Hardware Acceleration of YOLOv3-tiny Object Detection

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Computer Science
and Engineering by Research

by

V.V.S.Prithvi
2018900104
prithvi.velicheti@research.iiit.ac.in

International Institute of Information Technology
Hyderabad - 500 032, INDIA
April 2023

International Institute of Information Technology
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled **Hardware Acceleration of YOLOv3-tiny Object Detection**, has been carried out under my supervision and is not submitted elsewhere for a degree.

Date

Adviser: Dr Suresh Reddy Purini

To my parents V. Jhansi Vijaya Lakshmi & V. Sri Krishna Baba

Acknowledgments

I am deeply grateful to everyone who made this journey possible and enjoyable. I sincerely thank my supervisor, Dr Suresh Purini, for entrusting me with a great deal of freedom and, at the same time providing continuous guidance, support, and encouragement. I learned much from him at the technical, personal, and research levels. He transformed me from a very anxious individual to a calm, curious researcher. I am forever indebted to my manager at Indian Space Space Research Organisation (ISRO), Dr G Prasad, whose unconditional support helped me to groom my career. I am incredibly privileged to work under him. I also want to thank my colleague M.Srikanth Yadav at ISRO, for introducing me to the world of FPGAs and building an incredible infrastructure, creating an excellent growth-oriented work environment. I am also grateful to my wonderful mentor Dr Lavanya Ramapantulu, who gave me incredible support and guidance during the most needed times. I feel fortunate to work with my coauthor Sivani, a very dedicated friend and researcher. I am grateful to work under my new manager Mythili, who provides excellent opportunities to learn and evolve. I am very thankful to my friends and colleagues at ISRO, Vaibhav, Ankit, Santoshi, Rohit, Venkatesh and Suma, particularly Vaibhav, from whom I learnt a lot technically and personally and Santoshi, who is always present to hear me out. Finally, I would like to thank my mother, Jhansi, and nothing would have been possible without her love and support and my late father Sri Krishna Baba, whom I am always proud of and look up to.

Abstract

FPGAs are increasingly significant for deploying convolutional neural network (CNN) inference models because of performance demands and power constraints in embedded and data centre applications. The compute intensity of these models makes prototyping highly complex and time-consuming with traditional RTL approaches. The release of new generation high-level synthesis tools (HLS), such as Intel FPGA SDK for OpenCL, and Xilinx's VITIS Unified Software Development platform, have significantly reduced the time and complexity of prototyping complex designs on FPGA. This work involves building custom FPGA accelerators for image recognition systems using OpenCL-HLS. Object detection and classification are vital steps in building image recognition systems. The first part of the work concerns building an FPGA accelerator for the Traffic Sign Classification problem, a vital step in building traffic sign recognition (TSR) systems that employ vehicle-mounted cameras that identify traffic signs while driving on the road. However, the CNNs for the classification still need the ability to be spatially invariant to the input data. Spatial Transformers are learnable modules that, upon integration with CNN, would allow the spatial manipulation of data within the network, making it invariant to affine transformations. Generic Matrix multiply (GEMM) methods that express convolution as matrix multiplication are widely used in deep-learning frameworks like Caffe, Theano, and Torch with GPU support. *im2row* is one of the commonly used GEMM methods. In this work, we built a GEMM-based accelerator for a CNN with a Spatial transformer module. We proposed the channel adaptive *im2row* method, with a lesser on-chip memory footprint than *im2row*. The system attains a latency of 202 ms ($\tilde{5}$ fps), running at 202 MHz on Intel Arria10 GX FPGA, and attains a speedup of ($>$ 5 X) compared to the CPU. The performance is not state-of-the-art and calls for more FPGA-specific optimizations. Further, from the learnings, we designed a Systolic Array accelerator with a novel load pattern for accelerating the widely-used object detector YOLOv3-tiny optimized explicitly for embedded applications. We build the accelerator for multiple precisions (FIXED8, FIXED16, FLOAT32) of YOLOv3-tiny. The architecture uses a homogenous systolic array architecture with a synchronized pipeline adder tree for convolution, allowing it to be scalable for multiple variants of YOLO with a change in the host driver. It is a deeply pipelined architecture that also exploits three-dimensional spatial parallelism. We evaluated the design on Terasic DE5a-Net-DDR4. The Fixed point (FP-8, FP-16) implementations attain a throughput of 57 GOPs/s ($>$ 23 %) and 46.16 GOPs/s ($>$ 340 %) running at 234 MHz and 227 MHz. We synthesized the first FLOAT32 implementation attaining 11.22 GFLOPs/s, running at 172 MHz.
Keywords: FPGA, OpenCL, CNN, Spatial Transformer, GEMM, Systolic Array, YOLOv3-tiny

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Object Detection and Classification

Computer vision has many applications in self-driving cars, robotics, video surveillance, sports analytics, etc. In recent years, the landscape of computer vision has been drastically altered and driven forward by adopting a fast, scalable, end-to-end learning framework, the Convolutional Neural Network (CNN) [22]. We now see a plenitude of CNN-based models achieving state-of-the-art results in classification, localisation , semantic segmentation , and action recognition tasks, amongst others. Object detection and classification are essential in computer vision applications, specifically for autonomous driver assistance systems (ADAS) and video surveillance. Object detection technology deals with the problem of detecting instances of objects in images and videos. Accurate object detection is often required to form the basis for the rest of the computational pipeline. Recent advances in deep learning (DL) give rise to efficient approaches for extracting features from images. Existing DL-based models can be categorized into two categories: two-stage detectors based on region proposals and one-stage detectors based on regression/classification. Two-stage detectors follow the traditional approach by scanning the whole image and focusing on regions of interest. The first stage generates region proposals known as candidate bounding boxes, and the second stage extracts the features from each bounding box to perform classification and bounding box regression. The most popular two-stage detectors are R-CNN [13], Fast R-CNN [12], R-FCN [10], Faster R-CNN [36], and Mask R-CNN [14]. However, these approaches fail to achieve real-time speed due to the expensive running process and the inefficiency of region propositions. To provide object detectors with lower computational requirements , research has focused on the one-step approach, where the bounding boxes around the objects are predicted directly through the DNN rather than having a separate region proposal step . One-stage detectors treat object detection as a regression/classification problem using a unified framework to obtain the labels and locations. These detectors map straightly from image pixels to bounding box coordinates and class probabilities. Two of the most famous such frameworks are the Single Shot MultiBox Detector (SSD) [27] and the YOLO (You only look once) [33]. The most well-known one-stage detector is You Only Look Once (YOLO) and its successors YOLOv2, YOLOv3 and YOLOv4. SSD is based on a

VGG16 network and has been extended by custom convolution layers to generate bounding boxes. SSD uses a set of predefined anchor boxes for detection at various scales impacting the framework's precision and computational load. The YOLO framework relies on a single DNN, DarkNet [32], to predict both the position of the objects (i.e. bounding boxes) and their classification. Early versions of the YOLO approach exhibited low computational loads by trading the classification precision for low latency, which led to their deployment in embedded systems. YOLOv3-tiny is a lightweight version of YOLOv3 [35] with fewer layers to optimize for edge computing applications. The ultimate goal of autonomous vehicle research is to develop a fully automated system that would perform well in all kinds of scenarios. The system should understand the intricate traffic patterns and make real-time decisions based on the visual data from cameras and information from other sensors like LIDAR. Traffic sign recognition systems are vital in autonomous driving environments and advanced driver assistance systems (ADAS). Traffic Sign Classification is an essential step in building traffic sign recognition (TSR) systems that employ vehicle-mounted cameras which identify traffic signs while driving on the road. Building a Convolutional Neural Network for the classification task would be an optimal approach. However, they are still limited by the lack of ability to be spatially invariant to the input data. Spatial Transformers [16] are learnable modules that, upon integration with CNN, would allow the spatial manipulation of data within the network, making it invariant to affine transformations. In this thesis, firstly, we build an FPGA-based accelerator for a Traffic Sign Classification System using Convolution Neural networks with Spatial Transformer Networks. Further, we accelerate the most commonly used Yolov3-tiny for efficient deployment and acceleration on edge.

1.2 Motivation

Deep learning is driving a technological and societal revolution. Deep Neural Networks (DNNs) are now the foundation for many modern artificial intelligence (AI) applications. Due to the tremendous accuracy obtained by these models in image and speech recognition, the trend has motivated researchers to use DNN algorithms in a myriad of applications, from self-driving cars to detecting cancer, to playing complex games, in finance. The superior accuracy of DNNs comes at the cost of high computational complexity. To date, general-purpose compute engines, especially Graphics Processing Units (GPUs), have been the mainstay of much DNN inference. GPU is a CPU-like architecture with a specialized parallel structure that allows them to have more cores than traditional CPUs. However, a single CPU core is more capable than a GPU core. Unlike CPUs which are general purpose, GPUs are designed for specific functions like graphics rendering. GPUs are ideal for accelerating programs implemented in Single Instruction Multiple Data fashion and can be programmed with software developer environments like CUDA [29] and OpenCL [18]. GPUs are power-hungry devices, and their power efficiency improvements are reaching their limits. With power consumption and efficiency being the main bottleneck in designing and employing large High-performance computing (HPC) and embedded applications, the usability of GPU is subject to many power and cooling limitations, especially in embedded environments.

In these dwindling days of Moore's era, there is a need for more specialized hardware to improve compute performance and energy efficiency. Field Programmable Gate Array (FPGA) is a reconfigurable device with programmable interconnects. A programmer can configure its logic to adopt any architecture i.e. pipelined, systolic [20], dataflow, SIMD/Multiple Instruction Multiple Data (MIMD) etc. With appropriate hardware design, FPGAs can attain higher power efficiency (Performance/Watts) over a GPU. They can attain high Performance for sequential programs using deeply pipelined architectures. FPGAs are usually programmed using Hardware Description languages (HDL), mainly Verilog and VHDL, which has an entirely different programming model compared to standard software programming languages like C and Python, which usually makes them tougher to adopt among software programmers. However, the advent of High-Level Synthesis (HLS) tools, which allow software programmers to express their FPGA design in the standard software programming language, enabled rapid programmability on FPGA. The release of the new generation HLS tools, such as Intel FPGA SDK for OpenCL and Xilinx's VITIS Unified Software Development Platform, significantly reduced the time and complexity of prototyping complex designs on FPGA.

Over the past few years, there has been a significant amount of research on the efficient processing of Deep Neural networks (DNNs). DNN inference is a very compute-intensive task. It is challenging to meet performance metrics such as latency and throughput while optimizing power. Special-purpose ASICs and FPGAs are suitable candidates to simultaneously meet these power and performance budgets. FPGAs are becoming increasingly significant for deploying deep neural network (DNN) inference models both on the server side in the data centres and at the edge. Rapidly evolving CNN architectures involve novel convolution operations such as point convolutions, depth separable convolutions, etc. This leads to substantial variation in the computational structure across CNNs and layers within a CNN. Because of this, FPGA reconfigurability provides an attractive tradeoff compared to ASICs. FPGA-based hardware designs can address the structural variability issue by generating a network-specific accelerator for a single network or a class of networks. Unfortunately, the vast majority of FPGA implementations of CNNs have only been implemented in the convolutional layers limiting the benefit of the approach since other layers may quickly become the bottleneck of the neural network.

This work focuses on building efficient architectures for accelerating Convolution Neural Network (CNN) based object classification tasks with specialized layers like Spatial Transformer and Yolo layer. Spatial Transformers are learnable modules that, upon integration with CNN, would allow the spatial manipulation of data within the network, making it invariant to affine transformations. They are widely used in Traffic Sign Recognition systems for Traffic Sign Classification. We built a specialized architecture for traffic sign classification CNN with a spatial transformer network in this work. Further, we built a custom accelerator for the widely used YoloV3-tiny network. We built the CPU and FPGA heterogeneous computing architectures using Intel FPGA SDK for OpenCL. We use Intel Arria10 GX / Terasic DE5Net-DDR4 FPGAs and the workstation-class Intel Xeon x86-64 CPU with 64 GB RAM to evaluate our designs.

1.3 Thesis Contributions

The thesis is majorly focused on building FPGA-based accelerators for image recognition problems. It comprises two parts. The first part concerns building an accelerator for a Convolution Neural network with a Spatial Transformer module for Traffic Sign Classification. Further, we develop a Systolic Array based FPGA Accelerator for the widely used YOLOv3-tiny object detector. The thesis contributions are as follows.

