Fast and Energy-Efficient Monocular Depth Estimation on Embedded Systems

by

Diana Wofk

B.S., Massachusetts Institute of Technology (2018)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2020

Author .....
Department of Electrical Engineering and Computer Science
May 12, 2020

Certified by .....
Vivienne Sze
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Fast and Energy-Efficient Monocular Depth Estimation on Embedded Systems

by

Diana Wofk

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2020, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Depth sensing is critical for many robotic tasks such as localization, mapping and obstacle detection. There has been a growing interest in performing depth estimation from monocular RGB images, due to the relatively low cost and form factor of RGB cameras. However, state-of-the-art depth estimation algorithms are based on fairly large deep neural networks (DNNs) that have high computational complexity and energy consumption. This poses a significant challenge to performing real-time depth estimation on embedded platforms. Our work addresses this problem.

We first present FastDepth, an efficient low-latency encoder-decoder DNN comprised of depthwise separable layers and incorporating skip connections to sharpen depth output. After deployment steps including hardware-specific compilation and network pruning, FastDepth runs at 27−178 fps on the Jetson TX2 CPU/GPU, with total power consumption of 10−12 W. When compared with prior work, FastDepth achieves similar accuracy while running an order of magnitude faster.

We then aim to improve energy-efficiency by deploying FastDepth onto a low-power embedded FPGA. Using an algorithm-hardware co-design approach, we develop an accelerator in conjunction with modifying the FastDepth DNN to be more accelerator-friendly. Our accelerator natively runs depthwise separable layers using a reconfigurable compute core that exploits several types of compute parallelism and toggles between dataflows dedicated to depthwise and pointwise convolutions. We modify the FastDepth DNN by moving skip connections and decomposing larger convolutions in the decoder into smaller ones that better map onto our compute core. This enables a 21% reduction in data movement, while ensuring high spatial utilization of accelerator hardware. On the Ultra96 SoC, our accelerator runs FastDepth layers in 29 ms with a total system power consumption of 6.1 W. When compared to the TX2 CPU, the accelerator achieves 1.5−2× improvement in energy-efficiency.

Thesis Supervisor: Vivienne Sze
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank Professor Vivienne Sze for advising my research. She gave me the opportunity to join her group when I was just a freshman, and she has continuously motivated me to grow as a researcher since then. I have been fortunate to learn a lot from Vivienne in many of her classes, and I am very grateful to have worked on the FastDepth project since its conception. Vivienne's guidance and commitment have been invaluable to its success. I would also like to thank Professor Sertac Karaman for helping supervise the FastDepth project.

I am deeply grateful to Fangchang Ma and Tien-Ju Yang, who collaborated with me on the development of FastDepth and with whom I co-authored my first publication. Their mentorship and assistance have been tremendous sources of knowledge and inspiration. I am also grateful to Yu-Hsin Chen and Amr Suleiman for providing helpful suggestions on hardware design during the second half of my MEng work.

Being a member of the EEMS group has been a memorable experience. Thank you to James Noraky, Nellie Wu, Gladynel Saavedra Pena, Jane Lai, Zhengdong Zhang, Peter Li, Soumya Sudhakar, and Theia Henderson. I have greatly enjoyed our discussions and camaraderie. A special thanks to Mehul Tikekar for mentoring me and nurturing my interest in research during my undergraduate years in the group.

The Undergraduate Office has been a constant source of support. Many thanks to Myriam Berrios, Vera Sayzew, Anne Hunter, Brandi Adams, and Katrina LaCurts. Their encouragement, humor, and willingness to listen brightened up my day whenever I would come visit. Additional thanks to Jessie Stickgold-Sarah from the WRAP Office for guiding me in my early thesis stages and helping me organize my writing.

I would like to sincerely thank my friends at MIT who made my MEng journey full of joy and cherished memories: Anna Sinelnikova, Jing Lin, Allison Lemus, Abigail Katcoff, Cassia Wang, Camila Thanos, Madeleine Waller, among many others!

Lastly, I would like to express my heartfelt gratitude to my dear grandmother for her unconditional support, understanding, and eagerness to help in any way she can. And to my late mother, who always believed in me and sacrificed absolutely everything she had for me to be where I am now. This thesis is dedicated to her.

This work was funded by Analog Devices, Inc., Intel, and the National Science Foundation, Real-Time Machine Learning (RTML) program, grant no. 1937501.

Contents

1 Introduction **25**

1.1 Monocular Depth Estimation 26

1.1.1 Problem Definition 27

1.1.2 Literature Review 27

1.2 Efficient Neural Network Design 32

1.2.1 Overview of Deep Neural Networks 33

1.2.2 Compact Network Architecture Design 36

1.2.3 Network Pruning 40

1.2.4 Network Quantization 41

1.3 Accelerators for Deep Neural Networks 42

1.3.1 CPU and GPU Acceleration 42

1.3.2 FPGA and ASIC Acceleration 44

1.3.3 Neural Network Compilers 45

1.3.4 Dataflow-Based Accelerator Design 46

1.4 Thesis Contributions 50

2 FastDepth, a Compact DNN for Monocular Depth Estimation **53**

2.1 Related Work 54

2.2 FastDepth DNN Architecture 56

2.2.1 Encoder Network 57

2.2.2 Decoder Network 58

2.2.3 Skip Connections 58

2.2.4 Layer Types Used 59

2.3 Training Environment . . . 59
2.4 Post-Training Evaluation and Analysis . . . 60
2.5 Ablation Studies . . . 62
2.5.1 Encoder Design Space . . . 62
2.5.2 Decoder Design Space . . . 63
2.5.3 Skip Connections . . . 67
2.6 Summary . . . 67

3 Real-Time Depth Inference on an Embedded CPU/GPU 69
3.1 Hardware-Specific DNN Compilation . . . 69
3.2 DNN Simplification through Pruning . . . 71
3.3 Post-Compilation Evaluation on the Jetson TX2 . . . 72
3.3.1 TX2 Power Consumption Modes . . . 75
3.4 Live Depth Inference on an Apple iPhone . . . 77
3.5 Summary . . . 79

4 Energy-Efficient Acceleration on an Embedded FPGA 83
4.1 Algorithm-Hardware Co-Design Strategy . . . 84
4.1.1 Design Considerations . . . 84
4.2 Dataflow Design . . . 86
4.2.1 Heterogeneous Dataflow for Depthwise Separable Layers . . . 86
4.2.2 On Serializing vs. Pipelining the Dataflow Design . . . 89
4.3 Accelerator Design . . . 94
4.3.1 Compute Core . . . 95
4.3.2 Network-on-Chip (NoC) . . . 103
4.3.3 On-chip Memory Hierarchy . . . 107
4.3.4 Off-chip Memory Interface . . . 113
4.4 Accelerator-Friendly FastDepth DNN . . . 115
4.4.1 Network Modifications . . . 116
4.4.2 Integer Quantization . . . 127
4.5 Mapping FastDepth onto the Accelerator . . . 133

