Real-time Computer Vision in Software using Custom Vector Overlays

by

Joseph James Edwards

B.A.Sc, University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Applied Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

July 2018

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Real-time Computer Vision in Software using Custom Vector Overlays

submitted by Joseph James Edwards in partial fulfillment of the requirements for the degree of Master of Applied Science

in Electrical and Computer Engineering

Examining Committee:

Guy Lemieux, Electrical and Computer Engineering

Supervisor

Mieszko Lis, Electrical and Computer Engineering

Supervisory Committee Member

Sid Fels, Electrical and Computer Engineering

Supervisory Committee Member

Abstract

Real-time computer vision places stringent performance requirements on embedded systems. Often, dedicated hardware is required. This is undesirable as hardware development is time-consuming, requires extensive skill, and can be difficult to debug. This thesis presents three case studies of accelerating computer vision algorithms using a software-driven approach, where only the innermost computation is performed with dedicated hardware. As a baseline, the algorithms are initially run on a scalar host processor. Next, the software is sped up using an existing vector overlay implemented in the FPGA fabric, manually rewriting the code to use vectors. Finally, the overlay is customized to accelerate the critical inner loops by adding hardware-assisted custom vector instructions. Collectively, the custom instructions require very few lines of RTL code compared to what would be needed to implement the entire algorithm in dedicated hardware.

This keeps design complexity low and yields a significant performance boost. For example, in one system, we measured a performance advantage of 2.4× to 3.5× over previous state-of-the-art dedicated hardware systems while using far less custom hardware.

Preface

In all chapters, the soft-vector processor and the software libraries provided to program it were provided by VectorBlox Computing. Portions of chapters 3 of this thesis appear in "Real-time object detection in software with custom vector instructions and algorithm changes" by Edwards and Lemieux [12]. Both hardware customization and software design was performed by the author with supervision provided by Guy Lemieux. In chapter 4, all software design was performed by the author. Initial hardware customization was also written by the author but final designs used in this thesis were written by Aaron Severance at VectorBlox Computing. Supervision was provided by Guy Lemieux. Portions of chapters 5 of this thesis appear in "TinBiNN: Tiny Binarized Neural Network Overlay in Less Than 5,000 4-LUTs" by Edwards, Vandergriendt, Severance, Raouf, Watzka, Singh, and Lemieux [13]. Software was written by the author with hardware customization provided by Joel Vandergriendt at VectorBlox Computing. Board design was performed by the authors at Lattice Semiconductor. Supervision was provided by Guy Lemieux.

Table of Contents

List of Tables

List of Figures

Glossary

ASIC Application-specific Integrated Circuit 42

BNN Binary Neural Network 6

BRAM Block Random Access Memory 63

CNN Convolutional Neural Network 4

CVI Custom Vector Instruction 21

CVI custom vector instructions 21

DMA Direct Memory Access 19

DRAM Dynamic Random Access Memory

DSP digital signal processor 42

FPGA Field-programmable Gate Array 2

GOPS giga operations per second 65

GPU Graphics Processing Unit 5

ILP Integer Linear Programming 4

IoT internet-of-things 1

LBP Local Binary Pattern 12

LUT Lookup table 6

LVE Lightweight Vector Extensions 79

MAC multiply-accumulate 6

mAP mean average precision

MB-LBP Multi-Block Local Binary Pattern 13

MIMD Multiple Instruction, Multiple Data 31

PE processing element 42

RAM Random Access Memory 76

ReLu Rectified Linear Unit 16

ROM Read-only Memory

RTL Register Transfer Language 2

SIMD Single Instruction, Multiple Data 10

SVD Singular-Value Decomposition 49

SoC System-on-Chip 2

TOPS tera operations per second 78

YOLO You Only Look Once 4

Acknowledgments

I would like to thank Dr. Guy Lemieux for his support and guidance during my time as a graduate student.

I would also like to thank both Lattice and Xilinx for donating hardware and software licenses used in this research, and NSERC for providing funding.

Chapter 1

Introduction

1.1 Motivation

Advanced computer vision algorithms and sophisticated image processing are becoming important workloads for embedded systems. This push for embedded systems to run demanding computations is partly fueled by the growing internet-of-things (IoT) movement and need for edge processing. Advanced embedded systems are needed for security analysis, self-navigating automobiles or drones, quality control systems and a host of other applications.

Higher frame rates, higher resolutions, and more complex operations push embedded systems well into the GOPS, demanding a higher degree of parallelization. On desktop machines, where these algorithms are first developed, neither cost nor power are paramount. Real-time performance is often achieved using tuned libraries leveraging multi-core, SIMD instructions or general purpose GPU processing. On embedded systems, where these algorithms are deployed and run at scale, both cost and power matter.

Custom hardware solutions are often required to meet both performance and power constraints. In many cases, Field-programmable Gate Arrays (FPGAs) represent ideal platforms for creating a System-on-Chip (SoC), as the configurable logic can be leveraged to perform computationally demanding processing in a highly parallel, efficient manner. FPGAs are also intrinsically low power, interface directly with other devices, and typically only need external memory to complete a system. The flexibility and performance of FPGAs comes at a price, however, requiring complex Register Transfer Language (RTL) design and considerable time to implement and debug. To reduce development effort, FPGA users should aim to minimize the amount of domain-specific RTL their designs require. A trade-off between performance and design complexity may be needed.

1.2 Approach

To mitigate this trade-off, our approach allows for, but minimizes the amount of domain specific RTL needed. We propose a hybrid hardware/software approach where the majority of development effort is kept in software, keeping design complexity low (measured in lines of RTL hardware vs lines of software).

Our approach is summarized in the following steps. First, a baseline is established, running the program on a host processor on the FPGA. Second, software is rewritten to take advantage of a soft vector overlay, increasing performance over the baseline. Third, after profiling the application, targeted hardware is added to the vector overlay in the form of custom vector instructions to produce a large performance boost. This last step can be revisited or revised as current bottlenecks are removed and new bottlenecks are discovered. By iterating through bottlenecks until performance requirements are met, hardware development is kept to a mini-

mum. The majority of processing does not require acceleration beyond the initial software-based vectorization as the majority of the runtime is usually isolated to a small region of code that can be further accelerated by a small amount of custom RTL. Our approach quickly produces a custom vector overlay capable of billions of operations per second (GOPS).

Several case studies follow, showing the applicability of our approach. We develop a custom vector overlay for both traditional machine learning pipelines and state-of-the-art deep learning pipelines. In all cases studies, we produce complete end-to-end demonstration systems. We present the speedup of our accelerated version to the host processor, report additional lines of RTL required (including whitespace and comments) and compare to previous implementations when possible.

1.3 Contributions

Our first case study presents a real-time face detection system, using a variation of the classic Viola-Jones algorithm. Our result is competitive with similar systems designed completely in hardware. This method of face detection serves as a good example of our hybrid approach, both due to the resulting performance of the complete system and because parallelizing the algorithm is not straightforward.

Our custom overlay achieves a speedup of 248.4$\times$ compared to the hard ARM Cortex-A9 processor and requires less than 800 lines of custom RTL. The previous efforts implementing face detection on FPGAs have produced good results [6, 7, 16]. However, these solutions are single-use, fixed implementations that can be difficult to maintain and the developed hardware is only useful for object detection. Our approach results in faster detection rates and can be modified to include other

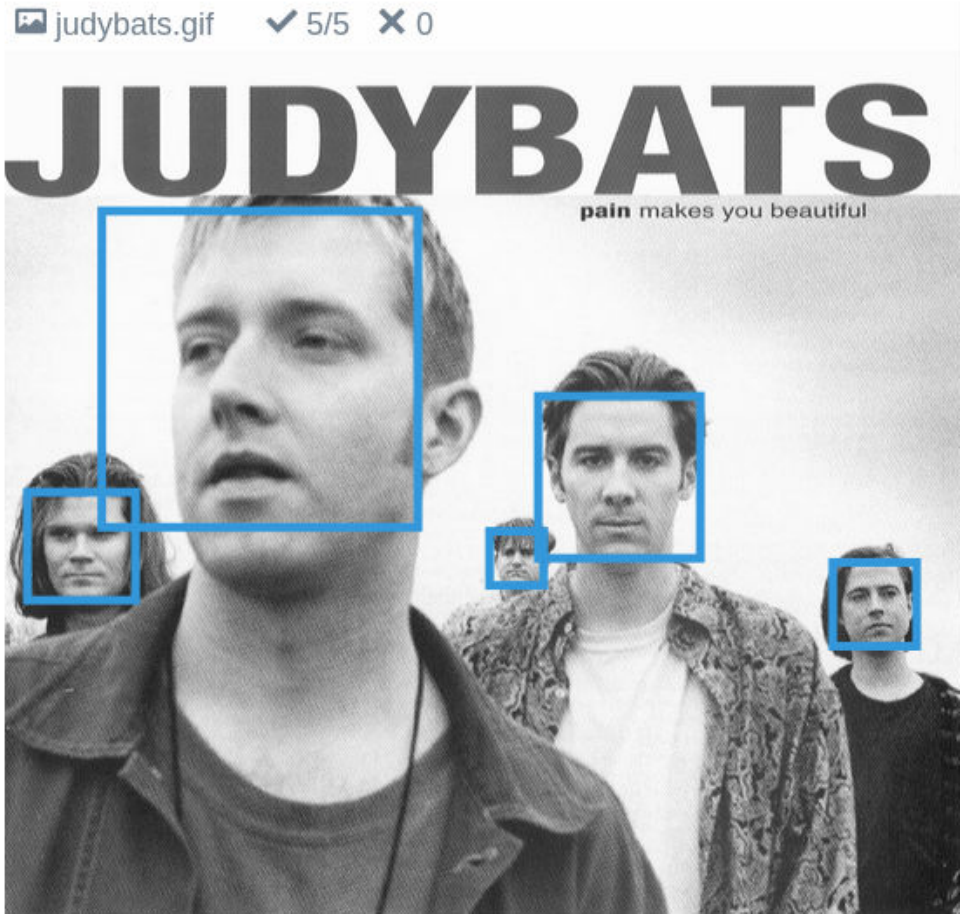


Figure 1.1: Results after running frontal face detection

types of pre-processing and post-processing.

In developing our solution, we also provide a novel contribution using Integer Linear Programming (ILP) to quantize our generated object detection cascades. Our face detection system produces the results seen in Figure 1.1.

The second case study accelerates Convolutional Neural Network (CNN) inference of multi-class object detection and localization. CNNs represent the state-of-the-art in object detection. The You Only Look Once (YOLO) project [28] uses a

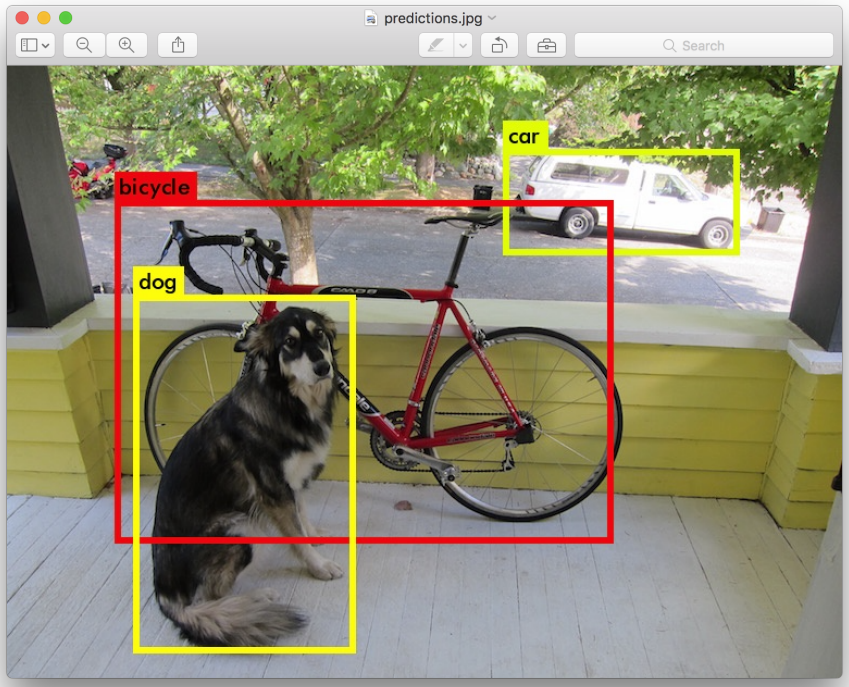


Figure 1.2: Results after running the Tiny YOLO v2 (COCO) network

new approach focusing on real-time operation. The project's reduced size network can reach up to 200 fps on a (then) top-of-the-line 16 nm Nvidia Titan-X Graphics Processing Unit (GPU) using 416×416 input images. However, the Titan-X GPU is not suitable for embedded applications due to power, form factor, the need for an x86 host, and an inability to directly communicate with sensors and I/O devices.

We produce a CNN overlay that uses a reduced data width of 8 bits. We run several example networks including 80-category and 20-category variants of the YOLO object detector. Sample outputs of Tiny YOLO are shown in Figure 1.2.

Our CNN overlay, in a single core configuration, reaches 12.4 fps. A dual-core

system, showing nearly perfect 2:1 scaling, achieves 24.4 fps. This is 1359.0× speedup over the ARM processor. The addition of the convolution accelerator only requires 1300 lines of additional RTL. Scaling is expected to continue as we increase the number of cores, and moving to larger, more modern FPGAs with more DSP blocks. This will close the gap with GPUs and previous work on FPGAs that implement dedicated CNN accelerators. The custom vector overlay approach is flexible, offers end-to-end processing, and can easily be extended to handle changes in neural network workloads.

In the third and final case study, several Binary Neural Network (BNN) classifiers derived from BinaryConnect [8] are accelerated using a lightweight overlay. Typically, the gains that deep learning provides in accuracy is offset by the number of multiply-accumulate (MAC) operations required. BinaryConnect networks, however, use 1-bit weights to eliminate multiplications and achieve state-of-the-art error rates using only addition. The 1-bit weights also reduce storage requirements and improve power efficiency.

The small BNN networks examined here are accelerated using a lightweight implementation of the vector processor to produce a minimal area solution that still benefits from the custom overlay approach. The reduced-precision arithmetic improves the size, cost, power and performance of neural networks in digital logic. This lightweight overlay uses only about 5000 4-input Lookup tables (LUTs) and fits into the lowest cost FPGA. We show it can run CIFAR-10 [20], a 10-category classification task, and that our low-precision fixed-point activations do not increase the error. A typical input image is shown in Figure 1.3.

A secondary network, trained as a single-category person classifier, is also presented. The error rate is less than 1%. Our custom vector overlay improves the

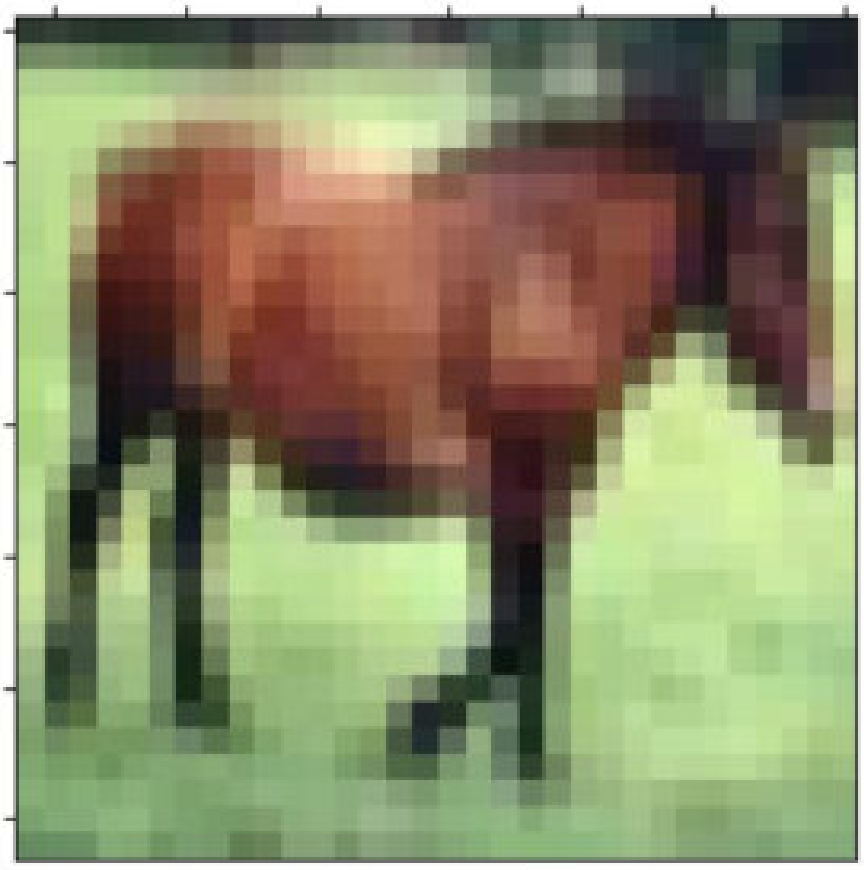


Figure 1.3: Example CIFAR-10 “horse” image

runtime 71× over the ORCA RISC-V host, and only requires 200 lines of custom RTL.

1.4 Thesis Organization

This thesis presents our approach across three separate case studies, and is organized as follows. Chapter 2 provides background material for all examples in-

cluding Viola-Jones object detection, CNNs, and the soft vector processor overlay. Chapter 3 presents the face detection example. Chapter 4 presents the CNN overlay. Chapter 5 presents the lightweight BNN overlay. Chapter 6 concludes the thesis and lists future work.