FPGA Accelerator for Traffic Sign Recognition using Spatial Transformer networks

1. Built a Generic Matrix Multiply (GEMM) based accelerator for a CNN with a spatial transformer module for Traffic Sign Classification. The method uses a channel adaptive im2row approach, with a lesser memory footprint than the traditional im2row.

2. Evaluated the design on Intel Arria10 GX FPGA. The system attains a latency of 202 ms ($\tilde{5}$fps), running at 202 MHz with a speedup of ($>$5X) compared to the multithreaded CPU implementation.

Systolic Array based FPGA Accelerator for YOLOv3-tiny

1. Proposes a deeply pipelined 1D Systolic array accelerator with a novel load pattern for accelerating the convolution of YOLOv3-tiny to reduce global interconnects and large multiplexers, thereby reducing data movements to obtain high throughputs.

2. The design exploits 3D spatial parallelism using specialized MAC tree architecture, allowing multiplications to run synchronously with a pipelined adder tree. We use the Intel OpenCL framework for the architecture design, which is scalable for multiple variants of YOLO.

3. Evaluated the design on the Terasic DE5anet-DDR4 FPGA for multiple precisions (FIXED-8, FIXED-16, FLOAT32). While running YOLOv3-tiny for 416x416 RGB image, the fixed point (FP-8) attains a throughput of 57 GOPs/s with a framerate of 10.2 fps running at 234 MHz. Fixed point (FP-16) attains 46.16 GOPs/s with a framerate of 8.278 fps, running at 227.78 MHz. The floating-point (FLOAT32) design achieves 11.22 GFLOPs/s running at 172.92 MHz.

Chapter 2

FPGA Design with High Level Synthesis

Heterogeneous computing is proliferating in data centres and the cloud. Microsoft uses FPGAs to speed up search engines and machine learning for cloud services for power efficiency [8]. AWS provides EC2 F1 [5] instances which comprise FPGAs as custom hardware accelerators. FPGAs are usually programmed using Hardware Description languages (HDL), mainly Verilog and VHDL, which has entirely different programming model compared to standard software programming languages, usually making them tougher to adopt among software programmers.For many years, High-Level Synthesis (HLS) tools have been developed to make FPGAs usable by software developers. Such tools allow software programmers to describe their FPGA design in a standard software programming language and then convert this high-level description to a low-level description based on Verilog or VHDL. Many such tools have been developed since the inception of HLS. Altera (now Intel FPGA) introduced their Intel FPGA SDK for OpenCL to provide a similar possibility for software programmers based on the open-source and royalty-free OpenCL programming language. Eventually, Xilinx followed suit and introduced their OpenCL SDK named SDAccel, now called Vitis Unified development software platform. With official HLS tools being directly developed and supported by FPGA manufacturers, a sudden shift in the HLS ecosystem happened that enabled more widespread adoption of FPGAs among software programmers. This chapter presents critical concepts involved in developing heterogeneous FPGA accelerators with the Open Computing Language (OpenCL).

2.1 Field Programmable Gate Arrays

FPGA is an integrated circuit that can be reconfigurable after manufacturing. They are generally regarded as a middle ground between ASICs and general-purpose processors. This notion comes from their reconfigurability, making them more flexible than ASICs and power efficient than general-purpose processors. FPGAs are primarily composed of SRAM cells arranged in the form of Loop-Up Tables (LUT), a plethora of registers, and programmable routing. The devices can be rapidly reconfigured to implement different logic just by changing the content of the LUTs and the routing configuration. Apart from the soft-logic LUTs, modern FPGAs also include hard-logic components such as Digital

Signal Processors (DSP), large memory blocks (Block RAMs) and different I/O controllers (DDR PCI-E, network, etc.). These components implement specialized logic that would otherwise take up too much space if implemented using LUTs.

2.1.1 FPGA Architecture

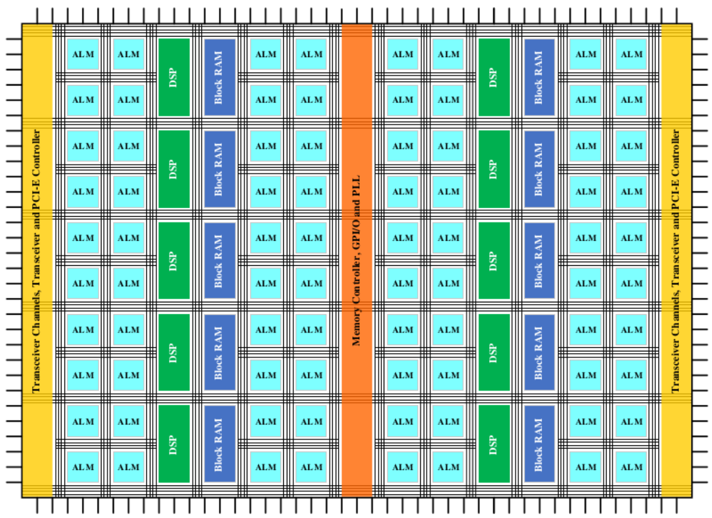


Figure 2.1 Arria10 GX FPGA architecture [30]

Figure. 2.1 presents the hardware architecture of the Intel Arria10 GX architecture. FPGAs consist of soft logic and hard logic. The soft logic inside Arria10 GX consists of Adaptive Logic Modules (ALM), and the hard logic consists of DSPs, Block RAMs, multiple controllers, Transceivers and Phase-Locked Loops (PLL). Each ALM consists of multiple-input LUTs, adders and carry logic, and registers (Flip-Flops). ALM contains various LUT-based resources that can be divided between two combinational adaptive LUTs (ALUTs) and four registers. With up to eight inputs for the two combinational ALUTs, one ALM can implement various combinations of two functions. Figure. 2.2 presents the internal structure of ALM. Multiply Accumulate (MAC) operations are crucial for most numerically compute-intensive tasks. DSP blocks are critical in performing these operations. Intel provides optimized variable precision DSP blocks to support higher bit precision in high-performance DSP applications. Each DSP can implement an IEEE-754-compliant [1] single-precision floating-point addition, multiplication, Fused Multiply and Add (FMA) operation, or one 27-bit-by-27-bit integer or fixed-point multiplication or two 18-bit fixed point multiplication/additions. Furthermore, multiple DSPs can be chained to implement dot products or other complex operations. Figure 2.3 gives the structure of the variable precision DSP block. BlockRAM (BRAM), also called embedded memory (EBR), is the on-chip memory on FPGA. Block RAMs come in a finite size usually, 4/8/16/32 kb (kilobits). They have a customizable bit width and depth. Each Block RAM in the Intel Arria 10 device, called an M20K block, is capable of storing a maximum of 20 Kbits of data.

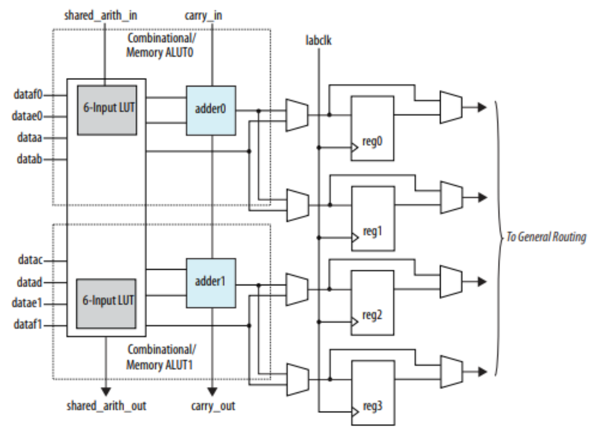


Figure 2.2 Structure of Adaptive Logic Module [30]

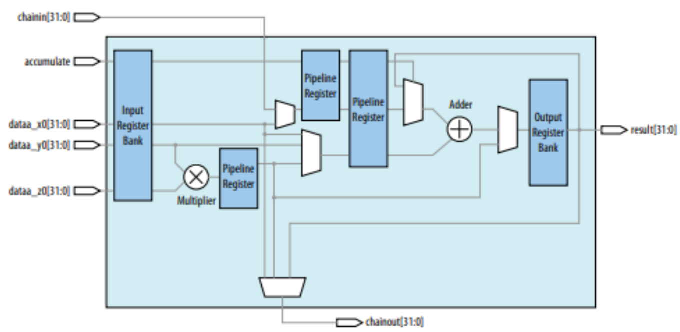


Figure 2.3 Structure of DSP block [30]

The BRAM has two ports that operate independently and can satisfy one read and one write operation simultaneously. Data can be stored in each block with a maximum width of 40 bits, in which case the address size will be 9 bits (512 addresses). Apart from implementing multiple-ported RAM or ROMs, each M20K can also be used to implement First-In, First-Out buffers (FIFO) or shift registers. Multiple M20K blocks can also be chained to implement larger buffers. The BRAMs can operate in Single-port or Dual-port configurations.

2.1.2 Logic Synthesis

Hardware Description languages (HDLs) define FPGA designs and are tool-independent. Logic synthesis converts a high-level description using HDL to an optimized gate-level netlist. The synthesis tools are part of the standard EDA toolchain for ASICs and FPGAs provided by the vendor. First, the hardware description is synthesized into a netlist. This step determines all coding errors. In the next step, the mapping process maps all functions in the netlist to functions available as the hard logic on the FPGA. Soft Logic (LUTs) implement other functions that cannot be synthesized. Further, the netlist undergoes placement and route (P & R). Place-and-route (P &R) describes several processes where the netlist elements are physically placed and mapped to the FPGA resources to create a bitstream file that can be downloaded into the FPGA chip. Placement will fail if a design requires more instances of a specific function than are available on the FPGA. In the next step, the routing process will determine routing resources and routes to meet timing constraints. Since routing resources are limited, routing could also fail in case of routing congestion. Finally, the tool generates the bitstream, and the device is programmed or flashed through the JTAG chain or PCIe interface.

2.2 Hardware-Software Co-Design with OpenCL

2.2.1 CPU-FPGA interconnect

Traditionally CPUs and FPGAs are coupled using bus interconnect. Figure. 2.4 presents the two possible bus interfaces in current CPU-FPGA heterogeneous systems used for hardware acceleration. Figure 2.4.a presents System-On-Chip(SOC) based system where CPU and FPGA are fabricated on the same chip, generally used in low-power embedded systems. In contrast, FPGAs can be connected with an external bus such as PCI-express(PCIe) with high bus transfer rates, typically for high bandwidth communication. These systems are usually used for high-performance computing in data centres and the cloud, with no power constraints. FPGA boards come with features like external memory like DDR4 or High Bandwidth Memory (HBM) and I/Os: PCIe, ethernet etc.

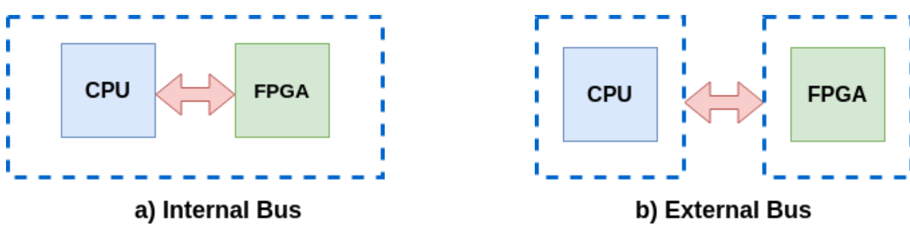


Figure 2.4 CPU-FPGA bus interface

2.2.2 OpenCL for CPU-FPGA Platforms

OpenCL is C based API that allows designers to perform computation on the host CPU, communication between the host and devices through Direct Memory Access (DMA) and computation on the accelerators (GPU, FPGA, etc.). Typically, HLS tools generate the datapath for the algorithm inside the FPGA but do not build the circuits for the interface between the algorithm (OpenCL kernel) and external memory (DDR), DDR and CPU. HLS with OpenCL solves these issues by providing an end-to-end solution. The OpenCL platform comprises the host (CPU) and accelerators (devices), and its execution model comprises two components: kernels and host programs. The host code orchestrates the computation on the device. Kernels are the executable programs on the device and can be data- or task-parallel, or deeply pipelined. A processing element is a unit of kernel execution.Figure. 2.5 depicts the abstraction of computation running on the OpenCL platform. A device can comprise multiple compute units with multiple processing elements running within. The host program executes on the host system, defines devices *context*, and queues kernel execution instances using *command queues*. Context provides the environment for host-device communication, memory management and device control. The host issues three types of commands: kernel execution, memory (*read*, *write*), and synchronisation. Kernels are queued in order but can be executed in order or out of order. OpenCL exploits parallel computation on compute devices by defining the problem into an N-dimensional index space. An index space is defined when a kernel is queued for execution by the host program.

