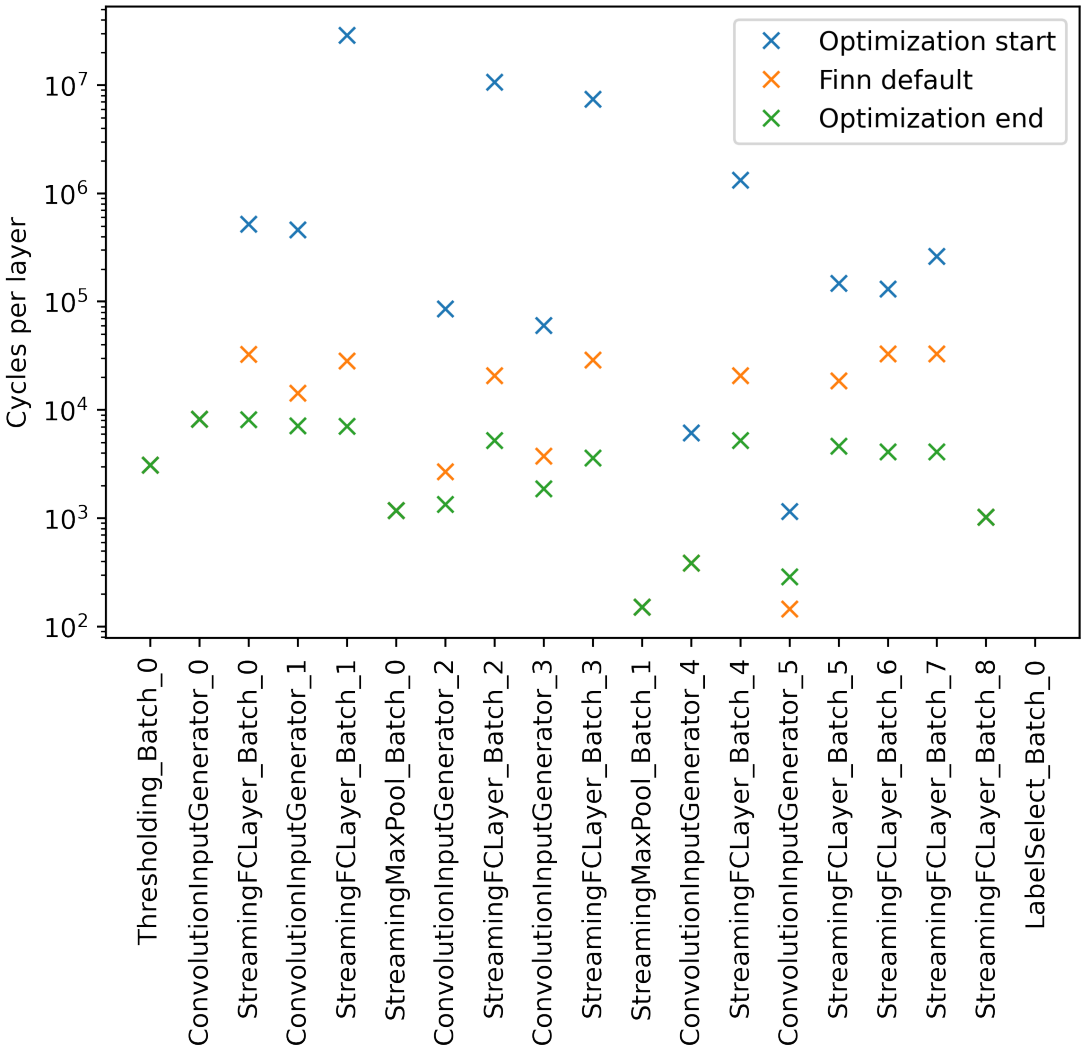


Figure 4.6: Latency for each layer of the convolutional network in binarized form for the balanced parameter prioritization, x-axis: Layer name, y-axis: layer latency in cycles in log-scale, with starting settings in blue, FINN defaults in orange and the finalized optimization in green. The case for *ConvolutionInputGenerator_ 5* is discussed in detail in the text.

results of these tests are shown for the binarized convolutional network in Tables 4.3, 4.4 and 4.5.

In all three cases the values generally fulfill the expectation: For the balanced case (Table 4.3) both values are often of similar size, with the SIMD parameter being favored for individual increments. Here the first and last layers are notable exceptions compared to this expectation. The reason for this can be traced back to the constraints on the maximum size of the SIMD and PE parameter, which are explained in Section 4.2.1. In other words, these exceptions can be viewed as artifacts from the low number of input channels in the first layer and low number

Layer group	Conv 1		Conv 2		Conv 3		Fully connected		
SIMD	3	64	64	64	32	16	8	8	1
PE	64	64	32	64	16	8	4	8	5

Table 4.3: SIMD and PE settings after optimization and balanced parameter prioritization for the binarized version of the convolutional example network. As highlighted in Figure 3.2 each convolutional block contains two layers, resulting in two sets of parameters per convolutional block. In addition, the network contains three fully connected layers, resulting in one set of parameters for each.

Layer group	Conv 1		Conv 2		Conv 3		Fully connected		
SIMD	3	64	64	128	128	128	32	64	1
PE	64	64	32	32	4	1	1	1	5

Table 4.4: SIMD and PE settings after optimization and prioritization of the SIMD parameter, for the binarized version of the convolutional example network. As highlighted in Figure 3.2 each convolutional block contains two layers, resulting in two sets of parameters per convolutional block. In addition, the network contains three fully connected layers, resulting in one set of parameters for each.

output channels in the last layer.

For the two cases in which one of the two parameters is prioritized over the other (Tables 4.4 and 4.5) the prioritized parameter is notably larger than the other. Exceptions are found again in cases in which the SIMD and PE parameter are constrained by the number of input and/or output channels, for example for the second layer, where both parameters are equal to 64.

In conclusion, the test confirms that the developed optimization algorithm for SIMD and PE values can fulfill both the primary and secondary goal.

Following this accomplishment, it is tested if also the second optimization algorithm for the LUT budget (Algorithm 2) correctly adjusts the LUT budget of a given FPGA to compensate for estimate inaccuracies and to improve the resource usage.

For posing the test into the right context: The default SIMD and PE setting shown in Table 4.2 uses about 38% of all available CLBs when a bit-file is generated for the Ultra96V2. To test the algorithm, the convolutional network is synthesized for different bit-widths in terms of activations and weights. The expectation here is that the CLB utilization should be notably higher than for the default settings and in good cases above 90%. The optimized maximal LUT budget on the other hand should vary around 100%.

The results for the CLB utilization are shown in Figure 4.7, while the LUT bud-

Layer group	Conv 1		Conv 2		Conv 3		Fully connected		
SIMD	3	64	16	16	2	4	1	1	1
PE	64	64	128	128	128	16	16	32	5

Table 4.5: SIMD and PE settings after optimization and prioritization of the PE parameter for the binarized version of the convolutional example network. As highlighted in Figure 3.2 each convolutional block contains two layers, resulting in two sets of parameters per convolutional block. Summarily, the network contains three fully connected layers, resulting in one set of parameters for each.

get which is used to generate these bit-files is shown in Figure 4.8. Notable is that not all tested settings can be actually synthesized. These are either left blank in the result plots or marked with 0 LUT budget. The reason for this behavior is that some settings simply do not fit on the FPGA. Affected are in particular all settings close to the largest tested weight and activation bit-width. Additionally, at the time of testing there was still a bug in the FINN framework, which prohibited the synthesis of weights with a bit-width larger than 1 in combination with binary activations. Nonetheless, the expectations for the designed algorithms are fulfilled for all test settings for which the synthesis succeeded. In particular the CLB utilization is consistently high, between 80% and 91%. While the adjusted LUT budget is generally around 100%, ranging from 140% to 70%. The variation in the adjusted LUT budget shows, that there are certain scenarios where FINN can not properly estimate the number of LUTs used after synthesis and place-and-route.

These findings demonstrate that the algorithm for optimizing the LUT budget works consistently well over different bit-widths for weights and activations. Secondarily, Figure 4.8 also highlights that FINN currently struggles to correctly estimate the LUT budget for larger activations and for binarized networks with high SIMD and PE values. These findings were shared with the FINN maintainers at XILINX and work towards improving these estimates is currently underway[5].

In order to further demonstrate that the presented method maximizes the throughput on a given FPGA, the goal is set to find the so-called Pareto frontier between the network prediction accuracy and the throughput on the Ultra96V2.

The Pareto frontier, named after Vilfredo Pareto [Kir16], aims to find possible trade-offs between two related, but not directly comparable metrics. In the case of this work they are the network prediction accuracy and the throughput. Both of these metrics are important, however they are not directly comparable and vary largely with the bit-widths of weights and activations. To find the Pareto frontier all possible settings for the given model are now plotted in terms of their accuracy and throughput. The Pareto frontier is then defined by the number of points, which maximize accuracy and throughput. This leads to a line on which the most efficient

[5] https://gitter.im/xilinx-finn/community?at=5fc8e5d2657e0c48225b7fa4

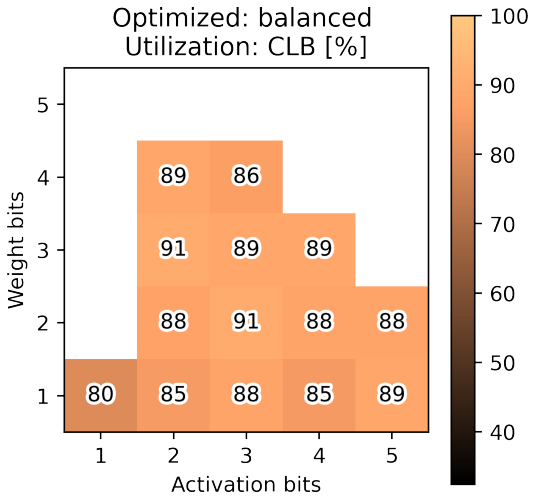


Figure 4.7: CLB utilization for different weight and activation bit-widths on the Ultra96. The utilization is given in percent of the CLB count of the FPGA, shown in Table 4.1.

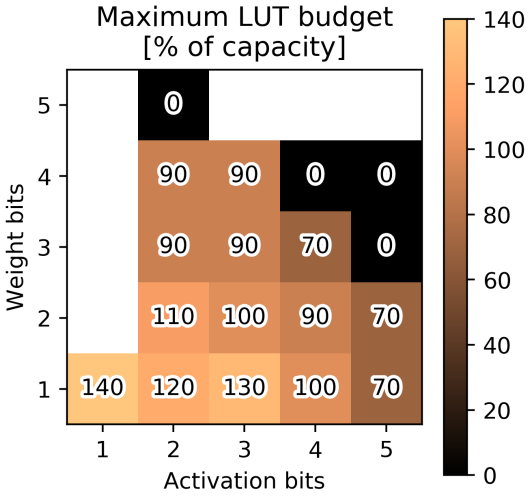


Figure 4.8: Optimized maximum LUT budget for different weight and activation bit-widths on the Ultra96. The LUT budget is given in percent of the actual CLB count of the FPGA, shown in Table 4.1.

configurations lie. Configurations below this line are then considered inefficient and configurations above this line are not realizable, with the investigated method.

The result of this procedure is shown in Figure 4.9. Here the convolutional networks with different weight and activation bit-widths are tuned for maximum throughput using the presented algorithms, locating them on the x-axis for their achieved value of throughput. Then, all running networks are trained using the training script described in Chapter 5, setting their location on the y-axis with the accomplished accuracy. The resulting points at the outer edge towards the top right corner then form the Pareto frontier (solid line) and show how trade-offs between network accuracy and throughput can be made.

In conclusion: These results prove that both algorithms together are able to improve resource utilization for a given FPGA and network, while also maximizing the throughput at the same time. Without the development of these algorithms, many of the following results obtained in Chapter 5 about pruning would not have been possible. Although neither of the two algorithms were integrated into the public FINN project, they still constitute a significant contribution by quantifying inaccuracies for the resource estimation of FINN, which are being investigated by

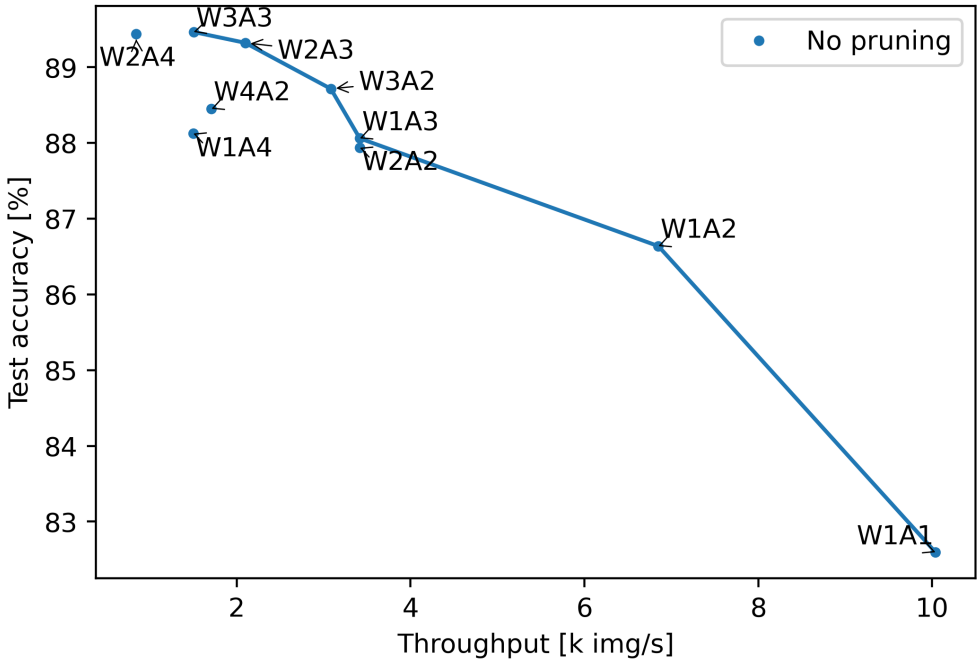


Figure 4.9: Pareto frontier discovery on the Ultra96V2 FPGA. All results (blue) are annotated with their given weight and activation bit-width in the template form WXAY, where X is the weight bit-width and Y the activation bit-width. Additionally, the Pareto frontier is drawn as a solid line, x-axis: Throughput in thousands of images per second, y-axis: Inference accuracy on the test dataset at the end of the training for a given network.

the FINN maintainers[6].

4.5 Possible points of improvement

As mentioned in Section 4.1, the SIMD and PE parameter are not the only settings in FINN, which influence the inference performance. Notably, there is also the size of the FIFOs between layers and the clock frequency. Both of these settings are also investigated extensively during this work, but ultimately neither of them is used in the final result. Details of the investigations are given below.

4.5.1 Automatic clock tuning

The most intuitive parameter left to influence the performance is likely the FPGA clock frequency. This frequency defines how quickly the CLBs within the FPGA can change their state. The throughput is then expected to be directly proportional to

[6] https://gitter.im/xilinx-finn/community?at=5fc8e5d2657e0c48225b7fa4

the clock frequency. This expectation is confirmed by early experiments in hardware and RTL simulations. Figure 4.10 shows one of these experiments when running the convolutional network with two-bit activations and weights and the respective default parameters for SIMD and PE.

