Efficient processing of vision kernels and deep neural networks on reconfigurable computing architectures

Murad Qasaimeh
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Recommended Citation
Qasaimeh, Murad, "Efficient processing of vision kernels and deep neural networks on reconfigurable computing architectures" (2020). Graduate Theses and Dissertations. 18381.
https://lib.dr.iastate.edu/etd/18381

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Efficient processing of vision kernels and deep neural networks on reconfigurable computing architectures

by

Murad Qasaimeh

A dissertation submitted to the graduate faculty in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY

Major: Computer Engineering (Computing and Networking Systems)

Program of Study Committee:
Phillip H. Jones, Major Professor
Joseph A. Zambreno
Diane T. Rover
Akhilesh Tyagi
Matthew Darr

The student author, whose presentation of the scholarship herein was approved by the program of study committee, is solely responsible for the content of this dissertation. The Graduate College will ensure this dissertation is globally accessible and will not permit alterations after a degree is conferred.

Iowa State University
Ames, Iowa
2020

Copyright © Murad Qasaimeh, 2020. All rights reserved.
DEDICATION

To Dad, may God bless his soul
TABLE OF CONTENTS

LIST OF TABLES ................................................................. vi
LIST OF FIGURES ............................................................... viii
ACKNOWLEDGMENTS ............................................................. xi
ABSTRACT ................................................................. xii

CHAPTER 1. INTRODUCTION .................................................. 1
  1.1 Challenges and research opportunities ................................. 1
  1.2 Research problems .................................................. 3
  1.3 Organization of the dissertation .................................... 4

CHAPTER 2. MODIFIED SLIDING WINDOW ARCHITECTURE ................. 5
  2.1 Related work .................................................. 7
  2.2 Traditional sliding window architecture ......................... 8
  2.3 Proposed compression algorithm .................................. 9
    2.3.1 Integer wavelet transform (IWT) ............................ 10
    2.3.2 Bit packing ............................................. 10
    2.3.3 Bit unpacking ........................................... 12
  2.4 Proposed hardware architecture .................................. 13
    2.4.1 Integer wavelet transform module ........................ 14
    2.4.2 Bit packing module ...................................... 15
    2.4.3 Bit unpacking module .................................... 17
    2.4.4 Inverse integer wavelet transform (IIWT) module ...... 19
    2.4.5 Memory unit ............................................ 19
  2.5 Results and analysis ............................................. 21
    2.5.1 Memory resource savings ................................ 21
    2.5.2 Hardware resource utilization ......................... 24

CHAPTER 3. RUNTIME CONFIGURABLE HISTOGRAM ARCHITECTURE ....... 27
  3.1 Related work .................................................. 29
  3.2 Histogram-based descriptors ..................................... 30
  3.3 Histogram binning technique .................................... 31
    3.3.1 One-dimensional histogram ................................ 32
    3.3.2 Two-dimensional histogram .............................. 32
  3.4 Proposed hardware architecture ................................ 34
    3.4.1 Gradient computation block .............................. 34
3.4.2 A 2D array of processing elements (PEs) ............................... 35
3.4.3 Normalization ................................................................. 40
3.5 Evaluation and verification ..................................................... 41
3.5.1 Accuracy comparison ......................................................... 41
3.5.2 Hardware resource utilization ............................................. 43
3.5.3 Comparison with dedicated hardware architectures .................. 44

CHAPTER 4. ANALYZING THE ENERGY EFFICIENCY OF VISION KERNELS .... 49
4.1 Introduction ................................................................. 49
4.2 Related work ......................................................... 51
4.3 Background ................................................................. 53
4.3.1 Embdedded platforms ................................................... 53
4.3.2 Computer vision libraries .............................................. 55
4.3.3 Neural network inference frameworks ................................. 56
4.3.4 Categories of vision kernels .......................................... 57
4.3.5 Neural networks ............................................................ 59
4.4 Experimental methodology .................................................. 60
4.4.1 Performance metrics ..................................................... 60
4.4.2 Measurement techniques and platforms ............................... 62
4.4.3 Benchmarking approach ................................................. 64
4.5 Experimental results ....................................................... 67
4.5.1 Single kernel performance ............................................. 67
4.5.2 Complete vision pipeline performance ............................... 71
4.5.3 Neural network inference performance ............................... 74

CHAPTER 5. AN EFFICIENT STRUCTURED SPARSE PRUNING OF CONVOLUTION LAYERS USING LFSR ................................................................. 80
5.1 Introduction ................................................................. 80
5.2 Background and motivations ................................................ 82
5.2.1 Convolution operations in CNNs .................................. 82
5.2.2 DNN weight pruning .................................................. 83
5.2.3 Sparse matrix representation ........................................ 84
5.2.4 Linear feedback shift register (LFSR) ................................. 86
5.3 Related work ................................................................. 87
5.4 Proposed pruning method .................................................... 88
5.4.1 LFSR patterns configuration ........................................... 88
5.4.2 Finding best seeds ..................................................... 89
5.4.3 Training with regularization .......................................... 91
5.4.4 Pruning and re-training ............................................... 91
5.5 Hardware accelerator ...................................................... 91
5.5.1 Hardware architecture and dataflow ................................ 92
5.5.2 Computation optimization ........................................... 94
5.5.3 Memory access optimization ......................................... 95
5.6 Experiment ................................................................. 96
5.6.1 Evaluation of pruning algorithm ................................... 97
5.6.2 Evaluation of hardware architecture ............................... 98
### LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>BRAMs utilization of traditional sliding window</td>
<td>21</td>
</tr>
<tr>
<td>Table 2.2</td>
<td>Number of BRAMs (18Kb) for resolution (512×512)</td>
<td>23</td>
</tr>
<tr>
<td>Table 2.3</td>
<td>Number of BRAMs (18Kb) for resolution (1024×1024)</td>
<td>23</td>
</tr>
<tr>
<td>Table 2.4</td>
<td>Number of BRAMs (18Kb) for resolution (2048×2048)</td>
<td>23</td>
</tr>
<tr>
<td>Table 2.5</td>
<td>Number of BRAMs (18Kb) for resolution (3840×3840)</td>
<td>24</td>
</tr>
<tr>
<td>Table 2.6</td>
<td>Hardware resource utilization of architecture units</td>
<td>25</td>
</tr>
<tr>
<td>Table 2.7</td>
<td>Hardware resource utilization for resolution (2048×2048)</td>
<td>26</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Area under ROC (AUC) for software and hardware</td>
<td>42</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Processing element hardware resources</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Gradient computation block hardware resources</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.4</td>
<td>Normalization hardware resources</td>
<td>44</td>
</tr>
<tr>
<td>Table 3.5</td>
<td>2D array of PE hardware resources</td>
<td>45</td>
</tr>
<tr>
<td>Table 3.6</td>
<td>SIFT descriptor extraction time</td>
<td>46</td>
</tr>
<tr>
<td>Table 3.7</td>
<td>SIFT hardware architectures</td>
<td>46</td>
</tr>
<tr>
<td>Table 3.8</td>
<td>HOG descriptor extraction time</td>
<td>47</td>
</tr>
<tr>
<td>Table 3.9</td>
<td>HOG hardware architectures</td>
<td>47</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Theoretical peak performance of hardware platforms [13]</td>
<td>63</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Data movers energy consumption measurements</td>
<td>66</td>
</tr>
<tr>
<td>Table 4.3</td>
<td>Ratios of energy/frame reduction (reference CPU)</td>
<td>71</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Table 4.4</td>
<td>FPGA’s reduction ratios with respect to GPU</td>
<td>74</td>
</tr>
<tr>
<td>Table 4.5</td>
<td>Top-5 (Top-1) classification accuracy</td>
<td>75</td>
</tr>
<tr>
<td>Table 4.6</td>
<td>FPGA’s performance with respect to GPU FP16 [b=128 and t=8]</td>
<td>78</td>
</tr>
<tr>
<td>Table 5.1</td>
<td>Sparse representation overhead</td>
<td>86</td>
</tr>
<tr>
<td>Table 5.2</td>
<td>Primitive polynomials</td>
<td>86</td>
</tr>
<tr>
<td>Table 5.3</td>
<td>List of used DNN models</td>
<td>97</td>
</tr>
<tr>
<td>Table 5.4</td>
<td>Available resource in ZCU102 platform</td>
<td>100</td>
</tr>
<tr>
<td>Table 5.5</td>
<td>Performance comparison with related work [(T_h = 8, T_w = 8, T_n = 24)]</td>
<td>101</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Line-buffering sliding window architecture</td>
<td>9</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of the compression algorithm for an $8 \times 8$ window size</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Memory requirement as the window slides across the image</td>
<td>11</td>
</tr>
<tr>
<td>2.4</td>
<td>The proposed sliding window architecture</td>
<td>13</td>
</tr>
<tr>
<td>2.5</td>
<td>2D Haar integer wavelet transform block</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Bit packing unit</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Find minimum number of bits architecture</td>
<td>16</td>
</tr>
<tr>
<td>2.8</td>
<td>Bit unpacking unit</td>
<td>18</td>
</tr>
<tr>
<td>2.9</td>
<td>Example of unpacking process</td>
<td>18</td>
</tr>
<tr>
<td>2.10</td>
<td>2D Haar inverse integer wavelet transform block</td>
<td>19</td>
</tr>
<tr>
<td>2.11</td>
<td>Memory mapping options</td>
<td>20</td>
</tr>
<tr>
<td>2.12</td>
<td>Memory requirement as the window slides across the image</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Four steps to compute histogram-based feature descriptors</td>
<td>31</td>
</tr>
<tr>
<td>3.2</td>
<td>Computing 1D histogram of gradients</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Computing 2D histogram of gradients</td>
<td>34</td>
</tr>
<tr>
<td>3.4</td>
<td>Gradient computation block with window size $(3 \times 3)$</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>Processing element (PE) architecture and its configurations</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>Processing element configurations key</td>
<td>37</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of $4 \times 4$ PEs (3 regions, 1 bin)</td>
<td>38</td>
</tr>
</tbody>
</table>
Figure 3.8  Configurations for SIFT, GLOH, DAISY, AB-SIFT and HOG algorithms . . 39
Figure 3.9  4x4 array of PEs configured into three regions . . . . . . . . . . . . . . . . . 40
Figure 3.10 ROC curves for the 8 oxford image sets . . . . . . . . . . . . . . . . . . . . 42
Figure 3.11 SIFT implementations (hardware cost vs. execution time) . . . . . . . . . . 48
Figure 3.12 HOG implementations (hardware cost vs. execution time) . . . . . . . . . . 48
Figure 3.13 HOG implementations (accuracy vs. energy) . . . . . . . . . . . . . . . . . . 48
Figure 4.1 Steps involved in compute and system performance measurements . . . . . 61
Figure 4.2 Measuring power on the platform’s CPU cores, and its FPGA or GPU . . 63
Figure 4.3 VisionWorks vs. OpenCV CUDA in terms of frame rate and energy/frame . 65
Figure 4.4 Input processing operations kernels . . . . . . . . . . . . . . . . . . . . . . . 68
Figure 4.5 Arithmetic operations kernels . . . . . . . . . . . . . . . . . . . . . . . . . 68
Figure 4.6 Filters operations kernels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
Figure 4.7 Image analysis operations kernels . . . . . . . . . . . . . . . . . . . . . . . . 69
Figure 4.8 Geometric transforms kernels . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Figure 4.9 Composite kernels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
Figure 4.10 Background subtraction pipeline components . . . . . . . . . . . . . . . . . 72
Figure 4.11 Color segmentation pipeline components . . . . . . . . . . . . . . . . . . . . 72
Figure 4.12 Harris corners tracking pipeline components . . . . . . . . . . . . . . . . . . 73
Figure 4.13 Stereo block matching pipeline components . . . . . . . . . . . . . . . . . . . 73
Figure 4.14 FPGA outperforms GPU and CPU in energy/frame consumption and EDP 74
Figure 4.15 System performance for inception-v2 network . . . . . . . . . . . . . . . . . 76
Figure 4.16 System performance for resNet50 network . . . . . . . . . . . . . . . . . . . 76
Figure 4.17 System performance for resNet-18 network . . . . . . . . . . . . . . . . . . . 77
Figure 4.18 System performance for squeezeNet-v2 network .......................... 77
Figure 4.19 System performance for mobileNet network ............................... 78
Figure 5.1 Convolution layer with two filters ............................................... 82
Figure 5.2 Coordinate list (COO) ................................................................. 84
Figure 5.3 Compressed sparse row (CSR) ...................................................... 85
Figure 5.4 Compressed sparse column (CSC) ............................................... 85
Figure 5.5 Linear feedback shift register (LFSR) .......................................... 87
Figure 5.6 Four different pattern configurations for pruning weights ............... 89
Figure 5.7 Example of pruning filters in perCoord configuration ...................... 90
Figure 5.8 Overview of FPGA-based CNN accelerator design ....................... 92
Figure 5.9 Hardware architecture of sparse convolution engine .................. 94
Figure 5.10 Element-matrix convolution operation ....................................... 95
Figure 5.11 An accuracy vs. sparsity comparison .......................................... 98
Figure 5.12 Resource utilization \( H_iW_jN_k \) where \( T_h = i, T_w = j, T_n = k \) ....... 99
ACKNOWLEDGMENTS

This dissertation becomes a reality with the kind support and help of many individuals. I would like to extend my sincere thanks to all of them.

Foremost, I would like to express my gratitude and appreciation towards my family for the support and encouragement to complete this work. Thanks also to all my friends, who were an integral part of this experience. Special gratitude and thanks to Karishma Bendale for your continued support and encouragement.

I am particularly grateful to my advisor Dr. Phillip Jones and Dr. Joseph Zambreno for their guidance and support in all aspects of this research as well as their invaluable mentorship which has helped me to navigate my academic career and to plot the course of my professional career. I would also like to thank my committee members for their efforts and contributions to this work: Dr. Diane Rover, Dr. Akhilesh Tyagi, Dr. Henry Duwe and Dr. Matt Darr. I am also thankful to Kristof Denllof and Kees Vissers for their invaluable mentorship during my internship at Xilinx.

Lastly, my thanks and appreciation go to my colleagues in the RCL lab and people who have willingly helped me out with their abilities.
Computer vision algorithms empowered with the recent advances in deep learning play a fundamental role in solving many problems that seemed impossible just a decade ago. However, the computational complexity and memory footprint of these algorithms keep increasing to either enhance accuracy or to solve more complex problems. This places a heavy load on computing platforms used to run these algorithms. Moreover, the exponential growth in computing platforms’ capability has slowed due to the saturation in Moore’s law, which makes the problem even more challenging. In this dissertation, I propose hardware/software co-optimization techniques to improve the performance of the most commonly used components in vision pipelines and deep neural networks: Sliding window structures, histogram computing, and convolution operations. The main contributions of this work include the following: (1) Proposing a modified sliding window architecture for reducing Block RAMs resources on FPGA. I propose a new sliding window architecture that takes advantage of most of an images’ information residing in low frequencies to reduce the required on-chip memory. The architecture uses a novel sliding window compression algorithm that can be efficiently implemented in hardware and gives comparable compression ratios to the state-of-the-art compression algorithms. It also has the flexibility of changing its compression ratio based on a threshold parameter. (2) Implementing a run-time programmable architecture for computing histogram-based feature descriptors. I propose a configurable hardware architecture that has the flexibility to compute different types of descriptors in real-time. The architecture is capable of computing several feature description algorithms using a single datapath. The architecture is configurable in terms of patch sizes, number of regions, and number of bins per region. Using optimization techniques, I was able to reduce the complexity of computing 2D histograms from $O(n^2)$ to $O(n)$. (3) Evaluating the energy-efficiency of computer vision kernels and DNNs on FPGAs. I benchmark representative vision kernels, complete vision pipelines and DNNs for the ARM57 CPU,
Nvidia Jetson TX2 (GPU-accelerated) and Xilinx UltraScale (FPGA-accelerated) and give insight into the reasons behind the observed run-time, power, and energy consumption performance for each evaluated platform. (4) Proposing a hardware-aware pruning algorithm that generates geometrically structured sparse weights. The locations of non-zero weights follow pseudo random patterns generated by Linear Feedback Shift Registers (LFSRs). I also propose an FPGA-based inference engine for sparse convolution. It uses pruned models generated by my approach to speed up the convolution computation. This way copying sparse weight indices from off-chip memory is avoided by computing these indices in real-time. The proposed techniques in this work show an improvement in the performance of sliding window structures, histogram computing, and convolution operations in terms of their hardware resource utilization, programmability, and runtime and bandwidth requirements.
CHAPTER 1. INTRODUCTION

Computer vision algorithms empowered with deep neural networks have achieved remarkable success in many challenging tasks including image classification, object detection, and image segmentation. Although they provide state-of-the-art accuracy, they require considerable amounts of storage, memory bandwidth and computational resources, which limit their deployment on embedded platforms. However, the importance of their applications combined with their computational intensity has led many researchers to investigate custom hardware acceleration as a solution to deploy these computationally intensive algorithms on limited resources embedded platforms. Different kinds of acceleration engines have been proposed in industry and academia, including: (1) Dedicated units for vision acceleration within system on chips (SoCs) have been implemented. For example, Apple A12 bionic SoC with dedicated neural processing engine (NPU), Google’s tensor processing unit (TPU) and Intel’s Myriad2 vision processing unit (VPU) [45]. (2) GPUs with multi-threading programming models are highly popular in this domain as well. GPUs provide massively parallel execution resources and high memory bandwidth. However, their high performance comes at the cost of high-power dissipation [22]. (3) FPGAs offer opportunities for exploiting low-level fine-grained parallelism by customizing data paths to the requirements of a specific algorithm/application. In term of GOPs/watt/dollar, FPGAs are considered to be an appealing platform for accelerating vision applications [66].

1.1 Challenges and research opportunities

Developing efficient hardware accelerators on FPGAs still has many challenges that impose limitations on porting many vision algorithms into hardware efficiently. These challenges include:

**Resource Utilization.** FPGA chips are made of a finite number of predefined resources with programmable interconnects to implement a reconfigurable digital circuit. These resources
include: (1) Lookup tables (LUTs) for implementing arithmetic/logical functions, (2) Flip flops (FFs) for storing small and distributed data elements, (3) Block RAMs for storing relatively larger data elements (a few MBs), and (4) DSPs for implementing high-efficient arithmetic functions. A significant fraction of current FPGAs’ silicon area and power budget are devoted to BRAMs, which makes integrating larger amounts of on-chip memory more difficult [85]. Therefore, current FPGAs have limited on-chip memory capabilities (i.e. 6MB of on-chip memory on high end Xilinx’s Virtex 7 FPGA). However, many computer vision algorithms are memory-bandwidth bounded not compute-bounded, which makes the available on-chip memory a limiting factor in scaling most of these algorithms [31]. Therefore, proposing techniques to efficiently utilize the available memory resources on FPGAs is essential to reach the highest performance.

**Programmability.** Many hardware architectures have been proposed in literature to accelerate computer vision applications on FPGAs, but most of these architectures are realized as static hardware pipelines optimized for only a single algorithm. The lack of flexibility in these architectures increases development effort if deployed applications need to be modified or upgraded. Moreover, many computer vision algorithms have very similar data structures and operations, which make it possible to take advantage of the common characteristics between different algorithms to build a generic and flexible hardware accelerators. Therefore, implementing a flexible hardware architecture capable of computing a wide range of algorithms has many advantages.

**Energy Efficiency.** Energy efficiency is a growing concern in many embedded vision applications. This is due to the gap between the current advancements in battery technology and the demands of embedded systems [89]. Moreover, heat dissipation becomes a major issue with high energy consumption. Therefore, analyzing how different hardware platforms may impact the energy efficiency of vision algorithms is an important problem. Moreover, proposing hardware optimization techniques to reduce the energy consumption of these algorithms is essential to the continues advancement in embedded vision applications.

**Bandwidth Requirements.** The performance of many vision applications on embedded platforms is bounded by memory bandwidth requirements. Therefore, proposing software and
hardware optimization techniques to reduce the amount of data transfer by either compression or pruning redundant and less important data while maintaining acceptable accuracy is essential as the amount of data and algorithm complexity increases.

1.2 Research problems

My thesis focuses on software and hardware co-optimization techniques for efficient processing of computer vision algorithms on reconfigurable architectures. The proposed techniques aim to improve architecture’s efficiency in terms of run-time, energy consumption, resource utilization and programmability.

In this dissertation, I investigate the following questions: (1) To what degree can a light-weight compression algorithm be used to efficiently reduce resource utilization of reconfigurable vision architectures? (2) Can we implement run-time programmable architectures for vision algorithms while maintaining comparable performance to hardwired architectures? (3) To what extent can generating sparse indices on-chip improve the performance and bandwidth utilization in sparse convolution?

Contributions. Motivated by these questions, we proposed techniques to improve the performance of vision algorithms on reconfigurable architectures, with a focus on the commonly used components in these algorithms, such as: sliding window, histogram computing, and convolution. The main contributions of this work include the following: (1) Proposing a modified sliding window architecture for reducing Block RAM resources on FPGAs [76], (2) Implementing a run-time programmable architecture for computing histogram-based feature descriptors [77], (3) Evaluating the energy-efficiency of computer vision algorithms on FPGAs [78] [74] [73], and (4) Proposing a hardware-aware structured pruning method that uses pseudo random sequences generated by LFSRs and designing an accelerator that takes advantage of the proposed pruning method and applying a set of architecture optimizations for sparse CNNs [75].
1.3 Organization of the dissertation

The remainder of my dissertation is organized as follows. Chapter 2 describes a compression algorithm and hardware architecture for a modified sliding window. Chapter 3 describes my implementation of a run-time programmable architecture for computing histogram-based feature descriptors. Chapter 4 describes an energy-efficiency evaluation of computer vision algorithms on CPU, GPU and FPGAs. Chapter 5 describes my proposed structured pruning method and its hardware architecture. Finally, conclusions are drawn and potential directions for future research are discussed in Chapter 6.
CHAPTER 2. MODIFIED SLIDING WINDOW ARCHITECTURE

Sliding window is one of the most commonly used techniques in image processing algorithms. Implementing it in hardware requires buffering image rows on-chip to exploit data locality and avoid redundant off-chip pixel transfers. However, scaling this architecture to large window sizes or high resolutions linearly increases on-chip memory utilization. This imposes limitations on porting many image processing algorithms into hardware efficiently. In this work, we propose a new sliding window architecture that utilizes less on-chip memory resources while maintaining performance as compared to the traditional method. The proposed architecture exploits that most natural images have smooth color variations with fine details in between these variations to compress images. It decomposes non-zero image pixels into their wavelet components and represents each wavelet coefficient with a minimum number of bits. The architecture is flexible to use lossless or lossy compression based on a configurable threshold value. The FPGA implementation of our proposed architecture shows memory saving of 25-70% compared to the traditional architecture using lossless compression, and for lossy compression with up to a mean square error of 5 achieves up to 84% in memory savings.