4.5.1 Tiling Feature Maps 133
4.5.2 Mapping FastDepth Layers 135
4.5.3 Utilization of the PE Array 137
4.6 Implementation Results 142
4.6.1 System Overview 142
4.6.2 Logic Performance Analysis 145
4.6.3 System Performance Analysis 148
4.6.4 External Memory Accesses 151
4.6.5 Challenges 154
4.7 Evaluation of FastDepth on the Ultra96 SoC 159
4.7.1 Against FastDepth on the Jetson TX2 159
4.7.2 Against Other Workloads on the Ultra96 160
4.8 Summary 163

5 Conclusion **165**

5.1 Key Takeaways 167
5.2 Future Work 169

List of Figures

1-1 This thesis studies learning-based monocular depth estimation. Left: a simple visualization of this task, where an input color image is processed to produce an dense depth map containing depth measurements for every pixel in the image. Right: a diagram depicting learning-based depth estimation, where a deep neural network (DNN) is used to predit pixel-wise dense depth from the input image. 27

1-2 Diagram of a convolutional layer. Each of the M filters is first convolved channel-by-channel with the input feature map. The results are then added element-wise to yield a single output channel. This repeats for all M filters, producing M output channels in total. Output channels may be subject to a channel-wide bias that is added after convolution. 35

1-3 A depthwise separable layer factorizes a standard convolution into two smaller ones: a depthwise convolution performing channel-wise convolution, and a pointwise convolution performing element-wise channel aggregation. 37

1-4 Diagram of a depthwise separable layer. This type of layer consists of a depthwise convolution shown in (a) and a pointwise convolution shown in (b). 38

1-5 Diagrams showing different dataflows. Each dataflow variant aims to exploit data reuse of a different datatype. Figures taken from [1]. . . 49

1-6 The row stationary dataflow aims to maximize overall convolutional reuse of all datatypes. Every processing element (PE) operates on one row of filter weights (reused horizontally through the array) and one row of input activations (reused diagonally through the array). Figure taken from [1]. 50

2-1 Examples of two depth estimation DNN structures. 55

2-2 Our FastDepth network architecture. The encoder is shown in blue; decoder is shown in yellow. Dimensions of intermediate feature maps are given as height × width × channels. Arrows from encoding layers to decoding layers denote additive (rather than concatenative) skip connections. 57

2-3 Visual representations of different upsample operations we consider for decoders: (a) UpProj [2], (b) UpConv [2], (c) DeConv5, (d) NNConv5. 64

2-4 Visualized results of depth estimation on the NYU Depth v2 dataset after training. (a) input RGB image; (b) ground truth; (c) our model, FastDepth, without skip connections; (d) our model, FastDepth, with skip connections. 66

3-1 Number of input channels to each layer in our network architecture after pruning. The shaded part represents the architecture before pruning. The very first layer to the network (`mobilenet.0`) is not shown since the channel size of the input fed into the network remains fixed at 3 channels (RGB). 72

3-2 Visualized results of depth estimation on the NYU Depth v2 dataset, now including results from FastDepth after compilation and pruning. (a) input RGB image; (b) ground truth; (c) our model, without skip connections, unpruned; (d) our model, with skip connections, unpruned; (e) our model, with skip connections, pruned; (f) error map between the output of our final pruned model and ground truth, where redder regions indicate higher error. 73

3-3 Accuracy vs. framerate plot comparing FastDepth against prior works. Our network is to the far right of the curve, indicating a better performance tradeoff. 75

3-4 Power consumption over time when running FastDepth inference on the Jeston TX2. Code used to generate power traces sourced from [3]. 76

3-5 Our FastDepth CoreML model running live at 40 fps on iPhone X. Demo video available at `http://fastdepth.mit.edu/`. 78

3-6 Reduction in inference runtime on the TX2 achieved with different steps our approach. Stacked bars represent encoder-decoder breakdown; total runtimes are listed above the bars. The row of δ_1 accuracies listed at the bottom shows the impact of individual steps in our approach on accuracy. Relative to ResNet-50 with UpProj, our final model achieves 65× speedup while maintaining accuracy. 80

4-1 Motivation for an output stationary dataflow. In (a), partial sum outputs are written out to on-chip memory (and potentially to off-chip memory), then read back in so they can be accumulated or updates as the PE continues to compute MACs. In (b), partial sums are held stationary in PE storage until accumulation is done, and the completed partial sum output is written out once. Since data movement across memory levels (PE ↔ on-chip buffer ↔ off-chip DRAM) gets increasingly more expensive, both in terms of latency and energy [1], option (b) is a desirable choice. 88

4-2 Row-stationary dataflow for depthwise convolution. Each processing element (PE), depicted as a gray box, takes a row of input feature map values and a row of depthwise filter weights as input; after convolving the rows, the PE sends its result to the PE below it for spatial accumulation. The bottom-most PEs contain completed results from the 3×3 convolution and can send those out to memory. This dataflow computes up to 7 rows from a single channel of depthwise outputs at once. 90

4-3 Output-stationary dataflow for pointwise convolution. This begins after the row-stationary dataflow has completed *all* depthwise channel outputs. These depthwise outputs are then streamed channel-by-channel back into the processing elements. Each row of PEs in the array receives pointwise weights for all channels from one filter. This allows every PE in the row to complete channel-wise aggregation for a unique row of a single pointwise output channel. Different rows of PEs work on different pointwise output channels. Every PE will hold its row of pointwise partial sums stationary until channel-wise accumulation is complete. This dataflow computes up to 7 rows from 3 channels of pointwise outputs at once. 91

4-4 Partitioning the PE array for serialized vs. pipelined dataflow approaches. A serialized approach requires a reconfigurable PE array that can toggle, e.g., between depthwise and pointwise convolution dataflows. This allows for more flexible allocation of PE resources but increases the complexity of PE design and control logic. A pipelined approach sets aside subsets of the PE array dedicated to each of the pipelined operations, which in this case are the depthwise and pointwise convolutions. This allows for simpler PE designs in each of the sub-arrays; however, since pointwise computations dominate in quantity over depthwise computations, a pipelined approach makes load balancing across dedicated PE arrays more difficult. 93

4-5 Comparing serialized vs. pipelined dataflow approaches. Our analysis shows that a pipelined approach will be at least slightly slower than a serialized approach over a wide range of PE array sizes. This figure plots the speed overhead of a pipelined approach relative to a serialized one, where speeds are proxied by analytically computing clock cycles for all depthwise separable convolutional layers in FastDepth's MobileNet encoder. 94

