CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Energy Efficiency of Convolutional Neural Network Inference on FPGAs and Accelerated GPUs

Master's thesis in Embedded Electronic System Design

SEDAT ÜNALACAK

JOSHYKA JOTHI SINGAARAVADIVELU

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Energy Efficiency of Convolutional Neural Network Inference on FPGAs and Accelerated GPUs

SEDAT ÜNALACAK
JOSHYKA JOTHI SINGAARAVADIVELU

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Energy Efficiency of Convolutional Neural Network Inference on FPGAs and Accelerated GPUs
SEDAT ÜNALACAK
JOSHYKA JOTHI SINGAARAVADIVELU

Supervisor: Pedro Petersen Moura Trancoso, Department of Computer Science and Engineering
Advisor: Jonas Tallhage, Volvo Cars
Examiner: Per Larsson-Edefors, Department of Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2021

Energy Efficiency of Convolutional Neural Network Inference on FPGAs and Accelerated GPUs
SEDAT ÜNALACAK
JOSHYKA JOTHI SINGAARAVADIVELU
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Energy efficiency of convolutional neural networks (CNN) can be improved by using low-precision data types. FPGAs and GPUs are widely used to implement CNN inference due to their parallel processing capabilities. Some GPU-based SoCs include accelerator cores that perform low-precision operations efficiently for certain data types. FPGAs can be configured to carry out arbitrary bit-width operations. This thesis examines and compares the energy efficiency of FPGAs and accelerated GPUs for low-precision CNN inference applications. We implemented convolution, fully connected and pooling building blocks for CNN inference on both platforms, verified functionality, measured and compared performance with each other and the state of the art. Accelerator cores on our GPU-based SoC improved the energy efficiency for some design cases at the expense of increased latency and base power consumption. Depending on the design parameters and the type of the layers, FPGA provided up to 23.11 times better energy efficiency, 28.31 times less power consumption and 6.59 times lower latency than accelerated GPU, and GPU provided up to 1.64 times better operational energy efficiency. FPGA worked with even higher energy efficiency for variety of low bit-width data types that cannot be processed by accelerated GPU. Accelerated GPU delivered reasonable energy efficiency levels and required comparably less design time. We also included detailed analysis of the effects of the design parameters on energy efficiency.

Keywords: Accelerator, CNN, Convolution, Energy Efficiency, FPGA, Fully Connected, GPU, HLS, Pooling, TensorRT.

Acknowledgements

Throughout our master thesis work, we have received continuous support and essential advice from both Chalmers University of Technology and Volvo Cars Corporation. We would like to sincerely thank everyone, especially:

- **Prof. Pedro Petersen Moura Trancoso** (Academic Supervisor)
- **Jonas Tallhage** (Company Advisor)
- **Prof. Per Larsson-Edefors** (Examiner)

Sedat Ünalacak, Joshyka Jothi Singaaravadivelu, Gothenburg, August 2021

Contents

List of Figures

List of Tables

Abbreviations

ANN Artificial Neural Network
AXI Advanced eXtensible Interface

BRAM Block Random Access Memory

CAFFE Convolutional Architecture for Fast Feature Embedding
CNN Convolutional Neural Network
CPU Central Processing Unit
CTC Computation to Communication
CUDA Compute Unified Device Architecture
CUDNN NVIDIA CUDA Deep Neural Network Library

DLA Deep Learning Accelerator
DNN Deep Neural Network
DRAM Dynamic Random-Access Memory
DSP Digital Signal Processing

EDA Electronic Design Automation

FC Fully Connected
FFT Fast Fourier Transform
FIFO First In First Out
FM Feature Map
FPGA Field-Programmable Gate Array
FPS Frames per Second

GOPS Giga Operations per Second
GOPS/KLUT Giga Operations per Second per kilo Lookup Table
GOPS/W Giga Operations per Second per Watt
GPU Graphics Processing Unit

HLS High Level Synthesis

IP Intellectual Property

LUT Lookup Table
LUTRAM Lookup Table Random Access Memory

MNIST Modified National Institute of Standards and Technology database
MRI Magnetic Resonance Imaging

NVDLA NVIDIA Deep Learning Accelerator

OS Operating System

PL Programmable Logic
PS Processing System
PYNQ Python Productivity for Zynq

RAM Random Access Memory
RELU Rectified Linear Unit
RGB Red, Green and Blue
RTL Register Transfer Level

SAIF Switching Activity Interchange Format
SDK Software Development Kit
SOC System on a Chip
SOM System on Module

VHDL The Very High Speed Integrated Circuit Hardware Description Language

Glossaries

Accelerated GPU GPU with accelerator cores to support specific operations

Energy Efficiency Performance per unit power where performance can be measured in different ways including number of operations per time

Feature map Output tensor in a CNN when kernel filter is applied to an input tensor

Kernel filter Tensor in a CNN which is convolved with an input tensor to obtain a feature map which carries specific features of the input

Latency Time that passes between the start and the end of processing of an input instance

Overflow Inability of representing a number due to insufficient number of bits of a data type. This might cause erroneous results and might occur after some mathematical operations which cause bit-growth

Overlay IP cores implemented on Programmable Logic part of PYNQ boards which can be called by software running on Processing System like a library function to perform certain operations with high performance

Tensor Multi-dimensional array

Throughput Number of instances processed per time. It can be frames per second for a convolutional neural network, or operations per second for an image processing system in general.

1

Introduction

Convolutional neural networks (CNN) are widely used in various applications including object detection and recognition with image and video processing, speech recognition and language processing [3]. CNNs can be computationally intensive and power-demanding. Different methods to reduce power demand of CNNs have been suggested in literature, including use of lower-precision data types [4]. Different hardware platforms can take advantage of the lower-precision in order to reduce the power consumption at different levels. In this section we will explain the basics of CNNs, studies that have been conducted on using lower-precision data types in CNNs, our goals and research questions in this thesis and an overview of the thesis.

1.1 Overview of CNNs

Machine learning methods have the capability of learning how to perform a task without being provided an explicit algorithm to follow. Artificial neural networks (ANN) are inspired from the structure of brain where nodes called neurons perform calculations on their inputs by using some weight and bias values and generate outputs. It is possible to obtain more accurate results by adjusting these weights, biases and number of neurons. Adjusting weights can be performed by providing a set of inputs to ANN and backpropagating where ANN learns to process inputs correctly in order to provide desired outputs or feature extractions, which is called training. Apart from the input and output neurons, ANNs can include sequentially connected hidden layers with hidden neurons. ANNs with hidden layers are called deep neural networks (DNN). DNNs are popular and widely used since they allow feature extraction at different stages to provide more accurate outputs [5].

CNNs are a subset of DNNs which can efficiently process image and video inputs with high accuracy. CNNs rely on convolution operations performed on tensors. Convolution operations can successfully extract features from images since they can detect local and temporal correlations. These convolution outputs can be processed with non-linear functions to provide flexibility on the features that can be extracted. Subsampling layers can remove unnecessary neurons from the network which reduces the computational complexity without reducing accuracy of the final results [3].

During design of a CNN, different design parameters such as the number of layers, the number of neurons and activation functions can be experimented with to obtain

accurate results. CNNs should also be trained with a set of inputs. Once training of a CNN is complete, it can be fed with different inputs which were not included in the training data set. Running a CNN with predefined weights and biases to perform a task is called inference. CNNs can perform inference in a static or dynamic manner. Dynamic CNNs can activate or deactivate components of CNNs such as layers to improve power and performance, adapt to a new set of inputs by changing their parameters to prepare themselves to new samples and provide compatibility for advanced deep learning methodologies with their flexibility [6]. However, many well-known CNNs have been implemented in a static manner where parameters of the CNN are fixed after training and not updated during inference. Several versions of ResNet [7] and VGG [8] with different number of layers and parameters are some of the well-known examples of CNNs with static inference for image recognition and classification applications. Static inference is less computationally intensive than training. However, training can be performed in a lab environment for a limited number of times during the design process with virtually infinite energy, high computational power and relaxed timing requirements to complete the task. On the other hand, inference is performed many times during the lifetime of the device on which the CNN is implemented possibly for timing critical missions with limited power budget such as object detection and classification in automotive applications. Therefore in this thesis we focus on static inference operations of CNNs which are more common and power critical.

1.2 CNN Implementations on FPGAs and GPUs

High performance and energy efficient CNN implementations are essential for low-power and timing-critical systems. Bigger-sized networks with high number of layers, parameters and weights for improved accuracy increase the demand on performance and energy efficiency.

As it will be discussed in more detail in Chapter 2, CNN inference operations are highly parallelizable which makes FPGAs and GPUs favourable platforms on which CNNs can be built. GPUs include many cores which provide high levels of parallelization and performance at the expense of power. Despite their relatively lower clock frequency compared to GPUs, FPGAs can offer power efficient implementations [9]. CNNs can provide highly accurate outputs without using full-precision floating point data types [4]. GPUs can benefit from lowered precision for certain data types in terms of energy efficiency. FPGAs, however, show higher levels of flexibility and can be configured for arbitrary bit-width integer operations.

FPGAs, GPUs and CNN implementations on these platforms have been heavily researched. Colangelo et al. [4] investigated the relations between accuracy of CNNs and throughput on FPGA platforms for inference with low-precision data types. Abdelouahab et al. [10] presented efficient implementation methodologies of CNN inference on FPGA platforms. Li et al. [9] suggested reducing bit-width down to 1 bit for FPGA implementations, suggested methods to realize such compression with training and compared their inference performance results with other results

for FPGA and GPU platforms cited from various papers. Specialized libraries for CNN applications on GPU platforms have been published and widely being used in industry applications [11]. Wong et al. [12] created and published a compact 8-bit object detecting CNN implementation on a GPU platform and compared its inference performance results with competing implementations. Yao et al. [13] implemented several image recognition and object detection algorithms on GPU with different precisions using quantization and published inference performance and energy efficiency of these CNN implementations layer by layer. Results of these studies provide enough data of performance and energy efficiency to compare FPGA and GPU implementations of CNN inference. However these studies either run CNN inference on GPU platforms without accelerator support or do not emphasize or mention the utilization of accelerator cores during inference which makes it impossible to have a reliable comparison between FPGAs and accelerated GPUs.

1.3 Our Contribution

Some of the modern GPU based SoCs include accelerator cores for deep learning applications. Unlike GPUs without accelerator cores, accelerated GPUs which include accelerator cores can perform low-precision operations with high performance and energy efficiency. Accelerated GPUs are efficient for data types with which their accelerator cores work, therefore performance and energy efficiency of FPGAs for low-precision operations should be compared with accelerated GPUs instead of bare GPUs. Furthermore, even though accelerated GPUs provide high efficiency for certain low-precision data types, FPGAs still has the advantage of being able to carry out operations with low-precision arbitrary bit-width data types due to their vast number of LUTs and flexibility.

Our primary goal for this thesis is to design, implement and compare the performance and efficiency of major CNN inference building blocks such as convolution, fully connected and pooling individually on both FPGA and accelerated GPU platforms. We will take into account and study different design methodologies for efficient implementations. We will suggest optimized building block designs and evaluate, measure and benchmark performance and energy efficiency of these implementations considering common key metrics for both platforms to provide better comparison results with each other and with the results from research papers for various low-precision data types supported on respective platforms.

Utilization of low-precision data types might reduce accuracy of a CNN. A CNN with rounded parameters might have to be retrained to regain the lost accuracy. CNNs even have to include more number of layers and nodes to keep same accuracy levels for low-precision data types. Such optimizations and analysis might be performed case basis and are out of the scope of this thesis. As mentioned before CNNs can achieve acceptable accuracy levels by utilizing a wide range of data types from binary to floating points and our purpose is to investigate building blocks for different data types.

1.4 Thesis Overview

We start off by giving a brief introduction on CNN along with mentioning related research work and our contribution in the current chapter. In the following Chapter 2, we will give more in-depth CNN description, the theoretical information of major CNN building blocks and design methodologies and discuss the performance metrics needed to benchmark our results for both platforms. Chapter 3 will help the reader to understand the toolchain and describe the workflow for FPGA and GPU implementations. In Chapter 4, we will include our results of the common metrics for the respective low-precision data types, compare them with the state of the art and each other and comment on our findings. In the final Chapter 5, we will give the conclusions drawn from our work and suggest some future work.

2

Theory

There are vast number of sources in literature which discuss high performance energy efficient design methodologies for FPGAs and GPUs to implement CNNs. In this chapter we will explain general CNN structure, building blocks used in CNN inference, discuss design, verification and profiling of CNN implementations on FPGAs and GPUs and the performance metrics which are frequently used to compare different designs.

2.1 Structure of CNNs

Neural networks consist of numerous artificial neurons which are grouped into layers. Artificial neurons are connected to each other to form input-output relations. Similar to biological neurons, an artificial neuron combines several inputs to calculate a weighted average by assigning different weights to the inputs, adding an optional bias and applying an optional activation function to the obtained sum as shown in Fig. 2.1. A typical neural network consists of several consecutive layers every one of which is fed by the artificial neurons on the preceding layer and feeds the artificial neurons of the following layer with the outputs it generates. Fig. 2.2 shows a fully connected 4-layer neural network where each artificial neuron on each layer is fed by every artificial neuron in the previous layer. Such one-way propagation of outputs is called feed-forward network [14]. A CNN differs from any neural network due to the convolution layers which reduce the number of parameters of the CNN with convolution operations. As opposed to a fully connected neural network where there is an assigned weight for every artificial neuron pair between two consecutive layers, convolution layers in CNNs use the same kernel filter with weights throughout the input layer. Reuse of a small set of weights to define the relations between two consecutive layers enables the scale-up of image sizes by keeping the number of parameters of CNN and its memory footprint at reasonable levels [1].

Feed-forward operations are used for inference, whereas CNNs can make use of feedback loops for training phase. There are several methods for training and learning in CNNs. In unsupervised training unlabeled inputs are provided to train the CNN for image clustering purposes based on the similarities and the differences in the training data set. Supervised learning where the CNN is provided the labeled inputs can be used for image classification purposes. In supervised learning CNNs

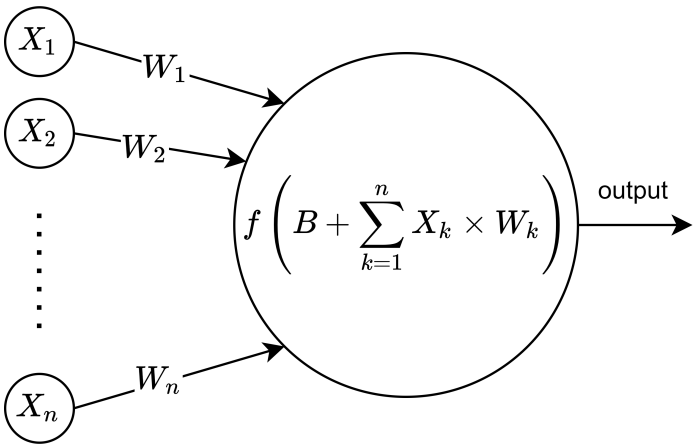

Figure 2.1: An artificial neuron with activation inputs, weights, bias and an activation function inspired by [1].

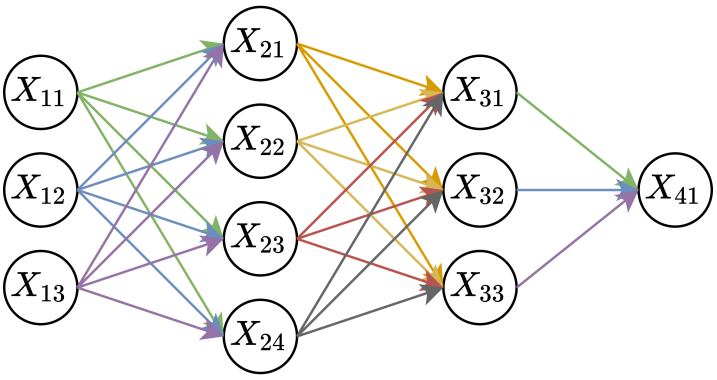

Figure 2.2: A fully connected neural network with 4 layers inspired by [1].

can learn by adjusting their weight parameters according to the error they get at their output in each iteration of the training phase. Weight adjustments can be performed with backpropagation by making use of the error gradient [14]. In this thesis we did not focus on feedback loops and backpropagation on CNNs and only considered the building blocks for inference operations for feed-forward networks.

CNNs have numerous layers with different purposes and functionalities. Layers can perform feature extraction, add non-linearities to processing in order to detect non-linear relations, prune unnecessary information and reduce the workload of the following layers or connect different parts of the images or different feature extractions to detect variety of correlations. Number, type and order of the layers together with the selection of the parameters of the layers are crucial decisions for a network to perform the desired task with high accuracy, performance and efficiency. There are limited number of types of layers in a CNN among which convolution, pooling and fully connected will be discussed in Section 2.2 as some of the major building blocks for CNN inference. An example 3-layer CNN given in Fig. 2.3 illustrates some of the major layers where input image is convolved with 3 different kernel filters to extract various features, pooling reduces the computational intensity and fully connected layer combines outputs of all artificial neurons to calculate the probability of the image belonging to each one of the 10 classes as inspired by [1].

Although there are only a few types of layers in a CNN, these layers have design parameters which provide flexibility in CNN design but also add a challenge to efficient implementation of CNNs on hardware platforms due to variety within layer types. In a typical CNN, several convolution layers are used to extract different features. Each convolution layer can be followed by a pooling layer which reduces the output tensor size and indirectly affects the sizes of all the following layers and consequently requires several number of implementations of the same types of layers in a single CNN for different tensor sizes. Differences in CNN structure and the connections between the artificial neurons in separate layers also bring new challenges for an efficient hardware implementation. As an example, having multiple convolutions with the same feature map with different filters can demand more computational resources whereas using different feature maps for every convolution demands more memory and bandwidth. Building blocks should provide required flexibility and performance for different kinds of CNN implementations.

2.2 Building Blocks for CNN Inference

CNN inference operations can be implemented as a combination of a few building blocks. These building blocks can also be implemented as separate optimized entities, possibly on accelerators for improved energy efficiency. In this section we will discuss some of these building blocks.

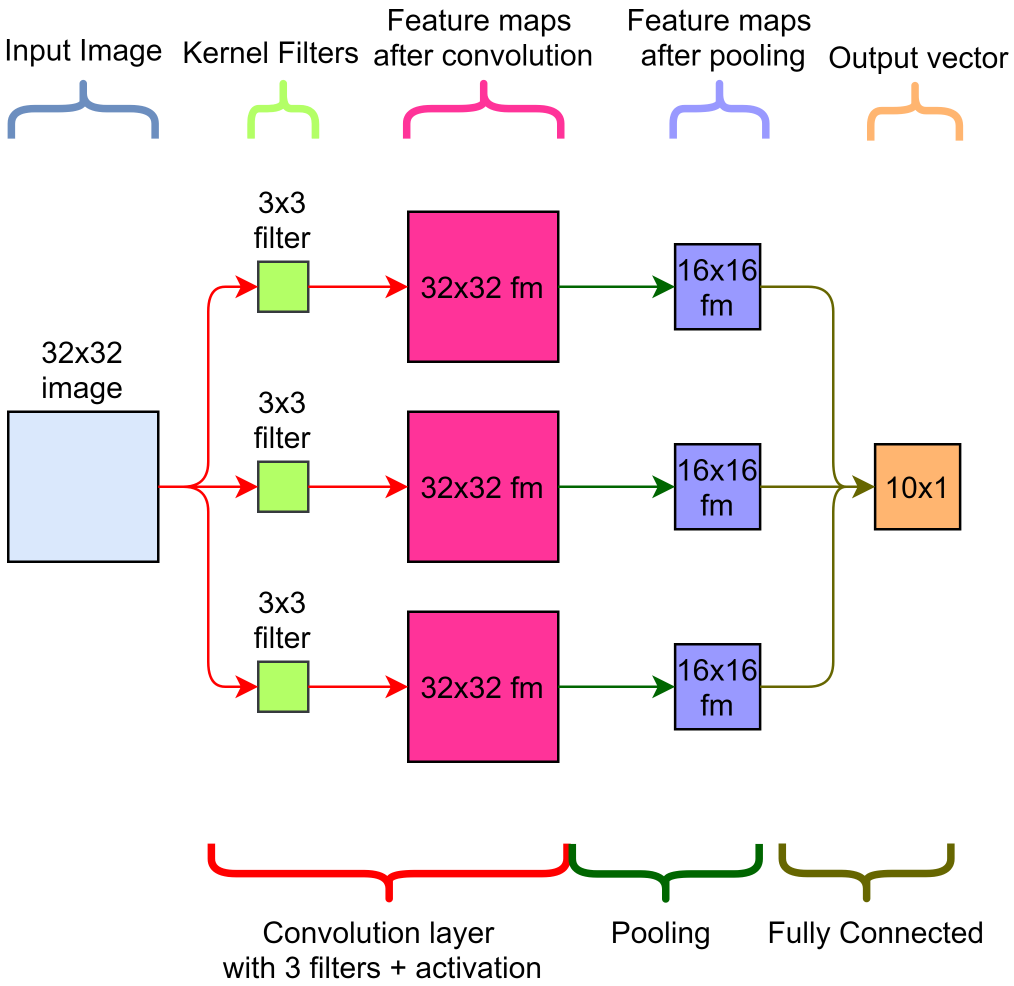

Figure 2.3: A simplified CNN example which classifies images into 10 classes inspired by [1].

2.2.1 Convolution

Convolution is an essential part of CNNs which provides feature extraction using local correlations. In this stage input tensor which is typically an image or a video frame is divided into smaller pieces and convolved with a kernel filter which helps in detecting certain features such as edges. Use of different weights provides unique properties to kernel filters. Same kernel filter can be used throughout the input tensor, which provides weight sharing and reduces the memory footprint of the CNN [3].

2D convolution can be formularized as

$$\text{Out}[i,j] = \sum_{n=0}^{N-1} \sum_{m=0}^{M-1} in[i-n, j-m] \cdot k[n,m] \tag{2.1}$$

where *out* is the output feature map, *in* is the input tensor, k is the kernel filter with N rows and M columns. Although this formula presents the general 2D convolution mechanism, CNNs frequently utilize additional functionalities called stride and padding to handle edge cases differently and reduce network size [15]. These parameters define how kernel filter moves around the input tensor to calculate outputs. Stride defines the number of cells kernel filter shifts in x or y direction between two consecutive operations. Stride can be useful to reduce computational demand of a convolution layer when information carried by neighboring cells overlaps enough to permit exclusion of some outputs without losing much information. Stride is by default equal to one in a standard convolution operation. Padding around the input tensor can be used to adjust the output feature map size by defining the kernel filter behaviour at edges and corners of the input tensor. Information loss at the edges can be prevented with padding. Zero padding is typical in CNNs. Overall, size of the output feature map in one dimension is given by

$$\text{OutputSize} = \frac{F + 2P - K}{S} + 1 \tag{2.2}$$

where F is the input tensor size, P is the padding size, K is the kernel filter size and S is the stride factor [15].

It is common to use 3D input vectors as input data for CNNs as it is the case for RGB and magnetic resonance imaging (MRI) which are stacks of imaged layers. Different convolution methods are being applied to 3D images. It is possible to apply separate 2D kernel filters to every 2D layer of the input image. However, in order to extract spatial features, 3D convolution of 3D input and 3D kernel filter can be necessary. In case filter depth is smaller than image depth, filter can move in all 3 directions and output a 3D tensor. Moving a 2D filter in all 3 axes in 3 separate channels is also an alternative to 3D convolutions which can preserve 3D features [16]. Due to its common use cases and simplicity, in this thesis we focused on 2D convolutions.

In a typical CNN, kernel filter size is much smaller than input tensor and feature map sizes. Small kernel sizes reduce the number of parameters in a CNN, training time and computational complexity of inference. As an example, VGG uses 224×224 RGB images where kernel filter size is only 3×3 [8]. AlexNet processes the same 224×224 RGB image size with a combination of 3×3, 5×5 and 11×11 kernel filters [17]. Using odd-sized kernel filters is typical as they preserve symmetry and generate an output window with a center pixel.

2.2.2 Fully Connected

Outputs of convolution and pooling layers carry the features from a limited spatial domain. Investigating dependencies and relations between different features of different parts of the input can improve accuracy of the final estimations. Fully connected layers combine outputs of the neurons of the previous layer as a weighted sum with some bias factor and pass it through an activation function in order to detect inter-neuron relations. Fully connected layers are typically placed at the end of the network after all convolution layers. Fully connected layers include different weight sets for every input-output neuron pair they include, therefore typically include the most number of design parameters in a CNN, which also increases their memory footprint [18].

Operations in a fully connected layer for any one of the n_l output neurons excluding the activation function can be described as

$$z_i^{(l)} = b_i^{(l)} + \sum_{j=1}^{n^{(l-1)}} \omega_{i,j}^{(l)} y_i^{(l-1)} \tag{2.3}$$

where $z_i^{(l)}$ is the value of a neuron at the output layer (l^{th} layer), $b_i^{(l)}$ is the corresponding bias value, n_{l-1} is the number of input neurons at the input layer ($(l-1)^{th}$ layer), $\omega_{i,j}^{(l)}$ is the corresponding weight value and $y_i^{(l-1)}$ is the value of a neuron at the input layer [19]. Such fully connected layer requires

$$\text{NumberOfParameters}_{\text{fc}} = (n_{l-1} + 1)n_l \tag{2.4}$$

parameters to be stored including the weights ($\omega_{i,j}^{(l)}$) and the biases ($b_i^{(l)}$).

Effectivity and functionality of fully connected layers highly depend on the activation functions they use. In fully connected layers, neurons take several inputs and a bias value in order to calculate one output. Activation functions define how these inputs will be combined and transformed into an output. Convolution operations and weighted sum of convolutions are linear by their nature. In both training and inference, non-linearities can be desirable to detect non-linear features of the input. Activation function can be linear or non-linear in order to broaden the set of features that can be detected. There are different kinds of activation functions with varying

purposes which are used at different parts of CNNs. Rectified linear unit (ReLU) is the most widely used activation function [20]. It preserves some linear properties due to its partial linearity, has higher performance and speeds up training process. Softmax function outputs between 0 and 1 where sum of all outputs are equal to 1, and used to convert outputs into probabilities at the final output layer [20].

2.2.3 Pooling

Pooling is a downsampling operator which is typically applied after convolution layers. Once a convolution layer has extracted the features using spatial correlations, feature information can be transferred to the following layers without exact knowledge of the feature location. Pooling can reduce network size and computational intensity without accuracy loss. Pooling can also prevent over-fitting which happens when a CNN is overly fit into its training data set with little error margins causing it to lose its flexibility and giving erroneous results for some inputs that are not similar to the training data set. Pooling, similar to convolution kernel filter, is applied locally on the input tensor and moved around the input tensor, possibly with features such as stride and padding. Various types of pooling are used in CNNs. Max pooling and average pooling can be performed by taking the maximum and average of a local region, respectively [3].

2.3 FPGA Implementations of CNNs

FPGAs provide flexible design opportunities for CNNs. Designer can maximize the performance and the energy efficiency by utilizing the appropriate design methodologies. In this section we will discuss the design methodologies for FPGA implementations of CNNs, power estimation and profiling techniques and AXI to transfer data.

2.3.1 Design Methodologies

FPGAs offer highly parallel computation opportunities for CNN inference. Exploiting parallelism while keeping hardware utilization and power consumption at acceptable levels might require careful design strategies. Methods for efficient hardware architectures have been studied in research.

CNN inference operations are parallelizable at different levels. CNN can utilize batch parallelism by using the same kernel filter to process more than one frame which also increases data reuse. Operations on one layer depend on the outputs of the previous layers. Pipelining layer operations considering these dependencies and starting operations of a layer before its all inputs are ready can provide inter-layer parallelism. Different feature maps can be processed independently which is called inter-feature map parallelism. Intra-feature map parallelism can be achieved by processing different parts of a feature map simultaneously. 3D convolutions can be calculated by summing up 2D convolution results which permits inter-convolution

parallelism. Different parts of a 2D convolution array can be processed in parallel which is called intra-convolution parallelism [10].

Reducing the precision of the data types used can reduce the power and hardware resource consumption of CNNs implemented on FPGAs while preserving reasonable levels of accuracy [4]. However utilizing low-precision data types might not be enough, architecture should exploit low-precision in an efficient manner. DSPs are optimized for efficient multiply-add operations which are widely used in CNNs. Multiply-add operations for fixed-sized data types with at most 16 bits can fit inside DSP blocks in XILINX FPGAs. Low-precision data types with a bit-width lower than 16 might cause underutilization in DSP blocks. Merging multiple operands into one operand to perform virtually multiple multiplications with one 16-bit wide DSP multiplication can be used to increase the utilization ratio of DSP blocks. It is also possible to use LUTs instead of DSPs to avoid underutilization for low-precision data types [21].