Chapter 2

Background

In this chapter we discuss essential background needed for several real-time computer vision algorithms, as well as the vector processor used to realize them. In particular, we cover the seminal work by Viola and Jones on AdaBoost object detection and recent advances in deep learning which use neural networks for multi-object classification and detection. To accelerate both classes of computer vision algorithms on an embedded system, we require a high degree of hardware parallelism. The VectorBlox MXP, a soft vector overlay, is used to provide this parallelism. The VectorBlox MXP is fully software programmable, allowing for rapid development and is extensible, allowing for simple integration of custom hardware for domain-specific acceleration. Details of the operation and key features of the soft vector processor are described in the last section.

2.1 AdaBoost Object Detection

One of the most popular computer vision algorithms is AdaBoost-based object detection, popularized by Viola and Jones [34]. This algorithm was an important

move towards accurate, real-time object detection. AdaBoost remains a popular form of object detection in embedded systems. Uses for detection can include improving auto-focus where faces are found, or producing safety warnings when pedestrians are detected in the paths of vehicles. The detection algorithm finds objects at any size (scale-invariant) and any location (spatially-invariant) in an image. This is done with high accuracy, at high resolutions and, if parallelized well, at high frame rates.

2.1.1 Algorithm Overview

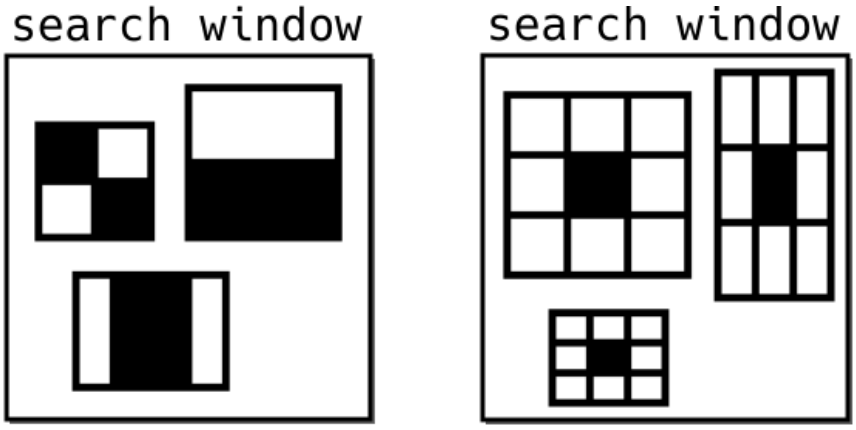
For this algorithm, several image pre-processing steps are required. This begins with an initial grey-scaling of the image, followed by the generation of an image pyramid. The images in the pyramid decrease in size until either the width or height is less than the search window dimensions. A search window is then slid across each location of each image in the pyramid. At every location, a detection cascade, consisting of a series of weak classifiers, is evaluated to determine if an object is present. Figure 2.1 shows the search window scanning the image pyramid.

Each weak classifier looks for a particular feature in the search window, using a quick computation to determine if it is present or not. Evaluating these weak classifiers is the central computation in the algorithm. During training, features are ranked by how discriminative they are and are then grouped into stages. Each stage eliminates a percentage of false positives and allows the search at a given location to terminate early as soon as enough weak classifiers in a stage fail. This early-exiting, variable execution property of the algorithm significantly speeds computation when performed sequentially, but makes feature computation across multiple locations inefficient with Single Instruction, Multiple Data (SIMD)-style execu-

Figure 2.1: A search window traverses an image pyramid, evaluating a cascade of classifiers at each position.

tion. In the worst case, all of the locations being processed in parallel must be computed.

OpenCV, a popular open source library for computer vision, provides efficient implementations for both training and detection routines for these classifiers and contains support for classifiers that look for various types of features [5].



(a) Haar features **(b)** LBP features

Figure 2.2: Examples of typical features

2.1.2 Feature Types: Haar vs LBP

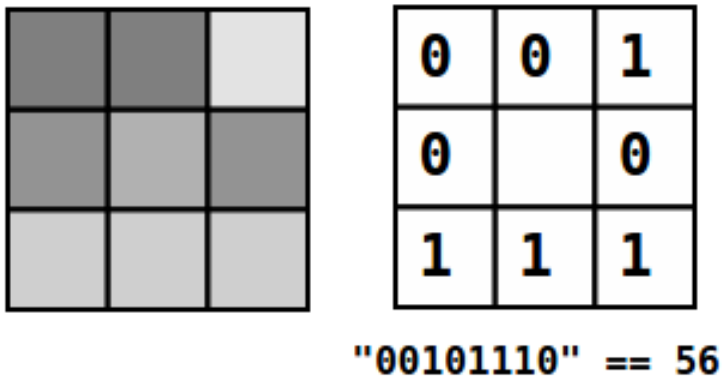
Viola and Jones used Haar features in their work. Haar features, as shown in Figure 2.2a, use the average intensity of several rectangular regions. A check is performed to see if a specified subregion (shown in white) is sufficiently brighter or darker than another subregion (shown in black). To make computing these regions efficient, integral images are often used. Integral images, or sum area tables, contain the sum of all pixels above and to left of each location. This means the sum of pixels for any given rectangular area can be obtained with only four lookups, one at each corner. The variance of the search window, calculated by summing the square of the difference between each pixel and the mean value, is also needed. Variance normalization makes the features more robust to lighting conditions. To efficiently calculate this, a squared version of the integral image is also generated.

Alternative feature types have been used successfully and are more amenable to quick calculation. Local Binary Pattern (LBP) features, shown in Figure 2.2b,

rely on generating an 8-bit pattern based on comparing a central region, or cell, to its eight neighboring regions. Each comparison yields 1 bit of this result. This 8-bit pattern then serves as an index into a feature's lookup table, returning a pass or fail value. The operations used in calculating LBP features map well to embedded processors, using integer arithmetic and avoiding division and square-root operations. Although more regions are needed to calculate each feature, the size of these regions are regular and have the potential for reuse. Also, each LBP feature tends to be more expressive than an individual Haar feature, which results in cascades with fewer features overall.

Liao et al. introduced Multi-Block Local Binary Pattern (MB-LBP) features, where each of the cell sizes used to generate the LBP pattern may be larger than one pixel [25]. Integral images may also be used to calculate these regions faster. MB-LBP features increased accuracy over single-pixel based LBP patterns. Examples of typical Haar and MB-LBP features are contrasted in Figure 2.2, shown as they would appear inside a search window. MB-LBP features are used in our implementation due to the computational advantages.

An overview of MB-LBP feature calculation is shown in Figure 2.3, highlighting the calculation of one of the weak classifiers in the cascade. Each classifier compares the average brightness of a center cell to its 8 neighbour cells, producing the 8-bit comparison bitfield. This 8-bit LBP pattern serves as an index to the features' LUT, which produces either a PASS or FAIL value. This value is summed with all other features in the stage. A stage fails if the sum is less than its required threshold. Objects are detected at a given location only if all stages pass.



(a) compute average brightness **(b)** compare to neighbors

```
if (LUT[56]) {
  VALUE = PASS
} else {
  VALUE = FAIL
}
```

```
STAGE += VALUE
```

(c) lookup pattern **(d)** add pass or fail

Figure 2.3: Overview of MB-LBP feature computation.

2.2 Convolutional Neural Networks

There has been a growing trend away from traditional machine learning that uses hand-tuned features, such as Haar and LBP, toward features that are learned. This goes under the title of representation learning or feature learning. Coinciding with this move to feature learning are the use of deeper, hierarchical models, which combine learned features of earlier layers into more complex features. This allows for more abstract, less rigid factors of variation to be learned by the models.

These large shifts to deep learning have been made possible by vastly increased

datasets, computational resources, and algorithmic advances, allowing large networks to train towards better and better solutions. Hinton's introduction of back-propagation [29] and use of deep feed-forward networks [20] represent two key contributions in this shift. LeCun's work using CNNs for MNIST digit recognition [22] proved these networks were adept for computer vision tasks.

Convolutional neural networks are now the leading approach across several computer vision challenges [20]. These include MNIST digit recognition [22], CIFAR classification tasks [17] and the much larger classification and localization challenges on ImageNet [11]. A pivotal moment occurred in 2012 when AlexNet, a deep CNN, beat previous, traditional approaches by a large margin of 10% in ImageNet's 1000-category classification task [21]. Since this point, CNNs have continued to be the dominant approach.

Though CNNs have been gaining in accuracy, most localization or detection implementations, like R-CNN, require running classification multiple times in a sliding window approach, similar to what was discussed for Viola Jones. This makes running these algorithms in real time a difficult challenge, as running the network even at a single window location requires a GPU for good performance, much less a resource-constrained embedded system. Improvements such as Faster-CNN have lowered this significantly, but recent research, in the YOLO project, have introduced a new CNN-based architecture that makes real-time inference practical.

2.2.1 Algorithm Overview

CNN networks typically consist of three layer types: convolution layers, pooling (or subsampling) layers and fully connected layers. In convolution layers, output

maps are generated by convolving each input map with its unique kernel, summing the results at each location. A bias is usually applied to the output map. This is repeated for each output map in the layer. For a given layer with maps of size $N \times M$, containing C input maps and K output maps, $C \times K$ kernels are applied over $N \times M$ pixels. This leads to a large number of multiply-accumulate (MAC) operations. The kernels are typically 3×3, but can be larger or smaller, depending on the network. Pooling layers take a given output map and reduce its size. A 2D subregion of a map, often 2×2, is reduced to a single value. Common modes of pooling include max pooling and average pooling, either taking the maximum value or average value respectively. Fully connected layers, or dense layers, take an input vector and multiply it against a matrix of weights to produce an output vector. A bias term at every position in the output vector is applied. As a matrix of size $N \times M$ is required for an input vector of N and output vector of M, and inputs and outputs can commonly be greater than 1024 in number, memory bandwidth tends to dominate in fully connected layers. Some networks replace fully connected layers with 1×1 convolution layers, decreasing this bandwidth demand tremendously.

Activation functions often follow convolution and fully connected layers. These are usually pointwise functions. Which function is used varies from network to network. They can include the sigmoid, tanh, and softmax functions. Simpler functions such as Rectified Linear Unit (ReLu), which requires only setting negative values to zero, have been used successfully in many networks and is commonly used after convolution layers. A typical network, like those implemented on the VectorBlox MXP vector overlay is shown in Figure 2.4.

Unlike the AdaBoost approach, modern CNNs require minimal preprocessing as features are learned directly from raw pixel data. CNN networks typically take

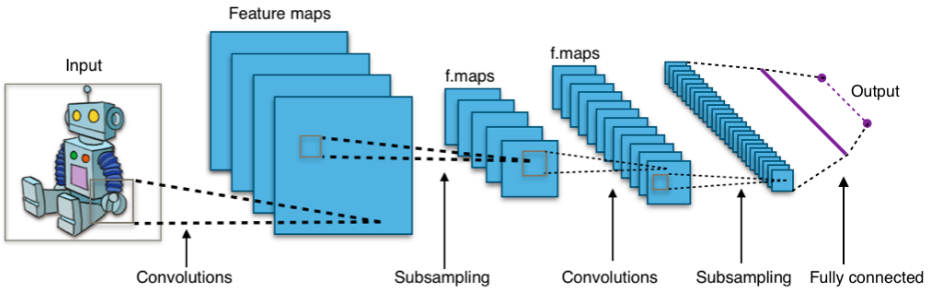


Figure 2.4: Example CNN network, showing multiple layers.

the 3 color channels (red, green and blue) as inputs, and proceed through many layers of convolution layers and pooling, ending with one or more fully connected layers (or 1×1 convolution layers). Map sizes tend to decrease in later layers due to pooling, but the number of maps tend to increase to preserve information content. Execution is not variable and operations are simpler and more regular. Instead a much larger, deterministic number of operations are required.

2.2.2 Binary Neural Networks

The large number of multiply-accumulate (MAC) operations (often greater than 90% of the operations in CNNs) and apparent robustness of large networks has led researchers to explore low-precision weights. A promising area is in the extreme case of binary weights, where weights take on one of two values, usually +/-1. Several methods have been introduced that allow networks to be trained where the weights in convolution and fully connected layers are fixed to +/-1. Examples of this include BinaryConnect [8], BinaryNet [9] and XNOR-Net [27].

Binary weights are ideal for eliminating costly multiply operations, replacing them with additions. Memory bandwidth and power are also significantly reduced. This research is promising as it allows even highly constrained embedded systems

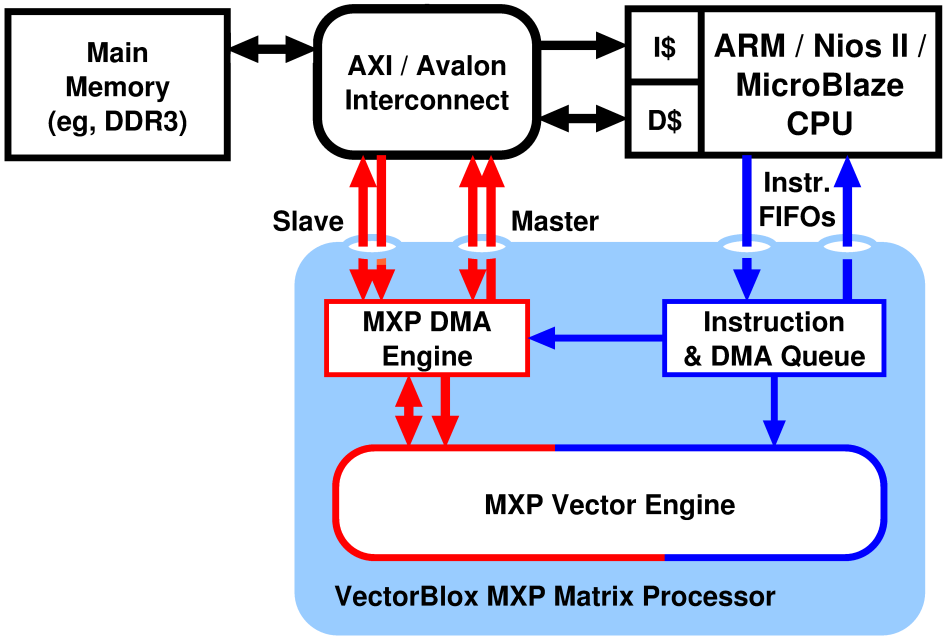


Figure 2.5: Overview of the VectorBlox MXP processor

to leverage the accuracy gains provided by deep learning. We explore accelerating these lightweight networks in our final case study.

2.3 VectorBlox MXP Architecture

For this thesis, we adopt a parameterizable and extensible soft vector processor for our overlay. The VectorBlox MXP [30] is shown in Figure 2.5. The main reason for selecting a soft vector processor is to minimize the development effort required to accelerate algorithms. Software-based iterations greatly reduce compilation time and enable rapid debugging with familiar software development tools. The processor itself has been pre-designed and pre-verified, saving significant effort for producing computationally demanding solutions.

At a high level, VectorBlox MXP consists of a host processor coupled to a

vector core and Direct Memory Access (DMA) engine. The host processor handles all control flow and I/O, and dispatches instructions to the vector core and DMA engine. All data processed by VectorBlox MXP is held in a memory-mapped, multi-bank scratchpad memory. A vector can be of any length and start at any address in the scratchpad. Data is transferred in and out of the scratchpad using the DMA engine. If there are data hazards between the vector instruction and a DMA transfer, the vector engine stalls until the hazard is resolved.

When a new vector instruction is ready to issue, the address generation logic will generate the addresses for the vector operands. The vector operands are processed one wavefront at a time, where the size of a wavefront corresponds with the number of parallel ALUs. The addresses are incremented each cycle until the end of the vector is reached.

Vector instructions are 2-input, single-output, variable-length SIMD instructions. The engine size is configured as the number of 32-bit vector lanes. Subword-SIMD is supported, so 8-bit and 16-bit data types have $4\times$ and $2\times$ more parallelism, respectively. 1D, 2D, and 3D variants of the instructions are supported for vector and matrix operations.

Given the operands must be stored in the scratchpad memory, a DMA engine moves the vectors in between the scratchpad and main memory. This is handled by the programmer. Double buffering can be used to hide data transfer overhead through pre-fetching. Long vectors help reduce potential pipeline stalls due to hazards, allowing the engine to stay fully utilized.

2.3.1 Parameterization

The vector engine is parameterized in terms of number of vector lanes and the scratchpad memory size. This allows various systems to be instantiated for the target applications. The ability to trade performance for area is important for different targets.

Multi-core systems can be instantiated, allowing the work to split between the multiple vector engines. This is useful when long vector lengths are difficult to achieve and lane scaling leads to diminishing returns as clock speed diminishes with size.

2.3.2 Programming Model

From a users perspective, programming the VectorBlox MXP requires three steps. First, the operands must be moved into the scratchpad. The requires allocating space for all operands in the scratchpad and calling the DMA engine to move operands from main memory into the scratchpad. Second, the vector length must be specified. Third, the vector operations must be executed using intrinsics. The resulting output can be used by further vector operations or written back to main memory via DMA.

Figure 2.6 shows the steps needed to allocate and move two arrays into the scratchpad to be operated upon. A second code example shown in Figure 2.7 demonstrates how the vector engine would be used to compute a fully-connected layer, versus the host processor. This consists of multiplying an input vector by a weight matrix, producing and output vector, and adding a bias vector to this result.