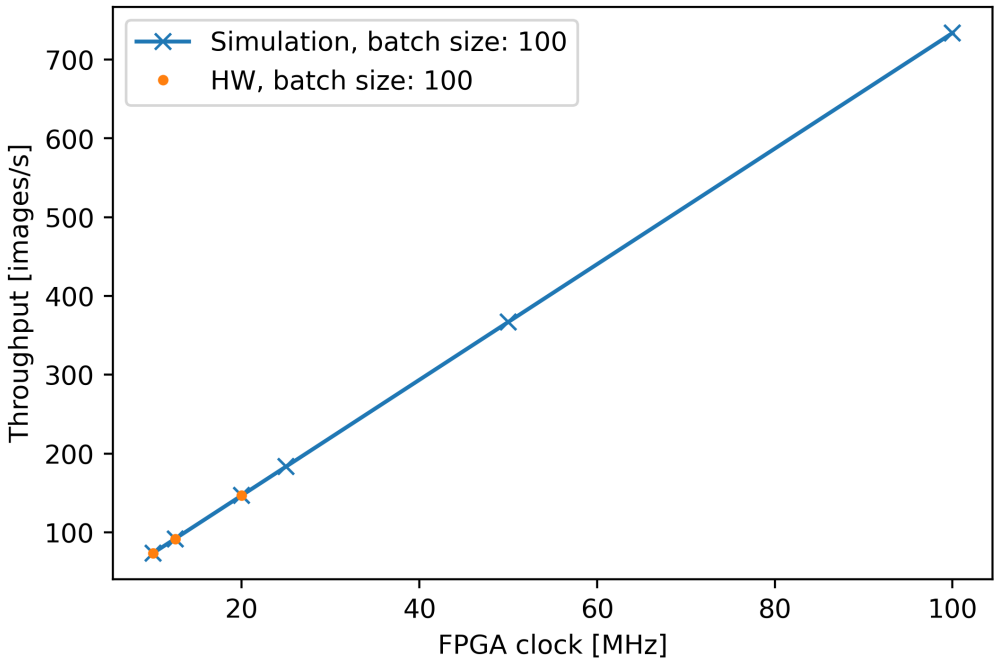


Figure 4.10: Throughput as a function of the FPGA clock frequency for early experiments. Shown in blue are results from RTL simulations and in orange the corresponding results from hardware measurements, x-axis: FPGA clock frequency in MHz, y-axis: Throughput during inference in images per second.

It is likely that increasing the clock frequency would also drastically improve the throughput, due to this measured linearity. As shown in Table 4.1 the Ultra96V2, which is used in almost all experiments supports clock frequencies up to 300 MHz. Thus, a potential three-fold improvement over all results in this thesis, which are run at 100 MHz, could be expected.

As an extension to Section 4.2.3, an investigation into the resource usage for varying clock frequencies is performed. Similarly, Figure 4.11 and 4.12 show the LUT and FF utilization when varying the clock period and the SIMD parameter. The FF usage increases with clock frequency, because more buffering is required to meet the more stringent timing requirements. On the other hand, the LUT utilization shows no changes at all, which promises that higher clock frequencies may be reached with little resource overhead.

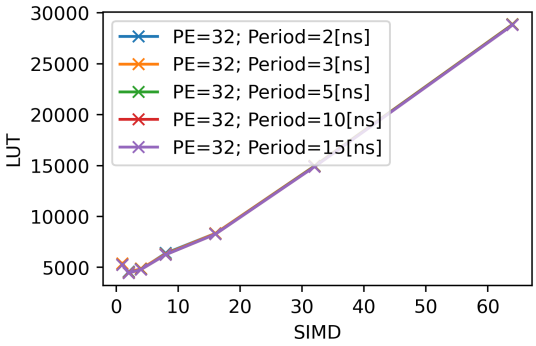


Figure 4.11: LUT utilization after synthesis for varying the SIMD parameter and clock period, while keeping the PE parameter fixed at 32, x-axis: SIMD parameter, y-axis: Number of LUTs instantiated.

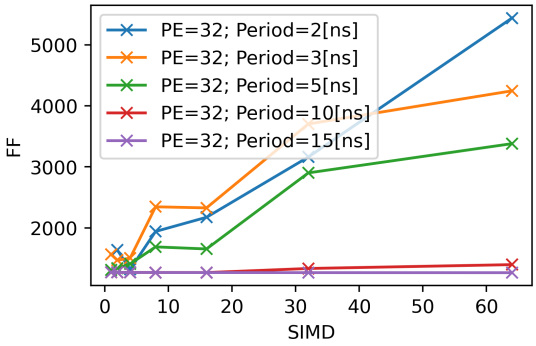


Figure 4.12: FF utilization after synthesis for varying the SIMD parameter and clock period, while keeping the PE parameter fixed at 32, x-axis: SIMD parameter, y-axis: Number of FFs instantiated.

These very high frequencies are generally hard to reach for complex circuit designs, such as those produced by FINN. The time between clock cycles can become too short to perform all required calculations on a signal, before the result needs to be available on the next clock cycle. This behavior is characterized by the Worst Negative Slack (WNS). It quantifies by how much time the data signals within the FPGA lack behind the clock signal in the worst case. By default, the WNS is reported by *Vivado* after a given design is placed and routed for a given FPGA. The WNS is positive when all signal computations are completed faster than one clock period and the size reflects how much faster these calculations are completed. However, when the WNS becomes negative, then some calculations are not completed within one period of the FPGA clock. The routed circuit might still run, but errors which are hard to debug will often surface. From this metric the theoretical maximum frequency for a given design can be calculated as follows:

$$F_{\text{max}} = \frac{1}{T_{\text{period}} - T_{\text{WNS}}}, \tag{4.2}$$

with F_{max} being the theoretical maximum frequency, T_{period} the clock period for the placed and routed design and T_{WNS} the WNS for the same design.

Because the WNS also varies with the FPGA clock, equation 4.2 can only be taken as an estimate. Consequently, a potential optimization of the clock frequency must take an iterative approach to find the actual F_{max}, likely similar to the LUT budget optimization in the Algorithm 2 realization. This would likely increase the overall optimization time, since multiple synthesis runs must be started to find a

good value for $F_{\max}$.

Results from tests made for Chapter 5 reveal that the WNS for a clock frequency of 100 MHz can be as high as 3.0 nanoseconds, suggesting a potential improvement in throughput of up to 42.8%.

This potential improvement makes the approach a very interesting topic to investigate in the future. Unfortunately, during the investigation in FINN 0.4b a bug was discovered, which locked the clock frequency for the Ultra96V2 to 100 MHz[7]. Due to time constraints in this work and this aspect being a topic of lower priority it was not attempted to fix the bug.

As a consequence, all results in this work are accomplished with an FPGA clock frequency of 100 MHz.

4.5.2 Automatic FIFO buffer sizing

Another approach to improve performance is to optimize the size of FIFOs between layers. In theory, the pipeline which FINN produces to run on an FPGA does not require FIFOs to run. However, some layers show burst-like behavior in how they output their result data[8]. This can result in the stalling of the following layers caused by input data not being constantly available, causing an effectively decreased throughput, because some layers are now idling at certain times. This issue can be remedied by introducing FIFO buffers after bust-like behaving layers. Data stored in these buffers can then be output on demand, which reduces the idle time of downstream layers and improves the overall performance.

This behavior is reproduced in our early experiments. Figure 4.13 shows a particularly interesting experiment. Here all FIFOs are set to the same size. This size is then varied from 5 to 255. Because these designs are generally too large to run on one of our FPGAs, the throughput is measured by running the designs in a dedicated RTL simulation. The results clearly demonstrate that for most of the measurement range the throughput increases linearly with increasing FIFO depth, confirming the expectations. At a FIFO depth of about 230 the throughput then saturates, likely due to the FIFOs already absorbing all irregularities from burst-like behaving layers. Below FIFO depths of about 50 a somewhat irregular behavior is revealed, which might indicate that in this region multiple layers are affected by the small FIFO sizes.

These early results already indicate that tuning the depth of individual FIFOs is likely a good measure to increase the performance for a given network overall.

While this is not investigated further in this work, the most recent release of FINN (0.5b) includes preliminary support for automatically setting these FIFO depths[9]. The method works by first setting all FIFOs to a defined maximum size, then the network is simulated on RTL to find out how far the FIFOs fill up at most. After-

[7] https://github.com/Xilinx/finn/issues/241

[8] https://gitter.im/xilinx-finn/community?at=5f0879148342f46274083650

[9] https://xilinx.github.io/finn//2020/12/17/finn-v05b-beta-is-released.html

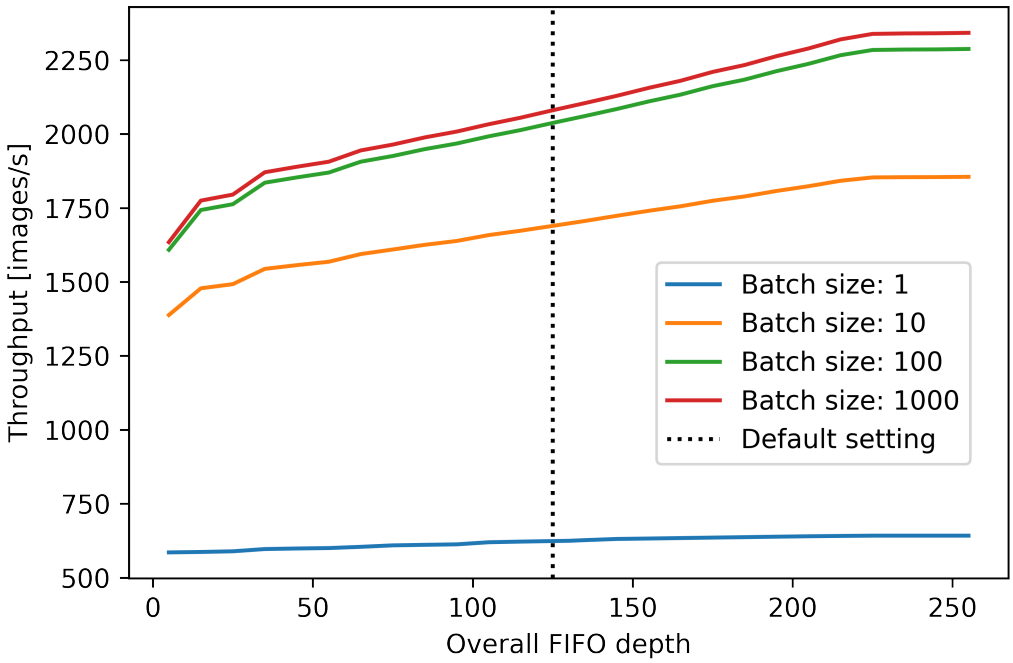


Figure 4.13: Throughput as a function of the FIFO depth for different batch sizes. The horizontal line labeled "Default setting" shows the maximum FIFO depth set as FINN defaults, x-axis: FIFO depth for all FIFOs in the network, y-axis: Throughput for RTL simulations in images per second.

wards the FIFOs are shrunk back down to their measured, maximum fill state. And finally, miscellaneous edge-cases are handled.

In tests within the experiments of this work, this method showed particularly good results for the binarized convolutional network. Compared to the default settings, the throughput increased by about 19% for the binarized network and for higher bit-width networks improvements of up to 18% are measured, though sometimes changes as low as 2% are seen.

While these are in general important results, the automatic FIFO sizing is not used within this thesis. Mainly because the method requires significant time investments due to the relatively slow RTL simulation involved. In addition, the trade-off between increased performance and time was not worthwhile for the already extensive experiments performed for Chapter 5.

5 Pruning in FINN

With neural network architectures becoming increasingly complex over time, the inference on low-power devices becomes more difficult and slower. To combat this issue parameter compression can often be employed to improve inference throughput. Popular compression methods are for example pruning ([Han+15] and [GYC16]) and quantization, see [Zha+18].

While FINN was built with support for quantization from the beginning, support for pruning is currently lacking.

This chapter aims to explore the feasibility of implementing support for sparse weight tensors and methods to generate an envisioned sparsity through pruning in FINN. Furthermore, different implementation approaches are tested and different sparsity structures investigated. These are further analyzed in terms of the trade-offs in the accuracy of the prediction and the throughput performance.

5.1 Convolutions in FINN

Convolutional operations, as they are often employed in modern neural network architectures, can be reformulated as matrix multiplication operations, see [CPS06]. The same approach is realized in FINN, see [Umu+17]. The implementation is shown schematically in Figure 4.1. In practice this means that in FINN convolutions are split into two operations: First the input is converted into an image matrix by employing the method of a sliding window unit. This operation is often simply called image2col and the FINN HLS implementation is called *ConvolutionInputGenerator*. Afterwards the image matrix is multiplied with the weight matrix in the MVTU, the corresponding HLS implementation in FINN is called *StreamingFCLayer_Batch*.

For the study of the impact of sparsity the focus is set on modifying the image2col operation of FINN. In this way the image matrix and the weight matrix can be shrunk at the same time, while the matrix multiplication of the MVTU is left untouched. This keeps the pruning implementation relatively simple, while leaving most of the required calculations to the already existing MVTU. An additional benefit is that leaving the MVTU untouched preserves all performance optimizations, which already went into this central part of FINN.

5.2 Explored pruning approaches

Both methods, the fine and coarse-grained method, presented within this work modify which information is given as output from the image2col unit in FINN. Most of

the original implementation is left untouched and data is only discarded directly before it would normally be sent to the following MVTU. This allows for the introduction of structured sparsity.

While both the fine and the coarse-grained method use this basic design principle, the sparsity patterns which they can produce vary a lot.

5.3 Coarse-grained pruning

The coarse-grained pruning method is the first pruning method developed and implemented in this work. In many aspects it can be described as the path finder method for the fine-grained pruning method, because in general all implementations and experiments are first completed with the coarse-grained method, before moving forward with the fine-grained method.

5.3.1 Implementation

As noted previously the implementation modifies the image2col operation of the finn-hlslib, where this operation is called *ConvolutionInputGenerator*. Thus, a new HLS function with the name *ConvolutionInputGeneratorPruned* is designed for the first pruning method. Additionally, a testbench is designed to verify functionality in *C++* and RTL simulations. The implementation[1] and testbench[2] are published in Hendrik Borras's public fork[3] from the official repository[4]. Additionally, changes to code in the FINN main repository had to be made to realize the integration into the FINN framework. These changes and additions were made on a fork[5] to the development branch of the FINN repository. At the time of forking, the development branch was just about to be merged into the master branch for release *0.5b*. As such the fork by Hendrik Borras includes all features of release *0.4b* and most features of release *0.5b*. A complete overview of where the code developed within this work can be found given in Chapter 9.

Schematically the implementation is shown in Figure 5.1. At first the input image is read from an HLS stream[6], the main method for the HLS components in FINN to communicate with each other. The received image data is then saved into an internal buffer. At the same time data is also read from the buffer. During this second read process the image data is reordered to fit the column format required by the following matrix processing unit. Finally, the reordered data is written to an

[1] https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/slidingwindow.h#L331

[2] https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/test_swg_pruned.tcl and https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/swg_pruned_tb.cpp

[3] https://github.com/HenniOVP/finn-hlslib/tree/feature/col_pruning

[4] https://github.com/Xilinx/finn-hlslib

[5] https://github.com/HenniOVP/finn/tree/feature/0.4_cutting_pruning

[6] https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_stream_library.html

HLS output stream, which is connected to the matrix processing unit. Both input and output streams can be implemented as FIFO queues and can optionally fulfill buffering purposes as discussed in Chapter 4.5.2.

The above description does not yet include any pruning and mirrors how the image2col operation is already implemented in FINN. Additional instructions are added, which enable or disable the write call to the output stream. This effectively skips column data in the output matrix, pruning the data for the next layer. The algorithm is then setup such that the user can define which columns to skip, better called to be pruned, by setting an external Boolean array, called *ColsToPrune*.

Of course, for the correct setup the user needs to know how many columns the *ConvolutionInputGeneratorPruned* expects to transmit for the whole image processing. At this stage a fundamental limitation in the resulting sparse structure becomes apparent. The *ConvolutionInputGenerator* is capable of outputting multiple columns in parallel. To enable this feature the width of the output HLS stream is modified to fit multiple values at the same time. Consequently, also the write instruction to the stream requires to write multiple columns in one go. The number of columns written at the same time is controlled by the SIMD parameter, introduced in Chapter 4.2.1 and is an essential parameter for controlling the amount of parallelization present in the matrix processing unit. As demonstrated in Chapter 4.4 the SIMD parameter can become very large, especially for binarized networks.