Different algorithms which fit into hardware better can be utilized to improve efficiency. Using 2D FFT to calculate 2D convolution can reduce power consumption. Assuming N is the input tensor size in one dimension and K is the kernel filter size in one dimension, a standard convolution operation has a computational complexity of $\Theta(N^2K^2)$ whereas the same output can be obtained with $\Theta(N^2 \log K)$ complexity by using FFT with overlap-and-add algorithm. It is important to note that CNNs typically use small kernel filters which limits the gain from this algorithm [22].

Many operations on CNN inference have limited data dependency which enables higher parallelization levels. Loop unrolling is utilized to exploit the parallelism in CNNs. As an example, bigger kernel filters can be divided into smaller kernel filters for convolution operations. Unrolling factor is a critical decision which affects the hardware utilization ratio. In case loop trip count is not divisible by unrolling factor, the hardware utilization ratio can be smaller than one. Memory access and data path are also affected by the unrolling operation. Data dependencies between loop iterations also must be carefully examined. Design space exploration besides other methods can be used to determine the optimum unrolling factor [21].

Memory utilization and data path management are critical aspects which affect throughput and maximum clock frequency. Input tensors, kernel filters and output feature maps can consume large amount of memory and not entirely fit inside on-chip memory. Data reuse for different computations can improve the efficiency of on-chip memory utilization. Broadcasting data into many parallel computational units can increase fan-out and limit clock frequency. Systolic array approach where data is moved serially between different computational units can reduce fan-out at the expense of latency [21].

Performance in terms of operations performed per time can be bounded by the computational capacity of the platform or the external memory bandwidth. A roofline model can explain the limiting factor for the system performance [2]. Performance is proportional to memory bandwidth and operations performed per byte transferred until total computational capacity of the system has been reached. Com-

putational roof can be defined as the maximum number of operations that can be completed on the system bounded by the hardware resources. Operational intensity or computation to communication ratio (CTC ratio) can be defined as the number of operations performed per memory units transferred from/to external memory. Theoretical maximum attainable performance of the system is bounded by the minimum of the computational roof or CTCRatio $\cdot$ BW where BW represents off-chip memory bandwidth [23]. Roofline model is visually represented in Fig. 2.4 [2]. In Fig. 2.4 design1, design2 and design3 share the same memory bandwidth and are bounded by the same bandwidth ceiling for small operational intensities. For higher operational intensities they are bounded by their computational ceilings which are different for each design which could be due to the differences between the amount of hardware resources they have. design4, design5 and design6 are bounded by the same computational ceiling for high operational intensities. For lower operational intensities they are bounded by their bandwidth ceilings which are different for each design which could be due to different memory access methodologies such as burst operations or prefetching. Roofline model is useful to locate the limiting factor for the performance while optimizing the design.

On-chip memory is typically not big enough to store feature maps and communication with off-chip memory is slow with high latency. Typically data has to be fetched and buffered in the on-chip memory in a way that minimizes the need to access to the off-chip memory especially for designs bounded by their bandwidth. Double buffering can be utilized to perform memory operations and computations simulta-

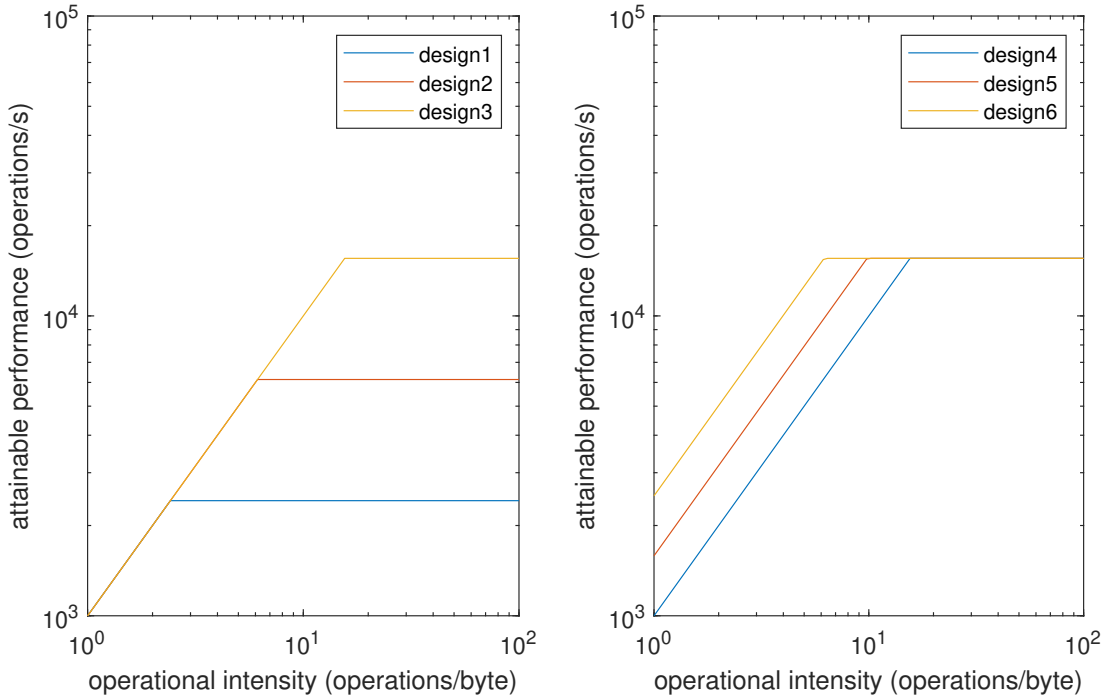

Figure 2.4: Visual representation of the roofline model with mock-up designs inspired by [2].

neously where one buffer works with the off-chip memory and the other works with computational units. Loop tiling can be utilized to efficiently utilize the on-chip memory. Loop tiling keeps the data that will be needed in the next loop iteration in a closer memory (e.g. LUTRAM or on-chip BRAM) instead of a further memory with higher latency and limited bandwidth (e.g. off-chip memory) and utilizes locality principle. Tile size affects the data paths, parallelism and computation to communication ratio and therefore must be carefully selected. Data dependencies between tiles must be taken care of in the outer loop. Increasing tile size can reduce bandwidth needed by increasing data reuse while increasing the on-chip memory required per tile. One can perform local memory promotion by moving memory access to the outermost possible loop which improves data reuse between loop iterations and computation to communication ratio. Balancing off-chip memory access and computation speed is possible with proper selection of loop unrolling and tiling factors. Data dependencies between loop iterations can limit the gains from loop unrolling. Pipelining is a key technique to resolve data dependencies while keeping throughput and hardware utilization ratio high [23].

Systems do not always fully utilize the attainable memory bandwidth which further reduces the bandwidth ceiling in the roofline model. Different CNN layers have different needs for computational power and bandwidth, as an example fully connected requires more bandwidth than computational power as opposed to convolution layer. Getting actual bandwidth closer to the theoretical maximum bandwidth of the system is important especially for layers with higher need for bandwidth to get a balanced design. Especially for tiled designs, input tensor reads from and output feature map writes to the off-chip memory follow a mixed pattern which can be quite different from the row or column order. Accessing scattered addresses in the off-chip memory for a small amount of data proportional to the tile size obstructs the use of long burst accesses. Arranging data in the off-chip memory in the same order as computational units need enables the use of long burst transfers which improves actual off-chip memory bandwidth [24]. Loop interchange can also be utilized to make on-chip and off-chip memory access more organized [10].

2.3.2 Power Estimation

Having good power estimations with high confidence levels is an essential part of this thesis to compare energy efficiency of the designs. FPGA design software can include power estimation tools as well since power closure is a problem that should be considered starting from the early stages of the design for modern high performance systems.

Power consumption is equal to the sum of dynamic, static and short circuit power consumptions [25]. Short circuit power can be minimized with careful design of transistors hence static and dynamic power are the biggest contributors to the power consumption. Static power is caused by the leakage currents on transistors. Static power is not negligible for newer scaled-down process nodes due to the increasing sub-threshold leakage and therefore must be taken into account. Relative contribution of static power is even higher for systems that underutilize the hardware.

Dynamic power consumption is generated by the losses during switching activity where capacitance of the transistors is getting charged and discharged. Dynamic power consumption is given by

$$P_{dyn} = \frac{1}{2} C_{out} V_{dd}^2 \alpha f_{clk} \tag{2.5}$$

where C_{out} is the total MOS transistor capacitance, V_{dd} is the supply voltage, α is the gate transitions per clock cycle and f_{clk} is the clock frequency [25]. At post-synthesis stage an EDA tool has a good estimate of C_{out}. User can also enter V_{dd} and f_{clk} that the FPGA will use in the tool. It is necessary to have a realistic value of α for every gate to obtain accurate dynamic power estimations which can be achieved by several methods.

Probabilistic methods can be used to obtain correct switching information. User can enter switching statistics of input nodes which are toggle rate of the nodes and static probability of the nodes being high at any moment. These switching statistics can be propagated through the design with simulations. For better estimations, the tool needs to consider spatial and temporal correlations of the switching activities between nodes. The tool also needs delay models of the gates in order to detect the switching activities at glitches. Probabilistic models provide fair estimates of the power consumption at earlier stages of the design [25].

Another method is simulation-based techniques. It is possible to obtain the switching activity of the design by running simulations. With simulation-based techniques it is crucial to run enough number of simulations which cover a wide range of possible switching activities. Like probabilistic models, it is required to have delay models to detect glitches. Running real test cases to obtain the switching activity gives good estimates of the power consumption but is computationally intensive [25]. Montgomerie-Corcoran et al. [26] showed that using Xilinx Power Estimator or Vivado post-implementation estimation without simulation to estimate power consumption of a CNN inference can be 624% and 560% erroneous respectively whereas using simulation-based estimation with Vivado can reduce error down to 5% at the expense of increased evaluation time.

2.3.3 Advanced eXtensible Interface (AXI)

AXI is a communication protocol which is widely used by XILINX IP cores. AXI buses can provide high performance parallel communication with different bit-widths and supported by extensive documentation and IP support. Therefore we used AXI to interface our FPGA building blocks.

Different AXI protocols can be utilized for systems with different complexities. Memory-mapped communications can use a high performance AXI4 interface which also allows burst operations. AXI4-Lite is also a memory-mapped protocol which includes a subset of the signals in AXI4 with limited functionality. AXI4-Lite is a simple solution with relatively lower resource utilization for systems with low

throughput demands, such as control signals. AXI4-Stream is for streaming data with high bandwidth efficiency without an address phase [27].

In AXI4 interface master initiates all write and read operations. Data can be transferred bidirectionally and simultaneously through separate channels. AXI4 protocol is flexible and can be customized to improve bandwidth. Data throughput can be increased by changing port bit-width, buffering outstanding read and write operations and bursting multiple data by specifying only one address [27]. Reaching the theoretical memory bandwidth specification of a device can require the use of a minimum port bit-width and consecutive access length. Increasing the number of ports also improves memory bandwidth. Although it is necessary to have enough data flow from memory to obtain high performance, aggressive improvements on bandwidth might require arrays in the on-chip memory to be partitioned and therefore increase utilization, reduce memory efficiency and performance. Hardware utilization and computational limits of the device should be considered while memory bandwidth is being improved [28].

AXI modules contribute to the resource utilization of the overall system. Utilization of an AXI module depends on its functionality. An AXI module that is expected to handle longer burst operations, higher number of outstanding calls and higher bit-width typically utilize more hardware and FIFO buffers. These parameters should be chosen so that they will allow prefetching, provide low enough latency and high enough memory bandwidth and at the same time will not utilize unnecessarily large amount of hardware. Relations between such parameters and resource utilization have been studied with some benchmarking tools developed for high level synthesis design tools [29]. Such theoretical estimations can be used at the design stage and synthesizer results for AXI utilization can be monitored through utilization reports provided by the design tool.

2.4 GPU Implementations of CNNs

In this section we will give a brief introduction to general GPU structure, explain some of the important libraries and accelerators and introduce some power measurement methodologies.

2.4.1 GPU Introduction

Graphic processing units (GPU) are widely used for parallelizable applications. GPUs include many cores and several levels and types of memories to run operations concurrently while considering data dependencies. Tasks can be shared among threads which can run on separate cores concurrently. Multi-core central processing units (CPU) can perform parallelization as well, however the number of cores is limited and context switching between threads running in parallel can be costly. GPUs need to perform context switching, parallelization of tasks among threads and memory synchronization in more efficient ways than CPUs which requires parallel computing models. Compute Unified Device Architecture (CUDA) is a parallel

computing and programming architecture created by NVIDIA [30].

CUDA runs on a CPU based host computer and can call GPU to run parallelizable operations with high performance. These GPU calls are defined as kernels. Kernel divides operations into many threads within blocks within grids. Threads under one block share a high-speed memory and can be synchronized for operations with data dependencies. Different blocks should be able to work independently. Blocks running in parallel provide coarse grain parallelism. Tasks which are required to be performed sequentially due to data dependencies can be called with atomic calls [30].

Several layers of memories in GPUs must be utilized efficiently for performance and energy efficiency. Use of shared memory and registers gives the fastest memory access. It is significant to improve data reuse in such closer and faster memories and reducing memory access to global memory can improve performance and power consumption. Due to its significance, memory access aspects of algorithms must be considered when selecting an algorithm. As an example, in DNN applications, a method called "lowering the convolution" which can be performed by unrolling nested loops that define the convolution and converting the convolution operation to a few matrix multiplications can improve performance significantly. However this method requires several copies of data in memory which reduces memory performance. Considering the consequences of loop unrolling from memory perspective and preventing data duplicates as much as possible during the design process can save memory, improve performance and reduce power consumption [31].

Libraries such as CUDA Deep Neural Network (cuDNN) to perform CNN operations efficiently have been developed [11]. cuDNN includes primitives to provide high performance optimizations for standard layers like convolution, pooling, fully connected and activation in DNNs.

2.4.2 TensorRT

TensorRT is a specialized C++ based library built on CUDA for optimizing deep learning inference models of pre-trained networks for NVIDIA GPUs. TensorRT is specialized in improving performance of neural networks with lower precision values like INT8 and FP16. It supports Python language as well. It can work along with different frameworks like Caffe, TensorFlow and PyTorch to contribute to throughput and efficiency [32].

In TensorRT, pre-trained network model prepared with some other workflow can be imported and loaded with the available C++ parser API in TensorRT form. Another method could be adding network definition from scratch with C++ API and then building the engine by calling the TensorRT builder for the supported precisions required to execute the network. Then, the inference can be carried out on the inference engine built. TensorRT provides the flexibility to set the precision for every layer independently [32].

2.4.3 Accelerator Cores and NVDLA

Accelerator cores can contribute to energy efficiency and performance of GPUs. As an example, NVIDIA includes tensor cores in its Volta architecture. As opposed to regular cores in GPU, tensor cores cannot execute instructions from the whole set of instructions. Instead, tensor cores are specialized for matrix multiply-accumulate functions and execute them in one clock cycle [33]. Tensor cores in Volta architecture can work with FP32 or FP16 data types whereas Turing architecture also includes INT8, INT4 and INT1 [34].

NVIDIA Deep Learning Accelerator (NVDLA), or shortly deep learning accelerator (DLA), in NVIDIA devices can also be utilized for high performance energy efficient implementations. DLA is a hardware architecture that is designed specifically to perform accelerated inference operations that are computationally intensive as it provides high performance solutions. Mathematical operations in DNNs and their memory access are predictable which makes specialized hardware architectures more viable. DLAs support different common operations in DNNs such as convolution, fully connected, pooling and activation, also provide several options to perform the same operation for optimized performance. As an example, convolution can be carried out in four different ways including direct convolution and Winograd convolution. DLAs include several cores for different operations. Individual linear or non-linear functions can be applied to certain elements using Single Data Point Processor. Planar Data Processor supports pooling operations with different sizes. Bridge Direct Memory Access (BDMA) module configures and reshapes the date transferred from DRAM to enable faster operations. Efficient external interfaces and busses are used for communication [35].

As configurable hardware entities, DLAs provide flexible operation for different tensor sizes and data types which improves energy efficiency and performance for smaller systems. Several data types within the same network can be utilized where conversion between data types are also handled by DLAs. DLAs can also be used in different modes. Cores within a DLA can be used individually in independent mode, but they can also pass data to each other directly in fused mode which improves efficiency [36].

2.4.4 Power Measurement

Understanding GPU power consumption is crucial to locate the problems and inefficiencies for further development. GPU power can be modelled, profiled or measured. Different benchmark software are being used to compare different platforms. These benchmark software can be intended to load a specific area in a GPU or represent a common use case [37].

Measuring power with sensors is an accurate way to quantify power consumption. Moreover, measurements can be used to validate any model or simulation as a first step. Reading power in periodic intervals and extrapolating from the measured data to estimate energy consumption is a viable method. Measurements can be taken by internal or external sensors. Some GPU platforms provide sensors within the

hardware which enables detailed power readings from more localized parts or cores of the GPU. Such internal sensors enable measurements without external modifications in the hardware. However their measurement frequencies and resolutions might be limited and documentations might be lacking details [37].

Although measuring total power consumption gives an idea of energy efficiency, more detailed models are needed for optimization purposes. Power consumption of individual operations such as memory access or arithmetic operations can be estimated from measurements. GPU power models such as Activity-Based Model can be utilized to obtain fine-grained power estimations from measured total power [38]. Activity-Based Model establishes arithmetic formulas to model the total power consumed where unknowns of the formulas including power consumption of components and base power can be calculated after a few measurements. Estimating power consumption of individual cores or micro-architectures enables designer to make better power-oriented optimizations for complex systems. Micro-benchmarks can be utilized to load specific cores with specific operations to obtain more accurate power estimations.

GPUs consume some amount of power even when not loaded which affects the measured power and therefore commonly included in GPU power models. Hong and Kim formulate the total power consumption of GPU as

$$\text{GPU\_power} = \text{Runtime\_power} + \text{Idle\_power} \tag{2.6}$$

where *Idle_power* is measured when GPU is on without any application being executed and *Runtime_power* is the additional necessary power to run a program [39]. Power required to run an application on an already powered-on GPU can be approximated with this model. It is important to note that power gating on cores which is very common in modern GPUs cuts both *Idle_power* and *Runtime_power* in this model.

2.5 Performance Metrics

Many CNN implementations have been performed in research on FPGA ([4] [9] [22] [23] [24]) and GPU ([12] [13]) platforms whose results can be used as references for future implementations. Performance metrics of these implementations should be understood to have meaningful and fair comparisons.

In CNN implementations some papers prefer to present their results according to the number of images processed per unit time or energy consumed per image for a given data set. Since image and kernel filter sizes and computational workload of different CNNs vary and we are focusing on building blocks instead of full CNN implementations, we used more universal metrics that can be shared between data sets and are applicable to single building blocks.

Number of operations performed is needed to calculate some of the performance metrics. Number of operations can be analytically calculated for a given program according to some design specifics which can vary among building blocks and layers. Number of operations for convolution depends on input tensor size, kernel filter size, padding and stride. For a 2D convolution operation where there is no stride or padding, number of operations is equal to

$$\text{NumberOfOperations}_{\text{conv2d}} = 2n_{out}n_{in}h_k\omega_k(h_{in} - h_k + 1)(\omega_{in} - \omega_k + 1) \tag{2.7}$$

where n_{out} and n_{in} are the number of output feature maps and input tensors, h_k and ω_k define the kernel filter size, h_{in} and ω_{in} define the input tensor size [40]. Result gives the number of multiply and accumulate operations required for convolution operation and can be easily adapted for convolution with padding and stride. Factor 2 comes from the separation of additions and multiplications as different operations.

In a fully connected layer all outputs of the input layer are multiplied with corresponding weights and added up with each other and a bias factor. Hence number of operations for a fully connected layer is given by

$$\text{NumberOfOperations}_{\text{fc}} = 2n_{in}n_{out} \tag{2.8}$$

where n_{in} is the number of neurons in the previous layer (e.g. NM for a fully connected layer connected to $N \times M$ feature map output of a convolution layer), n_{out} is the number of neurons in the following layer and factor 2 comes from the separation of additions and multiplications.

Number of operations in a max pooling layer without padding or stride is given by

$$\text{NumberOfOperations}_{\text{maxpool}} = n_{in}(h_k\omega_k - 1)(h_{in} - h_k + 1)(\omega_{in} - \omega_k + 1) \tag{2.9}$$

n_{in} is the number of input tensors, h_k and ω_k define the pooling window size, h_{in} and ω_{in} define the input tensor size. For every output value calculated, $h_k\omega_k - 1$ comparison operations must be completed. Since there is no separate multiplication and addition operations, result is not multiplied by 2 unlike convolution. In average pooling, for every output value calculated, a division operation in addition to $h_k\omega_k - 1$ addition operations must be performed, hence number of operations is given by

$$\text{NumberOfOperations}_{\text{averagepool}} = n_{in}h_k\omega_k(h_{in} - h_k + 1)(\omega_{in} - \omega_k + 1) \tag{2.10}$$

It is significant to note that types of operations differ among max pooling, average pooling and convolution.

Latency is a significant performance metric for timing critical systems. Latency target is application specific, but a reasonable target might be finishing processing an image before the next one starts [41]. Therefore 25 ms is a reasonable target for 40FPS camera input in an automotive application.

Throughput defines the number of operations that can be performed per unit time and a common unit is giga operations per second (GOPS) which can be easily calculated given that number of operations and run time is known. In literature some designs have been compared by their GOPS performance [9]. Papers which focus on comparing algorithms rather than hardware platforms tend to also include resource efficiency with a unit of GOPS/kLUT for FPGAs which provides a fairer comparison between different hardware platforms manufactured with different process nodes. Energy efficiency with a unit of GOPS/W can be calculated from the number of operations and measured or estimated power consumption and provides a performance metric significant for systems with limited energy budget. For low-precision applications data types used must be taken into account in comparisons.

3

Methods

In this chapter we will discuss the tools, algorithms and implementation details for both platforms.

3.1 FPGA

FPGAs and hardware description languages allow flexible design methodologies and give high levels of control to designer due to their low-level nature. In this section we will discuss the toolchain used and the algorithm details for our proposed 2D convolution, fully connected and pooling blocks.

3.1.1 Overview of Tools

Software tools for FPGAs support design, implementation and verification. It is possible to test and verify the system at different design stages, also to get good estimates of power and resource utilization. High level synthesis (HLS) tools are also being used in FPGA design to facilitate the design process and reduce the time spent on the design. Research teams which work on FPGA implementations of CNNs have also worked with HLS tools [23]. Fig. 3.1 visualizes the toolchain used as it is described in this section.

We used PYNQ-Z2 board as hardware to verify our FPGA designs. PYNQ-Z2 includes ZYNQ-7020 SoC with ARM Cortex-A9 dual-core processor as processing system (PS) and FPGA fabric as programmable logic (PL). The PL includes 53200 LUTs, 220 DSP slices and 630 KB BRAM [42]. It is possible to implement accelerators in the PL called "overlays" which can be called by the PS like a software library function. We have implemented our FPGA building blocks as overlays in the PL to be used by the PS. In order to develop overlays we used Vitis HLS (2020.2) and Vivado (2020.2) on our computer with Windows 10 Enterprise operating system. In order to run the PS which calls overlays we used Jupyter notebooks.

3.1.1.1 High Level Synthesis and Verification

We used Vitis HLS for high level design and synthesis of our building blocks. Vitis HLS supports C programming and some C libraries. C code can be developed and tested in C level with built-in simulator. Finished and tested C code can be

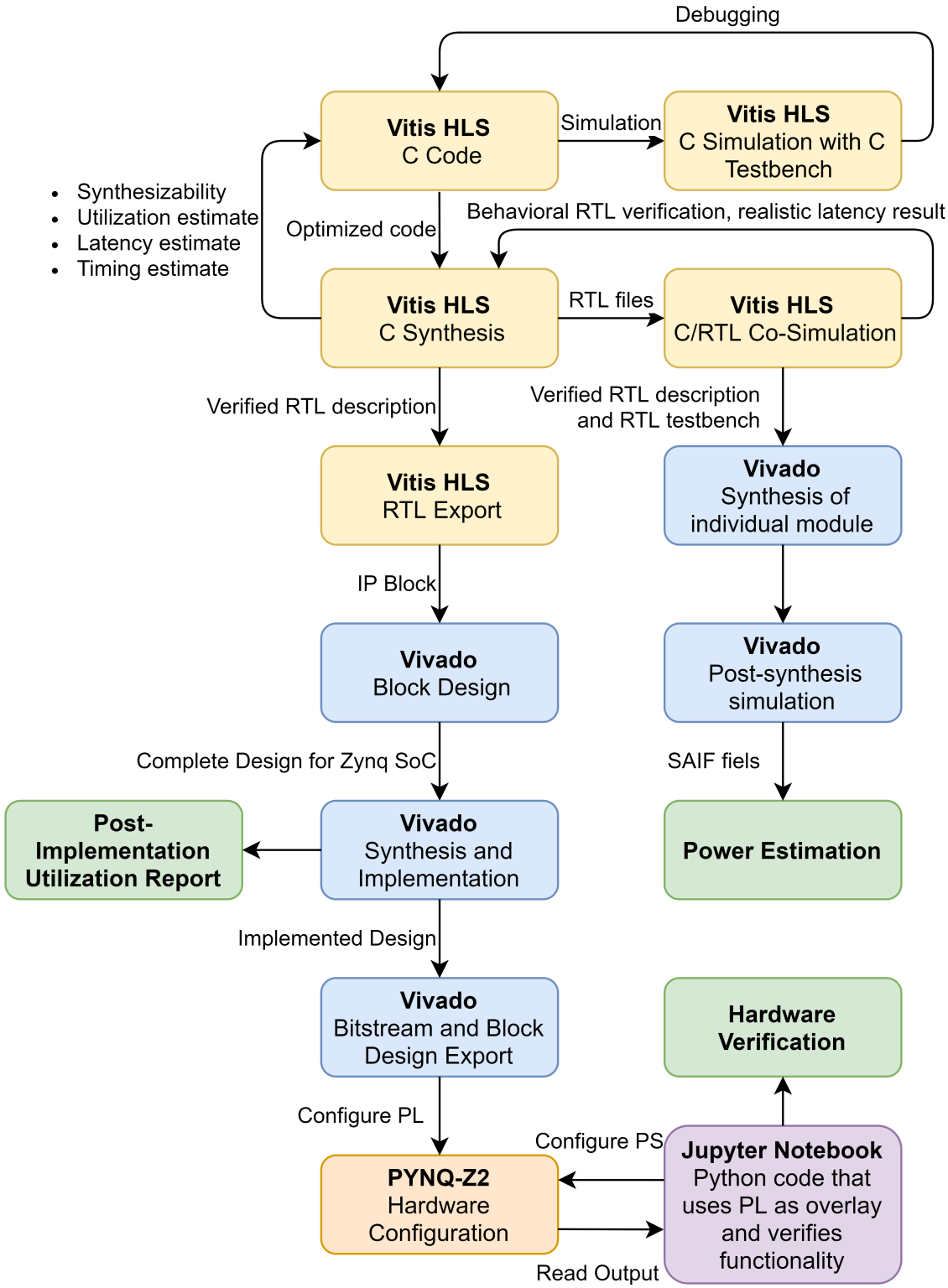

Figure 3.1: Overview of the tools used in FPGA design, verification, power and performance estimation stages

synthesized by Vitis HLS. At this stage Vitis HLS also checks the synthesizability of the C code. At synthesis stage Vitis HLS generates RTL descriptions of the C code in VHDL and Verilog. It also provides estimated resource utilization and latency information depending on the target hardware device. Synthesized design which has been recorded as RTL descriptions can be further simulated in RTL level. Vitis HLS can provide C-RTL co-simulation by automatically converting the C simulation testbench into the corresponding RTL testbench in either VHDL or Verilog. RTL simulation gives a better estimate of latency values compared to the estimates provided after RTL code generation at synthesis stage. Input and output signals during RTL simulation can also be recorded to be run and observed after the simulation in Vivado. RTL simulation also records the testbench files which can be utilized afterwards for further simulations in Vivado environment. Finally verified design can be exported as an IP block in VHDL or Verilog to be used in block designs [43].

Use of pragmas separates Vitis HLS design from ordinary C programming. Pragmas can be used to manipulate synthesis stage from different aspects such as directing hardware design methodologies (e.g. pipelining, unrolling), restricting hardware utilization on certain resources, limiting latency and specifying memory type or computational unit to be used. C design also must be structured in a way that allows hardware optimizations [44]. There are numerous pragmas and their functions are not limited to the examples given above, hence we did not discuss all in this thesis and only explained the important ones we used as we describe the algorithms used.

We included many generic parameters in our designs including arbitrary bit-width of INT data types. These generic parameters are fixed pre-synthesis, meaning that a synthesized module cannot process different image sizes and data types. User who needs to generate an overlay with different parameters should enter these parameters at the very beginning of the design flow, in the C .h file where these parameters are declared and re-synthesize. Fixing these parameters at an earlier stage of the design flow might help the tools to achieve better optimizations by trading off some flexibility for task specialization.