To fully utilize the vector engine, the programmer should aim for long vectors, overlap any DMA transfers with computation, and use small data operands to

```
int *arrA = (int*)malloc(10*sizeof(int));
int *arrB = (int*)malloc(10*sizeof(int));

vbx_word_t *vA = (vbx_word_t*)vbx_sp_malloc(10*sizeof(vbx_word_t));
vbx_word_t *vB = (vbx_word_t*)vbx_sp_malloc(10*sizeof(vbx_word_t));
vbx_dma_to_vector(vA, arrA, 10*sizeof(int));
vbx_dma_to_vector(vB, arrB, 10*sizeof(int));
```

Figure 2.6: Code required to allocate and move data inside the scratchpad

```
scalar code:
for (int o = 0; o <= vector_out_length; o++) {
    vector_out[o] = 0;
}

for (int o = 0; o <= vector_out_length; o++) {
    for (int i = 0; i <= vector_in_length; i++) {
        vector_out[o] += vector_in[i] * weights[o*vector_out_length+i];
    }
}

for (int o = 0; o <= vector_out_length; o++) {
    vector_output[o] += bias[o];
}

MXP code:
// set inner loop vector length and outer loops iterations
vbx_set_vl(vector_in_length, vector_out_length);
// set outer loop increments; output++, reuse inputs, row++ of weights
vbx_set_2D(1, 0, vector_out_length);
vbxx_acc(VMUL, v_out, v_in, v_weights);

// set inner loop vector length
vbx_set_vl(output_vector_len);
vbxx(VADD, v_output, v_output, v_biases);
```

Figure 2.7: Calculating a fully-connected layer on a host processor versus the equivalent VectorBlox MXP instructions

exploit the available subword SIMD speedup.

2.3.3 Custom Vector Instructions

The VectorBlox MXP allows the use of custom vector instructions (CVI) [31]. Once added, these CVIs are used by a programmer the same as any other vector instruction, taking full advantage of the soft vector processor's pipelining and data

marshalling. The number of lanes for each CVI can be set independently from the vector processor, allowing the designer to instantiate only the number of lanes required for performance, minimizing the area used. This becomes especially important for custom instructions that are too large to replicate across all vector lanes. CVIs provide the ability to leverage custom hardware while keeping the design effort to a minimum.

2.3.4 Wavefront Skipping

A common problem that inhibits SIMD parallelism is control flow divergence. Branches in SIMD engines are commonly handled by masked or predicated instructions, but this requires all elements in the vector to be processed even if masked. This wastes performance by occupying potential execution slots with no-operations.

To reduce the impact of control flow divergence, the VectorBlox MXP supports wavefront skipping [32]. This is beneficial when multiple instructions are executed with the exact same mask. On a first pass, the mask is analyzed to memorize those wavefronts where all elements are masked off. On subsequent passes, during instruction execution, these 'empty' wavefronts can then be skipped entirely and occupy 0 cycles of the Arithmetic Logic Units (ALUs). This results in faster execution as a mask narrows to have fewer and fewer enabled elements. Custom vector instructions in the VectorBlox MXP can also be masked if the number of custom instruction lanes matches the number of vector processor lanes.

Chapter 3

Local Binary Pattern Custom Vector Overlay

Detecting and locating objects is a necessary first step in complex vision applications. In this chapter we implement an LBP-based object detection overlay and produce an end-to-end, real-time system for high accuracy face detection. Running on the soft vector overlay, we first achieve a 25× speedup over the baseline ARM Cortex-A9 processor. We then accelerate the critical inner loops by adding two hardware-assisted custom vector instructions to the overlay, for an additional 10× speedup. The custom vector overlay yields a total 248.4× speedup over the initial Cortex-A9 baseline. Collectively, the custom vector overlay requires fewer than 800 additional lines of custom RTL, including comments and blank lines. Compared to a previous hardware-only face detection system of comparable size, this work is 2.6× faster.

3.1 Approach

In object detection systems, objects can be found anywhere in an image frame. Traditional scanning approaches solve this by looking for a small fixed-size object at all positions in the image. This requires massive parallelism to process frames in real time. Viola and Jones introduced several key optimizations to make this manageable. First, integral images were used to accelerate feature calculation. Second, by using cascades of features, early, more discriminative features are used to reject candidate locations, which avoids evaluating any remaining features. These optimizations still do not provide enough performance at high resolutions. To go faster, many custom hardware engines have been developed that extract SIMD parallelism, testing the same feature at many locations in parallel. However, variable execution due to exiting early means typical SIMD processing is not efficient. This limits scaling of these solutions, and in turn, their performance.

In the approach presented here, we use a software-programmable vector processor, creating a vector overlay that uses SIMD parallelism with variable-length vector instructions. The processor contains a masking feature, which allows us to address the exit-early nature of the algorithm, skipping entire sections of a vector that have exited. We also focus on subword data sizes, which allows for more parallel computation. Our contributions include the creation of a custom vector overlay for LBP detection and quantifying the performance of each step in the optimization process. This includes the initial vectorization, the pre-computation of restricted MB-LBP patterns, the use of vector masking and the addition of each CVI (targeting LBP pattern generation and the LBP LUT operation). We also introduce a novel ILP formulation to solve for 8-bit representations of feature pass

and fail values. Our software-driven approach is 2.4× to 3.5× faster than previous custom hardware solutions, showing only minimal additional RTL is needed to achieve state-of-the-art results.

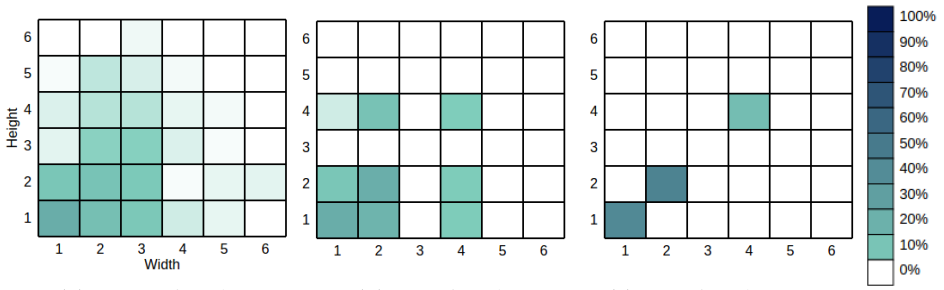
3.2 Adaptation

In this section, we adapt the algorithm to make full use of the soft vector overlay. Two changes allow for efficient vectorization. The first change restricts the block sizes of LBP features, allowing pre-computation of patterns to become feasible. The second change introduces a novel way to quantize stage thresholds and feature pass/fail values to 8 or fewer bits, using an ILP formulation. This allows the inner loop to produce and consume 8-bit values exclusively, enabling a high degree of subword SIMD parallelism.

3.2.1 Restricting LBP Block Sizes

Multi-block LBP patterns allow any block size to be used for a given feature. Using OpenCV's frontal face detector as an example, the histogram distribution of block (cell) sizes across features is shown in Figure 3.1(a). This graph indicates that smaller block sizes are used more often and that blocks typically have matching width and height.

By restricting block sizes, two things occur. First, when using only powers of two, the computation better fits SIMD-style execution. Second, adopting the same block size across many features leads to computational efficiency due to aliasing, which allows us to memoize. This is explained as follows: given a single window at a specific $x0$,$y0$ position, each feature uses a different offset, so there is no aliasing and all computations are necessary. However, across multiple windows at different



(a) Unrestricted **(b)** Restricted **(c)** Restricted Square

Figure 3.1: Distribution of block sizes of LBP features in the trained classifiers

starting positions, the computation for some feature i may be aliased to a feature j with the same block size. Thus, we can transform the innermost computation, which performs redundant work due to aliasing, into a lookup operation, where the work is done just once as a pre-computation. We must pre-compute an entire image worth of results, once for each block size. Reducing the number of block sizes also increases the amount of aliasing, which further improves the result. When restricting the block widths and heights to powers of two, we get the distribution shown in Figure 3.1(b). If add the additional restriction that width must equal height, we get the distribution shown in Figure 3.1(c).

Recent work by Bilaniuk et al. [4] had made similar observations: restricting block sizes to powers of two is better for SIMD engines, and using just three block sizes (1×1, 2×2, and 4×4) makes it more efficient to use pre-computation. In their results, these restrictions did not significantly change true positive detection rates, but it did increase false positives from about 1% to 5%.

To measure the accuracy tradeoff with restricted block sizes, we created three versions of the cascade. We used OpenCV's `traincascade` program to train

```
40,41c40,41
<   for( int w = 1; w <= winSize.width / 3; w++ )
<       for( int h = 1; h <= winSize.height / 3; h++ )
---
>   for( int w = 1; w <= winSize.width / 3; w=w*2)
>       for( int h = 1; h <= winSize.height / 3; h=h*2 )
```

(a) Restricted

```
40,41c40,42
<   for( int w = 1; w <= winSize.width / 3; w++ )
<       for( int h = 1; h <= winSize.height / 3; h++ )
---
>   for( int w = 1; w <= winSize.width / 3; w=w*2 ){
>      int h = w;
>      if (h <= winSize.height / 3)
```

(b) Restricted Square

Figure 3.2: Minimal code modifications to `lbpfeatures.cpp` generate cascades with restricted block sizes

the new cascade. By modifying `lbpfeatures.cpp`, we can exclude unwanted features sizes from the training process. Three versions of `traincascade` were used: an unmodified, unrestricted version; a set where width and height are individually restricted to powers of two; and a final restricted set, where width and height must match and be restricted to a power of two. This requires minimal changes to the source code, with differences shown in Figure 3.2.

Each cascade produced is valid and can be verified and used outside of our embedded implementation. Like Viola and Jones, we use the MIT-CMU test set (sets A, B, C) to quantify the trade off in accuracy and restricted features. The results for OpenCV's default frontal face cascades and the newly trained classifiers are presented in Table 3.1. The unrestricted and restricted classifiers perform better than OpenCV's default LBP frontal face cascade.

Our results are similar to Bilaniuk et al. Accuracy remains high across the restricted sets, though false positives do increase by a small amount. We proceed

Cascade	True Positives	False Positives	Positive Predictive Value
OpenCV lbpcascade_frontalface.xml	396	27	0.94
unrestricted_lbp_frontalface.xml	385	9	0.98
restricted_lbp_frontalface.xml	381	12	0.97
restricted2_lbp_frontalface.xml	386	17	0.96
Bilaniuk et al. [4]	364	23	0.94

Table 3.1: Accuracy of frontal face cascades ran on the MIT-CMU test set

to develop an implementation taking advantage of the restricted block sizes. sets.

3.2.2 ILP Formulation to Reduce Data Size

By supporting subword SIMD, the VectorBlox MXP provides increased performance for the smaller data sizes of 8 or 16 bits. Our vector code aims to use the 8-bit data size to maximize performance.

In most AdaBoost implementations, a stage passes or fails after a series of 32-bit floating-point values (representing pass or fail scores) are added together and compared to a 32-bit floating-point threshold. This uses more bits than necessary and may even be susceptible to round-off errors since floating-point addition is not commutative or associative. In this work, we show that this computation can use precise 8-bit integers instead. Integer computation is not susceptible to round-off errors. However, the pass/fail values for each feature must be carefully chosen to avoid overflows and the number of features per stage must be limited.

In training, the AdaBoost algorithm determines which features in a stage must pass for the stage to pass. More discriminative features are deemed more important, and given larger weight in the form of increased values in their pass or fail scores. Ultimately, however, this decision logic is encoded into a summation and comparison to a determined threshold. In our implementation, we use the exact same decision logic to write out a series of constraints for an ILP solver (Microsoft

Research's Z3 theorem prover [10]). These constraints allow the ILP solver to assign 8-bit pass and fail values, p_i and f_i, for each feature i, thus preserving the original logic. We are also able to assign a threshold of 0, so a positive stage total passes and a negative stage total fails.

For example, in a stage with 5 features, there are 2^5 constraints representing all combinations of each feature either passing or failing. Each constraint is given one clause to pass the stage (result ≥ 0) or fail the stage (result < 0). However, a complementary second clause is also necessary to avoid overflows (result ≤ 127 or result ≥ -128). An additional 10 constraints force each of the p_i and f_i values to be within the 8-bit signed integer range. Z3 is used to solve for these 10 values. An example of these constraints is shown in Figure 3.3.

One limitation of this approach is that Z3 starts having trouble with stages with too many features (e.g., 15 or more) using signed 8-bit constraints. We avoid this in practise by limiting the number of features per stage. Our final trained cascade uses 98 features across 12 stages, with at most 9 features per stage. Limiting the features per stage changes how often we perform the early-exit check, but does not affect accuracy.

Note that the vectorized version thus far remains bottlenecked on the preceding LUT computation that generates the pass or fail result for each feature. Hence, there is little performance advantage in switching from 32-bit weights to 8-bit weights in software at this point. However, in the next section, we will add a custom vector instruction to directly support the LUT operation and to include this accumulating logic to ultimately determine whether the stage will pass. At that point, the inner loop produces and consumes 8-bit operands exclusively, greatly increasing performance. This is an example of a hardware-motivated software change.

```
Subject To:

// fail stage if all features fail
f0+f1+f2+f3+f4 <=  -1
f0+f1+f2+f3+f4 >=-128

// pass stage if only 4 passes
f0+f1+f2+f3+p4 >=   0
f0+f1+f2+f3+p4 <= 127

// fail stage if only 3 passes
f0+f1+f2+p3+f4 <=  -1
f0+f1+f2+p3+f4 >=-128

// pass stage if both 3 and 4 pass
f0+f1+f2+p3+p4 >=   0
f0+f1+f2+p3+p4 <= 127
...

// pass stage if all features pass
p0+p1+p2+p3+p4 >=   0
p0+p1+p2+p3+p4 <= 127

Bounds:

-128 <= f0 <= 127
-128 <= f1 <= 127
...
-128 <= p4 <= 127
```

Figure 3.3: Sample ILP constraints for a stage with 5 features using Z3

3.3 Vectorization

The initial scalar implementation was compiled with `gcc -O3`, running bare metal on the ARM Cortex-A9 processor at 667 MHz. An object dump shows that the compiled code contains NEON instructions, but we did not try to optimize the scalar code to aid NEON auto-vectorization.

Next, manual vectorization for VectorBlox MXP across rows is quickly implemented and verified against the scalar implementation. This initial vectorization gives us an understanding of which functions are amenable to SIMD parallelization, and how performance may scale as more ALUs are added. We did not vectorize all of the code; we used profiling to identify and vectorize only the most compute-intensive loops.

Using a 320×240 test image, we perform a dense scan (scaling factor = 1.1, stride = 1) to compare with previous work [6]. The scalar core takes 1.9 seconds to complete. Our initial vectorized version of the code, running on 16 vector lanes, processes the test image in 0.76 seconds, a improvement of nearly $2.5\times$. The vector processing is done at 183 MHz, which is less than 1/3 of the clock rate of the scalar engine.

Only the innermost loops, which test many possible *x*,*y* starting positions of a given feature, are easily vectorized by SIMD instructions. These loops are nested within a loop testing all features within a stage, which is in turn, nested in a loop testing each stage in the cascade. These outermost loops cannot be easily SIMD-vectorized because each feature and each stage have unique properties. Furthermore, the *early exit* condition means that as soon as a stage fails, no other stages need to be tested at that position. This performs well in scalar or Multiple Instruc-

tion, Multiple Data (MIMD) implementations, but leads to low utilization in SIMD engines as the early exit positions are masked off within the vector, but continue to use execution slots as no-ops until all positions fail or all stages are tested.

After the initial vectorization, the inner loop computing each feature of a given stage continues to consume the most runtime (93%). Other components, including grey-scaling and image pyramid creation (via bilinear interpolation), consume the next-most runtime (7%). This downscaling later becomes a key bottleneck and is later vectorized. The merging of detected features to produce the final results consumes minimal runtime ($<$1%) and is not vectorized.

The innermost loop consists of two parts: computing a LBP pattern for the current feature, and performing a table-lookup using this pattern as the index to determine the contribution towards passing the stage. By using the restricted features discussed above, it is fairly simple to hoist the computation of the LBP pattern out of the inner loop into a pre-compute step. This means that the inner loop needs to touch less data and fewer computations are required. Also, overlapping features due to aliasing do not need to be recomputed. The switch to restricted square features with pre-computed features produces an additional 1.8$\times$ speedup over the initial vectorized solution when using 16 lanes.

3.3.1 Applying Wavefront Skipping

Long vector lengths, often good for performance, can act against the benefit of exiting early. This is seen in Figure 3.4, where image (b) shows the amount of computation done at each location in the original image (a); bright pixels indicate highly probable locations for a face, where most features will pass. In contrast, black pixels indicate an exit-early condition. The extra work done by row-based

vectorization is shown in image (c). Here, we see computation for the whole row must continue until the last pixel is done. Finally, image (d) shows how some work can be skipped when using VectorBlox MXP masks: entire wavefronts within a vector that are masked off can be skipped entirely and do not consume execution slots.