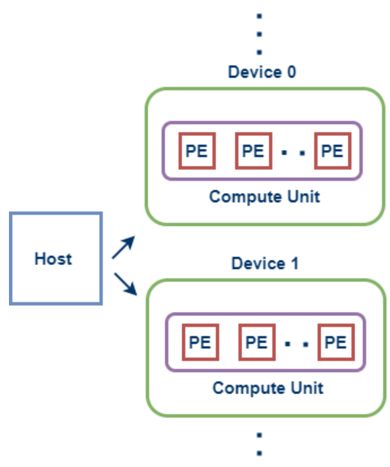


Figure 2.5 OpenCL Platform

Each independent element of execution in this index space is called a *work-item*. There are two variants of OpenCL kernels, NDRange (GPU-like SIMD) kernels and single-work-item (CPU-like task) kernels. NDRange kernels are defined by an N-dimensional index space, where multiple work items operate along the N dimensions, sharing the on-chip memory. Single work-item kernels share data among multiple loop iterations. Multiple work items can be grouped into *work-groups*. Figure 2.6 presents the OpenCL execution model. The host program is responsible for setting up the devices, program objects (usually bitstream or collection of kernels), and memory objects (memory buffers mapped common to the host and device).

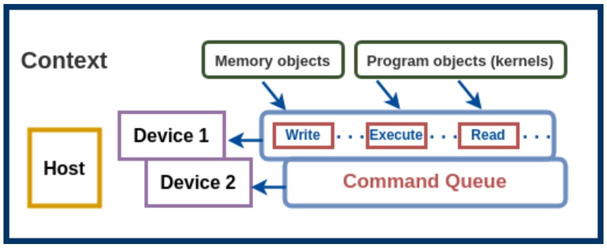


Figure 2.6 OpenCL execution model with Context and Command Queues

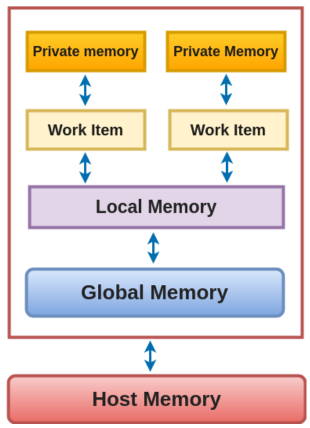


Figure 2.7 OpenCL memory model

2.2.3 Memory Model

OpenCL memory model defines four regions of memory accessible to work items when executing a kernel. Host memory is the memory of CPU.Figure 2.7 presents the OpenCL memory model.

- Global Memory: This memory space resides on the device's off-chip (external) memory. The content of this memory space is visible to all work-items of all workgroups. Global memory consistency is only guaranteed after a kernel is executed entirely. The host can allocate it only during run time.

- Local Memory: This memory space resides on the on-chip memory of the OpenCL device. Each work group has its own memory space and can be shared with its work items, and cannot be accessed by other work groups. Local memory is usually in the order of a few Megabytes, and its consistency is secured through barriers.

- Constant Memory: Region of global memory that stays constant throughout the execution. Work items have only read access to this region.

- Private Memory: Region of memory that is private to a work-item.

2.3 Overview of Intel FPGA SDK for OpenCL

OpenCL for FPGA platforms use board support packages (BSPs), custom-made or provided by the vendor. BSPs comprise IP cores that support interfaces with external memory controllers (DDR3/DDR4/ethernet), PCIe and enable DMA. In the compilation phase, the kernel partition is merged with the BSP partition, enabling kernels to access I/O. Since the kernel code is able to be detached from the BSP, it can be used for multiple boards provided by the vendor within some resource constraints. Intel FPGA SDK for OpenCL provides the necessary APIs and run-time to program and use PCIe-attached or SoC-FPGAs. Figure. 2.8 presents the BSP overview.

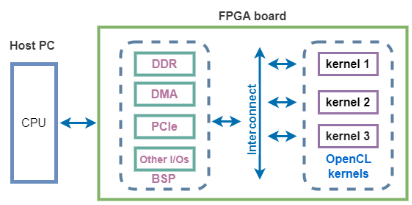


Figure 2.8 BSP overview

2.3.1 Intel FPGA SDK flow

Figure. 2.9 defines the compiler flow of Intel FPGA SDK for OpenCL. The host-side C compiler compiles your host program and links it to the Intel FPGA SDK for OpenCL runtime libraries. The host compiler is typically g++. Altera offline compiler (AOC) is the FPGA compiler that uses LLVM back end and is used to generate the bitstream (.aocx).

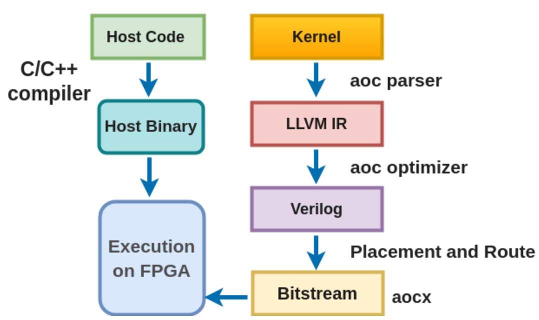


Figure 2.9 Intel FPGA SDK for OpenCL flow

2.3.2 OpenCL channels

The Intel FPGA SDK for OpenCL channels [31] extension allows kernels to communicate directly with each other through FIFO buffers. They can be used for communication and synchronization. Implementation of channels decouples data movement between concurrently executing kernels. Data written to a channel remains in a channel provided that the kernel program remains loaded on the FPGA device. In other words, data written to a channel persists across multiple workgroups and NDRange invocations.

2.3.3 Autorun Kernels

Autorun kernels [31] are special kernels used whenever we omit communication between the host and the kernel. As a result, the compiler need not generate the logic required for the communication between them, thereby reducing the logic utilization. Autorun kernel starts automatically without any explicit invocation by the host. And it restarts as soon as it finishes its execution. They are typically used whenever the kernel reads data from one or more kernel-to-kernel channels, processes the data and writes the results to one or more kernel-to-kernel channels.

2.4 HLS optimizations

Intel provides many standard HLS optimizations like loop unrolling, loop coalescing, loop fusion, disabling pipeline etc., that can be directed using #**pragma** directives. Loop fusion fuses adjacent loops of the iteration. By default, loops are inherently pipelined unless in the case of loop-carried dependency or data dependency from the previous iteration. Loop unrolling is used to increase the degree of spatial parallelism. Unrolling loops allow multiple data segments to process in one clock cycle in a SIMD fashion. Multiple hardware units are generated, thereby increasing resource utilization. By default, if we do not mention the unroll factor, loops are fully unrolled, which often does not allow the circuits to be synthesized, due to resource constraints. Loops can be unrolled by using an unrolling factor that defines the extent of parallelism. If we do not want loop unrolling, the #**pragma unroll** 1 is used. In FPGAs, unrolling is not just removing the loop index operations. It significantly changes the structure of a kernel by applying more parallel operations. As a result, although the performances are increased, the resource utilization can be substantially extended. The optimal solution can be obtained through design space exploration. Merging multiple memory access transactions into one is called memory access coalescing. Coalescing is required for efficient memory access. In addition, coalescing simplifies the datapath of the memory access, thereby reduce in resource utilization, and resulting in higher clock frequency. Coalescing is usually possible when there is no random access to off-chip memory. The #**pragma loop_coalesce** can be used to coalesce nested loops

Chapter 3

FPGA based Accelerator for Traffic Sign Recognition using Spatial Transformer Networks

3.1 Traffic Sign Recognition

The ultimate goal of autonomous vehicle research is to develop a fully automated system that would perform well in all kinds of scenarios. The system should understand the intricate traffic patterns and make real-time decisions based on the visual data from cameras and information from other sensors like LIDAR. Traffic sign recognition systems play a vital role in autonomous driving environments and for advanced driver assistance systems [49]. The traffic sign recognition problem involves two steps: traffic sign detection and Traffic Sign Classification. Traffic sign detection modules localize targets in the pictures, which is handled with computationally-inexpensive algorithms such as colour thresholding [37]. Traffic Sign Classification is a vital step in building traffic sign recognition (TSR) systems that employ vehicle-mounted cameras which identify traffic signs while driving on the road. Building a Convolutional Neural Network for the classification task would be an optimal approach to identifying the type of targets detected. Nevertheless, the classification task provides inherent challenges due to distortions caused by adverse variations like motion blurs, occlusions, and bad-view points. Their performance also gets affected even when the input has undergone affine transformations. Jaderberg [16] introduced Spatial Transformer Network, a learnable module that can be embedded into ConvNets, making them invariant to translation, scale, rotation, and warping. However, they are still limited by the inability to be spatially invariant to the input data. Spatial Transformers are learnable modules that, upon integration with CNN, would allow the spatial manipulation of data within the network, making it invariant to affine transformations. With specially designed hardware, FPGA-based neural network inference accelerators have shown promising results in achieving energy-efficient processing. We propose an OpenCL-based FPGA accelerator for Convolutional Neural Networks with a Spatial Transformer module for traffic sign classification using the German Traffic Sign Recognition Benchmark (GTSRB) dataset.

3.2 Spatial Transformer Networks

Initially, Convolution Neural Networks used in object classification could not be spatially invariant to the input data in a computationally efficient manner. Spatial transformer networks allow spatial data manipulation within the network. It is a differential module which applies a spatial transformation to a single feature map during a single forward pass, where transformation is conditioned to a particular input U, producing a warped single output feature map V. For multichannel inputs, similar warping is applied to each channel. This differentiable module can be inserted into the standard neural network architectures, allowing them to spatially transform feature maps without any extra training supervision or modification to the optimization process—the usage results in models which learn invariance to transition, scale, rotation and warping. Spatial transformers condition individual data samples with appropriate behaviour learnt during training. The module is a dynamic mechanism that can actively transform an image (or feature map) by producing appropriate transformations for each input. Notably, spatial transformers can be trained with standard back-propagation, allowing for end-to-end training of the models we inject. They achieve spatial invariance by adaptively transforming their input to a canonical, expected pose, thus leading to better classification performance. Figure 3.1 gives the architecture of the spatial transformer module.

The input feature map U is streamed to the localization network, which regresses the transformation parameters θ. Grid generator transforms the spatial grid G over output feature map V to sampling grid $\tau_\theta(G)$, that is applied to the input U to generate warped output feature map V. This combination of localization network and sampler defines a spatial transformer. Following sections elaborate on the internals of spatial transformers.

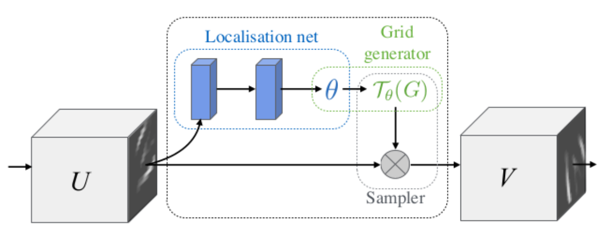


Figure 3.1 Spatial Transformer Networks

3.2.1 Localization Network

The localization network generates θ, the parameters of the transformation τ_θ, from the input feature map $U \in R^{HXWXC}$, where H, W, and C stand for the height, width and number of channels. The size of θ can vary based on the transformation type. For affine transformations θ is 6-dimensional. The localization network function $F_{loc}()$ can take any form, such as a fully-connected network or a convolutional network, but should include a final regression layer to produce the transformation parameters $\theta : \theta = F_{loc}(U)$.

3.2.2 Grid Generator

Grid Generator applies transformations τ_θ on regular grid G, a set of points with target coordinates. It warps the grid according to the affine transformation parameters θ. After the transformation $\tau_\theta(G)$, it outputs the coordinates to the sampler.

3.2.3 Sampler

Based on warped target coordinates received sampler projects the output feature map V onto a mesh grid. The spatial Transformer uses a bilinear transformation resampling technique.