The *ConvolutionInputGeneratorPruned* will thus generally need to output multiple columns at once. Because the pruning method turns this output on and off, this inherently limits the granularity with which image data are pruned. Meaning, that for SIMD > 1 blocks of adjacent columns are pruned. Consequently, the pruning method is called "coarse-grain", because the resulting structures of sparsity are limited in minimum size and are inherently coarse. This limitation was clear already at the early design stages and it was made sure to adapt other parts in the design layout to better cope with the predicted limitation. As such, this limitation is the primary driver for introducing flexible parameter prioritization in the design of the automatic tuning algorithm for the SIMD and PE parameter in Chapter 4.3.

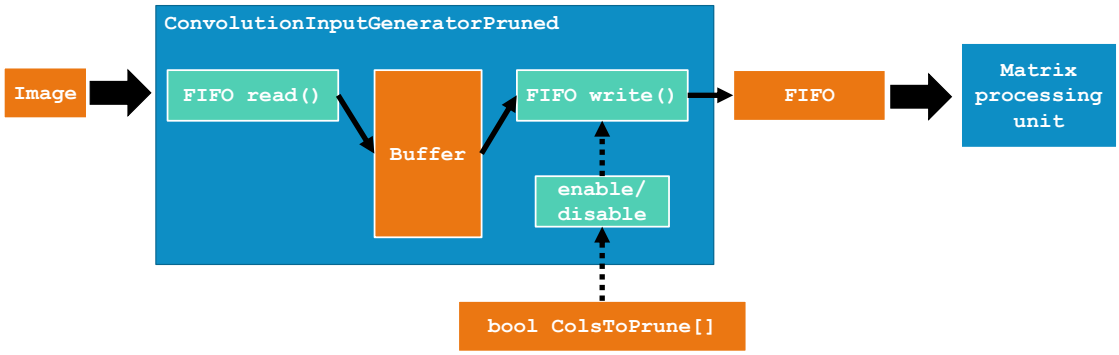


Figure 5.1: Schematic of the image2col operator for coarse-grain pruning, as implemented in HLS.

For the coarse-grain pruning method specifically, the prioritization of the PE is favored, because a large PE parameter implies a small SIMD parameter. And a small SIMD parameter allows for more fine-grained sparsity with this method. The sparsity becomes finer, because the number of columns in one block, which is pruned, is reduced.

5.3.2 Impact on throughput

It is expected for pruning methods, which do not adjust the performance parameters of FINN, that a given convolutional network should become faster by increasing the sparsity, because less calculations are needed to be performed for each image.

This expectation is confirmed in experiments, where the amount of pruned data is adjusted from 0% to 90% in 10% increments. To run these experiments without any training, the *ColsToPrune* array is filled with random data, which adhered to the set pruning percentage. The corresponding plot is shown in Figure 5.2.

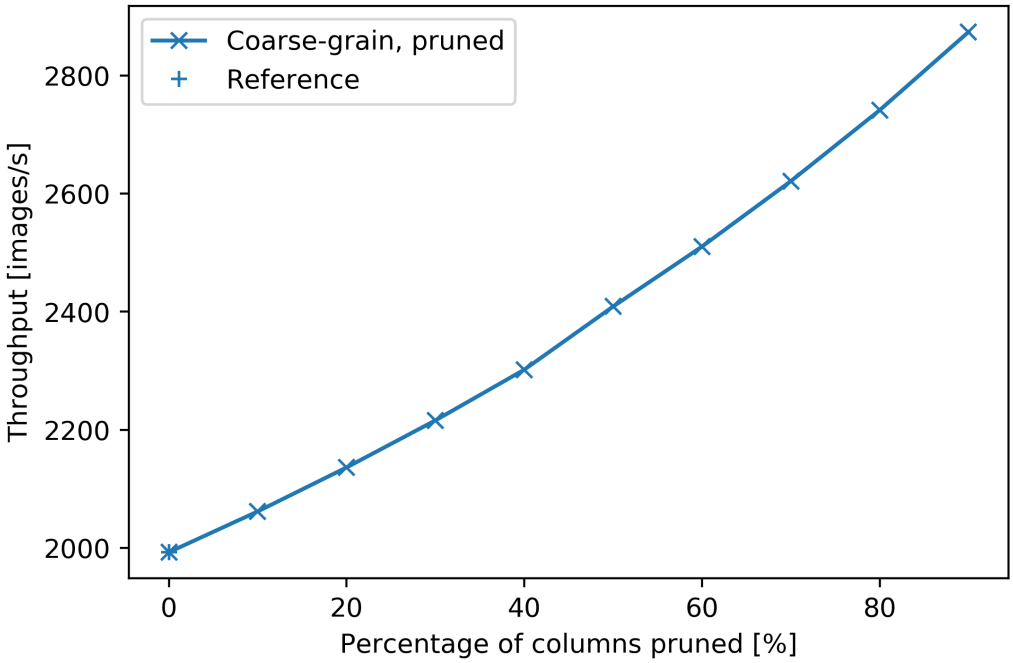


Figure 5.2: Throughput as function of the pruning percentage for the binarized convolutional network with default performance parameters. The throughput is measured on hardware (blue line and "x"). Additionally a reference measurement with the original implementation of the *ConvolutionInputGenerator* is shown with a blue "+", x-axis: Percentage of available columns pruned in 10% steps, y-axis: Throughput in images per second.

Here the throughput for the binarized convolutional network is measured for different percentages of columns pruned. To keep the performance consistent, the default SIMD and PE parameters (Table 4.2) are used. The throughput is measured on

hardware (blue line and "x"). Additionally, to confirm consistency with the original implementation, the throughput is measured with the *ConvolutionInputGenerator* instead of the *ConvolutionInputGeneratorPruned* (blue "+").

As expected the throughput scales in an almost linear relation with the percentage of data pruned, exhibiting a slightly increasing slope with increasing pruning percentage.

5.3.3 Impact on FPGA resources

The primary impact of this method is anticipated to materialize in terms of freeing up parts of the BRAM, because the size of the weight tensors is reduced. A linear reduction with the pruning percentage is forecasted, since the weight tensors are shrunk proportionally to the amount of sparsity achieved through pruning. In terms of utilization of logic resources on the FPGA, such as for CLBs and LUTs, no change at all is expected, primarily, because no changes to any performance parameters are made. A notable exception might be the logic memory, since the *ColsToPrune* array is assumed to take up additional space.

For all resource measurements focus is put on metrics which show utilization larger than 10%, because for these measurements resource requirements below that threshold are generally not critical.

The results in terms of BRAM and logic utilization for one such measurement are shown in Figure 5.3 and 5.4. Here the binarized convolutional network is tested with varying percentages of pruning. Just as before, the defaults for the performance parameters are used to make the results comparable to each other. Since no pruning data is trained at this point in time, the *ColsToPrune* array is filled with random data, such that it matches the desired pruning percentage.

As expected the BRAM utilization exhibits significant reductions in Figure 5.3. in particular in the metric for the *Block RAM Tile*, which reflects the overall BRAM utilization. Similar behavior is revealed for *RAMB36/FIFO*, which corresponds to how the utilized memory is allocated. As expected, a linear decrease for this parameter can be found for pruning percentages above 50% and slightly slower linear decrease with smaller slope between 20% and 50%. Below this threshold of 20% however, there is a sudden step in utilization between 10% and 20%. This behavior is likely due the process of allocation of BRAM on the FPGA. A given BRAM tile (*Block RAM Tile* metric in Figure 5.3) can only be allocated as a single element of 36 kb memory (*RAMB36/FIFO* metric in Figure 5.3) or two elements of 18 kb memory (*RAMB18* metric in Figure 5.3) [Xil21a, p.7]. Each *StreamingFCLayer_Batch* layer in FINN allocates it's own number of BRAM elements for storing the necessary weight matrices. As shown in Figure 5.3 the BRAM tiles are mostly allocated as 36 Kb elements. This also means that for each *StreamingFCLayer_Batch* layer the pruning has to shrink the weight matrix by 36 Kb before any BRAM tiles can be freed up, in the worst case. So much memory is likely not freed with the first 10% of sparsity and thus no change is observed, even if in fact the implementation uses less memory. The very large step from 10% utilization to 20% also fits within

this explanation, because it is likely that after initially no elements were freed up, now suddenly a lot more can be left unused. In other words, the non-linearity seen below 50% sparsity are likely a side effect of how the BRAM allocation is not continuous, but quantized into distinct steps. While this is a potential and very fitting explanation to justify the BRAM behavior, it might be worthwhile to follow up with a more in-depth study in the future.

Figure 5.4 shows how the usage of logic resources develops for the same network depending on the amount of sparsity. As expected the CLB usage is nearly constant, as are most other metrics. A notable exception to this expectation is the utilization of LUTs, which are instantiated as memory (*LUT as Memory*). Here the utilization actually increases with sparsity. The reason for this is likely that with more sparsity more random information gets stored in the *ColsToPrune* array, which then might take up significantly more space.

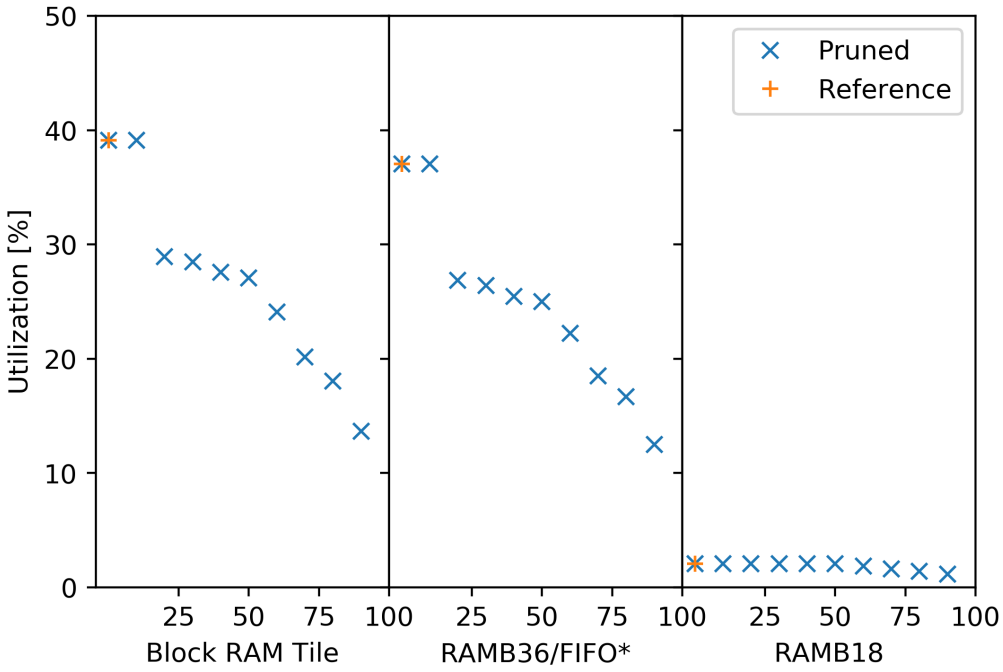


Figure 5.3: BRAM utilization for different metrics as a function of varying amounts of sparsity for the corase-grained pruning method on the binarized convolutional network, x-axis: Sparsity for three different metrics, y-axis: BRAM utilization in percent.

*Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E2 or one FIFO18E2. However, if a FIFO18E2 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E2.

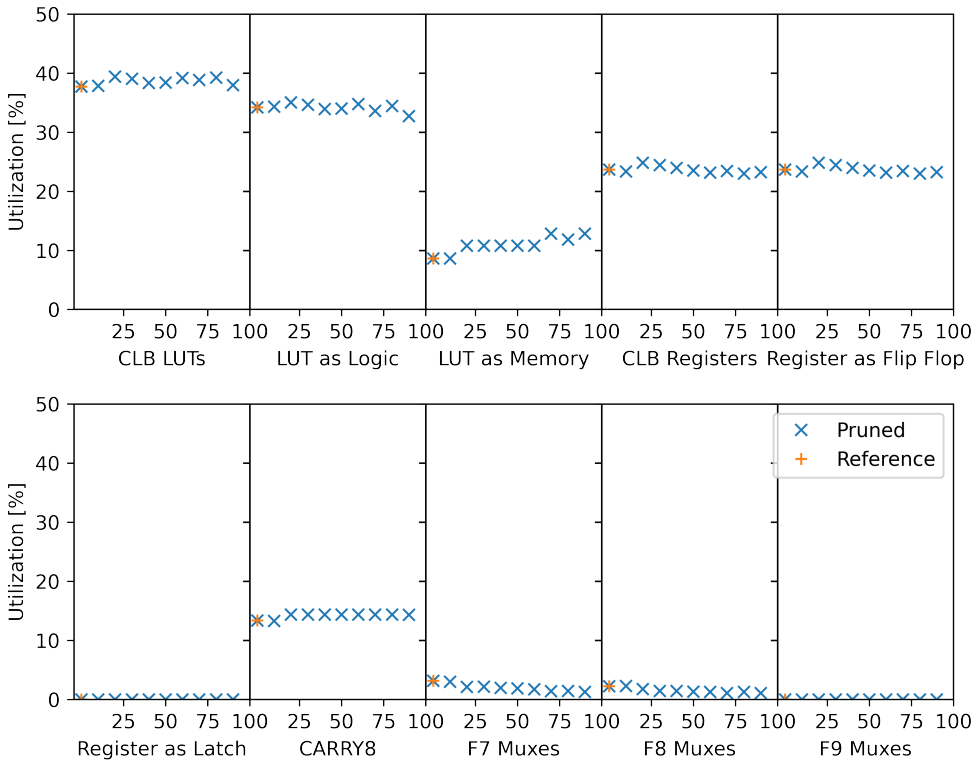


Figure 5.4: Logic utilization as a function of varying amounts of sparsity for the corase-grained pruning method on the binarized convolutional network, x-axis: Sparsity for ten different metrics, y-axis: Logic utilization in percent.

5.3.4 Training methodology and network accuracy

For evaluating the accuracy of the tested pruning methods a pruning class is implemented for *PyTorch* and the models are trained using *PyTorch* and *Brevitas*.

Since *Brevitas* and FINN are designed to work together, the code for training the sample convolutional network of FINN was already published in the *Brevitas* repository [Ale+21]. This code is employed as the basis for implementing the network training. To be specific, *Brevitas* version 0.2 and *PyTorch* 1.4 are used. The final training script created during this work can be found in the accompanying repository[7] (see also Chapter 9).

In a first attempt a random pruning method is implemented. This method first calculates which structures the pruning method could create by considering the size

[7] https://github.com/HenniOVP/MA_ZITI/blob/main/training/Brevitas_train_pruning_from_FINN_json.py

of the feature map and the SIMD parameter of a given layer. Then the method randomly prunes some of those structures in the amount corresponding to the given percentage of sparsity. The model is then trained for a given sparsity setting for 200 epochs and, in each epoch the accuracy is evaluated on a test portion of the CIFAR10 data set [KNH]. CIFAR10 is a data set commonly used for bench-marking machine learning algorithms for visual image recognition. It contains 32x32 pixel color images, that are divided into 10 categories. For each category there are 6000 individual images.