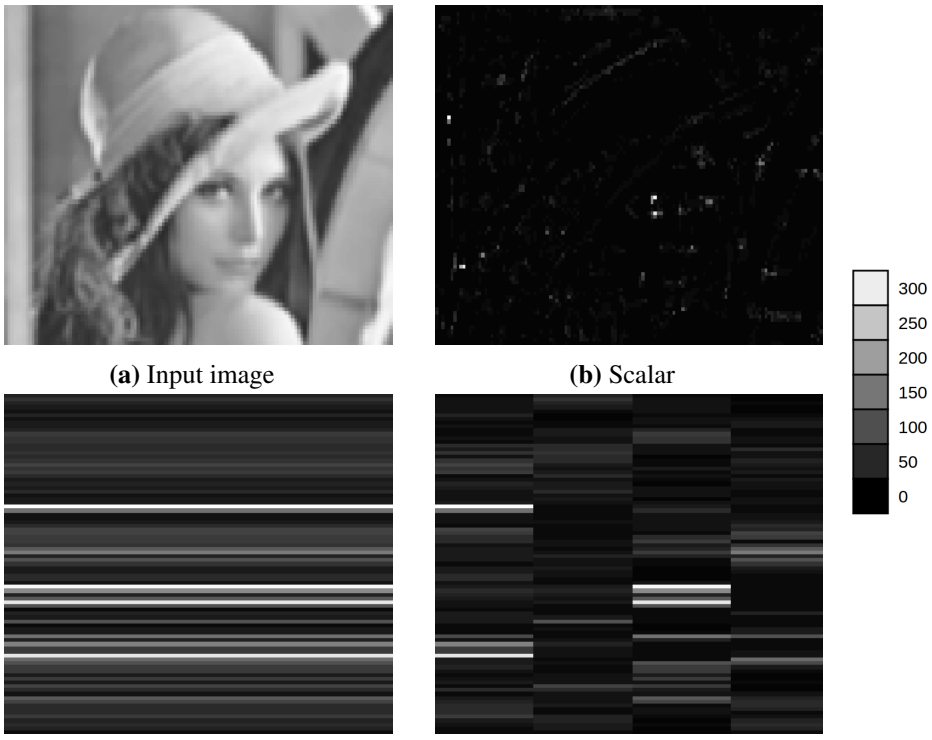
After setting up a mask for the vector, we iterate through stages and update the mask after every stage. When all locations within a wavefront exit early, the vector engine skips that wavefront. The speedup when using masked instructions is 6× versus the non-masked version.

3.4 Custom Vector Instructions

In this section, we accelerate the computation further with two key custom vector instructions (CVIs). After the algorithmic refinements above, profiling reveals that the table lookup operation dominates runtime. This operation is an ideal candidate for implementation as a custom vector instruction. Once accelerated, the LBP pre-computations become the bottleneck. The LBP pre-computation also requires a CVI. These two custom vector instructions, one for table lookup and one for LBP pre-computation, are described below.

3.4.1 LBP Table Lookup Instruction

The LBP table lookup operations are accelerated with the first custom instruction. Algorithmically, this instruction implements the two steps shown in Figure 2.3(c) and (d), representing the lookup followed by an addition. The input is an 8-bit value corresponding to the LBP pattern, shown as the value 56 in the figure, which is the result of the 8-way LBP comparison. (Further details about the LBP comparison



(a) Input image **(b)** Scalar

(c) Simple rows **(d)** Masked rows

Figure 3.4: The number of features calculated at every location is shown. The bottom demonstrate parallelizing across a row, with the latter taking advantage of masked instructions

are given in the next section.) The output is an 8-bit value, produced by the second step where two parallel table lookup results are added together.

The first step is a table lookup into a 256-entry table with a 1-bit output. The output bit selects one of two 8-bit values for the feature (aptly named `PASS` or `FAIL`). Thus, each feature requires 272 bits of storage.

To maximize performance, we perform two of these table lookups in parallel, using the two 8-bit input operands of the custom instruction for two subsequent features. Since the custom instruction can only provide a single output, the PASS

or FAIL results of each of the two features are added internally into an 8-bit partial stage total. This is the output of the second step of our custom instruction.

To determine whether a stage passes or fails, the results of all features in the stage need to be accumulated. This is done using regular 8-bit vector add instruction, outside of this custom instruction. It accumulates all the partial stage totals into a final stage total. After processing all features, this final stage total is compared to a threshold of zero to determine whether the stage passes.

As mentioned, the custom instruction invokes two parallel table lookups in parallel in each lane. This is done by reading two different 8-bit LBP patterns and presenting them to the CVI as operands A and B, respectively. This requires a 544-bit wide memory. However, since all 8-bit vector lanes are processing the same feature, this memory is shared across the entire vector engine. Back-to-back custom instructions that perform table lookups automatically increment a feature counter, which is used to address into the 544-bit wide memory. This memory must be large enough for all features across all stages; the starting address is determined by the stage number. This face detection cascade contains a total of 12 stages and 98 features. With 2 features per row, and 8 stages having an odd number of features, a total of 53 rows of memory are required (less than 32 kB in total). The memory itself is initialized using another simple custom instruction.

Four of these dual-8-bit-lookups can fit into a 32-bit lane of a custom vector instruction. For example, in a 16-lane configuration, 128 table lookups are done every cycle.

Without the CVI, this table lookup requires approximately 15 regular vector instructions, several of which operate on 32-bit operands. The majority of the runtime was originally spent computing and checking the features of each

```
#define VLUT VCUSTOM0
for(f = 0; f < cascade[stage].n; f+=2) {
  // sz = MB-LBP size, w = image_width
  feat_a = cascade[stage].feats[f];
  feat_b = cascade[stage].feats[f+1];
  v_lbp_a = v_lbp[fa.sz] + feat_a.dy*w + feat_a.dx;
  v_lbp_b = v_lbp[fb.sz] + feat_b.dy*w + feat_b.dx;
  vbx_masked(VVB, VLUT, v_lut, v_lbp_a, v_lbp_b);
  vbx_masked(VVB, VADD, v_sum, v_sum, v_lut);
}
```

Figure 3.5: The inner loop using a custom vector instruction

stage. Even after restricting LBP block sizes and transforming this work into a pre-computation, the lookup table operation and stage pass/fail computation still occupies 57% of the runtime. Two of these table lookups (30 regular vector instructions) are reduced into a single CVI. Since the CVI operates only on 8-bit data, more parallelism is available than regular 32-bit vector instructions.

Due to the ILP formulation, the summation of 8-bit values is guaranteed to be sufficient without risk of overflow or rounding. This allows each 8-bit input pattern to produce an 8-bit output.

The final code of the innermost loop is presented in Figure 3.5. The scalar housekeeping is done in parallel with two vector operations: vector table-lookup, and vector add.

Since the LUT contents are not hardcoded, but loaded at runtime, this system can be used to detect different types of objects beyond faces. It can also be used in detection chains, e.g. first detect a face, then detect eyes within a face.

Adding the custom table lookup instruction to the masked vectorized algorithm results in an additional 4.2$\times$ speedup.

3.4.2 LBP Pattern Instruction

After the table lookup is sped up with the first custom vector instruction, computation of the LBP patterns becomes the bottleneck. This is the process of computing the 8-way neighbour comparison to the center block in each 3×3 windowed region of the source image. This must be repeated for 1×1, 2×2 and 4×4 block sizes in our restricted design, producing three separate 8-bit arrays as output. Due to feature reuse at different offsets, this is done as a pre-computation rather than on-the-fly. Restricting the variety of LBP features (in terms of block sizes used) increases the frequency of reuse, and improves the advantage of the pre-computation.

Since regular ALU operations have two inputs and one output, computing LBP patterns with 9-inputs and 1-output for a 3×3 window is not straight-forward. The fan-in is larger still for the larger block sizes.

To solve this, we use a stateful pipeline that processes columns of data, one vertical stripe at a time. The output stripe width is a set of 8-bit values that match the wavefront width of the overlay's SIMD execution unit, as shown in Figure 3.6. Producing output values at the far left or right edges of this stripe requires reading source image pixels past the left edge or right edge. To accomplish this, the custom instruction is supplied with two overlapping rows as input operands: one operand includes the input pixels past the left edge of the output stripe, while the other operand is offset enough to include the input pixels past the right edge of the output stripe. The custom vector instruction iterates one row at a time down the image. The custom instruction has three modes, one for each of the block sizes, and therefore requires three passes, before moving over to the next column. When starting the next column, a 2D DMA is performed to convert the vertical stripe in

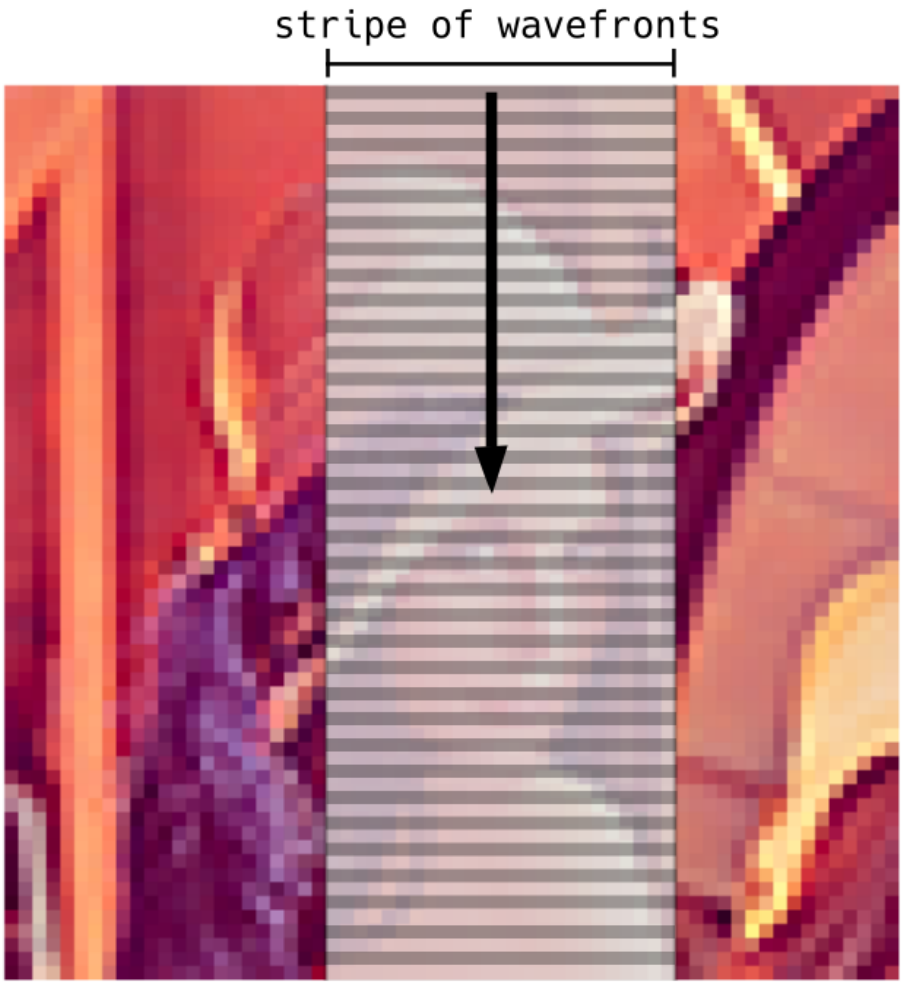


Figure 3.6: Accelerating the pre-computation of LBP patterns

the source image into a packed image array in the internal scratchpad.

Each pass of the custom vector operation can be separated into two steps. The first step reduces (adds) the rows and columns of byte-sized image data according to the LBP block size. This is skipped when block size is one, but requires reducing 2×2 and 4×4 regions for other block sizes. The second step takes the reduced values, and compares the center to its 8 neighbours, producing the 8-bit LBP pat-

terns. Adding the CVI to pre-compute the LBP patterns results in an additional 2.4× speedup.

3.5 Results

Below, we first detail the performance and area of the LBP overlay at each stage of customization. In the following sections, we will compare our results to previous work.

3.5.1 Experimental Setup

Results were obtained using a Xilinx ZC706 FPGA development board with a Zynq 7Z045-2 device. FPGA builds were done using Vivado 2014.2. All designs are synthesized for a 200 MHz target clock speed, with the reported worst-case negative slack used to compute Fmax. The Zynq 7000 series of FPGAs are all manufactured on a 28 nm silicon technology node.

Processing time is reported for one 320×240 image, producing an image pyramid with scale factor of 1.1, and using a stride of 1 to produce a dense scan. These default settings match those of Brousseau [6]. Generally, however, 1080p60 video can be consumed and produced, and processing can be sped up with larger scale factors and strides.

A full face-detection system was implemented with a 1080p HDMI video camera, FPGA board, and display. A photo of the display output is shown in Figure 3.7, where a test image is presented to the camera using an iPhone. In this case, 49 of 50 faces are detected in 36 ms, despite the small face sizes and the angle of presentation.

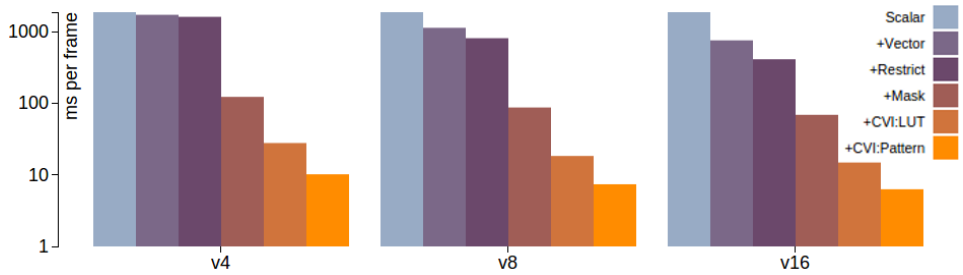
Figure 3.7: Photo of video output showing 49 of 50 faces detected on a 1080p image in 36 ms

3.5.2 Performance

The contributions of each optimization to performance are shown in Figure 3.8. Wavefront skipping and the LUT custom vector instruction boost performance the most. With all optimizations in place, diminishing returns with respect to vector length become apparent, showing a limit to our SIMD approach. For additional performance gains, a multi-core implementation would be required.

3.5.3 Area

The FPGA resources required for the base system (containing only HDMI I/O) and various vector engine sizes are shown in Table 3.2. As a rough comparison, the hardware face detection system developed by Brousseau [6] runs at 125 MHz on a Stratix IV 530 (the largest available) and uses similar area to our largest configuration, but our work is 2.6× faster, offers a complete end-to-end (camera to display)



Vector Lanes	Software Only				With Custom Instructions	
	Scalar	Cumulative optimizations				
		+Vector	+Restrict	+Masked	+CVI:LUT	+CVI:Pattern
4	1,885.4 (1.0×)	1,731.8	1,646.6	125.2 (15.1×)	28.0	10.3 (183.9×)
8	1,885.4 (1.0×)	1,150.4	823.6	88.5 (21.3×)	21.0	8.5 (222.9×)
16	1,885.4 (1.0×)	831.4	452.5	75.6 (24.9×)	18.0	7.6 (248.4×)

Figure 3.8: Performance in milliseconds (speedup) on 320 × 240 image pyramid, 1.1 scale factor, unit stride

	# FF	# LUTs	Mem. (LUTs)	Mem. (BRAM)	# DSP	Fmax (MHz)
ZC706 Max.	437,200	218,600	70,400	545	900	-
video only	8,873	7,028	514	12	8	236

# Lanes	CVIs						
4	no	16,843	16,437	869	54.5	36	197
	yes	25,679	25,237	1,194	88.5	36	199
8	no	22,235	23,223	1,123	47	64	198
	yes	37,856	39,377	1,651	81	64	175
16	no	36,165	36,672	1,672	48	120	183
	yes	65,430	68,444	2,629	82	120	166

Table 3.2: Resource usage and Fmax

system, and keeps the majority of the face detection system in software.

3.6 Previous Work

Embedded face detection systems have been explored on various platforms. We compare our work with several FPGA implementations use custom hardware en-

gines and a SIMD-based Application-specific Integrated Circuit (ASIC) implementation. We match input image sizes and scale factors.

Brousseau [6] implemented a 32-core hardware engine on Stratix IV 530 FPGA, using Haar features. The design runs at 125 MHz and uses similar number of LUTs and digital signal processors (DSPs) as our largest configuration, using approximately 160K LEs and 550 18-bit multipliers. They were able to process 320×240 images in 20 ms. Each processing element (PE) processes a different location, and each PE stops processing once the cascade exits, saving power. Images are sent over USB and post-processing of detected features is performed on an external PC. This design computes integral images to assist with feature computation.

Cho [7] implemented an 8-core hardware engine, processing Haar features, on a Xilinx Virtex-5 FPGA. The design uses nearest-neighbour interpolation with a scaling factor of 1.2 and can synthesize to run 320×240 and 640×480 images. The design could process 320×240 images in 16.4 ms, and 640×480 images in 37.4 ms. Like the previous design, integral images are used to minimize feature computation.

Gao [16] implemented a 16-core hardware engine, again processing Haar features with a scaling factor of 1.2. Images are restricted to 256×192 pixels and can be processed at 10.2 ms. The design runs at 125 MHz, implemented on a Xilinx Virtex-5 FPGA. Only the Haar classifier step was implemented on the FPGA, leaving pre-processing and post-processing on the host PC.

The previous approaches provide good performance, but are fixed in their implementation details. Our custom overlay approach can accelerate both pre- and post-processing and allows scaling factors and interpolation schemes to be changed quickly without changing the hardware.

Prior Work	Feature Type	Platform	Image Res.	Scale Factor	Prior (fps)	This (fps)	Speedup
Brousseau [6]	Haar	FPGA (40nm)	320x240	1.1	50	132	2.6
Cho [7]	Haar	FPGA (65nm)	320x240	1.2	61	214	3.5
			640x480	1.2	16	56	3.5
Gao [16]	Haar	FPGA (65nm)	256x192	1.2	98	236	2.4
Bilaniuk [4]	LBP	ASIC (unknown)	640x480	1.1	5	34	6.8

Table 3.3: Previous work comparison

Using a programmable approach, Bilaniuk [4] implemented their embedded face detection system on a SIMD-based ASIC with 96 16-bit lanes. Like our overlay, LBP features were used for computational efficiency. The authors also restricted the LBP cell sizes to powers of two (1×1, 2×2 and 4×4). The cascade they produced increased false positives, similar to what we observed. Their design was able to process 640×480 images in 200 ms. This approach is close to our soft vector overlay, but as custom hardware is not available, the algorithm cannot be accelerated beyond the use of its dedicated SIMD instructions.