3.1.1.2 Block Design and Power/Performance Estimations

Vivado is an FPGA development tool which supports both VHDL and Verilog, performs synthesis and implementation, generates hardware utilization reports and power consumption estimates at different design stages to help the designer. We used Vivado to build our design including building blocks inside PYNQ-Z2 and to get power estimations and hardware resource utilization information. Overlay which has been exported as an IP block can be included inside a block design in Vivado. Connections between the PS and the PL together with other connections with the board such as reset and clock signals are completed in the block design. Address assignments for memory accesses between the PS and the PL are also performed at the block design stage. Verified block design must be synthesized and implemented in Vivado. Vivado can also generate timing report for implemented design. Resource utilization reports at post-implementation stage were used for performance

comparisons. Implemented design can be exported to configure PYNQ-Z2.

We also performed power estimations for our overlays which were implemented on PYNQ-Z2. The Verilog testbenches and design files exported during synthesis and C-RTL co-simulation of Vitis HLS can be imported by Vivado for further investigation of the design. We performed behavioral simulation and synthesis of our overlay designs in Vivado from the imported files. We set clock frequency and generated switching activity interchange format (SAIF) files which include switching activity of the nodes by performing post-synthesis functional simulations in order to make simulation-based power consumption estimations with high confidence levels. We used these high confidence power reports for analysis and comparisons with results from research and accelerated GPUs.

We used random numbers covering full range of the utilized data types in order to obtain SAIF files. Ideally using a large set of realistic images and obtaining SAIF files and performing power measurement for every one of them would provide more accurate results. However post-synthesis RTL simulations take time. Therefore we investigated different data sets to obtain a single data set that represents all kind of operations. As an example, for convolution, input tensor and kernel filter has to be decided before the simulation. We used several road and object images from BelgiumTS data set [45] and edge detector kernel filters and compared results with some randomly generated input tensors and kernel filters. We observed that power consumption does not considerably change for different kernel filters and input tensors as long as these are non-zero. Also considering input tensors in inner layers are feature maps and not images, we used random numbers as a representative of an input tensor instead of a real image due to time and computing power limitations.

Latency of the system was not deterministic at Vitis HLS synthesis stage due to uncertainties at AXI port interface stimulation. Therefore post-synthesis simulations gave higher latency than the estimations at the synthesis stage. We mitigated the extra latency problem by providing a known port latency factor to the synthesizer. We used post-synthesis simulation latency values which are longer and realistic for performance calculations.

3.1.1.3 Functional and Hardware Verification

We automated test and verification by generating test vectors in MATLAB. We used these test vectors in C simulations, C-RTL co-simulations and hardware verification with PYNQ-Z2. We tried to cover edge cases by generating variety of test cases.

We used Jupyter notebooks to program the PS in PYNQ-Z2 with Python language. We configured the PL with generated bit files of our overlays. We used the PS to allocate memory for input and outputs of the building blocks, call our overlay in the PL, read the output and verify the accuracy of hardware behaviour.

3.1.2 2D Convolution Block

2D convolution is widely used in CNNs. Convolution layers highly rely on multiply-accumulate operations and are computationally intensive. In a typical CNN inference operation convolution layers can take more than 90% of run time [46]. Therefore we primarily focused on having an optimized 2D convolution building block.

We implemented the convolution module as generic as possible to enable flexible use. We made image size in both dimensions, kernel filter size in both dimensions, stride factor and number of input tensors and kernel filters generic. User can use arbitrary bit-width INT data types for input tensor, kernel filter and output feature map, independently. We also included some generic design parameters regarding buffering, tiling and pipelining methods which will be discussed together with the design specifications.

We assumed stride factor $S = 1$ and padding factor $P = K - 1$ with zero padding as we described our convolution algorithm for simplicity, unless otherwise noted. This setting generates the maximum sized output matrix without missing any calculations as described by eq. (2.2).

3.1.2.1 Unrolling and Tiling for Computational Units

Convolution (eq. (2.1)) behaviorally can be described as multiple nested loops where each element of the input tensor is multiplied by each element of the kernel filter and the result is added to the corresponding element of the output array. Execution of these loops can be parallelized, tiled and pipelined, memory management can be optimized and external data interface can be configured for higher performance and lower resource utilization as described in Section 2.3.1.

We tiled the input tensor in order to divide the task into smaller parts which can be pipelined. Input tensor sizes can be very high which makes it unreasonable to move the whole input tensor into the PL memory at once before processing starts. Tiling enables pipelining of input memory access, processing steps inside the PL and output memory access. We processed one tile of the input tensor in each iteration of the pipeline. As shown in Fig. 3.2, we started processing the input tensor from the top-left corner, continued towards right until the row of tiles (i.e. yellow, green and blue tiles) has been completed, then started processing the next row of tiles from left to right. For each tile, we calculated the convolution of the tile and kernel filter with $S = 1$ stride and $P = K - 1$ zero padding, which corresponds to an output matrix which is bigger than the tile itself by eq. (2.2). Outputs of a tile convolution are visually represented in Fig. 3.3. Elements of the resulting matrix at the upper-left side (yellow cells in Fig. 3.3) which has the same size as the convolved input tensor tile, together with the values carried from previous tile operations, are used to finalize computation of some elements in the output feature map which will be buffered to be sent to the PS since they are finalized. Elements of the resulting matrix at the upper-right side (green cells in Fig. 3.3) which overlap with the next tile must be stored and transferred to the next tile convolution since they will be added to the results of the next tile convolution. Elements at the bottom of the

resulting matrix (blue cells in Fig. 3.3) also must be stored since they overlap with the next row of tiles and their results must be added to calculate corresponding final output feature map elements.

Although every input tensor tile can be convolved with the kernel filter independently, outputs of neighboring tile convolution operations have to be used together to calculate final outputs as described above. Such data dependencies has to be considered while forming the pipeline stages. We chose tile size so that processing one tile consumes reasonable amount of hardware when it is fully unrolled. Hence we built the pipeline in a way that at any moment exactly one tile is being convolved with the kernel filter. Since tiles are being processed serially, overlapping outputs of one tile can be carried to the next tile operation to be summed up with its outputs

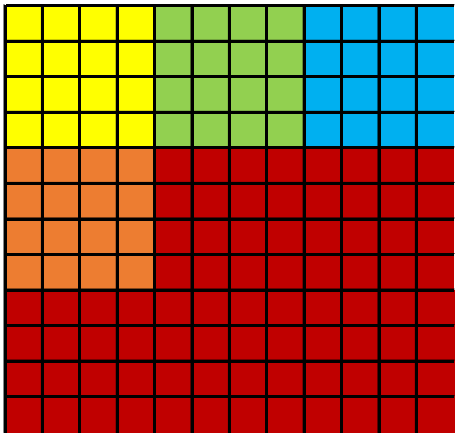

Figure 3.2: Order of input tensor tiles being processed for 4×4 tiles. Red background represents 2D input tensor. Yellow, green, blue and orange tiles are processed in this order and then the processing continues in a similar manner

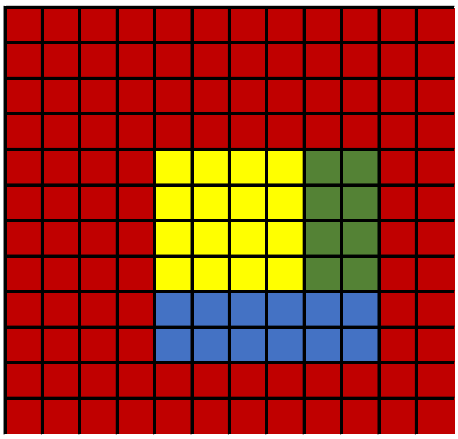

Figure 3.3: Representation of tile convolution where a 4×4 tile of 2D input tensor is convolved with a 3×3 kernel filter

before the next tile starts being processed. For every tile operation, firstly necessary outputs from the previous tile operations are fetched and written into an array. Secondly, the tile and the kernel filter are convolved. Thirdly, outputs are carried into a buffer, to be either sent to the PS since they are finalized or to be stored for the next tile operations. These three steps are not necessarily in serial order and are optimized and parallelized by the tool since the operations are unrolled. Described tile convolution operation defines one stage of the pipeline of our algorithm as we will discuss in detail later. All other pipeline stages serve this stage by forming the data path.

3.1.2.2 Memory Management and Data Path

Storage types used in the data path must be selected carefully to keep the resource utilization low while not limiting the throughput. Data can be stored in LUTRAM as distributed RAM or in BRAM as block RAM. BRAMs have limited number of input and output ports and can perform read/writes only as many as the number of ports they have per clock cycle. LUTRAMs provide higher data bandwidth but consume precious LUT slices which could have been used for computational operations. LUTRAMs can be used to store data which will be randomly accessed frequently in parallelized operations in order not to limit the throughput by memory bandwidth. Input and outputs of tile convolution is a good example of such operations. Since every element of the tile must be multiplied with every element of the kernel filter and results must be summed up sequentially, we allowed the tool to access any element of the tile and kernel filter any time by storing them in LUTRAMs. This flexibility might provide the tool the chance to perform further optimizations and parallelizations. Outputs of the tile convolution are also stored in LUTRAMs. Utilization of LUTRAMs sped up the process of moving outputs of current tile convolution to the next tile convolution stage as well since memory data shift can be performed in parallel. We used BRAMs for buffers which keep large amounts of data to save resources. The memory we used to keep the outputs for the next row of tiles is a good example for the use of BRAMs. Size of this memory is proportional to the size of the whole image which might not be manageable with LUTRAMs.

Data path includes instances which transfer data between LUTRAMs and BRAMs. Speed of the data transfer between LUTRAMs and BRAMs is limited by the speed of BRAMs due to their lower bandwidth. In order to perform the data transfers between LUTRAMs and BRAMs together with arithmetical operations on the data stored on LUTRAMs, we used double buffering on the data path as suggested in [23]. We also used buffering at AXI inputs and outputs in order to enable long burst operations to improve bandwidth efficiency, as suggested in [24].

3.1.2.3 Pipeline Design

Every input tensor consists of tiles. A set of consecutive tiles which sit on the same rows of the input tensor form a row of tiles. In this design we processed different rows of tiles sequentially. However, processing of a row of tiles is pipelined to improve throughput. Pipeline design with five stages is given in Fig. 3.4 where white boxes

are pipeline stages, yellow boxes are memory elements with indicated types, blue arrows represent pipeline order, black arrows represent data path and pipeline for one row of tiles is shown inside dashed lines. Three sets of inputs/outputs of the overall system are input tensor, output feature map and kernel filter. These inputs/outputs are read/written through master AXI ports. The kernel filter is typically much smaller than the input tensor and therefore is directly read completely and written into a LUTRAM at the beginning of all operations, hence kernel reading is not included inside the dashed lines in Fig. 3.4 as a part of the pipeline. Input tensor and output feature map interfaces have high data traffic and therefore are included in the pipeline. In order to enable burst operations at AXI ports, buffering stages (first and last stages) can buffer more than one tile. Number of tiles to be buffered is defined with a generic parameter. Middle stages are pipelined to perform one tile per interval. Stages with higher initiation intervals are in return executed less frequently but perform more operations per interval which makes it possible to match data flow of different stages. Buffered input tensor tiles are divided into individual tiles and written into LUTRAMs in Initializations stage. Calculation results which were calculated during the execution of the previous row of tiles and stored in Row Memory and correspond to the output feature map cells that will be calculated in this stage are also written into a LUTRAM at this stage. Tile Convolution stage works with only LUTRAMs to maximize parallelization and throughput. It takes the current input tensor to be processed, data from the calculations from the previous row of tiles, the kernel filter and outputs from the calculation of the previous tile which contribute to the output feature map cells whose calculation will be executed at this stage. Calculated results are stored in another LUTRAM. At Storing Results stage, some outputs are written into Row Memory to be used in the execution of the next row of tiles, remaining finalized outputs are written into Output FM (feature map) buffer to be sent to the PS. Buffering Outputs stage sends buffered finalized outputs to the PS with burst operations. Every memory unit in the pipeline uses double buffering.

In order to form such pipeline Vitis HLS code must be formed in correct structure with correct pragmas. Once it is commanded to pipeline a loop, Vitis HLS unrolls every sub-loop and functions called inside the loop and builds a pipeline. However, in order to force tool to build the pipeline in a specified structure, more commands and coding methodology are required. If further instructions are not provided and Vitis HLS fails to build an optimized pipeline which is more likely for complicated designs, unrolled loops and functions might result with too high hardware utilization. We implemented the pipeline which processes a row of tiles as a loop. In order to define pipeline stages, we built every stage as a separate function and forced tool to build them as separate entities rather than merging them with main function for optimizations. We called different functions with data dependencies in that order and at consecutive iterations of the loop instead of in the same iteration, in order to be able to parallelize every function call in each iteration. We explicitly marked false dependencies within the same loop iteration between different pipeline stages in order to make tool not behave unnecessarily conservatively. We marked pipeline stages with high initiation intervals and provided their execution frequency

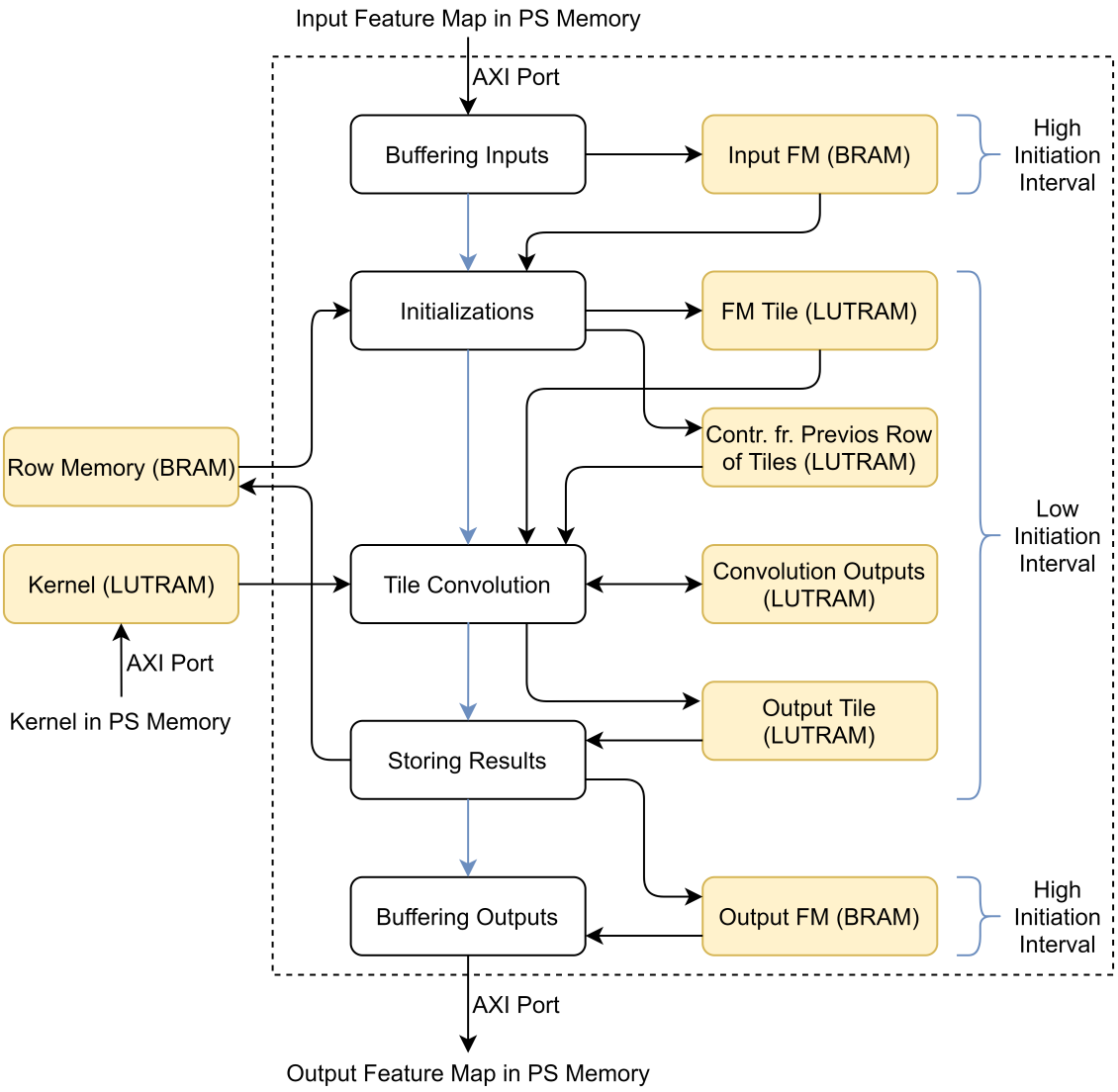

Figure 3.4: Pipeline diagram of a 2D convolution block which performs convolution for one row of tiles

to the tool, which enabled tool to execute these stages in multiple pipeline intervals with less hardware. Initiation interval of the pipeline is equal to the initiation interval of the stages with low initiation interval whereas stages with high initiation intervals take a few pipeline initiation intervals to execute. We allocated resource limitations to different functions and also limited the number of entities that can be synthesized inside hardware in case a function is called multiple times in order to improve hardware reuse.

3.1.2.4 Interface Design

We aimed to balance memory bandwidth and operating speed to improve resource efficiency. Bigger tile size and lower initiation interval increase number of parallel operations to be performed and hardware utilization. Increased hardware utilization per building block might not improve throughput if improved latency does not compensate for the increased hardware utilization per building block and reduced total number of blocks implemented on the device due to reasons such as hardware underutilization. Data flow speed should be adjusted in a way that it catches up with the processing speed. However increasing data flow speed at different parts of the design requires methods such as partitioning BRAMs and widening AXI ports which can also cost hardware.

We used master AXI4 ports at inputs and outputs to obtain high throughput. Using master AXI4 enables the overlay to reach any memory location in the PS with burst operations. Since the overlay is the AXI master and in charge of initiating memory operations, the PS which is running the main Python code needs other means to call the overlay to perform the required function. Such control is provided by an AXI-Lite port where the PS is the master and overlay is the slave. The PS uses this port to provide memory addresses of the input tensor, kernel filter and output feature map in the PS memory and also initiate an operation by calling the overlay.

We used bit-widening in AXI ports in order to increase memory throughput. AXI is a parallel bus protocol and supports busses with different widths. Using wider busses is a simple way of improving memory bandwidth by a factor. This method is especially very handy for low-precision data types which inherently underutilize the bus. Moreover, more computations per second can be performed on hardware for low-precision data types, which requires higher memory bandwidth. BRAMs on the data path also must be partitioned in order to catch up with the speed of the AXI ports. Memory partitioning increases BRAM and LUT utilization due to increased number of multiplexers. Therefore excessive port-widening should be avoided. We merged three AXI ports (input tensor, output feature map and kernel filter) into one port in order to reduce hardware utilization of AXI ports. Merging these ports did not reduce performance since the kernel filter is typically very small and the input tensors and output feature maps are read/written in different directions on the bidirectional bus.

3.1.2.5 Stride

We added stride functionality to observe its effect on power consumption. Due to stride some outputs did not need to be calculated which reduced the computational complexity and CTC ratio. In order to prevent hardware underutilization due to lower CTC ratio and reduce latency by taking advantage of low computational complexity, we increased the memory bandwidth. We used extra AXI ports when we could not increase bit-widening factor due to hardware limitations. We changed tiling structure so that tile size is equal to the stride factor, and one output is calculated in every initiation interval. Due to high amount of structural changes, we developed a separate version of the code to perform stride with generic stride factors.

3.1.2.6 Data Types and Overflow

We made data types of the input tensor, kernel filter and output feature map adjustable. We used INT data types with arbitrary bit-width which can be processed by LUTs in FPGAs. We made a version of the code where all input tensor, kernel filter and output feature map shared the same data type and overflow during operations were allowed. We also developed another version of the code where data types for the input tensor, kernel filter and output feature map could be independently selected and overflow was prevented by taking bit-growth into account in intermediate stages. We used quantization by truncation to obtain required output feature map precision whenever necessary. Low-precision convolution which prevents overflow is functionally equivalent to the accelerated GPU operation and therefore can be used to make fair comparisons.

3.1.2.7 Functional Verification and Assumptions

We designed the system to work with edge cases and verified functionality with tests. Design works even if image size is not a multiple of tile size. It also works when total number of tiles to be processed is not a multiple of tile buffering size. However, even though functionally it is not important, having proper tile and buffer sizes can improve resource and energy efficiency.

3.1.3 Fully Connected Block

Fully connected layers process high amount of data and therefore might require higher memory and memory bandwidth depending on the implementation. We implemented and investigated fully connected building block due to its common use in CNNs and tendency to be bounded by bandwidth rather than computational resources as opposed to the convolution block.

We made number of input and output artificial neurons in the fully connected layer generic. Integer data types for input neurons, weights, biases and output neurons can have arbitrary bit-width and be selected independently. In all fully connected blocks we prevented overflow by using higher bit-width intermediate data types.

In this implementation we only included multiply and accumulate operations in the fully connected layer and excluded activation functions. Activation function can be one of the many types some of which do not work well with integer data types, also applying activation to limited number of output neurons typically is not a high workload for the host machine. Therefore we left activation function task to the host platform which is using FPGA as an accelerator. Examples of excluding activation functions from FPGA implementation can be found in the literature. Qiu et al. [24] implemented some non-linear activation functions within FPGA fabric but performed Softmax on CPU due to the high design cost for the little performance improvement FPGA would bring. Implementing activation functions whose output is feeding another layer implemented on FPGA might reduce data traffic between the host platform and the FPGA, however might also bring limitations to the complexity of the activation function. Suda et al. [47] implemented ReLU on FPGA due to its simplicity as opposed to exponent functions, Nguyen et al. [48] preferred using leaky ReLU. Although it could be possible for us to implement a relatively simpler function such as ReLU on hardware, we did not include activation functions in the fully connected blocks to keep them more generic and simpler.

As described in eq. (2.3), in order to calculate the value of an output neuron in a fully connected layer all input neurons are multiplied with a weight unique to the corresponding input-output neuron pair and results are summed up together with a bias factor. Since weights are unique to the input-output neuron pairs and biases are unique to output neurons, data reuse is only possible for input neuron values. Due to limited data reuse possibility we increased memory bandwidth utilization with bit-widening and occasionally used multiple master AXI ports. In order to make the design compatible with the fully connected layers with high number of input neurons on FPGA platforms with limited resources, we avoided storing all input neuron values in BRAMs. Such approach would have maximized the input neuron data reuse but also increased the resource utilization. Instead, we increased data reuse by calculating more than one output simultaneously by reusing the same input neuron data transferred.

Our fully connected block calculates different outputs sequentially. However, calculation of each output neuron value is pipelined as shown in Fig. 3.5 where white boxes are pipeline stages, yellow boxes are memory elements with indicated types, blue arrows represent pipeline order, black arrows represent data path and pipeline is shown inside dashed lines. Reading input neuron values and weights from AXI ports is a part of the pipeline. These values are stored in separate BRAMs. Since input neuron values and weights are not accessed multiple times during calculations due to low data reuse, using BRAMs instead of LUTRAMs does not reduce performance unlike the convolution layer. However these BRAMs are partitioned enough to allow multiple data access operations in each clock cycle which permits transfer of multiple data per clock cycle in AXI busses with bit-widening. Buffering stage has higher initiation interval to allow longer burst operations but processes more data per interval to keep up with calculations stage with lower initiation interval. Calculations stage uses input neuron and weight data provided by buffering stage to calculate output neuron values. Calculations and buffering happen simultane-

ously with double buffering. Calculations stage uses Temp Output memory to store temporary outputs. Temp Output memory is initialized with biases before pipeline starts functioning. Calculations stage updates Temp Output values by summing its values up with new summands calculated from incoming weight and input neuron values. Pipeline calculates one or a few output values throughout its operations.

All bias values are read and stored before the start of the calculations. Similarly, all outputs are stored and sent through AXI ports after all of them are calculated. Transferring all biases and outputs together enables use of bit-widening and long AXI bursts and reduces latency. We assumed that number of output neurons is not too high which limits the BRAM size required to store outputs. This design decision can be revisited for the fully connected layers with high number of output neurons. In this case, output AXI access can be pipelined similar to the convolution layer at the expense of some increased latency.

Latency is limited by the memory bandwidth utilization. We experimented with different amount of bandwidth utilized to obtain different fully connected implementations that answer different needs. In case a CNN designer does not want to allocate high amount of bandwidth for one fully connected layer, they can use a low-bandwidth fully connected layer with higher latency. We implemented 2 different fully connected layers named fast and slow. Fast fully connected layer uses 4 AXI ports and calculates 3 output values simultaneously in the same pipeline which increases data reuse and reduces latency. Slow fully connected layer uses 2 AXI ports and calculates 1 output in one pipeline which reduces hardware resources and memory bandwidth utilized.

We tested functionality of our fully connected blocks for different input/output neuron numbers by comparing their outputs with fully connected layers implemented

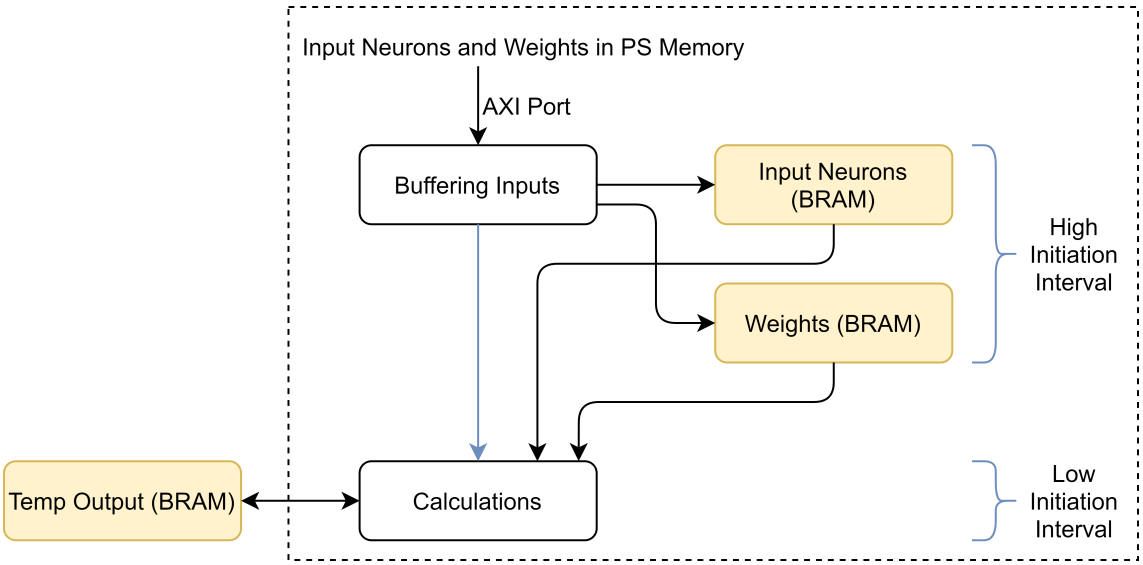

Figure 3.5: Pipeline diagram of a fully connected block which calculates one or a few output neuron values

on MATLAB with Deep Learning Toolbox [49]. Blocks are functionally verified for various use cases, however in order to maximize resource efficiency some conditions should be met. As an example, design is more resource efficient when number of input neurons is a multiple of burst length of the AXI.

3.1.4 Pooling Block

Pooling blocks are commonly used in CNNs to reduce computational complexity. We implemented max and average pooling blocks that can be used as accelerators.

Similar to the convolution block, we made many design parameters including input tensor size and pooling window size generic. Max and average pooling can be obtained from the same C code by changing a generic parameter pre-synthesis. Level of parallelization can be increased to reduce latency with some generic design parameters for different use cases. Stride can also be adjusted with generic parameters to some extend, however due to substantial structural changes in the case of increased number of AXI ports for improved bandwidth, we had to develop several separate versions of the code for stride. Arbitrary bit-width INT data types for input and output tensors can also be assigned with generic parameters independently.

Unlike our input-oriented convolution blocks where all calculations based on an input cell are performed simultaneously, we followed a more output-oriented approach in pooling blocks where all calculations that are required to obtain an output cell are performed simultaneously. In convolution blocks we took a tile of inputs, performed every calculation that needs to be performed with that tile of inputs and then either directly used the outcomes of the calculations or stored them for future use. In pooling blocks we stored or fetched inputs from global memory in a way that enables the design to calculate a tile of outputs in every clock cycle. Although both approaches require the same total number of operations, we found the output-oriented approach easier to manage and cleaner in the code especially for pooling with stride where certain outputs do not need to be calculated.