4-6 High-level accelerator diagram. The innermost module is the processing element (PE) that computes multiply-accumulate operations. PEs are arranged in blocks that compute depthwise and pointwise convolutions. Blocks can be replicated for more compute parallelism. The resulting PE array interfaces with on-chip memory consisting of local PE storage, block-level storage, and larger global buffers (GLBs). . . 95

4-7 Diagram of a processing element. The PE performs multiply-accumulate operations (MACs) and bias addition; it also applies ReLU and quantization functions to values being written out to on-chip buffers. The PE performs row-wise processing (e.g., 1D convolutions of rows, element-wise addition of rows) and stores a row of an input feature map, a filter, and partial psums at a time. Some datapaths and control logic are reconfigurable based on the convolution mode (depthwise vs. pointwise). 97

4-8 Logic breakdown comparing depthwise- and pointwise-specific logic within the PE. Logic here refers primarily to LUTs and registers found in the PE netlist after synthesis. Common logic mostly includes shared registers. Depthwise- and pointwise-specific logic includes counters and control logic for those convolutions. 98

4-9 Row-wise output parallelism in the PE block. Each column of PEs works on a different row of convolution outputs. The number of columns equals the number of rows in an output tile, which is selected to be 7, the greatest common factor of output feature map dimensions in FastDepth layers. 100

4-10 Diagram illustrating how the PE array processes channels during depthwise and pointwise convolution. Shown for a single tile that may be smaller than the input or output feature map (e.g., as shown here by the darker-shaded portions within feature maps). To cover the entire feature map, tiles are iterated through in a raster scan — first horizontally, then vertically. 102

4-11 Network-on-Chip (NoC) connecting PE blocks to on-chip memory. Input feature maps, weights, and biases are delivered to PE blocks from on-chip global buffers (GLBs). There is a separate GLB for each of the different datatypes, and all GLBs are banked to provide parallel read access to all PE blocks. After convolution, outputs are held within PE blocks; each PE block has dedicated storage for the depthwise and pointwise output channels computed by that block. In-order read access for those outputs is controlled via block selectors (multiplexers). 106

4-12 Handling convolution strides in the NoC and the PE. 108

4-13 Buffer and alignment logic to facilitate convolution striding and adding skip connections. In both cases, four feature map tiles are buffered on-chip at once. 109

4-14 Top-level design wrapper and the DMA interface. 114

4-15 Shifting skip connections in the decoder so that they terminate *before* interpolation allows us to ensure that nearest-neighbor interpolation is always followed by a 5×5 convolution; this enables the decomposition of the 5×5 convolution into smaller 3×3 ones. Additionally, this shift allows us to downsample feature maps from the encoder prior to them being passed along the skip connection, which contributes to a reduction in overall feature map data movement. 117

4-16 Decomposition of a 5×5 kernel into 4 smaller 3×3 kernel. This decomposition is valid when the 5×5 convolution is preceded by nearest-neighbor interpolation. The red boxes here indicates windows of pixels in the interpolated input feature map that have identical values. As the 5×5 kernel slides across, kernel values inside the 2×2 windows get multiplied by identical feature map values. instead of performing 4 multiplications and 4 additions, the kernel values can first be pre-added and then multiplied once by the shared pixel value. 120

4-17 After filter decomposition, each of the four smaller 3×3 filters can be convolved with the *non-interpolated* input feature map. This produces four outputs that can be interleaved. The resulting output feature map with match the one coming from the original 5×5 convolution performed after nearest-neighbor interpolation. 121

4-18 Receptive field refers to the window of the input that is visible to an element of a filter in a convolution (shown here by the dashed squares). The receptive field of two cascaded 3×3 convolutions matches that of a single 5×5 convolution. 124

4-19 Accelerator-friendly FastDepth DNN topology. The two key modifications are in the decoder: (1) shifting skip connections to terminate *before* nearest-neighbor interpolation and downsampling feature maps passed along the connections, and (2) replacing the first 5×5 convolution with a cascade of two 3×3 convolutions. 125

4-20 Comparing FastDepth network topologies after channel pruning with NetAdapt [4]. Figures show the number of input channels to each layer in the network. The shaded part represents the topology before pruning. The very first layer to the network (`mobilenet.0`) is not shown since the channel size of the input fed into the network remains fixed at 3 channels (RGB). Overall, the shapes of the two topologies look similar. The modified network contains an extra layer at the beginning of its decoder; to compensate for this, several other layers get pruned more. 126

4-21 For feature map sizes to be preserved during 3×3 convolution, the height and width of the input feature map must be padded with a single row or column of elements on each side. Here, the input feature map is shown in blue, the kernel is shown in grey, and the output feature map is shown in teal. Figure taken from [5]. 134

4-22 Padding and tiling input feature maps for 3×3 convolutions. In the FastDepth accelerator, the output feature map tile height and width are set to be 7×7, meaning that input feature map tiles must have a height and width of 9×9. This figure illustrates an example of how a $14\times14\times C$ input feature map is padded and then tiled into four $9\times9\times C$ tiles. Overlapping regions (called halos) amongst adjacent tiles are depicted with a diagonal pattern. 135

4-23 Layer-by-layer spatial utilization of the PE array. Our accelerator achieves high spatial utilization for almost all FastDepth layers. The only significant exception is the final layer that produces just a single output channel. 140

4-24 Layer-by-layer temporal utilization of the PE array. There are two sources of temporal utilization hits: PE idleness while feature map GLBs are filled up with data from DRAM (this impacts depthwise temporal utilization) and PE idleness while output feature maps are written out to DRAM (this impacts pointwise temporal utilization). Since there is far less depthwise computation in FastDepth than there is pointwise computation, the depthwise temporal utilization hit is far more noticeable. 141

4-25 Accelerator logic breakdown by module type. 144

4-26 Layer-by-layer runtime in simulation (clocked at 250 MHz). Pointwise computation dominates active time, as is to be expected since there are more pointwise MACs than depthwise MACs in FastDepth. Depthwise idle time is due to PEs waiting for the input feature map GLB to fill up (which is why layers 0 and 2, requiring 4× as many input activations due to convolution strides of 2, experience high depthwise idle times). Pointwise idle time is due to PEs waiting for output buffers to clear out. 146

4-27 Layer-by-layer power consumption in simulation. 147

4-28 Power consumption breakdown from simulation. BRAM power consumption dominates, followed by logic power and signal power. 148

4-29 Layer-by-layer runtime on hardware (clocked at 250 MHz), compared with the previously reported layer-by-layer runtime from simulation. . 149