Table 3.3 compares our performance, in frames per second, to the prior work discussed above. Our performance is $2.4\times$ to $3.5\times$ faster than these pure hardware implementations. The last row of this table is a fixed-width SIMD CPU implemented as an ASIC where our work is $6.8\times$ faster. In all of these cases, we were able to set the image resolution, scaling factor and stride to match the prior works.

3.7 Design Complexity

The majority of the code is accelerated by software-based vector instructions. The bottlenecks in the accelerated algorithm, used for LBP feature calculation and table lookup, were accelerated as CVIs. These two CVIs require an additional 795 lines of RTL. Together, they are responsible for the calculation of LBP features,

accounting for 87 lines of software. The remainder of the software requires approximately 2500 lines of C code.

The first CVI, used to pre-calculate the LBP patterns, requires 420 lines of RTL. This hardware replaces approximately 60 lines of the original scalar code. The second CVI requires 375 lines of RTL. This hardware replaces approximately 30 lines of the original scalar code.

3.8 Summary

The LBP custom vector overlay achieves a speedup of 248.4× over the ARM Cortex-A9 software implementation. It represents a software-driven, flexible solution that is 2.4× to 3.5× faster than previous custom hardware solutions and 6.8× faster than an ASIC-based SIMD processor. The overlay can be adjusted to run at various resolutions and can add or modify additional processing without a hardware rebuild. Customizing the overlay only requires about 800 lines of custom RTL, keeping the majority of the development effort on software.

Chapter 4

Convolutional Neural Network Custom Vector Overlay

Rapid advances in deep learning are vastly improving the accuracy of object recognition and localization. In this chapter, a custom vector overlay is developed to accelerate inference computations in convolutional neural networks with 8-bit quantized activations. Several popular neural network instances are used to demonstrate the capability of the overlay. A dual-core overlay is implemented, which uses both a hard ARM Cortex-A9 processor and a soft MicroBlaze processor. Each processor is augmented with vector instructions and convolution accelerators. At 116 MHz on a 28 nm Xilinx Zynq 7Z045-2, the dual-core overlay has a peak performance of 352.6 GOPS. On one example network, a throughput of 175.0 GOPS is realised. In comparison, the same network running under the Darknet software framework achieves roughly 7.0 GOPS throughput on a single core of an Intel i7-2600 CPU implemented in 32 nm technology.

To produce this custom overlay, only a single custom vector instruction to accelerate convolutions is required. The instruction is described in 1300 lines of custom RTL. We anticipate that larger systems can be built on more modern FPGAs and achieve a level of performance that rivals GPUs. A key benefit of this software-driven overlay approach is that new networks of any size can be compiled and run just like a CPU or GPU, without regenerating the FPGA bitstream. This is important in embedded systems where frequent updates or changes to the neural networks are required.

4.1 Approach

The explosion in popularity of deep learning and convolutional neural networks has resulted in many hardware-based approaches for fast inference of CNNs. Most of these approaches result in a fixed solution which has limited flexibility and requires regeneration if the network changes significantly. These systems are quite complex in that they must analyze the network to determine an appropriate set of buffer sizes and the size of hardware components to compute each layer in a balanced way. In contrast, this chapter follows the theme of the thesis in applying a software-based approach to accelerate CNNs. This approach retains enough flexibility that many network changes can be incorporated without regenerating the hardware, yet still provides excellent performance that rivals the hardware approaches.

In particular, on VGG, we find that a custom vector overlay with a single custom vector instruction is enough to achieve a speedup of over 1450 times compared to the host ARM Cortex-A9 processor. The vector overlay sufficiently accelerates all other processing not done by the CVI, including fully-connected layers and activation functions. In this chapter we optimize several popular networks, reduc-

ing the precision from 32-bit floating point to smaller, fixed-point representations. These are implemented both with and without the CVI that accelerates the 3×3 convolutions required by the networks. This instruction requires 1300 lines of custom RTL. We also show that a dual-core implementation achieves nearly perfect $2\times$ speedup over a single core.

4.2 Adaptation

In this section, we examine and adapt network properties needed to support computer vision within the overlay framework. First, we observed the trend in CNNs is towards smaller filter sizes of 3×3 and 1×1 with unit strides, so we selected two popular networks for implementation, YOLO and VGG, that follow this trend. Next, to maximize the parallelism available in FPGAs, we verified that fixed-point quantization does not lead to significant loss of precision.

4.2.1 Tiny YOLOv2

Two variants of Tiny YOLOv2 are used, trained with both the 20-category VOC and the 80-category COCO challenge datasets. Tiny YOLOv2 VOC is used to measure the effect of the network quantization and reduction described below. In general, these networks consist of alternating 3×3 convolution layers and max pooling layers. A 1×1 pointwise convolution layer is used for the last layer. The networks' hyperparameters are both the same except for this last layer, as COCO has four times as many categories as VOC. The Tiny YOLOv2 VOC network structure is described in Figure 4.1.

The output stage, interpreting the network's output values, is performed by the ARM Cortex-A9 host processor. This stage currently uses expensive, floating-point

Type	Filters	Size/Stride	Output
Convolutional	16	3×3	416×416
Maxpool		$2 \times 2/2$	208×208
Convolutional	32	3×3	208×208
Maxpool		$2 \times 2/2$	104×104
Convolutional	64	3×3	104×104
Maxpool		$2 \times 2/2$	52×52
Convolutional	128	3×3	52×52
Maxpool		$2 \times 2/2$	26×26
Convolutional	256	3×3	26×26
Maxpool		$2 \times 2/2$	13×13
Convolutional	512	3×3	13×13
Maxpool		$2 \times 2/1$	13×13
Convolutional	1024	3×3	13×13
Convolutional	1024	3×3	13×13
Pointwise Convolutional	125	1×1	13×13

Table 4.1: Tiny YOLOv2 VOC hyperparameters

operations and adds 6-18 ms to every frame processed, depending on the network. No attempt was made to speed this up or translate it to fixed-point. Without the speed of the ARM core, this could be vectorized as well. No other stage of the detection process depends upon the ARM core, so it potentially can be replaced by soft cores with minimal performance impact.

Input maps to the convolution layers are zero padded, keeping the output maps the same size. Pooling layers in this network take a 2×2 region and reduce it to a singular, maximum value. Padding and pooling take minimal time compared to the convolution computations.

4.2.2 VGG16 SVD500

VGG16 is a very deep 1000-category ImageNet classifier, consisting of 16 layers. Like the YOLO networks, VGG16 consists mainly of alternating 3×3 convolution

layers and max pooling layers. The padding and pooling seen in YOLO match this network. Unlike YOLO, however, VGG16's last layers consist of large fully-connected layers. These layers are difficult to accelerate, as they quickly become memory bound, containing millions of weights. One approach to reducing the size of fully-connected layers is Singular-Value Decomposition (SVD) which splits a fully-connected layer into two smaller fully-connected layers. We adapt the same SVD decomposition as Qui et al [26], allowing implementations to be compared. The final network structure of VGG16 is presented in Figure 4.2.

Post-processing is significantly simpler than YOLO, as each of the 1000 outputs represents a score for a particular category and only requires sorting.

4.2.3 Quantization

Most neural networks are trained with 32-bit floating-point to maintain high accuracy during back propagation. However, low-precision fixed-point values can be used during training as well as inference [18] [19] [27]. In the next chapter, we explore networks where low precision is taken to the extreme, using binary values for weights. In this chapter, however, we reduce networks originally trained with floating-point values to use low-precision fixed-point integers, a process called quantization. Our goal is not to find the best possible quantization method. Instead, it is simply to show that it can be done with low accuracy loss, allowing us to use low precision in our custom vector overlay.

Low-precision, fixed-point quantization of floating-point networks can be seen in Tensorflow, a leading neural network framework [1]. Earlier work by Farabet and LeCun exploring CNN inference on FPGAs used 16-bit weights, 8-bit activations, and larger 48-bit values for accumulation [14]. Following this reduced-

Type	Filters	Size/Stride	Output
Convolutional	64	3×3	224×224
Convolutional	64	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	128	3×3	112×112
Convolutional	128	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	256	3×3	56×56
Convolutional	256	3×3	56×56
Convolutional	256	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	512	3×3	28×28
Convolutional	512	3×3	28×28
Convolutional	512	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	512	3×3	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Fully-Connected		25088	500
Fully-Connected		500	4096
Fully-Connected		4096	1024
Fully-Connected		1024	1000

Table 4.2: VGG16 SVD500 CNN hyperparameters

precision quantization trend, we quantize the networks to expose more parallelism in the soft vector overlay by exploiting subword SIMD instructions and building low-precision structures in the custom vector instructions.

The quantization scheme is kept simple. First, we run several hundred sample images through the original floating-point network, recording the ranges of values seen in each layers' output. Weights and biases are then scaled on a layer-by-layer basis, normalizing a layer's outputs to 1.0. This allows for a simple fixed-point representation. More complex quantization schemes could be explored in

the future. Code book compression schemes like those discussed in [26] may be necessary to minimize memory bandwidth with larger fully-connected layers, but were not needed for this work.

Both an initial 32-bit floating-point and 16-bit fixed-point versions were implemented on the ARM host processor. These serve as baseline versions to compare to lower-precision versions or their vectorized equivalents. Vectorization of the 16-bit version is described later in the chapter.

Translating to 8-bit or even 16-bit fixed-point values can be problematic. This was explored with the bit-accurate VectorBlox MXP simulator, where significant problems with underflows and overflows were noticed. In particular, a quick glance at the network structure in Table 4.1 shows values from up to 1024 input maps must be summed to produce each output map. If the summation occurs with just simple 8-bit arithmetic, it is clear that overflows will be a problem. We also detected significant problems with underflows; values that were normally very close to zero, but negative, tended to get truncated down to -1 rather than 0; these accumulate very quickly across 1024 inputs.

For the 16-bit fixed-point version, VectorBlox MXP's fixed-point specific addition and multiply instructions address these issues. The fixed-point instructions saturate outputs to prevent overflow, and the fixed-point multiply rounds outputs, which mitigates underflow. When we explored further reducing data sizes, i.e., setting both weights and activations to 8-bits, significant loss in accuracy was observed. As the vector processor requires both input operands to be of the same size, neither weights nor activations can be switched to 8-bits independently.

To get around this restriction posed by the processor, i.e. address the loss in accuracy of 8-bit operands, and accelerate inference on the overlay, a CVI is added

to perform convolutions. The CVI, discussed later in the chapter, loads in higher-precision weights ahead of time so low-precision 8-bit activations can benefit from the increased parallelism. The CVI internally adds results from multiple input maps at once and carries forward up to 29 bits of precision. The partial summations produced by the CVI are rounded down to 16 bits. These partial output maps are summed to produce the final output maps, which can be eventually truncated to 8 bits. In experiments running on the simulator, this strategy achieves good accuracy.

4.2.4 Quantizational Impact on Accuracy

To measure the impact of quantization for each implementation, we run Tiny YOLOv2 VOC and calculate the mAP, the standard method for measuring accuracy. Calculating the mAP requires several steps. First, inference is run on the VOC2007 challenge test set. The set contains 4952 images. The floating-point version of Tiny YOLOv2 VOC runs on Darknet, while the reduced-precision versions run on the overlay. Second, for each of the 20 output categories, precision-recall curves are generated and the average precision for each category is measured, taking the area under the curve. Recall is the percentage of actual objects found and precision is the percentage of true positives in the found results. The recall is set to specific values by raising or lowering the cutoff threshold. The precision is plotted for these recall values. A typical curve for the "person" class is shown in Figure 4.1. Finally, the mean of the average precision values found from each category is calculated, to produce the final accuracy metric. Taking the average precision across all categories, we calculate the mAP for each quantization strategy or implementation.

The resulting mAP of the different implementations is presented in Table 4.3. We start with implementations using standard vector instructions. The 16-bit fixed-

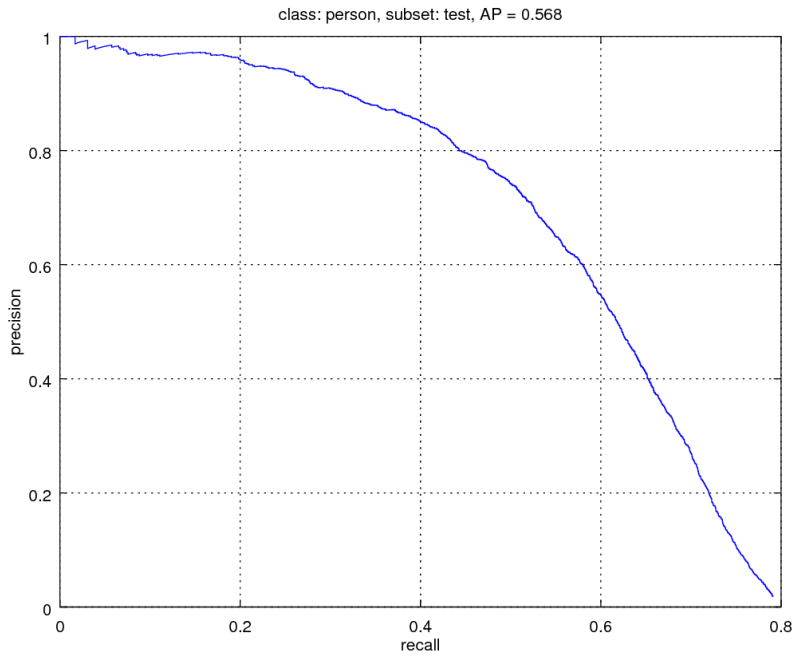


Figure 4.1: Average precision curve for "person" class

point version sees a minimal drop in mAP score, while reducing both activations and weights further to 8-bit sees a large decrease.

The last three rows of the table shows implementations that use the CNN custom instruction. Input and weight data sizes can vary independently, as weights are pre-loaded separately. Weights, which are reused across input maps, can be set at higher precision, while inputs can be set to 8 bits to maximize throughput. The custom instruction is parameterized by input and weight size. Depending on the number of bits used for inputs and weights, the number of DSP blocks required varies. We find the accuracy loss becomes large as both operands approach 8 bits. The quantization scheme we choose for the custom vector overlay uses 8-bit inputs

Operand type	Activation bits	Weight bits	Platform	mAP
floating-point	32	32	Darknet	56.5
floating-point	32	32	ARM A9	56.5
fixed-point	16	16	Overlay	56.1
fixed-point	8	8	Overlay	40.5
fixed-point	8	16	Custom Overlay	54.9
fixed-point	8	12	Custom Overlay	54.3
fixed-point	8	8	Custom Overlay	42.8

Table 4.3: Mean average precision of various versions of Tiny YOLOv2 VOC

and 12-bit weights, as accuracy remains high while the CVI still maps very efficiently to the DSPs in the FPGA. Note the fully-connected layers in VGG16 use standard vector instructions, hence inputs and weights remain at 16 bits.

4.3 Vectorization

With the quantization effects understood, we start by vectorizing the 16-bit fixed-point scalar implementation. Inference consists of three steps; pre-processing, computing each layer in the network, and post-processing.

Pre- and post-processing are needed to produce a complete system. Pre-processing consists of separating interleaved RGB pixels into input maps, followed by down-scaling. This is similar to the pre-processing required in the previous chapter, and the vectorized routines are reused here. Post-processing for YOLO is kept on the ARM host, as it accounts for a small amount of the runtime. The operations could be vectorized, but need not be. For VGG, post-processing requires a simple sorting of the output vector, and is again kept on the host. Our approach allows software reuse, and requires that we only vectorize what is necessary for performance.

The majority of vectorization effort is spent creating efficient vectorized versions of each layer, in particular convolution and fully-connected layers. Although

the majority of the operations in these networks are in the convolution kernel, additional operations require vectorization to make a fully functioning system without bottlenecks. This includes padding, pooling and activation functions.

As inputs and outputs must be operated on inside the scratchpad, maps are continuously being transferred by DMA between off-chip DRAM and the scratchpad. To minimize memory transfers, when processing a map, all additional operations, including padding, pooling and activation functions, are chained with the core convolution or matrix multiply operations. This allows the outputs to be fully processed before being sent back to main memory. Larger maps, too big to fit fully in the scratchpad, are processed in a tiled fashion.

Standard vector instructions efficiently accelerate these additional operations. The ReLU and leaky ReLU activation functions, which retain zero and 10% of negative values, respectively, are used in the example networks. Maxpool layers, commonly performing a 4:1 data reduction, keeping the maximum value across every 2×2 pixel group, are also used heavily. These operations make heavy use of the comparison (subtract) and conditional-move instructions provided by the VectorBlox MXP processor.

The vectorized implementation, which makes use of saturation and rounding incorporated into the fixed-point multiply and addition instructions, serves as the soft vector overlay result. A speedup of $11.6\times$ over the baseline is observed.

4.4 Custom Vector Instructions

In this section, we accelerate the computation further with a single custom vector instruction (CVI). The vast majority of operations are in the convolution kernel; all input maps are convolved and summed repeatedly to produce each output, while all

other operations are applied only once at the end of each layer. The custom vector instruction to accelerate 3×3 convolutions is described below.

4.4.1 3x3 Convolution Instruction

As kernels are kept constant across input maps, bandwidth is wasted if kernels are used as one of the input vectors. By pre-loading weights, multiple input maps can instead be processed at once, increasing throughput and decreasing the amount of partial summations required outside the CVI. This is amplified by interleaving the input maps so every 32-bit element of the input vector contains four 8-bit elements from four successive maps. Instead of accelerating a single convolution kernel, the instruction accelerates eight kernels simultaneously, which we refer to as a super-kernel.