The accuracy (acc) here is defined as the number of correctly classified samples (C), divided by the total number of samples (N),

$$\text{acc} = \frac{C}{N}. \tag{5.1}$$

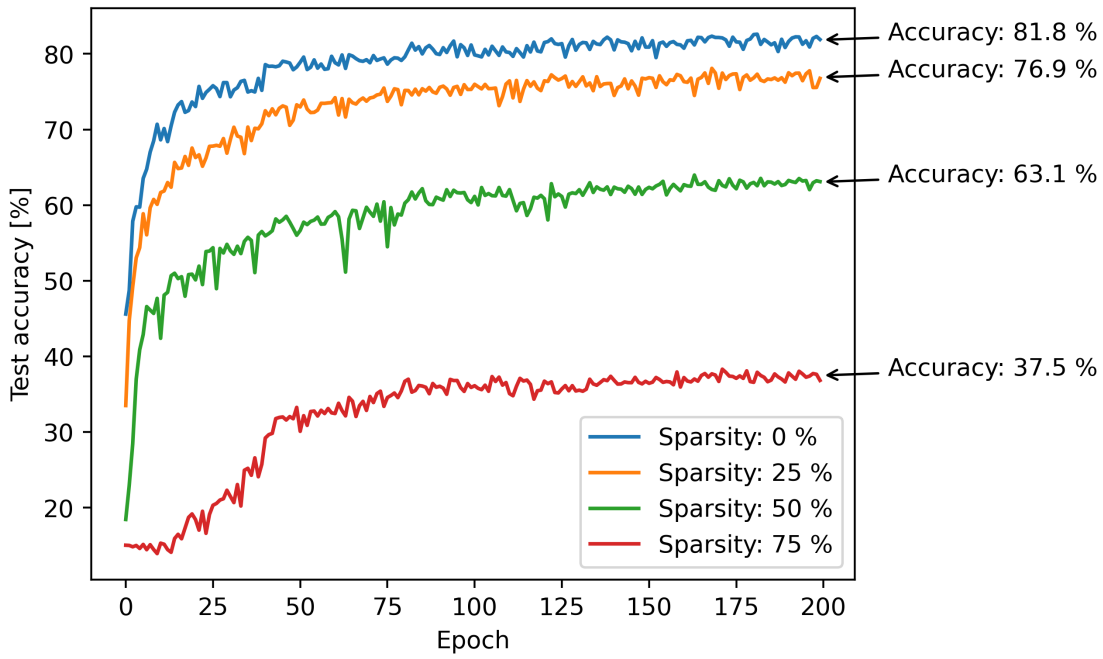


Figure 5.5: Random training method for coarse-grain pruning: Test accuracy as a function of the epoch number for different sparsity settings, x-axis: Epoch number, y-axis: Test accuracy in percent.

Figure 5.5 shows the training results from this first experiment. Here the binarized version of the convolutional network is trained with all SIMD parameters set to 4, which is relatively close to a best-case scenario for this pruning method. For each sparsity setting a new training run is started and the final test accuracy is calculated by averaging over the last ten epochs and is shown on the right side of the plot. As expected the accuracy significantly decreases with increased amount of sparsity.

The experiment reveals that selecting structures randomly for pruning results in sub-optimal accuracy. To combat this to a certain degree it was decided to change

the method to select blocks of columns to be pruned. A so-called ℓ^1-norm approach, as outlined by [Sch+20] and [Mao+17] is chosen. The ℓ^1-norm is defined as:

$$\|\mathbf{x}\|_1 := \sum_{i=1} |x_i|, \tag{5.2}$$

with $\|x\|_1$ the ℓ^1-norm of the vector $\mathbf{x}$ and x_i the individual values within this vector. The idea is that weights with smaller ℓ^1-norm should contain less important information and can be pruned more safely resulting in a smaller loss in accuracy.

The ℓ^1-norm of the weights for each block of columns is calculated and in the new method, the blocks with the lowest ℓ^1-norms are pruned. This method requires that the weights of a given network are already trained before any pruning occurs. Therefore, an iterative approach is followed to train first and then prune a given network. The Pseudo-Code for this iterative method is shown in Algorithm 3.

Algorithm 3: Iterative pruning

Data: finalSparsityAmount [0,1], NumStartEpochs, NumIntermediateEpochs, NumStopEpochs
Result: Pruned and trained model

```
currentAmount = 0.1;
model = create neural network model;
model.train(epochs=NumStartEpochs);
while currentAmount < finalSparsityAmount do
    model.prune(amount=currentAmount);
    model.train(epochs=NumIntermediateEpochs);
    currentAmount += 0.1;
end
model.prune(amount=finalSparsityAmount);
model.train(epochs=NumStopEpochs);
return model;
```

Here, a network is first trained for a set number of epochs *NumStartEpochs* before it is pruned to achieve 10% sparsity. Then the network is trained again for *NumIntermediateEpochs* before being pruned by an amount 10% higher than the previous one. This intermediate process of training the network for *NumIntermediateEpochs* then pruning it by increasing amounts is repeated until the targeted amount of sparsity (*finalSparsityAmount*) would be overstepped by the intermediate sparsity amount. In the final step of Algorithm 3, the network is first pruned by *finalSparsityAmount* and then trained for *NumStopEpochs*. The result is a trained network with a sparsity of *finalSparsityAmount*.

By using the combination of ℓ^1-norm pruning and the iterative approach it is possible to achieve drastically improved performance. A direct comparison to the previous procedure to pruning is shown in Figure 5.6.

Here the random pruning approach from before is marked in blue, while the iterative procedure is shown in green. The parameters of *NumStartEpochs*, *NumIntermediateEpochs*, *NumStopEpochs* of Algorithm 3 are all set to 300 epochs in this initial test. Although all 10% steps in sparsity are trained for the iterative method, only results for 0%, 25%, 50% and 75% are shown in the picture to enable clear visibility. Note that some statistical fluctuation between multiple training runs occurs and thus the results for 0% sparsity do not line up exactly.

As expected the ℓ^1-norm pruning method vastly outperforms the randomized one, by up to 8.4%. By comparing the shape of the curves between both methods it can be inferred that the iterative method benefits from information learned in previous iterations. While the random method always starts at a relatively low accuracy percentage at epoch 0, the iterative procedure starts closer to what the final accuracy is for that sparsity percentage. This is particularly apparent for the sparsity setting of 75%, where the random method starts at about 10% accuracy and ends at 37.9%. Here, the iterative method already starts at about 38% accuracy and ends at 46.3%.

However, the iterative approach is not without issues itself. In particular to compute the results of Figure 5.6, the training time practically explodes. For a sparsity of 75%, the iteratively trained network needs to be trained eight times longer than the one with the random approach: The network needs to be retrained more often until the final pruning percentage is reached.

Luckily, the iterative Algorithm 3 can be adjusted to significantly speed up the

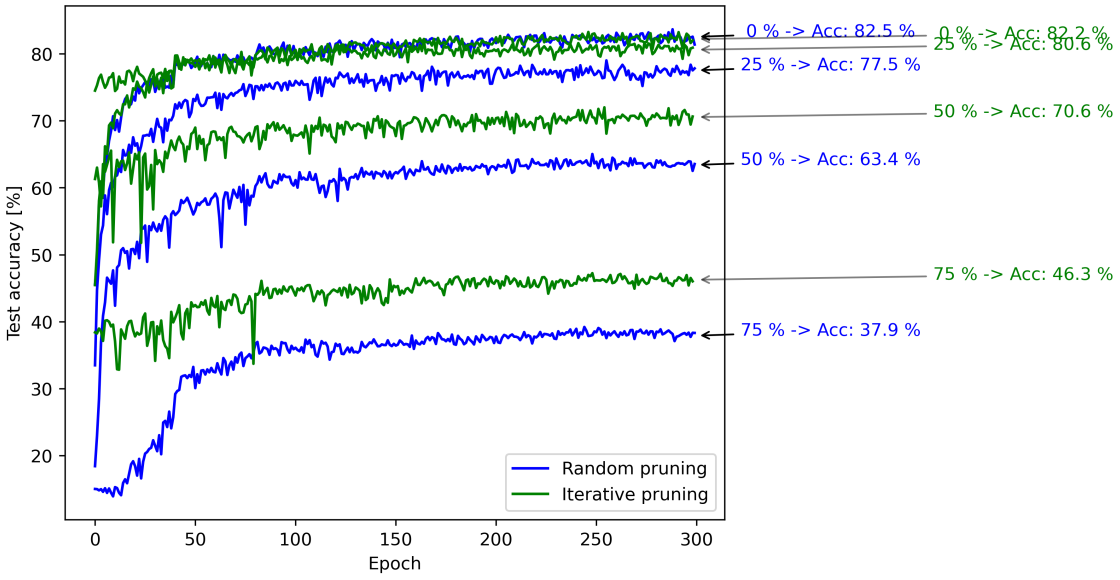


Figure 5.6: Comparison of training results for the random pruning method (blue) and iterative ℓ^1-norm pruning method (green) for coarse-grain pruning. For each sparsity setting the percentage of sparsity and final accuracy is shown on the right in the corresponding colors, x-axis: Epoch number, y-axis: Test accuracy in percent.

training process for a given network. In particular the number of epochs for the intermediate sparsity settings can be significantly reduced, because these networks do not need to be fully trained. The results of each intermediate training are only used for a coarse adjustment to the remaining non-pruned weights, and the next pruning step can be taken, because the pruning structures are relatively coarse and are only accounting for large changes in the weights. The same argument can be applied to the initial training at 0% pruning. However, since this first setting trains from completely random weights without any prior training this first setting is trained slightly longer than the intermediate ones, as shown below.

For all further experiments the following settings are adapted for the number of epochs per sparsity percentage:

Parameter	Setting	Description
NumStartEpochs	100	for 0% sparsity
NumIntermediateEpochs	50	for intermediate sparsity settings
NumStopEpochs	300	for the final sparsity setting

Table 5.1: Parameter settings for Algorithm 3.

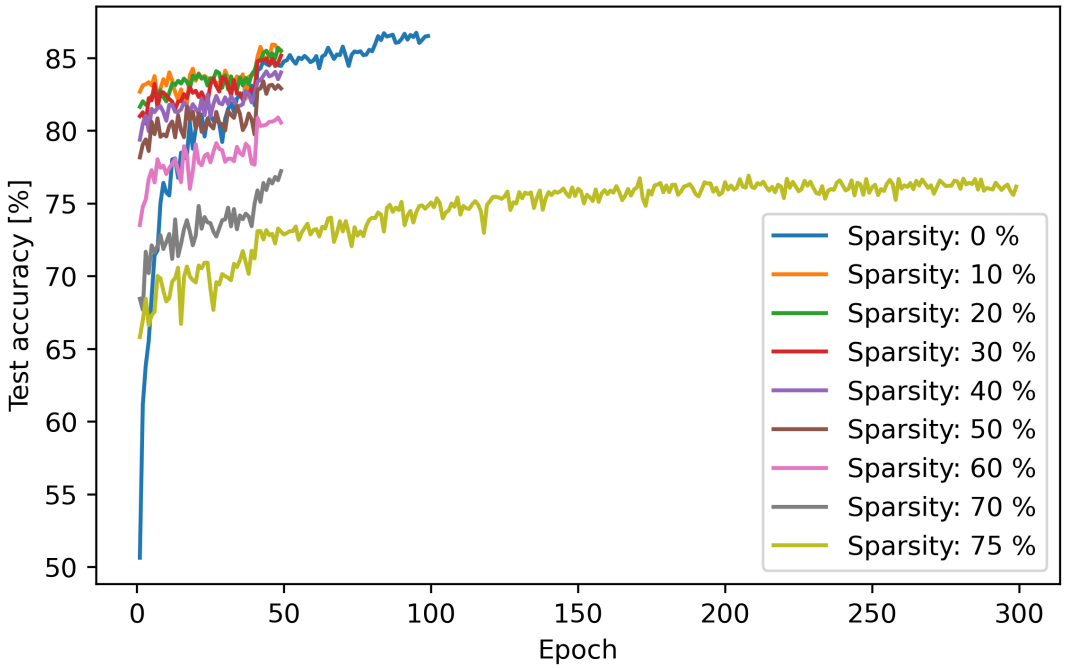


Figure 5.7: Training procedure for the iterative ℓ^1-norm training method for coarse-grain pruning. Each sparsity setting is shown as an individual curve. Here a network with 3-bits weights and 2-bits activations is trained, x-axis: Epoch number, y-axis: Test accuracy in percent.

The resulting training plot with the modified parameters from Table 5.1 is shown

in Figure 5.7. Here the convolutional network is setup with 3-bits weights and 2-bits activations. In contrast to Figure 5.6 all intermediate settings are shown as well.

The resulting training procedure now runs significantly faster, only running 1.5 times longer than the equivalent randomly pruned training. This leads to a speed-up of 3.6 times compared to the initial settings for Algorithm 3.

5.3.5 Exploring network design space for the Ultra96V2

By now the pruning method is implemented as a proof of concept in both FINN and *Brevitas*, such that tests with different network settings can be performed.

In consequence of this implementation, the training script is adapted to run with data produced by the automatic performance tuning algorithm described in Chapter 4. And a workflow could be established, that allows to first explore a given model design on hardware and then to train the resulting model settings with *Brevitas*. **Notably, no end-to-end training took place, however, tests in hardware are made with random pruning data to explore the corresponding SIMD settings and these settings are then trained in *Brevitas*.**

The workflow is then used to run an exhaustive search over multiple dimensions. For the coarse-grain pruning these are as follows:

- Weight bit-widths: 1 to 5 bits
- Activation bit-widths: 1 to 5 bits
- Sparsity percentages: 25%, 50%, 75% and 87.5%
- Performance parameter priority: Balanced and PE (see Chapter 4.3)

Excluding combinations of 1-bit activations and multi-bit weights, which did not synthesize in FINN, this search tested 168 network settings. Although not all of the combinations ran on the hardware provided by the Ultra96V2, mostly due to resource constraints, this created a large data set to be explored.

The very first analysis conducted is to look at how strongly the parameter priority influences the throughput and accuracy. Table 5.2 shows how much the accuracy changes in absolute percentage points and how much the throughput in hardware changes in percent. The change in accuracy is calculated as:

$$\Delta_{i,\text{Accuracy}} = X_{i,\text{PE}} - X_{i,\text{Balanced}}, \tag{5.3}$$

with $\Delta_{i,\text{Accuracy}}$ the change in accuracy, $X_{i,\text{PE}}$ the accuracy in percent for a given network setup (activation bits, weight bits, pruning percentage) with PE as the priority parameter and $X_{i,\text{Balanced}}$ the accuracy in percent for the same network setup but with the balanced parameter priority.

Since the throughput is given in images per second and not percent, this change is calculated differently:

$$\Delta_{i,\text{Throughput}} = Y_{i,\text{PE}}/Y_{i,\text{Balanced}} - 1, \tag{5.4}$$

with $\Delta_{i,\text{Throughput}}$ the change in throughput, $Y_{i,\text{PE}}$ the throughput in images per second for a given network setup (activation bits, weight bits, pruning percentage) with PE as the priority parameter and $Y_{i,\text{Balanced}}$ the throughput in images per second for the same network setup but with the balanced parameter priority.

Sparsity	25%	50%	75%	87.5%	All
Accuracy gain [%]	0.10 ± 0.27	0.43 ± 1.04	1.8 ± 2.4	4.2 ± 3.9	1.96 ± 3.06
Throughput gain [%]	-76 ± 23	-78 ± 22	-85 ± 16	-53 ± 30	-72 ± 27

Table 5.2: Parameter priority comparison

Table 5.2 displays the averages and standard deviations for all $\Delta_{i,\text{Throughput}}$ and $\Delta_{i,\text{Accuracy}}$, as a function of the sparsity percentages. The goal here is to evaluate how the accuracy and throughput develop from one pruning percentage to the next one. While the values for throughput and accuracy are next to each other in Table 5.2 it makes little sense to compare them directly to each other.

In general, the data collected is broadly spread as evident by the large standard deviations for all values. This is due to fact that the whole data set comprises many different combinations of weight and activation bits, which themselves strongly influence both throughput and accuracy. Nonetheless, it is possible to identify trends and to draw conclusions from the aggregated results. In particular the expectation that prioritizing the PE performance parameter improves network prediction accuracy is visible as a tendency for most settings. Though due to the large spread, this is not particularly significant.