4-30 System runtime including PYNQ API calls and feature map transformations between layers. Feature map transforms involve aligning and merging output tiles, which incurs significant runtime overhead (to be discussed in Section 4.6.5). 150

4-31 System power consumption during end-to-end FastDepth inference on the Ultra96. In its idle state, the system consumes around 4.75 W of power. During inference, the system consumes around 6.1 W. 151

4-32 External memory accesses in our design vs. target minimums. 152

4-33 Timing diagram illustrating memory accesses overlapped with depthwise and pointwise computation. The red boxes highlight two potential sources of idle time in the PE array. Both represent challenges in hiding memory access latency. 155
4-34 Flow diagram of feature map transformations taking place between layers. These transformations convert the output stream coming from the accelerator into a high-dimensional tensor that is padded and tiled before being fed back into the accelerator via an input stream. In our system, these transformations are done by the CPU onboard the Ultra96, while layer processing is done on the FPGA. 158

List of Tables

1.1 Shape parameters for layers in a DNN alongside their descriptions. . . 34

2.1 Comparing FastDepth against prior work. For δ_1, higher is better. For all others, lower is better. Statistics for cited works come from our re-implemented models. Reported runtimes are measured in PyTorch on an NVIDIA Jetson TX2. Our network design achieves close to state-of-the-art accuracy with a significant reduction in MACs and GPU runtime. 61

2.2 Comparison of encoder variants in our ablation study. RMSE and δ_1 are for encoder-decoder networks with the decoder fixed as NNConv5. All other metrics are for the encoder in isolation. Runtimes are measured in PyTorch on a TX2. We select MobileNet as the best encoder option. 62

2.3 Comparison of decoder variants in our ablation study. RMSE and δ_1 are for encoder-decoder networks with a MobileNet encoder. All other metrics are for the decoder in isolation. Runtimes are measured in PyTorch on a TX2. We select NNConv5 as the best decoder option. . 64

2.4 Impact of depthwise decomposition and skip connections in the decoder on network complexity and TX2 runtime. 65

3.1 Hardware-specific compilation enables inference speedup on both the CPU and GPU when incorporating depthwise separable layers in our network. Additive skip connections do not add noticeable runtime overhead after compilation, as is expected. All runtimes are measured on the Jetson TX2. 70

3.2 Impact of pruning on our encoder-decoder network. Pruning together with compilation enable real-time inference throughput on the CPU at 27 fps and further increase throughput on the GPU to 178 fps. Reported runtimes are measured after compilation for the Jetson TX2. 73

3.3 Comparing our pruned and compiled FastDepth network against prior work. For δ_1, higher is better. For all others, lower is better. Statistics for cited works come from our re-implemented models. Reported runtimes are measured in PyTorch on an NVIDIA Jetson TX2 in max-N mode. Our final network design surpasses real-time inference speeds on both the GPU and CPU. Overall, FastDepth achieves close to state-of-the-art accuracy while running an order of magnitude faster. 74

3.4 Summary of NVIDIA Jetson TX2 power modes, taken from [6]. Max-N mode allows for the highest performance (throughput) at the cost of higher power consumption. Max-Q mode aims to provide the best power-throughput tradeoff. 75

3.5 Inference runtime and peak power consumption when deploying FastDepth on the Jetson TX2 in high performance (max-N) and high efficiency (max-Q) modes. Active power consumption can be estimated by subtracting the idle power consumption from the reported total power consumption. In both power modes, FastDepth consumes less than 10 W of active power. 77

3.6 Inference runtime of our FastDepth CoreML model on Apple iPhone. 79

4.1 Selecting a dataflow for pointwise convolution: choosing between weight-stationary and output-stationary. The weight-stationary dataflow suffers from high overhead of writing and reading high-bitwidth pointwise partial sums. The output-stationary dataflow avoids this overhead by holding partial sums within processing elements, thus achieving roughly 10× reduction in data movement. This makes it a more appealing choice for pointwise convolution. 89

4.2 Datatypes and bitwidths processed within a processing element. . . . 97

4.3 Block RAM usage for on-chip GLBs and output buffers. Total size refers to how much memory our design actively uses for each datatype. BRAM used refers to how many 18Kb and 36Kb block RAMs are placed-and-routed in our design by the synthesis and implementation process. 112

4.4 Impact of FastDepth network modifications on accuracy and data movement (reads and writes to off-chip DRAM). Compared to the original DNN, our modified accelerator-friendly DNN achieves equivalent accuracy with a 21% reduction in DRAM accesses. Quantization to 8-bit activations and 10-bit weights results in an additional 75% reduction in data movement, with negligible degradation of accuracy. 127

4.5 Logic utilization of the accelerator deployed on the Ultra96 FPGA. . 143

4.6 FastDepth accelerator runtime and energy efficiency given average power consumption of 6.1 W. The accelerator achieves real-time inference at over 30 fps. 149

4.7 Datatypes being streamed into or out of the accelerator, listed alongside their default and extended bitwidths. DRAM onboard the Ultra96 has a width of 32 bits, thus limiting our stream word size to 32 bits. We extend datatype bitwidths as necessary to facilitate packing into 32-bit words. 153

4.8 Evaluation of our accelerator against FastDepth on Jetson TX2. When compared to the TX2 CPU, our accelerator achieves about 1.5−2× improvement in energy-efficiency. FastDepth on the TX2 GPU, however, still achieves a higher energy-efficiency due to its much higher supported framerate. 159

4.9 Evaluation against related accelerators on the Ultra96 SoC. Definition of task abbreviations: IC = image classification, OD = object detection, DE = depth estimation. Our accelerator design consumes less power than many of the cited works, while performing inference at higher precision on a similar or more complex task. 160

Chapter 1

Introduction

Depth sensing is a fundamental task in computer vision, with numerous applications in 3D reconstruction and robotics. Extracting depth information from input sources such as digital imagery is a key step in autonomous robot navigation, as it enables localization of current position within an environment, mapping of the surrounding environment, and obstacle detection. Traditional depth sensing methods have typically relied on techniques such as stereo vision, where depth information for an object is computed from a pair of stereo images by determining differences (or disparities) in object locations across the images. Other commonly used depth sensing techniques rely on sensors emitting signal pulses and then computing round trip times upon detecting reflected pulses; examples of these time-of-flight sensors include sonar, radar, and lidar. These sensors face limitations in range and resolution and tend to be bulky with high power consumption due to active signal pulsing.