Many image processing algorithms use a sliding window technique as part of their algorithm. The sliding window operation repeatedly gathers a rectangular region of pixels, calculates an output for that window, and then slides across the input image. This operation is computationally and data intensive and benefits from hardware acceleration on FPGAs, especially for real-time applications [28]. Hardware implementations typically buffer image rows on-chip to exploit data locality and avoid redundant pixel transfers. Although these implementations provide significant performance improvement, scaling them to large window sizes or high resolutions linearly increases on-chip memory utilization. This imposes limitations on porting many image processing algorithms into hardware efficiently.
There are many image processing algorithms currently limited by the number of Block RAMs (BRAMs) available on FPGAs [42][12][11]. For example, in object detection algorithms, the maximum detectable size is limited by the window size supported in hardware. Increasing the window size will increase the chances of detecting more objects, but will also require more BRAMs to store additional image rows on-chip. A common solution is to scale down and re-scan the whole image [42]. Implementing lens distortion correction on FPGAs is another example. The maximum distortion coefficients supported by a hardware implementation is also limited by the window size supported. Increasing the window size will increase the mapping range supported by the distortion correction core, but increases on-chip memory requirements [12]. Supporting larger window sizes for image filters often increases their accuracy. For example, for a Gaussian smoothing filter, the size of the window should be at least 5 times its standard deviation to not lose precision by trimming the kernel’s small values. Moreover, most image processing algorithms consists of 2-5 sequential sliding window operations, where the output of one operation is fed via line buffers to the following operation. These implementations require a high number of BRAMs for implementing multiple sets of buffer lines [11].

In this work, we propose a modified sliding window architecture that utilizes on-chip memory resources more efficiently than the traditional architecture (introduced in Section 2.2) by storing compressed instead of raw pixels values. It uses the 2D Haar wavelet transform to decompose the active window’s pixels into its wavelet coefficients: approximation, horizontal details, vertical details, and diagonal details sub-bands. Then, it finds the minimum number of bits required to represents these coefficients. Because natural non-random images have most of their information in the approximation sub band and small details in the other sub-bands, the compression algorithm is able to represent the coefficients in the details sub-bands using less bits. The proposed sliding window architecture uses this simple compressing method to pack the compressed bits, and store them into the buffering system without any degradation in computing throughput performance. The architecture is also parametrizable to have the flexibility to change the compression ratio based on available on-chip memory and a threshold parameter that allows lossless or lossy compression.
2.1 Related work

A number of works can be found in the literature that aim to reduce on-chip memory requirements of the traditional sliding window architecture. Some works try to solve the problem by proposing new buffering methods. For example, in [97] and [96], instead of using the traditional line-buffering method, they use a block buffering technique. This method starts by reading a block of pixels with size greater than the size of the operation window. This allows for processing multiple windows without the need to load new data. While processing the current block, data for the next block is buffered. This method reduces the required on-chip memory but it is not as efficient as the traditional architecture, as its average number of off-chip accesses is greater than 1 pixel per window operation.

Others attempt to reduce the required memory by dividing the input image into segments and process each individually [27]. They partition the data array into segments along a row. Once the current segment has completed processing, the next segment of data is processed until the current row is finished. This approach can save some BRAMs, but is not efficient for streaming applications when pixels come directly from a camera sensor, as it requires pixels to be in off-chip memory.

The authors in [11] try to avoid utilizing sliding window buffering between an algorithm’s operations by using a larger window size for the first operation and uses several parallel processing units to compute the next operations. This method reduces the BRAMs usage but it consumes high amounts of combinational logic, and is only applicable when sequential operations can be composed into one composite operation.

In this work, we investigate reducing memory requirements for sliding window architectures by storing compressed instead of raw pixels values. The compression algorithm, to be suitable for our purpose, must compress an entire column of pixels in the current window every clock cycle. It should have a good compression ratio and can be implemented in hardware with low resource overhead. It should be lossless to recover the original image and have the flexibility to be lossy to recover an approximation of the original image with different mean square errors when additional compression is required.
Existing lossless and lossy compression algorithms have good compression and signal-to-noise ratios, but are not suitable for our purposes. For example, FPGA implementations of the standard lossless compression algorithm, JPEG-LS [72], consume too many resources and reduces system performance. It has a 6-stage pipeline and its maximum operational frequency is around 27MHz. The JPEG compression algorithm [81] is also not a good choice as it uses a fixed size window of 8×8, while in our case we want to compress the pixels in the left-most column of the window. The number of pixels in these columns depends on the window size.

Several other wavelet transform based compression algorithms exist in the literature. The most popular algorithms are: Embedded Zerotree Wavelet, Set Partitioning In Hierarchical Trees, and Embedded Block Coding with Optimized Truncation [47]. These algorithms are designed for variable bit rate image transmission and require three dynamically updated lists that make them unsuitable for high speed applications. This lead us to propose a new simpler image compression algorithm based on 2D Haar transform that satisfy our needs and can be implemented in hardware with relatively low hardware resources compared to the other compression algorithms.

### 2.2 Traditional sliding window architecture

Line-buffering is the most popular approach for implementing sliding window architectures [28]. It consists of a set of FIFOs connected to shift registers, as shown in Figure 2.1. The number of FIFOs and their size depends on the input image resolution and the window size. For an image of resolution (W×H) pixels and a window of size (N×N), the number of FIFOs is (N-1) with depth (W-N) pixels, and the size of the window should be N×N shift registers. The input pixels are pushed into the first line of the active window and the processed pixels are pushed out from the last line. The shift registers in the right-most column are connected to the input of the next FIFO lines. The outputs of the FIFOs are connected as inputs to the shift registers in the left-most column, as shown in Figure 2.1.

The architecture has three main states: (1) Fill the FIFOs: in which we need to wait until the FIFOs are full with valid pixels. We only receive input pixels and push them into the buffer lines,
and no output or operations are done in this stage; (2) Processing stage: we read one input pixel and process the active window to generate an output. This processing is done in one clock cycle; (3) Empty the FIFOs: in this stage, there are no more input pixels to read. But there are still valid pixels inside the FIFOs that need to be processed until no pixels are left inside the FIFOs.

The sliding window architecture produces an output of size (N×N), in other words, one value for each pixel in the input image [82]. For example, for an image of size 512×512 and a window of size 3×3, the first window to process is a square between (0,0) and (2,2). The required on-chip memory for this example, assuming 8-bit pixels, equals (512-3)×2×8 bits. This relatively small example can be implemented in hardware. However, for high resolutions and large window sizes, such as a window of size 120×120, an image of HD resolution (2048×2048), and 24-bit colored pixels, the required on-chip memory is at least (2048 - 120)×120×24 bits = 5,422Kb. While FPGAs like the XC7Z020 has a total on-chip memory of 5,018Kb.

### 2.3 Proposed compression algorithm

This section describes the proposed compression algorithm used in our modified sliding window architecture. The algorithm is composed of three main steps. First, the integer wavelet transform (IWT) decomposes the pixels in the active window into its wavelet components. Second, the Bitpacking step finds the minimum number bits required to represent each wavelet coefficient in

![Figure 2.1: Line-buffering sliding window architecture](image)
each of the four sub-bands. The bits of the non-zero coefficients, only, are packed into chunks and stored in the buffering system. Third, BitUnpacking reconstructs the original pixels from the compressed bits to be used by the next window. The following sub-sections describe these three steps in further detail.

2.3.1 Integer wavelet transform (IWT)

Forward IWT transforms image pixels into a set of integer coefficients. The transformation is reversible, meaning the pixels can be recovered without any loss using the inverse integer wavelet transform (IIWT). In this work, we use a 2D single-level Haar wavelet transform [101] to generate four sub-bands: (1) Approximation (LL), (2) Horizontal details (LH), (3) Vertical details (HL), and (4) diagonal details (HH) sub-bands. Figure 2.2 shows an example of a window of size $8 \times 8$ after it has been decomposed into its wavelet sub-bands.

![Figure 2.2: Example of the compression algorithm for an 8×8 window size](image)

2.3.2 Bit packing

In the Bit Packing step, wavelet coefficients are compared to a threshold value. If the absolute value of the coefficient is less than the threshold, it is considered insignificant and replaced with zero, otherwise it will not be modified. Figure 2.2 shows this step for a threshold value of zero (lossless). The coefficients did not change because the threshold value is zero. In the next step, for
each column of each sub-band the minimum number of bits (NBits) required to encode the largest pixel value in 2’s complement is determined and stored in the compressed window. Then, the least significant NBits bits of each coefficient for each sub-band column are packed together to form the compressed version of the original coefficients.

Figure 2.2, for simplicity, shows only the bit packing process for the vertical (HL) and diagonal (HH) sub-bands. It shows the NBits bits required to represent the first column of HL (Pixels: 13, 12, -9 and 7) is 5. So, the least significant 5 bits of the non-zero pixels (01101, 01100, 10111 and 00111) will be packed together and stored in the compressed window. BitMap is used to distinguish between zero or non-zero coefficients. The BitMap of the first column is 1111 because all the coefficients have non-zero values, while the BitMap of the last column is 0011 because the first two coefficients are zeros.

To see the efficiency of this algorithm on real images, Figure 2.3 shows the amount of memory required to buffer image rows (lossless compression) as a window of size 64×64 slides across an image of size 512×512. It shows that the number of bits required for the approximation (LL) sub-band is almost two times higher than the three detail sub-bands. It also shows that in the worst case we need around 40 Kbits to store the coefficients of the LH, HL and HH sub-bands and
around 65 Kbits for the LL sub-band. In total, we can store the coefficients of the 64 image rows in 185 Kbits plus 32 Kbits of management bits (total = 217 Kbits) compared to 230 Kbits using the traditional sliding window method. As image resolution increases so does the memory efficiency of this algorithm (see section VI).

In this example, the threshold value equals zero (lossless compression), increasing the threshold value (lossy compression) reduces the total number of bits because the number of zeros in the sub-bands increases and the algorithm uses only one bit for zero coefficients and compresses only non-zero coefficients. Increasing the threshold value increases the compression ratio, but decreases the quality of the reconstructed image.

2.3.3 Bit unpacking

To reconstruct the original coefficient values, we first read a BitMap value. If it equals zero, then the output should be zero, otherwise we read the least significant NBits and sign extend to the pixel size (8 bits) and send it to the active window. If NBits is less than 8, the remaining bits are kept for the next output. Even if this compression algorithm seems simple, it shows good compression ratios. But it also introduces management bits (BitMap and NBits) that need to be taken into consideration. For an image resolution \((W \times H)\) and a window size \((N \times N)\), the total management bits equals \(2 \times (W-N) \times 4\) bits for NBits and \((W-N) \times N\) bits for BitMap.

In this algorithm, we used a one-level Haar wavelet transform because adding more levels complicates the architecture for both the forward and inverse wavelet transform blocks. Moreover, using 2 or 3 levels of decomposition did not increase the compression ratio significantly. We also chose the Haar wavelet transform instead of other transformations like 5/3 and 7/9 for the same reasons. In the Bit Packing step, we find the minimum number of bits for each column in each sub-band instead of other options like for each coefficient or for each sub-band because there was a tradeoff between the compression ratio and the number of management bits.
2.4 Proposed hardware architecture

This section presents the proposed hardware architecture and its building blocks. The architecture consists of five modules: (1) 2D integer wavelet transformation (IWT), (2) Bit Packing, (3) Memory Units, (4) Bit Unpacking and (5) 2D inverse integer wavelet transformation (IIWT). Figure 2.4 shows an abstract high-level overview of the modified sliding window architecture and its components. The active window is implemented using shift registers so that a processing kernel can directly access all pixels of the active window each clock cycle. As an example, a 2D image filter could multiply each pixel in the active window with a corresponding constant in the filter kernel, and output these results as a sum or weighted sum.

Figure 2.4: The proposed sliding window architecture

The input pixels $\text{Pixel}(i)$ coming directly from a camera sensor or off-chip memory are stored in the first register of the first row in the active window and previous pixels are shifted to the right. The Integer Wavelet Transform module reads the right most column of the active window each clock cycle, and decomposes the pixels into their wavelet components. The resulting coefficients are fed to the Bit Packing module that represents these coefficients with the minimum number of bits and compresses them. When 8 bits have been accumulated in the Bit Packing unit, it writes the packed bits into the Memory Unit along with its management bits: (1) NBits: number of bits
of each compressed coefficient, and (2) BitMap: one bit to distinguish whether the coefficient has a zero or non-zero value. When the number of coefficients in the memory unit equals the image width, the Bit Unpacking unit reads the compressed bits from the FIFOs and reconstructs the coefficients' values. Finally, the reconstructed coefficients are transformed back to the original pixels by the Inverse Integer Wavelet Transform module. The results from IIWT module are written into the left most column of the active window and the previous pixels are shifted to the right. This process is repeated until all pixels in the image pass through the architecture. The following subsections describe the architecture’s blocks in further detail.

2.4.1 Integer wavelet transform module

This module receives input from the active window right-most registers, and sends its output to the Bit Packing module. Each clock cycle, it reads N pixels and generates N/2 low frequency coefficients (LL), and N/2 Horizontal details (LH) coefficients or N/2 vertical details (HL) coefficients, and N/2 diagonal details coefficients. Where N is the window size. In this work, we used the Haar wavelet transform because it maps to a simple hardware structure. The Haar wavelet transform equations are shown in Equations (2.1) and (2.2), where \(i\) and \(j\) are pixels coordinates.

\[
L(i, j) = X(i, j + 1) + H(i, j)/2 \tag{2.1}
\]

\[
H(i, j) = X(i, j) - X(i, j + 1) \tag{2.2}
\]

The hardware implementation of the 2D forward Haar wavelet transformation is shown in Figure 2.5. Each 1D block consists of one adder, one subtractor and one division by 2 (implemented as a shift right by 1). The 2D transformation is implemented by connecting four 1D blocks together as follows: the two low-frequency outputs (L) in the first stage are connected as inputs to the top block in the next stage, and the two high-frequency outputs (H) are connected to the bottom block in the next stage. The inputs are four pixels (X1, X2, X3 and X4) and the results are four sub-
band coefficients: LL (Low-low: Approximation sub-band), LH (Low-High: vertical details), HL (High-low: horizontal details) and HH (high-high: diagonal details).

Figure 2.5: 2D Haar integer wavelet transform block

2.4.2 Bit packing module

The Bit Packing block receives input from the IWT module, compresses each coefficient, and further packs the compressed coefficients together before storing to the Memory Unit. It has three steps: First, it finds the minimum number of bits required to represent the largest of the input coefficients (NBits). Second, it compares the coefficients values with a threshold parameter that determines whether the compression is lossless or lossy. If coefficients values are less than the threshold, they will be considered as insignificant and replaced by zero. The third step involves the actual compression. Each clock cycle, the block collects a coefficient’s NBits least significant bits. Once the number of collected bits reach the maximum bit width, it writes these packed bits to the Memory Unit along with the management information needed to restore the original pixels later.

Figure 2.6 shows the hardware architecture of the Bit Packing unit. It consists of three Registers: (1) Current number of bits (CBits): a 4-bit register that keeps track of the number of bits in the Yout_current register; (2) Current concatenation of bits (Yout_Current) is an 8-bit register that stores bits during the concatenation process; (3) The output register (Yout_Reg): is an 8-bit register used when the number of valid bits in Yout_Current reaches BitMax (8 bits), and the value of Yout_Current will be copied to Yout_Reg. The Bit Packing block also sets write enable WEN=1 to write valid output to the Memory Unit. The block has two comparators: one compares the input
(Xin) with a threshold value. If the input coefficient value is less than threshold, the BitMap value is set to 0 otherwise to 1. The other comparator compares the current number of bits (Cbits) to BitMax. It also has one adder that is used to add Cbits to the input number of bits (NBits). The number of Bit Packing blocks in the proposed architecture is equal to the window size, i.e. if the window size is 100×100, there will be 100 Bit Packing blocks.

To find the minimum number of bits required for encoding n coefficients (X₀-Xₙ₋₁) of the just inputted column of a sub-band, we first compare the sign bit (bit7) with bits 0-6 to find the first location that has a different value than the sign bit for each coefficient. We used 2-input XOR gates to do the equality comparison, as shown in Figure 2.7. The first input to all XOR gates is the sign bit and the second input is bits 0-6. To find the minimum number of bits required to
represent all the coefficients, we use n-input OR gates to find if any of the XOR gate’s outputs was 1. If the OR gates output is 1, the minimum number of bits should be at least 2 bits, else if the OR gates output is less than 4, it should be 3 bits, and so on. Figure 2.7 shows the block’s architecture when n=3. As an example, for $X_1 = (-6) \ 0b'11111010$, $X_2 = (-2) \ 0b'11111110$ and $X_3 = (6) \ 0b'00000110$, the output of the XOR gates will be 0000101, 0000001 and 0000110 respectively. The output of the OR gates will be 0000111, which indicates for each coefficient in this sub-band column the minimum number of bits required will be 4.

2.4.3 Bit unpacking module

The task of the Bit Unpacking module is to reconstruct the original pixels values from the compressed coefficients stored in the Memory Unit. The module first reads one value from each of the three Memory Unit buffers: (1) number of bits (NBits), (2) Bit map, and (3) Pixel FIFO. If the Bitmap bit equals zero, that means the original coefficient was less than the threshold value or zero. Otherwise, it extracts the lower NBits of the value read from the Pixel FIFO, then sign extends and sends it to the output. If NBits is less than 8 bits, the module keeps the remaining bits to be used for the next coefficient.

Figure 2.8 shows the hardware architecture of Bit UnPacking module. It has three registers: (1) CBits, similar to Bit Packing: it is a 4 bit register that keeps track of the remaining bits in the Yout_current register; (2) The remaining bits register, Yout_rem, a 16 bit register that is used to store the remaining bits after each output. For example, if the block reads 8 bits and NBits is only 2 bits, it will keep the remaining 6 bits to be used in the next output; (3) The output register, Yout_Reg. The block also has two comparators: one to check if the Bitmap bit is zero or one. Another to make sure that the block always has enough bits for the next output by checking if the CBits register value is less than 8. It has a multiplexer that selects bits from Yout_rem and/or Xin to be copied to Yout_reg. It also has one adder connected as shown in Figure 2.8.

Figure 2.9 shows an illustrative example of the bit unpacking process. It shows the compressed bits of five pixels (A, B, C, D and E), and the values of Yout_rem and Yout_reg in the first four
steps. In the first step, the block will read 8 bits that contain pixel (A)’s bits and part of pixel (B)’s bits. Then it will extract the lower NBits, and sign extend and write it into Yout_reg. Because the current number of bits (CBits) is now less than 8 bits, in the next step it will read another 8 bits as shown in Figure 2.9. The same process is applied to extract the lower NBits and sign extend the value and write it into Yout_reg. The size of Yout_reg is 16 bits because the worst case is when the previous step has NBits equals to 1 and in the next step NBits equals the max number of bits (8). In this case, we need the size of Yout_rem to be enough to store two consecutive reads from Pixel FIFO.
2.4.4 Inverse integer wavelet transform (IIWT) module

The inverse wavelet transform module takes input from Bit Unpacking and regenerates the original pixels values from the four wavelet sub-bands. The inverse Haar wavelet transformation equations are shown in Equations (2.3) and (2.4):

\[
X(i, j) = \left(\frac{H(i, j)}{2} - L(i, j)\right) + H(i, j) \tag{2.3}
\]

\[
X(i, j + 1) = \frac{H(i, j)}{2} - L(i, j) \tag{2.4}
\]

Figure 2.10 shows the architecture of the 2D inverse wavelet transformation block. It looks similar to the Forward Wavelet Transform block. Each 1D block has one addition, one subtraction and one division by 2. The two low frequency outputs (L) in the first 1D stage are connected to the input of the top block in next stage. The two high-frequency outputs (H) are connected to the input of the bottom block in the second stage. For an architecture with window size N, the number of 2D inverse wavelet transform modules equals N/2.

![2D Haar inverse integer wavelet transform block](image)

Figure 2.10: 2D Haar inverse integer wavelet transform block

2.4.5 Memory unit

The memory unit is where all the compressed pixels and management bits are stored. It contains buffers for storing the packed bits (the output of BitPacking blocks), number of bits (NBits), and BitMap. Based on the compression ratio, the output of the Bit Packing blocks can be mapped to FIFO buffer lines by one of the following options: First, store one packed image row in one
FIFO buffer line. This option has memory savings equals 0% because it is similar to the traditional sliding window architecture. The second option is to store two packed image rows in one FIFO. The memory savings will be around 50% compared to the traditional method. Third, storing four image rows in one FIFO to have an approximate saving of 75%. Fourth, to store eight image rows in one FIFO. The memory saving will be around 87.5%. Figure 2.11 shows the four options of mapping the packed bits to buffer lines.

Because each column in the decomposed image has two sub-bands (LL and LH or HL and HH), as shown in Figure 2.11, the total number of bits required for NBits equals \(2 \times 4 \times (\text{Image width} - \text{window size})\) bits. For BitMap, we need one bit for each pixel, so the total number of bits equals \((\text{image width} - \text{window size}) \times \text{window size}\) bits. Because for each column we need \(2 \times 4\) bits for NBits, mapping NBits to Block RAMs can be done by configuring 18Kb BRAMs to be \(2K \times 9\) in simple dual port mode. Mapping BitMap bits depends on the window size. For example, if the window size is 8, 16, 32, 64, or 128, and image width 512, the 18Kb BRAMs will be configured as \(2k \times 9, 1k \times 18, 512 \times 36, 2 \times (512 \times 36)\) and \(4 \times (512 \times 36)\) respectively.

Current Limitations. Our architecture currently has one limitation that the compression ratio should be known at design time. That means, the number of BRAMs and their configurations used in the memory unit should be known. This will not be an issue in cases where the image scenes do not change significantly. In this case the memory unit will be configured to the worst-case scenario. But in cases of bad frames or random images, the compression ratio will be very low and the size of the packed bits will be greater than the available BRAMs. This can be fixed in the future by...
making threshold values automatically adjustable based on the available memory and the current frame compression ratio. Moreover, the Bit Unpacking block in our architecture consumes a large number of LUTs resources compared to the other blocks, due to a large multiplexer. But our architecture LUTs resources is only a function of window size. While increasing image resolution does not impact the architecture’s LUT resources, allowing for increased BRAM savings with higher resolutions at no additional LUT cost.

2.5 Results and analysis

This section presents the experimental results for testing the performance of the proposed compression method, and evaluates the memory savings gained by using it in our sliding window architecture compared to the traditional architecture. It presents the hardware resources used to implement each block in the architecture, and the overall system.

2.5.1 Memory resource savings

To evaluate our compression method, we used 10 randomly selected images from the MIT Places Database for Scene Recognition [102]. This dataset includes indoor and outdoor scenes. We computed the average memory saving gained by compressing these images for different window sizes and image resolutions. Equation (2.5) shows the memory saving formula.

\[
\text{MemorySaving} = (1 - \frac{\text{Compressed}}{\text{Uncompressed}}) \times 100\% \quad (2.5)
\]

Table 2.1: BRAMs utilization of traditional sliding window

<table>
<thead>
<tr>
<th>Window size</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>3840</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>128</td>
<td>128</td>
<td>256</td>
</tr>
</tbody>
</table>
For lossless compression, the memory savings of our compression algorithm in comparison to the traditional sliding window approach for an 2048×2048 image resolution is around 26-34% for different window sizes. The compression ratio increases as we increase threshold value from 0 to 2, 4 and 6 (i.e. as the algorithm becomes more lossy). The saving is around 41-54% when threshold value is set to 6. These numbers take into account the overhead associated with the management bits (BitMap and NBits).