We reused significant amount of code from our convolution implementation and came up with a mostly similar design. Pooling blocks tile the output tensor. Different rows of output tensor tiles are calculated sequentially. However, calculation of each row of tiles is performed within a pipeline for improved throughput. An input cell is needed to calculate many neighboring output cells depending on the pooling window size. Therefore some input values have to be reused to calculate different output tensor tiles possibly on different rows of tiles in our output-oriented approach. We stored as much rows of the input tensor as needed in an on-chip BRAM called Input Memory to calculate next row of output tiles. A few rows of tiles are buffered in Input Memory before operations begin. Input Memory is updated constantly throughout the operations as new input cells are fetched from global memory as needed.

Pipeline that calculates a row of output tensor tiles is shown in Fig. 3.6. Data path of pooling resembles that of convolution. Input tensor values are fetched from

PS memory as relatively bigger chunks to enable bit-widening and burst operations for better memory bandwidth utilization. Small tiles of inputs stored in partitioned BRAMs called Input FM which stores recently fetched input cells and Input Memory which stores previously fetched rows of inputs are moved to a fully partitioned LUTRAM at Initializations stage for fast calculations. Initializations stage also prepares a LUTRAM to update the Input Memory in the next stage with new values. Tile Pooling stage performs tiling and calculates a tile of outputs. Initiation interval of this stage is one for highest hardware utilization ratio. Calculated outputs are stored in a LUTRAM which is eventually accumulated in a BRAM and transferred to PS Memory with long bursts.

Max pooling and average pooling differ only at calculation stage. We implemented max pooling and average pooling calculations as separate functions and synthesized either one of them depending on a generic parameter.

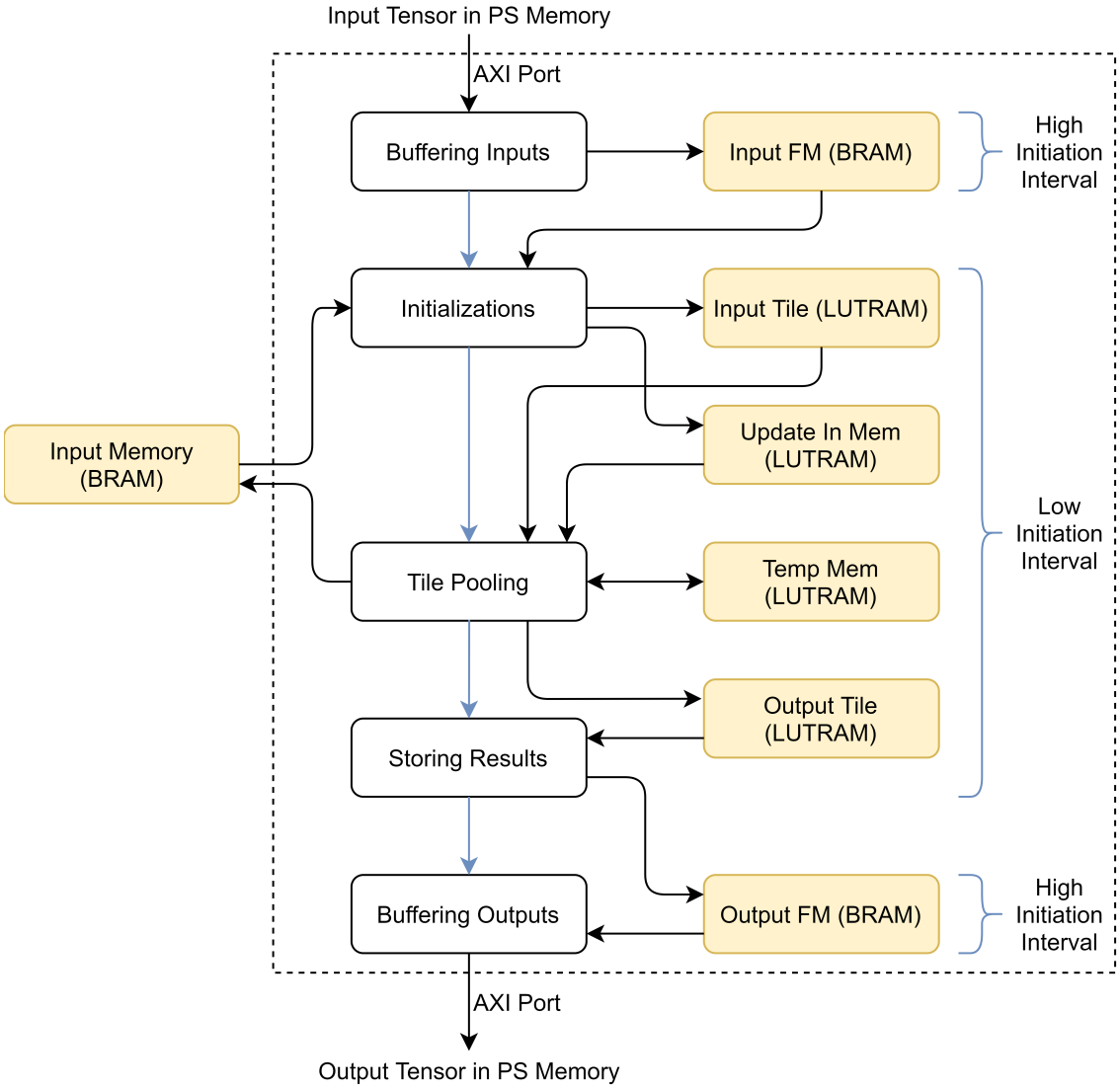

Figure 3.6: Pipeline diagram of a pooling block which generates one row of outputs

We used more number of AXI ports for designs with higher stride factors in order to take advantage of reduced CTC ratio. We prevented overflow in calculations. Designs were functionally verified for different generic parameters. However, performance and energy efficiency of the synthesis result highly depend on some design specifications due to certain assumptions made during optimizations.

3.2 GPU

GPU's parallel computing platform is found helpful to run massive inference operations with its optimized libraries. In this section, we will discuss the required toolchain and specific library functions used to implement the convolution, fully connected and pooling blocks. We will also discuss the power measurement technique we used to have meaningful comparisons with the FPGA implementations.

3.2.1 Overview of Tools

We used NVIDIA Jetson AGX Xavier Developer Kit to implement our CNN building blocks. The Jetson AGX Xavier module with Volta GPU architecture built in 12 nm FinFET process node supports low precisions like FP16 and INT8. The system-on-module (SoM) consists of 64 tensor cores and 2 DLAs in addition to 512 CUDA cores [50] [51]. We made use of NVIDIA SDK Manager to set up JetPack SDK 4.3 to flash Jetson Linux OS and install other SDK components like TensorRT 6.

Jetson AGX Xavier Developer Kit with operating system provides basic functionality to run C++ scripts, reading and writing files, graphical user interface and controls. Commands to read sensors, configure the hardware and run programs can be executed in Linux environment. Clock frequencies and power mode of the module can also be configured. In our tests, we used default 15 W power mode.

Fig. 3.7 describes the workflow which will be discussed in detail later in this chapter. Initially building block is designed in C++ using TensorRT API. After design is compiled, it can be run with or without DLA support for either FP16 or INT8 data types. Building block is run once and outputs are compared with the reference golden outputs generated in MATLAB. This cycle can be repeated a few times to debug and verify the code. Once the building block is verified, TensorRT building block is run in a loop for power and performance measurements. Latency and power measured by hardware and software tools provided by Jetson AGX Xavier can be used to calculate energy efficiency.

We designed our building blocks using TensorRT C++ API by defining the layers with API rather than importing them. To start with, we used MNIST API code [52] in TensorRT. We modified the code to obtain individual building blocks. We made modifications in the code to write the outputs during verification stage and to load the GPU and measure the latency during energy efficiency measurement stage.

We made data type and DLA utilization configurable in our TensorRT code. User can decide on these by defining parameters in the command used to call TensorRT

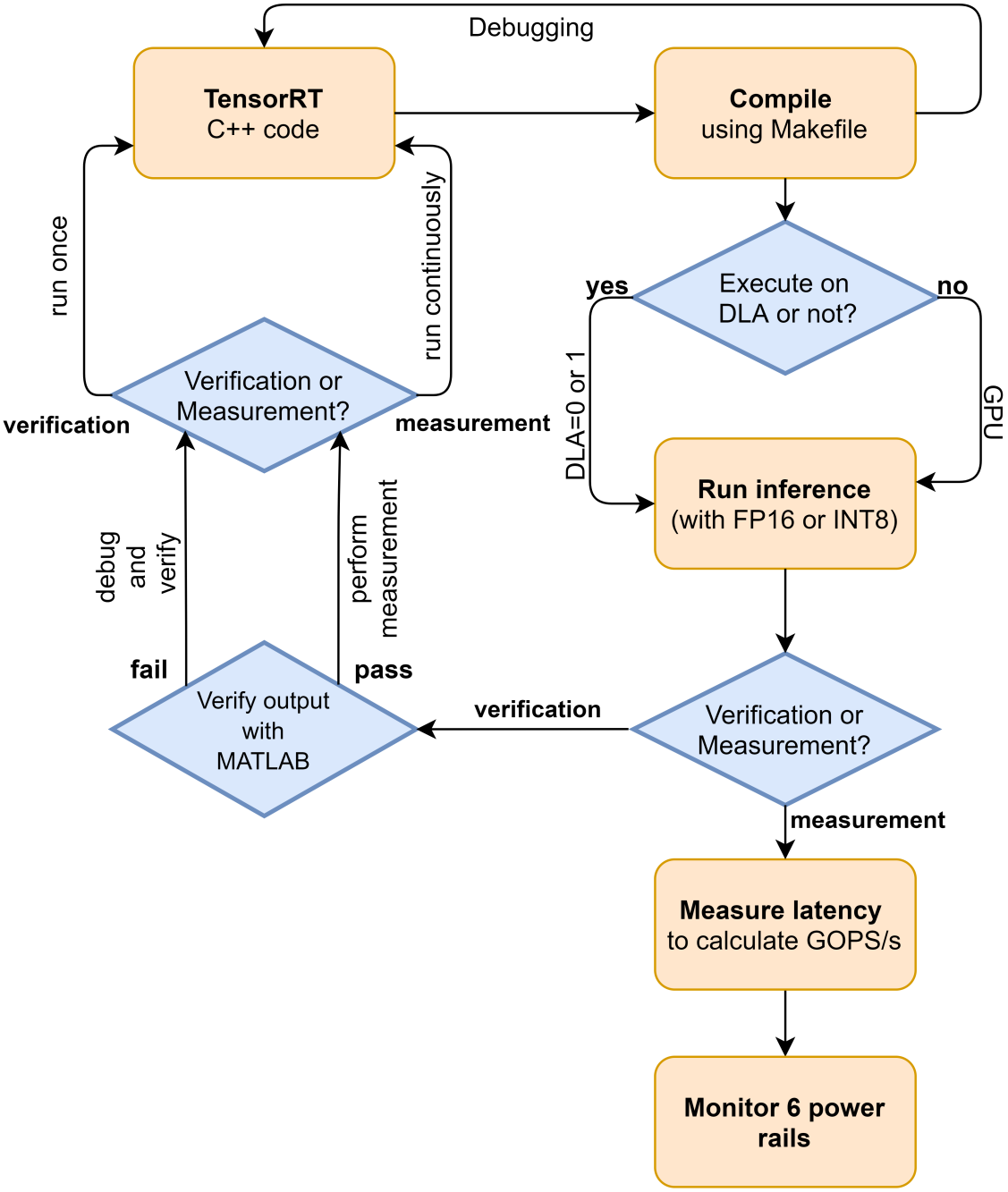

Figure 3.7: GPU workflow for implementation, verification and power measurement of the building blocks

API. Two DLA cores in Jetson AGX Xavier support FP16 and INT8 operations to run inferences which we used during our tests [53].

We used MATLAB to verify our building blocks. We implemented MATLAB scripts to generate input files for GPU tests. We replicated the functions performed in GPU, including data type conversions and rounding operations for supported data types. We compared GPU outputs with the outputs we generated in MATLAB using built-in functions and tools. We used self-checking testbenches to complete functional verification.

3.2.2 Power Measurement

We used built-in sensors in Jetson AGX Xavier to measure power. There are six power rails in Jetson AGX Xavier for all of which separate power sensors and sensor reading commands are provided [54]. It is possible to query the voltage on the rail, the current measured over a shunt resistor and the power calculated from the voltage and current. These respective six rails give the power consumed on GPU, CPU, SoC, DLA, DRAM and 5V system rail for I/O, separately. Total power consumption of the module is given by the sum of all these six layers. NVIDIA uses the total module power to benchmark Jetson AGX Xavier for CNN applications [55], therefore we as well used total power consumption in our analysis.

AGX Xavier includes INA3221 sensor with an analog-to-digital converter with 40 µV least significant bit step size over shunt resistance [56]. This sensitivity corresponds to $\frac{40 \times 10^{-6}}{5 \times 10^{-3}} = 8\,\text{mA}$ current resolution over 5 mΩ shunt resistor. This current resolution gives $19.4 \times 8 \times 10^{-3} = 155.2\,\text{mW}$ power resolution on 19.4 V GPU rails, and even better resolution on rails with lower voltage. We found this resolution acceptable due to its low error percentage over total measured power.

We used a similar approach to *Idle_power* and *Runtime_power* to model power measurement given in eq. (2.6). We measured total power and base power, and obtained operational power by subtracting the base power from the total power. When we measured the base power, we did not leave GPU completely idle. Instead, we ran a small application that kept the cores awake and prevented power gating with regular and sparse intervals. We made GPU to sleep between the calls of the small application. In order to verify the small application, we changed the workload and the calling frequency of the small application and then measured the power. We observed that power readings did not change and concluded that operations in the small application did not consume considerable amount of extra power. The purpose of this operation is to not include the idle power of the cores in the operational power. As an example, DLA cores consume significant idle power, however this power is not included in the power measurement when GPU is completely idle due to power gating of the cores. For a complete network implemented to be run on DLAs, DLAs cannot sleep due to continuous calls from the network, regardless of a call from one individual block. By preventing power gating on DLA cores, we obtained more accurate operational power resulting from individual building block calls which use DLAs.

3.2.3 Inference Modes

TensorRT engines mainly contribute towards inference implementations with FP16 and INT8 precisions along with the regular FP32 precision. TensorRT performs necessary data type conversions for improved low-precision inference with high throughput and energy efficiency [57].

TensorRT has a limitation that it cannot accept quantized weights and inputs but always accepts them in FP32 form only. We automated our weights file generation using MATLAB for individual building blocks for verification and measurement and kept the dynamic range limitations for FP16 and INT8 in mind to avoid saturation in computation results.

To implement the inference with FP16, we specified the precision for each layer. In the INT8 inference mode, we could consider two different settings, either to provide with dynamic range values for input and output tensors or to enable a calibrator method. We chose to perform the dynamic range mapping and avoided using a calibrator to have overall control over input data buffer. Further necessary settings required for INT8 implementation includes providing FP32 weights, dynamic range information, precision for each layer and also setting the respective output precision [32].

We made a notable observation for INT8 inference case that TensorRT engine performs floating point operations to convert FP32 inputs and weights to INT8 numbers to be processed within the specified dynamic ranges [58] [59]. Therefore internal floating point operations cannot be avoided, thus it is not purely INT8 and seems to be considered as mixed precision inference. During execution of INT8 implementation of an individual building block, converted FP32 weights and inputs undergo INT8 multiplications and get accumulated in INT32 data type to avoid overflow and thereby preventing the saturation of the computation results [60]. However while using FP32 and FP16 modes, there is no such requirement for an intermediate accumulation data type.

3.2.4 Individual Blocks and Loading

We implemented the building blocks such as convolution, fully connected and pooling by carefully removing other blocks from the official TensorRT MNIST API sample code. After performing functional verification of the individual blocks, we carried out our power measurements by running the inference execution part of code in a loop to avoid any interruptions during measurements.

Due to low-resolution of the sensors, we read only base power values when we did not load GPU much with math intensive operations. We loaded GPU to perform more operations per unit time to obtain higher power variations which could be noticeable even with the low-resolution current sensors we used. Meanwhile we utilized both of the DLA cores present in the Jetson AGX Xavier GPU for maximum resource utilization.

For the convolution block implementation, we loaded the GPU with more convolution kernel filters. We further implemented fully connected block in the same way but loaded them with more number of similar fully connected layers. We also implemented the pooling block by making use of input 3-D tensors which consists of many 2-D images meant to be processed with the same pooling window size.

We monitored the six power rails using software commands and then summed them up to obtain total module power consumed. We also took latency measurements for individual building blocks without loading.

4

Results and Discussion

We estimated, profiled or measured the performance and energy efficiency of CNN inference building blocks on FPGAs and GPUs. In this chapter we will discuss the results and also compare them with the state of the art.

We will share our insights on the effects of changing design parameters on energy efficiency and performance when we compare our designs among each other. We will focus on the outcomes when we focus on the comparison with the state of the art. Note that our main purpose with this thesis is to compare different design platforms with our building blocks. Therefore we will focus on comparability rather than superiority when we compare our building blocks with the state of the art. Energy efficiency and performance depend on the type of the network and many design parameters. Since various design parameters have been used in research papers and we only have building blocks rather than full networks, it is only possible to show comparability. Hence we focused on comparability rather than superiority when we compare our results with the state of the art. Once it is shown that our building blocks are comparable with the state of the art, we will be able to use them to make realistic comparisons of different design platforms for different design parameters.

4.1 FPGA

In this section we will compare some of our building blocks implemented on FPGA with the state of the art. We will also present and compare our designs among themselves for different specifications and design parameters and share our insights.

In power estimations, we included dynamic power estimations together with total power estimations. Since our building blocks utilize a small portion of hardware resources, ratio of static power to total power consumption is higher than usual. Using dynamic power values when we compare our designs with each other for different data types can provide more meaningful comparisons.

4.1.1 Comparison with the State of the Art

Table 4.1 shows resource and energy efficiency of some of our designs together with some FPGA-based CNN accelerators we have found in recent research papers. Due

to its considerable effect on performance, we grouped the results according to the data types used. As discussed in more detail in Section 4.1.2 and Section 4.1.3, design parameters and specifications affect the performance of our building blocks. Hence we used some average building blocks with typical features in these comparisons. The convolution and pooling blocks work with 1024×80 sized images, 5×5 kernel filters or pooling windows and 1×1 stride factor. The fully connected blocks use 256 input and 256 output artificial neurons. All of our blocks in Table 4.1 prevent overflow by considering bit-growth. VGG16-SVD [24] is implemented on ZYNQ-7045 which belongs to the same family as our hardware platform and therefore provides fair comparisons. Our convolution block with DSP support provides 1.17 times better power and 1.39 times better resource efficiency whereas our convolution block without DSP support is 1.18 times less energy efficient compared to the convolution layers in VGG16-SVD. It is known that fully connected layers are more bandwidth intensive and perform less calculations and therefore are typically less efficient in GOPS/kLUT and GOPS/W metrics [24] which explains the 1.37 times lower resource efficiency of the overall VGG16-SVD compared to the convolution layers of VGG16-SVD and also overall lower performance of our fully connected layers compared to our convolution layers. Since convolution operations typically make up most of the operations in a CNN, overall performance of CNNs are typically closer to performance of their convolution layers [24] [47] and therefore we will compare our convolution layers with overall networks when there is no layer by layer performance data available. Our 16-bit convolution layers have better energy efficiency than YOLOv2 [61], STD_CNN [62] and DS_CNN [62] and either higher or close resource efficiency depending on DSP utilization. Using variety of precisions within the same CNN is a common practice to maximize efficiency. VGG16 [47] with varying bit-width has lower energy and resource efficiency compared to our DSP supported convolution block with higher (always 16-bit) precision. Tiny YOLO-v2 [48] and Sim YOLO-v2 [48] on a relatively advanced board with DSP support and varying bit-width have up to 16.36 times better resource efficiency than our 2-bit and 8-bit convolution blocks, higher energy efficiency than our 8-bit convolution block and either higher or close energy efficiency compared to our 2-bit convolution block. BinaryNet [63] has 2.65 times better resource efficiency but 1.24 times worse energy efficiency compared to our 2-bit convolution block. Our blocks are generally better at higher bit-widths but are worse than but not too distant from the state of the art for lower precisions. It is important to note that our blocks are single building blocks rather than complete networks, which makes them vulnerable to higher static power in GOPS/W comparisons and energy efficiency calculated from dynamic power could be considered as a better metric. On the other hand single building blocks have the advantage of not suffering from network level inefficiencies due to idle hardware caused by inter-layer data dependencies. Furthermore full CNNs typically include activation functions with relatively higher resource and energy cost. Depending on the network type and various design parameters such as bit-width, our designs showed both higher and lower energy and resource efficiency compared to the state of the art. Overall, we can conclude that energy and resource efficiency of our building blocks are comparable with the state of the art.

Table 4.1: Resource and energy efficiency of our blocks in comparison with other FPGA implementations from research papers

Bit-width	Network	Platform	GOPS/kLUT	GOPS/W	GOPS/W (dyn.)
16-bit	**Conv2d, no DSP (ours)**	ZYNQ-7020	0.30	16.57	19.89
	Conv2d, with DSP (ours)	ZYNQ-7020	1.01	27.11	36.07
	FC, fast (ours)	ZYNQ-7020	0.23	6.32	10.11
	VGG16-SVD (conv) [24]	ZYNQ-7045	0.86	19.50	-
	VGG16-SVD (overall) [24]	ZYNQ-7045	0.63	14.22	-
	YOLOv2 [61]	ZYNQ Ultrascale+	0.37	8.64	-
	STD_CNN [62]	Arria10 GX 1150	0.20	10.07	-
	DS_CNN [62]	Arria10 GX 1150	0.23	11.61	-
8∼16-bit	VGG16 [47]	Stratix-V	0.98	6.17	-
8-bit	**Conv2d, no DSP (ours)**	ZYNQ-7020	0.74	32.80	47.75
	Conv2d, with DSP (ours)	ZYNQ-7020	1.23	31.88	45.84
	FC, fast (ours)	ZYNQ-7020	0.30	7.62	13.49
	Max pooling (ours)	ZYNQ-7020	0.67	19.12	35.63
1∼16-bit	Tiny YOLO-v2 [48]	Virtex7	5.40	53.29	-
	Sim YOLO-v2 [48]	Virtex7	12.11	102.62	-
2-bit	**Conv2d, no DSP (ours)**	ZYNQ-7020	1.67	54.61	111.90
	FC, fast (ours)	ZYNQ-7020	0.47	9.35	19.88
1∼2-bit	BinaryNet [63]	ZYNQ-7020	4.43	44.2	-

4.1.2 2D Convolution Block

Due to flexibility of FPGA we had the chance to modify the generic parameters of our convolution block which enabled us to observe the effect of different design choices on performance and energy efficiency. We tried different kernel filter sizes, stride factors, hardware mappings and data types with and without overflow in arithmetic operations. Due to vast range of the design space, every time we changed only one parameter and fixed the other parameters.

In order to limit simulation time, we used relatively lower number of rows in images. We extrapolated performance results for bigger images with higher number of rows from the data we obtained for images with lower number of rows. Since different rows of tiles are being processed in series, increasing number of rows does not increase number of LUTs utilized. Power utilization was also assumed to be constant since different row operations are in series. Before and after an image is processed completely, some memory operations related to the Row Memory which are not included in the pipelines add some extra latency which does not scale with the number of rows. Hence we used

$$GOPS_{ext} = GOPS_{sim} \cdot \frac{lat_{sim}}{rows_{sim}} \cdot \frac{rows_{ext}}{\frac{lat_{sim}-lat_{init}}{rows_{sim}} \cdot rows_{ext} + lat_{init}} \tag{4.1}$$

to extrapolate GOPS where $GOPS_{ext}$ and $GOPS_{sim}$ are the extrapolated and simulated GOPS, lat_{sim} is the simulated latency, lat_{init} is the total latency of the operations before and after the image is processed which does not scale with the number of rows, $rows_{sim}$ is the number of rows of the simulated image and $rows_{ext}$ is the number of rows of the extrapolated image. Performance increases with the number of rows. In the graphs included in this section, measured results belong to 1024×80 input tensors whereas extrapolated results belong to 1024×1024 input tensors. In this section, all kernel filter sizes are 5×5, stride factors are 1×1 and data types are 8-bit integer unless otherwise is explicitly noted.

Fig. 4.1, Fig. 4.2 and Fig. 4.3 show resource and energy efficiency of the 2D convolution block for different data types. Table 4.2 shows hardware utilization and latency values. In this comparison we used integer data types with different bit-widths for the input tensor, kernel filter and output feature map where all three always used the same data type and overflow was allowed. In Fig. 4.2 effect of the static power is prominent since we are testing a single building block, hence Fig. 4.3 which compares energy efficiency calculated from the dynamic power can give a better comparison between our designs with different bit-widths. Results clearly show that using lower precision improves GOPS/kLUT and GOPS/W. According to the power estimation reports we obtained, lowering precision reduces power consumed on the slice logic and registers significantly with lighter calculations. However power consumed on the I/O (AXI ports), BRAMs and clocks did not scale-down with lowered bit-width as much as power consumed on the slice logic. Therefore even for INT1 we observed significant amount of dynamic power consumption. Utilizing DSP resources

improved GOPS/kLUT by letting LUTs offload some arithmetic operations whereas reduced GOPS/W due to increased power consumption on the signals and DSPs.