The 3×3 CVI is shown in Figure 4.2 for a V4 configuration with 2 'CNN super-kernel' operators. The maximum number of super-kernels is always $kernel\ width - 1$ less than the number of 32-bit lanes. Interleaved input matrices are passed into the instruction, and each super-kernel convolves these 8 input matrices (A, B, C, D, E, F, G and H) in parallel and sum-reduces them into a single output, enabling a 8:1 sum-reduction to be done with extended precision inside the CVI.

Each convolution contains 17 operators, so each super-kernel contains 143 operators ($17 \times 8 + 4 + 2 + 1$). The implementation uses a V16 CVI with 13 parallel CNN super-kernels, totalling **1859 parallel operations per cycle**. While upto 14 parallel CNN super-kernels could be used here, we only instantiated it 13 times to save a bit of area. In one cycle, the CVI receives 15 horizontally adjacent 8-bit pixels from each of the eight input matrices, and computes 13 horizontally adjacent 16-bit outputs. The CVI proceeds to the next row each cycle. After computing

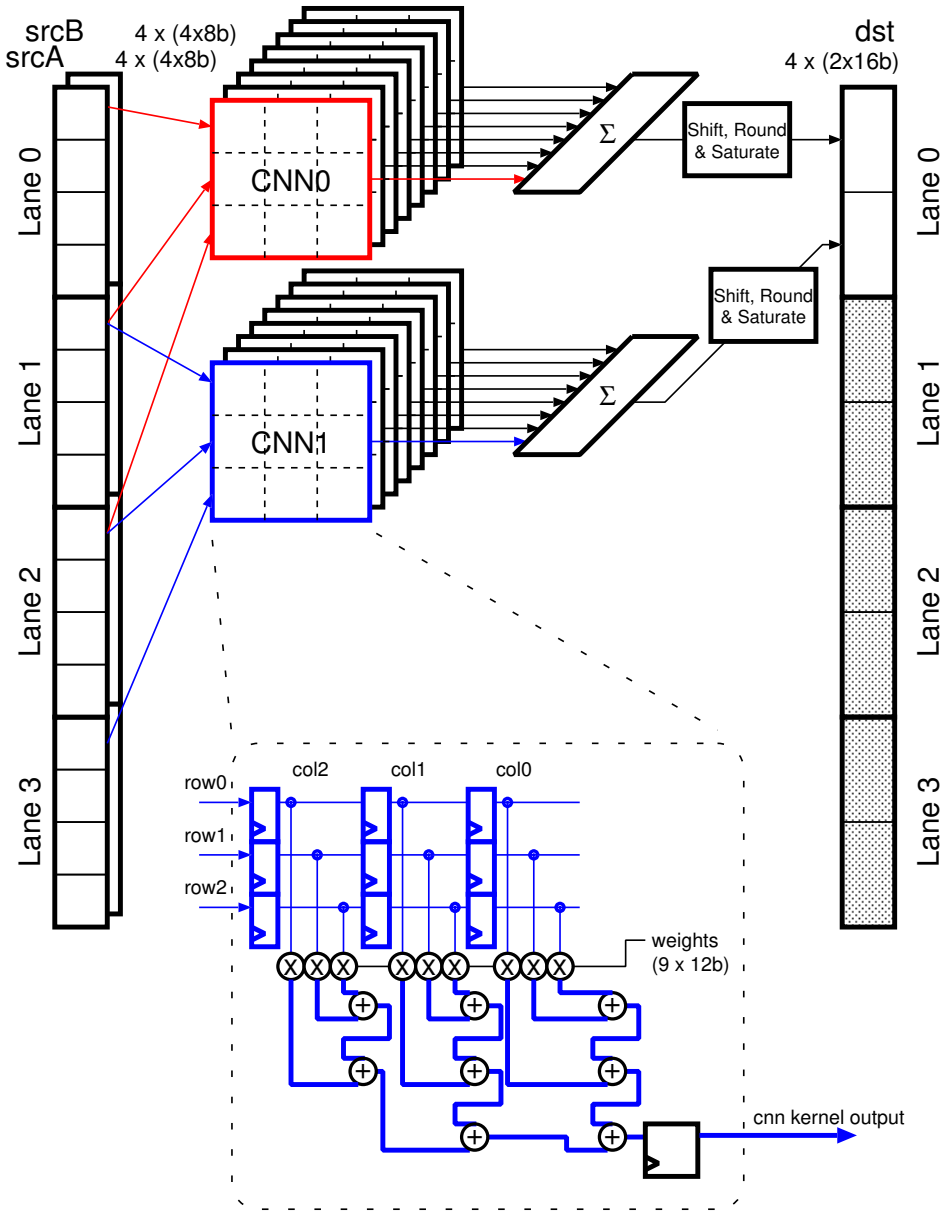


Figure 4.2: Convolution instruction (showing a V4 system with 2 CNN super-kernels attached)

an entire column, the CVI advances horizontally by 13 pixels to the next column. Input padding ensures all output data is computed.

Inputs to the inner convolutions are 8-bit, and internal widths increase as required. The vector srcA and srcB operands each hold four matrices of interleaved 8-bit data. The output, after the 8:1 reduction uses 29 bits. After this, shifting/rounding/saturating logic reduces the output to 16 bits. These latter operations are not counted as ops by YOLO, since they are not needed in a floating-point implementation. We don't count them here either.

4.5 Results

Results for the custom vector overlay are presented in this section. Initially, we used a single ARM Cortex-A9 as the host processor. To investigate potential for multi-core scaling, we also add implementations containing a second core, also equipped with a VectorBlox MXP processor. The second core host processor is a much slower MicroBlaze soft processor.

4.5.1 Experimental Setup

Results were obtained using a Xilix ZC706 FPGA development board with a Zynq 7Z045-2 device. FPGA builds were done using Vivado 2017.1. The ARM Cortex-A9 processor and camera/video framebuffers primarily use memory attached to the processing system (32-bit wide DDR3), while the VectorBlox MXPs primarily uses the faster memory attached to programmable logic (64-bit DDR3) for weights and activations. Example detections produced when running Tiny YOLOv2 VOC are shown in Figure 4.3.

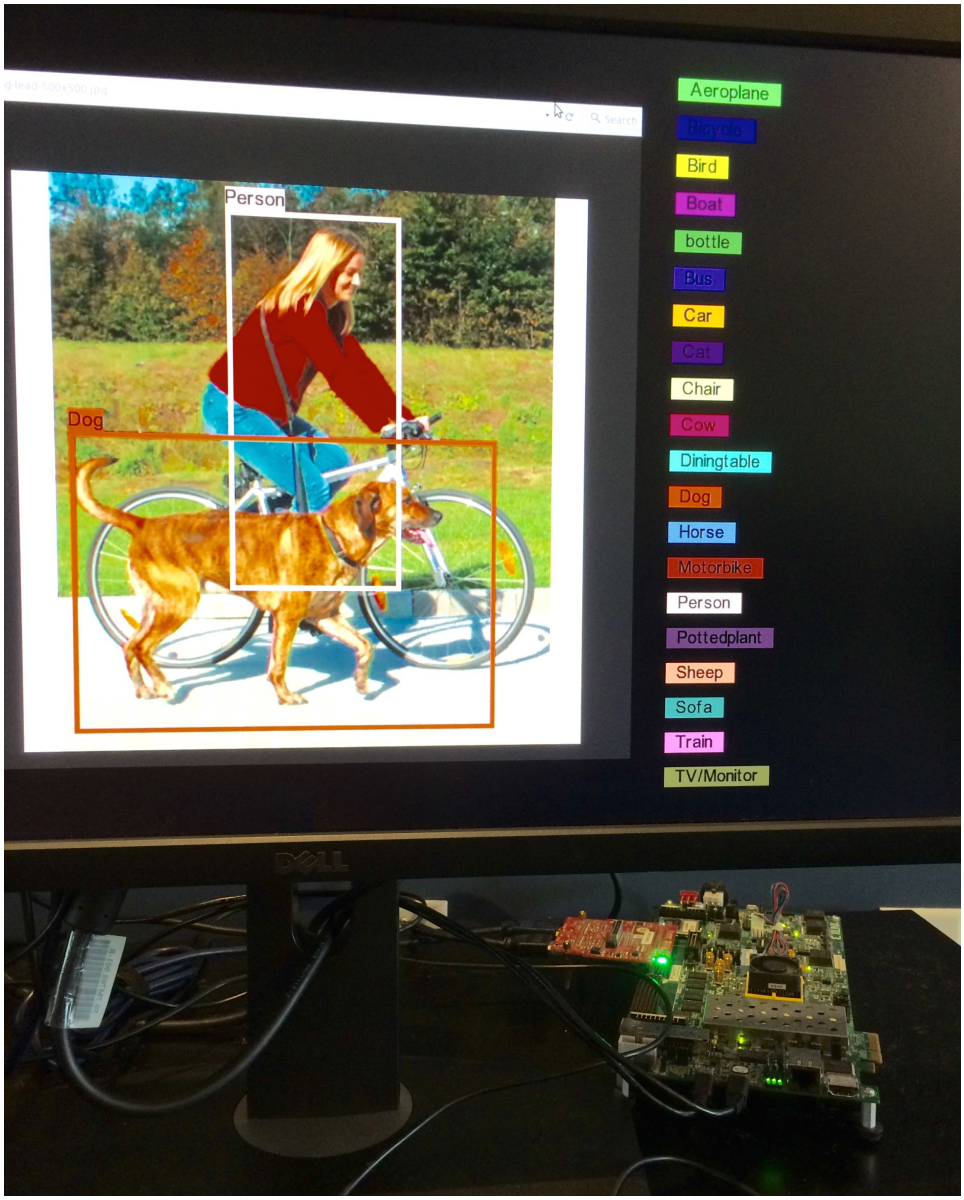


Figure 4.3: Example YOLO detection

4.5.2 Overlay Instances

Increasingly performant CNN implementations will be discussed below:

1. 32-bit floating-point host baseline, single-core (ARM Cortex-A9)
2. 16-bit fixed-point soft vector overlay, single-core (ARM Cortex-A9)
3. 8-bit fixed-point custom vector overlay, single-core (ARM Cortex-A9)
4. 8-bit fixed-point custom vector overlay, dual-core (ARM Cortex-A9 + MicroBlaze)

This overlay instantiates multiple cores. The first host processor (ARM Cortex-A9) is used to control any slave processors (MicroBlaze). Each processor has a vector engine attached. With two cores running at 116 MHz, each capable of 1859 operations per cycle, the overlay has a theoretical peak throughput of 431.3 GOPS.

To minimize latency of inference, the processing of each layer in the network is split between the two cores. This is accomplished by generating half the output maps on each core, with both cores using the same input maps in shared memory. Using simple message passing, the host starts the slave core to generate the slave's outputs of the current layer before beginning its own. Upon finishing, the host blocks until the slave is finished, and then moves on to the next layer. Workloads are relatively balanced and can continue to be divided as more cores are added. Multi-core scaling is discussed below.

4.5.3 Performance

A layer-by-layer runtime breakdown for Tiny YOLOv2 VOC is presented in Table 4.4. The soft vector overlay provides a significant performance increase, with

Type	Filters	ARM baseline	16-bit soft vector		8-bit custom vector	
			single core	dual core	single core	dual core
Conv	16	1294.37	46.42	23.37	8.86	4.67
Conv	32	3224.92	115.14	57.68	5.71	2.99
Conv	64	3170.09	122.02	61.08	4.18	2.18
Conv	128	3048.22	122.24	61.19	3.54	1.83
Conv	256	2988.95	156.99	78.53	3.49	1.77
Conv	512	3038.33	626.48	313.38	4.53	2.28
Conv	1024	12163.71	2502.87	1251.88	15.18	7.61
Conv	1024	24505.00	5004.52	2503.17	29.17	14.61
Conv	125	653.36	615.83	308.04	3.61	1.86
Total		54087.00 (1×)	9313.4 (5.8×)	4658.9 (11.6×)	78.3 (690.8×)	39.8 (1359.0×)

Table 4.4: Tiny YOLOv2 VOC runtime breakdown (ms)

Type	Filters	GOP/Layer	8-bit custom vector	8-bit custom vector (dual)	Scaling factor
Conv	16	0.15	16.9	32.0	1.89
Conv	32	0.40	69.7	133.2	1.91
Conv	64	0.40	95.2	182.5	1.92
Conv	128	0.40	112.6	217.2	1.93
Conv	256	0.40	114.1	224.8	1.97
Conv	512	0.40	88.0	174.7	1.99
Conv	1024	1.59	105.0	209.5	1.99
Conv	1024	3.19	109.3	218.2	2.00
Conv	125	0.04	12.0	23.3	1.94
Total		6.97	89.3	175.0	1.97

Table 4.5: Tiny YOLOv2 VOC throughput breakdown (GOPS)

the dual-core implementation speeding up the baseline by 11.6×. However, not until the CVI is added, does the overlay achieve real-time performance. The dual-core custom vector overlay provides a speedup of 1359.0× over the hard ARM baseline, an additional 117.2× over the soft vector overlay.

Tiny YOLOv2 VOC and Tiny YOLOv2 COCO networks require 6.97 and 7.07 GOPS, respectively. Throughput for Tiny YOLOv2 VOC is presented in Table 4.5, showing GOPS on a layer-by-layer basis for both single-core and dual-core overlays. Nearly perfect scaling can be seen when moving to the dual-core overlay and scaling is expected to continue as cores are added. The layer with the high-

Type	Filters	ARM baseline	16-bit soft vector		8-bit custom vector	
			single core	dual core	single core	dual core
Conv	64	1442.21	52.40	26.35	11.22	5.86
Conv	64	29597.55	1009.19	504.78	34.94	17.73
Conv	128	14658.75	539.57	269.93	17.79	9.08
Conv	128	29302.65	1076.98	538.64	30.99	15.72
Conv	256	14250.50	540.73	270.49	16.15	8.16
Conv	256	28504.01	1080.06	540.16	30.34	15.27
Conv	256	28510.84	1080.37	540.30	30.12	15.16
Conv	512	13977.83	626.55	313.45	18.21	9.13
Conv	512	28070.75	1252.16	626.32	35.20	17.64
Conv	512	28074.34	1252.58	626.50	35.19	17.62
Conv	512	7003.18	1251.45	625.99	12.75	6.39
Conv	512	7003.12	1251.41	626.00	12.75	6.39
Conv	512	7004.00	1251.84	626.18	12.62	6.32
FC	500	164.43	7.20	3.68	7.20	3.69
FC	4096	26.36	3.80	2.03	3.80	2.02
FC	4096	226.39	11.85	6.05	11.88	6.06
FC	1000	55.25	2.89	1.51	2.90	1.51
Total		237872.23 (1×)	12291.6 (19.4×)	6148.8 (38.7×)	324.1 (733.9×)	163.8 (1452.2×)

Table 4.6: VGG16 SVD500 runtime breakdown (ms)

est throughput reaches 224.8 GOPS. The last convolutional layer is 1×1, so the convolution CVI, designed to accelerate 3×3 kernels, is not fully utilized.

Including pre- and post-processing, the Tiny YOLO VOC network takes 50.3 ms to run, while the Tiny YOLO COCO takes 68.3 ms. The networks are identical except for the size of the last layer. The networks require an additional 6-15 ms for post-processing which is excluded from the reported speeds. As this now is a significant part of the computation time, it may require vectorization in the future.

The VGG16 SVD500 network takes 164.3 ms to run. A breakdown of runtime is shown in Table 4.6. The throughput breakdown is shown in Table 4.7, with the highest layer throughput reaching 243.9 GOPS. The fully-connected layers, which are bandwidth-limited in the dual-core system, reach 6.8 GOPS. Post-processing is greatly simplified, only requiring a final sort of the 1000-entry output.

We also compared performance to a desktop-based solution. Table 4.8 shows

Type	Filters	GOP/Layer	8-bit custom vector	8-bit custom vector (dual)	Scaling factor
Conv	64	0.17	15.4	29.6	1.91
Conv	64	3.70	105.9	208.6	1.97
Conv	128	1.85	104.0	203.5	1.96
Conv	128	3.70	119.4	235.2	1.97
Conv	256	1.85	114.5	226.5	1.98
Conv	256	3.70	121.9	242.2	1.99
Conv	256	3.70	122.8	243.9	1.99
Conv	512	1.85	101.5	202.4	1.99
Conv	512	3.70	105.1	209.7	2.00
Conv	512	3.70	105.1	210.0	2.00
Conv	512	0.92	72.5	144.6	1.99
Conv	512	0.92	72.5	144.6	1.99
Conv	512	0.92	73.3	146.2	2.00
FC	500	0.03	3.5	6.8	1.95
FC	4096	0.04	1.1	2.0	1.88
FC	4096	0.03	2.8	5.5	1.96
FC	1000	0.01	2.9	5.6	1.92
Total		30.76	88.8	176.3	1.98

Table 4.7: VGG16 SVD500 throughput breakdown (GOP/s)

the runtime for the networks tested, including post-processing, compared to the runtime of the original Darknet framework, for both a NVIDIA 1080 GPU, and running on a single-core of a 4.00 GHz Intel i7-4790k CPU. The overlay is about 20 times faster than the CPU, but is still nearly an order of magnitude slower than the GPU when running Tiny YOLOv2. This gap is expected to narrow if a larger FPGA is targeted. Note that though inference on both Tiny YOLOv2 networks were greatly sped up when NVIDIA's cuDNN library was used, VGG saw little improvement, leading to our implementation running slightly faster.