Table 2.1 shows the number of BRAMs used in the traditional sliding window architecture for different image widths (512, 1024, 2048 and 3840) and window sizes (8, 16, 32, 64 and 128). It shows that each FIFO line is realized in hardware by one 18Kb BRAM except for image width 3840. This is because each pixel is 8 bits and an 18Kb BRAM configured as 2k×9 can fit up to 2048 pixels. Thus, image rows of width 512, 1024 and 2048 can fit in one BRAM, while image widths greater than 2048 require cascading multiple BRAMs to store one image row.

Table 2.2, Table 2.3, Table 2.4, Table 2.5 and Figure 2.12 can be used to compare our approach’s memory usage to the traditional sliding window approach. We examine the impact of varying window size, image resolution, and specified lossiness. Table 2.2 shows that with our compression algorithm the packed bits for window size 8 can fit in two 18Kb BRAMs for the lossless case (T=0) with two BRAMs for management bits. This represents a 50% memory saving compared to the traditional architecture. Increasing the threshold value to 6 (most lossy case considered) makes it possible to fit more rows in one Block RAM. This represents a memory saving of 62.5%. These numbers show that the proposed architecture is more efficient in exploiting BRAMs by compressing the pixels and packing more than one image’s row into one BRAM.

Next, we take a closer look at the managment bits (BitMap and NBits) memory requirements. For window size 8 and image resolution 512×512, the size of BitMap equals 8×(512-8) bits and the size of NBits equals 4×2×(512-8) bits. The BitMap can be buffered in one 18Kb BRAM with a 2k×9 configuration. The NBits can also fit in one 18Kb BRAM with a 2k×9 configuration where bits 0-3 used for low sub band and bits 4-7 are used for the high sub band. Table 2.2 shows a total of two BRAMs for window size 8 and image resolution 512×512.
Increasing window size to 16 shows similar results for 512×512 resolution. The compressed bits of every four rows can fit in one BRAM for lossless compression and for lossy compression with threshold equals 2. This shows a memory saving of 62.5%, if the compression ratio is increased by increasing the threshold value to 4 and 6 to make every 8 rows fit in one BRAMs. The color in the tables represent number of image rows mapped to one BRAM. Green cells represent mapping each 8 rows in the input image to one BRAM. Blue cells represent mapping each 4 rows to one BRAM. Yellow cells represent mapping each two rows to one BRAM.

Table 2.2: Number of BRAMs (18Kb) for resolution (512×512)

<table>
<thead>
<tr>
<th>Window size</th>
<th>Packed bits</th>
<th>Management bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T=0</td>
<td>T=2</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>64</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>128</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 2.3: Number of BRAMs (18Kb) for resolution (1024×1024)

<table>
<thead>
<tr>
<th>Window size</th>
<th>Packed bits</th>
<th>Management bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T=0</td>
<td>T=2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 2.4: Number of BRAMs (18Kb) for resolution (2048×2048)

<table>
<thead>
<tr>
<th>Window size</th>
<th>Packed bits</th>
<th>Management bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T=0</td>
<td>T=2</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>32</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>64</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>128</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
Table 2.5: Number of BRAMs (18Kb) for resolution (3840×3840)

<table>
<thead>
<tr>
<th>Window size</th>
<th>Packed bits</th>
<th>Management bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>T=0</td>
<td>T=2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>128</td>
</tr>
</tbody>
</table>

Increasing image width from 512 to 1024 and 2048 increases the compression ratio, but the number of pixels in each row increases, so increases the total number of bits. This can be seen clearly when the image width increases from 1024 to 2048 in Table 2.3 and Table 2.4. When the Threshold value is 4 and 6, the memory saving is around 75% for 1024. That means the architecture was able to pack 8 rows of 1024 pixels in one BRAM. Increasing the image width to 2048 allows for packing 4 rows of 2048 pixels in one BRAM. A saving of approximately 50% compared to the traditional architecture.

The architecture currently uses a threshold to configure the system for lossless to varying degrees of lossy compression. Our evaluations show thresholds of 2, 4 and 6 gives mean square errors (MSEs) of 0.59, 3.2 and 4.8 respectively. Other options can be investigated to vary lossiness, such as using the average of previous pixels.

2.5.2 Hardware resource utilization

This section presents the Post-Synthesis hardware resource utilization for each block in our architecture. Each table shows the number of LUTs, registers and the maximum operating frequency. We used Xilinx Vivado 2015.3 tool and Xilinx Zynq 7020 (XC7z020) FPGA in our experiments. It has a total of 53,200 LUTs and 106,400 registers. Table 2.6 shows the resources of the forward and inverse integer wavelet transform blocks. The resources of these two blocks are similar because they have similar architectures. Table 2.6 shows the resources of the Bit Packing unit for different window sizes. It shows that resources are linearly increasing with the window size from 1% LUTs for window size 8 to 3%, 7%, 16% and 32% for window sizes 16, 32, 64 and 128. The Bit Unpacking
Table 2.6: Hardware resource utilization of architecture units

<table>
<thead>
<tr>
<th>Unit</th>
<th>Window size</th>
<th>LUTs</th>
<th>Registers</th>
<th>Freq (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>386 (0%)</td>
<td>166 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>770 (1%)</td>
<td>326 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td>IWT</td>
<td>32</td>
<td>1538 (2%)</td>
<td>646 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>3074 (3%)</td>
<td>1276 (1%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>6146 (11%)</td>
<td>2566 (2%)</td>
<td>592.1</td>
</tr>
<tr>
<td>Bit Packing</td>
<td>8</td>
<td>1061 (1%)</td>
<td>200 (0%)</td>
<td>538.6</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>2083 (3%)</td>
<td>400 (0%)</td>
<td>538.6</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>4047 (7%)</td>
<td>801 (0%)</td>
<td>538.6</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>8598 (16%)</td>
<td>1856 (1%)</td>
<td>538.6</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>17179 (32%)</td>
<td>3712 (3%)</td>
<td>538.6</td>
</tr>
<tr>
<td>Bit UnPacking</td>
<td>8</td>
<td>2130 (0%)</td>
<td>203 (0%)</td>
<td>343.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>4246 (7%)</td>
<td>387 (0%)</td>
<td>343.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>8039 (15%)</td>
<td>817 (0%)</td>
<td>343.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>15660 (29%)</td>
<td>1637 (1%)</td>
<td>343.1</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>31660 (59%)</td>
<td>3237 (3%)</td>
<td>343.1</td>
</tr>
<tr>
<td>Inverse IWT</td>
<td>8</td>
<td>386 (0%)</td>
<td>130 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>770 (1%)</td>
<td>258 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>1538 (2%)</td>
<td>529 (0%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>3074 (3%)</td>
<td>1055 (1%)</td>
<td>592.1</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>6146 (11%)</td>
<td>2108 (2%)</td>
<td>592.1</td>
</tr>
<tr>
<td>Overall</td>
<td>8</td>
<td>4994 (9%)</td>
<td>1643 (1%)</td>
<td>230.3</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>9432 (17%)</td>
<td>2792 (2%)</td>
<td>230.3</td>
</tr>
<tr>
<td></td>
<td>32</td>
<td>17773 (33%)</td>
<td>5091 (4%)</td>
<td>230.3</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>35751 (67%)</td>
<td>9680 (9%)</td>
<td>230.3</td>
</tr>
</tbody>
</table>

block consumes more resources compared to the other blocks. It consumes 15%, 29% and 59% LUTs for window size 32, 64 and 128. This is due to a large multiplexer in the block that selects bits from both the remaining bits from the previous read and the new input. Table 2.7 shows the overall resources of the whole architecture. The LUTs for window size 32 and 64 was around 33% and 67% of the total LUTs on the chip. For a window size of 128 the LUTs exceed this device resources.
Table 2.7: Hardware resource utilization for resolution (2048×2048)

<table>
<thead>
<tr>
<th>Window size</th>
<th>Traditional Sliding Window</th>
<th>Modified Sliding Window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LUTs</td>
<td>FFs</td>
</tr>
<tr>
<td>8</td>
<td>1808</td>
<td>1032</td>
</tr>
<tr>
<td></td>
<td>(3%)</td>
<td>(1%)</td>
</tr>
<tr>
<td>16</td>
<td>3670</td>
<td>2032</td>
</tr>
<tr>
<td></td>
<td>(6%)</td>
<td>(1%)</td>
</tr>
<tr>
<td>32</td>
<td>5479</td>
<td>4053</td>
</tr>
<tr>
<td></td>
<td>(10%)</td>
<td>(3%)</td>
</tr>
<tr>
<td>64</td>
<td>7288</td>
<td>6064</td>
</tr>
<tr>
<td></td>
<td>(13%)</td>
<td>(6%)</td>
</tr>
</tbody>
</table>

Figure 2.12: Memory requirement as the window slides across the image
CHAPTER 3. RUNTIME CONFIGURABLE HISTOGRAM ARCHITECTURE

Feature description is an essential component of many computer vision applications. It encodes the visual contents of images in a manner that is robust against various image transformations. Computing these descriptors is computationally expensive, which causes a performance bottleneck in many embedded vision systems. Although many hardware architectures have been proposed to accelerate feature description computation, most target a single feature description algorithm under specific constraints. The lack of flexibility of such implementations increases development effort if deployed applications need to be modified or upgraded. In this work, we propose a software configurable hardware architecture capable of computing different types of histogram-based feature descriptors without the need for re-synthesizing the hardware. The architecture takes advantage of data streaming to reduce the computational complexity of computing this class of descriptor. To illustrate the efficiency of our architecture, we deploy two of the most commonly used descriptors (SIFT and HOG) and compare their quality with software implementations. The architecture is also evaluated in terms of execution speed and resource usage and compared with dedicated hardware architectures. Our flexible architecture shows a speed up of 3× and 5× compared to state-of-the-art dedicated hardware architectures for SIFT and HOG, with resource usage overheads of (1.1×, 15×, and 1.6×) and (6.4×, 7×, and 32×) for LUT, FFs, and DSPs.

In computer vision, image features refer to salient points within a scene that are distinctive, repeatable and have enough intensity variation to be tracked reliably between images even under different image transformations and distortions, such as scaling, rotation, shearing, etc. Two processes are associated with image features: feature detection, and description. In feature detection, every pixel is examined to check if there is a feature of a given type (e.g. edges, corners, blobs) present at that pixel location. In feature description, the visual content around a detected feature
is captured by computing robust numeric representations from the surrounding pixels’ low-level information such as intensity, gradients, local binary pattern (LBP), etc. Image features are used in many applications, such as object detection and recognition, image retrieval, 3D reconstruction, and image mosaicing.

Histogram-based feature descriptors are the most commonly used feature description type due to their distinctiveness and robustness to various image transformations. However, computing these descriptors is computationally expensive and is the bottleneck of many vision systems [61]. For this reason, many hardware architectures have been proposed in the literature to accelerate this class of descriptor. Although these architectures reach high performance, most are hard-wired to support a specific histogram-based feature descriptor configuration, and cannot be modified without completely re-synthesizing and reprogramming the FPGA. These architectures are realized as static hardware pipelines optimized for only a single application. The lack of programmability in these architectures increases development effort when the deployed applications need to be modified or upgraded.

Implementing a flexible hardware architecture capable of computing a wide range of histogram-based descriptor variants has many advantages. For example, in robotic applications each feature description algorithm has been designed with strengths and weaknesses that make them more efficient in some scenarios compared to others [52]. Having a hardware accelerator that can be re-configured at runtime to transition between many of these algorithms when the environment changes is advantageous as compared to fixed hardware accelerators. A configurable hardware architecture can make prototyping new applications easier and faster for application developers. Changing the algorithmic structure for an existing application or designing a new application would not require writing HDL code, which can be prone to error and time consuming. Moreover, a configurable architecture can be used as an IP core to speed up prototyping embedded vision systems, thus reducing development effort.

Feature description algorithms often use histograms to encode a patch of pixels around interest points into a descriptor. However, these algorithms differ in: (1) Patch size, (2) Number of regions in
that patch, (3) regions’ patterns (rectangular or polar), (4) Number of histogram bins in each region, and (5) Type of low-level information (intensity, gradient, or LBP). In this work, we take advantage of common characteristics between different feature description algorithms to build a generic and flexible hardware accelerator. A major concern that arises when implementing a generic hardware architecture is hardware resources overhead. For this reason, we propose optimization techniques to reduce the computational complexity and hardware resource usage. The proposed architecture leverages data streaming to reduce the cost of computing and updating histogram values.

### 3.1 Related work

Generality and efficiency of computing systems are often inversely related to each other. The more general a computing system is, in terms of reasonably executing a wide range of tasks, the less efficient it tends to be in performing a specific task as compared to a system designed for only one task. It is important to find the right balance between these often competing characteristics. A general-purpose computing system such as CPUs are highly programmable but usually cannot achieve the highest performance. For example, computing SIFT descriptors (one of the most popular histogram-based description algorithms) for an image of size $640 \times 480$ takes 600 ms on a 3GHz CPU [88]. While dedicated hardware architectures can achieve high performance, but they are not programmable. Designers might be able to compromise small degradation in performance to increase system programmability.

Many dedicated hardware architectures for histogram-based feature descriptors with real-time performance have been proposed in the literature [51][20][88][46][87][65][64][91][34][79], but a configurable hardware architecture capable of supporting more than one descriptor algorithm with real time performance has not been proposed. However, efforts for implementing flexible hardware architectures for other image processing algorithms have been made [40][38][95][10]. In [40], a generic architecture for image feature detectors was presented. They proposed a generic hardware architecture that integrates two image feature detectors (Harris and SUSAN) in a single datapath. They proposed window-based image processing operators and a buffering scheme that allowed the archi-
tecture to adapt to different situations by changing its detection algorithm at run time. Another example of a generic architectures appears in [38]. They proposed a generic VHDL template for fast window-based stereo block matching correlation. It was fully scalable in functional parameters like image size, window size, and disparity range. In [95], the authors proposed a parallel hardware architecture for computing integral histogram images. The architecture was generic in terms of its input types: grayscale intensity, gradient, or binary pattern. In [10], a generic pixel distribution architecture for video processing was proposed. It could be configured to different window sizes and sliding steps in both horizontal and vertical directions.

In this work, we propose a software configurable hardware architecture that computes spatial histograms. Several parameters can be configured at run-time: window size, number of regions, number of bins. While our architecture supports computing any multiple-variable histogram, our discussion focuses on computing histograms of gradients, as it has many applications in computer vision.

### 3.2 Histogram-based descriptors

Histogram-based feature description algorithms are the most commonly used feature description algorithms in computer vision applications due to their robustness (stability under various image transformations and distortions), and distinctiveness (ability to capture and reflect distinction when feature information in image block changes locally). The process of computing these descriptors can be divided into four steps: (1) local patch accessing, (2) low-level information extraction, (3) feature pooling, and (4) vector normalization. In local patch accessing, a window of size N × N pixels are accessed from an image. In the second step, low-level information such as intensity, color, gradient or local binary pattern (LBP) is extracted from the local patch. The patch of pixels is divided into regions in order to enhance the discriminability of the computed descriptor. Then, the low-level information in each region is pooled to create a vector. The final descriptor is constructed by concatenating and normalizing the resulted vectors. Figure 3.1 shows these steps for computing
a histogram-based feature descriptor when intensity gradient is the low-level information and the patch is divided into 5 regions.

Many histogram-based feature descriptor algorithms have been proposed in the literature, such as SIFT [57], HOG [24], GLOH [61], DAISY [86], etc. These algorithms have similarities and differences. The main differences are: window sizes, number of regions, number of pixels in each region, regions’ pattern (rectangular, polar), number of histogram bins in each region. These differences give each algorithm strengths and weaknesses under different states and conditions, such as change in rotation and scaling, effect of blur, change in image intensity, and change in affine transformation. In this work, we propose a hardware architecture that can support different histogram-based feature descriptors. The proposed architecture is scalable to support any window size and can be configured to divide the window into any number of regions, and compute histograms in each region for up to 16 bins.

### 3.3 Histogram binning technique

The histogram binning technique used in our architecture take advantage of overlapping between windows as a sliding window moves across images. This technique reduces the number of operations required to compute histograms by avoiding redundant operations between consecutive windows. The next two sections illustrate in more detail the algorithm for computing 1D and 2D histograms of image gradients.

---

**Figure 3.1:** Four steps to compute histogram-based feature descriptors
3.3.1 One-dimensional histogram

Image gradients refer to the directional change in pixel intensity and can be represented using two values: a magnitude ($M$) and orientation ($\theta$). To compute histograms of gradients, the entire range of gradient orientation ($0^\circ \leq \theta < 360^\circ$) is divided into classes (also called bins) where $B$ is number of bins. The first histogram, $H_1$, of a window of size $N \times 1$ pixels is computed as shown in Equation (1), where $i$ represents the index of the first pixel. For each bin $\theta_k$, ($k = 1$ to $B$), the histogram value equals the summation of gradient magnitudes for all pixels that have $\theta_k \leq \theta_x < \theta_{k+1}$.

The next histogram $H_2(\theta_k)$ can be computed by updating the previously computed $H_1(\theta_k)$, as shown in Equations (2) and (3). Only one pixel is added to the sliding window and one pixel is removed. Figure 3.2 shows an example of computing the histogram of two consecutive windows ($H_1$ and $H_2$) for window size 5×1 and 4 bins. The dark regions represent the shared data points that did not exit the window as it slides one pixel to the right. $\forall \theta_k \in \{\theta_1, \theta_2, \ldots, \theta_B\}$

\[
H_1(\theta_k) = \sum_{x=i}^{i+N} M_x, \quad \text{if } \theta_k \leq \theta_x < \theta_{k+1} \tag{3.1}
\]

\[
H_2(\theta_k) = \sum_{x=i+1}^{i+N+1} M_x = \sum_{x=i}^{i+N} M_x - M_i + M_{i+N+1} \tag{3.2}
\]

\[
H_2(\theta_k) = H_1(\theta_k) - M_i + M_{i+N+1} \tag{3.3}
\]

Using this technique, we can save $N-2$ addition operations, by performing only one addition and one subtraction to compute histograms for a window of size $N \times 1$ pixels, instead of $N$ addition operations in the straightforward method. This reduces the computational complexity from $O(n)$ to $O(1)$, because the number of operations is fixed regardless of window size.

3.3.2 Two-dimensional histogram

The same idea can be extended to a window of size $N \times N$. Figure 3.3 shows an example for a $3 \times 3$ window. It shows computing histograms for two consecutive windows as the window slides...
one pixel to the right. The new histogram values can be computed by subtracting the left-most column and adding the right-most column to the previously computed histogram. This concept is applicable not only for regions with uniform shape, but also can be applied to any connected regions. Using this technique, we can compute histograms for a window of size $N \times N$ using only $N$ addition and $N$ subtraction operations $O(2N)$ instead of using $N \times N$ addition operations $O(N^2)$ in the straightforward approach. Equations (4-6) show the mathematical formulation for this technique, where $i$ and $j$ represent the index of the upper left corner pixel. Equation (4) shows the equation to compute histograms of the first window $H_1(\theta_k)$. Equations (5-6) show how the histogram of the next window $H_2(\theta_k)$ can be computed by only updating the previous histogram $H_1(\theta_k)$. $\forall \theta_k \in \{\theta_1, \theta_2, \ldots, \theta_n\}$

$$H_1(\theta_k) = \sum_{x=i}^{i+N} \sum_{y=j}^{j+N} M_{x,y}, \quad \text{if } \theta_k \leq \theta_x < \theta_{k+1} \quad (3.4)$$

$$H_2(\theta_k) = \sum_{x=i}^{i+N} \sum_{y=j}^{j+N} M_{x,y} - \sum_{y=j}^{j+N} M_{i,y} + \sum_{y=j}^{j+N} M_{i+N+1,y} \quad (3.5)$$

$$H_2(\theta_k) = H_1(\theta_k) - \sum_{y=j}^{j+N} M_{i,y} + \sum_{y=j}^{j+N} M_{i+N+1,y} \quad (3.6)$$

Figure 3.2: Computing 1D histogram of gradients
This section presents the proposed hardware architecture and its building blocks. The architecture consists of three modules: (1) Gradient computation block, to compute gradient magnitude and orientation at each pixel, (2) Two-dimensional array of processing elements (PEs) to compute histograms of gradients, and (3) Normalization block. The following three subsections illustrate each module in more detail.

### 3.4.1 Gradient computation block

For each pixel intensity value \( I(i, j) \) in an image, gradient magnitude \( M(i, j) \) and orientation \( \theta(i, j) \) can be computed as shown in Equations (7-10), where \( i \) and \( j \) represent pixel coordinates for the \( x \) and \( y \) axis. We use the sum of absolute values as an approximation to compute gradient magnitude. This method is used in [46] and [87] to avoid using the square root operation.
\[ \Delta x = I(i, j + 1) - I(i, j - 1) \]  \hspace{1cm} (3.7)  
\[ \Delta y = I(i + 1, j) - I(i - 1, j) \]  \hspace{1cm} (3.8)  
\[ M(i, j) = \sqrt{\Delta x^2 + \Delta y^2} \approx |\Delta x^2| + |\Delta y^2| \]  \hspace{1cm} (3.9)  
\[ \theta(i, j) = \arctan \frac{\Delta y}{\Delta x} \]  \hspace{1cm} (3.10)

The block diagram of the gradient computation module is shown in Figure 3.4. It buffers two image rows in two FIFOs to create a 3×3 sliding window of pixels. Two subtractors, two multipliers, one adder, and sign-bit checker logic is used to compute gradient magnitude based on Equation (9). To avoid using a large number of LUTs for storing precomputed arctangents, we used the coordinate rotation digital computer (CORDIC) algorithm in vector mode with 11 iterations to reach an accuracy of better than 1 degree. The output angle computed by CORDIC ranges from \(-\pi\) to \(\pi\). For this reason, 3 bits were used to represent an angle’s integer part, and 5 bits for its fractional part. We used 5 bits to keep precision within one degree. (1° = 0.0174 Radian).

![Figure 3.4: Gradient computation block with window size (3×3)](image)

### 3.4.2 A 2D array of processing elements (PEs)

A 2D array of processing elements arranged in a systolic structure is the main building block of our architecture. It is used to buffer gradient pixels generated by the gradient computation block, quantizes their orientations, and compute histograms. The internal architecture of one PE is shown in Figure 3.5. Each PE sends and receives data from/to its four neighbors using four input ports:
R(in), L(in), U(in), and D(in), and four output ports: R(out), L(out), U(out), and D(out). The size of these ports equal $NBins \times 2 \times 8$, where $NBins$ is number of histogram bins, each bin has 2 values for gradient magnitude (8 bits) and orientation (8 bits). Each PE stores one gradient magnitude and orientation ($M$ and $\theta$) locally. Every time a new gradient is computed, the PE updates its gradient value with the values coming from $pixel(in)$ and sends the old gradient value to its $pixel(out)$ port. It also contains a memory array (H) of size $NBins$, which is used to store the computed histogram values.