Addressing these depth sensing limitations has motivated research into more compact sensing, e.g., lidar-on-chip [7], as well as more energy-efficient algorithms for depth estimation from camera imagery. In particular, there has been a significant and growing interest in depth estimation from a single RGB image, due to the relatively low cost and size of monocular cameras. These cameras can be part of a larger navigational system including inertial measurement units (IMUs) and low-resolution depth sensors. Depth estimation from RGB images can be augmented by incorporating sparse depth measurements coming from onboard depth sensors or from simulta-

neous localization and mapping (SLAM) algorithms such as visual-inertial-odometry (VIO). However, monocular depth estimation remains the more accessible option as it assumes the presence of just a single camera onboard a navigating robot. This makes monocular depth estimation more appealing for deployment onto platforms that can be carried by miniaturized resource-constrained robots, e.g., micro aerial vehicles.

In this thesis, we explore fast and energy efficient monocular depth estimation on embedded systems. Many state-of-the-art depth estimation approaches are learning-based, and while they achieve high accuracy rates, they tend to be computationally complex and unsuitable for real-time, low-power inference. We address this challenge by developing a monocular depth estimation approach across the entire algorithm-to-hardware stack. Our approach involves (1) designing a compact deep neural network (DNN) for monocular depth estimation that achieves competitive accuracy rates at only a fraction of the size of DNNs in prior works, (2) refining the DNN topology that defines the shapes of layers within the network and applying hardware-specific compilation to achieve real-time inference on an embedded CPU/GPU, and (3) developing a custom dedicated dataflow and accelerator design for deployment on a low-power embedded FPGA to achieve energy efficient inference.

Our work intersects several research fields, including research focused on learning-based monocular depth estimation, research exploring compact and efficient neural network design, and research in developing hardware accelerators for deep neural networks. This chapter introduces these fields and their representative works as well as key terminology that will be used throughout the thesis.

1.1 Monocular Depth Estimation

In this section, we introduce the problem of monocular depth estimation. We survey relevant literature on learning-based monocular depth estimation as well as several related works deploying depth estimation DNNs on embedded systems. We then discuss limitations of learning-based approaches and highlight application areas to motivate the work presented in this thesis.

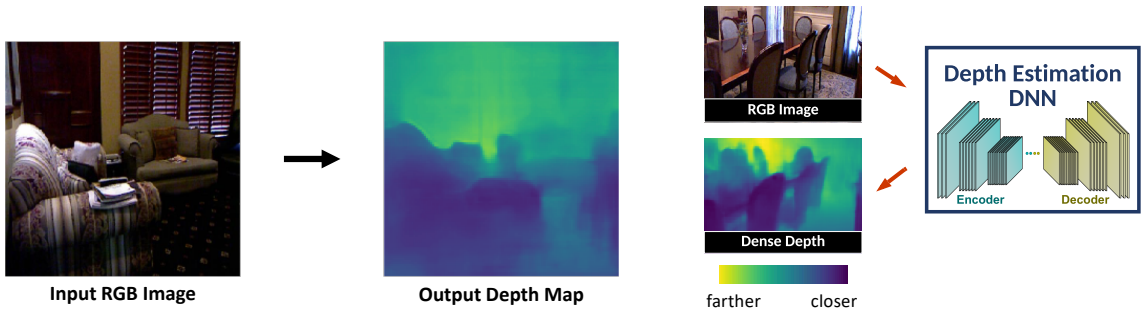


Figure 1-1: This thesis studies learning-based monocular depth estimation. Left: a simple visualization of this task, where an input color image is processed to produce an dense depth map containing depth measurements for every pixel in the image. Right: a diagram depicting learning-based depth estimation, where a deep neural network (DNN) is used to predit pixel-wise dense depth from the input image.

1.1.1 Problem Definition

Monocular depth estimation refers to the task of estimating pixel-wise depth measurements from a single two-dimensional color image. The input to this problem is a 2D RGB color image, and the output is a dense depth map, as illustrated in Figure 1-1. This task is challenging as it is inherently ill-posed and ambiguous. A 2D RGB image provides pixel-wise color information, which helps in identifying the relative placement of objects in the scene; however, the image alone provides no sense of scale. It is possible to imagine, then, how different RGB images could yield identical depth maps, e.g., images that maintain relative placement of objects but with scaled distances in the depth dimension that is perpendicular to the plane of the image. This presents a challenge for monocular depth estimation algorithms, as they must infer the proper scale in order to generate accurate pixel-wise depth measurements.

1.1.2 Literature Review

Monocular Depth Estimation

Early works on monocular depth estimation relied on hand-crafted features and probabilistic models [8], as well as non-parametric approaches [9–12]. More recently, the rise of deep learning and deep neural network algorithms along with their success on various computer vision tasks (e.g., image classification, object detection) has

prompted research in developing learning-based approaches for monocular depth estimation. These approaches typically involve training a convolutional neural network, a type of deep neural network, on large datasets of RGB image and dense depth map pairs. Our work in this thesis directly builds upon these methods.

Eigen et al. [13] proposed a two-stage convolutional neural network (CNN) design, with the first stage predicting the global coarse scale and the second stage refining local details. In [14], Eigen et al. added a third stage to increase output resolution and incorporated auxiliary prediction tasks like generating surface normals and semantic labeling into their depth estimation network. Liu et al. [15] combined a deep CNN with a continuous conditional random field and attained visually sharper transitions and local details. Laina et al. [2] developed a deep residual network based on ResNet [16] and achieved higher accuracy than [14, 15]. Qi et al. [17] trained networks to jointly estimate both depth and surface normals as a way to enforce geometric consistency between depth and normals in planar regions; this helped address blurriness in depth output. Semi-supervised [18] and unsupervised learning [19–21] approaches have also been explored for disparity image prediction. For instance, Godard et al. [21] formulated disparity estimation as an image reconstruction problem, where neural networks were trained to warp left images to match the right. Mancini et al. [22] proposed a CNN that took both RGB images and optical flow images as input to predict distance. Ma et al. [23, 24] explored fusion of RGB images and sparse depth measurements as input to improve the accuracy of depth estimation.

Popular RGB-Depth Datasets

Several key datasets have been developed for supervised training of neural networks for tasks like scene segmentation and depth estimation. Two of the most popular datasets for depth estimation include:

- the NYU Depth v2 dataset [25], containing pairs of RGB color images and densely labeled depth maps recorded in a variety of indoor scenes with the Microsoft Kinect [26] (a structured-light sensor).

- the KITTI dataset [27], containing RGB and depth pairs from outdoor road scenes, with ground truth depth collected using a Velodyne [28] lidar scanner.

The evaluations presented in this thesis are primarily done on NYU Depth v2.