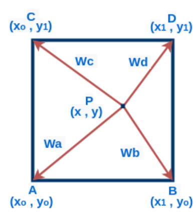


Figure 3.2 Bilinear Interpolation Technique

3.2.4 Bilinear Interpolation

Bilinear Interpolation based sampler is used to estimate intensity at the transformed location. It calculates the intensity as the weighted sum of weighted sum of the intensities of the four nearest pixels in the grid as shown in Figure 3.2. The approximation for output intensity I_{out} is given by the following equation

$$I_{out} = (w_a * I_a) + (w_b * I_b) + (w_c * I_c) + (w_d * I_d) \tag{3.1}$$

Where I_a, I_b, I_c, I_d are the pixel intensities at the 4 nearest locations in the grid. The values of weights are given by:

$$w_a = (x_1 - x) * (y_1 - y) \tag{3.2}$$

$$w_b = (x_1 - x) * (y - y_0) \tag{3.3}$$

$$w_c = (x - x_0) * (y_1 - y) \tag{3.4}$$

$$w_d = (x - x_0) * (y - y_0) \tag{3.5}$$

3.3 Model Architecture

In this work, we used a model [19] that consists of one spatial transformer module at the beginning of the network, which transforms the input image and forwards it to CNN for the classification task. Figure 3.3 presents the model architecture of the traffic sign classification model with a spatial transformer network. The localization network learns the transformation parameters which are to be applied to the input image. It can be another standalone CNN or Fully Connected Network (FCN), which takes in the feature map and outputs the transformation parameters. We have implemented a four-layered CNN, which consists of two convolutional layers, two fully connected layers and a max pooling layer in between the convolutional layers. Figure 3.4 represents the design of the Localization Network.

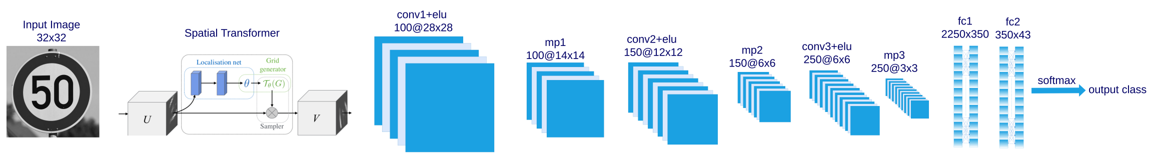


Figure 3.3 Model Architecture of the Traffic Sign Classification System

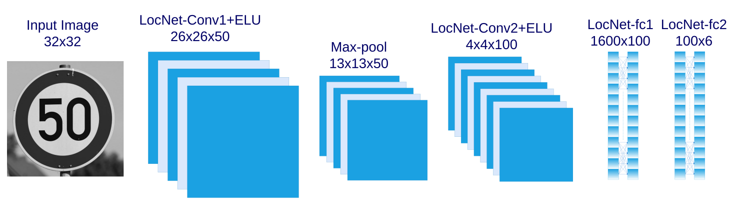


Figure 3.4 Localization network

Due to high throughput, reconfigurability, and energy efficiency, FPGA-based accelerators play a significant role in the inference of convolutional neural networks targeting embedded applications. FPGA implementation of Convolutional Neural Networks can be seen in [6, 47, 43] where various algorithms were used to perform convolution effectively. This paper presents an FPGA-based inference engine for CNN with Spatial Transformer Module for Traffic Sign Classification. Section. 3.4 presents the background and related work.

3.4 Related Work

With rapid advances in Deep Learning, Convolutional Neural Network-based classifiers have surpassed traditional learning methods for traffic sign classification. German Traffic Sign Recognition

Benchmark (GTSRB) [39] is one of the most popular and widely used datasets for testing and validating traffic sign classification algorithms. The dataset contains traffic sign samples with different resolutions and image distortions extracted from 1-second video sequences. The training set has 39,209 images, and the validation set consists of 12,630 images belonging to one of 43 existing classes. Ciregan et al. [9] proposed a Multi-Column Deep Neural Network comprising a committee of 25 CNN's achieved the highest accuracy of 99.46% in the GTSRB challenge. Sermanet et al. [38] proposed a multi-scale CNN and achieved an accuracy of 98.31%. Markl et al. [2] implemented a CNN for traffic sign classification based on Lenet [23] and attained an accuracy of 97.65 %.

Arcos-García Á et al. [50] proposed a traffic recognition system based on a Convolution Neural network that includes three Spatial Transformer modules. They obtained 99.71% accuracy for the GTSRB dataset. The spatial transformer module improves CNN's performance by making them invariant to translation, scale, rotation, and warping. We have used a model [19] that consists of one spatial transformer module at the beginning of the network, which transforms the input image and forwards it to CNN for the classification task. Reviewing the past literature, we have found out that FPGA implementations for traffic sign recognition were focused more on detection and classification using traditional methods. In [21], a binary neural network is used to classify only 10 classes of the GTSRB benchmark with 96.1% accuracy. In contrast, our implementation can classify images belonging to all 43 classes of the GTSRB data set with an accuracy of 99.2%. Usage of a Spatial Transformer with CNN enabled us to obtain a higher accuracy than the existing benchmark [26] deployed on Arria10 FPGA using the OpenVino toolkit, and also, there has been a significant reduction in the number of parameters making it more efficient to be implemented on embedded devices. Various methods have been proposed to effectively perform convolutions on FPGA. A scalable convolution block was implemented by mapping 3-D convolutions into matrix multiplication by flattening and rearranging the input features in [40]. An FPGA accelerator based on systolic arrays for implementing GEMM-based methods on FPGA can be seen in [48]. Kala et al [17] proposed a Generic Matrix Multiply GEMM-based accelerator based on Unified Winograd Algorithm. The im2col/im2row [7] is one of the GEMM-based methods for implementing convolutions and is widely used in deep-learning frameworks like Caffe, Theano, and Torch. The explicit im2row approach utilizes an extra memory, which is ($K^2 - 1$) times larger than the input, where K is the size of the filter, making it challenging to implement on FPGA due to on-chip memory constraints. To address this, we propose a channel-adaptive approach of im2row/im2col in order to handle the memory constraints and achieve high throughput. Section 3.5 gives the methodology followed in implementing the accelerator.

3.5 Methodology

The model uses three sets of CONV-ELU-MP layers for feature extraction for the transformed image obtained after passing the input image through the spatial transformer module. The output of the feature extractor is passed to fully connected layers and a soft-max wrapper to obtain the class of the input image. The convolutional layer is the most compute-intensive layer in the neural network. Accelerating the convolutional layer is crucial in building neural network accelerators.

3.5.1 Convolution

The convolution operation is essentially a 2D multiply-accumulate (MAC) operation that can be defined by

$$F(i,j) = \sum_{c=1}^{C} \sum_{k_x=0}^{k-1} \sum_{k_y=0}^{k-1} I_c(i-k_x, j-k_y) * K(k_x, k_y) \tag{3.6}$$

The input feature map of size (H_1, W_1, C) is fed into the convolutional layer with M filters, each of size (k,k,C) is used to generate an output feature map of size (H_2, W_2, M). All the convolutions are performed with a stride of one and with *valid* padding. A great majority of computations in CNN's are from the 2D convolutional layers. Various methods have been proposed to efficiently perform convolutions in-order to reduce the compute time by exploiting the parallelism provided by various hardware accelerators like GPUs and FPGA's. General Matrix Multiply (GEMM) based methods are widely used to perform convolutions due to their ease of parallelization. Many deep learning frameworks such as Pytorch, Theano, and Caffe also use GEMM based approaches like im2row, im2col to perform Multi-Channel Multi-Kernel (MCMK) convolution on GPU. The key idea in GEMM based methods is to express 2D convolution as matrix multiplication. We followed an im2row based GEMM method in our implementation.

3.5.2 IM2ROW:

The im2row stands for *image to row*, which reflects the process of reshaping the image tensor into a matrix form that can be multiplied with the filter. In im2row, the filter kernel is slid over the input feature map, extracting a patch of pixels that defines the spatial extent of the input tensor used to compute the activation of a neuron in the output feature map. These patches of pixels are transformed into row vectors stacked to form an input patch matrix, which has to be multiplied with the filter patch matrix to generate the output matrix. For example, if an input feature map of dimensions (H x W x C) is to be convolved with M filters of size (k x k x C) where C is the number of channels, with stride 's', we would first need to compute the number of patches ($num\_patches$) that can be extracted from the input tensor given by $(H-k+1)/s*(W-k+1)/s$. Then the input patch matrix will be formed with dimensions ($num\_patches$, (k x k x C)) by stacking the row vectors vertically. To compute the output of the convolution operation, we would reshape the filter tensor into a filter patch matrix with dimensions ((k x k x C) , M). Then matrix multiplication will generate an output matrix with dimensions ($num\_patches$, M) that can be reshaped to the desired form. The im2row method can be computationally efficient for small filter sizes and large input tensors, as it allows for efficient matrix multiplication operations instead of expensive convolution operations. However, it may be less efficient for larger filter sizes, as the resulting patch matrix can be huge and may require a large amount of memory to store. So we slightly modified im2row and designed channel adaptive-im2row, where the patch matrices are tiled channel-wise, to support low-memory devices.

3.5.3 Our Algorithm

- Create an input patch lookup matrix of size ($num\_patches$, (k x k)) which contains the lookup addresses for a patch matrix corresponding to one channel.

- Create filter patch matrix corresponding to filters belonging to the channel with dimensions ((k x k), M).

- Perform GEMM operation with the given feature map with the addresses from patch lookup matrix and filter patch matrix.

- Repeat the process for all channels.

3.6 Hardware Architecture

This section presents the overall hardware architecture as shown in Figure. 3.5. The hardware architecture consists of 4 major modules: The transformer module, which performs localization, transformation, and bilinear interpolation. The convolution engine performs convolution, max-pooling and ELU activation. The Batch normalizer normalizes convolution output, and Fully connected layers perform the matrix-vector multiplication using loop tiling. The data is transferred from the host CPU to DDR using the PCIe interface. The on-chip memory and DDR are interfaced with Avalon Memory Mapped interconnect (Avalon MM). We implement the Convolution engine, Spatial Transformer module, batch normalizer and FC module as independent OpenCL kernels, which we schedule using a host driver based on the network architecture.

3.6.1 Spatial Transformer Module

The Localization net consists of 2 convolution kernels and 2 Fully Connected kernels. It outputs the transformation parameter theta, which is placed in an OpenCL pipe. The Intel FPGA SDK for OpenCL provides channel extension as an abstraction of OpenCL pipes for passing data between kernels and synchronizing kernels with high efficiency and low latency. Two separate channels were used to pass the image and theta to the transformer module, which applies the transformation to the input image with the value of theta. The bilinear sampling kernel is integrated with the transformer module to compute intensities at the transformed locations.

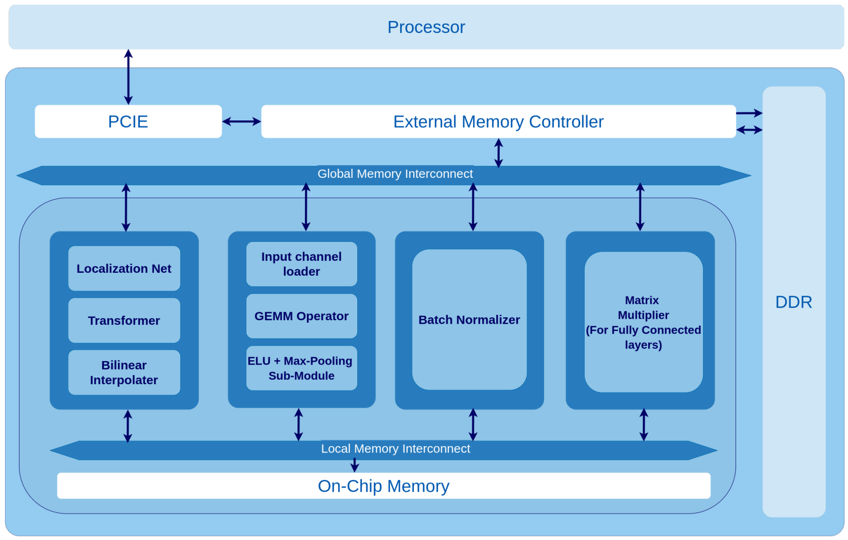


Figure 3.5 Hardware Architecture of the accelerator

3.6.2 Bilinear Sampling Kernel

After obtaining the transformed coordinates, we pass them through the nearest neighbour calculator, which calculates the four nearest integer coordinates for the given point. Weights are calculated based on the distance of the given point from its neighbours. We estimate the output pixel value by performing MACC operation with these weights and intensities. The transformed image is sent to the classifier using the channel.