For every new gradient $pixel(in)$, a PE quantizes its gradient orientation into the nearest two angular bins. For example, if the PE is configured to compute histograms with 4 bins, that means the bin angles are $0^\circ$, $90^\circ$, $180^\circ$, and $270^\circ$. If the input gradient orientation is $215^\circ$, the orientation will be quantized into the closest two angles, which are $180^\circ$, and $270^\circ$. The gradient magnitude will be divided into two values using linear interpolation. Because $215^\circ$ is closer to $180^\circ$, the gradient magnitude for this bin will be higher than the $270^\circ$ bin. The quantitation unit is shown in Figure 3.5 with symbol (Q). It reads gradient angle $\theta$ and selects two bins of the $DEMUX0$ output.

8 bits ($S_7 - S_0$) are used to configure each PE. These bits configure four MUXs and one DEMUX. It allows PEs to compute different permutations of its four inputs and send the result to one of its four output ports, as shown in Figure 3.5. For example, $S_0$ controls MUX0 output to be D(in) or 0. The same concept applies to bits $S_{5-1}$. We used -R(in) to subtract the values that leave each region (algorithm proposed in Section IV), because we assume that the data is streaming in razer-scan (left to right). Finally, $S_{7-6}$ select the output port. For example, when $S_{7-6} = 00$, the computed value will be sent to the D(out) port, and when $S_{7-6} = 01,10,11$, the computed value will be sent to U(out), L(out), R(out), respectively.

Each PE can be configured into up $2^8$ different configurations, in Figure 3.6 we only show cases when a PE’s input port is not identical to its output port. For example, we did not show cases when PEs read data from D(in) and send the result back to D(out), as this is not a common case. Figure 3.6 shows different symbols used to distinguish between different configurations. Each symbol contains an arrow pointing to the output port and the tail of the arrow represents a PE’s
input ports. For example, when \( S=00000010 \), a PE sends its current gradient value into D(out) port, and when \( S=000000110 \), a PE adds the input L(in) to gradient value and sends the result into port D(out). The first row in Figure 3.6 shows different configurations of a PE when its output is D(out). The same is applied to the second row (U(in)), third row (L(out)) and fourth row (R(out)). The binary numbers over top of each symbol represents configuration bits \( S_{5-0} \).

Figure 3.6: Processing element configurations key

To compute histograms for a window of size \( N \times N \), we construct the 2D array architecture with \( N \times N \) PEs arranged in a systolic structure, as shown in Figure 3.7. It shows an example of \( 4 \times 4 \) array of PEs, where each PE is connected to its four neighbors. For simplicity, in this example, we...
configured PEs to compute histograms of three regions with number of bins equal to 1. The upper right section of Figure 3.7 shows the PEs’ configuration for computing histograms of three regions. PEs located on the right edge of each region are configured to send their gradient pixels to L(out). In our example, PEs (PE_{01}, PE_{11}, PE_{21}, PE_{03}, PE_{13}, PE_{23}, and PE_{33}) are configured to send their gradient pixels to L(out). The architecture needs to keep track of these pixels, because they need to be subtracted from the total histogram to keep the computed histograms correct. PEs located within regions such as (PE_{31}, and PE_{32}) are responsible of passing the values coming from their R(in) to L(out). PEs located at the left edge of each region such as (PE_{00}, PE_{10}, PE_{20}, PE_{02}, PE_{12}, and PE_{22}) are responsible for subtracting the values coming to their R(in) ports from their gradient pixel. The computed results are then passed to their U(out). For each region, one PE is needed to keep track of the computed histogram value. In our example, we used (PE_{00}, PE_{02}, and PE_{20}) to compute histograms of the three regions (H_1, H_2, and H_3).

Figure 3.8 shows an example of data streaming into a 4×4 array of PEs. Initially, all PEs have zeros in their local gradients and histograms. In the first cycle, PEs in the first column update their gradient values and their histograms, as follows: $H_1=(1-0)+(4-0)=5$, $H_3=(4-0)+(2-0)=6$, and $H_2=(0-0)+(0-0)+(0-0)=0$. In the next cycle, PEs in the first column will shift their gradients pixels
into the second column and recomputed their histograms, as follows: $H_1=(2-0)+(3-0)+ H_1 = 10$, $H_3=(1-0)+(2-0)+ H_3 = 9$, and $H_2=(0-0)+(0-0)+(0-0)=0$. In this step, $H_1$ has a valid value, which represents the summation of all pixels in the first region, since in this example histograms of 1 bin are computed. After 4 cycles, the computed histograms in each region will be valid: $H_1=10$, $H_3=18$, and $H_2=13$. After the initialization steps, the new pixels will be added to the corresponding region and old pixels will be subtracted to keep the computed histogram values correct.

Many image feature description algorithms can be implemented using our architecture. Figure 3.9 shows five examples of such algorithms: SIFT, GLOH, DAISY, AB-SIFT, and HOG, and their configurations. For SIFT, a window of size $16\times 16$ is divided into $4\times 4$ regions, and each region is configured into 8 bins. A SIFT descriptor is represented by a vector of length $4\times 4\times 8=128$. For GLOH, a window of size $16\times 16$ pixels is divided into a log-polar grid with three bins in the radial direction and 8 in the angular direction (17 regions). In each region, a histogram of 16 bins is computed. A GLOH descriptor length equals $17\times 16=272$. DAISY computes descriptors in 25 regions with 8 bins in each. DAISY descriptor length equals $25\times 8=200$. For AB-SIFT, the window is divided into a log-polar grid with three bins in the radial direction and (5, 8, and 10) in the angular direction. Histograms of (4, 6, and 8 bins) are computed. An AB-SIFT descriptor length equals $5\times 8+8\times 6+10\times 4=128$. Finally, HOG descriptors typically use a window of size $64\times 128$, but it can be computed using an array of size $8\times 8$ PEs and concatenating the resulting histograms.

![Figure 3.8: Configurations for SIFT, GLOH, DAISY, AB-SIFT and HOG algorithms](image)
Our architecture is scalable to compute histograms of any window size, number of regions, and number of bins (up to 16 bins). These algorithms are examples of gradient-based feature descriptors, but our architecture can be used with other low-level information such as intensity, or local binary pattern (LBP), etc. It also can be used to compute histograms for other applications with 2D spatial data representation.

3.4.3 Normalization

In this step, the resulting histograms from each region are normalized to a unit length to gain invariance to changes in illumination. The normalized histograms are computed by dividing their histogram values by the total sum of all histograms, as shown in Equation (11). Where $H_{i\text{ norm}}$ is the normalized histogram, $\|H\|$ is the summation of all elements of histograms in the window, and $\varepsilon$ is a constant to avoid a zero denominator. In the normalization unit, we used a division IP core to compute the inverse of the total sum (computed one time). The result is then multiplied by each histogram value as shown in Equation (11).

$$H_{i\text{ norm}} = \frac{H_i}{\|H\| + \varepsilon} = \left(\frac{1}{\|H\| + \varepsilon}\right) \ast H_i$$ (3.11)
3.5 Evaluation and verification

The proposed architecture is evaluated in terms of accuracy, execution speed, and resource usage. In this section, we present a detailed description of the evaluation procedures and comparison with related works. The accuracy is evaluated using SIFT descriptors matching rate, and detection rate of HOG descriptors. The execution speed is measured and compared with related work for computing one SIFT and HOG descriptor. This section also presents the hardware resources used to implement each block in the proposed architecture and the overall resource usage for different configurations.

3.5.1 Accuracy comparison

The accuracy of the proposed architecture is evaluated for two applications: (1) SIFT feature matching, and (2) Pedestrian detection using HOG and an SVM classifier.

3.5.1.1 SIFT feature matching

SIFT descriptors are computed by dividing a window of size $16 \times 16$ pixels into $4 \times 4$ regions, and computing histograms of gradients for 8 orientation bins in each region. The resulting SIFT descriptor is the concatenation of the 16 outputted histograms. The length of SIFT descriptors equal $16 \text{ (regions)} \times 8 \text{ (bins)} = 128$ elements. To validate our architecture, we compared the quality of SIFT descriptors computed by software implementation (MATLAB) and our architecture using a feature matching problem. We used the Oxford dataset [62] in our experiments. It consists of 8 image sets with 5 different imaging conditions: viewpoint changes (graf, wall imagesets), scale changes (bark, boat imagesets), image blur (bikes, trees imagesets), JPEG compression (ubc imageset), illumination (leuven imageset).

These experiments consist of four steps: (1) SIFT descriptors are computed for two images of each image set, (2) descriptor matches are found by computing the distance between each descriptor in the first image and all descriptors detected in the second image. If the sum of absolute differences (SAD) is less than a threshold, it is considered a match. The threshold determines the
strictness of matching criterion. (3) Compute true positive rate and false positive rate. A match is considered a true match, if the corresponding features have overlapping pixels after computing the homography between the two images. (4) change the threshold value to draw the Receiver Operating Characteristic (ROC) curve.

Matching ability of a descriptor is measured by the area under the ROC curve. An area of 1 represents a perfect descriptor, and an area of 0.5 represents random guesses. Figure 3.10 shows ROC curves computed for the software implementation and our architecture for the 8 benchmarks of the Oxford dataset. Table 3.1 also shows the computed areas under these curves. To compare ROC curves, we used a statistical significance test (Hanley method). It shows the difference in area under the ROC curves are insignificant for the matching problem. The results of hardware and software do not exactly match due to using fixed point representation for gradient magnitude and orientation instead of using floating point and using linear interpolation instead of trilinear interpolation. An exact match with software can be achieved at the cost of additional hardware resources.

Table 3.1: Area under ROC (AUC) for software and hardware

<table>
<thead>
<tr>
<th>Benc</th>
<th>ubc</th>
<th>leuven</th>
<th>bikes</th>
<th>wall</th>
<th>graf</th>
<th>trees</th>
<th>boat</th>
<th>bark</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software</td>
<td>0.94</td>
<td>0.88</td>
<td>0.85</td>
<td>0.79</td>
<td>0.56</td>
<td>0.82</td>
<td>0.67</td>
<td>0.58</td>
</tr>
<tr>
<td>Hardware</td>
<td>0.94</td>
<td>0.88</td>
<td>0.84</td>
<td>0.79</td>
<td>0.54</td>
<td>0.81</td>
<td>0.65</td>
<td>0.56</td>
</tr>
</tbody>
</table>

Figure 3.10: ROC curves for the 8 oxford image sets
3.5.1.2 Pedestrian Detection

In this experiment, we used a HOG descriptor with a linear SVM classifier to detect pedestrians in a diverse set of images from the CVC-02 dataset [32]. The detector uses a sliding window of size $64 \times 128$. At each position, a HOG descriptor is computed, and fed to a trained SVM to classify it as either person or not a person. To recognize persons at different scales, the image is down sampled to multiple sizes. To train our SVM classifier, we used 1016 positive and 7650 negative images. In the testing stage, images are scanned and a list of $(x,y)$ coordinates of the top-left corner of each window containing a person is generated. This list is then compared to ground truth to find the true positive and false positive rate. To validate our architecture, we implemented a MATLAB model that represents precisely the behavior of the implemented hardware. We ran the experiment on 250 images containing 587 annotated pedestrians, the proposed hardware achieved detection accuracy of 86.2% with 4.7% false negative, while the software implementation (floating point) achieved accuracy of 86.2% with 4.7% false negative. This shows that our hardware implementation achieved the same accuracy as the software implementation for the pedestrian detection application.

3.5.2 Hardware resource utilization

This section presents the Post-Synthesis hardware resource utilization for each block in our architecture. Table 3.2, Table 3.3 and Table 3.4 show number of LUTs, registers, DSPs and the maximum operating frequency for PE unit, gradient computation, and normalization blocks, respectively. We used Xilinx Vivado 2017.2 and a Xilinx Zynq-7 ZC706 FPGA in our experiments. It has a total of 218,600 LUTs, 437,200 registers, and 900 DSPs. Table 3.2 shows the resource usage for one processing element (PE) when the number of bins equals 2, 4, 8, and 16. Increasing the number of bins in a PE unit increases the number of addition/subtractions operations and increases the size of its memory array. This explains the trend in LUTs and FFs shown in Table 3.2. The gradient computation block uses 2 BRAMs (18Kb) to buffer image rows and build a $3 \times 3$ window of pixels. Table 3.5 shows the resource usage for a 2D array of PEs for different numbers of bins (2, 4, 8, and 16) and window sizes ($4 \times 4$, $8 \times 8$, and $16 \times 16$). It shows that LUTs and FFs scale
linearly with the number of bins for a given window size. It also shows that LUTs and FFs scale exponentially with window size, for the same number of bins.

Table 3.2: Processing element hardware resources

<table>
<thead>
<tr>
<th>Num. Bins</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>Max. Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>112</td>
<td>0</td>
<td>614 MHz</td>
</tr>
<tr>
<td>4</td>
<td>221</td>
<td>208</td>
<td>0</td>
<td>614 MHz</td>
</tr>
<tr>
<td>8</td>
<td>492</td>
<td>400</td>
<td>0</td>
<td>614 MHz</td>
</tr>
<tr>
<td>16</td>
<td>939</td>
<td>784</td>
<td>0</td>
<td>614 MHz</td>
</tr>
</tbody>
</table>

Table 3.3: Gradient computation block hardware resources

<table>
<thead>
<tr>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>BRAMs</th>
<th>Max. Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>469</td>
<td>180</td>
<td>2</td>
<td>2</td>
<td>360 MHz</td>
</tr>
</tbody>
</table>

Table 3.4: Normalization hardware resources

<table>
<thead>
<tr>
<th>Num. Bins</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
<th>Max. Freq</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>619</td>
<td>742</td>
<td>32</td>
<td>488 MHz</td>
</tr>
<tr>
<td>4</td>
<td>1237</td>
<td>1482</td>
<td>64</td>
<td>488 MHz</td>
</tr>
<tr>
<td>8</td>
<td>2472</td>
<td>2962</td>
<td>128</td>
<td>488 MHz</td>
</tr>
<tr>
<td>16</td>
<td>4943</td>
<td>5922</td>
<td>256</td>
<td>488 MHz</td>
</tr>
</tbody>
</table>

3.5.3 Comparison with dedicated hardware architectures

To measure the performance of the proposed architecture, we compared the execution time of our implementation with OpenCV implementations of SIFT and HOG algorithms running on an ARM Dual-Core Cortex-A9 and an Intel Core i7 processors. We also compare execution times with other hardware architectures implementing the SIFT and HOG algorithms. Table 3.6 shows the time required to compute one SIFT descriptor, frame sizes, frame rates, and maximum number of SIFT descriptors that can be computed within the frame rate mentioned (KP Ratio). KP ratio is computed as shown in Equation (12). It represents the percentage of pixels that can be processed at a specific frame rate. Table 3.6 shows that our architecture with window sizes, 16×16, 8×8,
Table 3.5: 2D array of PE hardware resources

<table>
<thead>
<tr>
<th>Window Size</th>
<th>Num. Bins</th>
<th>LUTs</th>
<th>FFs</th>
<th>BRAMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>4×4</td>
<td>2</td>
<td>1,568</td>
<td>1,543</td>
<td>3</td>
</tr>
<tr>
<td>4×4</td>
<td>4</td>
<td>3,437</td>
<td>2,952</td>
<td>3</td>
</tr>
<tr>
<td>4×4</td>
<td>8</td>
<td>6,724</td>
<td>5,518</td>
<td>3</td>
</tr>
<tr>
<td>4×4</td>
<td>16</td>
<td>13,274</td>
<td>10,638</td>
<td>3</td>
</tr>
<tr>
<td>8×8</td>
<td>2</td>
<td>7,462</td>
<td>6,875</td>
<td>7</td>
</tr>
<tr>
<td>8×8</td>
<td>4</td>
<td>15,549</td>
<td>12,556</td>
<td>7</td>
</tr>
<tr>
<td>8×8</td>
<td>8</td>
<td>29,930</td>
<td>23,862</td>
<td>7</td>
</tr>
<tr>
<td>8×8</td>
<td>16</td>
<td>61,011</td>
<td>46,390</td>
<td>7</td>
</tr>
<tr>
<td>16×16</td>
<td>2</td>
<td>29,952</td>
<td>27,859</td>
<td>15</td>
</tr>
<tr>
<td>16×16</td>
<td>4</td>
<td>63,256</td>
<td>51,659</td>
<td>15</td>
</tr>
<tr>
<td>16×16</td>
<td>8</td>
<td>125,790</td>
<td>98,950</td>
<td>15</td>
</tr>
<tr>
<td>16×16</td>
<td>16</td>
<td>221,626</td>
<td>192,512</td>
<td>15</td>
</tr>
</tbody>
</table>

and 4×4 achieved a speed-up of 4-5 order of magnitudes compared to the ARM processor and 3-4 order of magnitudes compared to the Intel Core i7. When the fully parallel architecture P(16×16) is used, our architecture outperforms the work done in [87] by 3×, because our architecture can run at 167 MHz while the work in [87] is running at 21.7 MHz (one cycle= 46ns) only. However, our architecture shows an overhead in resource usage of 1.1×, 15×, and 1.6× for LUT, FF, and DSPs. This is due to the generality of our architecture to support many algorithms instead of only one. For the sequential implementation P(8×8), our architecture can compute one SIFT descriptor in 0.37μs (64 cycles) which outperforms other hardware architectures by 6.1×, but with a resource overhead of 1.2×, 2.7×, and 1.5× for LUT, FF, and DSPs compared to the fastest sequential implementation in [46]. In the sequential implementation of P(4×4), our architecture can compute one SIFT descriptor in 0.75μs (128 cycles), which achieved a speed up of 2.9×, with resource overhead of 0.8×, 0.68×, and 1.6× for LUT, FF, and DSPs, as shown in Table 3.7.

\[
KP\ Ratio(\%) = \frac{Frequency(Mcycles/sec)}{Size \times Rate \times (#Cycles/keypoints)}
\] (3.12)

Table 3.8 compares the performance of the proposed architecture with an OpenCV implementation of HOG running on an ARM Dual-Core Cortex-A9, Intel Core i7 processors, and dedicated FPGA architectures proposed in the literature. It shows frame sizes and the maximum frame rates achieved by each work. It order to find a common metric, we computed pixels/second from frame
Table 3.6: SIFT descriptor extraction time

<table>
<thead>
<tr>
<th>Paper</th>
<th>Exec. time</th>
<th>Frame Size</th>
<th>Frame Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM A9</td>
<td>5.13 ms</td>
<td>800×640</td>
<td>0.11 fps</td>
</tr>
<tr>
<td>Intel i7</td>
<td>0.49 ms</td>
<td>800×640</td>
<td>0.89 fps</td>
</tr>
<tr>
<td>Kim [51]</td>
<td>60 µs</td>
<td>320×240</td>
<td>30 fps, feat.&lt;550</td>
</tr>
<tr>
<td>Chiu [20]</td>
<td>5.5 µs</td>
<td>640×480</td>
<td>30 fps, feat.&lt;6000</td>
</tr>
<tr>
<td>Wang [88]</td>
<td>8.3 µs</td>
<td>1280×720</td>
<td>60 fps, feat.&lt;2000</td>
</tr>
<tr>
<td>Jiang [46]</td>
<td>2.23 µs</td>
<td>512×512</td>
<td>150 fps, feat.&lt;2900</td>
</tr>
<tr>
<td>John [87]</td>
<td>46 ns</td>
<td>640×480</td>
<td>70 fps (parallel)</td>
</tr>
<tr>
<td>P(16×16)</td>
<td>5.9 ns</td>
<td>800×640</td>
<td>195 fps (parallel)</td>
</tr>
<tr>
<td>P(08×08)</td>
<td>0.37 µs</td>
<td>800×640</td>
<td>60 fps, feat.&lt;2600</td>
</tr>
<tr>
<td>P(04×04)</td>
<td>0.75 µs</td>
<td>800×640</td>
<td>30 fps, feat.&lt;2600</td>
</tr>
</tbody>
</table>

Table 3.7: SIFT hardware architectures

<table>
<thead>
<tr>
<th>Paper</th>
<th>FPGA</th>
<th>Max Freq</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kim [51]</td>
<td>StratixII</td>
<td>50 MHz</td>
<td>16,832</td>
<td>5,729</td>
<td>8</td>
</tr>
<tr>
<td>Chiu [20]</td>
<td>Virtex-6</td>
<td>100 MHz</td>
<td>57,598</td>
<td>24,988</td>
<td>8</td>
</tr>
<tr>
<td>Wang [88]</td>
<td>Virtex-5</td>
<td>159 MHz</td>
<td>18,437</td>
<td>13,007</td>
<td>52</td>
</tr>
<tr>
<td>Jiang [46]</td>
<td>Virtex-5</td>
<td>79.4 MHz</td>
<td>26,398</td>
<td>10,310</td>
<td>89</td>
</tr>
<tr>
<td>John [87]</td>
<td>CycloneIV</td>
<td>21.7 MHz</td>
<td>120,917</td>
<td>6,719</td>
<td>77</td>
</tr>
<tr>
<td>P(16×16)</td>
<td>ZYNQ-7</td>
<td>167 MHz</td>
<td>128,731</td>
<td>102,092</td>
<td>130</td>
</tr>
<tr>
<td>P(08×08)</td>
<td>ZYNQ-7</td>
<td>167 MHz</td>
<td>32,871</td>
<td>28,004</td>
<td>130</td>
</tr>
<tr>
<td>P(04×04)</td>
<td>ZYNQ-7</td>
<td>167 MHz</td>
<td>10,093</td>
<td>9,542</td>
<td>130</td>
</tr>
</tbody>
</table>

rates and sizes, as shown in Equation (13). To compute our architecture frame rate, we multiply the number of HOG cells in one frame by the number of clock cycles needed to finish one cell, and divide the result by the maximum operational frequency, as shown in Equation (14). Table 3.8 shows that our architecture achieved a speed-up of 1-2 order of magnitude compared to the ARM processor and 1 order of magnitude compared to the Intel Core i7. It also shows that our architecture P(16×16) outperforms other works by 3-46×, because it can compute one HOG cell in one clock cycle, but at the expense of resources. P(16×16) has resource usage overheads of 6.4×, 17×, and 32× for LUT, FF, and DSPs compared to the latest implementation in [79]. Our sequential architectures, P(8×8) and P(4×4), shows a compromise in terms of performance to reduce the resource usage overhead as shown in Table 3.9.
\[ \text{Pixels/Second} = \text{Frame Size} \times \text{Frame Rate} \]  
\[ \text{Frame Rate} = \frac{(\# \text{HOG Cells}) \times (\# \text{Cycles/Cells})}{\text{Frequency (Mcycles/sec)}} \]  
\[ \# \text{HOG Cells} = \frac{\text{Frame Size}}{\text{Cell Size} (8 \times 8)} \]  

Table 3.8: HOG descriptor extraction time

<table>
<thead>
<tr>
<th>Paper</th>
<th>Frame Size</th>
<th>Frame Rate</th>
<th>Pixels/Second</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM A9</td>
<td>1920×1080</td>
<td>0.63</td>
<td>1.3×10^6</td>
</tr>
<tr>
<td>Intel Core i7</td>
<td>1920×1080</td>
<td>5.31</td>
<td>11.1×10^6</td>
</tr>
<tr>
<td>Negi [65]</td>
<td>320×240</td>
<td>112</td>
<td>8.6×10^6</td>
</tr>
<tr>
<td>Mizuno [64]</td>
<td>1920×1080</td>
<td>30</td>
<td>62.2×10^6</td>
</tr>
<tr>
<td>Xie [91]</td>
<td>320×240</td>
<td>293</td>
<td>22.5×10^6</td>
</tr>
<tr>
<td>Hahnle [34]</td>
<td>1920×1080</td>
<td>64</td>
<td>132.7×10^6</td>
</tr>
<tr>
<td>Rettkowski [79]</td>
<td>1920×1080</td>
<td>40</td>
<td>82.9×10^6</td>
</tr>
<tr>
<td>P(16×16)</td>
<td>1920×1080</td>
<td>192</td>
<td>398.1×10^6</td>
</tr>
<tr>
<td>P(8×8)</td>
<td>1920×1080</td>
<td>48</td>
<td>99.5×10^6</td>
</tr>
<tr>
<td>P(4×4)</td>
<td>1920×1080</td>
<td>12</td>
<td>24.8×10^6</td>
</tr>
</tbody>
</table>