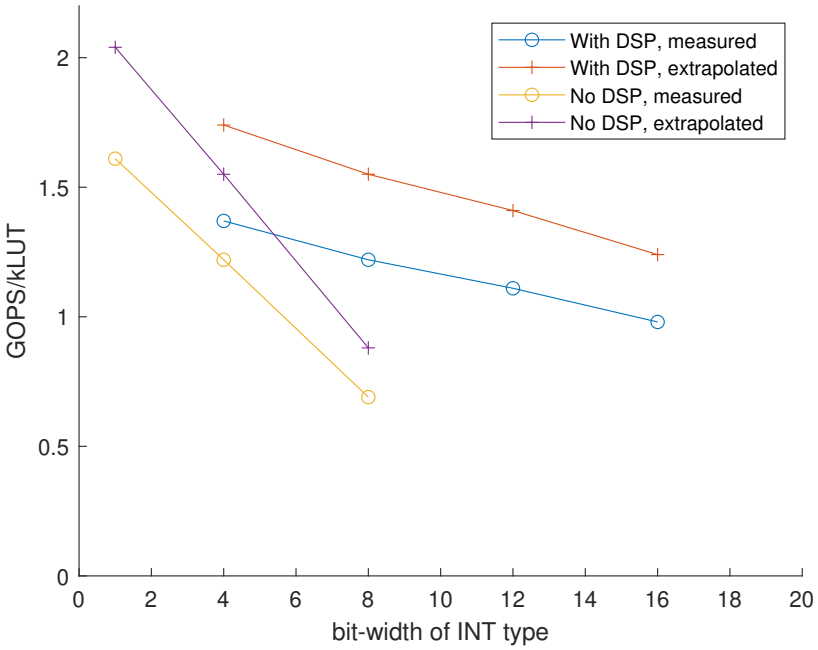

Figure 4.1: GOPS/kLUT results where all inputs and outputs share the same data type and overflow is allowed

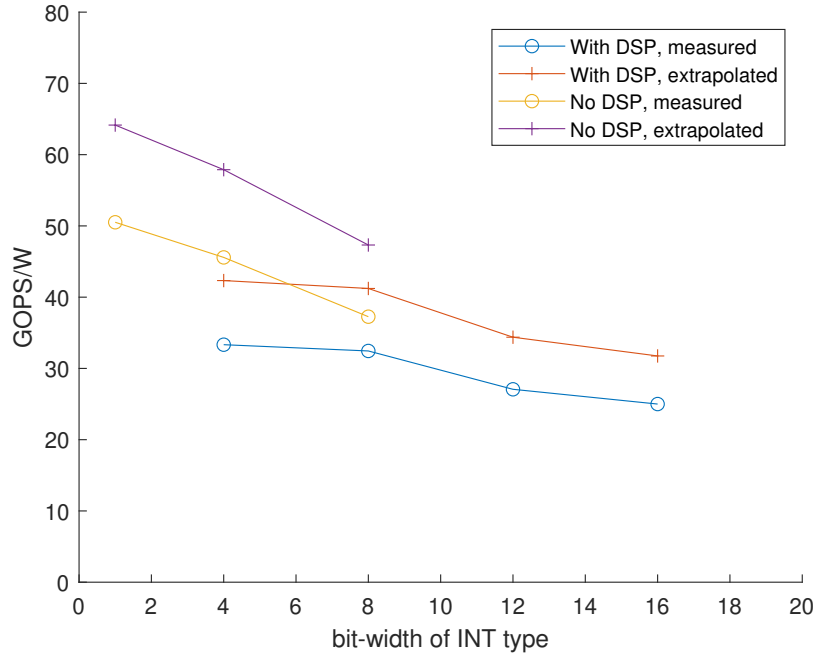

Figure 4.2: GOPS per total power (W) results where all inputs and outputs share the same data type and overflow is allowed

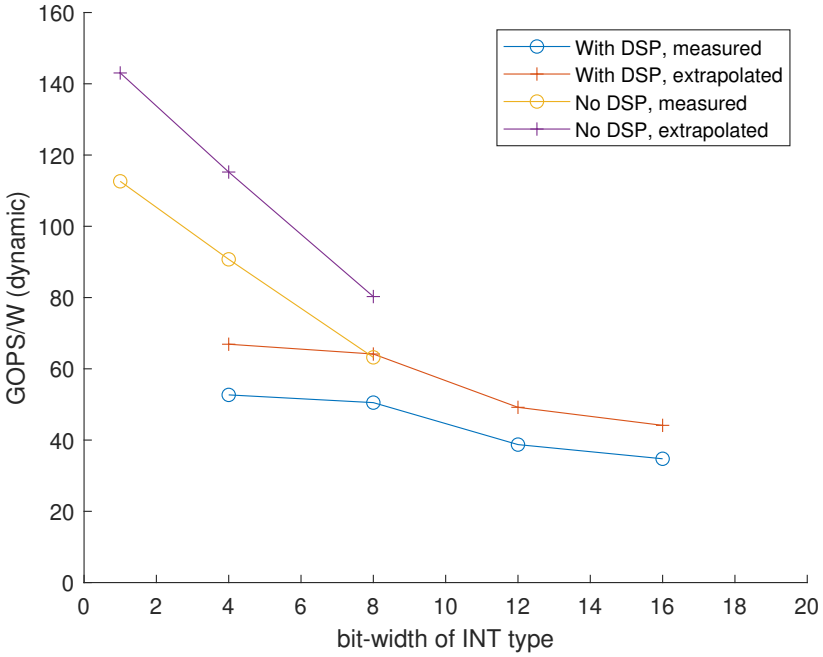

Figure 4.3: GOPS per dynamic power (W) results where all inputs and outputs share the same data type and overflow is allowed

Table 4.2: Resource utilization and latency results where all inputs and outputs share the same data type and overflow is allowed

		1-bit	4-bit	8-bit	12-bit	16-bit
No DSP	LUTs	6097	8034	14286		
	DSPs	-	-	-		
With DSP	LUTs		7154	8059	8847	10034
	DSPs		200	200	200	200
Latency(ms) 1024×80 FM		0.418	0.418	0.418	0.418	0.418
Latency(ms) 1024×1024 FM		∼4.21	∼4.21	∼4.21	∼4.21	∼4.21

Fig. 4.4, Fig. 4.5 and Fig. 4.6 show resource and energy efficiency of the 2D convolution block when data types of the input tensor, kernel filter and output feature map are independently changed. Table 4.3 shows hardware utilization and latency values. In this comparison we changed the bit-width of one of the three input/outputs (i.e. input tensor, kernel filter and output feature map) and kept the bit-width of the other two as 8-bit. We prevented overflow by taking bit-growth into account, therefore these results can be fairly compared with GPU which also prevents overflow. In these designs we did not use the DSP resources of FPGA. Reducing the input tensor or kernel filter bit-width improves power and resource efficiency and reduces resource utilization. Reducing the input tensor precision is slightly more effective than reducing the kernel filter precision since input tensors are typically much larger and reducing their bit-width reduces power consumption of signals even more. On the other hand, reducing bit-width of the output feature map does not affect energy efficiency or resource utilization considerably since quantization happens at the very end of the process and calculations are still carried out with high precision numbers. However adjustable output feature map precision is still an important feature as it defines the input precision of the next layer in CNN. Preventing overflow by utilizing high bit-width intermediate data types inside the building block increased power and resource consumption. 8-bit design which prevents overflow utilizes 15256 LUTs (Table 4.3) and have 47.75 GOPS/W dynamic energy efficiency (Fig. 4.6) whereas equivalent 8-bit design which does not prevent overflow utilizes 14286 LUTs (Table 4.2) and have 115.23 GOPS/W dynamic energy efficiency (Fig. 4.3).

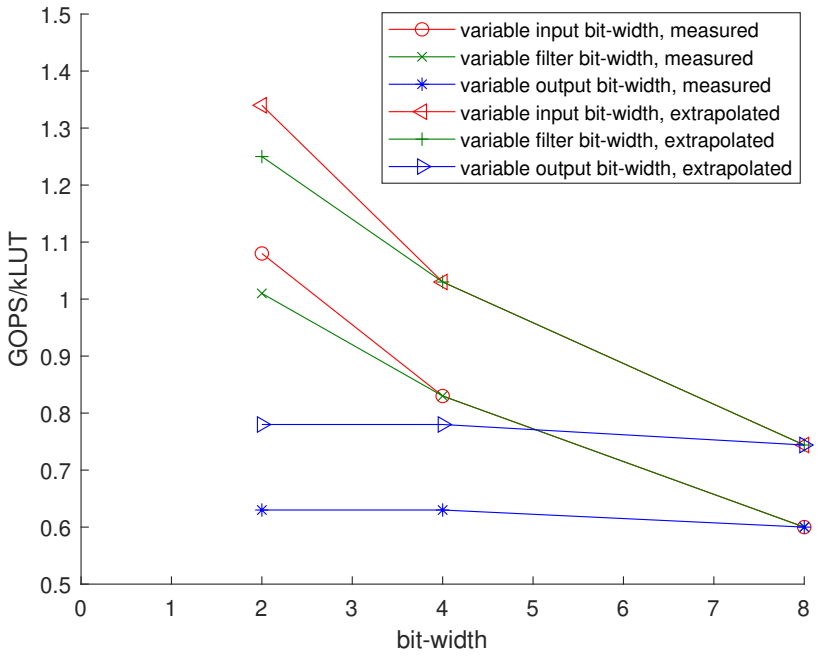

Figure 4.4: GOPS/kLUT results for variable input FM bit-width where kernel filter and output FM are 8-bit (red), variable kernel filter bit-width where input FM and output FM are 8-bit (green) and variable output FM bit-width where input FM and kernel filter are 8-bit (blue)

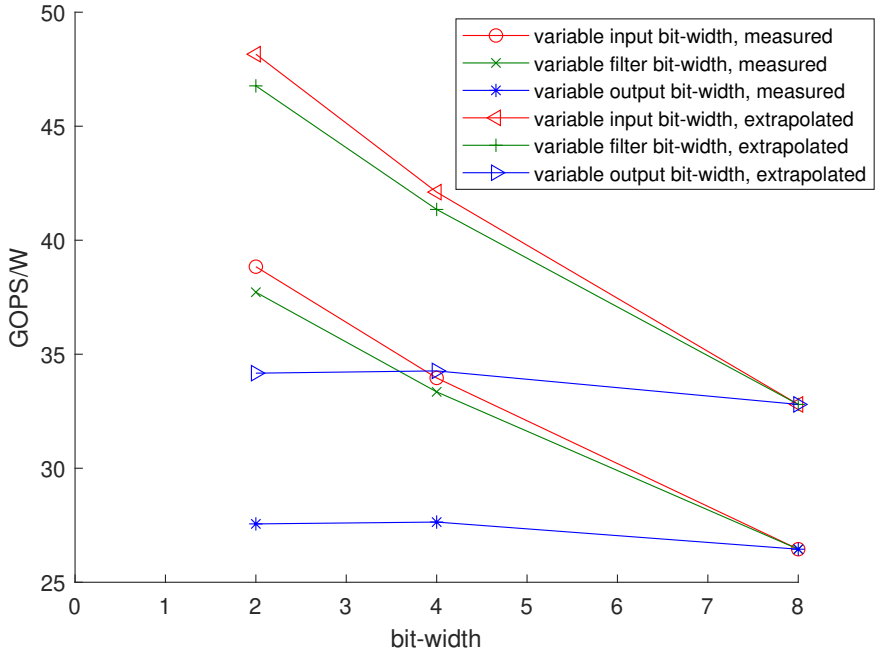

Figure 4.5: GOPS per total power (W) results for variable input FM bit-width where kernel filter and output FM are 8-bit (red), variable kernel filter bit-width where input FM and output FM are 8-bit (green) and variable output FM bit-width where input FM and kernel filter are 8-bit (blue)

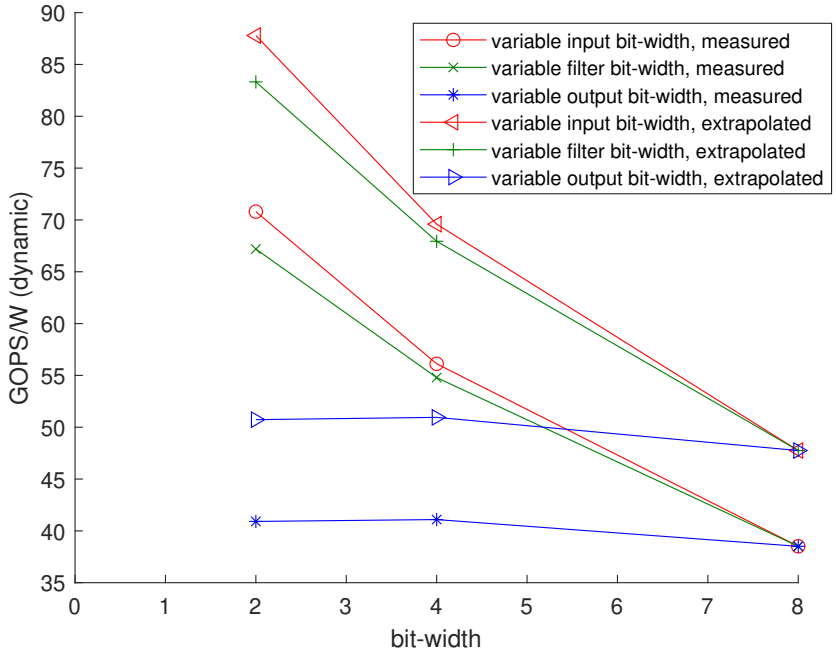

Figure 4.6: GOPS per dynamic power (W) results for variable input FM bit-width where kernel filter and output FM are 8-bit (red), variable kernel filter bit-width where input FM and output FM are 8-bit (green) and variable output FM bit-width where input FM and kernel filter are 8-bit (blue)

Fig. 4.7, Fig. 4.8 and Fig. 4.9 show resource and energy efficiency of the 2D convolution block for different kernel sizes of 3×3, 5×5 and 7×7. Table 4.4 shows hardware utilization and latency values. We prevented overflow in these designs. Increasing the kernel size increased LUT and DSP utilization since our design performs operations for the whole kernel filter in parallel. Increasing the kernel filter size increased also dynamic power consumption of the design. However, as shown in the figures, GOPS/kLUT and GOPS/W results are better for bigger kernel filter sizes due to increased number of operations and therefore increased GOPS. Increased kernel filter size increases the number of computations per transferred input tensor cell and therefore CTC ratio as well, which improves performance from a computational efficiency point of view. However, from high-level network point of view, reducing kernel size of a convolution layer is still beneficial to the overall system energy efficiency given that other parameters are the same since smaller kernels consume less power. LUTs are more energy efficient compared to DSPs for small kernel filter sizes (3×3 and 5×5) but for 7×7 kernel filter utilizing DSPs improved energy efficiency. In our experiments kernel filter size also changed latency since we used $P = K - 1$ padding factor which increases the size of the output feature map by eq. (2.2) as kernel filter size increases. Increased output feature map size increases latency in our design since the amount of data transferred to memory is almost proportional to the latency. In a typical CNN application where padding size is adjusted to equate the output feature map size to the input tensor size, latency would remain same in our design.

Fig. 4.10, Fig. 4.11 and Fig. 4.12 show resource and energy efficiency of the 2D convolution block for different stride factors of 1×1, 2×2, 2×4, 4×4 and 4×8. Table 4.5 shows hardware utilization and latency values. We prevented overflow in these designs. Increasing the stride factor reduced GOPS/kLUT and GOPS/W results due to reduced number of operations performed and therefore reduced GOPS. Stride reduces data reuse of the input tensor cells and therefore reduces CTC ratio which lowers GOPS/kLUT and GOPS/W parameters. However, due to reduced computational intensity, latency and resource utilization can be reduced with in-

Table 4.3: Resource utilization and latency results for variable input FM bit-width where kernel filter and output FM are 8-bit, variable kernel filter bit-width where input FM and output FM are 8-bit and variable output FM bit-width where input FM and kernel filter are 8-bit

		2-bit	4-bit	8-bit
Variable Input FM	LUTs	8536	11085	15256
Variable Kernel Filter	LUTs	9097	11089	15256
Variable Output FM	LUTs	14701	14716	15256
Latency(ms) 1024×80 FM		0.445	0.445	0.445
Latency(ms) 1024×1024 FM		∼4.59	∼4.59	∼4.59

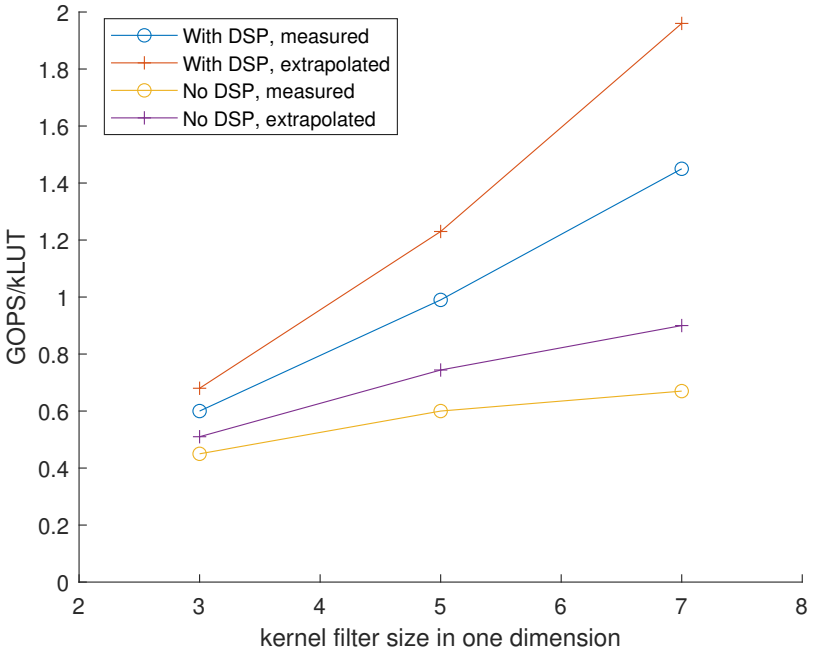

Figure 4.7: GOPS/kLUT results for varying kernel filter sizes

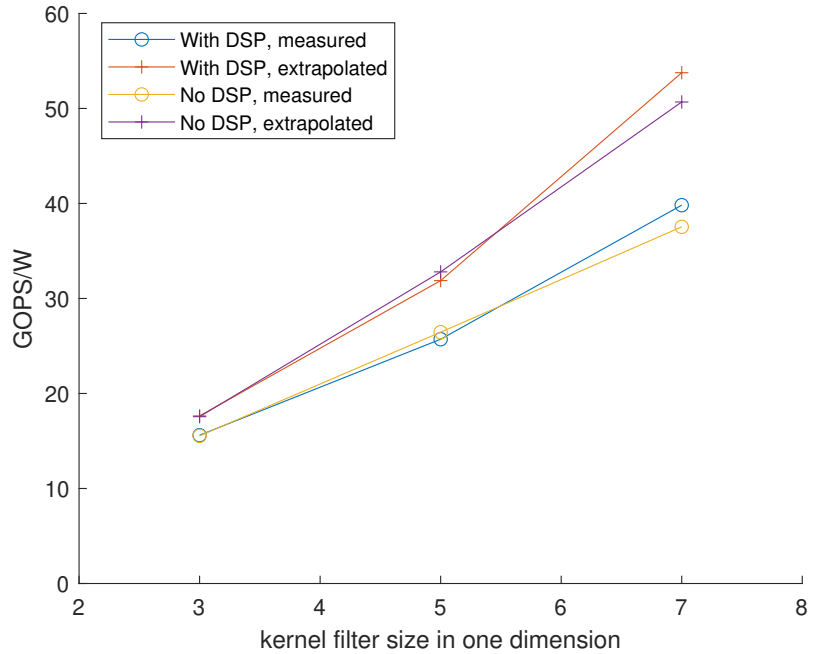

Figure 4.8: GOPS per total power (W) results for varying kernel filter sizes

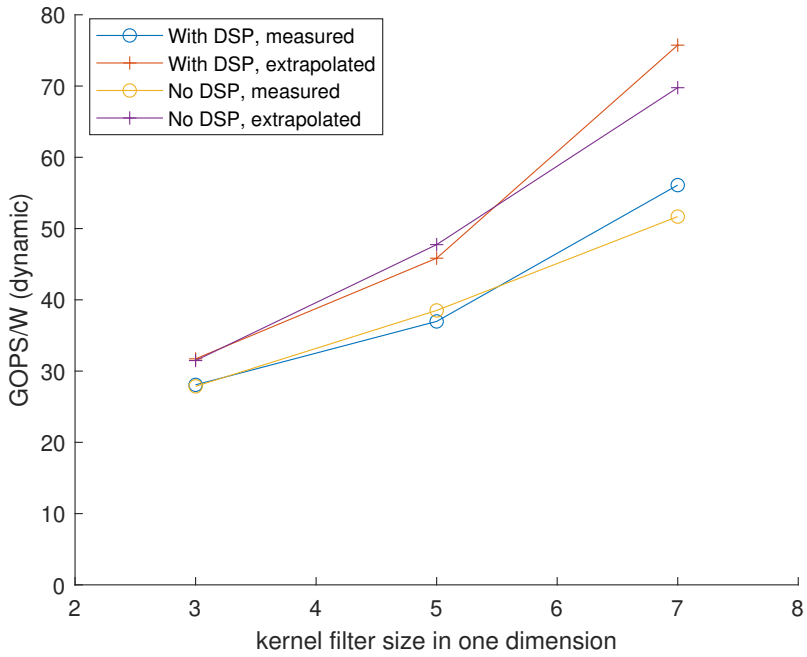

Figure 4.9: GOPS per dynamic power (W) results for varying kernel filter sizes

Table 4.4: Resource utilization and latency results for varying kernel filter sizes

		3×3	5×5	7×7
No DSP	LUTs	8378	15256	22609
	DSPs	-	-	-
With DSP	LUTs	6285	9276	10387
	DSPs	36	100	196
Latency(ms) 1024×80 FM		0.392	0.445	0.532
Latency(ms) 1024×1024 FM		$\sim$4.44	$\sim$4.59	$\sim$5.04

creased stride factor. As an example, as shown in Table 4.5, changing stride factor from 1×1 to 2×2 and from 2×4 to 4×4 improved both latency ad resource utilization. When we changed stride factor from 2×2 to 2×4 and from 4×4 to 4×8 we added extra AXI busses which increased resource utilization but improved latency significantly. Stride can be exploited to reduce latency, resource utilization or a combination of both for different design cases and needs.

Parameters that have been discussed so far come from the specifications of the building block. Design parameters of the convolution which do not affect the behaviour can also be important for the energy efficiency and performance of the convolution. As an example, Table 4.6 compares two functionally identical convolution blocks implemented with different tile sizes. These blocks work with 8-bit data types and allow overflow. Design with 4×2 tile has an initiation interval of 2 in the pipeline where design with 2×2 has an initiation interval of 1 which increases hardware reuse, reduces LUT and DSP utilization and improves GOPS/kLUT significantly.

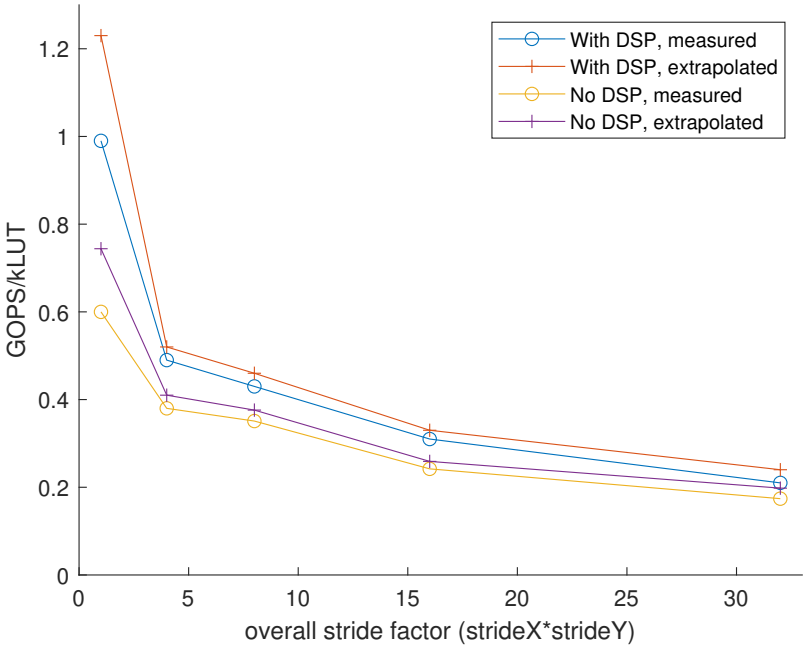

Figure 4.10: GOPS/kLUT results for varying stride factors

Table 4.5: Resource utilization and latency results for varying stride factors

		1×1	2×2	2×4	4×4	4×8
No DSP	LUTs	15256	7486	8240	8133	10202
	DSPs	-	-	-	-	-
With DSP	LUTs	9276	5862	6686	6360	8439
	DSPs	100	25	25	25	25
Latency(ms) 1024×80 FM		0.445	0.356	0.177	0.130	0.072
Latency(ms) 1024×1024 FM		∼4.59	∼4.26	∼2.12	∼1.56	∼0.81

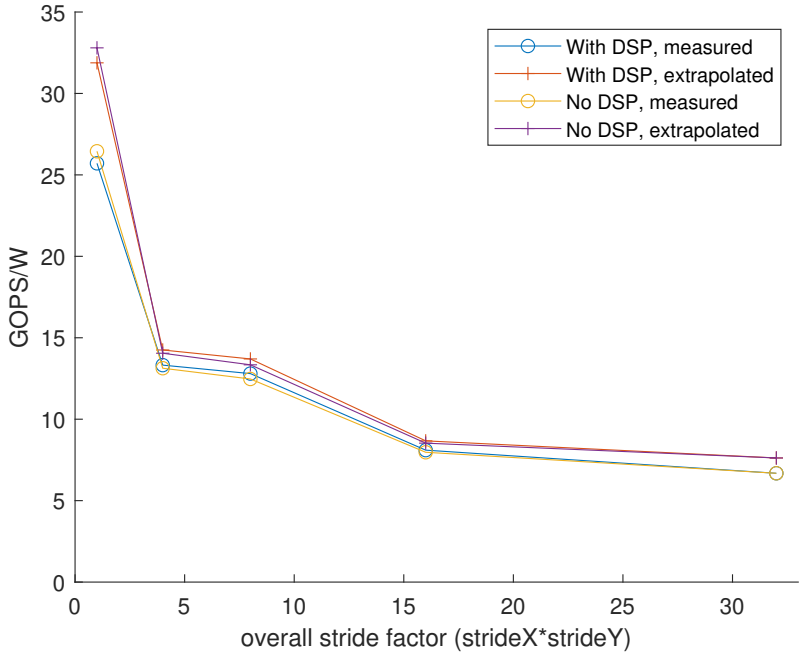

Figure 4.11: GOPS per total power (W) results for varying stride factors

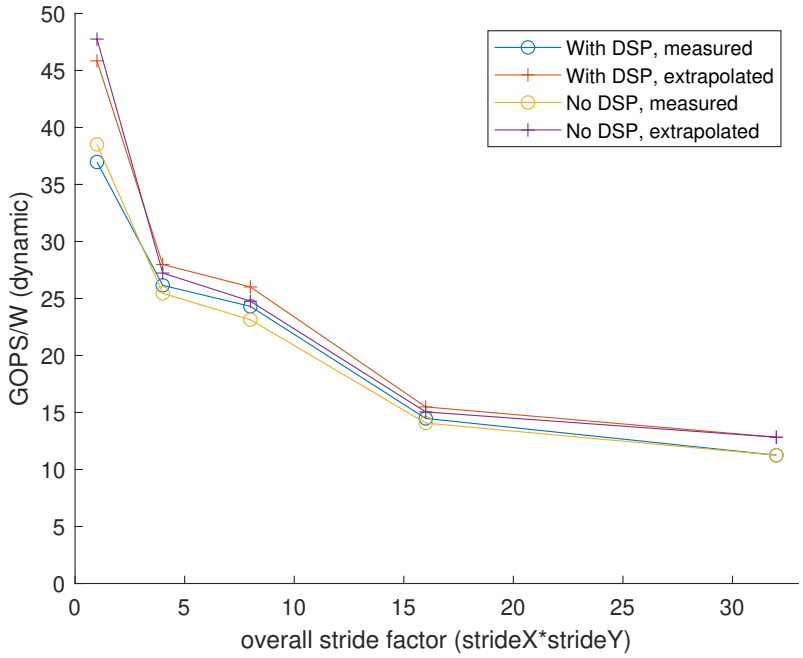

Figure 4.12: GOPS per dynamic power (W) results for varying stride factors

GOPS/W performances are only slightly different in both designs since they carry out the same calculations in similar amount of time.

Number of columns in an image also affects performance of the convolution block. In order to reduce the extra latency caused by AXI address exchange in our pipelines, we used longer bursts. Operations on a single row of tiles are pipelined in our designs. When a set of data is serially processed by a pipeline, underutilization of the pipeline stages at the beginning and the end of the pipeline for the beginning and the end of the processed data set causes inefficiencies. Together with longer bursts which contribute to the initiation interval of pipelines, these inefficiencies are more prominent for smaller images with lower number of columns.

4.1.3 Fully Connected Block

Fully connected layers perform less number of operations and utilize memory bandwidth more and therefore are less resource and energy efficient compared to convolution layers when efficiency is measured with GOPS related metrics. Difference between convolution and fully connected in terms of computational power and memory bandwidth demand was also visible in power estimation reports we obtained. Unlike the convolution blocks, the fully connected blocks consumed less power on slice logic but more power on BRAMs which made up up to half of dynamic power in some cases. We did not utilize DSP resources in our fully connected blocks.

Fig. 4.13, Fig. 4.14 and Fig. 4.15 show resource and energy efficiency of both fast and slow implementations of the fully connected block as described in Section 3.1.3 when data types of input, weight/bias and output are independently changed. Table 4.7 shows hardware utilization and latency values. In this comparison we changed the bit-width of one of the three input/outputs (i.e. input, weight/bias and output)

Table 4.6: Comparison of different tile sizes

		2×2	4×2
No DSP	LUTs	10225	14286
	DSPs	-	-
	GOPS/kLUT	0.90	0.69
	GOPS/W	36.53	37.26
	GOPS/W (dyn.)	63.48	63.22
With DSP	LUTs	7016	8059
	DSPs	100	200
	GOPS/kLUT	1.31	1.22
	GOPS/W	30.08	32.45
	GOPS/W (dyn.)	46.49	50.51
Latency(ms) 1024×80 FM		0.445	0.418
Latency(ms) 1024×1024 FM		∼4.59	∼4.21

and kept the bit-width of the other two as 8-bit. Although weight and bias bit-widths can also be different in our building blocks, we assumed that they are the same in our analysis since bit-width of the bias has very little effect on resource and power utilization. We prevented overflow by taking bit-growth into account. We used 256 input artificial neurons and 256 output artificial neurons. In these designs we did not use the DSP resources of FPGA. Reducing input or weight/bias bit-width improves power and resource efficiency and reduces resource utilization. Especially for fast design, reducing weight/bias bit-width increases efficiency more than reducing input bit-width since there are more number of AXI ports that carry weights than inputs and inputs are reused about thrice more. Reducing output bit-width does not affect calculations much and therefore does not change power and resource efficiency considerably. Fast design is always more resource and energy efficient than slow design due to increased data reuse and lower latency, but slow design utilizes less hardware due to hardware reuse.