The impact on throughput however is very evident for all settings. Here losses between 80% and 50% seem to be possible in all cases. While the spread is usually still large, the distance to the zero hypothesis in sigmas is in general larger than two, often larger than even three, confirming the impression that the impact on throughput can be very detrimental.

From this one can conclude that the expected positive impact on accuracy is hugely out-weighted by the drop in throughput. Consequently, all following results for the coarse-grain method are analyzed for the balanced parameter priority for Algorithm 1.

Additionally, these experiments are accompanied by an exhaustive search over the same parameter region for non-pruned networks. Some of these results were already shown in Chapter 4.4 in Figure 4.9. To compare the pruned and non-pruned results, the average differences in accuracy and throughput are calculated, similar to equation 5.4 and 5.3:

$$\Delta_{i,\text{Accuracy}} = X_{i,\text{pruned}} - X_i, \tag{5.5}$$

with $\Delta_{i,\text{Accuracy}}$ the change in accuracy, $X_{i,\text{pruned}}$ the accuracy in percent for a given network setup (activation bits, weight bits, sparsity percentage) with coarse-grain pruning and X_i the accuracy in percent for the same network setup, but without any sparsity.

And for the throughput:

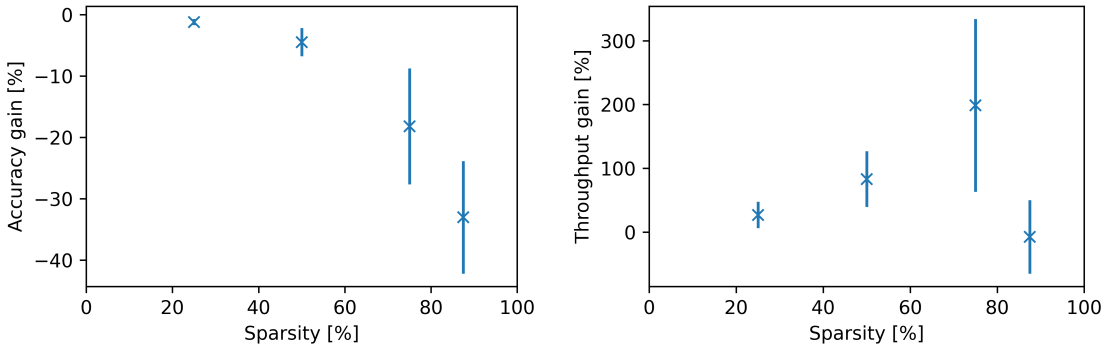
$$\Delta_{i,\text{Throughput}} = Y_{i,\text{pruned}}/Y_i - 1, \tag{5.6}$$

with $\Delta_{i,\text{Throughput}}$ the change in throughput, $Y_{i,\text{pruned}}$ the throughput in images per second for a given network setup (activation bits, weight bits, sparsity percentage) with coarse-grain pruning and Y_i the throughput in images per second for the same network setup, but without any sparsity.

Table 5.3 displays the averages and standard deviations for all $\Delta_{i,\text{Throughput}}$ and $\Delta_{i,\text{Accuracy}}$, as a function of the sparsity percentage, similar to Table 5.2 before. Additionally, the data from Table 5.3 are visualized in Figure 5.8a for the accuracy gain and in Figure 5.8b for the throughput gain.

Sparsity	25%	50%	75%	87.5%	All
Accuracy gain [%]	-1.2 ± 0.5	-4.5 ± 2.3	-18.2 ± 9.4	-33.0 ± 9.2	$-14. \pm 14.$
Throughput gain [%]	27 ± 21	83 ± 44	199 ± 136	-8 ± 58	75 ± 110

Table 5.3: Comparison between experiment results for coarse-grain pruning and non-pruned training and hardware experiments.



(a) Accuracy gain

(b) Throughput gain

Figure 5.8: Visualized data from Table 5.3, x-axis: Sparsity in percent, y-axis: Accuracy or Throughput gain in percent.

As expected the accuracy in Figure 5.8a decreases with increased sparsity. While the change is relatively manageable for 25% and 50%, the accuracy drastically worsens above 50% sparsity. For these large amounts of sparsity, changes in the network architecture are likely required to maintain accuracy at an acceptable level, see also Chapter 5.4.5. A general outlook which kind of changes might be useful will be given in Chapter 5.7.

The throughput also behaves similar as expected. It increases approximately linearly from 25% to 75% sparsity. However, at 87.5% sparsity the throughput suddenly decreases. Similar results are found for the fine-grain implementation in

Chapter 5.4.5. The cause of this degradation in performance for both implementations is not clear. One potential reason could be reaching an edge-performance case with FINN, in which the data-flow implementation reacts strangely, to low bandwidth data transmissions between layers. Another reason could be a potential need for optimization of the modified image2col implementation which could reach some internal limitation in the present version. In either case more profiling would be needed to clarify the cause of this unexpected behavior.

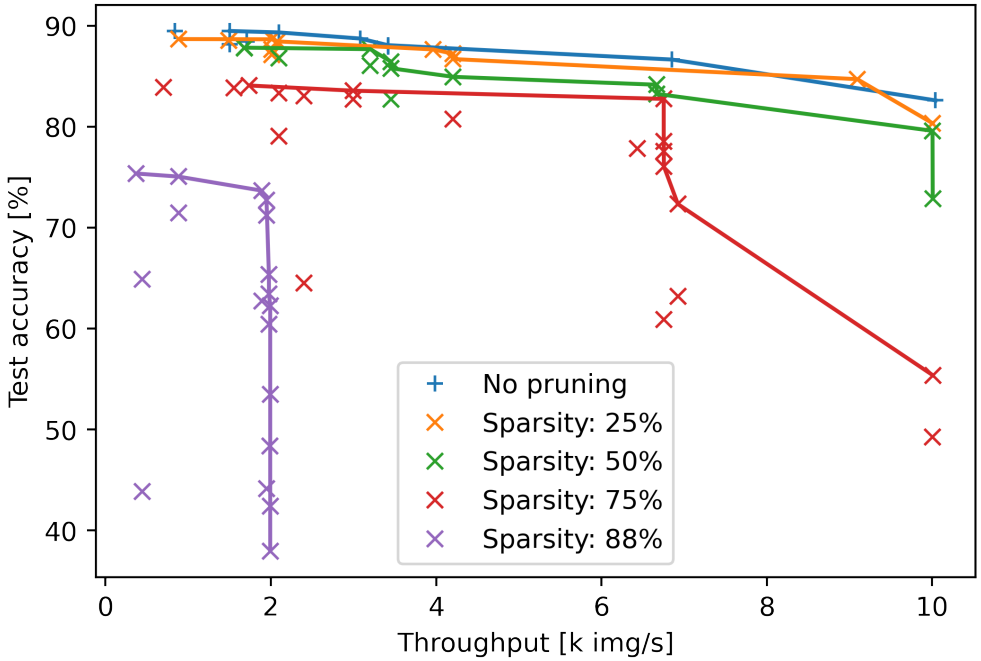


Figure 5.9: Pareto frontier discovery on the Ultra96V2 FPGA for coarse-grain pruning. Additionally, results from non-pruned networks are overlaid for comparison. For each sparsity setting the Pareto frontier is drawn in the same color as a solid line. Increasing sparsity leads to an increasing limitation in throughput. X-axis: Throughput in thousands of images per second, y-axis: Inference accuracy on the test dataset at the end of the training for a given network.

Along with investigating these general trends, another investigation into the Pareto frontier discovery is conducted. With the Pareto frontier it is possible to map the possible trade-offs at maximum value for two otherwise unrelated metrics, in this case the network accuracy and throughput. The result is shown in Figure 5.9. Here the data from all experiments from the coarse-grain pruning and the non-pruned experiments are overlaid in a combined plot, which shows the throughput and accuracy for each experiment. The Pareto frontier is then drawn as a solid line in the corresponding color. The frontier maps out how trad-offs can be made between the throughput and accuracy. Additionally, everything above this line is not achievable by the method it represents, meaning that the frontier also maps out the maximum

capabilities of a given method.

Thus, if the Pareto frontier of one of the pruning methods is cleanly separated from the non-pruning results in a positive direction in both axes, then this indicates that the pruning method outperforms the non-pruning results. Of course the opposite can also be true, if the frontier of the pruning method is cleanly separated at the lower end of throughput and accuracy, then the method can not deliver better overall performance. The pruning needs to be handled with care if special requirements in accuracy or throughput have to be fulfilled.

Looking at the actual frontiers towards the top of the plot it is visible that both 25% sparsity and 50% sparsity look competitive with the non-pruned results. In these experiments the errors cannot straightforwardly be specified, however, there is no point at which the coarse-grain pruning shows clearly better results and cleanly separates from the frontier created by the non-pruning data.

In contrast and as expected from previous results both the 75% and 87.5% sparsity settings perform significantly worse in terms of accuracy and the range of results is strongly spread out. The reason for the very low accuracy here is likely that for low bit-widths and large amounts of sparsity, not enough data can be saved in the model to meaningfully learn information from the CIFAR10 dataset. However, in terms of the throughput something unexpected appears to happen: increasing sparsity leads to an increasing limitation in throughput. Looking at the frontiers for 75% and 87.5% sparsity, they both appear to be limited in some way in terms of throughput. This is particularly visible for 87.5% sparsity, which appears to not be capable to improve beyond two thousand images per second, mostly independent of the set configuration. It is likely that the issue, which is occurring here is the same, as the one visible in Figure 5.8b. And it seems that some fundamental limit of either the implementation or FINN itself is reached. Unfortunately, there was not enough time to properly study this issue, so that it is currently unclear what the exact cause is.

5.4 Fine-grained pruning

While implementing the coarse-grain pruning method and investigating the SIMD parameter, another idea on how to implement structured pruning in FINN came up: Employ the SIMD parameter in a such way to enable smaller sparsity structures than those created by the coarse-grain pruning. With this idea in mind the second pruning method is implemented. It explicitly aims for higher prediction accuracy compared to the first method.

5.4.1 Implementation

Seen from a high-level, the way in which the fine-grained pruning method is implemented is similar to the previous implementation. Here as well, the image2col implementation is modified and pruning takes place when data is output to the next layer. Again, a new HLS function is implemented, called *ConvolutionInputGeneratorSIMDPruned*. Accompanying the image2col implementation a testbench is designed to verify correctness. Both, the implementation[8] and testbench[9] are published in Hendrik Borras's public fork[10] from the official repository[11], see also the links in Chapter 9.

The implementation is shown schematically in Figure 5.10. In contrast to the

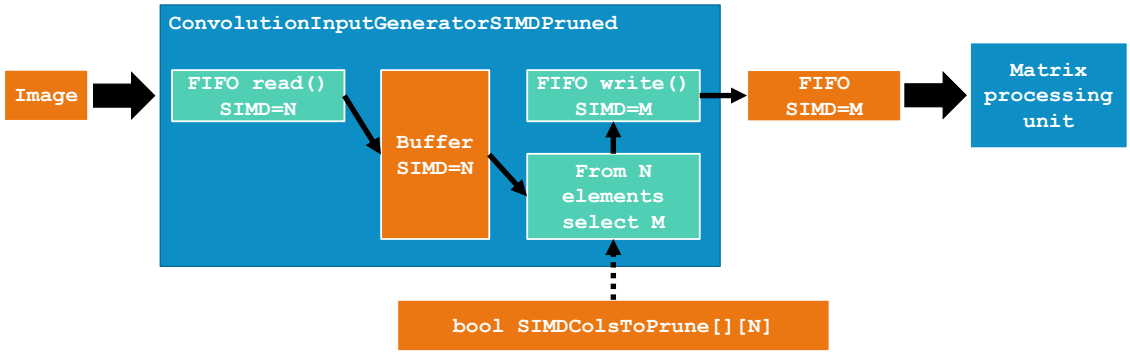


Figure 5.10: Schematic of the image2col operator for fine-grain pruning, as implemented in HLS.

previous implementation, this schematic also highlights the value of the SIMD parameter in each step. In the coarse-grain implementation these values are the same in all steps and thus not highlighted. Similar to the first implementation, an input

[8] https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/slidingwindow.h#L175

[9] https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/test_swg_SIMD_pruned.tcl and https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/swg_SIMD_pruned_tb.cpp

[10] https://github.com/HenniOVP/finn-hlslib/tree/feature/col_pruning

[11] https://github.com/Xilinx/finn-hlslib

image is first read into the buffer, both the input stream and the buffer having a SIMD value of N and thus each stored element has a width of N data. From these N elements M are selected and passed to the output. Which of these elements are selected, is defined by the *SIMDColsToPrune* boolean array. The FIFO between the im2col and the matrix processing unit then has a SIMD width of M.

This step of going from a SIMD width of N to M effectively allows to prune single columns within one block of columns. In contrast, the coarse implementation can only prune blocks of columns, not single columns. In order to give the fine-grained implementation the highest flexibility in choosing which columns to prune a large SIMD parameter is preferred. Therefore, a corresponding priority setting is introduced for the automatic performance tuning algorithm in Chapter 4.3.

However, both N and M can not be chosen freely and are bound to an extended sub-set of the constraints presented in Chapter 4.2.1. These are in particular:

1. $M < N$
2. $N > \mathrm{CH_{IN}}/1024$
3. $N \leq \mathrm{CH_{IN}}$
4. $\mathrm{CH_{IN}} \bmod N = 0$
5. $\mathrm{CH_{IN}} \bmod M = 0,$

where N and M represent the integer values of the SIMD parameters shown in Figure 5.10 and $\mathrm{CH_{IN}}$ represents the number of input channels for a given layer.

This in particular means that the allowed percentages of sparsity for this method are defined by the following equation:

$$S_i = \left(1 - \frac{1}{2^i}\right) \cdot 100, \tag{5.7}$$

with i a natural number, indicating the degree of sparsity and S_i the percentage of sparsity in percent for a given degree i. The minimum N for a given sparsity percentage S_i is further constrained by:

$$N \geq 2^i \tag{5.8}$$

These constraints severely limit the sparsity settings, which can be chosen for a given network. Thus, in this work only 50%, 75% and 87.5% sparsity are explored for the fine-grained method.

A different but non-obvious limitation, which can be derived from the list of constraints concerns the maximum amount of parallelism in a convolutional layer. The maximum amount of parallelism for previous methods can be defined as:

$$P_{\mathrm{max,\ coarse}} = N_{\mathrm{max}} \cdot \mathrm{PE_{max}}, \tag{5.9}$$

with $P_{\text{max, coarse}}$ the maximum amount of parallelism for the coarse-grain implementation and also the non-pruned implementation, N_{max} the maximum of N with the constraints given above and PE_{max} the maximum for the PE parameter according to the constraints given in Section 4.2.1. For the fine-grained pruning method P_{max} now shrinks to:

$$P_{\text{max, fine}} = M_{\text{max}} \cdot \text{PE}_{\text{max}}, \tag{5.10}$$

with $P_{\text{max, fine}}$ the maximum amount of parallelism for the fine-grain implementation and M_{max} the maximum of M with the constraints given above. Because $M < N$, the following is also enforced: $P_{\text{max, fine}} < P_{\text{max, coarse}}$.

While this constraint only sets an upper limit to the amount of parallelism in a convolutional layer, it is still important to keep in mind when setting performance goals.