Table 3.9: HOG hardware architectures

<table>
<thead>
<tr>
<th>Paper</th>
<th>FPGA</th>
<th>Freq.</th>
<th>LUTs</th>
<th>FFs</th>
<th>DSPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negi [65]</td>
<td>Virtex-5</td>
<td>44.8MHz</td>
<td>17,383</td>
<td>2,181</td>
<td>0</td>
</tr>
<tr>
<td>Mizuno [64]</td>
<td>Cyclone IV</td>
<td>76.2MHz</td>
<td>34,403</td>
<td>23,247</td>
<td>68</td>
</tr>
<tr>
<td>Xie [91]</td>
<td>Spartan-3e</td>
<td>67.7MHz</td>
<td>3,379</td>
<td>2,041</td>
<td>0</td>
</tr>
<tr>
<td>Hahnle [34]</td>
<td>Virtex-5</td>
<td>270MHz</td>
<td>5,188</td>
<td>5,176</td>
<td>49</td>
</tr>
<tr>
<td>Jens [79]</td>
<td>XC7z020</td>
<td>82.2MHz</td>
<td>21,297</td>
<td>5,942</td>
<td>4</td>
</tr>
<tr>
<td>P(16×16)</td>
<td>ZYNQ-7</td>
<td>167MHz</td>
<td>128,731</td>
<td>102,092</td>
<td>130</td>
</tr>
<tr>
<td>P(8×8)</td>
<td>ZYNQ-7</td>
<td>167MHz</td>
<td>32,871</td>
<td>28,004</td>
<td>130</td>
</tr>
<tr>
<td>P(4×4)</td>
<td>ZYNQ-7</td>
<td>167MHz</td>
<td>10,093</td>
<td>9,542</td>
<td>130</td>
</tr>
</tbody>
</table>

In this work, we proposed a configurable hardware architecture for computing different histogram-based feature description algorithms. The proposed architecture is configurable in terms of window size, number of regions, number of bins per region, and the pattern of these regions. We implemented different optimization techniques to reduce hardware resources and computational complexity for computing histograms of gradients. The experimental results show that our architecture can be used in a wide range of computer vision applications.
Figure 3.11: SIFT implementations (hardware cost vs. execution time)

Figure 3.12: HOG implementations (hardware cost vs. execution time)

Figure 3.13: HOG implementations (accuracy vs. energy)
CHAPTER 4. ANALYZING THE ENERGY EFFICIENCY OF VISION KERNELES

Developing efficient embedded vision applications requires exploring various algorithmic optimization trade-offs and a broad spectrum of hardware architecture choices. This makes navigating the solution space and finding the design points with optimal performance trade-offs a challenge for developers. To help provide a fair baseline comparison, we conducted comprehensive benchmarks of accuracy, run-time, and energy efficiency of a wide range of vision kernels and neural networks on multiple embedded platforms: ARM57 CPU, Nvidia Jetson TX2 GPU and Xilinx ZCU102 FPGA. Each platform utilizes their optimized libraries for vision kernels (OpenCV, VisionWorks and xfOpenCV) and neural networks (OpenCV DNN, TensorRT and Xilinx DPU). For vision kernels, our results show that the GPU achieves an energy/frame reduction ratio of 1.1–3.2× compared to the others for simple kernels. However, for more complicated kernels and complete vision pipelines, the FPGA outperforms the others with energy/frame reduction ratios of 1.2–22.3×. For neural networks [Inception-v2 and ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNet], it shows that the FPGA achieves a speed up of [2.5, 2.1, 2.6, 2.9 and 2.5]× and an EDP reduction ratio of [1.5, 1.1, 1.4, 2.4 and 1.7]× compared to the GPU FP16 implementations, respectively.

4.1 Introduction

Computer vision empowered with the recent advances in deep learning plays a fundamental role in solving many problems that seemed impossible just a decade ago. The computational complexity and memory footprint of these algorithms keep increasing to enhance accuracy or solve more complex problems [71]. This trend is driving the development of energy-efficient processing solutions, which are especially important for energy or thermal constrained real-time embedded
systems. Often their limited communication power budget or communication capabilities preclude them from streaming images to more powerful computing entities.

Both industry and academia have explored the development of acceleration engines to help meet the needs of embedded vision applications. Three common types of such accelerators are benchmarked in this case study: multicore CPUs, Graphic Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). Each of these accelerators takes a different approach to accelerating embedded vision applications. Multi-core CPUs make use of SIMD instruction extensions, such as: the ARM NEON SIMD engine, Intel’s family of SSE, and dedicated vision processing units (VPU), such as Myriad [45]. The multi-threading programming model has made GPUs highly popular in this domain. GPUs provide massively parallel execution resources and high memory bandwidth. However, their high performance comes at the cost of high power dissipation [21]. FPGAs offer opportunities for using low-level fine-grained parallelism by customizing processing/control units and data paths to the requirements of a specific algorithm or application [33].

Embedded vision applications can exhibit vastly different performance characteristics depending on their underlying hardware accelerator platform and compute fabrics [18]. This varying behavior fundamentally stems from differences in accelerator micro architectures, middleware support, and programming styles. This mixture of factors makes choosing the best application-to-accelerator mapping a nontrivial task for embedded vision application developers. They must take into consideration metrics, such as expected runtime performance, energy-efficiency, and programmability. Moreover, running vision pipelines on heterogeneous platforms requires partitioning them into phases that can run on the available accelerators in the most efficient and cost-effective manner.

Beside the broad spectrum of hardware architectures choices for vision applications, there are various complexity-accuracy algorithmic trade-offs that need to be explored [69]. Quantization and pruning are two examples of optimization methods that are used to reduce the computational complexity and memory requirement of vision algorithms but at the expense of accuracy loss. Another design choice that need to be explored is deciding whether neural network solutions are more
suitable choice compared to the traditional vision kernels for a specific vision application. The traditional (hand-engineered) algorithms are mature, proven, and optimized for performance and power efficiency, while neural networks (learned algorithms) offer greater accuracy and versatility, but demand large amounts of computing and resources. Moreover, the modeling capacity of traditional vision kernels are limited by the fixed transformations (filters) that stay the same for different sources of data. While learned features are data-driven and adapt based on the training data. Therefore, the complexity-accuracy trade-offs between neural networks solutions and traditional vision kernels need to be taken into consideration during the development process.

In order to clearly understand how different hardware architectures may impact the performance of vision kernels and neural networks, we analyze their performance on such accelerators. In this work, we evaluate the performance of three commonly used HW accelerators for vision applications: the ARM Cortex A57 CPU, Jetson TX2 GPU, and ZCU102 FPGA in terms of accuracy, run-time performance (latency and throughput), and energy efficiency. For vision kernel benchmarking, we propose an easily reproducible approach that only uses publicly available vision libraries: OpenCV, Nvidia VisionWorks and xfOpenCV, without adding any special platform specific code. We also evaluate the performance of neural network inference implementation of [Inception-v2 and ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNet] using OpenCV DNN module, Nvidia TensorRT and Xilinx Vitis AI frameworks running on these accelerators. All benchmark code is available at: https://github.com/isu-rcl/cvBench.

4.2 Related work

In this section, we take a look at existing benchmarking efforts in the literature that evaluate the performance of vision kernels and neural networks on embedded platforms. Even though most prior benchmarking efforts focus solely on comparing the performance of a limited number of vision kernels or cover only subsections of the embedded design space in evaluating neural networks performance, there are a few exceptions discussed in this section.
Vision kernels benchmarks. The comparison study in [22] analyzed the performance efficiency of FPGAs and GPUs on the GPU-friendly benchmark suite (Rodinia). They ported 15 of its kernels using Vivado HLS for the FPGA and OpenCL for host programs. The platforms used were a Xilinx Virtex-7 FPGA and Nvidia Tesla K40c GPU. Although this study included some vision kernels such as: GICOV, Dilate, SRAD and MGVF, it was not mainly focused on benchmarking vision algorithms; it included other kernels for data mining, fluid dynamic, and physics simulation, etc [17].

Other comparison studies each focused on a subset of vision kernels. For example, the study in [23] and [29] evaluated the performance of sliding window applications on FPGAs, GPUs and multicore CPUs. They compared the performance of three applications: Sum of Absolute Differences (SAD), 2D convolution, and correntropy. The platforms used in their study were an Altera Stratix IV FPGA, an Nvidia GeForce GTX 560, and an Intel Xeon Core i7. Another study in [16] focused on comparing the performance of morphological image filtering operations. The authors utilized the OpenCV library for CPU and GPU (cv::CUDA module) implementations. For the FPGA platform, they used Vivado HLS video libraries and hand-optimized implementations. The platforms used in their study were the Xilinx Zynq 7020 FPGA, Nvidia Tegra K1, and Intel core i7. The work in [30] also focused only on a subset of vision operations such as normalized cross correlation and finite impulse response (FIR) filters. This study’s evaluation included development time, component cost, and power consumption.

Neural networks benchmarks. There are two types of machine learning benchmarks based on the classification of [13]: (1) Machine learning (ML) benchmarks focus mainly on achieving high test accuracy, independent of the hardware implications. Examples of this kind of benchmarks are ILSVRC ImageNet competition [25] and MLBench [55], (2) Performance benchmarks focus on measuring performance metrics such as latency, throughput and power consumption. This category of benchmarks give algorithmic modifications freedom to reach the highest performance. Examples include DeepBench [5], SPEC [7] and STREAM [59]. A more complete benchmarking suite has been proposed in QuTiBench [13]. QuTiBench is a novel multi-tiered benchmarking methodology.
that supports algorithmic optimizations and couple hardware performance with accuracy at the
application level. It includes test suites at 4 levels of abstraction: (1) level-0 includes roofline
analysis that provides insight into the memory and compute requirements, (2) level-1 focuses on
the achievable compute performance for different compute patterns, (3) level-2 captures potential
bottlenecks in data movements, and (4) level-3 covers the system-level performance.

Our benchmark is exhaustive and energy-efficiency focused: we evaluate the accuracy, run-time,
and energy consumption of different embedded hardware platforms over a wide range of standard
vision kernels, vision pipelines and neural networks. The results are easily reproducible through
the use of open-source benchmarking templates that only use publicly available vision and neural
network libraries.

4.3 Background

In this section, we first present the characteristics of the hardware accelerators evaluated in
this study. Then, we briefly discuss three vision libraries and neural network inference frameworks
that are widely used with these accelerators. We group the vision kernels into categories based
on their characteristics to understand the implications of the underlying hardware architectures on
the performance of these kernels in their respective categories. Finally, we provide details on the
used neural networks models and their architectures.

4.3.1 Embedded platforms

The following are the three most common platforms used in embedded vision applications:

4.3.1.1 Central processing unit (CPU)

Modern CPUs are able to perform SIMD (Single Instruction, Multiple Data) instructions using
multiple ALUs. Such processing scheme exploit data level parallelism; there are simultaneous
(parallel) computations, but only a single process (instruction) at a given moment. These SIMD
instruction sets are useful in the context of image processing, where operations are often repetitively
applied to a continuous stream of data. This is particularly true in the context of computer vision, where most operations are performed over the entire image. Examples of SIMD architectures are: ARM NEON SIMD engine [1] and Intel’s streaming SIMD extensions (SSE) [6].

4.3.1.2 Graphic processing unit (GPU)

As compared to general purpose CPUs, which have developed SIMD instruction extensions to help parallelize image processing type tasks, GPUs have taken the direction of evolving into a specialized SIMD architecture. This specialization has led to GPUs having simpler processing cores than high-performance general purpose CPUs. For example, they have simpler control logic, typically no branch prediction or prefetch, and small per-core memory. Simpler computing cores allow GPUs to pack many more cores into a chip than a general purpose CPU. GPU architectures perform extremely well on workloads that have little to no branching conditions or data dependences. Additionally, GPU architectures have specialized their memory architecture to support high-speed data streaming for image processing. For example, the L2 cache in the Jetson TX2 (Pascal GPU) is 2048 KB, which can fit a 1080p grayscale image.

4.3.1.3 Field programmable gate array (FPGA)

Instead of having a fixed processor-like design, FPGAs consist of an array of logic blocks, DSPs, on-chip BRAMs, I/O pads, and routing channels. In FPGA, custom data paths can be architectured to stream pixels directly between computing units without needing to read/write from/to external memory. Moreover, the distributed on-chip BRAMs can be used to exploit data locality in vision kernels by keeping pixels on-chip (e.g. Zynq UltraScale MPSoC FPGA has 32.1 Mb on-chip memory). With FPGAs, developers need to ensure that their customized designs meet timing and space requirements.
4.3.2 Computer vision libraries

A number of vision libraries have been optimized to target the hardware platforms discussed in the previous section. In this work, we focused on the most complete and commonly used libraries, as follows:

4.3.2.1 OpenCV

OpenCV (Open Source Computer Vision Library) is the de-facto standard C/C++ library for image and vision processing [15]. It is used by the computer vision community to create desktop and embedded vision applications. It has more than 2500 optimized vision kernels, which includes a comprehensive set of both traditional and state-of-the-art vision and machine learning algorithms. OpenCV has bindings for languages such as Python and Java. The latest version of OpenCV (at the time of writing this chapter) is 4.1.1.

4.3.2.2 NVIDIA visionWorks

VisionWorks is a toolkit for computer vision and image processing released by Nvidia in 2015 [67]. It implements and extends the OpenVX standard, and is optimized for CUDA-capable GPUs. VisionWorks provides three programming models: (1) immediate mode which enables developers to easily port their applications, (2) graph mode which enables advanced optimizations such as: buffer reuse, efficient use of streaming and CUDA textures, tiling and pipelining functions at sub-frame level, and (3) CUDA API which enables developer with low-level access to manage data allocations and transfer, scheduling and pipelining. The latest version of VisionWorks is 1.6.

4.3.2.3 Xilinx xfOpenCV

The xfOpenCV library is a set of OpenCV functions optimized for Zynq, Zynq Ultrascale+, and Alveo FPGAs devices by Xilinx [92]. It was first released in 2017, as part of the Xilinx reVISION stack. xfOpenCV kernels are implemented using HLS to work in their SDx development environment and provides a software interface for building vision pipelines on FPGAs. The library
includes a set of 60+ vision kernels optimized to be mapped into the programmable logic. The latest version of the xfOpenCV library is 2019.1.

### 4.3.3 Neural network inference frameworks

In this work, we used the following three deep learning inference frameworks to benchmark neural networks:

#### 4.3.3.1 OpenCV DNN module

OpenCV DNN module has been promoted from the contrib repository to the main repository since the release of OpenCV 3.3 [4]. The module now supports deep learning frameworks such as Caffe, TensorFlow, and Torch/PyTorch. It only supports the forward pass by importing weights from pre-trained models. OpenCV DNN also includes a set of pre-processing functions for preparing images, such as: cropping, channel swapping, mean subtraction, etc. Examples of compatible network architectures include: GoogleLeNet, AlexNet, SqueezeNet, VGGNet and ResNet. The module supports SSE, AVX, NEON acceleration and Halide backend.

#### 4.3.3.2 NVIDIA TensorRT

TensorRT is a framework for implementing high-performance inference on NVIDIA GPUs [68]. TensorRT applies couple of optimizations to deep learning networks such as: (1) Weight and activation precision quantization to FP16 and INT8 to maximizes throughput while maintaining accuracy, (2) Optimizing the use of GPU memory and bandwidth by layer and tensor fusion, (3) Kernel auto-tuning to select best data layers and algorithms based on target GPU platform, (4) Dynamic tensor memory allocation to re-use memory efficiently, (5) Multi-stream execution to process multiple input streams in parallel. The integration of TensorRT with TensorFlow allows for applying TensorRT GPU optimizations within TensorFlow environment.
4.3.3.3 Xilinx Vitis AI

Xilinx Vitis AI is a deep learning framework that provides a combination of flexibility, high performance, low latency and low power consumption for deploying deep learning inference into Xilinx FPGAs and SoCs [8]. It allows for compressing DNN models to reduce their size without loss of accuracy, and compiling DNN models into DPU instruction code before deploying them into the target DPU platform. Xilinx DPU provides a customized and scalable overlay with ISA architecture for optimized DNNs implementations.

4.3.4 Categories of vision kernels

Computer vision algorithms can be grouped into six categories based on their functionality. The complexity of these kernels grows over the first five categories. The last category includes composite kernels, which are composed of kernels from other categories. The following discusses each category in more detail:

4.3.4.1 Input processing

The kernels in this group are usually used as pre-processing steps. They include simple arithmetic operations to change the input format or number of channels into a desired format. Some examples of these kernels are: channel combine, channel extract, color conversion, and bitdepth conversion.

4.3.4.2 Image arithmetic

Image arithmetic applies standard arithmetic/logic operations to one or more images. Because of the multi-dimensional nature of these pixel based operations, these kernels can benefit from highly parallel hardware architectures, such as GPUs and FPGAs. Furthermore, the data being processed is very localized; the algorithms can be distributed among different processing units without concerns of data dependencies. These operations include: thresholding, absolute
difference, addition/subtraction, bitwise and/or/xor/not, multiplication, accumulate, accumulate squared, and accumulate weighted.

4.3.4.3 Image filters

These algorithms compute the correlation between an input image and a kernel (small matrix of fixed-size). The data in these algorithms are local to the size of the kernel which is different from the arithmetic case where the operations were performed on a pixel basis. When the underlying hardware has enough local memory to accommodate the kernel size, the algorithm is still easily distributed among parallel processing units. On the other hand, nonlinear filters are more irregular as they have branching conditions. This impedes their decomposition into parallel blocks. These kernels include: filter2D, box filter, erode, dilate, median, pyramid up, and pyramid down.

4.3.4.4 Image analysis

Analytic kernels are typically used to understand characteristics of an image, such as color distribution, mean, maximum and minimum pixel value, etc. Also, they are usually placed at the end of vision pipelines to reduce the image into a decision variable (min/max locations). These kernels are filled with branching conditions and complex memory access patterns that negatively impact their performance on CPUs and GPUs. These operations include: histogram, mean/std, min/max location, table lookup, histogram equalization, and integral image.

4.3.4.5 Geometric transformation

Transformations in geometric space are essential to understanding the 3D world through the lens of a 2D image sensor. These kernels include matrix multiplication that map effectively into highly parallel architectures composed of simple computing blocks (e.g. GPU). While these kernels are simple, their performance is negatively affected by irregular memory access patterns. These kernels include: remap, resize, affine warp, and perspective warp.
4.3.4.6 Composite kernels

The kernels in this category are composed in part of kernels from the previously described categories. Examples of these composite kernels are: feature extraction, stereo block matching, and optical flow. Feature extraction is used to find interesting pixels in an image. Once features are extracted, they are no longer stored as a continuous block of adjacent pixels in memory. This forces other kernels to load non-continuous memory addresses, which may hinder parallelism performance. Stereo block matching uses two cameras, with known position and characteristics, to compute disparity by comparing overlapped regions, leading to a high computational load. Optical flow is used to estimate the apparent motion of objects between two consecutive images. Optical flow can be computed for each pixel (dense) or a subset of pixels (sparse).

4.3.5 Neural networks

Convolutional Neural Network (CNN) is a special class of multi-layer neural networks, designed to recognize and analyze visual patterns directly from pixel images. They are usually comprised of a sequence of convolutional layers, activation functions, pooling layers, fully connected layers and normalization layers. In this study, we focused on five CNNs: (1) Inception-v2: a 22 layers network that introduces a special 1x1 convolution, and using global average pooling instead of using fully connected layers [84]. (2) ResNet-50, short for Residual Network. It introduces the idea of (identity shortcut connection) that skips one or more layers to address the vanishing gradient problem [39]. It is a deep residual network of 50 layers. (3) ResNet-18: one of the residual network variants. It consists of 18 layers. (4) MobileNet: is a small model built upon the idea of using depth-wise separable convolutions as efficient building blocks [80]. (5) SqueezeNet: is a family of models that achieve AlexNet-level accuracy on ImageNet with 50x fewer parameters [44].
4.4 Experimental methodology

This section describes the performance metrics, hardware and software environments used in our benchmarking setup. It also describes measurement techniques, and introduces our benchmarking approach.

4.4.1 Performance metrics

In this work, we evaluate the efficiency of vision kernels and neural networks using four performance metrics: (1) accuracy, (2) run-time, (3) energy per frame, and (4) energy delay product (EDP). These metrics provide a fair way of comparison between different design points and a meaningful interpretation to make design choices. In this subsection, we discuss the performance metrics, as follows:

4.4.1.1 Accuracy

To evaluate the accuracy of vision kernels, we compute the absolute difference between the results generated by the GPU and FPGA implementations and compare them to the CPU implementation. For neural networks, classification accuracy (test error rate) is used to evaluate the CPU, GPU and FPGA implementations by comparing their results to the ground truth in the whole Imagenet validation set with 50k images. The top-1 and top-5 classification accuracy are reported in this work for Inception-v2, ResNet-50, ResNet-18, Mobilenet-v2 and SqueezeNe neural networks.

4.4.1.2 Run-time

There are two different types of run-time performance measurements: (1) Compute performance which measures only the compute part of vision kernel or neural network excluding potential bottlenecks for moving data from/to the external memory. Even though it doesn’t capture the application level performance, it reflects the efficiency in preforming various compute patterns, (2) System performance which measures the performance of the complete pipeline, including initialization, image
reading, pre-processing, post-processing and data transfer. It measures the un-optimized performance of the platform by capturing data movement bottlenecks. Figure 4.1 shows steps involved in each of the compute and systems performance measurement.

### 4.4.1.3 Energy

Energy consumption per frame quantifies the amount of electrical energy dissipated by hardware accelerators to perform a kernel’s operations on one frame. It is measured as the power consumed during the delay time to process a frame. Device power can be divided in two parts: (1) Static power: represents the amount of power consumed when no active computation is taking place (system is idle), (2) Dynamic power: represents the amount of power consumed above the static power level when the system is computing.

### 4.4.1.4 Energy-Delay Product (EDP)

Run-time or energy per frame alone do not show the entire picture. A hardware platform can be extremely low power while being too slow to be of practical use. The Energy Delay Product (EDP) [41] metric takes into account the throughput of the algorithm measured in (ms/frame) along with the energy consumed per frame (mJ/frame). EDP is the product of energy/frame and delay time. This way, a fair comparison can be made when deciding which hardware architecture is better suited for specific computation. Lower EDP is better which means that the hardware architecture can finish specific computation tasks using less power in less time.