Commonly-Used Evaluation Metrics

Many evaluation metrics have been used to evaluate monocular depth estimation methods. A few examples are absolute relative difference (AbsRel), squared relative error (SqRel), and root mean squared error (RMSE) [29]. These are defined as:

$$\text{AbsRel} = \frac{1}{N} \sum \frac{|d_n - d_n^*|}{d_n} \tag{1.1}$$

$$\text{SqRel} = \frac{1}{N} \sum \frac{|d_n - d_n^*|^2}{d_n} \tag{1.2}$$

$$\text{RMSE} = \sqrt{\frac{1}{N} \sum |d_n - d_n^*|^2} \tag{1.3}$$

where d_n and d_n^* are the ground truth and predicted depth values at a given pixel n and N is the total number of pixels. Another common metric is the δ_i accuracy that reports the percentage of pixels where the relative error is within a threshold:

$$\delta_i = \% \text{ of } d_n \text{ such that} \max\left(\frac{d_n}{d_n^*}, \frac{d_n^*}{d_n}\right) < 1.25^i \tag{1.4}$$

When evaluating depth estimation methods in this thesis, we primarily use RMSE and the δ_1 accuracy, i.e., the percentage of pixels that have predicted depth values within 25% of ground truth values. Of these, RMSE is more intuitive when gauging the accuracy of a particular depth estimation method during inference, since it indicates roughly how bad a depth prediction can get, e.g., whether on the order of a few centimeters or a few meters. The δ_1 accuracy metric is less intuitive as it does not indicate the magnitude of how inaccurate depth values outside the threshold may be.

Learning-Based Depth Estimation on Embedded Systems

As deployment of depth estimation DNNs in practical settings grows in importance, research focus is beginning to shift to compact depth estimation DNN design, targeting inference on embedded platforms. For instance, Poggi et al. [30] designed a pyramid-structure network consisting of network levels inferring depth at varying resolutions; their resulting PyD-Net successfully ran on an embedded ARM CPU onboard the Raspberry Pi 3, performing monocular depth inference on a KITTI dataset image in about 1.7 s. Another work, AnyNet [31], focused on stereo depth estimation and used a similar multi-stage structure computing feature maps at various output resolutions; the authors additionally incorporated a U-Net [32] feature extractor and disparity networks at different resolution stages to gradually refine disparity maps between left-right stereo image pairs. The multi-stage structure in AnyNet allowed for it to be queried at any time to output its best depth prediction at that moment; this produced a range of supported framerates. AnyNet was deployed onto NVIDIA's Jetson TX2 module, achieving $10-35$ fps on KITTI inference. More recently, in [33], Wang et al. made use of depthwise and pointwise layers and relied on automated network architecture search[1] to create their compact monocular depth estimation network; their resulting DepthNet Nano was then deployed onto NVIDIA's Jetson AGX Xavier module, achieving framerates of 8.8 fps on NYU Depth v2 and 7.7 fps on KITTI with power consumption of 15 W.

Our work on FastDepth, described throughout this thesis, focuses on monocular depth estimation on embedded systems. Instead of a multi-stage network producing predictions at different scales or resolutions, we use a more linear encoder-decoder structure that is primarily composed of efficient depthwise separable convolutional layers. We too target real-time inference on NVIDIA's Jetson TX2 module, though not only on the TX2 GPU but also on the TX2 CPU. We also take deployment two steps further: first by running FastDepth live on a mobile phone, and then by accelerating FastDepth on an embedded FPGA for reduced power consumption.

[1] Techniques to create compact neural networks, such as incorporating depthwise or pointwise layers and using network architecture search, will be discussed in Section 1.2.2.

Limitations of Learning-Based Approaches

Learning-based approaches for depth estimation face several limitations. One is inherent to how neural networks learn to perform tasks. In supervised training, a DNN is typically trained over a task- and environment-specific dataset. Upon successful training, the DNN will learn to perform similar tasks as it had seen in the dataset. As a case example, depth estimation DNNs that are trained on indoor datasets learn to estimate depth on an indoor scale (e.g., on the order of a few meters) and will not accurately estimate depth in outdoor scenes (where depths are on the order of, say, tens or hundreds of meters). This can be interpreted as the DNN learning a global scaling factor by which to predict dense depth measurements. Since this is highly dependent on the training dataset used, it presents a limitation in the interoperability of trained depth estimation DNNs across environments. Research into depth transfer learning or training over mixed datasets is being explored [34–36].

Another limitation is consistency of pixel-wise depth accuracy across time, e.g., when a video stream is passed through the depth estimation DNN. If the DNN does not contain any memory-like or feedback-like elements in its design, every consecutive input image will be analysed independently. Similar regions across consecutive frames will not always have similar depth estimates, e.g., due to noise, slight variations in lightning, occlusions, different depth ranges introduced by new objects at image edges, etc. This can manifest as flickering in frame-by-frame depth maps produced by the DNN. Some works [37, 38] have sought to resolve this and enforce temporal consistency in depth output by inserting long short term memory (LSTM) blocks or feedback loops into their designs. However, more research into this is still needed to improve the robustness and practicality of depth estimation DNNs.

Applications of Depth Estimation

Depth estimation is critical for many robotics applications, especially navigation. It is often a key step in localization and mapping (SLAM) algorithms, and several works have incorporated learning-based depth estimation into SLAM frameworks,

e.g., CNN-SLAM [39]. Depth estimation is also a key step in 3D reconstruction algorithms, with applications such as augmented reality and medical imaging [40, 41].

Monocular depth estimation becomes important when we consider miniaturized robots that are power-constrained, resulting in weight and compute constraints. The onboard sensor technology on such robots may be limited to a simple RGB camera, and no additional information (e.g., stereo image pairs, IMU measurements, sparse depth point clouds, optical flow) may be present. For such an application, the ability to estimate dense depth from just a single RGB image in a computationally efficient manner becomes a challenge and a goal.

1.2 Efficient Neural Network Design

One of the disadvantages of state-of-the-art deep learning approaches for monocular depth estimation is that they have become increasingly complex; networks are often designed for higher accuracy, leading to models that are deeper, contain larger layers, or involve additional post-processing. These all increase a network's computational complexity, which then increases the latency (runtime) and power consumption cost of running the network on hardware. Latency and power cost become especially critical when deploying neural networks for time-sensitive applications, e.g., autonomous navigation, that require real-time processing. For instance, consider a small robotic vehicle trying to navigate its environment, using depth output from a DNN to sense its surroundings and help plan its motion. If the depth estimation DNN is too slow, the robot will misinterpret obstacles and crash. If the DNN consumes too much power, there will be less power available for other computing tasks, limiting how much or how far the robot can explore. This example highlights the need for efficient neural network design that seeks to balance tradeoffs between accuracy, computational complexity, latency, and power consumption. In our work, the target latency for single-image DNN processing is under 40 ms as this enables framerates above 25 fps, which is considered real-time. Our target power consumption is defined by the types of embedded devices we deploy on, e.g., embedded CPU/GPU platforms can

consume on the order of 10−20 W, while a custom hardware design on a small FPGA can lower power consumption to under 5 W.