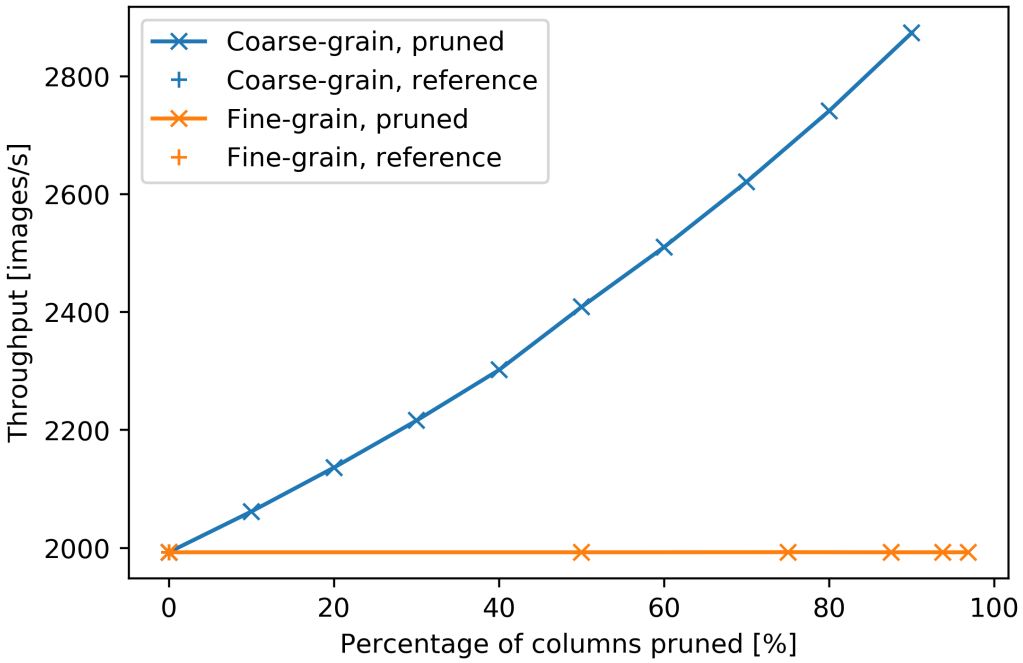


Figure 5.11: Throughput as a function pruning percentage for the binarized convolutional network with default performance parameters. The hardware throughput is shown once for the coarse-grain pruning (blue line) and another time for the fine-grain pruning (orange line). Additionally, the reference measurements with the original implementation of the *ConvolutionInputGenerator* are shown (blue and orange "+"), x-axis: Percentage of columns pruned, y-axis: Throughput in hardware in images per second.

5.4.2 Impact on throughput

Similar to the approach in the previous sections, the throughput is the first metric explored. In particular the throughput for the binarized convolutional network is studied for varying sparsity settings and for using only the default SIMD and PE parameters, as presented in Table 4.2.

The result from this experiment is shown in Figure 5.11, which now also contains the results from Figure 5.2. Here it becomes evident, that the method does not influence the throughput of a given network at all when the SIMD and PE parameters are kept constant. This is a very much expected behavior, because the method also reduces the SIMD parameter for the *StreamingFCLayer_Batch*, which follows the *ConvolutionInputGeneratorSIMDPruned*. This is the result of reducing the SIMD parameter from N to M within the *ConvolutionInputGeneratorSIMDPruned*. As a consequence this reduces the amount of parallelism contained in the *StreamingFCLayer_Batch* by the ratio M/N. At the same time the amount of data that needs to be processed is reduced by exactly the same ratio due to the introduced sparsity. Thus, the reduction in data to process and the reduction in parallelism cancel each

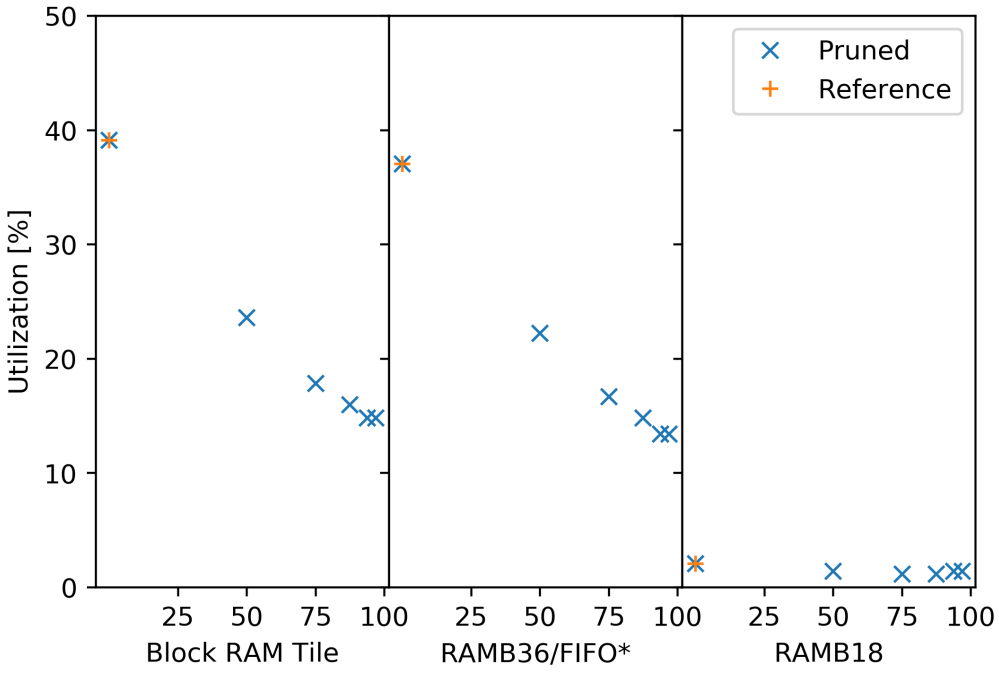


Figure 5.12: BRAM utilization for different metrics as a function of varying amounts of sparsity for the fine-grained pruning method on the binarized convolutional network, x-axis: Sparsity for three different metrics, y-axis: BRAM utilization in percent.

*Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E2 or one FIFO18E2. However, if a FIFO18E2 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E2.

other resulting in no throughput change.

Additionally, Figure 5.11 highlights how the possible settings in terms of sparsity are much more constrained for the fine-grained method. For the the fine-grained method the crosses, indicating measurement points, follow the Equation 5.7. Thus, the measurements are not as frequent in sparsity as for the coarse-grained method.

5.4.3 Impact on resources

In order to investigate the resource impact of this pruning method the results shown in Figure 5.11 are also investigated for their resource utilization on the Ultra96V2. Figure 5.12 shows the utilization for the BRAM and Figure 5.13 for the logic resources.

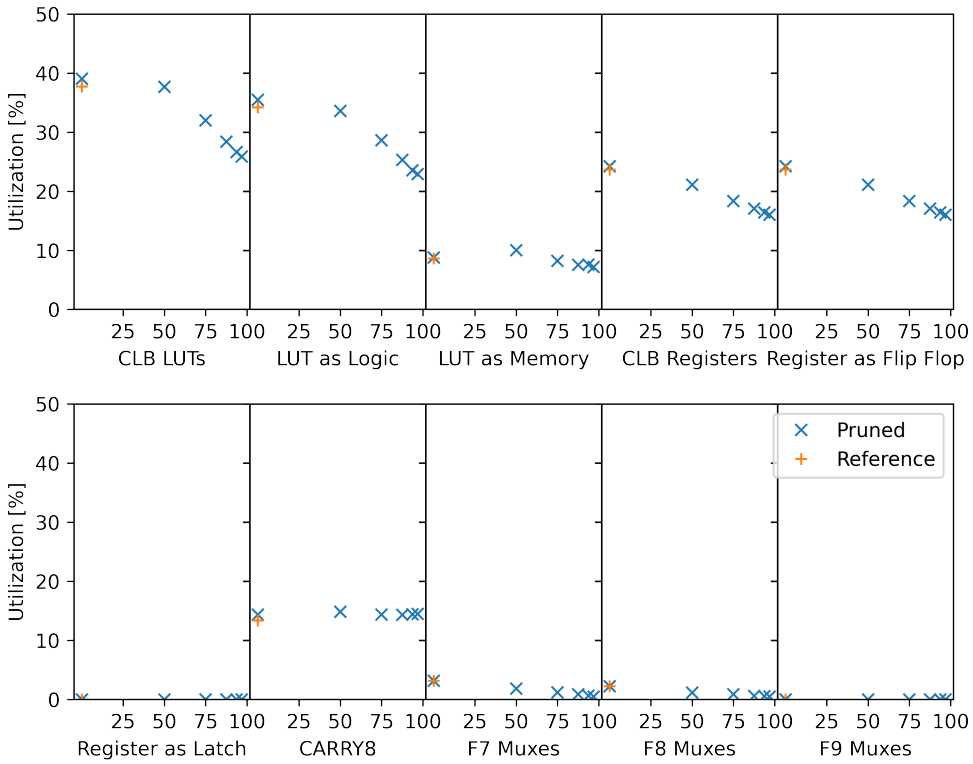


Figure 5.13: Logic utilization as a function of varying amounts of sparsity, x-axis: Sparsity for ten different metrics, y-axis: Logic utilization in percent.

In similarity to Figure 5.3 the amount of memory used in Figure 5.12 is reduced in a linear fashion. This is in-line with the expectation that the required memory for weights should be negative proportional to the sparsity in percent.

However, in contrast to Figure 5.4, the equivalent Figure for the fine-grained pruning (Fig. 5.13) shows notably different results. While there are still some metrics which show no change, there are now several which are revealed to be negative proportional to the sparsity setting. The reason for this reduction can be traced back to the implementation of the fine-grained pruning as described in Section 5.4.1. Because the fine-grained method reduces the parallelism contained in convolutional layers by M/N, the resource utilization in these layers should also decrease linearly as demonstrated in Chapter 4.2.3. Thus, the decrease in logic resources is fully expected and explained by previous experiments.

5.4.4 Training methodology and accuracy

In contrast to the coarse-grained method fewer intermediate steps are taken when implementing the training for the fine-grained method. Consequently, the training is directly implemented with the iterative ℓ^1-norm pruning approach explained in Chapter 5.3.4.

An example of the resulting epoch plot for the iterative ℓ^1-norm pruning approach is shown in Figure 5.14. In fact, this figure shows the same network as highlighted

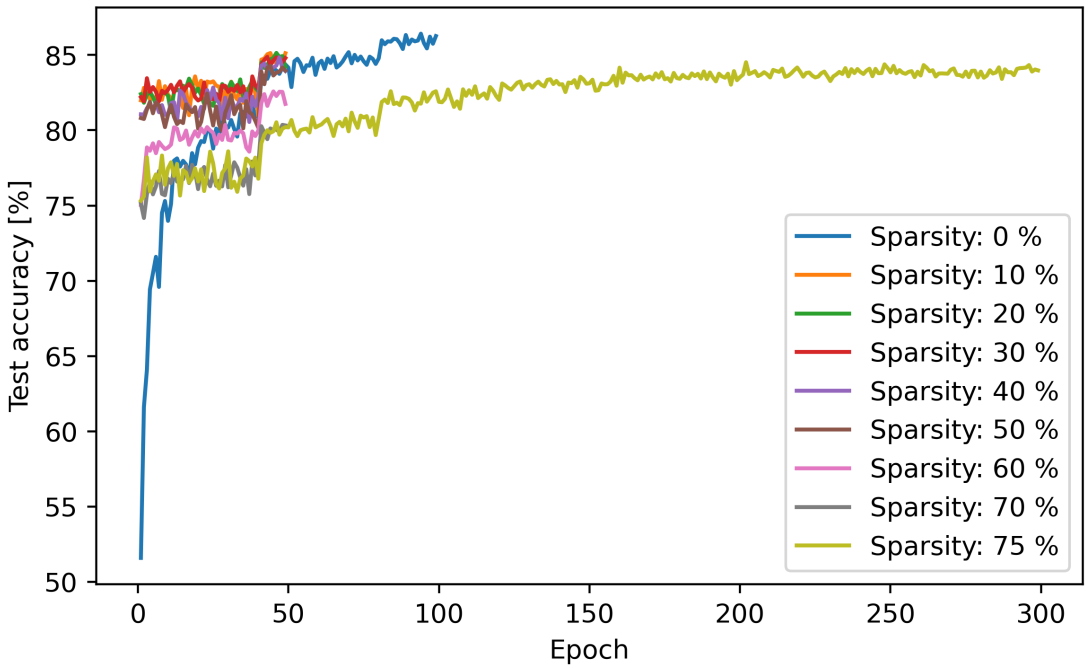


Figure 5.14: Training procedure for the iterative ℓ^1-norm pruning method for fine-grain pruning. Each sparsity setting during training is shown as an individual curve. Here a network with three-bit weights and two-bit activations is trained. It is of note, that the constraint for the possible amounts of sparsity from Equation 5.7 is not present during training. X-axis: Epoch number, y-axis: Test accuracy in percent.

in Figure 5.7. While the general shape of all curves is similar in both figures, the final test accuracy is notably higher in Figure 5.14, which highlights, that the fine-grain pruning can achieve a notably better test accuracy than the coarse-grained approach.

A less obvious feature of the implemented training method is that it can selectively enable or disable adherence to the constraints imposed by FINN as described in Chapter 5.4.1. This is particularly important, since the network needs to be trained in sparsity increments of 10%, which would generally not be possible with the constraints imposed on the parameter M, see Equation 5.7. However, because *Brevitas* and *PyTorch* do not have any requirement for these constraints most of them can be safely ignored during the intermediate training steps.

5.4.5 Exploring network design space for the Ultra96V2

Similar to the experiments conducted for the coarse-grain pruning, an exhaustive search is run for the fine-grain pruning as well. This search covered the following parameters:

- Weight bit-widths: 1 to 5 bits
- Activation bit-widths: 1 to 5 bits
- Sparsity percentages: 50%, 75% and 87.5%
- Performance parameter priority: Balanced and SIMD

Excluding again combinations of 1-bit activations and multi-bit weights, which did not run, this search tested 128 network settings, providing a large data set to be explored.

The first analysis conducted with this data is to answer the question if changing the parameter priority from Balanced to SIMD would yield better results in throughput and accuracy. Calculation wise equations 5.3 and 5.4 are adapted for this purpose. The results are shown in Table 5.4. As expected the accuracy tends

Sparsity	50%	75%	87.5%	All
Accuracy gain [%]	0.0 ± 0.2	0.5 ± 1.6	0.6 ± 1.2	0.46 ± 1.29
Throughput gain [%]	-45 ± 25	-5 ± 24	-12 ± 27	-14 ± 29

Table 5.4: Parameter priority comparison

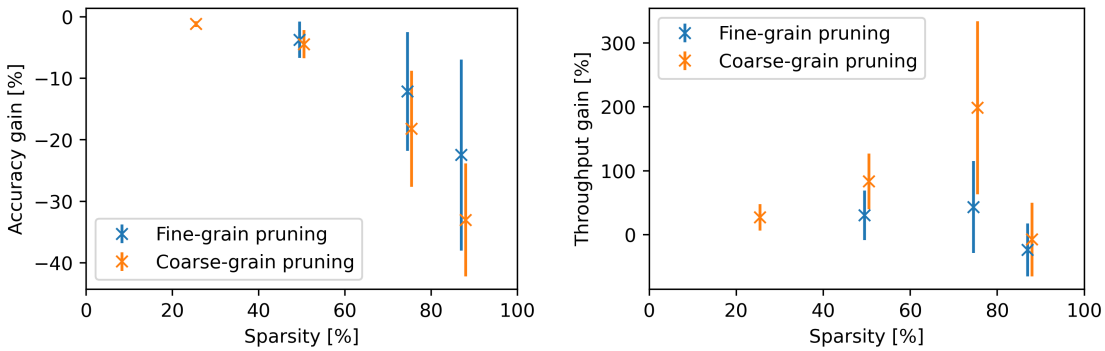
to increase, when prioritizing the SIMD parameter. However, this increase is very small and barely noticeable in the spread. Similar to the coarse-grain pruning, the throughput also decreases when switching to the specialized parameter priority. Although the decrease is not as large, the conclusion is the same. Overall it seems that

the trade-off between the accuracy gained and decreased throughput is not worthwhile. Therefore, all following experiments are run with the balanced parameter priority.