![Figure 4.1: Steps involved in compute and system performance measurements](image-url)
4.4.2 Measurement techniques and platforms

In this study, we evaluated two popular platforms for deploying embedded vision applications: Nvidia Jetson TX2 and Xilinx ZCU102. These platforms come equipped with an on-board power measuring IC that can measure multiple power rails such as: CPU cores and GPU cores on the Jetson, and programmable logic, full power CPU cores and low power CPU cores on the FPGA platform. On the Jetson TX2, shell scripts (running on its ARM CPU) sample power rails and log their values along with the system’s timestamp into text files. The act of measuring power consumes power, thus consequently affects the results. The presented data in this work has been corrected for this. On the ZCU102, a Python script is used to sample power rails that are accessible through the INA226 and are mapped to PS readable virtual files in sysfs.

For every benchmark, we first processed 1000 frames on the CPU core of the platform and then 1000 frames on the hardware accelerated part of the platform. This can be seen in Figure 4.2, where the first two vertical lines mark the first 1000 frames on the CPU and the following two lines mark the last 1000 frames on the hardware. We computed the average frame rate by measuring the time between vertical lines and divided it by 1000. The x-axis represents the number of power samples taken for each platform. Note that the ZCU102 has a different sampling rate than the TX2. For vision kernels and pipelines, input frames were in gray-scale with 1080p resolution. For neural networks, frames were in RGB with \(224 \times 224\) resolution. We also used 1024 frames instead of 1000 frames to have multiple of batch sizes.

4.4.2.1 Hardware environments.

In this work, we used Xilinx Zynq UltraScale+ MPSoC ZCU102 FPGA board. It has a 16nm XCZU9EG FPGA, and an on-board 4GB 64bit DDR4 RAM with a peak bandwidth of 136Gb/s. For GPU board, we used the Nvidia Jetson TX2 (Pascal 256 CUDA cores (16nm)) has 8GBs of 128bit DDR4 RAM with a peak bandwidth of 477.6 Gb/s. Both the FPGA and GPU have on-chip ARM CPU cores with NEON SIMD optimization.
Figure 4.2: Measuring power on the platform’s CPU cores, and its FPGA or GPU

The TX2 GPU board supports three operating modes with different clock frequencies and power consumptions, as follow: (1) Max-Q: (maximum energy efficiency) in this mode all components on the TX2 are configured to achieve the best power-throughput tradeoff. (GPU @ 0.85GHz). (2) Max-P: this mode increases the GPU’s clock frequencies to increase the performance sacrificing the power (GPU @ 1.12GHz), and (2) Max-N: (maximum clock) is the maximum performance mode allowing the TX2 to hit higher performance at the cost of some energy efficiency (GPU @ 1.30GHz).

On the ZCU102 FPGA board, the xfOpenCV FPGA kernels are clocked at 300 MHz, and the Xilinx DPU overlay at configuration mode (3xB4096) runs at 333 MHz. The ARM-A57 is clocked at 1.7 GHz. Table 4.1 shows the theoretical peak performance of the platform used in this work.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Configuration</th>
<th>Data types</th>
<th>Perf [TOPS]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM Cortex A57</td>
<td>-</td>
<td>FP32, FP64</td>
<td>0.41</td>
</tr>
<tr>
<td>Xilinx ZCU102</td>
<td>-</td>
<td>INT8</td>
<td>6.71</td>
</tr>
<tr>
<td>NVIDIA Jetson TX2</td>
<td>MaxN</td>
<td>FP16, FP32</td>
<td>1.33, 0.67</td>
</tr>
<tr>
<td>NVIDIA Jetson TX2</td>
<td>MaxP</td>
<td>FP16, FP32</td>
<td>1.15, 0.57</td>
</tr>
<tr>
<td>NVIDIA Jetson TX2</td>
<td>MaxQ</td>
<td>FP16, FP32</td>
<td>0.87, 0.44</td>
</tr>
</tbody>
</table>

4.4.2.2 Software environments.

We used three publicly available vision libraries: (1) OpenCV 3.4, (2) Nvidia’s VisionWorks 1.6, and (3) Xilinx’s xfOpenCV 2018.3. While the OpenCV code base already comes with some
GPU accelerated code, it does not come with FPGA support. For this purpose, we used OpenCV compatible C++ wrappers for xfOpenCV kernels [3]. With this wrapped functionality we were able to compile the same OpenCV code for both GPU and FPGA. Both OpenCV and VisionWorks support full IEEE FP precision, while xfOpenCV supports 8 bit precision.

In neural network benchmarking, NVIDIA TensorRT and Xilinx Vitis AI are used, since both are hardware-specific frameworks that are optimized for neural network inference on embedded GPUs and FPGAs, respectively. We used OpenCV 3.4 DNN module to evaluate the performance of ARM57 CPU. On NVIDIA side we used Jetpack 3.3 on TX2 and Jetpack 4.1.1 with corresponding TensorRT versions 1. For Xilinx platforms, the Xilinx Vitis AI framework version 1.1 is used.

### 4.4.3 Benchmarking approach

In this study, we intentionally focused on evaluating the performance of out-of-the-box kernels from publicly available libraries (without writing special platform specific code around kernel calls) to give a fair comparison in terms of development efforts. For this reason, we first ran single kernel calls from OpenCV and VisionWorks libraries on the CPU and GPU, respectively, and instantiated a single kernel from xfOpenCV in FPGA fabric (even though small kernels utilize few FPGA resources). We then measured the efficiency of representative vision pipelines on the three HW accelerators to quantify their speed and energy efficiency on these more complete vision applications.

For single kernel evaluation, we compared the efficiency of the HW accelerators in terms of their energy consumption per frame. We measured a vision kernel’s dynamic power while excluding the static power required to power the rest of the platform. This better reflects the actual workload that is being deployed to the system since certainly for small kernels, the compute energy [33] (energy consumed for computation only) and data transfer energy are usually dominated by the static power. In the vision pipeline evaluation, we compared the performance of HW accelerators in terms of their energy delay products (EDP). We used the total power consumption (static + dynamic), because it represents the actual power consumption when a complete system is deployed.
Equation (4.1) shows an FPGA’s frame rate when it is clocked at 300MHz for 1080p images. The theoretical frame rate on the FPGA is fixed for vision kernels that perform a single pass over the input image. We also measured the maximum frame rate achieved on the three HW accelerators. The theoretical frame rate on the FPGA is fixed for vision kernels that perform a single pass over the input image. The theoretical frame rate on the FPGA is fixed for vision kernels that perform a single pass over the input image.

Equation (4.1) shows an FPGA’s frame rate when it is clocked at 300MHz for 1080p images.

\[
FPS = \frac{300 \text{MHz}}{1080 \times 1920 \times 1\text{cycle/pixel}} = 144
\]  

(4.1)

In our experiments, we measure run-time as follow:

- **Vision Kernels**: We measure the compute performance and the time to transfer data from/to these kernels.

- **Complete Vision Pipelines**: We measure the compute performance and the time to transfer data from/to the input and output of these pipelines. Communication between the pipeline’s kernels is local on the FPGA (through FIFOs) and depends on the caching on the GPU.

- **Neural Networks**: Due to library limitations, we could not measure the isolated compute only performance in neural networks. The weights and activation maps need to be streamed from/to off-chip memory between layers. Therefore, only the system performance of neural networks is measured and reported in this work.
In our experiments, we measure the power as follows:

- **Vision Kernels**: dynamic power only using the following power rails: (VCCINT) in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in CPU.

- **Complete Vision Pipelines**: total power (static + dynamic) using the following power rails: (VCCINT) power rail in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in the CPU.

- **Neural Networks**: total power (static + dynamic) using the following power rails: (VCCINT) power rail in the FPGA, (VDD_SYS_GPU) in the GPU, and (VDD_SYS_CPU) in the CPU.

Table 4.2: Data movers energy consumption measurements

<table>
<thead>
<tr>
<th>Platform</th>
<th>Time/frame (ms)</th>
<th>Energy/frame (mJ/f)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPGA</td>
<td>6.945</td>
<td>0.41</td>
</tr>
<tr>
<td>GPU</td>
<td>1.298</td>
<td>0.19</td>
</tr>
</tbody>
</table>

In order to have a sense of the amount of energy consumed for computation only, we measured the energy consumption of data movers in the FPGA and GPU. We implemented passthrough kernels which copy pixels from one memory location to another without applying any arithmetic/logical operations. In the FPGA implementation, Xilinx’s SDx tool instantiates data movers [14] for each input or output port to transfer data between the memory mapped domain and the stream domain. Table 4.2 shows that FPGA takes 6.945 ms to copy an entire image (1080p) with 0.41 mJ/frame, while GPU takes 1.298 ms with 0.19 mJ/frame. These values can be used to give a sense of the ratio of energy consumed for computation to data transfer in each kernel.
4.5 Experimental results

This section first presents the benchmarking results of single vision kernels and a set of representative vision pipelines are evaluated. Then, it shows the results of a set of neural networks.

4.5.1 Single kernel performance

Before evaluating the run-time performance and energy/frame consumption of single kernels on the HW accelerators, we first compare two available GPU implementations: OpenCV CUDA module and Nvidia’s VisionWorks toolkit. The OpenCV GPU module is written using CUDA and as a result benefits from the CUDA ecosystem. The Visionworks library applies many optimization techniques to boost performance, such as buffer reuse, kernel fusion, efficient use of streaming and CUDA textures, automatic scheduling across processing units, tiling and pipelining vision functions at the sub-frame level. Figure 4.3 shows the frame rate (bottom) and energy per frame (top) achieved by running vision kernels on the Jetson TX2. The dark color represents OpenCV CUDA module, and the light color represents VisionWorks. We can observe that the VisionWorks implementation outperforms the OpenCV module in frame rate over all kernels. It achieved up to a $9.7 \times$ speedup compared to the OpenCV module. It also consumes less energy per frame over all kernels. It achieved up to a $6.3 \times$ reduction in energy consumption per frame. For this reason, in the rest of the chapter, we will use only the VisionWorks implementation for the GPU.

Next, we measured the energy per frame consumption of vision kernels from the following six categories: (1) input processing, (2) arithmetic operations, (3) filter operations, (4) image analysis, (5) geometric transformation, and (6) composite kernels.

*Input processing:* The energy/frame of input processing kernels is shown in Figure 4.4. These kernels mapped well to the GPU and FPGA compared to the CPU because of their significant data parallelism, low complexity, and no data dependency. The GPU and FPGA achieved an average reduction ratio of $1.79 \times$ and $1.41 \times$ in energy/frame compared to the CPU. It also shows that GPU’s implementation of bit-depth conversion achieved a $2.4 \times$ reduction compared to FPGA, because of the efficient use of streaming and CUDA textures in the VisionWorks kernel’s implementation.
Image Arithmetic: The performance of arithmetic/logic operations is shown in Figure 4.5. It shows that simple operations such as: threshold, absDiff, add/sub, and bitwise and/or/xor can be efficiently implemented by the CPU. However, the CPU starts to perform poorly in kernels with multiplication operations, such as: multiply, accumulate squared, weighted, magnitude and phase. The GPU has the lowest energy/frame compared to the CPU and FPGA. The GPU’s implementations achieved an average reduction ratio in energy/frame of 4.6× and 7.2× compared to CPU and FPGA, respectively. An expected result, as these algorithms can be granulated into many pieces that execute the same operation (SIMT).

Image Filters: In Figure 4.6, the results of filtering operations show that the FPGA performs better than the GPU and CPU for these kernels. The FPGA’s implementation achieved an average reduction ratio of 1.8× and 7.4× in energy/frame compared to the GPU and CPU, respectively. The memory access patterns and mathematical complexity of linear filters (filter2D, box filter, pyramid up and pyramid down) maps well to the parallel processing of the GPU and FPGA. Median filters, however, are unlike linear filters. They do not use sequential data access and multiply-and-accumulate operations, but sort input elements and select the median of them, which makes them less straightforward to implement efficiently on a GPU. The morphological operations (dilate and erode) use hit and miss functions over a structuring element. These functions are more difficult to implement than filtering functions due to comparison and branching. This explains
the low frame rate (as shown in Figure 2) and high energy/frame consumption of VisionWorks’s implementations of small (3×3) filter kernels.

**Image Analysis:** The results of the image analysis kernels are shown in Figure 4.7. For kernels such as lookup table, histogram, and histogram equalization, the energy/frame consumption of the FPGA achieves an average reduction of 1.2× compared to the GPU. While for kernels with more branching conditions and complex memory access patterns, such as integral image, mean/std, and min/max locations, the FPGA’s implementation achieved an average reduction ratio of 3.5× compared to the GPU.

**Geometric Transformation:** The results of the geometric transformation kernels are shown in Figure 4.8. The CPU performs poorly for these kind of operations compared to the GPU and FPGA. Also, the FPGA was more energy efficient compared to the GPU. It achieved a reduction of 1.6× in energy/frame for the resize and remap kernels, and 2× for affine warp and perspective warp kernels. The computations in the warp operations are more complex compared to resize and remap as mapping addresses need to be generated from 2×3 or 3×3 matrices before starting the mapping operation. The mapping process in these kernels is done from destination to source in order to avoid sampling artifacts and visiting every pixel in the destination image multiple times.

**Composite Kernels:** The last category in our study includes kernels for: (1) detecting image features (canny, fast and harris), (2) computing optical flow, and (3) computing disparity using stereo block matching. Figure 4.9 shows that the FPGA implementation of feature extraction
kernels (canny, fast and harris) were more energy-efficient compared to the CPU and GPU by an average reduction of $7.7 \times$ and $3.5 \times$, respectively. The steps to calculate sparse optical flow using the pyramid Lucas-Kanade algorithm includes extracting feature points from one frame and tracking them in the next frame. The FPGA implementation was able to detect 488 Harris corners compared to 94 for VisionWorks for the same input frame and parameters. Also, it was able to keep track of these points in the next frame. This explains the high energy/frame consumption in the FPGA implementation. Moreover, the VisionWorks’s implementations of StereoBM is not open sourced yet, so the number reported in this work is for the GPU implementation using OpenCV’s CUDA module instead.

The average energy/frame reduction for the GPU and FPGA is shown in Table 4.3. The ratio is with respect to CPU consumption (higher is better). We can observe a trend from simple kernels (top) to more complex kernels (bottom). The trend demonstrates that the performance of the GPU and FPGA compared to the CPU improves as kernels’ complexity increases. For simple kernels (input processing and image arithmetic), the GPU shows the highest performance/energy efficiency, while for more complicated kernels (image filters, image analysis and geometric transform), the FPGA shows the highest performance/energy efficiency. Moreover, as the complexity of kernels increase, the FPGA shows higher energy-efficiency compared to the GPU and CPU. This occurs due to the fact that more complex algorithms naturally occupy more resources on the programmable
logic, as well as the fact that GPUs do not scale well for problems that are not easily divisible (data locality) or have many conditions or complex memory access patterns.

| Table 4.3: Ratios of energy/frame reduction (reference CPU) |
|-----------------|---------|---------|
|                  | CPU     | GPU     | FPGA   |
| Input Processing | 1       | 1.79×   | 1.41×  |
| Image Arithmetic | 1       | 3.19×   | 2.93×  |
| Image Filters    | 1       | 3.17×   | 3.89×  |
| Image Analysis   | 1       | 2.34×   | 5.67×  |
| Geometric Transform | 1 | 10.3×   | 16.6×  |
| Features/OF/  | 1       | 7.44×   | 22.3×  |

For completeness, we did a frame rate comparison between the ARM57 CPU OpenCV and GPU VisionWorks implementations. Our result shows that VisionWorks implementations outperform OpenCV implementation by an average speedup of 2.9×, 4.2×, 6.3×, 3.9×, 42× and 4.5× for the six categories of vision kernels. The FPGA’s frame rate met the theoretical rate of Equation (4.1) for kernels performing a single pass over the input image (144 fps @300MHz for 1080p). To validate kernels’ accuracy, we used OpenCV’s output image as our reference and computed pixel-wise subtraction with the VisionWorks and xfOpenCV outputs to measure differences. We had no differences for all reported vision kernels.

4.5.2 Complete vision pipeline performance

In this section, we evaluated the performance of the HW accelerators for four representative pipelines. Common steps in many computer vision pipelines include: pre-processing, feature extraction, and post-processing. The pipelines used in our study follow this structure: (1) background subtraction, (2) color segmentation, (3) stereo block matching, and (4) Harris corner tracking. These pipelines are implemented on the GPU using VisionWorks OpenVX graph mode to enable its advanced optimization techniques (buffer reuse, kernel fusion, etc.). We also pipelined the execution of kernels on the FPGA at pixel/frame level using xfOpenCV modules. In this way, the FPGA can leverage the fact that image pixels stays within the programmable fabric and avoids
going back and forth to read/write from external memory. In terms of CV pipelines accuracy, the results in OpenCV matches the GPU and FPGA. The pipelines evaluated in this work are:

4.5.2.1 Background subtraction

The background subtraction pipeline is used to detect changes in image sequences [2]. It is mainly used when regions of interest are foreground objects. The pipeline components include: subtraction, Gaussian filtering, threshold, erode and dilate, as shown in Figure 4.10.

![Figure 4.10: Background subtraction pipeline components](image)

4.5.2.2 Color segmentation

This pipeline is used to partition an image into multiple segments based on a specific range of colors. It converts the color format from RGB to HSV, then applies range thresholding to its three channels, and applies erode and dilate operations, as shown in Figure 4.11.

![Figure 4.11: Color segmentation pipeline components](image)

4.5.2.3 Harris corners tracking

This pipeline is used to detect and track feature points in a set of successive frames of a video. It takes in the current and next frame as inputs. It computes Harris corners from the current frame and outputs a list of tracked corners in the next frame. The pipeline uses five kernels: Gaussian pyramid, Harris corner detection, Optical flow and update corners kernels, as shown in Figure 4.12.
4.5.2.4 Stereo block matching

This pipeline is used to generate a disparity map given the camera parameters and inputs from a stereo camera setup. It is used as a first step in creating a three dimensional map of an environment. The main components involved in the pipeline are shown in Figure 4.13. It consists of stereo rectification, remapping, and disparity estimation using a local block matching method.

Figure 4.14 plots the Energy/frame and EDP comparison of the four pipelines, and shows the FPGA implementations consume less energy/frame compared to the CPU and GPU for all pipelines. The FPGA is also more efficient in terms of EDP (lower EDP is better). The FPGA’s Energy/frame and EDP reduction ratio with respect to the GPU is listed in Table 4.4. As the complexity of the pipeline grows, the energy/frame and EDP reduction ratio increases. More complex vision pipelines can use more of the FPGA programmable logic, reducing the relative impact of static power consumption. Additionally, data communicated between modules of the pipeline are kept on-chip in the streaming FPGA implementation.
Table 4.4: FPGA’s reduction ratios with respect to GPU

<table>
<thead>
<tr>
<th>Pipeline</th>
<th>Energy/frame (mJ/f)</th>
<th>EDP (mJ.s/f²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Background Subtraction</td>
<td>1.74×</td>
<td>1.32×</td>
</tr>
<tr>
<td>Color Segmentation</td>
<td>1.86×</td>
<td>1.41×</td>
</tr>
<tr>
<td>Harris Corners Tracking</td>
<td>3.94×</td>
<td>2.65×</td>
</tr>
<tr>
<td>Stereo Block Matching</td>
<td>8.83×</td>
<td>107.7×</td>
</tr>
</tbody>
</table>

Figure 4.14: FPGA outperforms GPU and CPU in energy/frame consumption and EDP

4.5.3 Neural network inference performance

In this section, we measure the accuracy and performance of five different neural networks: Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet. We benchmark their implementation using OpenCV DNN, TensorRT, and Vitis AI frameworks on an ARM-57 CPU, Jetson TX2 GPU, and ZCU102 FPGA, respectively. We also evaluate the performance of hardware optimizations: reduced precision implementations (GPU and FPGA), multiple batch sizes and different operating modes (GPU) and different threads counts (FPGA). These implementations are evaluated using the following performance metrics: test accuracy and system performance (frame rate, energy/frame and EDP).
Table 4.5: Top-5 (Top-1) classification accuracy

<table>
<thead>
<tr>
<th>Library</th>
<th>InceptionV2</th>
<th>ResNet-50</th>
<th>ResNet-18</th>
<th>MobileNetV2</th>
<th>SqueezeNet</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCV (FP32)</td>
<td>91.1 (72.75)</td>
<td>91.85 (74.44)</td>
<td>88.48 (68.32)</td>
<td>86.1 (64.75)</td>
<td>78.13 (54.38)</td>
</tr>
<tr>
<td>TensorRT (FP16)</td>
<td>90.8 (72.01)</td>
<td>91.15 (72.86)</td>
<td>89.30 (69.93)</td>
<td>86.4 (65.40)</td>
<td>76.30 (52.29)</td>
</tr>
<tr>
<td>Xilinx DPU (INT8)</td>
<td>90.30 (71.68)</td>
<td>91.31 (73.34)</td>
<td>88.25 (66.94)</td>
<td>85.06 (63.54)</td>
<td>76.58 (50.26)</td>
</tr>
</tbody>
</table>

4.5.3.1 Accuracy

The Top-1 and Top-5 classification accuracy achieved by Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet implemented using OpenCV DNN, TensorRT and DPU is shown in Table 4.5. The accuracy is measured on the ImageNet-1K validation set (ILSVRC-2012). These networks have different computational complexity [GFOPs] and parameters size [MBs]. We listed these networks in the Table 4.5 based on their GOPs/MBs ratio: Inception-v2 (6 GOPs/ 91MB), Resnet-50 (4 GOPs/ 98MB), Resnet18 (2 GOPs/ 45MB), MobileNet-v2 (0.3 GOPs/ 14MB) and SqueezeNet (0.36 GOPs/ 5MB). The results show that reducing the bit precision from FP32 in OpenCV DNN to FP16 in TensorRT and INT8 in DPU keeps the Top-5 (Top-1) accuracy loss within $\sim 2\%$($\sim 4\%$). This suggests that these models can be used with precisions as low as INT8 which will reduce model complexity by 4x while maintaining an acceptable accuracy loss.

4.5.3.2 System performance

System performance measures the efficiency of the complete inference pipeline including its pre-processing, computation, data movement, and post-processing stages which gives insight into the actual performance achieved after deployment. It captures potential memory bandwidth bottlenecks in data copying between off-chip and on-chip memories.

In our experiment, we measure the performance of CPU, GPU and FPGA implementations of Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet networks. We measure their performance at multiple batch sizes ($b=1, b=2, \ldots, b=128$) and thread counts ($t=1, t=2, \ldots, t=8$) to evaluate the effect of increasing batch size on data reuse and data movements reduction. Figure 4.15 and Figure 4.16 show the experimental results of Inception-v2 and ResNet-50. The blue, green and
The results show that even with the limited memory bandwidth in the FPGA, it was able to achieve higher frame rates compared to CPU and GPU. For Inception-v2, the FPGA (t=6) achieves a speed up of 2.5× and 65× compared to the GPU FP16 (b=128) and CPU. For ResNet50, the FPGA (t=8) also achieves a speed up of 2.1×, and 77× compared to the GPU FP16 (b=128) and CPU, respectively. This speed-up comes from the low numerical precision (INT8) used in FPGA compared to the (FP32 and FP16) in CPU and GPU, as well as multiple optimizations supported by Xilinx Vitis AI framework such as: (1) memory allocation, scheduling and reusing, (2) node fusion/decomposition and (3) data stream optimization. Moreover, it is noticed that the improvement in frame rate starts to saturate as batch size and thread count increases due to reaching the maximum chip capacity. Another observations is that the FPGA implementations...
of Inception-v2 and ResNet50 are $4.72\times$ and $3.8\times$ more energy efficient when number of threads equals (t=8) compared to (t=1).