3.6.3 Convolution Engine

The convolution engine has a direct interface with global memory where all the input feature maps are stored. Upon invoking the kernel, we create an address-lookup matrix containing the addresses of data corresponding to one channel's patch matrix in the input feature map. The same lookup matrix is reused with a load of new channel data to create the corresponding patch matrix. This method reduces the computational cost of executing multiple transformations to generate the patch matrix for the following channel load. Results from each iteration are stored in the local memory and accumulated over the total number of channels in the input feature map. We apply ELU activation to the output. To perform max-pooling, we similarly create an address-lookup matrix for convolution that stores the addresses corresponding to one pooling operation for a kernel of size 2 and stride 2. We create this to parallelize various pool operations, as each row of the lookup matrix contains data that can be processed concurrently. We use a tree-based sorter to find the maximum of four elements, as shown in Figure. 3.6.

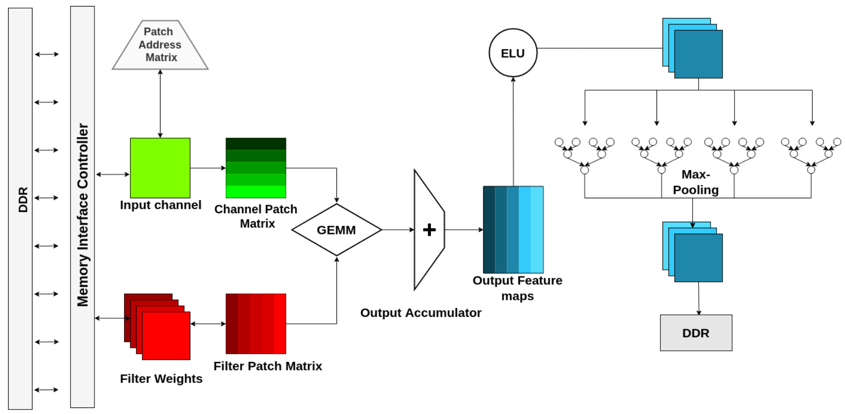


Figure 3.6 Hardware Architecture of Convolution Engine

3.6.4 Matrix Multiplier Engine for FC layers

The hardware architecture implements the Fully Connected (FC) Layer as a matrix multiplication problem. The output of the feature extractor (Conv) is flattened and multiplied by the weight matrix of the fully connected layer. The data is streamed from global memory in tiles one after the other, and the results from these computations accumulate to generate the output. For an input of size (M, N) and (N,1), data is loaded in chunks of size (M_1, N_1) and $(N_1, 1)$ where M_1 and N_1 represent the size of tile loaded. We used NDRange kernels for matrix multiplication, which are suitable for SIMD processing. The output size defines the total number of work items (global size). The work items are organized into work groups, where a work item represents a single execution thread of the kernel.

3.7 Experimental Setup and Results

The accelerator design is implemented using Intel FPGA SDK for OpenCL. The built ConvNet-STN was trained using PyTorch for 40epochs on Nvidia GeForce GTX 1080 Ti GPU. We used the Intel Arria10GX FPGA development kit shown in the Figure 3.7 with 2GB DDR4 for our experiments. The specifications of the device are presented in Table 3.1. The Host-machine is equipped with Intel Xeon CPU E5- 2630 v4 CPU with 64GB DDR4-2133 SDRAM runs RedHat Linux 7.5. We compare the result with the output obtained from the PyTorch model to verify the functional correctness. The GTSRB dataset is used for evaluation, and we achieved 99.5 % accuracy. Host interface schedules each layer sequentially. Table 3.2 gives the scheduled execution cycles for each layer.

The Intel FPGA dynamic profiler [31] for OpenCL uses performance counters to collect kernel performance data during the design's execution. The Profiler instruments and connects performance counters in a daisy chain throughout the pipeline generated for the kernel program. The host then reads the data collected by these counters. Figure 3.8 presents the system latency captured using the Intel dynamic profiler. Table 3.3 gives the resource utilization for the network.

Figure 3.7 Intel Arria10 GX FPGA development kit

Table 3.1 Specifications of Arria 10GX 1150

Logic Elements	854k
DSP blocks	1518
M20K RAM	2713
External memory	2 GB DDR4

Table 3.2 Execution cycles for each layer of the network

Module	Execution Cycles
LocNet-Conv1	398
LocNet-Conv2	499
LocNet-FC1	375
LocNet-FC2	305
STN-Transformer	385
Conv1+MP+ELU	400
Batch Norm1	75
Conv2+MP+ELU	575
Batch Norm2	75
Conv3	522
Batch Norm3	75
FC3	380
Batch Norm4	75
FC4+Softmax	279

Table 3.3 Resource Utilization

Frequency	202MHz
Logic used	21%
DSP	8%
BRAM	87%

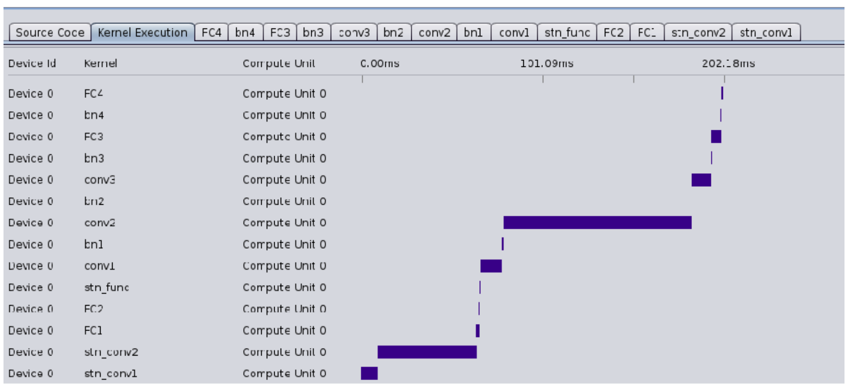


Figure 3.8 System Latency of STN-ConvNet

Table 3.4 Comparision with CPU

Accelerator	Latency
STN-CNN (20nm)	202 ms
CPU (40 core)	1.12 sec

Compared to CPU implementation, the design attains a speedup of ($>$ 5X). The latency obtained is not state-of-the-art. This is due to the inefficient use of available DSP resources. Another design bottleneck is the formation is Toeplitz matrices. This involves random access to external memory and causes an off-chip bandwidth constraint for the schedule generated. Existing GEMM-based methods are mainly focused on accelerating matrix multiplication using systolic arrays, loop-tiling etc. Even though kn2row-acc [41] could address the extra memory issue concerned with im2col, it suffers from off-chip bandwidth constraints. The performance of all the GEMM-based methods are generally context-dependent, with methods having excellent performance in some contexts and poor performance in others [48]. Hence these GEMM approaches can be ideal for SIMD devices like GPUs, where multiplications can be hugely parallelized in a vector fashion. FPGAs usually generate deeper pipelines, so the approaches of GPUs may only be optimal in some contexts.

3.8 Conclusion

This project explores High-Level Synthesis and observes the behavioural patterns of code generated in developing Convolution neural networks. It uses a modified GEMM approach *im2row* that is typically used in GPUs and Embedded CPUs for convolution. We synthesize a traffic sign classification neural network with a spatial transformer network. The system latency is 202 ms, around 5 fps, which usually makes it suitable for real-time implementations. However, the traffic sign classification task is generally accompanied by traffic sign detection for traffic sign recognition. Even though we attained a considerable throughput compared to a CPU ($> 5X$), the performance can be improved further. Hence, the project further explores FPGA-specific optimizations like deeper pipelining and systolic arrays with increased spatial parallelism, efficient DSP usage, and off-chip memory bandwidth.

Chapter 4

Systolic Array based FPGA Accelerator for YOLOv3-tiny

4.1 Introduction

Object detection is an important area of research in computer vision. It is widely used in aerospace, robotics, video surveillance, industrial detection, autonomous driving, etc. because it significantly reduces human efforts by detecting, locating, identifying, and classifying target objects. Through continuous effort in research, deep learning algorithms are proliferating with improved object detection performance. There are two types of object detection algorithms. Object detection algorithms using region proposal include RCNN [13], Fast RCNN [12], and Faster RCNN [36]. Two-stage detectors provide adequate accuracy but come with high computational latency. Therefore, one-stage detectors are proposed to process in less time by managing sufficient accuracy. Typical one-stage object detection models are Single Shot Multi-box Detector (SSD) [27], YOLO [33], and its successors YOLOv2, YOLOv3, YOLOv4. YOLO models concurrently predict bounding box coordinates and associated class probabilities without a complex pipeline resulting in high efficiency. YOLO, an accurate end-to-end algorithm, uses a convolutional neural network structure which inputs a complete image and directly outputs the target's location box position and category. It divides the image into a certain number of grids and predicts the target for each grid. YOLOv2 [34] introduces a new backbone called DarkNet-19 that is pre-trained on ImageNet and removes the fully connected layer and the last pooling layer in YOLO, which performs faster, achieves higher accuracy and recognizes more categories (more than 9,000). Besides, it uses anchors obtained by K-means clustering to predict bounding boxes for the first time. YOLOv3 [35] design a new backbone of DarkNet-53 and combines the Feature Pyramid Network (FPN) [25] to achieve better detection accuracy of small objects. YOLOv3-tiny is a lightweight version of YOLOv3 with fewer layers to optimize for edge computing applications. This work addresses the challenge of deploying multiple-precision YOLOv3-tiny implementations on an FPGA device, making it suitable for the data centre and edge applications. One of the first works related to YOLOv3-tiny uses a hardware-software co-design approach [4] , with convolutions handled by a parallel pipelined hardware design and the rest of the layers processed by the Microblaze softcore processor. However, it needs a large number of multiplexers to collect the output, which increases the fan-out. The study in

[3] expresses convolution as matrix multiplication using the *im2col* Generic Matrix Multiply (GEMM) approach; however, the design suffers from random memory accesses to transform the input and filter data into matrix forms. The authors in [46] propose a parametric architecture which performs computations concurrently within a layer batch, with parallelism along input, output channels and filter kernel's surface. However, the design suffers from low latency ($<$ 2 fps), which doesn't make it suitable for real-world applications. In this work, we adopt a homogeneous 1-D systolic array architecture for convolution to reduce the global interconnect and the usage of large multiplexers, thereby minimizing the data movements to attain high throughput. The key contributions of this work are as follows.

1. We propose a deeply pipelined 1D Systolic array accelerator with a novel load pattern for accelerating the convolution of YOLOv3-tiny to reduce global interconnects and large multiplexers, thereby reducing data movements to obtain high throughputs.

2. The design exploits 3D spatial parallelism using specialized MAC tree architecture, allowing multiplications to run synchronously with a pipelined adder tree. We use the Intel OpenCL framework for the architecture design, which is scalable for multiple variants of YOLO.

3. We evaluated the design on the Terasic DE5anet-DDR4 FPGA for multiple precisions (FIXED-8, FIXED-16, FLOAT32). While running YOLOv3-tiny for 416x416 RGB image, the fixed point (FP-8) attains a throughput of 57 GOPs/s with a framerate of 10.2 fps running at 234 MHz. Fixed point (FP-16) attains 46.16 GOPs/s with a framerate of 8.278 fps, running at 227.78 MHz. The floating-point (FLOAT32) design achieves 11.22 GFLOPs/s running at 172.92 MHz.

4.2 Background and Relevant Work

Convolutional Neural Networks are a vital class of deep neural networks most commonly applied to analyze visual imagery. In many applications, it is desirable to have the CNN inference processing at the edge near the sensor. FPGAs play a significant role in these implementations because of the high performance per watt they can deliver. The convolutional layers dominate the computation and storage in a CNN model. Therefore, the performance given by the architectural components associated with the convolutional layers majorly influences the overall performance of a CNN inference accelerator.

The input feature map IF_{map} of a convolution layer is a cuboid with dimensions $\langle H, H, N_d \rangle$ where N_d is the number of channels depth-wise. The dimensions of each channel are $H \times H$. The convolution layer consists of applying N_f filters each of dimension $\langle K, K, N_d \rangle$. This results in generating an output feature map OF_{map} which is again a cuboid of dimensions $\langle H - K + 1, H - K + 1, N_f \rangle$. The output pixel in the position (p, q) of the n^{th} channel, denoted by $OF_{map}[n, p, q]$ is defined as follows.

$$OF_{map}[n, p, q] = \sum_{d=1}^{N_d} \sum_{r=1}^{K} \sum_{c=1}^{K} IF_{map}[d, p + r, q + c] * F[n, d, r, c] \tag{4.1}$$

where $F[n, d, r, c]$ denotes the $(r, c)^{th}$ filter coefficient from the depth channel d in filter F_n. It can be noted that computing an output pixel is nothing but dot product involving Multiply-And-Accumulate (MAC) operations. Many studies [6, 44, 11, 48, 40, 42] have proposed FPGA based CNN accelerators due to the increased customizability and reconfigurability of FPGAs. Early works [6] exploited the parallelism in computation to utilise the abundant DSP resources on FPGA. Afterwards, the focus shifted to transforming memory bound convolutional kernels to compute bound by increasing data reuse, thereby reducing the traffic to the external memory [40, 42]. For achieving higher computer throughput, along with data reuse, the corresponding data fan-out increases. This results in congestion in routing paths and adversely affects the clock speed. Homogeneous systolic array architectures [44, 11, 48] with a standard layout, low global data transfer and high clock frequency enable us to address such challenges, making them suitable for large-scale parallel design on FPGAs. In this work, we present a deeply pipelined 1D-systolic array architecture for YOLOv3-tiny CNN, with a novel load pattern enabling us to exploit three-dimensional spatial parallelism.