4.5.4 Area

The area required for single- and dual-core overlay is reported in Table 4.9. The designs use the 28 nm 7Z045-2, and met timing at 116 MHz. In the dual-core design Block Random Access Memory (BRAM) becomes the most utilized resource,

Network	Darknet GPU	Darknet CPU	Dual-core custom vector overlay
Tiny YOLOv2 VOC			
ms	4.6	992.0	41.0
fps	218.1	1.0	24.4
Tiny YOLOv2 COCO			
ms	4.6	1186.3	42.5
fps	216.5	1.2	23.5
VGG16 SVD500			
ms	230.2	4977.8	164.3
fps	4.34	0.2	4.9

Table 4.8: Comparing inference speed to the Darknet framework

Table 4.9: Resource Usage

Fully Implemented System (Zynq-7000, 28nm, 8-bit fixed-point)					
	Logic 6-LUTs	64b LUT RAMs	FFs	BRAM (36kB)	DSP
7Z045 Device	218,600	70,400	437,200	545	900
VectorBlox MXP (V16)	29712	1053	25602	72	80
MicroBlaze	4490	926	3025	19	5
CNN CVI (13 super-kernels)	13897	0	24886	12	144
Per-core Interconnect/Peripherals	6710	205	10590	4	0
1-Core (MXP+CVI+Interconnect)	54809	2184	64103	107	229
1-Core Device Utilization	25.1%	3.1%	14.7%	19.6%	25.4%
2-Cores (1 using ARM)	98485	3234	114300	191	453
AXI Fabric	13314	1332	28412	32	0
MIG Memory Controller	10408	2234	9231	1	0
Video etc.	5834	370	9845	33	8
Complete System	128041	7170	161788	257	461
Device Utilization	58.6%	10.2%	37.0%	47.2%	51.2%

as they are used by both the MXP and the MicroBlaze. Optimizing the memory footprint of each host, and using smaller scratchpad sizes, is needed to fit more cores on this device.

4.6 Previous Work

Accelerating training and inference of CNNs has been an active research area. Fast inference of deep networks using FPGAs is important both in the data center and in the field.

Early work by Farabet [14], accelerated the face detection using a 5 layer CNN network, consisting of 3 convolutional layers and 2 fully connected layers. The design follows a flexible, vectorized approach and allows reprogramming of software without reconfiguration. Custom hardware is designed to accelerate each layer type. It was implemented on a Virtex-4 SX35 and ran close to around 4 GOPS. This design used 16-bit quantization.

More recently, Zhang et al. [35] implemented a convolution accelerator on Xilinx Virtex7 485T, at a much higher performance of 62.62 GFLOPS. The synthesized design ran at 100 MHz using Vivado HLS. Only the convolution computation was accelerated, ignoring the pointwise operations and pooling layer. Full 32-floating point operands were used, and design consumed 2240 DSP blocks.

Qiu et al. [26] implemented a CNN engine, which ran a modified VGG16 network, with reduced fully-connected layers. The design used the same development board as our system, a Xilinx ZC706, and ran at 116 MHz. Quantization uses 16 bits. Convolutional layers were measured at 187.5 giga operations per second (GOPS), while fully connected layers measured 1.2 GOPS. For the YOLO networks, where nearly all computation was in the convolutional layers, they measured 137.5 GOPS, using 90% of the DSP blocks in the design.

Most recently, Ayondat et al. [3] implemented a deep learning accelerator using OpenCL. It targeted the Intel Arria 10 1150 FPGA. The authors benchmarked their design on AlexNet, which contains CNN kernels greater than 3×3. The Winograd transform is used to boost performance of the of their design, which significantly speeds larger kernels. Using the Arria 10, which is a much faster chip than ours, the design runs at 303 MHz, nearly $3\times$ the clock speed of our current design.

This puts the throughput of this custom overlay above the performance of

	[14]	[35]	[26]	[3]	Ours
Operands	18-bit fixed	32-bit float	16-bit fixed	16-bit float	8-bit + 12-bit fixed
FPGA	Virtex-4	Virtex7 VX485T	7Z045	Arria 10	7Z045
MHz	200	100	150	303	116
LUT	-	186251	182616	246k alms 681K reg	128041
DSP	192	2240	780	1476	461
GOPS	4	61.62	137.5	1382	175.0
GOPS/GLUT	-	0.33	0.75	5.6	1.3
GOPS/DSP	0.021	0.028	0.176	0.936	0.361

Table 4.10: Previous work comparison

Zhang et al. and Qui et al. yet significantly below the performance of Aydonat et al.. Moving to a larger, newer FPGA, scaling up the overlay, and increasing the number of DSP blocks can increase performance. However, to obtain state-of-the-art performance per unit area, clock speed improvements are also necessary.

4.7 Design Complexity

This custom vector overlay only requires a single CVI as the vast majority of the operations are in the convolution kernel. The soft vector processor is capable of accelerating the remaining operations, including pointwise activation functions and pooling. The convolution CVI is designed only for 3×3 kernels and was developed fairly rapidly. The instruction requires 1300 lines of RTL, as it contains optimizations to maintain precision and high throughput. The remainder of the overlay consists of C code totaling approximately 4500 lines of software.

4.8 Summary

The CNN custom vector overlay is a performant, flexible solution for accelerating inference on FPGAs. With the fastest layer achieving 243.9 GOPS, the design is competetive with the state-of-the-art on FPGAs. Further refinement to the CVI and

scaling to larger FPGAs will help close the gap with GPUs.

As the overlay is software-driven, the addition of other network layers can be added rapidly, making it a useful tool for the fast-moving field of deep learning. The custom vector overlay approach minimizes the use of custom RTL, to achieve results competetive with custom hardware solutions.

Chapter 5

Binary-weight Neural Network Custom Vector Overlay

Reduced-precision arithmetic improves the size, cost, power and performance of neural networks in digital logic. CNNs using 1-bit weights can achieve state-of-the-art error rates while eliminating costly multiplications, reducing memory bandwidth and improving power efficiency. The BinaryConnect binary-weight network, for example, achieves 9.9% error using floating-point activations on the CIFAR-10 dataset.

In this chapter, we implement a lightweight vector overlay for accelerating inference computations with 1-bit weights and 8-bit activations. The entire overlay is very small, using about 5000 4-input LUTs, and fits into a low cost iCE40 UltraPlus FPGA from Lattice Semiconductor. Beyond the base vector processor, the overlay only requires 200 additional lines of custom RTL, keeping the majority of development effort in software. To show small networks can be useful, we run

two classification networks produced by shrinking the original BinaryConnect network. The first is a 10-category classifier with a 89% smaller network that runs in 1315 ms and achieves 13.6% error. The second is an even smaller, single category classifier that runs in 195 ms, and has only 0.4% error. In both classifiers, the error can be attributed entirely to training and network size, not to the reduction in precision of activations.

5.1 Approach

Deep CNNs provide increasingly accurate solutions but require an increasing number of MAC operations. Large networks can contain tens of GOPS on a single inference pass. Not all neural networks are as expensive to run, however, and advances for deep learning can benefit even small devices. Binary versions of CNNs allow deep learning networks to be deployed on small, low-cost FPGA chips, or to maximize the operations per second on a larger chip.

Using our software-based approach, we develop a lightweight, custom vector overlay for accelerating networks with binary weights. We target a low-cost iCE40 UltraPlus FPGA, and use a soft RISC-V processor as the host. The RISC-V processor contains lightweight vector instructions called LVE, which accelerate the overlay. In this chapter we make three contributions. First, we optimize the BinaryConnect system by reducing the network size and computing precision. Second, for performance, we implement a CVI that accelerates binary convolutions for networks with binary weights and 8-bit inputs, requiring only 200 lines of custom RTL. Third, we produce a lightweight custom overlay for accelerating binary networks, capable of targeting the small FPGAs.

5.2 Adaptation

In this section, we examine and adapt networks that allow deep-learning to be utilized in small, highly-constrained devices. First, we observe the research showing networks reduced to the extreme can produce state-of-the-art results. We take the work of BinaryConnect [8], where weights contain only binary values, to produce networks for a lightweight overlay. Next, we verify fixed-point quantization of the remaining calculations does not lead to loss in accuracy.

As in the previous chapter, the core compute kernel is a 2D convolution (again 3×3). This computation is repeated at every position in the padded input map, producing an output map of the same size. These outputs become inputs for the next layer, and the process is repeated. In this chapter, however, we explore networks with weights fixed to ± 1. Using the training method described in BinaryConnect, 1-bit weights are used to represent ± 1. This saves memory storage and bandwidth, and replaces all multiplications with addition and subtraction. Although using binary weights, BinaryConnect achieves 9.9% state-of-the-art error rates on CIFAR-10 [20]. Binary and ternary weights (where 0 is also allowed) allow FPGAs to exploit bit-level parallelism. We further save area and energy by processing fixed-point activations throughout the network. Importantly, our reduced fixed-point quantization does not introduce any further error.

Two networks are explored and adapted in this chapter; a downscaled version of BinaryConnect's CIFAR-10 network, and a custom, single category classifier, intended to be used in a low-power, always-on sensor handling sleep/wake events for a larger electronic device. The first network is larger and requires overlapping computation with DMA transfers of weights from the SPI flash ROM, while the

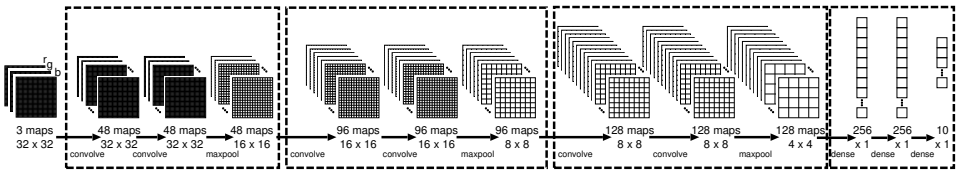


Figure 5.1: Reduced binary CNN containing 89% fewer operations than BinaryConnect

second can fit all parameters in the scratchpad, avoiding transfers beyond the initial bootup.

5.2.1 Network Reduction

We start by optimizing the BinaryConnect system by reducing both the network size and computed precision described below. We reduced the network size from:

$(2\times128C3)$-MP2-$(2\times256C3)$-MP2-$(2\times512C3)$-MP2-$(2\times1024\text{FC})$-10SVM

to:

$(2\times48C3)$-MP2-$(2\times96C3)$-MP2-$(2\times128C3)$-MP2-$(2\times256\text{FC})$-10SVM

where $C3$ is a 3×3 ReLU convolution layer, MP2 is a 2×2 max-pooling layer, FC is a fully connected layer, and SVM is a L2-SVM output layer.

This new network, shown in Figure 5.1, contains 89% fewer operations than the BinaryConnect reproduction and achieves 11.8% error on CIFAR-10. For performance, we also dropped ZCA whitening, increasing error to 13.6%.

All computations are converted to fixed-point, creating both an initial version using 32-bit values throughout, as well as a reduced-precision version. The reduced-precision network's inputs and activations use 8-bit unsigned integers and the intermediate sums use 16-bit and 32-bit signed integers. Importantly, the reduced-precision network maintains the same error rate of 13.6%.

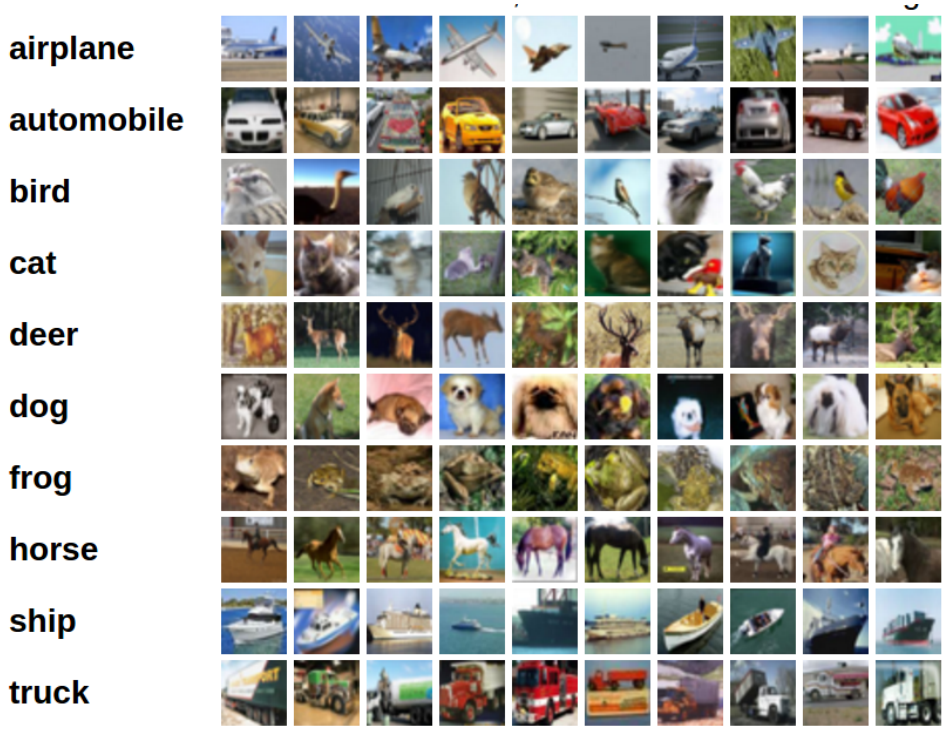


Figure 5.2: Samples of CIFAR-10 dataset

For a the second network, we start with the CIFAR-10 dataset, shown in Figure 5.2, but modify it slightly by replacing the 'deer' category with 'people' superclass from CIFAR-100. The 'people' images were duplicated to match the number of images in each of the CIFAR-10 categories. This change allows the network to function as a "person" detector.

Sample results are shown in Figure 5.3. The two bars for each category show the different scores between floating-point computation (lighter) and our 8-bit fixed-point computation (darker). A more positive score is a higher confidence classification.

After this initial network is successfully running on our overlay, we modify the

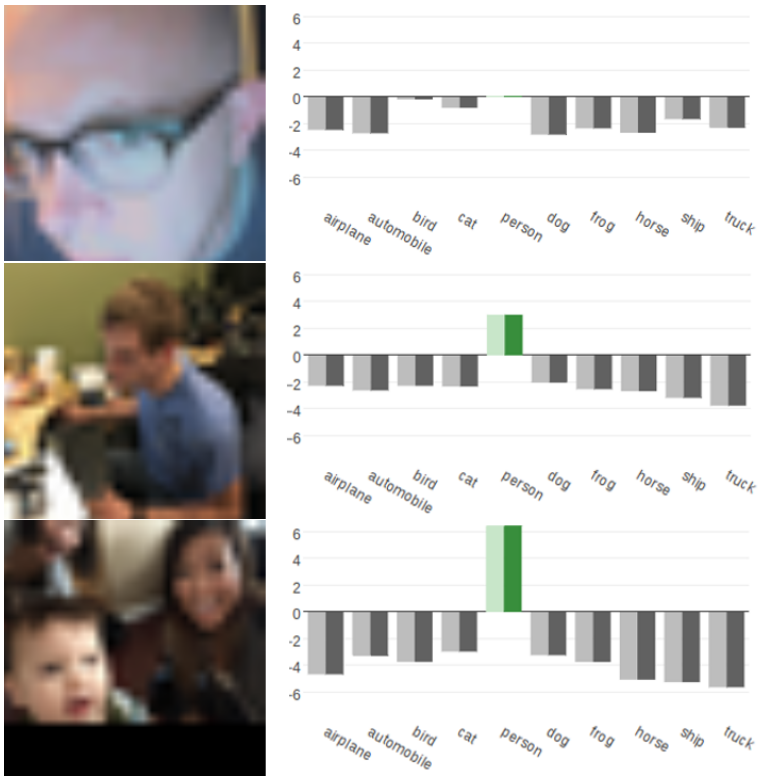


Figure 5.3: Person detector, sample results

second network to become a binary classifier, targeting only the "person" category. The network was shrunk in size and retrained on a larger database. As expected, a higher accuracy for this simpler task is achieved, with less than 1% error.

5.3 Vectorization

With the two networks, we start by vectorizing a 32-bit fixed-point scalar implementation. The soft vector overlay consists of an ORCA [24] soft RISC-V processor augmented with Lightweight Vector Extensions (LVE) [23] [1]. Like VectorBlox

[1] ORCA implements a pipelined RV32IM instruction set. LVE is a proprietary extension that differs from the proposed RISC-V vector extension, which is not yet finalized.

MXP, LVE enables efficient vector and matrix operations without any loop, memory access, or address generation overhead. LVE streams data through the RISC-V ALU, so subword operations are unavailable. LVE does allow CVIs to be inserted.

After completing this initial vectorization we see a speedup of 7.43× over the RISC-V scalar implementation, speeding inference from 93233 ms to 12543 . Profiling confirms the convolution layers consume the vast majority of the runtime. The fully-connected layers take only a small percentage of the runtime and do not require further acceleration.

5.4 Custom Vector Instructions

To accelerate the overlay further, with convolution operations consuming the majority of the runtime, a CVI is used. The binary convolution CVI is discussed below. In addition to this instruction, two minor CVIs are added to allow certain subword SIMD processing: a quad 16-bit to 32-bit SIMD add, and a 32-bit to 8-bit activation function. These two custom instructions allow us to maintain performance while avoiding overflows by accumulating the 16-bit convolution outputs into 32-bit sums, before ultimately producing 8-bit activations. As these latter two CVIs are straightforward, they aren't further discussed, but do count towards custom lines of RTL.