In this section, we provide a brief overview of deep neural networks, followed by a literature review of work on compact neural network architectures. We also discuss research on steps that can be taken after network training to further reduce network complexity and latency while maintaining accuracy. Several of the design methods described in this section are utilized in our own work presented later on in this thesis.

1.2.1 Overview of Deep Neural Networks

Deep neural networks consist of many interconnected layers that emulate functions of biological neurons. The topology of a DNN is inspired by the structure of the human visual system, so that processing performed by the DNN on a visual input mimics the human's response to similar stimuli. During training, DNNs extract high-level features from raw sensory data and learn over a large set of training data to obtain a representation of the input that can be used to infer information about new data. Running inference using successfully trained DNN models can then yield highly accurate classification and regression results [1]. This section describes different types of layers and convolutions found in these DNN models.

Layer Types

As DNNs have grown in complexity over time, numerous layer types have developed. We narrow down to those most commonly used in DNNs:

Convolutional layers perform high-dimensional convolutions on input feature maps and sets of filters to produce output feature maps. A feature map is just a collection of data in a 4D tensor with shape $N \times C \times H \times W$.[2] Table 1.1 summarizes what these shape parameters mean. Figure 1-2 illustrates the convolution performed by this layer. Activations from input feature maps passed into the layer consist of

[2]This format may vary for different deep learning frameworks, e.g., PyTorch [42] uses NCHW format by default, while TensorFlow [43] uses NHWC format by default. Conversion between such formats can be done through a simple permutation of dimensions.

Shape Parameter	Description of Parameter
N	batch size of feature maps
C	number of channels in input feature map
M	number of filters (and channels in output feature map)
H/W	height and width of input feature map
H_T/W_T	height and width of input feature map tile
R/S	height and width of filter
P/Q	height and width of output feature map
P_T/Q_T	height and width of output feature map tile

Table 1.1: Shape parameters for layers in a DNN alongside their descriptions.

C channels of 2D features. These are convolved with weights from M filters, where each filter contains C channels. Each channel of the input feature map is convolved with a single filter channel; the results from these C convolutions are then added element-wise to produce a single output feature map. Since there are M filters, the convolutional layer produces M total output feature maps per every batch of input feature maps. Batch sizes can be increased to benefit from filter reuse; in this scenario, the layer will produce an N-sized batch of M output feature maps each.

The operations taking place in convolutional layers are multiplications of filter values with input activations and accumulations of products in the spatial and channel dimensions. In combination, these are referred to as MAC (multiply-accumulate) operations. Once the MACs for a given feature map channel have been completed, a bias may be added at the output of the layer. The same bias value gets applied to an entire channel and simply shifts the channel values in a positive or negative direction. In this case, the layer will contain M biases, one per output channel.

Fully connected layers can be interpreted as special cases of convolutional layers; they too convolve input feature maps with filters, but in this case, the filter dimensions match those of the input feature map; that is, in fully connected layers, $R{=}H$, $S{=}W$, $P{=}Q{=}1$. With a batch size of 1, the output of this layer is just a 1-dimensional M-sized vector. Fully connected layers are often placed towards the end in networks performing classification tasks, where the output is a vector of classification estimates. They are less common in networks for regression tasks

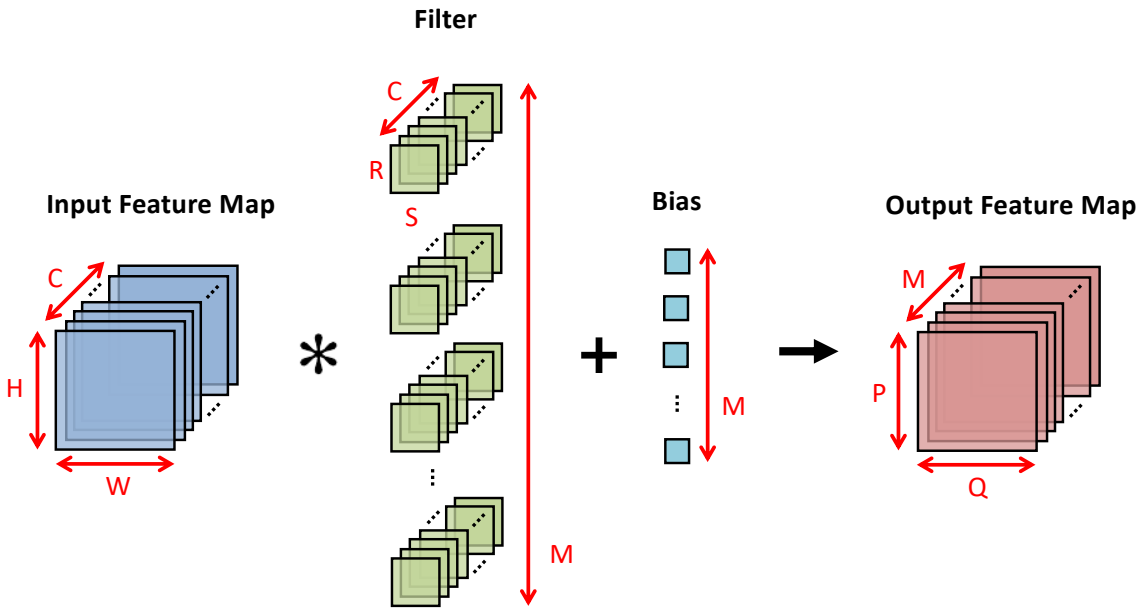


Figure 1-2: Diagram of a convolutional layer. Each of the M filters is first convolved channel-by-channel with the input feature map. The results are then added element-wise to yield a single output channel. This repeats for all M filters, producing M output channels in total. Output channels may be subject to a channel-wide bias that is added after convolution.

producing multi-dimensional feature maps, e.g., depth maps, at the output end.

Non-linearity functions — also called activation functions — are typically applied after a convolutional or fully connected layer. They are used to introduce non-linearity into the computation performed by the neural network, allowing the network to learn more complex representations. A commonly-used non-linearity function is the Rectified Linear Unit, or ReLU, that zeroes out negative values in a feature map. It can be represented as the function $y = \max(0, x)$.

Pooling layers operate on a channel-by-channel basis and reduce the height and width dimensions of a feature map by consolidating values in regions of the feature map (e.g., by taking the maximum value in a region, or averaging them).

Normalization layers help control the feature map distributions throughout the network by normalizing them using parameters learned during training; these are commonly applied after convolutional and fully connected layers.