4.2.1 Review of YOLOv3-tiny

YOLOv3-tiny is a faster and lighter version of YOLOv3 with fewer layers, allowing its deployment to resource-constrained devices. The reduced computational load comes with a penalty on the object detection precision. The original backbone network of YOLOv3 is Darknet53. Darknet-53 includes 52 fully convolution layers, in which 46 layers are divided into 23 residual units with 5 different sizes. The residual units are designed to avoid the vanishing-gradient problem inspired by the Resnet [15].

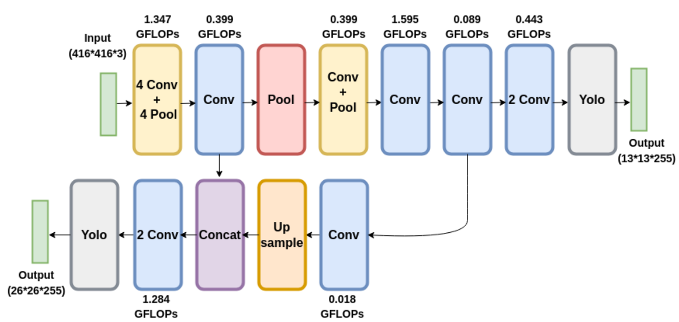


Figure 4.1 YOLOv3-tiny architecture

YOLOv3-tiny is a simplified version of YOLOv3. Its backbone network only includes 7 convolutional layers and 6 max-pooling layers. Out of 3 branches in the FPN, one branch (maximum scale prediction) is removed, and the number of convolutional layers in the other 2 branches is reduced. YOLOv3-tiny accepts an RGB image of 416 × 416 resolution as input and predicts bounding boxes at two feature map scales (13x13, 26x26). The framework divides the input image into 13x13 and 26x26 grid cells and generates outputs for three anchors in each cell. The network outputs a 3D tensor containing information on the bounding box dimensions, objectness confidence and class predictions for each scale. The network constitutes convolutional, max pooling, concatenation, upsampling and Yolo layers. Figure 4.1 gives the data flow of Yolov3-tiny architecture along with the floating-point operations needed at each convolutional layer.

4.2.2 Previous Research

Many studies have focused on customised designs for deploying various versions of the YOLO network onto FPGAs. Some perform an accurate mapping of YOLO, while others introduce certain approximations to tailor the hardware. The authors in [28] presented an FPGA architecture for YOLOv2-tiny and attained an inference throughput of 19 FPS in a Zynq 7035 FPGA. The study in [45] uses 16-bit Fixed-point representation for inputs and outputs of YOLOv2-tiny attained 69FPS using Arria10GX 1150 FPGA. The implementation in [4] uses hardware-software co-design for accelerating YOLOv3-tiny. Only the convolutions are handled in hardware, and all the other layers and activation functions are implemented in software. The architecture uses an 18-bit Fixed-point data type to allow one multiplication per clock cycle per DSP. It attains high throughput for the convolution on Virtex 7 VC707 using 2304 DSP blocks. However, there is no information on the overall latency and throughput for all the layers. The authors in [3] adapt the Generic Matrix Multiply (GEMM) based approach for accelerating convolutional layers of YOLOv3-tiny. They used low precision 8-bit Fixed-Point representation and attained a throughput of 31.50 GOPs on the Zynq ZU3EG device for all the layers. The study in [46] implements a parametrized FPGA-tailored architecture specifically for YOLOv3-tiny. The architecture exploits parallelism along the input channel, output channel, and filter kernel's surface dimensions. It uses a 16-bit Fixed Point precision and attained a throughput of 10.45 GOPs on the Zynq 7000 Soc device with a frame rate of 1.88 fps.

4.3 Analytical Modelling

In this section, we devise the strategy for mapping the CNN onto a resource-constrained FPGA device and detail the methodology followed to extract parallelism, improve data reuse and thus achieve overall performance measured in terms of GFlops/second, along with other metrics.

4.3.1 Architecture Abstraction

There are several properties of CNNs that the hardware can leverage to optimize, thus improving the throughput and design efficiency. The fundamental property here is that the computations in CNN involve many MAC operations which have no restriction on their order of execution. Another essential property that addresses the data movement and storage costs is that the same piece of data is often used for multiple MAC operations. Further, the $OF_{map}[i]$ corresponding to the i^{th} filter F_i is equivalent to the summation of 2D convolutions of IF_{map} with the corresponding depth slices of F_i. This can be expressed as

$$OF_{map}[i] = \sum_{d=0}^{N_d-1} IF_{map}[d] \circledast F_i[d] \tag{4.2}$$

Data Reuse: For convolutional layers, the filter surface $\langle K \times K \rangle$ is smaller than the size of the input feature map surface $\langle H \times H \rangle$, and the filter slides across different positions in the input feature map to generate an output feature map. As a result, each filter weight and input activation are further reused $(H - K + 1)^2$ and K^2 times respectively.

In our proposed methodology, we extracted three forms of parallelism from these properties.

1. **Depthwise Parallelism** $\langle n_d \rangle$: From the above equation (4.2), the depth slices of the filter and input feature map can be independently convolved, enabling channel parallelism. Let the parameter n_d denote the number of 2D convolutions computed parallelly across the channels of IF_{map}.

2. **Filter Parallelism** $\langle n_f \rangle$: We define the filter parallelism n_f as the number of output depth channels from OF_{map} that can be generated in parallel corresponding to the same input.

3. **Filter Reuse** $\langle FR \rangle$: As mentioned above, the filter weight is reused $(H - K + 1)^2$ to compute a surface of an output feature map. We define the parameter FR, $0 \leq FR \leq (H - K + 1)^2$ as the number of times the filter weight is reused to generate FR elements of output feature map concurrently. Let x_{par} and y_{par} denote the parallelism along X and Y directions of the surface of the output feature map where $FR = x_{par} \times y_{par}$. In our methodology, we chose $y_{par} = 1$ giving rise to $FR = x_{par}$, the number of elements which are computed parallelly in the X-direction of the output feature map.

Methodology – Three Dimensional Tiling:

The convolutional layer is optimized with three modelling parameters n_d, n_f and x_{par}. The output computation has been split into $\frac{N_f}{n_f}$ batches which exploits filter parallelism. To generate x_{par} outputs within a OF_{map} depth channel in parallel, an input feature map sub-volume of dimensions $(K + x_{par} - 1, K, N_d)$ needs to be convolved with N_f filters each of dimension (K, K, N_d). Each of the $x_{par} \times n_f$ convolutions are computed in $\frac{N_d}{n_d}$ stages. Thus the number of parallel multiplication operations in each clock cycle will be $n_f \times x_{par} \times n_d$.This leads to higher DSP utilization. Overall this strategy is nothing but three-dimensional volume tiling.We present the pseudocodes $\langle Algorithm.1, Algorithm.2 \rangle$ which map our methodology to the HLS framework. The former gives the inner loops of the HLS pseudocode, which map to the computational units. The latter partitions the whole input volume of IF_{map} and filters into batches, and feeds data to the computational units. Loops L1, L2, and L3 are fully unrolled, asserting 3-D spatial parallelism.

4.3.2 Resource Utilisation Model

Since the computation is mainly floating-point MAC operations, the DSP blocks and on-chip Block RAM (BRAM) are the two critical resource types. $x_{par} \times n_d$ multiplications are performed concurrently corresponding to each filter F_i. n_f filters are processed parallelly, giving rise to a total of $n_f \times n_d \times x_{par}$ parallel multiplications that are accumulated channel-wise to generate $n_f \times x_{par}$ outputs. If the number of DSPs on the FPGA are N_{DSP}, then we get the following constraint.

$$n_f \times n_d \times x_{par} < N_{DSP} \tag{4.3}$$

We use separate OpenCL kernels for data loading and computation, connected using blocking FIFOs (OpenCL channels). Data is prefetched by the loader kernels, while computation is performed in the processing elements, and FIFOs handle synchronization between them. Hence to enable this, twice the size of input memory (feature map, filter) needed for computation needs to be buffered in BRAM. To generate $x_{par} \times n_f$ outputs per clock cycle, $x_{par} \times n_d$ feature map data is convolved with $n_f \times n_d$ elements of filter data. The constraint model on BRAM can be given by (4.4), where N_{BRAM} gives the total number of available BRAMs.

$$2 \times (x_{par} \times n_d + n_f \times n_d) + x_{par} \times n_f < N_{BRAM} \tag{4.4}$$

Algorithm 1 Outer Loops of HLS Pseudocode

```
//no. of filter batches
for (0 ≤ n_1 < N_f/n_f) do
    // no. of channel batches
    for (0 ≤ d_1 < N_d/n_d) do
        //no. of segments along X
        for (0 ≤ p_1 < (H − K + 1)/x_par) do
            //no. of segments along Y
            for (0 ≤ q < (H − K + 1)) do
                //along filter row
                for (0 ≤ r < K) do
                    //along filter col
                    for (0 ≤ c < K) do
                        < Inner Loops of HLS code >
                    end
                end
            end
        end
    end
end
```

Algorithm 2 Inner Loops of HLS Pseudocode

```
#pragma unroll                                              ▷ L1: n_f filters in parallel
for (0 ≤ n_2 < n_f) do
    #pragma unroll                                          ▷ L2: n_d elements in parallel
    for (0 ≤ d_2 < n_d) do
        #pragma unroll                                      ▷ L3: x_par outputs in parallel
        for (0 ≤ p_2 < x_par) do
            output[n_1*n_f + n_2, p_1 * x_par + p_2, q] +=
            input[d_1*n_d + d_2, p_1 * x_par + p_2 + r, q + c] * filter[n_1*n_f + n_2, d_1 * n_d + d_2, r, c]
        end
    end
end
```

4.4 Micro Architecture

In this section, we present the hardware architecture of the accelerator. The proposed architecture is tailored to execute YOLOv3-tiny DNN, providing architectural support for accelerating the YOLO layer. The throughput is an essential performance indicator in accelerator design. There are multiple methods to improve the throughput

1. Exploiting parallelism in computation to utilise abundant logic resources.
2. Minimising data movements for efficient use of bandwidth.
3. Increasing the operating frequency of the overall system.

Exhaustive reuse and interaction between DSP blocks increase the fan-out. Also, large multiplexes are needed to collect the output. Adapting a homogeneous systolic array architecture can solve some of these issues. The global and large fan-out interconnect is split into local interconnects between neighbouring Processing Elements (PEs). With small interconnects, systolic arrays can attain high frequency with massive parallelisation.

4.4.1 Mapping to Systolic Array

We propose a deeply pipelined 1-D systolic array architecture to accelerate the CNN on FPGA. Figure 4.2 provides a high-level picture of the proposed architecture. The accelerator consists of data load/store units: Feature Map Read unit, Filter Fetch Unit for loading the input feature map (IF_{map}) data, filter data from external DDR memory, and Feature Map Write unit for writing back the results to DDR. The core of our architecture is a 1-D systolic processing element (PE) array that performs MAC operations in each PE. The Feature Map Read unit caches the IF_{map} in a shift register-based buffer and streams it to the first PE using the OpenCL channel (FIFO). Each PE receives different weights

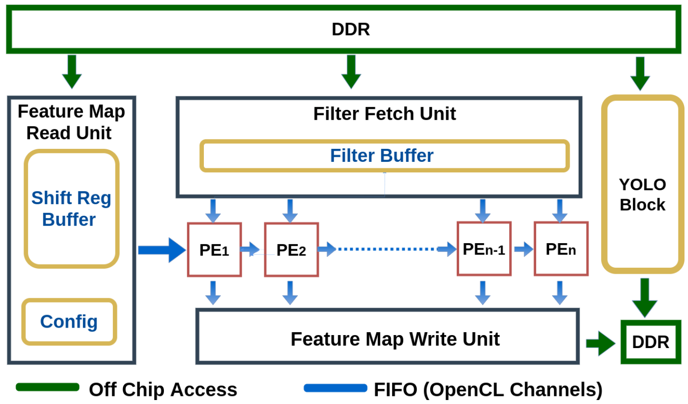


Figure 4.2 Hardware Architecture of the accelerator.

from the Filter Fetch Unit and IF_{map} data from adjacent PE. The partial OF_{map} rendered by the PE are accumulated and cached in its internal registers.