In terms of EDP, the FPGA implementations have lower EDP values compared to the CPU and GPU FP16 implementations. Figure 4.15 shows that FPGA implementation (t=8) of Inception-v2 has an EDP reduction ratios of $1.5\times$ compared to the GPU FP16 (b=128). Figure 4.16 shows that FPGA implementation (t=8) of ResNet50 has $1.1\times$ EDP reduction ratios compared to the GPU FP16 (b=128).

Figure 4.17, Figure 4.19 and Figure 4.18 show the experimental results for the small networks: ResNet-18, Mobilenetv2 and SqueezeNet. For ResNet-18, the FPGA (t=6) achieves a speed up of $2.6\times$ and $96\times$ compared to the GPU FP16 (b=128) and CPU. For Mobilenetv2, the FPGA (t=7) also achieves a speed up of $2.9\times$, and $55\times$ compared to the GPU FP16 (b=128) and CPU. For SqueezeNet, the FPGA (t=6) achieves a speed up of $2.5\times$, and $65.6\times$ compared to the GPU FP16 (b=128) and CPU.
Figure 4.19: System performance for mobileNet network

Table 4.6 summarizes the FPGA’s frame rate and EDP reduction ratio compared to the GPU FP16 implementations. The FPGA is $2.1\text{--}2.9\times$ faster and $1.1\text{--}2.4\times$ more energy efficient than GPU F16 implementations when running Inception-v2, ResNet-50, ResNet-18, Mobilenetv2 and SqueezeNet.

<table>
<thead>
<tr>
<th>Model</th>
<th>Frame Rate (fps)</th>
<th>EDP (mJ.s/f2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inception-v2</td>
<td>2.5×</td>
<td>1.5×</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>2.1×</td>
<td>1.1×</td>
</tr>
<tr>
<td>ResNet-18</td>
<td>2.6×</td>
<td>1.4×</td>
</tr>
<tr>
<td>Mobilenet-v2</td>
<td>2.9×</td>
<td>2.4×</td>
</tr>
<tr>
<td>SqueezeNet</td>
<td>2.5×</td>
<td>1.7×</td>
</tr>
</tbody>
</table>

The development of cost-efficient embedded vision applications is challenged in its initial design phase by the variety of hardware solutions and software libraries. This work performs an in-depth benchmark analysis of three embedded platforms, CPU, GPU- and FPGA-accelerated, evaluating the efficiency of their different hardware architectures towards vision kernels, complete vision...
pipelines and neural networks [Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet]. To support reproducibility, the benchmark only relies on publically available libraries and frameworks. Given the energy-efficiency focus, three key metrics are collected in the benchmarks: energy per frame, frame rate and energy delay product (EDP).

The experimental results show that many simple and easy-to-parallelize vision kernels perform well on GPUs (1.1–3.2× energy/frame reduction), but for more complete vision pipelines, FPGAs outperform GPUs and CPUs (1.2–22.3× energy/frame reduction). Moreover, FPGAs perform increasingly better as the complexity of vision pipelines grow. This is evident by the energy-delay product, a metric that not only takes into account the energy/frame, but also the throughput. The FPGA is 2.1–2.9× faster and 1.1–2.4× more energy efficient than GPU F16 implementations when running Inception-v2, ResNet-50, ResNet-18, MobileNetv2 and SqueezeNet.

Our future work will update this analysis to the latest platform generations, like Nvidia’s recently released AGX board, and will include more vision kernels and neural networks. Additionally, we will extend this benchmarking analysis to include popular neural processing units (NPUs) in mobile processors such as iPhone A13 Bionic, Samsung Exynos, Qualcomm Snapdragon, etc.
CHAPTER 5. AN EFFICIENT STRUCTURED SPARSE PRUNING OF
CONVOLUTION LAYERS USING LFSR

5.1 Introduction

Deep neural networks (DNNs) have achieved remarkable success in many challenging tasks
including image classification, object detection, and image segmentation [53] [60, 56]. Although
DNNs provide state-of-the-art accuracy, they require considerable storage, memory bandwidth and
computational resources, which limit their deployment to embedded environments. A major chal-
lenge in deploying DNNs on resources limited platforms is the large amounts of energy consumed
when accessing model parameters from external DRAMs [37]. For example, in 45nm CMOS tech-
nology, accessing a 32-bit DRAM memory requires 640pJ, which is 3 order of magnitudes higher
than a 32bit floating point add operation (0.9 pJ). Therefore, deploying DNNs with large memory
bandwidth requirements on battery constrained embedded platforms remains a challenging task.

One promising approach is model compression by pruning redundant and less important weights.
Recent research shows that significant redundancy exists in DNNs parameters that can be pruned
without sacrificing accuracy [37, 36]. However, despite the significant reduction in models’ param-
eters (up to 90%), these methods have hardly sped up inference time. The irregular distribution of
weights in pruned models poorly fit current general computing platforms such as CPUs and GPUs.
The speedup can be negative compared to dense convolution when the sparsity ratio is low [94].
Another problem with sparse convolution is the overhead of managing sparse representations. The
amount of data used to record non-zero weights locations could be high with low sparsity ratios. In
summary, mapping irregular sparse convolution to hardware is challenging and could require twice
the memory of the dense model.

The irregularity of sparse weights could be reduced by applying constraints on the locations
of weights during the pruning process. In unstructured pruning methods, only the magnitude of
weights are used to decide whether to prune or retain a weight value. All weights below a specific threshold will be pruned from the network (converting a dense model into a sparse model). The remaining weights are re-trained to recover accuracy loss. Since in unstructured pruning no constraints are placed on the non-zero weights pruned, the generated sparse weights have an irregular distribution within the weight tensor. In structured pruning methods, constraints on locations of pruned weights are applied (e.g. channel-wise, filter-wise, shape-wise, etc.). For example, in channel-wise pruning methods, all weights in a channel will be pruned or retained. A general advantage of structured pruning is the retention of hardware friendly regularity that can be leveraged to simplify sparse convolution operations. However, strict pruning constraints can negatively impact accuracy.

In this work, we propose a hardware-friendly pruning algorithm that generates regular geometrically structured sparse weights. The locations of non-zero weights follow pseudo random patterns generated by Linear Feedback Shift Registers (LFSRs). We also propose an FPGA-based inference engine for sparse convolution, which uses pruned models generated by our pruning approach to speed up convolution computation. We can avoid copying sparse weight indices from off-chip memory by computing these indices on-chip in real-time.

The contribution of this work can be summarized as follows:

- We propose a structured pruning method that uses pseudo random sequences generated by LFSRs with known seeds, to regularize and prune DNN models.
- We explore several pruning configurations and evaluate their accuracy across multiple sparsity levels on ImageNet.
- We design an FPGA-based accelerator by leveraging our proposed pruning method to create an architecture that efficiently performs sparse convolution operations.
The remaining of this chapter is organized as follows. Section 5.2 provides background on CNNs, pruning, sparse representations, and LFSRs. Section 5.3 reviews related work. In Section 5.4, the proposed structured pruning algorithm is presented. Section 5.5 explains the implementation details of our hardware architecture. Finally in Section 5.6, we discuss the experimental setup and results.

5.2 Background and motivations

In this section, we give an overview of convolution operations, weight pruning algorithms and common sparse representations. We then review the concept of linear feedback shift registers (LFSRs) and how they can be used to generate pseudo random sequences. Finally, we present the main challenges of sparse convolution computation.

5.2.1 Convolution operations in CNNs

In CNNs, convolution layers are used to extract patterns (features) from images. These patterns can be edges, corners, shapes, or more complex features. Cascading multiple convolution layers in deep models helps extract more complex information, which can be used to solve classification, detection, and segmentation problems. A single convolution operation uses small filters to extract features. The operation takes two inputs: an image tensor (I) and a list of filters (W). It outputs a map of extracted features also called feature map (O).

![Convolution layer with two filters](image)

Figure 5.1: Convolution layer with two filters
Algorithm 1: Convolution Operation

1 for $h = 0; h < H; h += 1$ do
2     for $w = 0; w < W; w += 1$ do
3         for $n = 0; n < N; n += 1$ do
4             for $c = 0; c < C; c += 1$ do
5                 for $r = 0; r < R; r += 1$ do
6                     for $s = 0; s < S; s += 1$ do
7                         $O(n,h,w) += W(n,m,r,s) \times I(m,h+r,w+s)$
8                     end
9                 end
10             end
11         end
12     end
13 end

A convolution layer can be represented by 6-nested loops as shown in Algorithm (1), where $H$ is the height of output feature map, $W$ is the width of output feature map, $C$ is the number of input channels that should equal number filters channels, $N$ is the number of output channels and number of filters. $R, S$ are the filter dimensions. Figure 5.1 shows a convolution layer with $H=4$, $W=4$, $C=3$, $R=3$, $S=3$ and $N=2$.

5.2.2 DNN weight pruning

Weight pruning is the process of removing unimportant and redundant weights from DNN models without scarifying accuracy. It is an efficient way to compress dense models by reducing the number of parameters and operations. During the pruning process, the importance of weights is defined based on a given metric, and less important weights are pruned first. The most common metric is the absolute value of weights, first presented by [37]. If a weight’s absolute value is less than a certain threshold, it is zeroed-out. As an example, when using magnitude-based pruning on the VGG-16 model, about 50% of the weights could be pruned without fine-tuning. With fine-tuning, over 80% of the weights were pruned without accuracy loss. Other criteria focus on the energy consumption of a CNN layer to guide the pruning process [93]. The geometrical location of
weights is also used to reduce the irregularity of sparse weights \[48\] \[54\] \[98\], such pruning schemes are referred to as structured pruning algorithms.

5.2.3 Sparse matrix representation

One solution to work with sparse weights efficiently is to use an alternative data structure to represent sparse data. Zero weights can be ignored, and only non-zero weights are stored. There are multiple representations that can be used to efficiently store sparse weights. For these representations, a tradeoff exists between the amount of extra data used to encode sparse weights, and the complexity of restoring data. Some methods store more data in order to simplify the process of restoring data, while others have less encoding data overhead at the expense of more computation. The three most common sparse representations are described below:

5.2.3.1 Coordinate list (COO)

In this representation, non-zero weights are stored as a list of tuples with each tuple containing: filter’s row and column indices, channel number, filter’s number and non-zero value, as shown in Figure 5.2. Table 5.1 shows encoding overhead and the minimum sparsity (SP) required for the COO representation to be smaller than the dense representation. For example when \(R=3\), \(S=3\), \(C=512\), \(N=128\) and \(SP=0.6\), the encoding data in MBs (assume 1 Byte for each index) = \(3 \times (3 \times 3 \times 512 \times 128) \times (1-0.6) = 0.67\)MB. The minimum sparsity required for the COO representation to be smaller than dense is 0.667.

![Figure 5.2: Coordinate list (COO)](image)
5.2.3.2 Compressed sparse row (CSR)

The compressed sparse row (CSR), or Yale format represents a matrix by three arrays containing nonzero values, column indices, and the extents of rows. It is similar to COO, but compresses the row indices. Figure 5.3 shows an example of compressing a sparse weight matrix into a CSR format. For a convolution layer with \( R=3, S=3, C=512, N=128 \) and SP=0.6, the encoding data in MBs= \((3 \times 3 \times 512 \times 128) \times (1-0.6) + 128 \times 512 = 0.28\text{MB}\). The minimum sparsity required for the CSR representation to be smaller than the dense is 0.55.

![Figure 5.3: Compressed sparse row (CSR)](image)

5.2.3.3 Compressed sparse column (CSC)

The compressed sparse Column (CSC) representation is similar to CSR except that a row index for each non-zero weight, and column pointers are stored. Figure 5.4 shows an example of compressing sparse weight matrix into CSC format. For a convolution layer with \( R=3, S=3, C=512, N=128 \) and SP=0.6, the encoding data in MBs= \((3 \times 3 \times 512 \times 128) \times (1-0.6) + 3 \times 3 = 0.22\text{MB}\). The minimum sparsity required for the CSC representation to be smaller than the dense is 0.50.

![Figure 5.4: Compressed sparse column (CSC)](image)
Table 5.1: Sparse representation overhead

<table>
<thead>
<tr>
<th>representation</th>
<th>Encoding Overhead</th>
<th>Required Sparsity</th>
</tr>
</thead>
<tbody>
<tr>
<td>COO</td>
<td>$3 \times (R \times S \times C \times N) \times (1-SP)$</td>
<td>$SP \geq 0.667$</td>
</tr>
<tr>
<td>CSR</td>
<td>$(R \times S \times C \times N) \times (1-SP) + N \times C$</td>
<td>$SP \geq 0.5(1/(R \times S) + 1)$</td>
</tr>
<tr>
<td>CSC</td>
<td>$(R \times S \times C \times N) \times (1-SP) + R \times C$</td>
<td>$SP \geq 0.5(1/(C \times N) + 1)$</td>
</tr>
</tbody>
</table>

5.2.4 Linear feedback shift register (LFSR)

In digital computing, a LFSR is a shift register whose input bit is a linear function of two or more its previous states [63]. It is commonly used for generating pseudo random sequences especially in networking and cryptograph domains. The advantages of using an LFSR are: (1) It can be easily implemented in hardware, (2) Its pseudo random sequence has statistical properties that preserve good randomness properties, and (3) It is deterministic, where the same input seed always give the same output sequence. Figure 5.5 shows an example of a 5 bits LFSR, and its sequence.

An n-bit LFSR consists of n cells, each of which holds a state variable $s_i \in \{0, 1\}$ and a coefficient (tap) $c_i \in \{0, 1\}$ , for $i = 0, 1, ..., n-1$. The feedback function (XOR function) computes the new state $s_n$ using the coefficients and the state values as shown in Equation 5.1. The period of a LSFR is a function of its coefficients and initial state values. The maximal sequence period generated by an n-bit LFSR is $2^n - 1$ unique states, since the all-zero state is excluded. The maximal period is generated when the coefficients form a primitive polynomial (irreducible). Table 5.2 lists primitive polynomials for LFSRs of order n = (2-9).

$$s_n = c_0 \cdot s_0 \oplus c_1 \cdot s_1 \ldots \oplus c_{n-1} \cdot s_{n-1}$$ (5.1)

Table 5.2: Primitive polynomials

<table>
<thead>
<tr>
<th>n=2</th>
<th>n=3</th>
<th>n=4</th>
<th>n=5</th>
<th>n=6</th>
<th>n=7</th>
<th>n=8</th>
<th>n=9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taps</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,2)</td>
<td>(0,1)</td>
<td>(0,1)</td>
<td>(0,2,3,4)</td>
</tr>
</tbody>
</table>
5.3 Related work

Recent works have explored the efficiency of implementing sparse convolution on hardware. One research area focuses on finding the best approach to store sparse weights after pruning. The goal is to compress nonzero weights into small data structures while keeping operations on them efficient. In [58], the COO format is used, where nonzero weights in the same input channel are compressed into a vector of 5-tuple elements. Although this method is efficient for sparse coordinate computation, it introduces a large overhead of data to represent sparse weights. When convolution computation is performed as a matrix-vector multiplication, CSR and CSC formats can also be used to compress sparse matrices. The CSC format is used in [35] [19] [43], as it provides lower memory bandwidth compared to the CSR format when the number of filters (N) is not significantly larger than the number of weights in the filter [83]. Although these methods can compress sparse weights into a compact format, they introduce a large amount of data overhead for indexing sparse weight matrices. In our work, we avoid the overhead of sparse weight indices in convolution layers by generating them on-chip using LFSRs. In [49], two LFSRs are used represent the row and column indices of sparse weights of fully connected layers.

Designing custom hardware architectures has been explored to efficiently support sparse DNNs on ASICs and FPGAs. In [35], a CNN accelerator (EIE) is proposed, which exploits the sparsity of both input feature maps and filters. However, it only focuses on sparse fully connected layers. Parashar et al. proposed an SCNN accelerator that supports processing convolutional layers in a compressed format. It uses pixel-oriented dataflow where the innermost computation is a Cartesian product [70]. However, this method requires significant coordinates computation to locate the sparse weights. Zhang et al. [100] present the Cambricon-X accelerator which applies step indexing
techniques and uses wide (256×16-bits width) memory and (256-to-1) multiplexers (MUXs) in convolution layers to gathering sparse weights into a vector, which needs to dynamically select the input vector. FPGA-based accelerators have also been proposed to accelerate sparse convolution. In [58], a tile look-up table and a channel multiplexer is used to match the index between sparse weights and input pixels. The work in [103] uses a vector generator module to match the index between sparse weights and input activations and uses shape-wise pruning to allow for sharing the same indexes of weights between processing units. A more detailed comparison between our work and FPGA-based accelerators is presented in Section (5.6).

5.4 Proposed pruning method

The proposed pruning method consists of four steps. First, we choose a suitable LFSR pattern configuration for our model. Second, we find the best seed for each LFSR register in that configuration. Best seeds are chosen based on the initial distribution of weights in the pre-trained models, where they generate sequences that keep important weights (high value) and prune unimportant weights (low values). Third, we train our models with regularization to minimize the importance of potentially pruned weights to further minimize accuracy loss. Finally, we prune and re-train our model. In this method, we also add constraint on the locations of non-zero weights within filters. We keep the number of non-zero weights in each filter fixed across channels. This helps simplify the hardware design for inference engine. The following sub-sections explain each step in more details:

5.4.1 LFSR patterns configuration

In this step, we explore four different pattern configurations for pruning weights in convolution layers. Some of these configurations add more strict constraints on the locations of non-zero weights within filters, while others add more loose constraints at the expense of complicating the pruning processes and inference engine. Figure 5.6 shows these configurations for a convolution layer of 3 filters of size (R=3, S=3, C=5). The configurations are described as follows:
• **perLayer**: one pattern is used to prune weights in all filters. For example, pattern (1-3-4) is used in all filters to represent non-zero weights indices, as shown in Figure 5.6. This configuration has the strictest constraints.

• **perFilter**: one pattern is used to prune weights for each filter. Sequences (1-3-4), (0-1-3) and (1-2-4) are used in filter (n=0), (n=1) and (n=2), respectively to represent non-zero weights indices. A total of N (number of filters) sequences is required for this configuration.

• **perCoordinates**: A total of $R \times S$ (filter’s width×height) patterns are required for this configuration. The same patterns is shared between different filters at the same coordinate (r, s).

• **perCoordFilter**: A total of $R \times S \times N$ patterns are used in this configuration. Each filter has his own version of patterns. The only constraint in this configuration is that the number of non-zero weights in each filter location is fixed across channels.

![Per Layer Per Filter Per Coordinate Per CoordFilter](image)

Figure 5.6: Four different pattern configurations for pruning weights

### 5.4.2 Finding best seeds

In LFSRs, different initial states (seeds) generate different pseudo random sequences. In this work, we use a ranking algorithm to find the best initial seeds for each LFSR register. We use the
magnitude of weights in pre-trained models. We evaluate if the sequence generated by a seed keeps most of the important weights and prune unimportant weights at multiple sparsity levels. In this way, we can guarantee using the best starting sequence for our model.

The proposed ranking algorithm explores all possible sequences and return the best LFSR seeds. Figure 5.7 shows an example for a sequence of length 10 and three filters in perCoordinate configuration (i.e. the same pattern is shared across filters). Based on the sparsity level \( S_p \), we use the first \( C \times S_p \) indices from the pattern as shown in the white cells in Figure 5.7, where \( C \) is number of channels. We also add a significance for each weights value. Weights that will be pruned first have lower significance compared to the weights pruned at higher sparsity level. For example: \( W_8 \) has the lowest significance value of 0.1 and \( W_7 \) has the highest significance value of 1. The input to this step is filter weights and the desired sparsity level \( S_p \). The output is a list of best seeds \( best(seed_i) \). For each possible seed value \( seed_i \), we generate its sequence of length \( C \) (\# of channels) and use it to compute seed rank \( rank(seed_i) \), as shown in Equation (5.2), where \( idx \) is channel index.

\[
rank(seed_i) = W(:, :, idx, :) \times (1 - idx/C)
\] (5.2)
5.4.3 Training with regularization

In the third step, we change the distribution of weights to follow the selected LFSR sequence generated by the best seeds. We use customized L1 regularization during the training process to force weights in specific locations to have low values or zero-out without degrading the accuracy. We added a regularization term to the cost function \( J \), as shown in Equation (5.3), where \( n \) is number of training samples. \( \lambda \) is the regularization parameter; note, large \( \lambda \) adds more penalty on weight values and make them closer to zero. \( W_l \) is the weight tensor of layer \( l \). \( S_l \) is the significance tensor of layer \( l \) with the same shape as \( W_l \). Each value in \( S_l \) falls between \([0-1]\) based on the importance of each corresponding weight. Finally, the regularization term is computed by a dot product of the weight tensor and the significance tensor. This way, we force the optimization algorithm to re-distribute weights within each tensor to follow our LFSR patterns.

\[
J = \frac{1}{n} \sum_{i=1}^{n} L(y_i - \hat{y}_i) + \lambda \sum_{i=1}^{L} \| W_i \cdot S_i \| \tag{5.3}
\]

5.4.4 Pruning and re-training

In the last step, we use iterative pruning and re-training process to prune weights in dense models follow LFSR patterns. We start with dense model (current sparsity \( S_c = 0 \)) and then increase current sparsity level \( S_c \) from \([0 - \text{desired sparsity level(} S_d\)]\) gradually over \( N \) steps. Every weight below the \( S_c \) threshold is zeroed out and the rest remain the same. Between each two pruning steps, we re-train our model to compensate the accuracy loss. After \( N \) steps, our model will have \( S_d \times R \times S \times C \times N \times L \) non-zero weights, where \( L \) is number of layers.

5.5 Hardware accelerator

In this section, we describe the hardware optimizations and implementation details of our sparse CNN accelerator and its sparse dataflow. We also provide a quantitative analysis of the computing throughput and required memory bandwidth by our accelerator.
5.5.1 Hardware architecture and dataflow

Figure 5.8 shows an overview of the CNN accelerator design on FPGA. It consists of four major components: processing elements (PEs), on-chip buffers, external memory, and on-chip interconnect [99]. Before computing, all data including input image and model’s weights are stored in an external memory. Due to limitations in the size of on-chip memory, we cannot copy all data from the external memory to on-chip buffers. First, we have to divide input data into small portions (Tiles) and cache them into on-chip buffers before feeding them to processing elements PEs. On-chip buffers are used to store tiles of input image, model’s weights and output (partial sums). Data communication channels between PEs and on-chip buffer banks are provided through on-chip interconnect. Finally, PEs are responsible for all computations.

To achieve high performance, we start our optimization from Algorithm 1. We transform the convolution computation from window-based to element-matrix multiplication. Using this approach, we can process each weight individually, which is more efficient when computing sparse convolution. Then, we apply loop tiling to keep a small portion of data stored on-chip, and increase data reuse and reduce external memory access. External memory access happens only when we finish all computations on the current tile and a new tile is needed. Loop tiling and ordering decide the dataflow from/to our accelerator.