Table 4.8 shows hardware utilization, resource efficiency, energy efficiency and latency of the 8-bit fast fully connected block for 256×256, 1024×36 and 8192×12 input/output artificial neuron numbers. Overall latency of the building block tends to increase as total number of operations by eq. (2.8) increases. However, input/output distribution of the neurons is also important. As seen in Table 4.8, 8192×12 has lower latency than 256×256 despite having higher number of operations. Blocks with higher number of input neurons are more resource efficient and also tend to

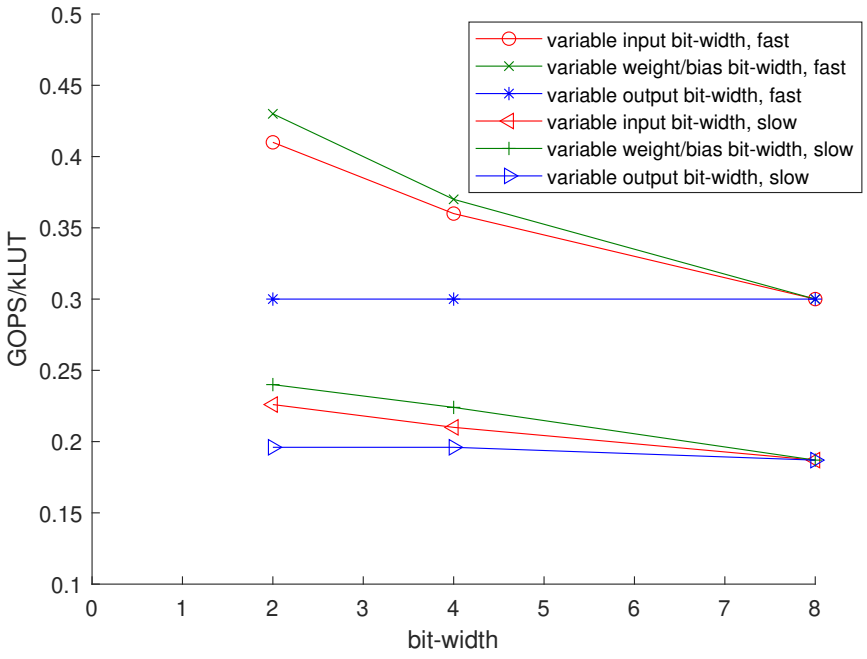

Figure 4.13: GOPS/kLUT results of separate FC implementations (fast and slow) for variable input bit-width where weight/bias and output are 8-bit (red), variable weight/bias bit-width where input and output are 8-bit (green) and variable output bit-width where input and weight/bias are 8-bit (blue)

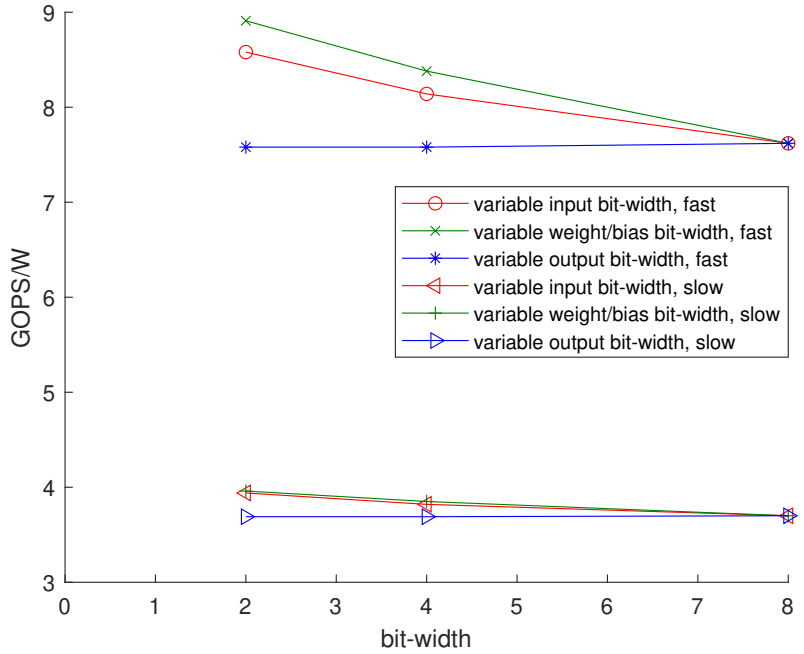

Figure 4.14: GOPS per total power (W) results of separate FC implementations (fast and slow) for variable input bit-width where weight/bias and output are 8-bit (red), variable weight/bias bit-width where input and output are 8-bit (green) and variable output bit-width where input and weight/bias are 8-bit (blue)

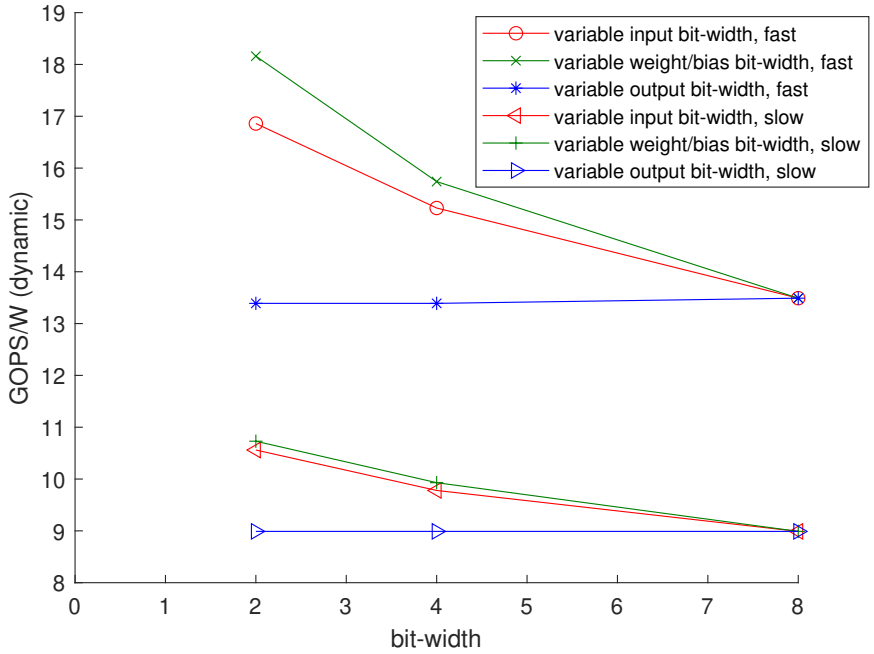

Figure 4.15: GOPS per dynamic power (W) results of separate FC implementations (fast and slow) for variable input bit-width where weight/bias and output are 8-bit (red), variable weight/bias bit-width where input and output are 8-bit (green) and variable output bit-width where input and weight/bias are 8-bit (blue)

have higher energy efficiency since all input neurons are processed in a pipeline for every output neuron and having more input neurons improves pipeline efficiency with longer uninterrupted pipelined operations. 8192×12 has higher LUT count than 256×256 despite being more efficient in general due to increased bit-growth as a consequence of the higher number of inputs.

4.1.4 Pooling Block

We analyzed efficiency and performance of max and average pooling for different data types, pooling window sizes and stride factors. In the graphs included in this section results belong to 1024×80 input tensors. Unlike convolution results, we did not use extrapolated input tensor sizes since we used pooling without padding in analysis which provides a performance almost independent from the number of rows of the input tensor. Pooling blocks were implemented without DSP resources. In this section, all pooling window sizes are 5×5, stride factors are 1×1 and data types are 8-bit integer unless otherwise is explicitly noted.

Power reports showed that pooling blocks typically consumed less power on slice logic and signals but more power on BRAMs compared to convolution blocks. Low computational intensity of pooling reduced the power consumed on slice logic. Due to low computational intensity we tried to parallelize more operations which required more data to be kept on-chip which explains the increased BRAM power utilization.

Table 4.7: Resource utilization and latency results of separate FC implementations (fast and slow) for variable input bit-width where weight/bias and output are 8-bit, variable weight/bias bit-width where input and output are 8-bit and variable output bit-width where input and weight/bias are 8-bit

			2-bit	4-bit	8-bit
Variable Input	LUTs	Fast	4625	5224	6297
		Slow	2948	3172	3557
Variable Weight/Bias	LUTs	Fast	4378	5067	6297
		Slow	2739	2966	3557
Variable Output	LUTs	Fast	6281	6281	6297
		Slow	3393	3393	3557
	Latency(ms)	Fast	0.069	0.069	0.069
		Slow	0.197	0.197	0.197

Table 4.8: Hardware utilization and resource efficiency, energy efficiency and latency results of the fast FC implementation for different input-output artificial neuron counts

In/Out Size	GOPS/ kLUT	GOPS/W	GOPS/W (dyn.)	LUTs	Latency (ms)
256/256	0.30	7.62	13.49	6297	0.069
1024/36	0.32	9.10	15.13	7623	0.030
8192/12	0.50	10.79	16.29	6907	0.056

Fig. 4.16, Fig. 4.17 and Fig. 4.18 show resource and energy efficiency of the max and average pooling blocks for different integer bit-widths that were used for both input and output tensors. Table 4.9 shows hardware utilization and latency values. Similar to convolution, results clearly indicate reduced resource and energy efficiency and increased hardware utilization with increased bit-width. Max and average pooling provided very similar resource and energy efficiency. Average pooling tends to utilize slightly more hardware possibly due to the difference in the type of operations performed such as inclusion of division operation in average pooling. Resource utilization difference between different pooling types is more prominent for higher bit-widths.

Fig. 4.19, Fig. 4.20 and Fig. 4.21 show resource and energy efficiency of the max and average pooling blocks for different pooling window sizes of 3×3, 5×5, 7×7 and 9×9. Table 4.10 shows hardware utilization and latency values. We used 8-bit input and output tensors for all designs. In the graphs we also included results for 8-bit convolution without DSP support for the same input tensor size for different kernel filter sizes as a reference. Similar to convolution, increasing pooling

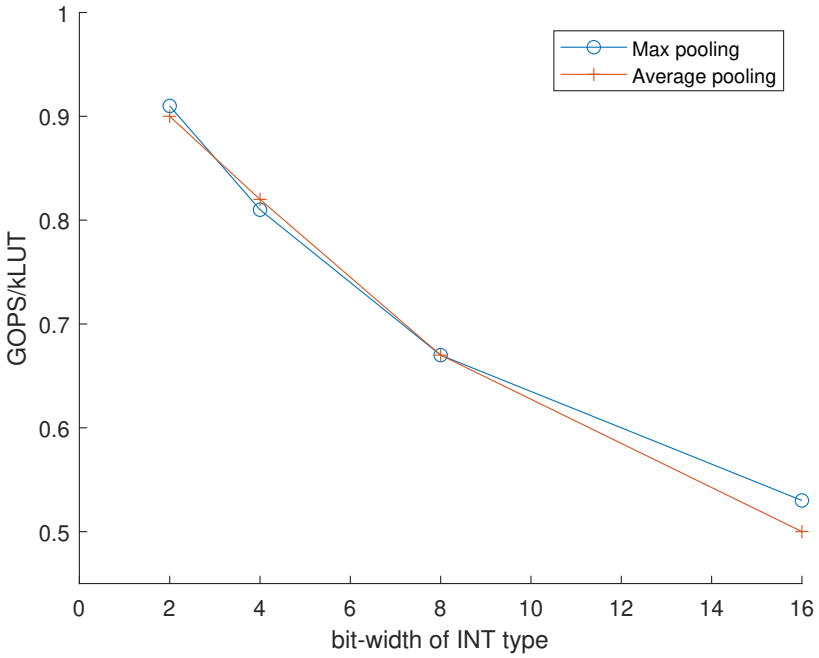

Figure 4.16: GOPS/kLUT results for variable input and output tensor bit-width

Table 4.9: Resource utilization and latency results for variable input and output tensor bit-width

		2-bit	4-bit	8-bit	16-bit
LUTs	Max	4855	5427	6582	8174
	Avr.	5127	5643	6874	9029
Latency(ms)		0.449	0.449	0.449	0.453

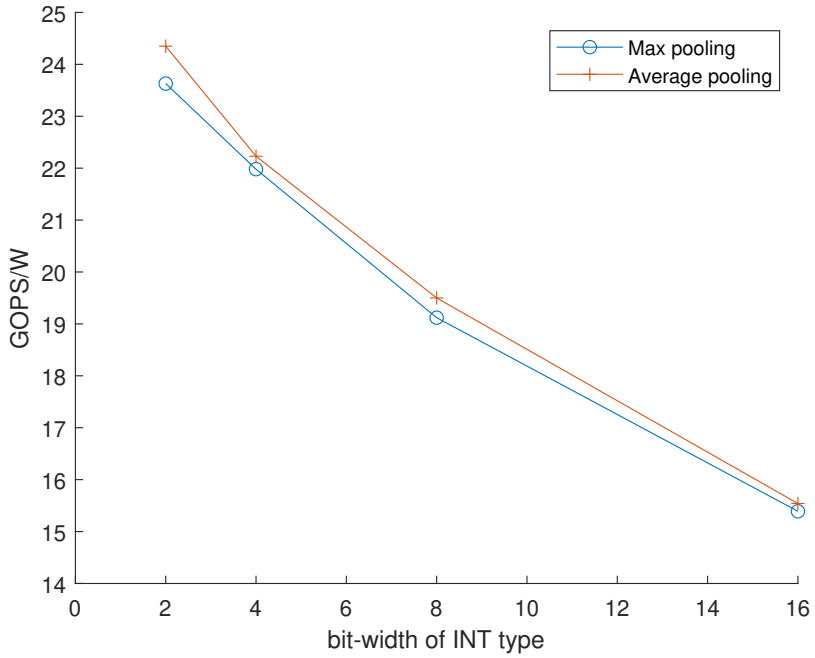

Figure 4.17: GOPS per total power (W) results for variable input and output tensor bit-width

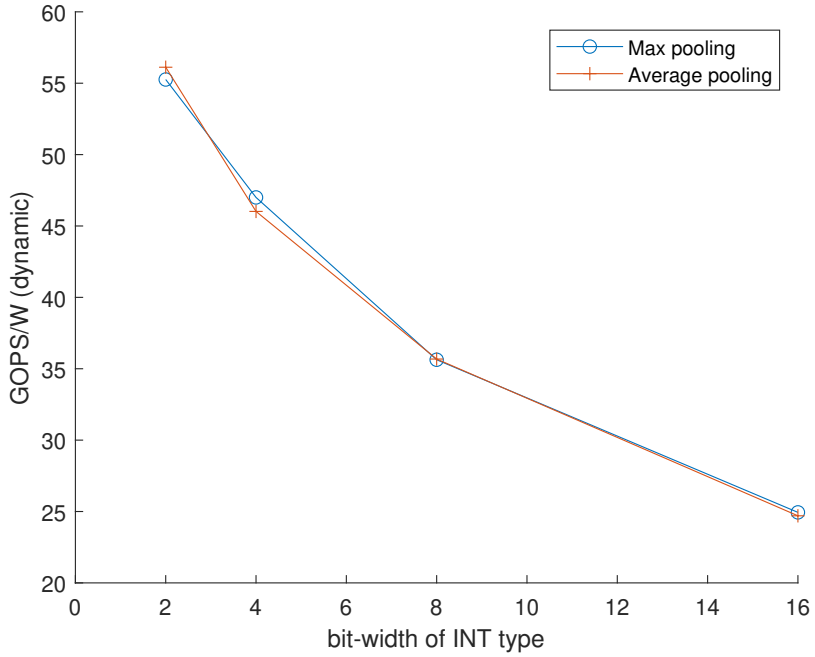

Figure 4.18: GOPS per dynamic power (W) results for variable input and output tensor bit-width

window size increased resource and energy efficiency due to increased number of operations. For some cases when we increased the pooling window size, we needed to reduce parallelization and number of outputs cells calculated per clock cycle in order to keep hardware utilization at reasonable levels which in return increased latency. The pooling blocks showed in general better resource efficiency but worse energy efficiency than the convolution blocks. Differences in the types of operations as described in Section 2.5 might explain such differences in performance. Pooling blocks highly rely on comparisons and additions which consume less hardware resources than multiplications which are used in convolutions. Reduced number of operations might have reduced the energy efficiency of the pooling blocks compared to the convolution.

Fig. 4.22, Fig. 4.23 and Fig. 4.24 show resource and energy efficiency of the max and average pooling blocks for different stride factors of 1×1, 2×1, 2×2, 2×4 and 4×4. Table 4.11 shows hardware utilization and latency values. Similar to convolution, increasing stride factor reduced resource and energy efficiency in general due to reduced number of operations. However, by improving bandwidth utilization, it

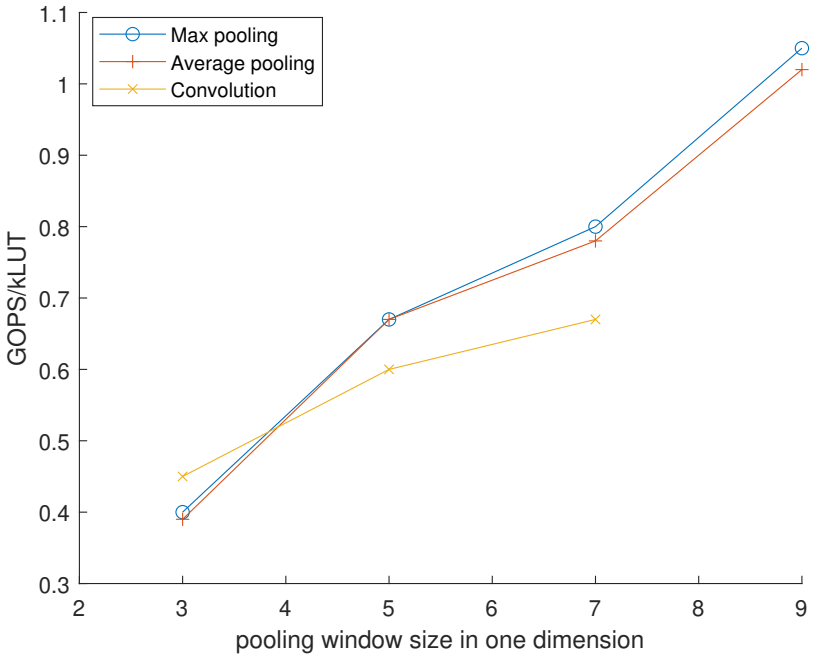

Figure 4.19: GOPS/kLUT results for varying pooling window sizes

Table 4.10: Resource utilization and latency results for varying pooling window sizes

		3×3	5×5	7×7	9×9
LUTs	Max	5875	6582	6156	7346
	Avr.	6634	6874	6302	7598
Latency(ms)		0.282	0.449	0.797	0.853

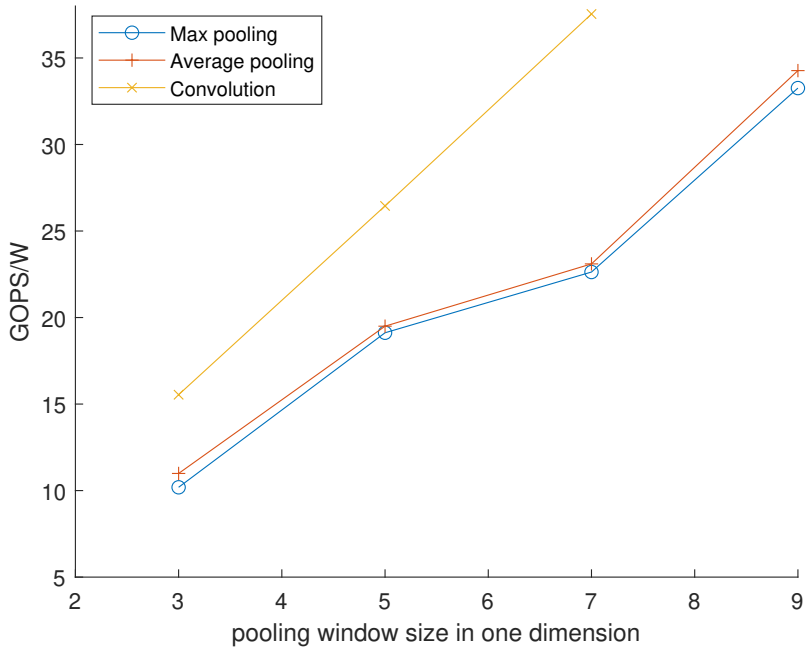

Figure 4.20: GOPS per total power (W) results for varying pooling window sizes

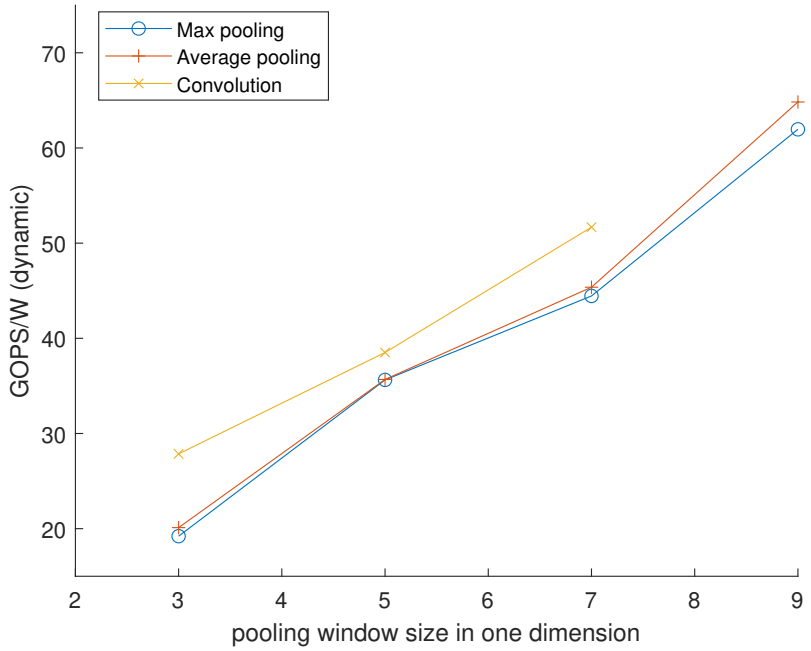

Figure 4.21: GOPS per dynamic power (W) results for varying pooling window sizes

is possible to keep number of operations performed per time at high levels at the expense of increased hardware utilization. As an example, we increased number of AXI ports from stride factor 2×2 to 2×4 which increased hardware utilization but reduced latency significantly and even improved resource and energy efficiency as opposed to the general trend.

4.2 GPU

In this section we will present energy efficiency, power and latency of the building blocks implemented on GPU and accelerated GPU platforms. We will compare our results with the state of the part and among themselves and make comments.

In this section the latency and power values presented in tables belong to individual building blocks. In order to calculate energy efficiency shown in plots we did not use power and latency values from the individual blocks. Instead, we ran multiple same type of blocks in parallel to achieve reasonable hardware utilization levels which are comparable to the levels achieved when complete CNNs are run on GPU platforms and can mask the effect of high base power of the GPU platform in energy efficiency

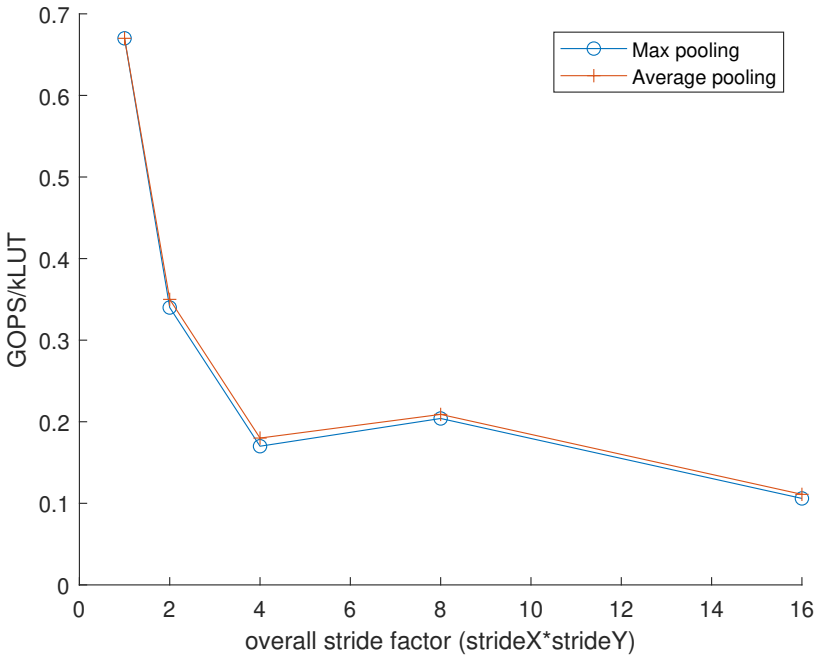

Figure 4.22: GOPS/kLUT results for varying stride factors

Table 4.11: Resource utilization and latency results for varying stride factors

		1×1	2×1	2×2	2×4	4×4
LUTs	Max	6582	6174	7016	8132	7774
	Avr.	6874	6212	7071	8246	7800
Latency(ms)		0.449	0.449	0.381	0.141	0.141

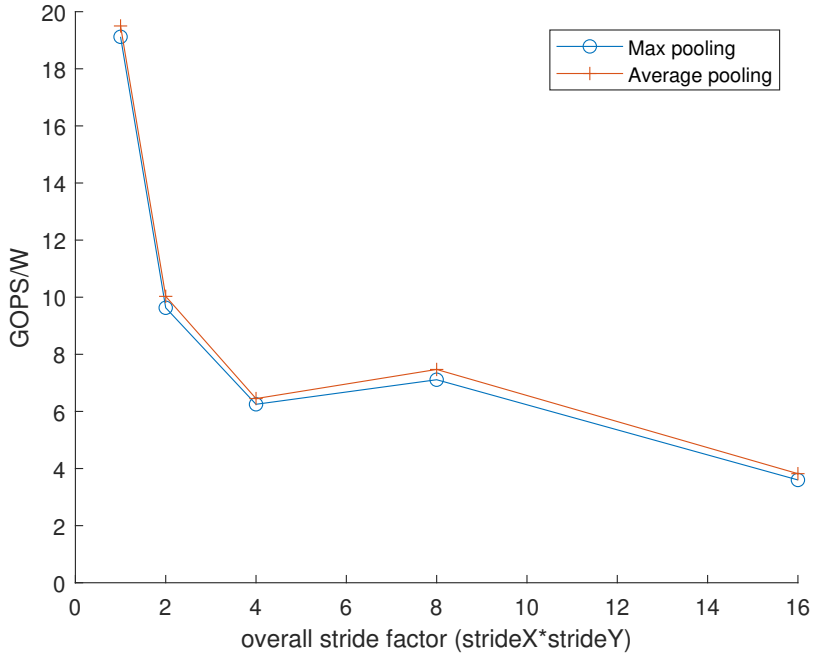

Figure 4.23: GOPS per total power (W) results for varying stride factors

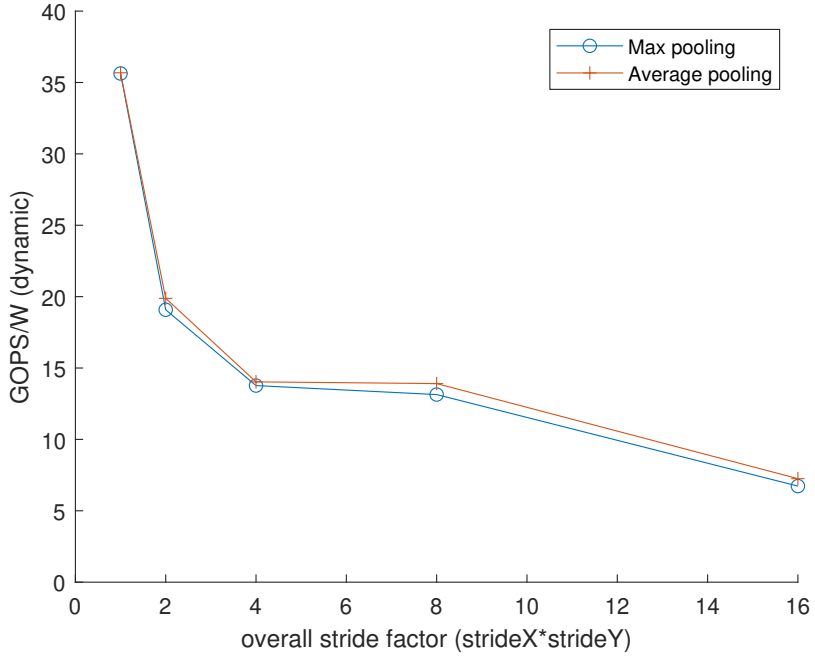

Figure 4.24: GOPS per dynamic power (W) results for varying stride factors

measurements.

4.2.1 Comparison with the State of the Art

Table 4.12 shows energy efficiency of some of our block tests together with some GPU-based CNN implementations from research. We used building blocks with typical or average parameters in our comparisons. The convolution and pooling blocks work with 1024×80 images, 5×5 kernel filters or pooling windows and 1×1 stride factor. The fully connected blocks work with 256 input and 256 output neurons. As we also mentioned in our thesis goals, literature lacks energy efficiency results for accelerated GPUs and provides limited analysis of energy efficiency on GPUs unlike extensive emphasis on performance, hence we compared our results with some GPU implementations for low-precision data types. We also included operational energy efficiency in the table since effect of base power is prominent as our experiments with building blocks did not reach high hardware utilization ratios. The fully connected and pooling blocks showed low energy efficiency as expected due to low computational intensity, hence we will use our convolution blocks in comparisons since typically most of the computations in a CNN happen in convolution blocks. Using DLAs increased the total power consumed due to the high base power of DLAs but improved operational energy efficiency for INT8 type which can be useful for design cases where DLA utilization ratio is high. YOLO Nano [12] which is implemented on the same hardware as our design provided 1.51 times higher energy efficiency than our accelerated GPU implementation and 1.20 times higher energy efficiency than GPU without accelerator cores. YOLO [13] has 1.18 times higher energy efficiency than our FP16 building block without DLA. VGG-16 [13] implemented on same hardware by the same research team as YOLO has much higher efficiency which also demonstrates the significance of the network type and structure for energy efficiency benchmarking. Using DLAs significantly reduced energy efficiency at FP16 which can be explained by the primary target of DLAs being offloading GPU and increasing performance rather than energy efficiency especially for FP16 type. ResNet-50 [64] which uses FP32 neurons achieved higher energy efficiency due to its use of ternary weights. We conclude that our building blocks are comparable with the state of the art and can be used to perform reasonable comparisons with the FPGA building blocks.