The *MAC-tree* engine within the PE uses this memory to sum up all the partial OF_{maps} corresponding to the input volume channels to produce the output feature map. Once the results are generated, data is sent to the Feature Map Write unit, which writes the generated OF_{maps} to off-chip memory. In the following section, we discuss the architectural internals of the accelerator.

4.4.2 Internals of the Architecture

4.4.2.1 Feature map Read Unit (FRU)

The FRU is responsible for fetching the IF_{map} data from the off-chip memory. The overall computation is performed block by block sequentially using loop tiling optimization. The input feature map is segmented blockwise, each block comprising x_{par} vectors. We fetch the data block from off-chip memory and cache it in a shift register-based buffer. Figure 4.3 gives the loading scheme of IF_{map} vectors. To generate x_{par} outputs, $(x_{par} + K - 1) \times K$ input vectors needs to be convolved with $K \times K$ filter vectors. x_{par} vectors are loaded in the first cycle, and the inputs are shifted K times, with the loading of a new vector for the next K cycles. Depthwise channel parallelism is given by n_d and thereby, the size of an input vector is given by n_d. Every clock cycle, x_{par} vectors are streamed to the first PE. The size of shift register $shift\_reg\_size$ is given by

$$shift\_reg\_size = x_{par} \times n_d \tag{4.5}$$

After computation of the current block of vectors, the load window propagates along the channel dimension. Once we exhaust the first batch of n_f filters, we process the next batch of filters. Further, the load window propagates along surface dimensions to process the entire IF_{map}. With this implementation, we can reuse all the elements in the buffer till we flush them out. It also eliminates the need

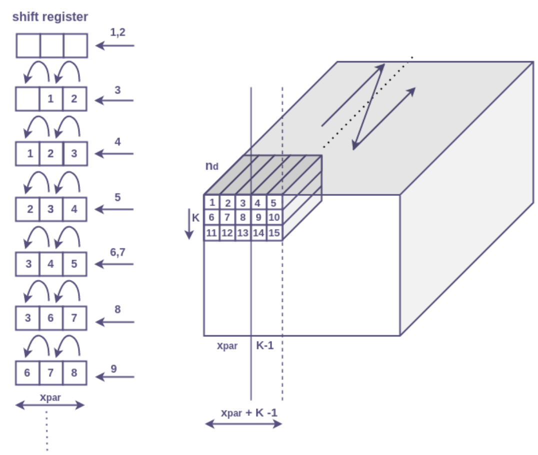


Figure 4.3 Data loading strategy of IF_{map} with $x_{par} = 3, K = 3$

for using wide multiplexers to feed the data into PEs and significantly simplifies the interconnections between the memory, reducing the critical path delay and improving the clock frequency.

4.4.2.2 Filter Fetch Unit (FFU)

The Filter Fetch Unit (FFU) is responsible for fetching the weight data from external memory to each processing element (PE). In the proposed architecture, the filter parallelism is n_f, i.e. n_f filters are processed at a time. Each PE processes a filter, which implies that the total number of processing elements will be n_f. Each PE receives a weight vector of dimension n_d from the FFU via a FIFO (OpenCL Channel). The data loaders FRU and FFU run concurrently, sharing the same configuration, enabling the suitable weight vector to load for convolution with the corresponding input.

4.4.2.3 Processing Element

The Processing Element (PE) forms the core of our architecture,which carries out the convolution. Figure 3.4 shows an overview of single PE hardware. The PE is a fully pipelined structure. It consists of *MAC-tree* engines for performing MAC operations on input vectors and registers for storing partial OF_{map} sums. In our proposed architecture, x_{par} number of OF_{maps} are processed at a time within a PE and reuse the same filter vector. Hence x_{par} number of *MAC-tree* engines are present within a PE, and each *MAC-tree* processes a vector of channel parallelism factor (n_d) elements at a time. Hereby each *MAC-tree* engine consists of n_d MAC units, that process MAC operations on shifted inputs. Further, we perform depth-wise accumulations using a pipelined adder tree as given in the Figure 4.4. Once we generate the OF_{maps}, we flush the outputs to Feature Map Write Unit (FWU) using channels.

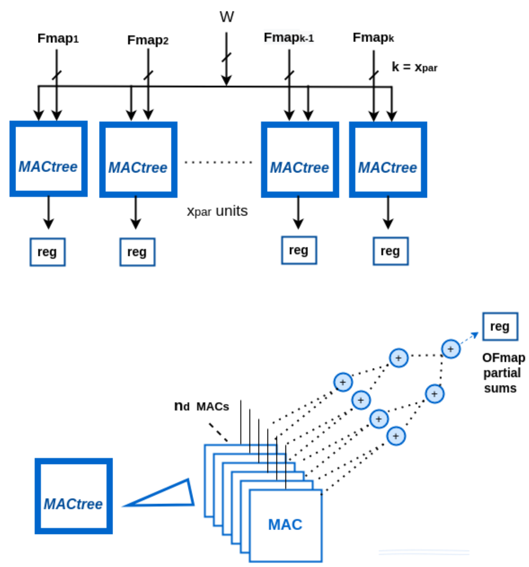


Figure 4.4 Overview of the Processing Element (PE) Architecture.

4.4.2.4 Feature map Write Unit (FWU)

The FWU is responsible for writing the generated OF_{maps} to off-chip memory. The Batch Normalization (BN) layer follows a convolutional layer that requires complicated floating-point arithmetic operations and consumes many logic and DSP resources. This paper follows a BN folding technique, where we fuse the BN and bias addition giving rise to updated weights and bias in the training phase. We carry out the bias addition process in the FWU. Before feeding the results to the off-chip memory, an activation function Leaky Rectified Linear Unit (ReLU), with a negative slope of 0.1, is applied to each pixel of OF_{map}.

4.4.2.5 YOLO block

The YOLO block implements the functionality of the YOLO layer, which is majorly composed of sigmoid activation. It fetches the outputs of both the 13×13 and 26×26 grids from external memory. Our architecture uses piece-wise linear approximation based on curvature analysis [24] for the sigmoid function to avoid expensive floating-point DSPs for synthesizing division operations and exponent module. The decoded information of bounding box coordinates, confidence values and class probabilities for all the three anchors is written to DDR.

4.4.2.6 Additional Units

The max-pooling, upsample and concatenation units are designed as independent OpenCL kernels scheduled by the host driver. The Maxpool Unit fetches IF_{map} data using FRU through FIFOs. The other two fetch input data directly from off-chip memory. We use the nearest neighbourhood upsampling technique for this CNN, and the Concat kernel performs depth-wise concatenation.

4.4.2.7 Host Interface

We use the Intel OpenCL framework for the development of the accelerator. The host driver is responsible for the data transfer of the image and weights for each layer. The data load/store units (FRU, FFU, FWU) are launched concurrently by the host interface using different command queues. Free-running autorun kernels are used to design processing elements (PEs) and to share configuration information among kernels. The host invokes the computation for each layer sequentially with the layer parameters.

4.5 Experimental Setup and Results

To run our experiments, we have used a Terasic DE5-anet DDR4 board with 4GB of external memory (DDR4). The board has Intel Arria10 GX FPGA. It complies with the PCIe Gen3 standard and has OpenCL BSP support. Intel FPGA SDK for OpenCL 20.2 compiles host code and device kernels. The host system is an HP Z640 workstation that contains Intel(R) Xeon(R) CPU E5-2630 with 40 cores. Data is transferred from the CPU to FPGA through the PCIe interface using Direct Memory Access (DMA). The target of our CNN accelerator is to fully utilize the available DSP blocks to maximize the parallel computations that would enable the CNN accelerator to achieve maximum performance.

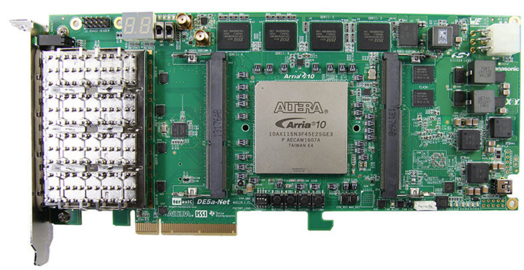


Figure 4.5 Terasic DE5anet DDR4 FPGA

The three architectural parameters defined for the accelerator, namely n_f, x_{par}, and n_d, can be used to scale up the DSP utilization. However, scaling up the architectural parameters x_{par} and n_d increases the size of the input shift register-based buffer Equation. (4.5) and corresponding weights buffer, thereby increasing the on-chip memory utilization. Raising the n_f increases the number of DSPs. However, the exhaustive reuse of DSP blocks increases the system fan-out, at times, will make it unable to serve the timing requirements. Based on Equations (4.3) and (4.4), and empirical timing analysis,we chose the values of n_f, n_d and x_{par} to be $\langle 16, 16, 3 \rangle$ to obtain optimized throughputs. We synthesized the designs for multiple precisions (Fixed-8, Fixed-16 and FLOAT32) and obtained latencies of 98ms (57 GOPs/s), 120.79 ms (46.16 GOPs/s) and 497 ms (11.22 GFLOPs/s) operating at 234.38 MHz, 227.78 MHz and 172.92 MHz respectively. Filter weights are quantized accordingly.

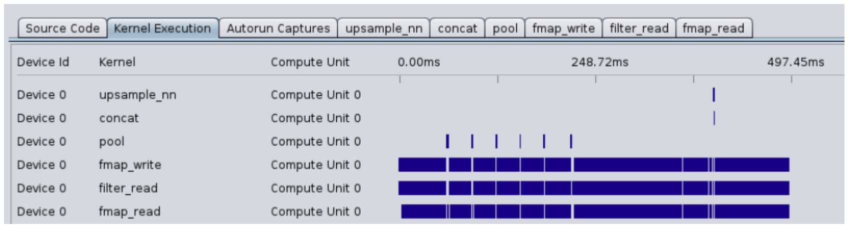


Figure 4.6 System Latency for Float32 precision

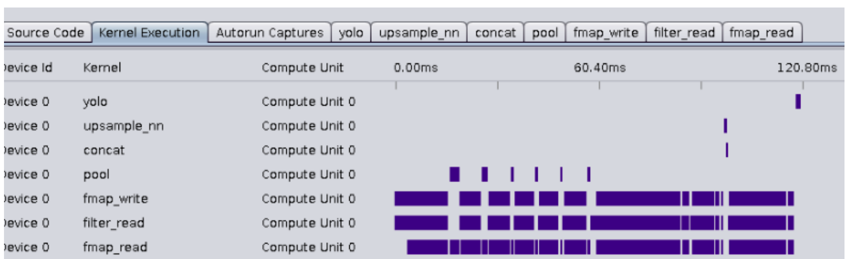


Figure 4.7 System Latency for Fixed16 precision

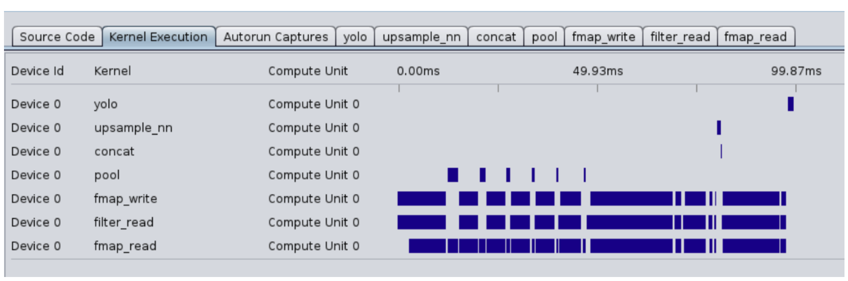
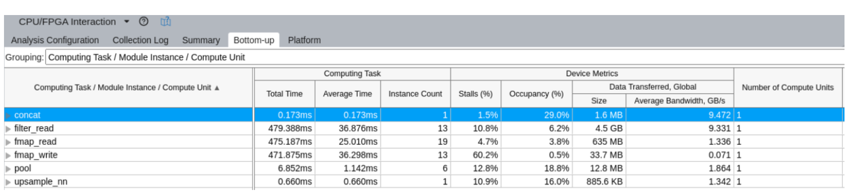