![Figure 5.8: Overview of FPGA-based CNN accelerator design](image-url)
In our dataflow (shown in Algorithm 2), lines 1-4 show the order in which we process tiles. In lines 5 and 6, we copy a tile of input pixels and weights from external memory into on-chip buffers. The size of input tile is \([T'_h, T'_w, T_c]\), where \(T'_h = T_h + R - 1, T'_w = T_w + C - 1\). The size of weight tile is \([T_r, T_s, T_c, T_n]\). We also clear an output tile of size \([T_h, T_w, T_n]\) in line 7. In this section, we explain the dataflow for perCoordFiler configuration. As discussed in Section 5.4.B, in perCoordFiler configuration, we generate a unique random sequence at each filter’s coordinates \((r, s, n_i)\). To achieve that, we load a unique LFSR seed in lines 11 and generate a sequence of indices of length \(T_c\) as shown in lines 13. Sparse weight at index \(c_i\) matches input pixel at index \(c_x\). In order to increase parallelism in our architecture, the nested loops in lines 10, 14 and 15 are unrolled and mapped to a parallel hardware. We choose tile and unrolling sizes such that we fully utilize of all computation resources provided by the FPGA hardware platform.

Figure 5.9 shows our hardware architecture when tile size equals \([T'_h = 4, T'_w = 4, T_n = 1, T_c = 1]\). In this architecture, we use \(4 \times 4\) BRAMs to stores input pixels. Each BRAM stores pixels at the same coordinates over all channels i.e. \(I(h, w, :)\). This way, we can read multiple pixels at the same time. To match sparse weights with their corresponding input pixels, we connect \(T_n\) LFSRs to the address port of BRAMs (broadcast). LFSRs are used to generate a sequence of sparse weight indices. Thus, every time a new index is generated by LFSR, a new data will be read from all BRAMs at the same channel. Then, the output ports of BRAMs are multiplied with the same weight value to perform element-matrix multiplication as shown in Figure 5.10. The partial sums from the multiplication operation are accumulated in \([T_h = 3, T_w = 3]\) registers. Multiplxers of size \((R \times S)\) are used to select the correct input to each accumulator. After \((R \times S \times C \times SP)\) clock cycles, the results in the accumulator registers become valid and can be streamed out to external memory, where \(SP\) refers to sparsity ratio.

The attainable system throughput is constrained by either computation (computation-bounded) or communication (memory-bounded). In [90], a roofline performance model is proposed to relate system performance to the peak performance provided by the hardware platform and off-chip memory traffic. The actual performance of an algorithm on a hardware platform is the minimum of
two terms, as shown in Equation (5.4). The first term is the peak throughput (GOP/s) provided by all computation resources in the platform assuming all data is available on-chip. The second term is the maximum performance that the memory system can provide. It depends on the algorithm's computation to communication (CTC) ratio and platform bandwidth. In the next two sections, we provide a quantitative analysis of the computation throughput and required memory bandwidth by our accelerator.

\[
\text{Attainable Perf.} = \min \begin{cases} 
\text{peak performance (P)} \\
\text{CTC} \times \text{bandwidth (BW)} 
\end{cases} 
\tag{5.4}
\]

5.5.2 Computation optimization

The peak computational performance (also called as computational roof) is the maximum number of operations per second provided by hardware when all required data is available on-chip.
Given a specific tile size \([T_h, T_w, T_n, T_c]\), the peak computational performance can be computed by Equations (5.5 and 5.6). It is a function of loop unrolling factors in the \(T_h, T_w, T_n, T_c\) dimensions. Loop unrolling increases peak performance but also increases the resources utilization in FPGA devices. In our architecture, we need \((C \times SP \times R \times S)\) clock cycles to finish the execution of each tile.

\[
P(GOP/s) = \frac{\text{total number of operations}}{\text{number of execution cycles}} \times f \tag{5.5}
\]

\[
P = \frac{2 \times H \times W \times N \times (C \times R \times S \times SP)}{\left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C \times SP}{T_c} \right\rceil \times (C \times SP \times R \times S)} \times f \tag{5.6}
\]

### 5.5.3 Memory access optimization

In this subsection, we show how reducing communication volume can increase attainable performance. We reduce memory traffic volume by generating sparse weight indices on-chip using LFSRs and applying efficient data reuse. Thus, we were able to increase computation to communication (CTC) ratio. CTC ratio describes the total number of computations per memory access. Equations (5.7 and 5.8) show CTC ratio calculation for our accelerator, where \(\alpha_{in}, \alpha_{weight}, \alpha_{out}\) and \(B_{in}, B_{weight}, B_{out}\) refer to memory access counts and on-chip buffer sizes for input, weights and output feature maps, respectively. \(SP\) is sparsity ratio and \(SP_{En}\) is buffer size used for storing sparse weight representation overhead (in our implementation \(SP_{En} = 0\)).

\[
CTC = \frac{\text{total number of operations}}{\text{total amount of external data access}} \tag{5.7}
\]
\[ CTC = \frac{2 \times H \times W \times N \times (C \times R \times S \times SP)}{\alpha_{in} \times B_{in} + \alpha_{weight} \times B_{weight} + \alpha_{out} \times B_{out}} \] (5.8)

where:

\[ B_{in} = T_c \times (T_h + R - 1) \times (T_w + S - 1) \] (5.9)

\[ B_{weight} = (R \times S \times C \times N) \times SP + SP_{En} \] (5.10)

\[ B_{out} = T_h \times T_w \times T_n \] (5.11)

\[ 0 < B_{in} + B_{weight} + B_{out} \leq BRAM \text{ capacity} \] (5.12)

\[ \alpha_{in} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C}{T_c} \right\rceil \] (5.13)

\[ \alpha_{weight} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C \times SP}{T_c} \right\rceil \] (5.14)

\[ \alpha_{out} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \] (5.15)

### 5.6 Experiment

In this section, we first evaluate the performance of our pruning algorithm. We measure the accuracy of its four different configurations at multiple sparsity levels. We compare our algorithm’s accuracy with dense and unstructured pruning models. Then, we evaluate the performance of our sparse convolution engine on three modern DNNs and compare its performance with previous dense and sparse accelerators.
5.6.1 Evaluation of pruning algorithm

To evaluate the performance of our pruning algorithm, we measure the classification accuracy of three modern DNNs models: VGG16, ResNet50 and InceptionV3 on ImageNet dataset (ILSVRC2012) of 1000 classes, 1.5M training images and 50k testing images [26]. A summary of the characteristics and top1 (top5) accuracies of the used models is shown in Table 5.3. Number of operations is computed at 224×224, 224×224 and 229×229 image resolution for VGG16, ResNet50 and InceptionV3, respectively. In our implementation, we extend the TensorFlow framework [9] by adding masks for weight tensors. We update these masks based on the selected algorithm’s configuration and generated patterns using LFSRs. During the pruning process, we start with weights from pre-trained models [50]. Then, we apply our pruning algorithm gradually in which the sparsity is increased from an initial sparsity ratio $s_i$ (usually 0) to a final sparsity ratio $s_f$ over a span of $n$ pruning steps. We re-train our models between each two pruning steps for 10 epochs. We carried out the experiments on TensorFlow 1.14 on Nvidia GeForce GTX TITAN X.

The Top1 (Top-5) classification accuracy of VGG16, ResNet50, and InceptionV3 models at $[0, 0.1, ..., 0.9]$ sparsity levels is shown in Figure 5.11. We use dense models and unstructured pruning models as our references. Figure 5.11 also shows the accuracy results of our algorithm at four different configurations: perLayer, perFilter, perCoordinate and perCoordFilter. It can be noticed that perLayer and perFilter configuration have the lowest accuracies, whereas the accuracy drops fast as we increase sparsity level. This is due to strict constraints added to these configurations and the complexity of the ImageNet classification problem. perLayer and perFilter could be useful when we deal with a simpler problem. While perCoordinate configuration has an acceptable accu-

<table>
<thead>
<tr>
<th>Model</th>
<th>Top1</th>
<th>Top5</th>
<th>#OP (GOP)</th>
<th>Size (MBs)</th>
<th># Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16</td>
<td>0.713</td>
<td>0.901</td>
<td>30.92</td>
<td>528 MB</td>
<td>138,357,544</td>
</tr>
<tr>
<td>ResNet50</td>
<td>0.749</td>
<td>0.921</td>
<td>6.65</td>
<td>98 MB</td>
<td>25,636,712</td>
</tr>
<tr>
<td>InceptionV3</td>
<td>0.779</td>
<td>0.937</td>
<td>11.25</td>
<td>92 MB</td>
<td>23,851,784</td>
</tr>
</tbody>
</table>
racy, perCoordFilter configuration achieves the best results. perCoordFilter was able to compress VGG16, ResNet50 and InceptionV3 models by 80%, 76% and 65% with accuracy (Top1) loss of less than 2%. It was also able to achieve around 55%, 52% and 46% sparsity ratio with accuracy loss of less than 1%.

![Graphs comparing accuracy vs. sparsity for VGG16, ResNet50, and InceptionV3 models.](image)

Figure 5.11: An accuracy vs. sparsity comparison

### 5.6.2 Evaluation of hardware architecture

#### 5.6.2.1 Experiments setup

We evaluate our design on the Xilinx ZCU102 platform. It consists of an UltraScale FPGA, quad ARM CortexA53 processors, 4GB PS DDR4 and 512MB PL DDR4 (14.9GB/s). In our experiments, we use Xilinx Vivado HLS (v2019.1) tool chain to transform an optimized C code into
RTL implementation. Our design is synthesized at 200MHz frequency on this platform. In this work, we evaluate the performance of our accelerator on: VGG-16, Resnet50 and InceptionV3 at 80%, 76%, 65% sparsity levels.

5.6.3 Resource utilization

Figure 5.12 shows the resource utilization of our accelerator at different configurations (parallelism factors) $[T_h, T_w, T_n]$. The utilization of BRAMs is determined by input tile size ($T_h, T_w$) and number filters in the weight tile $T_n$. We use BRAMs mainly to store the input and weight tiles on-chip. The parameters $T_h, T_w$ determine the size of input buffers and $T_n$ determines the number of weight buffers. LUTs and FFs utilization increases as the parallelism factors $T_h, T_w, T_n$ increase because the number and size of MUXs in our design increase, as shown in Section 5.5.B. We use FF registers to implement $T_h \times T_w \times T_n$ accumulators, because we need to access all registers at the same clock cycle. We also observe that the LUTs and FFs utilization is almost linear to the tile size. The number of DSPs used in our architecture can be calculated as $T_h \times T_w \times T_n$. Each DSP can perform a 16bit $\times$ 16bit multiplication operation. We can tell that our accelerator with $[T_h = 8, T_w = 8, T_n = 24]$ has almost fully utilized the FPGA’s hardware resources and achieved the highest peak computational performance. Table 5.4 reports the available resources in Xilinx Zynq ZCU102 platform.

![Resource Utilization Graph](image_url)
Table 5.4: Available resource in ZCU102 platform

<table>
<thead>
<tr>
<th></th>
<th>BRAMs</th>
<th>DSPs</th>
<th>FFs</th>
<th>LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZCU102</td>
<td>1,824</td>
<td>2,520</td>
<td>548,160</td>
<td>274,780</td>
</tr>
</tbody>
</table>

5.6.3.1 Performance analysis

In this section, we evaluate the performance of our accelerator using three modern CNNs: VGG16, ResNet50 and InceptionV3 at 55%, 52% and 46% sparsity levels. We set our accelerator configuration to \( [T_h, T_w, T_n] = [8, 8, 24] \). In this configuration, we utilize most of the FPGA resources. This configuration has peak computational performance of \( 2 \times 0.2 \text{GHz} \times 24 \times 8 \times 8 = 614.4 \text{GOP/s} \) when the width of operand is 16bits. Our accelerator achieves 534.2 GOP/s effective performance on sparse VGG16 which shows 1.7\times and 1.4\times speedup compared with [58] and [103], respectively. For Resnet50, our architecture achieves 456.0 GOP/s which is 1.56\times higher than the effective performance of [58]. On GoogLeNet, we achieve 458 GOP/s performance. The work in [103] has comparable performance to our implementation on VGG16.

On VGG16, we achieved a sparsity of 54% without accuracy loss and around 80% with 2% accuracy loss. In terms of the effective GOP/s, we have a 534 GOP/sec and our accelerator was faster than the work in [58] and [103], but the work in [103] has higher image/sec throughput because they were able achieve higher MAC reduction which reduce their workload per image. At 2% accuracy loss, our architecture can reach 86 image/sec. On Resnet50, our accelerator reach 1.5/2.7\times speedup compared to the work in [58] and on InceptionV3, our accelerator reach 1.2/1.8\times speedup compared to the work in [58]. In conclusion, our accelerator was able to achieve higher effective GOP/s and we achieved speedup compared to related work. but almost on all cases our sparsity was lower than related work which increase the workload per image that need be to done.

The speedup in our architecture is because our dataflow can effectively eliminate copying extra data for sparse representation and maintain high utilization of on-chip resources. This will increase the CTC of DNN models running on our accelerator. Previous implementations cannot avoid the extra overhead of sparse representation, which results in a waste of on-chip resources. However,
Table 5.5: Performance comparison with related work \([T_h = 8, T_w = 8, T_n = 24]\)

<table>
<thead>
<tr>
<th>CNN type</th>
<th>Device</th>
<th>Frequency (MHz)</th>
<th>Precision</th>
<th>DSPs</th>
<th>BRAMs</th>
<th>LUTs</th>
<th>FFs</th>
<th>MAC Reduction(%)</th>
<th>Accuracy loss(%)</th>
<th>Performance (GOP/s)</th>
<th>Image/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1144 (45%)</td>
<td>912 (48%)</td>
<td>68K (12%)</td>
<td>67.5%</td>
<td>67.5%</td>
<td>0%</td>
<td>309.0</td>
<td>31</td>
</tr>
<tr>
<td>ResNet50</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1144 (45%)</td>
<td>912 (48%)</td>
<td>68K (12%)</td>
<td>58.7%</td>
<td>58.7%</td>
<td>0%</td>
<td>291.4</td>
<td>104</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1144 (45%)</td>
<td>912 (48%)</td>
<td>68K (12%)</td>
<td>65.8%</td>
<td>65.8%</td>
<td>0%</td>
<td>257.4</td>
<td>65</td>
</tr>
<tr>
<td>VGG16</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1350 (53%)</td>
<td>1460 (80%)</td>
<td>278K (51%)</td>
<td>65.4%</td>
<td>65.4%</td>
<td>0%</td>
<td>495.4</td>
<td>46</td>
</tr>
<tr>
<td>VGG16</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1564 (62%)</td>
<td>840 (46%)</td>
<td>433K (79%)</td>
<td>54%</td>
<td>54% / 80%</td>
<td>0%</td>
<td>534.2</td>
<td>38 / 86</td>
</tr>
<tr>
<td>ResNet50</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1564 (62%)</td>
<td>840 (46%)</td>
<td>433K (79%)</td>
<td>52%</td>
<td>52% / 76%</td>
<td>0%</td>
<td>456.0</td>
<td>65 / 114</td>
</tr>
<tr>
<td>InceptionV3</td>
<td>ZCU102</td>
<td>200</td>
<td>16 bit fixed</td>
<td>1564 (62%)</td>
<td>840 (46%)</td>
<td>433K (79%)</td>
<td>46%</td>
<td>46% / 65%</td>
<td>0%</td>
<td>458.0</td>
<td>75 / 114</td>
</tr>
</tbody>
</table>

the inefficiency of our accelerator comes from two aspects. First, the output feature map size in the CNN layers cannot be divided into \(T_h, T_w, T_n\) evenly, which leads to waste of computation. Second, in the element-matrix multiplication dataflow, we are utilizing \((T_h - R/2 + 1) \times (T_w - S/2 + 1)\) DSPs out of the total \(T_h \times T_w\) DPSs, which also implies waste of computation.

In this chapter, we proposed a hardware-aware pruning algorithm that generates geometrically structured sparse weights. The locations of non-zero weights are derived in real-time using Linear Feedback Shift Registers (LFSRs). The advantage of using this approach is two-folds: First, eliminated the overhead of managing sparse representations; second, avoid copying extra data from external memory to on-chip buffers. We also proposed a hardware inference engine for sparse convolution on FPGAs. It uses LFSRs to compute the positions of non-zero weights within weight tensors on-chip. Experimental results demonstrate that our accelerator can achieve 456-534 GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 1.5-1.8× speedup over previous sparse CNN FPGA accelerators.
Algorithm 2: Pseudo Code of the Dataflow

```plaintext
for h = 0; h < H; h+ = T_h do
    for w = 0; w < W; w+ = T_w do
        for n = 0; n < N; n+ = T_n do
            for c = 0; c < NNZ; c+ = T_c do
                /* Load Input Tile */
                get(InputTile[T_h][T_w][T_c])
                /* Load Weight Tile */
                get(WeightTile[R][S][T_c][T_n])
                /* Clear Output Tile */
                A[T_h][T_w][T_n] = 0;
                for r = 0; r < R; r + + do
                    for s = 0; s < S; s + + do
                        for n_i = 0; n_i < T_n; n_i + + do
                            seed = readLFSRSeed();
                            for c_i = 0; c_i < T_c; c_i + + do
                                c_x = generateNext(seed);
                                for h_i = 0; h_i < T_h; h_i + + do
                                    for w_i = 0; w_i < T_w; w_i + + do
                                        A[h_i][w_i][n_i] += W[r][s][c_i][n_i] ×
                                        TData[h_i + r][w_i + s][c_x];
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
end
```

/* Store Output Tile */
store(A[T_h][T_w][T_n]);
```
CHAPTER 6. CONCLUSION

In this research work, we focus on software and hardware co-optimization techniques for efficient processing of computer vision algorithms on reconfigurable computing architectures. The proposed techniques aim to improve architecture’s efficiency in terms of run-time, energy consumption, resource utilization and programmability. We proposed techniques to improve the performance of vision algorithms on reconfigurable architectures. We focused on the commonly used components in these algorithms, namely: sliding window, histogram computing, and convolution.

First, we presented a new sliding window architecture that efficiently utilizes the available Block RAMs on chip. We proposed a lite-weight compression algorithm that take advantage that natural non-random images have most of their information in the approximation sub band and small details in the other sub-bands which allow for representing the coefficients in the details sub-bands using less bits. The proposed image compression algorithm gives good compression ratio and can be used in our architecture to reduce BRAMs at the expense of introducing more LUTs resources. The architecture can be configured to perform sliding window operations using lossless or lossy compression based on an application’s requirement. Evaluating the proposed architecture on a set of images selected from a benchmark dataset shows promising results. The memory saving reached 25-70% for lossless compression and up to 84% for lossy compression. The proposed architecture is fully pipelined, giving similar performance to the traditional architecture. The compression ratio is currently configured at design time, our future work will investigate making this automatically adjustable at runtime based on the previous frame compression ratio.

We also proposed a configurable hardware architecture for computing different histogram-based feature description algorithms. The proposed architecture is configurable in terms of window size, number of regions, number of bins per region, and the pattern of these regions. We implemented different optimization techniques to reduce hardware resources and computational complexity for
computing histograms of gradients. The experimental results show that our architecture can be used in a wide range of computer vision applications. This architecture can also be used to compute histograms for other low-level information such as local binary pattern (LBP) or grayscale intensity without any modifications. The architecture takes advantage of data streaming to reduce the computational complexity of computing this class of descriptor. To illustrate the efficiency of our architecture, we deploy two of the most commonly used descriptors (SIFT and HOG) and compare their quality with software implementations. The architecture is also evaluated in terms of execution speed and resource usage and compared with dedicated hardware architectures. Our flexible architecture shows a speed up of 3× and 5× compared to state-of-the-art dedicated hardware architectures for SIFT and HOG, with resource usage overheads [LUTs, FFs, and DSPs] of [1.1×, 15×, and 1.6×] and [6.4×, 7×, and 32×] for SIFT and HOG, respectively. Our future work will investigate the possibility of extending the proposed architecture to compute other kinds of window-based algorithms in computer vision.

In this work, we also performed an in-depth benchmark analysis of three embedded platforms, CPU, GPU- and FPGA-accelerated, evaluating the efficiency of their different hardware architectures towards vision kernels, complete vision pipelines and neural networks [Inception-v2, ResNet-50, ResNet-18, MobileNet-v2 and SqueezeNet]. Three key metrics are collected in the benchmarks: energy per frame, frame rate and energy delay product (EDP). The experimental results show that many simple and easy-to-parallelize vision kernels perform well on GPUs (1.1–3.2× energy/frame reduction), but for more complete vision pipelines, FPGAs outperform GPUs and CPUs (1.2–22.3× energy/frame reduction). Moreover, FPGAs perform increasingly better as the complexity of vision pipelines grow. This is evident by the energy-delay product, a metric that not only takes into account the energy/frame, but also the throughput. The FPGA is 2.1–2.9× faster and 1.1–2.4× more energy efficient than GPU F16 implementations when running Inception-v2, ResNet-50, ResNet-18, Mobilenetv2 and SqueezeNet. Our future work will update this analysis to the latest platform generations, like Nvidia’s recently released AGX board, and will include more vision kernels and neural networks. Additionally, we will extend this benchmarking analysis to include popular neu-
ral processing units (NPUs) in mobile processors such as iPhone A13 Bionic, Samsung Exynos, Qualcomm Snapdragon, etc.

Finally, we proposed a hardware-aware pruning algorithm that generates geometrically structured sparse weights. The locations of non-zero weights are derived in real-time using Linear Feedback Shift Registers (LFSRs). The advantage of using this approach is two-folds: first, eliminate the overhead of managing sparse representations; second, avoid copying extra data from external memory to on-chip buffers. We also proposed a hardware inference engine for sparse convolution on FPGAs. It uses LFSRs to compute the positions of non-zero weights within weight tensors on-chip. Experimental results demonstrate that our accelerator can achieve 456-534 GOP/s for the modern CNNs on Xilinx ZCU102. On VGG16, we achieved a sparsity of 54% without accuracy loss and around 80% with 2% accuracy loss. In terms of the effective GOP/sec, we have a 534GOP/sec and our accelerator was faster than the related work, but we could not achieve the highest image/sec throughput because they were able achieve higher MAC reduction which reduce their workload per image. However, at 2% accuracy loss, our architecture can reach 86 image/sec. On Resnet50, our accelerator reach 1.5/2.7× speedup, and on InceptionV3, our accelerator reach 1.2/1.8× speedup compared to the other work. In conclusion, our accelerator was able to achieve higher effective GOP/s and we achieved speedup compared to related work. but almost on all cases our sparsity was lower than related work which increase the workload per image that need be to done.
BIBLIOGRAPHY


[38] Konrad Häublein, Marc Reichenbach, and Dietmar Fey. Fast and generic hardware architecture for stereo block matching applications on embedded systems. In *ReConFigurable Computing and FPGAs (ReConFig), 2014 International Conference on*, pages 1–6. IEEE, 2014.


[58] Liqiang Lu, Jiamoing Xie, Ruirui Huang, Jiansong Zhang, Wei Lin, and Yun Liang. An efficient hardware accelerator for sparse convolutional neural networks on fpgas. In *2019 IEEE*