4.2.2 2D Convolution Block

We ran the convolution block on GPU platform with different kernel filter sizes and stride factors. We took measurements with both data types which allow DLA utilization (INT8 and FP16) with and without DLA support. In all measurements we used the same image size of 1024×80 as FPGA experiments. In this section, all kernel filter sizes are 5×5 and stride factors are 1×1 unless otherwise is explicitly noted.

Fig. 4.25 and Fig. 4.26 show energy efficiency of the convolution block for different kernel sizes of 3×3, 5×5, 7×7, 9×9 and 11×11. Table 4.13 shows total power and latency values for one block. Similar to FPGA case, increasing the kernel

filter size improved GOPS and therefore energy efficiency. Jetson AGX performed power gating and prevented any power consumption on DLA cores for the test runs without DLA support. Hence test runs without DLA performed better in terms of total power due to considerable base power required to run DLAs. However using DLAs improved operational energy efficiency for INT8. Using INT8 instead of FP16 did not necessarily improve energy efficiency. We observed that Jetson AGX performed FP16 convolutions faster than INT8 in some cases where it was loaded to run numerous convolution operations in parallel. Reduced latency for such test cases made FP16 more energy efficient than INT8 despite increased power consumption of FP16. Using INT8 with DLA cores provided the best operational energy efficiency and therefore can be useful for networks where DLA cores have high utilization ratio. However using DLAs also increases the latency of the individual convolution blocks.

Fig. 4.27 and Fig. 4.28 show energy efficiency of the convolution block for different stride factors of 1×1, 2×2, 2×4, 4×4 and 4×8. Table 4.14 shows total power and latency values for one block. Unlike FPGA, overall run time when we ran multiple convolution operations in parallel scaled down with increasing stride which kept energy efficiency close for different stride factors despite reduced total number of operations performed. Similar to the results for different kernel filter sizes, using DLAs and INT8 can improve operational energy efficiency but using DLAs contributes to overall power consumption and reduces total energy efficiency.

Table 4.12: Energy efficiency results from our building block tests in comparison with other GPU/accelerated GPU implementations from research papers

Data Type	Network	Platform	GOPS/W	GOPS/W (opr.)
FP16	**Conv2d, no DLA (our test)**	Jetson AGX	11.13	17.63
	Conv2d, with DLA (our test)	Jetson AGX	4.31	7.82
	FC, with DLA (our test)	Jetson AGX	0.41	2.19
	VGG-16 [65]	Jetson TX1	31.8	-
	YOLO [65]	Jetson TX1	13.1	-
INT8	**Conv2d, no DLA (our test)**	Jetson AGX	7.51	11.89
	Conv2d, with DLA (our test)	Jetson AGX	5.96	15.13
	FC, with DLA (our test)	Jetson AGX	0.43	3.53
	Max pooling, with DLA (our test)	Jetson AGX	1.25	3.09
	YOLO Nano [12]	Jetson AGX	9.00	-
INT2∼FP32	ResNet-50 [64]	Titan X	∼19	-

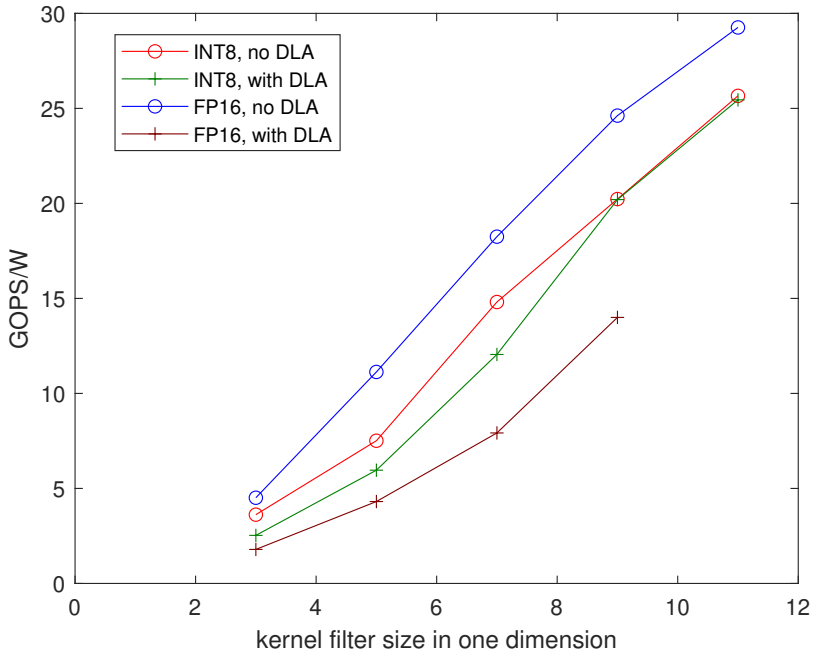

Figure 4.25: GOPS per total power (W) results for varying kernel filter sizes

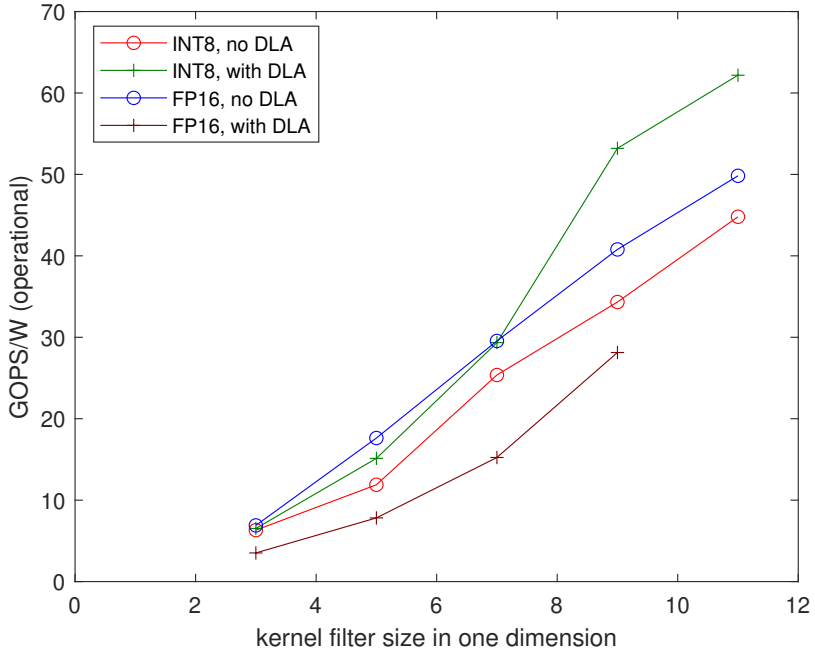

Figure 4.26: GOPS per operational power (W) results for varying kernel filter sizes

Table 4.13: Power and latency comparison of the convolution for different kernel filter sizes

		3×3	5×5	7×7	9×9	11×11
INT8, no DLA	Power(W)	8.37	9.30	9.61	9.57	9.53
	Latency(ms)	0.161	0.256	0.343	0.447	0.586
INT8, with DLA	Power(W)	7.76	7.73	7.45	7.41	7.06
	Latency(ms)	0.775	0.976	1.18	1.35	1.53
FP16, no DLA	Power(W)	10.12	8.94	9.10	9.68	9.38
	Latency(ms)	0.291	0.293	0.637	0.853	1.11
FP16, with DLA	Power(W)	8.18	7.76	7.55	7.35	7.48
	Latency(ms)	0.802	0.802	1.19	1.42	1.52

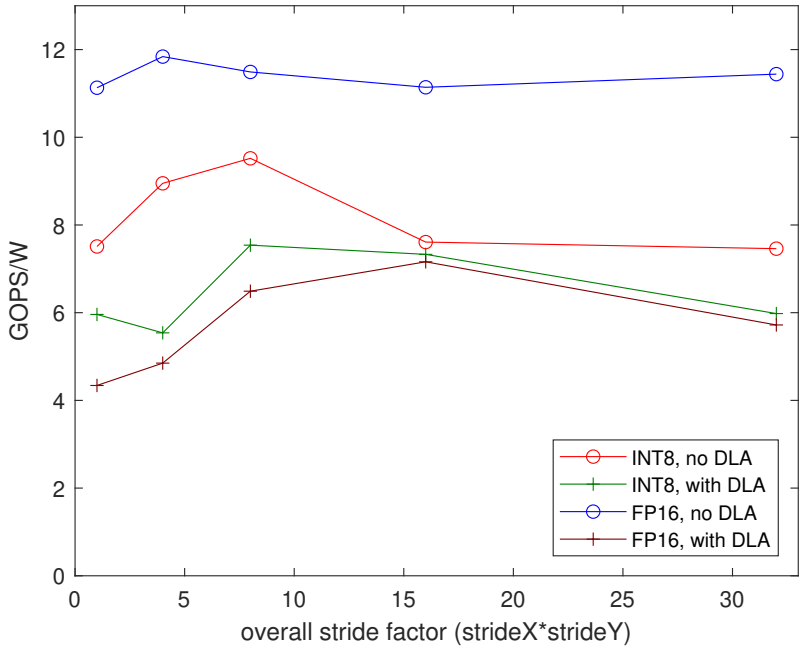

Figure 4.27: GOPS per total power (W) results for varying stride factors

Table 4.14: Power and latency comparison of the convolution for different stride factors

		1×1	2×2	2×4	4×4	4×8
INT8, no DLA	Power(W)	9.30	8.37	7.97	7.69	7.65
	Latency(ms)	0.256	0.146	0.138	0.130	0.114
INT8, with DLA	Power(W)	7.73	7.41	7.29	7.33	7.29
	Latency(ms)	0.976	0.587	0.508	0.475	0.458
FP16, no DLA	Power(W)	8.94	8.75	8.27	7.56	7.61
	Latency(ms)	0.293	0.214	0.168	0.156	0.130
FP16, with DLA	Power(W)	7.76	7.92	7.76	7.65	8.05
	Latency(ms)	0.802	0.642	0.555	0.519	0.496

We measured all six power lines separately and observed the results. GPU, DRAM and SOC power levels changed with workload significantly and defined the operational energy efficiency. CPU and System power remained almost constant for all measurements regardless of the kernel filter size, stride factor or number of parallel convolutions running simultaneously. DLA power only increased slightly with increasing workload. DLA cores had high base power compared to their operational power consumption which increased little with increasing workload, which also reduced the total energy efficiency of all designs with DLA support.

4.2.3 Fully Connected Block

We measured power and latency and calculated energy efficiency of the fully connected block on GPU platform for different input/output neuron sizes and data types with and without DLA support.

Table 4.15 shows power, latency and energy efficiency of the fully connected blocks with 256 input and 256 output neurons on GPU platform. Power and latency values belong to a single block whereas total and operational energy efficiency were measured while many blocks were running in parallel to load GPU. Similar to the convolution, using DLA increased latency and power consumption of a single block due to increased base power. However, using DLA improved operational energy efficiency for both data types, even improved total energy efficiency for INT8 despite increased base power. FP16 without DLA gave the best total energy efficiency whereas INT8 with DLA was superior in terms of operational energy efficiency.

We tested the fully connected block for different input/output neuron numbers.

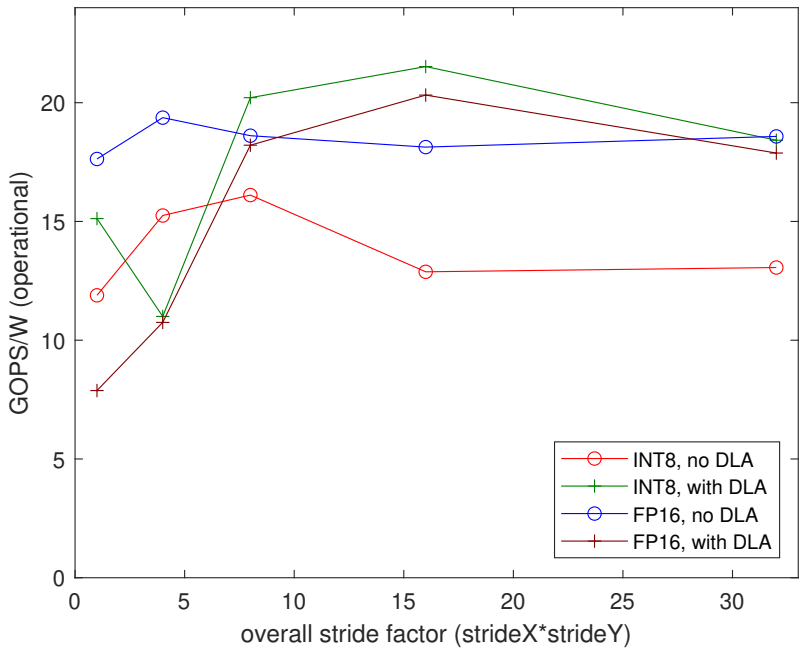

Figure 4.28: GOPS per operational power (W) results for varying stride factors

Increasing input and output neuron numbers increased the number of operations performed per block and improved energy efficiency for GOPS based metrics. Increasing the number of neurons did not increase latency much probably because an important part of the latency comes from data transfer phase, but increased the number of operations performed instead. We also observed the GPU offloading capabilities of DLA in fully connected experiments. GPU with DLA support managed to run more number of building blocks. However, using DLAs put a limit on the maximum number of input neurons that can be processed due to limited size of the matrix operations that run on DLAs.

4.2.4 Pooling Block

We measured power and calculated energy efficiency for different pooling window sizes, stride factors, data types with and without DLA support for 1024×80 input tensor. In this section, all pooling window sizes are 5×5, stride factors are 1×1 and data types are 8-bit integer unless otherwise is explicitly noted.

Fig. 4.29 and Fig. 4.30 show energy efficiency of the max and average pooling blocks for different pooling window sizes of 3×3, 5×5, 7×7 and 9×9. Table 4.16 shows total power and latency values for one block. Results are in parallel with previously shown results for convolution. Energy efficiency increases with pooling window size due to increased number of operations. Max and average pooling showed similar energy efficiency in general, except for some significant difference in 9×9 pooling window case. Pooling without DLA worked with less latency on single blocks and provided better energy efficiency in general. Pooling with DLA achieved better operational energy efficiency for higher pooling window sizes. However pooling window size is limited by 8 in Jetson AGX DLA cores [32] while higher pooling window sizes can be achieved on GPU without using DLAs.

Fig. 4.31 and Fig. 4.32 show energy efficiency of the max and average pooling blocks for different stride factors of 1×1, 2×1, 2×2, 2×4 and 4×4. Table 4.17 shows total power and latency values for one block. As opposed to the convolution case, latency in loaded measurements did not scale down with increased stride factor and reduced number of operations which reduced the energy efficiency for high stride factors. Max and average pooling showed similar energy efficiency. Best energy efficiency was obtained with INT8 data type without DLA. DLA significantly increased latency in single block operations.

Table 4.15: Power, latency and energy efficiency of the fully connected block

	Single		Multi	
	Power(W)	Lat.(ms)	GOPS/W (tot.)	GOPS/W (opr.)
INT8, no DLA	6.36	0.164	0.32	0.88
INT8, with DLA	7.02	0.455	0.43	3.53
FP16, no DLA	6.38	0.083	0.52	1.08
FP16, with DLA	6.79	0.494	0.41	2.19

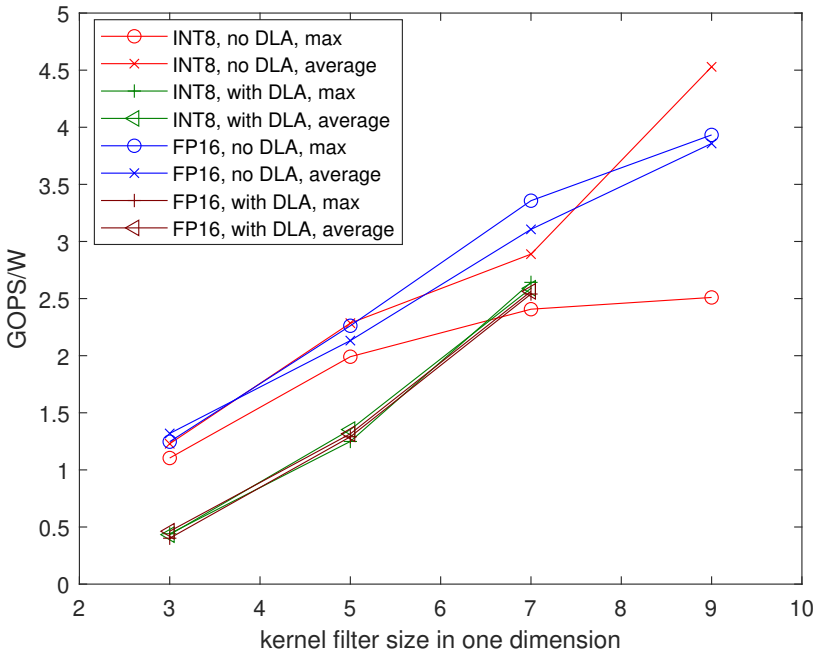

Figure 4.29: GOPS per total power (W) results for varying pooling window sizes

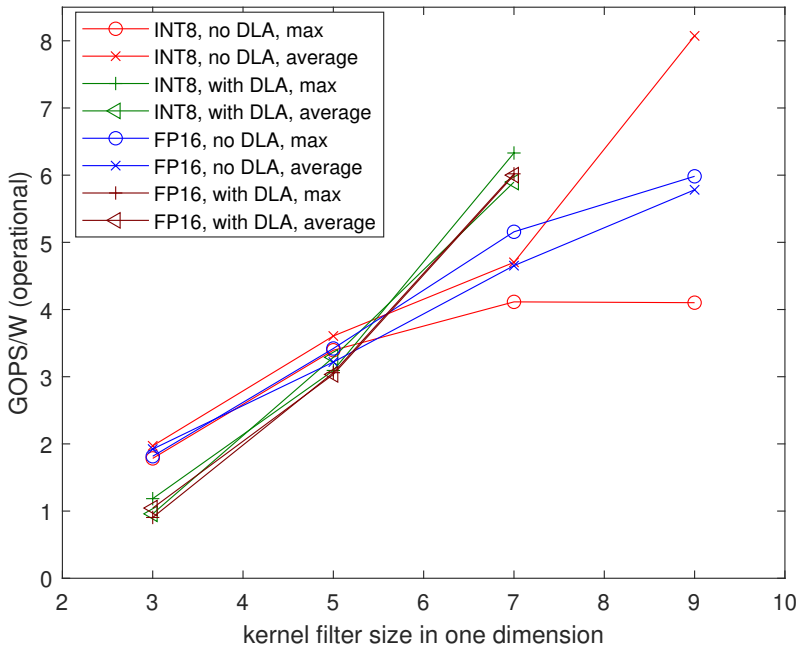

Figure 4.30: GOPS per operational power (W) results for varying pooling window sizes

Table 4.16: Power and latency comparison of the pooling for different pooling window sizes

			3×3	5×5	7×7
INT8, no DLA	Power(W)	Max	8.31	8.69	8.49
		Avr.	8.29	9.30	8.95
	Latency(ms)	Max	0.176	0.270	0.365
		Avr.	0.173	0.215	0.337
INT8, with DLA	Power(W)	Max	8.30	8.15	8.15
		Avr.	8.30	8.15	8.31
	Latency(ms)	Max	0.980	1.025	1.250
		Avr.	0.935	1.010	1.075
FP16, no DLA	Power(W)	Max	8.61	9.19	9.15
		Avr.	8.61	9.19	9.45
	Latency(ms)	Max	0.161	0.185	0.204
		Avr.	0.167	0.193	0.249
FP16, with DLA	Power(W)	Max	9.19	9.04	8.88
		Avr.	9.04	8.88	9.04
	Latency(ms)	Max	1.26	1.34	1.39
		Avr.	1.25	1.34	1.44

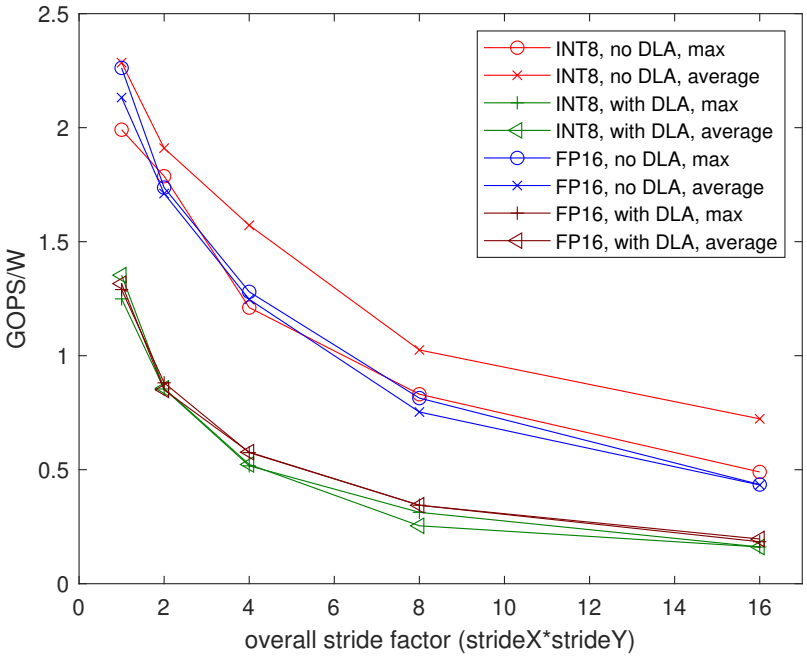

Figure 4.31: GOPS per total power (W) results for varying stride factors

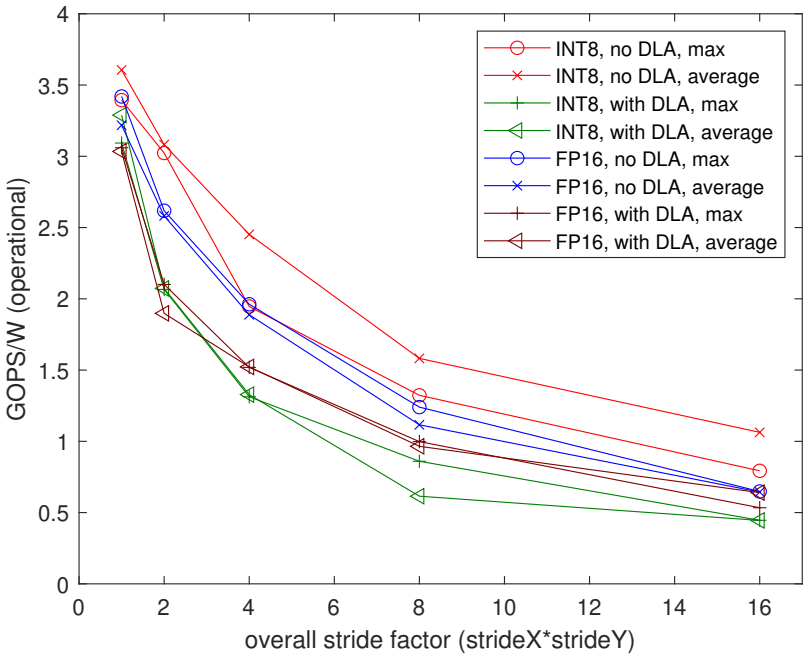

Figure 4.32: GOPS per operational power (W) results for varying stride factors

Table 4.17: Power and latency comparison of the pooling for different stride factors

			1×1	2×1	2×2	2×4	4×4
INT8, no DLA	Power(W)	Max	8.69	8.53	8.22	8.07	7.91
		Avr.	9.30	8.84	8.22	7.91	7.60
	Latency(ms)	Max	0.270	0.231	0.167	0.158	0.141
		Avr.	0.215	0.199	0.177	0.169	0.113
INT8, with DLA	Power(W)	Max	8.15	7.95	7.80	7.80	7.80
		Avr.	8.15	7.80	7-80	7.80	7.96
	Latency(ms)	Max	1.025	1.030	0.901	0.870	0.816
		Avr.	1.010	1.010	0.951	0.991	0.899
FP16, no DLA	Power(W)	Max	9.19	8.38	8.03	7.63	8.03
		Avr.	9.19	8.37	8.22	8.07	7.76
	Latency(ms)	Max	0.185	0.192	0.172	0.140	0.155
		Avr.	0.193	0.143	0.125	0.149	0.132
FP16, with DLA	Power(W)	Max	9.04	9.00	9.00	9.15	9.15
		Avr.	8.88	9.04	9.15	9.15	9.15
	Latency(ms)	Max	1.34	1.36	1.24	1.20	1.19
		Avr.	1.34	1.49	1.34	1.21	1.19

4.3 FPGA vs. Accelerated GPU

In this section we will compare FPGA and accelerated GPU platforms for equivalent operations. In these comparisons same image size (1024×80) for convolution and same input/output neuron numbers (256/256) for fully connected are used. We used INT8 on both platforms since we ran it on both DLA cores of the GPU and FPGA. Operations are functionally equivalent, both platforms generated the same correct output without overflow. We used GPU with DLA cores and FPGA without DSP support for the comparisons throughout this section.

Fig. 4.33 shows energy efficiency based on total, dynamic (FPGA) and operational (GPU) power consumption of the 2D convolution block for different kernel filter sizes of 3×3, 5×5 and 7×7 on FPGA and accelerated GPU platforms. Table 4.18 shows total power and latency values for a single convolution operation. We used 1×1 stride factor for all designs. Increasing kernel filter size improves energy efficiency in both platforms similarly. Latency increases in both platforms with increased kernel filter size. Power consumption increases on FPGA with increased kernel filter size however accelerated GPU power is not significantly affected. FPGA solution provided better energy efficiency, less latency and lower power for all kernel filter sizes.

Fig. 4.34 shows energy efficiency based on total, dynamic (FPGA) and operational (GPU) power consumption of the 2D convolution block for different stride factors of 1×1, 2×2, 2×4, 4×4 and 4×8 on FPGA and accelerated GPU platforms. Table 4.19

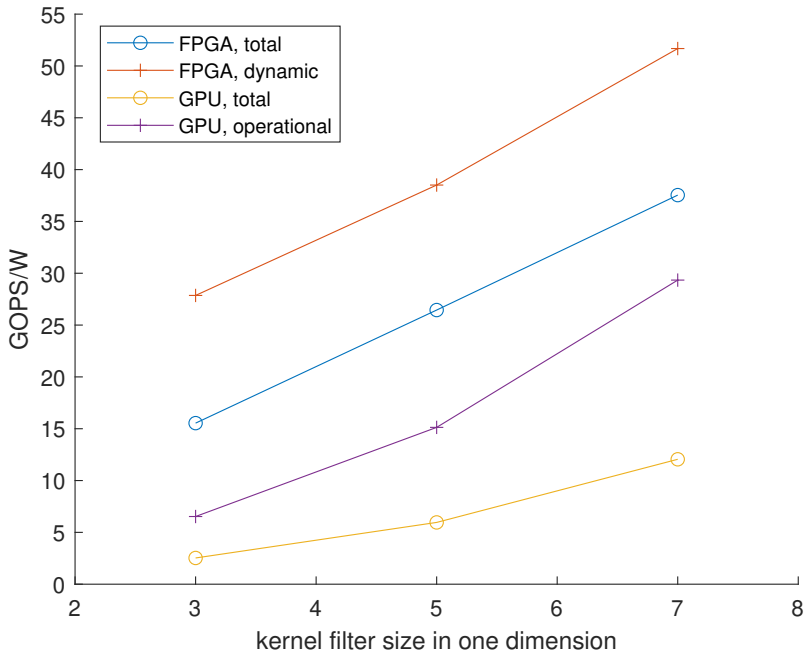

Figure 4.33: GOPS/W for the convolution with different kernel filter sizes on FPGA and accelerated GPU platforms

shows total power and latency values for a single convolution operation. We used 5×5 kernel filter size for all designs. Accelerated GPU energy efficiency remains more or less constant with increasing stride factor since overall latency of multiple blocks running in parallel scales down with stride, however FPGA performance reduces. Therefore accelerated GPU is more efficient for higher stride factors even though FPGA is more efficient for lower stride factors. FPGA provided better power and latency for single building blocks.