Figure 4.8 System Latency for Fixed8 precision



CPU/FPGA Interaction

Analysis Configuration | Collection Log | Summary | Bottom-up | Platform

Grouping: Computing Task / Module Instance / Compute Unit

Computing Task / Module Instance / Compute Unit	Computing Task			Device Metrics				Number of Compute Units
						Data Transferred, Global		
	Total Time	Average Time	Instance Count	Stalls (%)	Occupancy (%)	Size	Average Bandwidth, GB/s	
concat	0.173ms	0.173ms	1	1.5%	29.0%	1.6 MB	9.472	1
filter_read	479.388ms	36.876ms	13	10.8%	6.2%	4.5 GB	9.331	1
fmap_read	475.187ms	25.010ms	19	4.7%	3.8%	635 MB	1.336	1
fmap_write	471.875ms	36.298ms	13	60.2%	0.5%	33.7 MB	0.071	1
pool	6.852ms	1.142ms	6	12.8%	18.8%	12.8 MB	1.864	1
upsample_nn	0.660ms	0.660ms	1	10.9%	16.0%	885.6 KB	1.342	1

Figure 4.9 Peak Bandwidth measured using Vtune profiler

-	[3]	[46]	Our Work		
Platform	Ultra96 V2	Zynq 7000 Soc	Terasic DE5Anet-DDR4		
Precision	FIXED 8	FIXED 16	FLOAT 32	FIXED16	FIXED8
Clock Freq	250MHz	100 MHz	172.92 MHz	227.78 MHz	234 MHz
Logic Util	27.3K(17%)	25.9K(49 %)	337K (79 %)	212.5K (49.7 %)	122K (28 %)
BRAM	248 (61%)	185 (66%)	1075 (39.6 %)	693 (26.6 %)	555 (21.3 %)
DSP	242 (67%)	160 (72%)	957 (63 %)	477 (31.4 %)	477 (31.4 %)
Latency	121 ms	532 ms	497 ms	120.79 ms	98 ms
Throughput	31.50 GOPs/s	10.45 GOPs/s	11.22 GFLOPs/s	46.16 GOPs/s	57 GOPs/s

Table 4.1 Comparison with existing implementations

Figures. 4.6, 4.7, 4.8 present the latencies obtained for the accelerator for the corresponding precisions collected using Dynamic Profiler. Figure. 4.9 gives the average bandwidth measured using the Intel Vtune profiler. It can be observed that the system attains a peak bandwidth of 9.472 GBps. Table. 4.5 gives the resource utilization and comparison of the proposed design with the existing state-of-the-art implementations. One of the first works, [4], used a hardware-software co-design approach and attained a high parallelism factor of 2304 for convolutions using many DSPs. All the other layers are processed sequentially using a soft-core MicroBlaze. However, the overall latency and actual throughput are not reported. Hence the work cannot be considered for comparison. The authors in [3] adapt the im2col-based Generic Matrix Multiply (GEMM) for accelerating convolutional layers. They used Fixed-8 precision and attained a latency of 121 ms (31.50 GOPs/s) at 250 MHz. The study in [46] exploits parallelism along the input channel, output channel, and filter kernel's surface dimensions. It used a 16-bit Fixed Point precision and attained a latency of 532ms (10.45 GOPs/s). Knowing that works in [3] and [46] used smaller boards than us, it is impossible to have a faircomparison. However, the critical constraint in [3] is it converts convolution operations into matrix multiplication forms. Converting the input feature map and filter data into Toeplitz matrix forms usually involves complex memory access patterns. Also, expressing convolutions as matrix multiplication alters overall MAC operations for convolutions. Hence, with the increased number of layers, the architecture is not scalable for denser variants of YOLO. The work in [46] only attains 10.45 GOPs/s, with a framerate ($<$ 2 fps) which does not make the architecture suitable for real-time implementations. Our implementations exceed works in [3] and [46] by 23% and 340% as shown in Table. 4.5.

Chapter 5

Conclusions

This work presents a systolic array-based FPGA accelerator for accelerating Yolov3-tiny. The Intel OpenCL framework for synthesizing the design on Terasic-DE5a-Net DDR4 FPGA is used for our design. The proposed accelerator is instantiated and tested for (Fixed-8, Fixed-16, and FLOAT32) precisions. The architecture is scalable to other versions of YOLO with minor changes in the host driver. The thesis is mainly concerned with building accelerators for deep learning inference. Most of the work is comprised of optimizing the basic block for extracting the performance.

While working on OpenCL, there are two questions for which the complete answers have yet to be found.

1. Can the exact architectural idea be implemented without the intervention of the compiler?
2. How does the DRAM scheduler work? How to control the arbitration logic with which the compiler handles memory requests?

The system's performance deteriorates drastically when the compute engines perform irregular accesses to an off-chip memory (DRAM). It is observed that the performance of the OpenCL kernels decreases with increased communication (blocking channels). Also, the compiler generates the schedule to handle the worst-case outcome, especially in the case of loop-carried dependencies, memory dependencies, variable loop bounds, and scenarios in which the compiler cannot determine the latency at compile time. Working on the behavioural pattern of the compiler for a long time transitioned my research interest towards compiler design of High-Level Synthesis. In future, we aim to write LLVM-based optimization passes for efficient burst-coalesced accesses of DRAM memory and optimize blocking FIFO-based communication between kernels. Also, the possibility of integrating workload-specific constraints into the hardware generated by the compiler will be explored.

Related Publications

P. Velicheti, S. Pentapati and S. Purini, ”Systolic Array based FPGA accelerator for Yolov3-tiny,” 2022 IEEE High Performance Extreme Computing Conference (HPEC), 2022, pp. 1-2, doi: 10.1109/HPEC55821.2022.9926371.

Bibliography

[1] Ieee standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.

[2] Efficient implementation of neural networks on field programmable gate arrays. RARC, Proceedings, 2020.

[3] T. Adiono, A. Putra, N. Sutisna, I. Syafalni, and R. Mulyawan. Low latency yolov3-tiny accelerator for low-cost fpga using general matrix multiplication principle. *IEEE Access*, 9:141890–141913, 2021.

[4] A. Ahmad, M. A. Pasha, and G. J. Raza. Accelerating tiny yolov3 using fpga-based hardware/software co-design. In *2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2020.

[5] Amazon Web Services. *Amazon EC2 Instance*, 2022.

[6] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. An opencl(tm) deep learning accelerator on arria 10. *CoRR*, abs/1701.03534, 2017.

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

[8] D. Chiou. The microsoft catapult project. In *2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 124–124, 2017.

[9] D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3642–3649, 2012.

[10] J. Dai, Y. Li, K. He, and J. Sun. R-fcn: Object detection via region-based fully convolutional networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, page 379–387, Red Hook, NY, USA, 2016. Curran Associates Inc.

[11] A. Dua, Y. Li, and F. Ren. Systolic-cnn: An opencl-defined scalable run-time-flexible fpga accelerator architecture for accelerating convolutional neural network inference in cloud/edge computing. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 231–231, 2020.

[12] R. Girshick. Fast r-cnn. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1440–1448, 2015.

[13] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 580–587, 2014.

[14] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, 2017.

[15] 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)*, pages 770–778, 2016.

[16] M. Jaderberg, K. Simonyan, A. Zisserman, and k. kavukcuoglu. Spatial transformer networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[17] S. Kala, B. R. Jose, J. Mathew, and S. Nalesh. High-performance cnn accelerator on fpga using unified winograd-gemm architecture. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(12):2816–2828, 2019.

[18] Khronos OpenCL Working Group. *The OpenCL Specification, Version 1.1*, 2011.

[19] B. Kim. wolfapple/traffic-sign-recognition: First release, Sept. 2020.

[20] H. T. Kung. Why systolic architectures? *Computer*, 15(1):37–46, 1982.

[21] M. Lechner, A. Jantsch, and S. M. P. Dinakarrao. Resconn: Resource-efficient fpga-accelerated cnn for traffic sign classification. In *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, pages 1–6, 2019.

[22] Y. Lecun, P. Haffner, and Y. Bengio. Object recognition with gradient-based learning. 08 2000.

[23] Y. LeCun, L. D. Jackel, L. Bottou, A. Brunot, C. Cortes, J. S. Denker, H. Drucker, I. M. Guyon, U. Muller, E. Sackinger, P. Y. Simard, and V. N. Vapnik. Comparison of learning algorithms for handwritten digit recognition. 1995.

[24] Z. Li, Y. Zhang, B. Sui, Z. Xing, and Q. Wang. Fpga implementation for the sigmoid with piecewise linear fitting method based on curvature analysis. *Electronics*, 11(9), 2022.

[25] T. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 936–944, Los Alamitos, CA, USA, jul 2017. IEEE Computer Society.

[26] Z. Lin, M. Yih, J. M. Ota, J. D. Owens, and P. Muyan-Özçelik. Benchmarking deep learning frameworks and investigating fpga deployment for traffic sign classification and detection. *IEEE Transactions on Intelligent Vehicles*, 4(3):385–395, 2019.

[27] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.

[28] 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*, 27:1861–1873, 2019.

[29] NVIDIA, P. Vingelmann, and F. H. Fitzek. Cuda, release: 10.2.89, 2020.

[30] PSG. *Intel® Arria® 10 Core Fabric and General Purpose I/Os Handbook*. Intel, 2022.

[31] PSG. *Intel® FPGA SDK for OpenCL™ Programming Guide*. Intel, 2022.

[32] J. Redmon. Darknet: Open source neural networks in c. `http://pjreddie.com/darknet/`, 2013–2016.

[33] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. pages 779–788, 06 2016.

[34] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6517–6525, 2017.

[35] J. Redmon and A. Farhadi. Yolov3: An incremental improvement, 2018.

[36] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, page 91–99, Cambridge, MA, USA, 2015. MIT Press.

[37] S. S. M. Sallah, F. A. Hussin, and M. Z. Yusoff. Road sign detection and recognition system for real-time embedded applications. In *International Conference on Electrical, Control and Computer Engineering 2011 (InECCE)*, pages 213–218, 2011.

[38] P. Sermanet and Y. LeCun. Traffic sign recognition with multi-scale convolutional networks. In *The 2011 International Joint Conference on Neural Networks*, pages 2809–2813, 2011.

[39] J. Stallkamp, M. Schlipsing, J. Salmen, and C. Igel. The german traffic sign recognition benchmark: A multi-class classification competition. In *The 2011 International Joint Conference on Neural Networks*, pages 1453–1460, 2011.

[40] 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*, FPGA '16, page 16–25, New York, NY, USA, 2016. Association for Computing Machinery.

[41] A. Vasudevan, A. Anderson, and D. Gregg. Parallel multi channel convolution using general matrix multiplication. *CoRR*, abs/1704.04428, 2017.

[42] S. I. Venieris and C.-S. Bouganis. fpgaconvnet: A framework for mapping convolutional neural networks on fpgas. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 40–47, 2016.

[43] D. Wang, K. Xu, and D. Jiang. Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, pages 279–282, 2017.

[44] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong. Automated systolic array architecture synthesis for high throughput cnn inference on fpgas. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2017.

[45] K. Xu, X. Wang, and D. Wang. A scalable opencl-based fpga accelerator for yolov2. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 317–317, 2019.

[46] Z. Yu and C.-S. Bouganis. A parameterisable fpga-tailored architecture for yolov3-tiny. In F. Rincón, J. Barba, H. K. H. So, P. Diniz, and J. Caba, editors, *Applied Reconfigurable Computing. Architectures, Tools, and Applications*, pages 330–344, Cham, 2020. Springer International Publishing.

[47] J. Zhang and J. Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, page 25–34, New York, NY, USA, 2017. Association for Computing Machinery.

[48] W. Zhang, M. Jiang, and G. Luo. Evaluating low-memory gemms for convolutional neural network inference on fpgas. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 28–32, 2020.

[49] Z. Zheng, H. Zhang, B. Wang, and Z. Gao. Robust traffic sign recognition and tracking for advanced driver assistance systems. In *2012 15th International IEEE Conference on Intelligent Transportation Systems*, pages 704–709, 2012.

[50] Álvaro Arcos-García, J. A. Álvarez García, and L. M. Soria-Morillo. Deep neural network for traffic sign recognition systems: An analysis of spatial transformers and stochastic optimisation methods. *Neural Networks*, 99:158 – 165, 2018.