5.4.1 3x3 Binary Convolution Instruction

The convolution CVI accelerates CNNs with binary weights and 8-bit inputs. The instruction computes two overlapping convolutions in parallel. In use, input data is fetched down a column, accepting 8 consecutive bytes each cycle as its two 32-bit operands. Two passes of the same input data are required to compute four columns

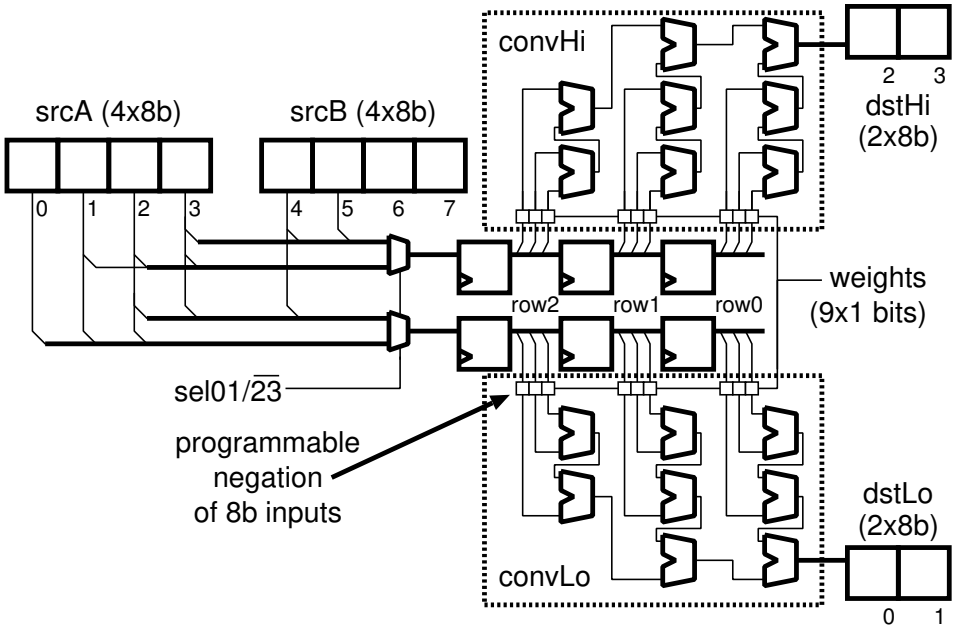


Figure 5.4: Binary convolution custom vector instruction

of 16-bit convolutions from all four possible byte offsets within a word. After that, the inputs advance by 4 bytes to maintain alignment. A diagram of the CVI is shown in Figure 5.4.

5.5 Results

Results for the performance and area of the lightweight custom vector overlay are presented in this section.

5.5.1 Experimental Setup

The entire system, shown in Figure 5.5, was built around a Lattice iCE40 UltraPlus Mobile Development Platform (MDP). Unlike the previous development board, there is no DRAM, and SPI Flash ROM is used to store the binary weights. In

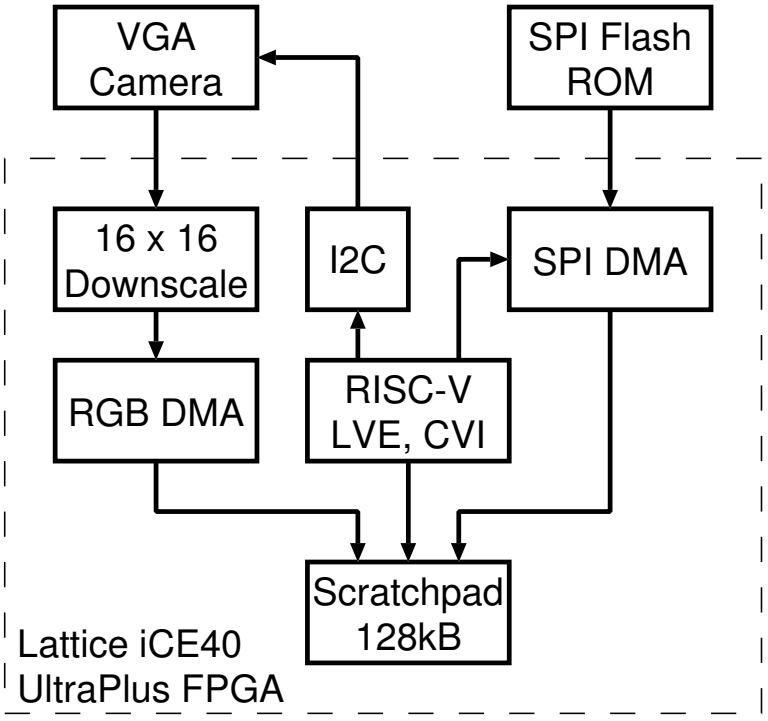


Figure 5.5: System diagram

this minimal design, the input and output maps must remain entirely within the 128 kB scratchpad memory. A lightweight port of VectorBlox's vector processor, titled LVE, is used in place of MXP. Like MXP, LVE instructions operate on data in a dedicated scratchpad, which is single-ported Random Access Memory (RAM). The scratchpad operates at 72 MHz to provide two read and one write every 24 MHz CPU clock. Operating concurrently with the CPU, a DMA engine transfers multiple 32-bit values from the SPI Flash ROM, which stores the binary weights (about 270 kb), into the scratchpad. A VGA-resolution RGB camera (640×480 pixels) using RGB565 is downscaled to 40×30 pixels in hardware, and uses DMA to write 32-bit aligned RGBA888 pixels into the scratchpad. Software

Network	Lasagne CPU	custom vector overlay
Reduced CIFAR10		
ms	6.4	1315
fps	156.3	0.8
Reduced Person classifier		
ms	2.0	195
fps	500.0	5.1

Table 5.1: Runtime of reduced networks, desktop vs custom overlay

de-interleaves the RGBA888 pixels into separate 8-bit R, G, and B pixel planes, where the algorthm uses a 32×32 centred and padded region of interest.

5.5.2 Performance

On the Mobile Development Platform, inference for the CIFAR-10 network takes 1315 ms. The accelerator improves RISC-V runtime of convolution layers $73\times$, and LVE improves runtime of fully-connected layers $8.3\times$, for an overall speedup of $70.9\times$. In comparison, a 4.00 GHz Intel i7-4790k desktop, using Python/Lasagne, the network takes 6.4 ms.

The reduced, binary classifier runs in 195 ms (2.0 ms on the i7-4790k). The complete system, including camera overhead, runs at 230 ms per frame.

5.5.3 Area

Resource utilization for the Lattice iCE40 UltaPlus-5K FPGA can be seen in Table 5.2. The RISC-V CPU runs at 24 MHz. The whole system uses 5036 of 5280 4-input logic cells, 4 of 8 (16-bit) DSP blocks, all thirty 4 kb (0.5 kB) BRAM, and all four 32 kB single-ported RAM.

Table 5.2: Resource Usage

Component	Logic 4-LUTs	BRAM (0.5 kB)	SPRAM 32 kB	DSP
iCE40 UltraPlus				
Totals	5,280	30	4	8
Binary-weight CNN overlay				
Complete System	5,036	30	4	4
Device Utilization	95.4%	100.%	100.0%	50.0%

5.6 Previous Work

Research into training binary CNNs is still fairly recent, however, several custom hardware inference engines have already been developed. Unlike current GPUs, these custom hardware solutions can fully exploit the reduced-bit computation, and avoid heavy usage of resource-constrained DSP blocks.

In 2016, Renzo et al. produced YodaNN [2], a low-power ASIC for accelerating binary-weight neural networks. Their design is capable of 1.5 tera operations per second (TOPS) at 1.2 V. The maximum performance seen on binary versions of several popular networks is 525.5 GOPS.

Zhao et al. [36] synthesized a binary-weight accelerator on a 7Z020 FPGA, targeting inference of CIFAR-10 networks. The design was capable of 207.8 GOPS across layers.

Umurolglu et al. [33] produced a binary neural network framework using 7Z045 FPGA, targeting several small networks, including CIFAR-10 and SVHN. The design used 1 bit inputs and activations, not just binary weights. The network uses XNOR popcount for accumulations. A maximum throughput of 11.6 TOPS was reported for a fixed 3×512 fully connected topology. Fraser et al. [15] expanded upon this work, and targeted larger networks on a larger, KU115 FPGA. The re-

ported a maximum throughput of 14.8 TOPS.

5.7 Design Complexity

The majority of the code is accelerated by lightweight vector instructions. As Lightweight Vector Extensions (LVE) does not support subword SIMD, two basic operations to operate on halfwords and bytes are added as well as the important binary convolution instruction. These small CVIs, adding subword support, required a total of 100 lines of RTL. These instruction replace approximately 10 lines of the original scalar code.

The CVI used to accelerate the binary-weight convolution kernels requires 100 lines of RTL. This hardware replaces approximately 6 lines of the original scalar code. The entire design requires a total of 700 lines of C code.

5.8 Summary

This custom vector overlay demonstrates that our approach scales down to the smallest FPGAs. Using an embedded RISC-V processor with lightweight vector extensions, we create a minimal soft vector overlay. Adding the binary-weight convolution CVI allows our design to run 70.9$\times$ faster. The design runs in real-time, despite the restricted resources available while only requiring an addition 200 lines of RTL.

Chapter 6

Conclusions

This thesis use three case studies to explore a hardware/software approach for accelerating computer vision algorithms using custom vector overlays. The algorithms are implemented with a software-driven approach and achieve performance comparable with full hardware solutions. These overlays only require a minimum of custom RTL, keeping the development effort on software. Our approach works both on traditional vision algorithms using AdaBoost cascades and state-of-the-art neural networks.

6.1 Summary

A summary of results for the three case studies is shown in Table 6.1.

In the first case study, accelerating LBP face-detection, the custom vector overlay achieves a speedup of 248.4× over an ARM Cortex-A9 software implementation. This gain can be decomposed into software and hardware contributions. Software-only vectorization using the general purpose soft vector overlay provides a 25× speedup, using the 16 parallel 32-bit ALUs. An additional 10× speedup

Algorithm	LBP	CNN	BNN
Cores	1	2	1
Host Processor	Hard ARM A9	Hard ARM A9	Soft RISC-V
Vector Processor	VectorBlox MXP	VectorBlox MXP	VectorBlox LVE
Vector Lanes	16	16	1
CVIs	2	1	2
Lines of RTL	800	1300	200
Lines of software	2500	4500	700
Vector Speedup	24.9×	11.6×	7.43×
CVI Speedup	248.4×	1359.0×	70.9×
CVI Speedup over Vector	10×	117×	9.6×

Table 6.1: Comparing the various overlays explored in this thesis

is achieved by customizing the vector overlay, implementing two custom vector instructions (CVIs): one for quickly computing the 8-way comparison, producing the 8-bit LBP, and another for the table-lookup operation.

The software-driven LBP-based face detection system runs 2.4× to 3.5× faster than previously published face detection systems implemented purely in hardware. Furthermore, the custom vector overlay only needs about 800 lines of custom RTL, keeping the majority of the development effort on software. This approach permits easy integration of additional processing, where the same overlay can be used.

A novel contribution made in this case study is the ILP formulation used to replace 32-bit floating-point or fixed-point computation with 8-bit integers. This simplifies hardware, produces a speedup, and provides an accuracy guarantee. We also show how restricting block sizes to be square powers of two allows the LBP computation to be performed in an efficient pre-computation step. Although this has been reported before on fixed-width SIMD systems [4], we found that it works for variable-length vector systems as well.

The second case study implements a CNN overlay running VGG and two networks based on YOLO [28]. The software-programmable vector overlay enables

rapid algorithm development, an important property for this fast-moving research field. Only a single CVI, requiring 1300 lines of RTL, was needed to create a custom vector overlay that supports these CNNs. The multi-core custom overlay is 1359.0× faster than the hard ARM core. The performance of the overlay is potentially competitive with top-of-the-line GPUs; this work achieved hundreds of GOPS on a relatively small FPGA. We found that using 8-bit precision is essential for performance and allows for maximal use of the DSP blocks as long as we used 12-bit weights. The CVI contributed to a 145× speedup over the base soft vector overlay.

The final case study implements a lightweight, binary-weight neural network overlay to address the heavy DSP usage in the previous example. We use a very low-cost FPGA and show the effectiveness of the design using several networks based on of BinaryConnect. The overlay is very small – it uses about 5000 4-input LUTs. Only 200 lines of custom RTL were required for this design. The accelerator improves ORCA RISC-V runtime of convolution layers 73×, and LVE improves runtime of dense layers 8×, for an overall speedup of 71×. The CVI contributed to a 9.6× speedup over the base LVE. The 1-category classifier runs in 195 ms(2.0 ms on the i7-4790k) with 0.48% error and consumes 21.8 mW. A power-optimized version, designed to run at one frame per second, consumes just 4.6 mW. In the classifiers we tested, the error can be attributed entirely to training and not reduced precision.

6.2 Limitations

We have only assessed our algorithmic changes using the cascades and networks discussed. For generality, we should further validate with cascades trained to detect

other objects and explore further networks in detail.

6.3 Future Work

Many of our optimizations may be applicable to GPUs or other SIMD systems as well, which would be useful to apply.

One obvious way to improve performance of our approach is to build larger, multi-core systems. Both the traditional Viola-Jones algorithm and neural networks can be divided efficiently between cores. We have shown this with our dual-core CNN overlay with near perfect scaling. Vector overlays can be parameterizable both in terms of cores and vector lanes, allowing any sized system to be targeted.

Beyond scaling to many core systems, we would like to expand the range of neural networks that can be run on the overlays. An overlay represents a practical way to accelerate other layers and incorporate the latest algorithmic advances.

To make our overlay useful for many applications, an automated tool, targeting one of the popular deep learning frameworks, should take an arbitrary network and translate it to run on our overlay.

Finally, accelerating the training step, instead of only inference would be interesting in domains that would benefit from low-latency, online training.

Bibliography

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016. → pages 49

[2] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. In *VLSI (ISVLSI), 2016 IEEE Computer Society Annual Symposium on*, pages 236–241. IEEE, 2016. → pages 78

[3] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu. An OpenCL deep learning accelerator on Arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 55–64. ACM, 2017. → pages 65, 66

[4] O. Bilaniuk, E. Fazl-Ersi, R. Laganiere, C. Xu, D. Laroche, and C. Moulder. Fast LBP face detection on low-power SIMD architectures. In *IEEE Computer Vision and Pattern Recognition Workshops*, pages 630–636, 2014. → pages 26, 28, 43, 81

[5] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. → pages 11

[6] B. Brousseau and J. Rose. An energy-efficient, fast FPGA hardware architecture for OpenCV-compatible object detection. In *ICFPT*, pages 166–173, 2012. → pages 3, 31, 39, 40, 42, 43

[7] J. Cho, B. Benson, S. Mirzaei, and R. Kastner. Parallelized architecture of multiple classifiers for face detection. In *IEEE ASAP*, pages 75–82, 2009. → pages 3, 42, 43

[8] M. Courbariaux, Y. Bengio, and J.-P. David. BinaryConnect: Training deep neural networks with binary weights during propagations. In *Advances in*

Neural Information Processing Systems, pages 3123–3131, 2015. → pages 6, 17, 70

[9] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016. → pages 17

[10] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Alg. for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. → pages 29

[11] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009. → pages 15

[12] J. Edwards and G. G. Lemieux. Real-time object detection in software with custom vector instructions and algorithm changes. In *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*, pages 75–82. IEEE, 2017. → pages iv

[13] J. Edwards, J. Vandergriendt, A. Severance, A. Raouf, T. Watzka, S. Singh, and G. G. Lemieux. Tinbinn: Tiny binarized neural network overlay in less than 5,000 4-luts. In *Proceedings of the 3rd International Workshop on Overlay Architectures for FPGAs (OLAF 2017)*, pages 23–25, 2017. → pages iv

[14] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pages 32–37. IEEE, 2009. → pages 49, 65, 66

[15] N. J. Fraser, Y. Umuroglu, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Scaling binarized neural networks on reconfigurable logic. In *Proceedings of the 8th Workshop and 6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, pages 25–30. ACM, 2017. → pages 78

[16] C. Gao and S.-L. Lu. Novel FPGA based Haar classifier face detection algorithm acceleration. In *FPL*, pages 373–378, 2008. → pages 3, 42, 43

[17] B. Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014. → pages 15

[18] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan. Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746, 2015. → pages 49

[19] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016. → pages 49

[20] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. Univ. of Toronto, 2009. → pages 6, 15, 70

[21] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. → pages 15

[22] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998. → pages 15

[23] G. Lemieux and J. Vandergriendt. FPGA-optimized lightweight vector extensions for VectorBlox ORCA RISC-V. 4^{th} RISC-V Workshop, July 2016. → pages 73

[24] G. Lemieux and J. Vandergriendt. ORCA FPGA-optimized RISC-V. 3^{rd} RISC-V Workshop, January 2016. → pages 73

[25] S. Liao, X. Zhu, Z. Lei, L. Zhang, and S. Z. Li. Learning multi-scale block local binary patterns for face recognition. *Advances in Biometrics*, pages 828–837, 2007. → pages 13

[26] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016. → pages 49, 51, 65, 66

[27] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016. → pages 17, 49

[28] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 779–788, 2016. → pages 4, 81

[29] D. E. Rumelhart, G. E. Hinton, R. J. Williams, et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988. → pages 15

[30] A. Severance and G. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor. In *CODES+ISSS*, pages 1–10, 2013. → pages 18

[31] A. Severance, J. Edwards, H. Omidian, and G. Lemieux. Soft vector processors with streaming pipelines. In *FPGA*, pages 117–126, 2014. → pages 21

[32] A. Severance, J. Edwards, and G. Lemieux. Wavefront skipping using BRAMs for conditional algorithms on vector processors. In *FPGA*, pages 171–180, 2015. → pages 22

[33] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74. ACM, 2017. → pages 78

[34] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Vision and Pattern Recognition*, volume 1, pages 511–518, 2001. → pages 9

[35] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015. → pages 65, 66

[36] R. Zhao, W. Song, W. Zhang, T. Xing, J.-H. Lin, M. B. Srivastava, R. G, depending upon the target frame ratoupta, and Z. Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *FPGA*, pages 15–24, 2017. → pages 78