Table 4.18: Power and latency comparison of the convolution for different kernel filter sizes on FPGA and accelerated GPU platforms

		3×3	5×5	7×7
FPGA	Power(W)	0.242	0.348	0.402
	Latency(ms)	0.392	0.445	0.532
Accl. GPU	Power(W)	7.76	7.73	7.45
	Latency(ms)	0.78	0.98	1.18

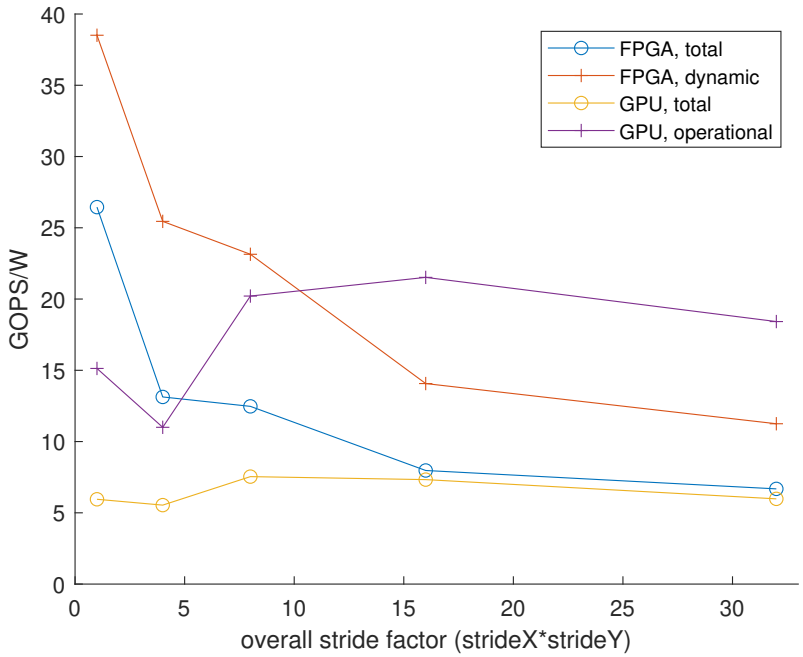

Figure 4.34: GOPS/W for the convolution with different stride factors on FPGA and accelerated GPU platforms

Table 4.19: Power and latency comparison of the convolution for different stride factors on FPGA and accelerated GPU platforms

		1×1	2×2	2×4	4×4	4×8
FPGA	Power(W)	0.348	0.219	0.232	0.247	0.266
	Latency(ms)	0.445	0.356	0.177	0.130	0.072
Accl. GPU	Power(W)	7.73	7.41	7.29	7.33	7.29
	Latency(ms)	0.98	0.59	0.51	0.47	0.46

Table 4.20 shows power, latency and energy efficiency of the fully connected blocks with 256 input and 256 output neurons. FPGA outperformed accelerated GPU in terms of every metric in the table. FPGA also can work with any number of input neurons, unlike accelerated GPU which can only process some limited sized input arrays due to limitations of the DLA cores.

Fig. 4.35 shows energy efficiency based on total, dynamic (FPGA) and operational (GPU) power consumption of the max pooling block for different pooling window sizes of 3×3, 5×5 and 7×7 on FPGA and accelerated GPU platforms. Table 4.21 shows total power and latency values for a single max pooling operation. We used 1×1 stride factor for all designs. Increase in latency and energy efficiency as pooling window size increases is common on both platforms. FPGA significantly outperformed accelerated GPU in terms of energy efficiency for all pooling window sizes. It is also important to note that DLA cores limited maximum possible pooling window size while we managed to implement bigger pooling windows on FPGA.

Fig. 4.36 shows energy efficiency based on total, dynamic (FPGA) and operational

Table 4.20: Power, latency and energy efficiency of the fully connected block on FPGA and accelerated GPU platforms

	Power(W)	Lat.(ms)	GOPS/W (tot.)	GOPS/W (opr.)
FPGA	0.248	0.069	7.62	13.49
Accl. GPU	7.02	0.455	0.43	3.53

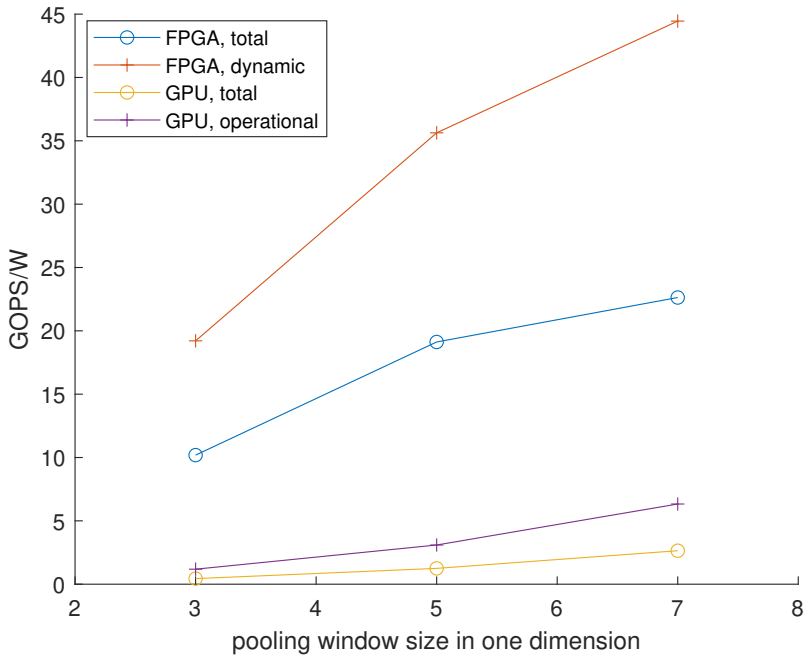

Figure 4.35: GOPS/W for the max pooling with different pooling window sizes on FPGA and accelerated GPU platforms

(GPU) power consumption of the max pooling block for different stride factors of 1×1, 2×1, 2×2, 2×4 and 4×4 on FPGA and accelerated GPU platforms. Table 4.22 shows total power and latency values for a single max pooling operation. We used 5×5 pooling window size for all designs. Both platforms suffered from reduced energy efficiency for higher stride factors similarly. FPGA implementations managed to reduce latency better than accelerated GPU for increasing stride factors. FPGA implementations are more energy efficient for every stride factor.

Table 4.21: Power and latency comparison of the max pooling for different pooling window sizes on FPGA and accelerated GPU platforms

		3×3	5×5	7×7
FPGA	Power(W)	0.228	0.231	0.218
	Latency(ms)	0.282	0.449	0.797
Accl. GPU	Power(W)	8.30	8.15	8.15
	Latency(ms)	0.980	1.025	1.250

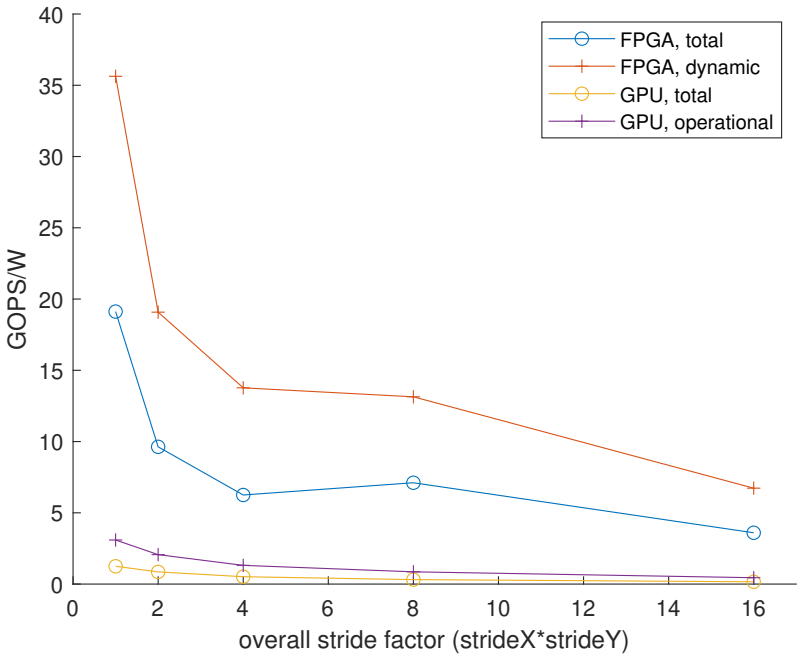

Figure 4.36: GOPS/W for the max pooling with different stride factors on FPGA and accelerated GPU platforms

Table 4.22: Power and latency comparison of the max pooling for different stride factors on FPGA and accelerated GPU platforms

		1×1	2×1	2×2	2×4	4×4
FPGA	Power(W)	0.231	0.216	0.196	0.233	0.230
	Latency(ms)	0.449	0.449	0.381	0.141	0.141
Accl. GPU	Power(W)	8.15	7.95	7.80	7.80	7.80
	Latency(ms)	1.025	1.030	0.901	0.870	0.816

Overall both platforms provided per block latency values that lied between the range of 0.06 ms - 0.5 ms which permits the use of up to 50-415 consecutive blocks before reaching 25 ms limit for a complete CNN and makes them eligible to be used in a CNN that will work with a 40FPS camera. Accelerated GPU utilized significantly higher amount of power for single blocks as shown in Table 4.18. However, power utilization of accelerated GPU did not increase with increasing workload as much as it did for FPGA which would improve its efficiency for complicated networks with high work-load especially when operational energy efficiency is considered as shown in Fig. 4.33. FPGA was more energy efficient for INT8 data type. However our FPGA blocks showed limited flexibility from some aspects, such as not allowing higher kernel sizes due to increasing hardware utilization. Accelerated GPU implementations had very less development time with high flexibility and functionality. Accelerated GPU also enabled the use of FP16 data type which was not implemented on FPGA. FPGA provided more flexibility on low-precision data type selection. Considering FPGA is more energy efficient for INT8, it is even more efficient for sub-INT8 data types since sub-INT8 implementations on FPGAs are more efficient than INT8 implementations on FPGAs and their closest equivalent on accelerated GPUs are INT8 implementations. Integer data types which has higher than 8-bit precision on FPGAs can also be more efficient than accelerated GPUs since their closest equivalent on accelerated GPU is FP16.

4.4 Summary and Key Outcomes

Our FPGA building blocks have comparable performance and energy efficiency with the state of the art. FPGA blocks are especially efficient for relatively higher bit-width integer data types. GPU building blocks used in tests also showed close and comparable energy efficiency to the state of the art.

DLA cores were found to be beneficial only for some specific design cases such as INT8 operations when operational energy efficiency is considered. Utilization of DLAs also increases base power consumption and latency for many design cases. Although DLAs improve system performance by offloading GPU cores, care must be taken when energy efficiency is also a concern.

Preventing overflow by using higher bit-width data types to store results inside the building block is closer to the way many GPU libraries handle CNN operations and can be desirable. FPGA blocks which prevented overflow consumed more power and resources. We used the minimum bit-width that prevents overflow as the intermediate data type in FPGA implementations whereas this data type is fixed to INT32 in accelerated GPU which limited the energy efficiency benefits of reduced bit-width on GPU platform.

Using lower bit-widths improved resource utilization and power consumption significantly on FPGAs. Especially for lower bit-widths, using LUTs for calculations instead of DSPs provided higher energy efficiency due to increased signal power and hardware underutilization in DSPs. However for higher bit-widths high performance

of DSPs which are specialized for such calculations outweighed the extra power consumed on signals and improved the overall energy efficiency. Although power consumed for calculations was reduced by lower bit-width significantly, power consumed on clocks and AXI interface did not scale-down as much. Using lower precision on GPU platform did not improve energy efficiency for all measurements. Some FP16 operations were performed faster on GPU without DLA support which made them superior in terms of energy efficiency. However, INT8 operations with DLA support gave the best operational energy efficiency among GPU implementations.

Increased kernel filter size in FPGA implementations of convolution increased overall power and hardware consumption, however increased GOPS/kLUT and GOPS/W due to increased CTC ratio and data reuse. Power consumption of GPU implementations did not increase much with the kernel filter size, energy efficiency improved in a similar manner as FPGA and latency increased as opposed to FPGA. Increased stride factor reduced GOPS/kLUT and GOPS/W due to reduced CTC ratio and data reuse in FPGA. However it can be exploited to reduce overall hardware utilization or latency on network level due to reduced number of operations needed. Increased stride factor preserved energy efficiency levels by reducing latency on GPU, which made accelerated GPU more energy efficient than FPGA for high stride factors when operational energy efficiency is considered.

Design parameters and choices which affect pipeline initiation intervals, hardware reuse and memory management in FPGAs changed power and resource efficiency significantly. Efficient design methodologies from literature survey helped us to improve performance of our designs. Having generic design parameters enabled fast design, test and verification of the building blocks. Generic design parameters can also be used to optimize designs for different resource, latency and power trade-offs for different applications.

Implementing FPGA accelerators was more time consuming than using GPU libraries which support reasonably wide range of data types and design parameters. Although FPGA accelerators provided better latency and energy efficiency in many cases, design time must be considered when a decision has to be made between these two platforms.

Purpose of using high level synthesis with Vitis HLS to implement FPGA building blocks was to save design time and develop/analyse more number of building blocks with different features in limited time. It would probably have taken more time for us to design these building blocks if we had used a low-level language such as VHDL. However using high level synthesis had its own challenges. It was occasionally hard for us to convey the hardware structure such as pipeline stages in our minds into C code. Since C description is not uniquely transferred to RTL description sometimes even with the help of pragmas, manipulating the synthesizer to construct a certain structure can become tricky. Although high level synthesis can provide very fast development for complicated systems, using low level languages can be beneficial and occasionally only way when a predetermined and well-defined hardware is to be implemented for high performance and energy efficiency.

This thesis is limited to comparison of individual building blocks. It is important to note that although reducing precision typically reduces resource utilization and power consumption of an individual block, it might also necessitate increased number of layers, kernels and parameters in a complete CNN to preserve required accuracy which may in turn increase resource utilization and power consumption. High-level investigation on performance and efficiency has to be performed for a complete CNN. Our results can help to estimate utilization of a CNN for different precisions when network-level exploration for an optimum CNN is performed.

5

Conclusion and Future Work

5.1 Conclusion

We identified and described some major CNN building blocks involved in various computer vision applications. We implemented 2D convolution, fully connected and pooling building blocks on both FPGA and accelerated GPU. We either estimated or measured performance and energy efficiency of these implementations. We compared our results with each other and the state of the art. Depending on the design parameters, our FPGA blocks showed up to 1.39 times better resource efficiency and 1.24 times better energy efficiency than the state of the art. Our GPU building blocks were close to state of the art with down to 1.18 less energy efficiency. Our building blocks are comparable with the state of the art and therefore can be used for reasonable comparisons of the design platforms. We made detailed analysis of design parameters on energy and resource efficiency. Depending on the kernel filter size size, FPGA can be up to 6.14 times more energy efficient for convolution blocks and 23.11 times more energy efficient for pooling blocks in terms of total energy efficiency. Accelerated GPU can be 1.64 more energy efficient in terms of operational energy efficiency for convolution with high stride factors but FPGA outperformed accelerated GPU for every stride factor for pooling. FPGA can perform fully connected operations with up to 28.31 times less power and 6.59 times lower latency than accelerated GPU.

We included resource utilization, performance, energy efficiency, latency and power consumption of our building blocks for different design cases. We specified advantages and disadvantages of both design platforms for various design parameters. We emphasized the effect of low-precision data types on performance and energy efficiency. A machine learning engineer who wants to implement a CNN by making use of the accelerators can refer to our results to have an estimate of resource and power required to utilize such accelerators. Our building blocks on FPGAs can also be directly utilized as accelerator IP cores for a complete CNN application due to their flexibility with generic design parameters.

5.2 Future Work

Our FPGA building blocks have been optimized to work under some conditions. In order to have long AXI bursts, avoid hardware underutilization at edge cases and extra hardware utilization to detect and cover edge cases, we made some assumptions on the input tensor size and stride factor in our blocks. Building blocks are functionally correct for generic input tensors, however they can still be optimized to process odd image sizes efficiently.

Our FPGA convolution building block tiles the input tensor but does not tile the kernel filter operations, instead performs all calculations with all kernel filter cells in parallel. Although this approach works well for small kernel filters which are more common in CNNs, it is not possible to implement convolution with bigger kernel filters. A design where operations over one kernel filter are sharing same hardware resources can make bigger kernel filters possible.

Our FPGA building blocks are less resource and power efficient than state of the art for sub-INT8 precision levels. These blocks can be improved further to take advantage of lowered bit-width. Sub-INT8 precision can also be exploited to further increase bandwidth by byte-sharing in AXI bus among different data.

Our building blocks are meant to be used to accelerate CNN applications. Our blocks have been individually implemented and verified on hardware. They can be used to accelerate a complete CNN and performance and energy efficiency improvement can be observed.

Clock frequency is one of the important factors that define the performance of systems. In this thesis we used the same clock frequency for all FPGA implementations of the building blocks. Since number of different clock signals on an FPGA board is limited and using multiple clock domains within the same FPGA brings extra obstacles at clock domain crossings, it is not always possible to tailor the clock frequency for individual building blocks. In a more likely scenario, building blocks might have to be implemented with the clock frequency assigned as a system level decision. Investigation of effect of clock frequency can be performed on a complete network level where multiple blocks are implemented on the same FPGA chip.

Bibliography

[1] Stanford University. "convolutional neural networks (CNNs / ConvNets)". Accessed on: April. 8, 2021. [Online]. Available: https://cs231n.github.io/convolutional-networks/.

[2] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, no. 4, p. 65–76, Apr. 2009. [Online]. Available: https://doi.org/10.1145/1498765.1498785

[3] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, "A survey of the recent architectures of deep convolutional neural networks," *Artificial Intelligence Review*, vol. 53, no. 8, pp. 5455–5516, 2020. [Online]. Available: https://app.dimensions.ai/details/publication/pub.1126862968

[4] P. Colangelo, N. Nasiri, E. Nurvitadhi, A. Mishra, M. Margala, and K. Nealis, "Exploration of low numeric precision deep learning inference using Intel® FPGAs," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 73–80.

[5] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, "A survey of deep neural network architectures and their applications," *Neurocomputing*, vol. 234, pp. 11–26, 2017. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231216315533

[6] Y. Han, G. Huang, S. Song, L. Yang, H. Wang, and Y. Wang, "Dynamic Neural Networks: A Survey," *arXiv e-prints*, p. arXiv:2102.04906, Feb. 2021.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[8] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2015.

[9] Y. Li, Z. Liu, K. Xu, H. Yu, and F. Ren, "A GPU-outperforming FPGA accelerator architecture for binary convolutional neural networks," *J. Emerg. Technol. Comput. Syst.*, vol. 14, no. 2, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3154839

[10] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, "Accelerating CNN inference on FPGAs: A survey," *CoRR*, vol. abs/1806.01683, 2018. [Online]. Available: http://arxiv.org/abs/1806.01683

[11] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *ArXiv*, vol. abs/1410.0759, 2014.

[12] A. Wong, M. Famouri, M. Shafiee, F. Li, B. Chwyl, and J. Chung, "YOLO Nano: a highly compact you only look once convolutional neural network for object detection," *ArXiv*, vol. abs/1910.01271, 2019.

[13] C. Yao, W. Liu, W. Tang, J. Guo, S. Hu, Y. Lu, and W. Jiang, "Evaluating and analyzing the energy efficiency of CNN inference on high-performance GPU," *Concurrency and Computation: Practice and Experience*, vol. 33, no. 6, p. e6064, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.6064

[14] M. Sewak, R. Karim, and P. Pujari, "Deep neural networks - overview," in *Practical Convolutional Neural Networks*. Birmingham, UK: Packt Publishing, 2018, pp. 6–17.

[15] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, 2017, pp. 1–6.

[16] J. Yang, X. Huang, Y. He, J. Xu, C. Yang, G. Xu, and B. Ni, "Reinventing 2D convolutions for 3D images," *IEEE Journal of Biomedical and Health Informatics*, pp. 1–1, 2021.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, p. 84–90, May 2017. [Online]. Available: https://doi.org/10.1145/3065386

[18] S. S. Basha, S. R. Dubey, V. Pulabaigari, and S. Mukherjee, "Impact of fully connected layers on performance of convolutional neural networks for image classification," *Neurocomputing*, vol. 378, pp. 112–119, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231219313803

[19] Technische Universität München. "layers of a convolutional neural network". Accessed on: April. 27, 2021. [Online]. Available: https://wiki.tum.de/display/lfdv/Layers+of+a+Convolutional+Neural+Network.

[20] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, "Activation functions: Comparison of trends in practice and research for deep learning," *ArXiv*, vol. abs/1811.03378, 2018.

[21] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[DL] A survey of FPGA-based neural network inference accelerators," *ACM Trans.*

Reconfigurable Technol. Syst., vol. 12, no. 1, Mar. 2019. [Online]. Available: https://doi.org/10.1145/3289185

[22] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 35–44. [Online]. Available: https://doi.org/10.1145/3020078.3021727

[23] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 161–170. [Online]. Available: https://doi.org/10.1145/2684746.2689060

[24] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, February 21-23, 2016*, D. Chen and J. W. Greene, Eds. ACM, 2016, pp. 26–35. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847265

[25] J. Monteiro, R. Patel, and V. Tiwari, "Power analysis and optimization from circuit to register-transfer levels," in *EDA for IC Implementation, Circuit Design, and Process Technology*, L. Scheffer, L. Lavagno, and G. Martin, Eds. Boca Raton, Florida: CRC Press, 2006, pp. 2–8.

[26] A. Montgomerie-Corcoran, S. I. Venieris, and C. Bouganis, "Power-aware FPGA mapping of convolutional neural networks," in *2019 International Conference on Field-Programmable Technology (ICFPT)*, 2019, pp. 327–330.

[27] XILINX. "AXI reference guide". Accessed on: Mar. 08, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf.

[28] A. Lu, Z. Fang, W. Liu, and L. Shannon, "Demystifying the memory system of modern datacenter FPGAs for software programmers through microbenchmarking," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 105–115. [Online]. Available: https://doi.org/10.1145/3431920.3439284

[29] R. Li, H. Huang, Z. ke Wang, Z. Shao, X. Liao, and H. Jin, "Optimizing memory performance of XILINX FPGAs under Vitis," *ArXiv*, vol. abs/2010.08916, 2020.

[30] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, 2010.

[31] H. Kim, S. Ahn, Y. Oh, B. Kim, W. W. Ro, and W. J. Song, “Duplo: Lifting redundant memory accesses of deep neural networks for GPU tensor cores,” in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 725–737.

[32] NVIDIA. "NVIDIA TensorRT Documentation". Accessed on: May 06, 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[33] C. A. Navarro, R. Carrasco, R. J. Barrientos, J. A. Riquelme, and R. Vega, “GPU tensor cores for fast arithmetic reductions,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 72–84, 2021.

[34] NVIDIA. "NVIDIA tensor cores". Accessed on: Mar. 18, 2021. [Online]. Available: https://www.nvidia.com/en-us/data-center/tensor-cores/.

[35] G. Zhou, J. Zhou, and H. Lin, “Research on nvidia deep learning accelerator,” in *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*, 2018, pp. 192–195.

[36] NVIDIA. "NVDLA Open Source Project-Hardware Architectural Specification". Accessed on: April. 1, 2021. [Online]. Available: http://nvdla.org/hw/v1/hwarch.html.

[37] R. A. Bridges, N. Imam, and T. M. Mintz, “Understanding GPU power: A survey of profiling, modeling, and simulation methods,” *ACM Comput. Surv.*, vol. 49, no. 3, Sep. 2016. [Online]. Available: https://doi.org/10.1145/2962131

[38] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, “Power aware computing on GPUs,” in *2012 Symposium on Application Accelerators in High Performance Computing*, 2012, pp. 64–73.

[39] S. Hong and H. Kim, “An integrated GPU power and performance model,” *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 280–289, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815998

[40] L. Cavigelli and L. Benini, “Origami: A 803-gop/s/w convolutional network accelerator,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 11, pp. 2461–2475, 2017.

[41] M. Yang, S. Wang, J. Bakita, T. Vu, F. D. Smith, J. H. Anderson, and J. Frahm, “Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: Addressing an industrial challenge,” in *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 305–317.

[42] TUL. "PYNQ-Z2 reference manual v1.0 ". Accessed on: Mar. 04, 2021. [Online]. Available: https://d2m32eurp10079.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf.

[43] XILINX. "Vitis high level synthesis user guide". Accessed on: Jan. 28, 2021. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf.

[44] XILINX. "HLS pragmas". Accessed on: Jan. 28, 2021. [Online]. Available: https://www.xilinx.com/html_docs/xilinx2020_2/vitis_doc/hls_pragmas.html?hl=pragma.

[45] R. Timofte, K. Zimmermann, and L. van Gool, "Multi-view traffic sign detection, recognition, and 3D localisation," in *Ninth IEEE Computer Society Workshop on Application of Computer Vision*, Snowbird, Utah, USA, December 2009, pp. 1–8.

[46] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *ICANN*, 2014.

[47] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, "Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 16–25. [Online]. Available: https://doi.org/10.1145/2847263.2847276

[48] D. T. Nguyen, T. N. Nguyen, H. Kim, and H.-J. Lee, "A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 8, pp. 1861–1873, 2019.

[49] MATLAB. "fullyconnectedlayer". Accessed on: April. 27, 2021. [Online]. Available: https://se.mathworks.com/help/deeplearning/ref/nnet.cnn.layer.fullyconnectedlayer.html.

[50] NVIDIA. Jetson AGX Xavier. Accessed on: Mar. 16, 2021. [Online]. Available=https://developer.nvidia.com/embedded/jetson-agx-xavier.

[51] NVIDIA. Jetson AGX Xavier Developer Kit. Accessed on: Mar. 16, 2021. [Online]. Available= https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[52] NVIDIA. "sampleMNISTAPI.cpp". Accessed on: May 11, 2021. [Online]. Available: https://github.com/NVIDIA/TensorRT/blob/master/samples/opensource/sampleMNISTAPI/sampleMNISTAPI.cpp.

[53] NVIDIA. "NVIDIA TensorRT documentation". Accessed on: April. 1, 2021. [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/support-matrix/index.html.

[54] NVIDIA. "Jetson AGX Xavier thermal design guide". Accessed on: Mar. 1, 2021. [Online]. Available: https://

static5.arrow.com/pdfs/2018/12/12/12/22/1/565659/nvda_/manual/jetson_agx_xavier_thermal_design_guide_v1.0.pdf.

[55] NVIDIA. "Jetson AGX Xavier: Deep learning inference benchmarks". Accessed on: May 11, 2021. [Online]. Available: https://developer.nvidia.com/embedded/jetson-agx-xavier-dl-inference-benchmarks.

[56] Texas Instruments. "INA3221 triple-channel, high-side measurement, shunt and bus voltage monitor with I2C-and SMBUS-compatible interface". Accessed on: April. 1, 2021. [Online]. Available: https://www.ti.com/lit/ds/symlink/ina3221.pdf.

[57] NVIDIA. "NVIDIA deep learning platform: Technical overview". Accessed on: May 12, 2021. [Online]. Available: http://www.nextplatform.com/wp-content/uploads/2018/01/inference-technical-overview-1.pdf.

[58] NVIDIA. "improving INT8 accuracy using quantization aware training and the NVIDIA transfer learning toolkit". Accessed on: April. 1, 2021. [Online]. Available: https://developer.nvidia.com/blog/improving-int8-accuracy-using-quantization-aware-training-and-the-transfer-learning-toolkit/.

[59] NVIDIA. "deploying quantization-aware trained networks using TensorRT". Accessed on: April. 1, 2021. [Online]. Available: http://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s21664-toward-int8-inference-deploying-quantization-aware-trained-networks-using-tensorrt.pdf.

[60] F. Zhu, R. Gong, F. Yu, X. Liu, Y. Wang, Z. Li, X. Yang, and J. Yan, "Towards unified INT8 training for convolutional neural network," 2019.

[61] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang, and Y. Wang, "An FPGA-based reconfigurable CNN accelerator for YOLO," in *2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, 2020, pp. 74–78.

[62] W. Ding, Z. Huang, Z. Huang, L. Tian, H. Wang, and S. Feng, "Designing efficient accelerator of depthwise separable convolutional neural network on FPGA," *Journal of Systems Architecture*, vol. 97, pp. 278–286, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762118304612

[63] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating binarized convolutional neural networks with software-programmable FPGAs," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 15–24. [Online]. Available: https://doi.org/10.1145/3020078.3021741

[64] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. O. G. Hock, Y. T. Liew, K. Srivatsan, D. J. M. Moss, S. Subhaschandra, and G. Boudoukh,

"Can FPGAs beat GPUs in accelerating next-generation deep neural networks?" *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[65] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-Eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 1, pp. 35–47, 2018.