Next it is investigated how the throughput and accuracy is influenced by the fine-grained pruning itself, when compared to non-pruned results. Here the same results for the non-pruned networks are used as in Section 5.3.5. Consequently, also the same metrics for calculating the change in accuracy (Equation 5.5) and throughput (Equation 5.6) are adapted. The results are presented numerically in Table 5.5 and graphically in Figures 5.15a and 5.15b. Additionally, Figures 5.15a and 5.15b

Sparsity	50%	75%	87.5%	All
Accuracy gain [%]	-3.7 ± 3.0	-12.2 ± 9.7	-23 ± 16	-13 ± 13
Throughput gain [%]	30 ± 39	43 ± 72	-24 ± 42	17 ± 60

Table 5.5: Comparison between experiment results for fine-grain pruning and non-pruned training and hardware experiments.



(a) Accuracy gain

(b) Throughput gain

Figure 5.15: Visualized data from Tables 5.3 and 5.5. For better visual clarity the two data sets are slightly offset from each other in x-direction. In reality the sparsity setting is the same for both methods and the points would normally overlap, x-axis: Sparsity in percent, y-axis: Accuracy or Throughput gain in percent.

contain data from the experiments performed with coarse-grain pruning in order to allow a better comparison between the two methods.

In general, the results are as expected. Figure 5.15b shows that the throughput increases with sparsity, although the increase is significantly lower than for the coarse-grain pruning it is still notable. A significant exception is again the throughput result at 87.5% sparsity. This is consistent with the result for coarse-grain pruning. At the moment it is not entirely clear why this decrease is observed. Both SIMD and PE parameter values indicate that the throughput should increase and not decrease for these networks. However, this is not the case and more in-depth

profiling will be required to clarify what is happening and how the issue can be prevented.

The test accuracy on the other hand behaves exactly as expected. It decreases with increased sparsity and at the same time performs significantly better than the coarse-grain pruning method. As the fine-grained pruning method is explicitly designed to improve accuracy these measurements confirm that the design indeed works as intended. The improvement is most notable for the sparsity settings above 50%.

Finally, the Pareto frontier of the fine-grained pruning approach is investigated in order to determine the maximum possible values for the trade-off between accuracy and throughput. The results are shown in Figure 5.16. Similar to the coarse-

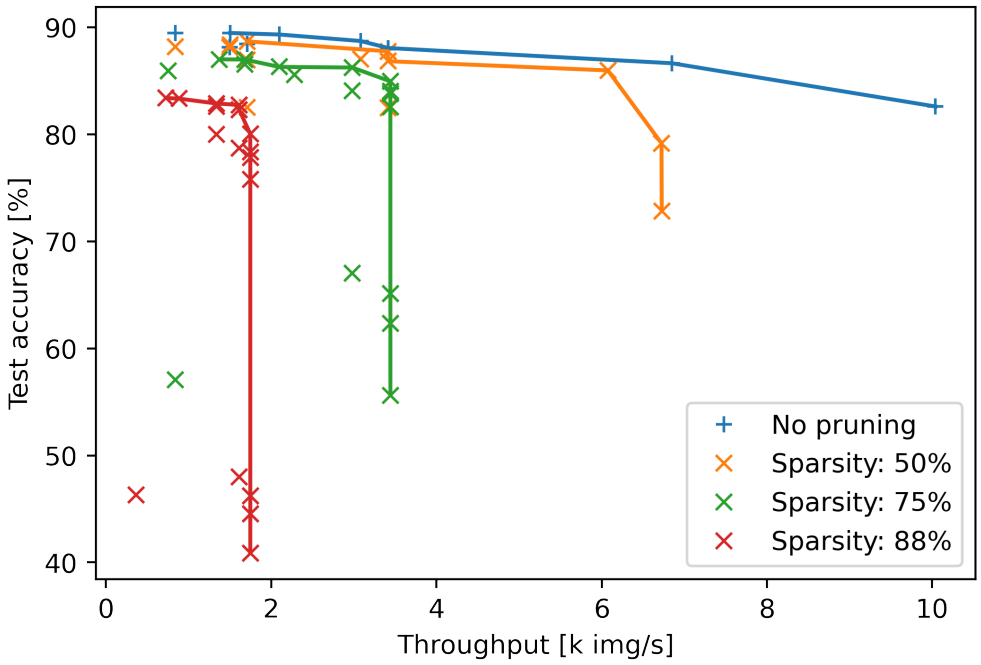


Figure 5.16: Pareto frontier discovery on the Ultra96V2 FPGA for fine-grain pruning. Additionally, results from non-pruned networks are overlaid for comparison. For each sparsity setting the Pareto frontier is drawn in the same color as a solid line. A similar limitation in throughput as for the coarse-grained pruning can be observed. X-axis: Throughput in thousands of images per second, y-axis: Inference accuracy on the test dataset at the end of the training for a given network.

grain pruning, the new method is competitive to networks without pruning for 50% sparsity. However, at no point is the fine-grained pruning clearly better. And as expected, the 75% sparsity setting for the fine-grained pruning is much closer to the non-pruning data in terms of accuracy, when compared to the same sparsity with coarse-grain pruning in Figure 5.9.

However, similar to the coarse-grain pruning networks with 75% and 87.6% sparsity appear to to be limited in their throughput performance. For the fine-grain pruned networks with 75% sparsity, the constraint seem to be at about 3.4 thousand images per second and for networks with 87.6% sparsity the constraint lies at about 1.8 thousand images per second. The cause for this limit is likely the same in both cases as well as for the coarse grain pruning, which is unfortunately due to time constraints left undetermined.

5.5 Implementation comparison

While Figure 5.15a and 5.15b already compare the two pruning implementations to a certain degree, a direct comparison is still to be conducted.

Similar to before, the average changes in accuracy and throughput are calculated. With the change in accuracy calculated as:

$$\Delta_{i,\text{Accuracy}} = X_{i,\text{coarse}} - X_{i,\text{fine}}, \tag{5.11}$$

with $\Delta_{i,\text{Accuracy}}$ the change in accuracy, $X_{i,\text{coarse}}$ the accuracy in percent for a given network setup (activation bits, weight bits, sparsity) with coarse-grain pruning and $X_{i,\text{fine}}$ the accuracy in percent for the same network setup but with fine-grain pruning.

The change in throughput is then calculated with

$$\Delta_{i,\text{Throughput}} = Y_{i,\text{coarse}}/Y_{i,\text{fine}} - 1, \tag{5.12}$$

with $\Delta_{i,\text{Throughput}}$ the change in throughput, $Y_{i,\text{coarse}}$ the throughput in images per second for a given network setup (activation bits, weight bits, sparsity) with coarse-grain pruning and $Y_{i,\text{fine}}$ the throughput in images per second for the same network setup but with fine-grain pruning.

Both $\Delta_{i,\text{Throughput}}$ and $\Delta_{i,\text{Accuracy}}$ now become positive when the coarse-grain pruning performs better than the fine-grained one. Table 5.6 displays the averages and standard deviations for all $\Delta_{i,\text{Throughput}}$ and $\Delta_{i,\text{Accuracy}}$, broken down the by sparsity settings.

Sparsity	50%	75%	87.5%	All
Accuracy gain [%]	-0.7 ± 0.8	-4.7 ± 3.6	-10.1 ± 6.8	-5.5 ± 6.0
Throughput gain [%]	51 ± 37	112 ± 48	16 ± 16	61 ± 55

Table 5.6: Comparison of coarse- and fine-grained pruning.

As expected the fine-grained method clearly outperforms the coarse-grained method in terms of the accuracy. This becomes most evident for sparsities of 75% and above. A bit less expected is that the coarse method outperforms the fine method in terms of throughput. This is especially significant, when neglecting the 87.5% sparsity

point, for which the throughput is likely suffering from an a yet to be identified issue for both methods.

In conclusion, this gives a clear trade-off to choose from. In cases where higher accuracy is preferred over raw performance, the fine-grained method can be chosen. For other cases, where throughput is paramount, the coarse-grained method is clearly favored.

5.6 Possible points of improvement

The most obvious point of improvement for both methods is likely an investigation of the throughput limitation at high sparsity. As highlighted in Figure 5.15b both implementations seem to suffer from significant throughput degradation at 87.5% sparsity. Similar behavior is also visible in Figures 5.9 and 5.16. It is likely that an in-depth investigation using multiple RTL simulations would lead to important results. These would possibly be able to reveal where a bottleneck might occur, further pointing towards where to optimize the HLS code of either the modified image2col implementation or FINN itself.

It is somewhat more likely that the issues lie with the modified image2col implementation since this is what was primarily modified during this work. If this turns out to be true, then there are multiple points at which the performance could be improved. For the fine-grain pruning one could try to leverage automatic HLS parallelization for selecting the channels to prune. This is likely to use more logic resources on the device, but it is also very likely to improve performance. For the coarse-grain pruning the optimization is a bit more difficult, because the implementation is already very simple. However, one thing that is likely improvable is the indexing overhead. This indexing overhead is similar for both methods, so improvements could be applied for the fine- and coarse-grain pruning at the same time. The index for the pruning mask (*ColsToPrune* and *SIMDColsToPrune*) is recalculated for each new block of columns. This operation is quite complex. In fact in both cases the calculation results in two additions, three multiplications and two divisions per block of columns. No other index calculation in the original image2col unit is this expensive. Thus, it is likely possible to simplify the calculation and it is likely possible to circumvent most multiplications and divisions required for each block of columns.

If however the limiting factor turns out to be the FINN framework itself, then the first points to investigate are likely the sizes of the FIFOs within the dataflow architecture. As mentioned in Chapter 4.5.2, since release 0.5b it is possible to automatically size these through the use of an RTL simulation. To properly integrate the automatic FIFO sizing it would likely be required to also add the BRAM as an optimization parameter to the automatic parameter tuning algorithm. However, for first tests whether the FIFOs are actually the issue, RTL simulations with larger FIFO sizes would likely be enough to determine their impact on the throughput for pruning settings with high sparsity.

Whatever the final result of this investigation might be, it is sure to lead to a better understanding of how both pruning methods interact with the overall FINN framework.

Alongside these general improvements in performance, better integration into FINN itself is certainly an important topic for future work, because the code is presently mostly in a proof of concept stage. In particular the *ConvolutionInputGeneratorPruned* and *ConvolutionInputGeneratorSIMDPruned* are not yet integrated into FINNs Python verification path and also not covered by the *ONNX* export capabilities of *Brevitas*. Especially the integration with *Brevitas* would allow for end-to-end training and verification of sparse networks. At the moment the test accuracy is extracted during training with *Brevitas*, but with an end-to-end tool flow, this could instead be done directly on the FPGA.

5.7 Outlook

An idea which could not be explored in this work is to adjust the network architecture to take advantage of the iteratively learned sparsity introduced in this chapter. Of particular interest would be to implement long distance correlation kernels in a more resource efficient fashion. This would increase the receptive field of a given kernel, while using less parameters.

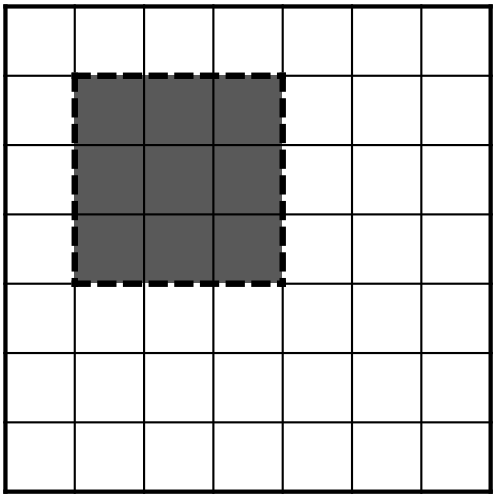


Figure 5.17: Standard 3x3 kernel on an 7x7 image.

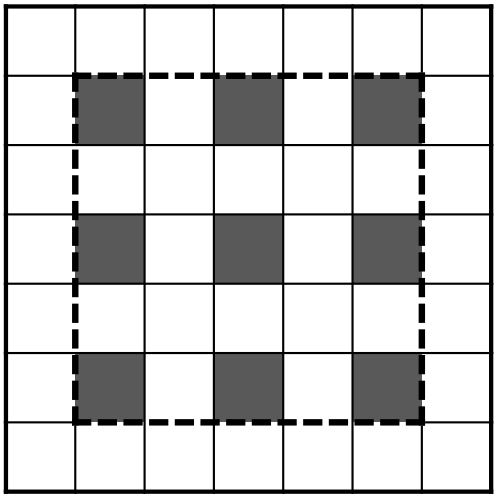


Figure 5.18: Dilated 3x3 kernel on an 7x7 image.

As an example, Figure 5.17 shows a 3x3 kernel schematically on a 7x7 image. Kernels of this size are also used in this work. If one now wants to correlate pixels over a longer range than 3 pixels one would need to stack multiple 3x3 kernels on top of each other. As example a 7x7 receptive field could be achieved by stacking three layers of 3x3 kernels. With each kernel requiring a parameter for each pixel,

the 3x3 kernel needs 9 parameters per layer to function. In total equating to 27 parameters for a 7x7 receptive field.

One approach to improve long-range correlation would work with dilated kernels, as proposed by [YK16]. Here a 3x3 kernel can cover a 9x9 receptive field while only requiring 9 parameters. Schematically this is shown in Figure 5.18. The downside is that now short-range correlations can no longer be employed imposing the requirement of additional kernels to regain the coverage of shorter ranges.

To fill this short-range gap, one could revert to larger kernels in stacked layers. For example, using two stacked 5x5 kernels a 9x9 receptive field can be achieved. While this keeps short-range correlations intact, the number of parameters increases drastically to 25 parameters per layer, totaling to 50 for two layers. Here sparsity can come into play to reduce the number of parameters per layer. For example, in pruning both layers by a total of 50% only 25 parameters remain. The nice

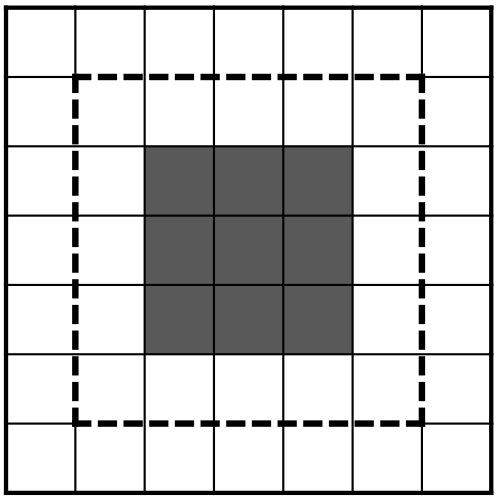


Figure 5.19: Sparse 5x5 kernel with short range correlations on an 7x7 image.

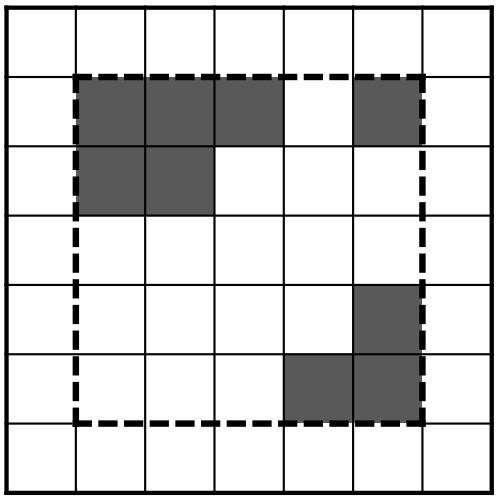


Figure 5.20: Sparse 5x5 kernel with long range correlations on an 7x7 image.

side-effect is then that the pruned structures can also be learned, meaning that a network can effectively select if short-range correlations, long-range correlations or a mixture of both are kept per kernel. A schematic example for keeping short-range correlations in a sparse 5x5 kernel is given in Figure 5.19, which is contrasted by the schematic 5x5 kernel for long-range correlations in Figure 5.20.

In conclusion, comparing two layers of sparse 5x5 kernels to three layers of 3x3 kernels results in about the same number of parameters, 25 and 27 respectively. However, the two sparse 5x5 kernels can now cover a significantly larger receptive field of 9x9 pixels, compared to the three 3x3 kernels, which could only cover 7x7 pixels.

This idea of course depends on how efficient the implemented pruning method is in practice. In particular, the memory usage and operations required per parameter

should be kept constant during pruning. The coarse-grain pruning method presented in this work might approximately fulfill this requirement, but to which extent this is true remains to be seen. Additionally, adjusting the kernel size in a similar manner as shown before requires significant changes to the neural network architecture. Following the example above one would start by increasing the kernel size, while decreasing the number of layers. However, in which fashion this is done best is a non-trivial question.

Investigating such modified network architectures is thus a potential next step to further explore the implemented pruning methods in FINN.

6 Discussion

In this chapter the results accomplished within this work are surveyed and critically examined. In particular it is investigated where each major contribution shows good results or short-comings and how further work could lead to improvements.

6.1 Parallel synthesis

Many experiments within this work are drastically accelerated by the use of parallel transformations in FINN. As such the *NodeLocalTransformation*, developed in Chapter 3 fulfills its goal with strong impact. However, during the further work with FINN and applying the implementation it became clear that there are still cases, where limitations are visible. One particular disadvantage is that the number of workers is statically allocated to a transformation during its complete run-time. Even if the transformation uses less than the number of allocated workers, there is currently no way to make use of these free resources. This is especially problematic when multiple synthesis jobs run in parallel. In this case multiple *NodeLocalTransformation* are executed at the same time, each with their own budget of parallel workers. This can quickly lead to unused resources, because, as shown in Figure 3.1, the number of workers in use can quickly drop and will stay below the maximum for a significant part of the overall run-time. Instead of a static setting, it would be beneficial if there would be a central scheduler for FINN, with which the number of active workers could be centrally managed. With such a feature the scheduler could dynamically allocate workers to *NodeLocalTransformation*s as they become available. This in turn could lead to a better overall resource usage. While *Python's multiprocessing*[1] module has no support for such a centralized scheduler, *Dask*[2] might be a viable alternative, that provides such a feature. However, switching to *Dask* would require significant changes to the *NodeLocalTransformation*, but the basic ideas presented in Chapter 3 would stay the same.

6.2 Automatic tuning of performance parameters

Similar to the *NodeLocalTransformation*, the automatic tuning of SIMD and PE parameters presented in Chapter 4 works overall well and fulfills its design goals. Nonetheless, the presented solution also has a fair share of details, which could be improved or extended. One such aspect is, that in some cases the algorithm takes

[1] https://docs.python.org/3.6/library/multiprocessing.html
[2] https://docs.dask.org/en/latest/

relatively long to produce results. There are potentially two independent solutions to improve the situation. On one side the estimates provided by FINN could be further refined, thus requiring less adjustments to the overall LUT budget. This solution is already being investigated by XILINX[3]. On the other hand, it appears that in some cases the algorithm tries to optimize for a design, which is bound to never fit onto a given FPGA, even with very small SIMD and PE settings. This may happen for example, when the design will never fit into the BRAM. Here a recognition for such "doomed" configurations would be very useful and would significantly speed up potential further exhaustive searches.

Additionally, it would be interesting if the optimization could take more parameters into account. One of these is the FPGA clock frequency, as explained in Section 4.5.1. Another one would be to also optimize for the BRAM and UltraRAM (URAM) utilization. However, the considerations, which need to go into the utilization of the two memory systems are rather complicated, especially since both systems can be used by multiple elements within a network synthesized by FINN. These are: the layer weights, the FIFO queue sizes and the buffers required in image2col units. The amount of weight memory required for a given model is fixed in size and location and thus less of a consideration. More important are the FIFOs and image2col buffers. While the buffers are also fixed in size, their location is not. In-fact for each image2col unit in a given network one can configure this buffer to be in LUT memory, BRAM or URAM. The configuration of these buffers thus influences all three resource utilizations. The FIFO sizes are somewhat on the opposite spectrum: their location is fixed in the BRAM, but their size is not. And as shown in Section 4.5.2 their size significantly influences the inference performance.

As such, combining the optimization of FIFO sizes and image2col buffer locations makes the resource optimization problem highly non-trivial. Nonetheless, automatically optimizing these parameters as well would enable FINN designs to run faster per default, while making the whole FINN framework easier to use. Thus, the focus of future work in this area should likely be to also take the BRAM and URAM into consideration during optimization.

6.3 Sparsity

The work on sparsity successfully explores how structured sparsity can be implemented in FINN. Two different approaches for pruning were developed and explored for their impact on throughput, resource usage and inference accuracy for the convolutional example network of FINN. Both methods produce comparable results at the Pareto frontier, when compared to equivalent experiments without pruning.

However, in order to improve in performance up to the level of the non-pruned results further work is needed. In particular changes to the network architecture are required. As explained in Section 5.7 employing larger kernels, while reducing the number of convolutional layers might lead to higher accuracy results. At the

[3] https://gitter.im/xilinx-finn/community?at=5fc8e5d2657e0c48225b7fa4

same time, it would be interesting to reduce the number of fully-connected layers at the end of the network. There are two main arguments for changing the fully-connected layers. The first being, that none of the fully-connected layers can be pruned by the implemented pruning approaches. Thus, when employing less fully-connected layers the pruning would affect more of the overall required resources of the network. Secondly, modern network architectures rarely employ more than one fully-connected layer at the end of a network. For example ResNet [He+15b] utilizes a final global average pooling layer, followed by a fully-connected layer for classification. Taking a similar approach in future experiments would thus be of great significance for the prediction accuracy and overall resource utilization.

Finally, further work is needed to move both pruning approaches away from the proof of concept stage. In addition, it would be beneficial to have both methods run in an end-to-end mode.

7 Conclusion

Over the course of this work major contributions to the FINN framework have been made and novel experiments are conducted. In particular this work enables the introduction of parallel transformations into FINN. The contribution significantly speeds up the build time for large networks, and corresponding code was contributed directly to the official XILINX repository. Since then it has been a part of all major releases of FINN. Even within this work the possibility for parallelizing the synthesis of networks was extremely helpful, and the development time was significantly reduced for all developments within this work.

Alongside the experiments on sparsity two methods are developed, which can automatically tune SIMD and PE parameters to achieve maximum throughput on a given FPGA with a given network. It is further shown that these automatic optimizations work well for varying quantization bit-widths. Finally, it is demonstrated that by using these algorithms it is possible to explore the Pareto frontier between inference accuracy and throughput for a given network and FPGA.

During the final part of this work two approaches are developed and explored to introduce structured pruning in FINN. These are evaluated for their performance in terms of throughput and inference accuracy. It is shown that both pruning methods can achieve significant throughput improvements (Figure 5.15b), which are to be considered as a trade-off to the accuracy loss incurred by the structured sparsity (Figure 5.15a). During the analysis of the Pareto frontier it is revealed that both approaches show comparable results to non-pruned networks, although the experiments are conducted on a convolutional network, which is not yet optimized for the use with sparsity. These results finally justify a positive conclusion towards exploring pruning for FINN further in future work. In particular with changes in the network architecture significant performance improvements above the already good results presented are expected.

8 Acknowledgments

This master thesis would not have been possible without the valuable support of many people. I would like to express my gratitude to Prof. Dr. Ulrich Brüning and Prof. Dr. Holger Fröning for providing me with the opportunity to conduct this master thesis with them. Especially, I would like to thank Prof. Dr. Holger Fröning for providing me with extensive computing resources to conduct the wide range of experiments within my thesis.

The regular meetings with Prof. Dr. Holger Fröning, Günther Schindler and Bernhard Klein always had strong positive impact on me and this work, leading to new developments and discoveries. For all their help and crucial questions, I am very grateful. In particular, I would like to thank Günther Schindler for the in-depth discussions we had and helping with the very tricky questions around Pruning and DNNs. Further, I would like to express my gratitude to Lorenz Braun for maintaining the computing infrastructure at our institute and always helping me when I had questions concerning our computing resources.

I'd like to give a big thank you to Yaman Umuroğlu, for enabling fruitful and open discussions in the community around FINN and always being ready to answer the complicated questions concerning FPGAs and FINN.

I sincerely thank my family and friends for their support during this thesis and their extremely valuable feedback during the editing process transforming the text into a nice flow of words.

9 Code developed in this work

For each contribution in this work significant work was put into developing code to run the presented experiments. This code is published as open-source for the public to inspect and use. In this chapter a short summary will be given of where this code is published.

- **Parallel synthesis, Chapter 3**
  The code for the parallel synthesis has been integrated into the official FINN repository. The pull request by Hendrik Borras, that contains this code, can be found here: https://github.com/Xilinx/finn/pull/78

- **Automatic tuning of performance parameters, Chapter 4**
  This contribution can be found in the repository for this thesis, created by Hendrik Borras. The script implementing the algorithms of Chapter 4 also interacts with parts of the Chapter on Pruning. Thus the script is designed to be run within the Docker container created by FINN. More information on this can be found in the read-me of the repository. The script itself can be found here: https://github.com/HenniOVP/MA_ZITI/tree/main/simd-pe-tuning

- **Pruning in FINN, Chapter 5**
  In this chapter multiple repositories of FINN were forked and different parts modified. The script developed for the training of pruned networks with the iterative ℓ^1-norm pruning method can be found here: https://github.com/HenniOVP/MA_ZITI/tree/main/training

  The modified HLS code can, which implements the fine- and coarse-grain pruning can be found in the Hendrik Borras's fork of the finn-hlslib repository:

  - Coarse-grain pruning: https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/slidingwindow.h#L331
  - Testbench for the coarse-grain pruning: https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/test_swg_pruned.tcl and https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/swg_pruned_tb.cpp
  - Fine-grain pruning: https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/slidingwindow.h#L175
  - Testbench for the fine-grain pruning: https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/test_swg_SIMD_prune

d.tcl and https://github.com/HenniOVP/finn-hlslib/blob/feature/col_pruning/tb/swg_SIMD_pruned_tb.cpp

The modified code of the FINN main repository can be found in Hendrik Borras's fork of the official repository: https://github.com/HenniOVP/finn/tree/feature/0.4_cutting_pruning

The repository was forked from the development branch of the offical FINN repository[1]. At the time of forking, the development branch was just about to be merged into the master branch for release *0.5b*. As such the fork by Hendrik Borras includes all features of release *0.4b* and most features of release *0.5b*.

[1] https://github.com/Xilinx/finn/tree/dev

A Bibliography

[Ale+21] Alessandro et al. "Xilinx/brevitas: Release version 0.2.0". In: (Feb. 2021). DOI: 10.5281/zenodo.4507767.

[Blo+18] Michaela Blott et al. "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks". In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (2018), pp. 1–23.

[CPS06] Kumar Chellapilla, Sidd Puri, and Patrice Simard. "High Performance Convolutional Neural Networks for Document Processing". In: *Tenth International Workshop on Frontiers in Handwriting Recognition*. Ed. by Guy Lorette. http://www.suvisoft.com. Université de Rennes 1. La Baule (France): Suvisoft, Oct. 2006. URL: https://hal.inria.fr/inria-00112631.

[Far+18] J. Faraone et al. "Customizing Low-Precision Deep Neural Networks for FPGAs". In: *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. 2018, pp. 97–973. DOI: 10.1109/FPL.2018.00025.

[GYC16] Yiwen Guo, Anbang Yao, and Yurong Chen. *Dynamic Network Surgery for Efficient DNNs*. 2016. arXiv: 1608.04493 [cs.NE].

[Han+15] Song Han et al. *Learning both Weights and Connections for Efficient Neural Networks*. 2015. arXiv: 1506.02626 [cs.NE].

[He+15a] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[He+15b] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[Hin+12] G. Hinton et al. "Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups". In: *IEEE Signal Processing Magazine* 29.6 (2012), pp. 82–97. DOI: 10.1109/MSP.2012.2205597.

[Iqb18] Haris Iqbal. "HarisIqbal88/PlotNeuralNet v1.0.0". In: (Dec. 2018). DOI: 10.5281/zenodo.2526396.

[Kir16] A. P. Kirman. "Pareto as an Economist". In: *The New Palgrave Dictionary of Economics*. London: Palgrave Macmillan UK, 2016, pp. 1–10. ISBN: 978-1-349-95121-5. DOI: 10.1057/978-1-349-95121-5_1368-1. URL: https://doi.org/10.1057/978-1-349-95121-5_1368-1.

[KNH] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "CIFAR-10 (Canadian Institute for Advanced Research)". In: (). URL: http://www.cs.toronto.edu/~kriz/cifar.html.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[Len16] Ian Lenz. "Deep learning for robotics". In: (2016). URL: https://hdl.handle.net/1813/44317.

[LL18] Yuanzhi Li and Yingyu Liang. "Learning Overparameterized Neural Networks via Stochastic Gradient Descent on Structured Data". In: *CoRR* abs/1808.01204 (2018). arXiv: 1808.01204. URL: http://arxiv.org/abs/1808.01204.

[Mao+17] Huizi Mao et al. "Exploring the Regularity of Sparse Structure in Convolutional Neural Networks". In: *CoRR* abs/1705.08922 (2017). arXiv: 1705.08922. URL: http://arxiv.org/abs/1705.08922.

[Pap21] Alessandro Pappalardo. *Xilinx/brevitas*. 2021. DOI: 10.5281/zenodo.3333552. URL: https://doi.org/10.5281/zenodo.3333552.

[Sch+20] Günther Schindler et al. "Parameterized Structured Pruning for Deep Neural Networks". In: *International Conference on Machine Learning, Optimization, and Data Science (LOD)* (2020). URL: https://www.ziti.uni-heidelberg.de/ziti/uploads/ce_group/2020-LOD.pdf.

[SZ15] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV].

[Umu+17] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. ACM, 2017, pp. 65–74.

[Umu19] Yaman Umuroglu. *Rebuilding FINN Rebuilding FINN*. 2019. URL: https://xilinx.github.io/finn//2019/10/02/rebuilding-finn-for-open-source.html (visited on 03/02/2021).

[Umu20] Yaman Umuroglu. *FINN HLS layer schematic*. 2020. URL: https://github.com/Xilinx/finn/blob/af783db8dc2a1d2e95bd569d39464b935520b6d2/notebooks/end2end_example/bnn-pynq/cnv-mp-fc.png (visited on 03/02/2021).

[Xil19] Xilinx. *Zynq-7000 SoC - Product Selection Guide*. 2019. URL: https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf (visited on 03/29/2021).

[Xil21a] Xilinx. *UG573 - UltraScale Architecture Memory Resources User Guide*. 2021. URL: https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf (visited on 03/19/2021).

[Xil21b] Xilinx. *Zynq UltraScale+ MPSoC - Product Tables and Product Selection Guide*. 2021. URL: https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf (visited on 03/29/2021).

[Yin+18] Chris Ying et al. "Image Classification at Supercomputer Scale". In: *CoRR* abs/1811.06992 (2018). arXiv: 1811.06992. URL: http://arxiv.org/abs/1811.06992.

[YK16] Fisher Yu and Vladlen Koltun. *Multi-Scale Context Aggregation by Dilated Convolutions*. 2016. arXiv: 1511.07122 [cs.CV].

[Zha+16] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: *CoRR* abs/1611.03530 (2016). arXiv: 1611.03530. URL: http://arxiv.org/abs/1611.03530.

[Zha+18] Dongqing Zhang et al. *LQ-Nets: Learned Quantization for Highly Accurate and Compact Deep Neural Networks*. 2018. arXiv: 1807.10029 [cs.CV].

Erklärung:

Ich versichere, dass ich diese Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 05.04.2021 .....