Convolution Types

The convolution depicted in Figure 1-2 can be thought of as **standard convolution**, since it has been a standard layer in many early image classification DNNs. As described earlier, this convolution operates on $R \times S \times C \times M$ filters, a product that grows as dimensions C and M grow within the network. The total number of feature map activations generated during the convolution is $H \times W \times C \times M$.

In an attempt to reduce this number of activations as well as the number of filter weights, works like [44] and [45] have explored replacing standard convolutional layers with **depthwise separable** layers, as shown in Figure 1-3. These layers essentially factorize a standard convolution into two smaller ones: a **depthwise convolution** that uses C single-channel filters and performs channel-wise convolution, followed by a **pointwise convolution** that uses M filters, each of shape 1×1 with C channels, performing element-wise channel aggregation. This reduces the number of activations generated between the two convolutions to just $H \times W \times C$. The number of filters is also reduced to $R \times S \times 1 \times C$ for the depthwise filter and $1 \times 1 \times C \times M$ for the pointwise filter. In this manner, depthwise separable layers require less parameters, generate less feature map data, and are less computationally costly, making them more likely to achieve higher efficiency[3] than standard convolutional layers.

1.2.2 Compact Network Architecture Design

The compactness and efficiency of a deep neural network factors into how well-suited the DNN is for practical applications. Oftentimes, inference latency and power consumption are of concern, especially when deploying networks onto resource-constrained mobile and embedded devices. This has motivated an abundance of work centered around compact network architecture design, i.e., streamlining neural network topologies by modifying shape parameters, while still targeting high accuracy. Methods used to design compact and efficient neural networks can be grouped into manual network design and automated network architecture search (NAS).

[3]Here, we define efficiency as the amount of computation performed on a given energy budget. Less computation with less data movement leads to shorter runtime and improved energy efficiency.

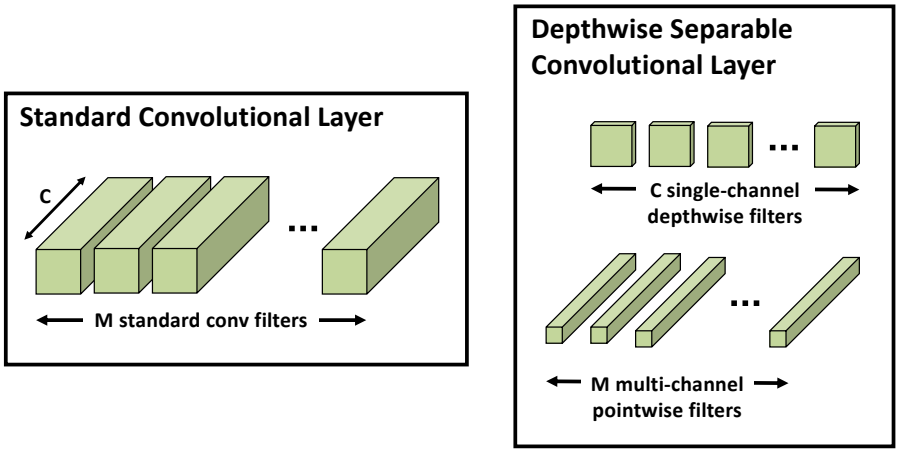
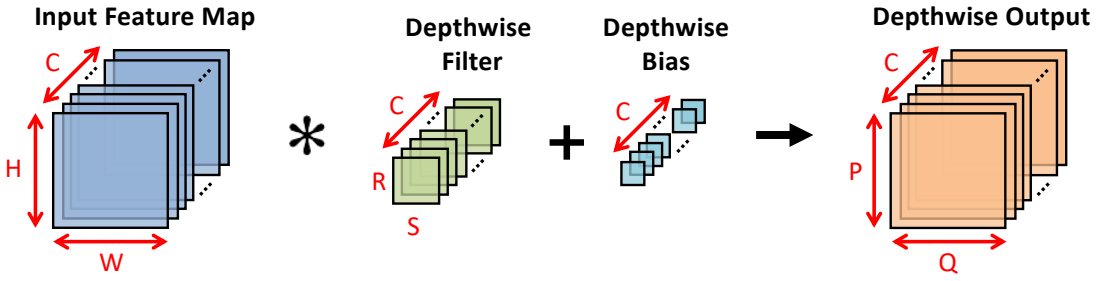


Figure 1-3: A depthwise separable layer factorizes a standard convolution into two smaller ones: a depthwise convolution performing channel-wise convolution, and a pointwise convolution performing element-wise channel aggregation.

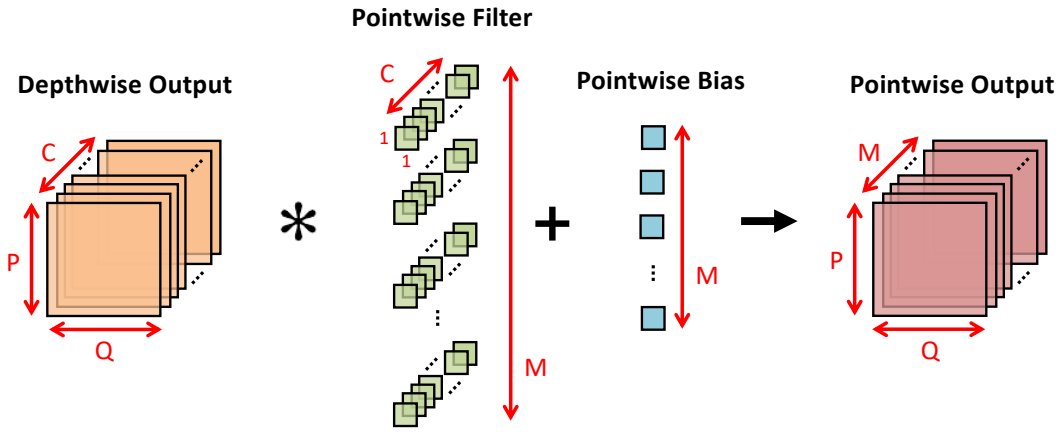
Manual Network Design

Manual network design refers to manually creating efficient building blocks out of which a neural network is to be constructed. This often involves simplifying network layers. The layer most prevalent in convolutional neural networks is the standard convolutional layer. This type of layer can be simplified in various dimensions. The spatial dimensions (height and width) can be reduced by replacing larger kernels with smaller ones — for instance, Simonyan et al. [46] proposed replacing 5×5 kernels with cascaded 3×3 kernels, while Szegedy el al. [47] showed that $M \times M$ convolutions can be replaced with cascaded $1 \times M$ and $M \times 1$ kernels. The channel dimensions of a layer can be reduced through bottleneck layers and group convolutions:

Bottlenecks can be inserted into a network in the form of 1×1 pointwise convolutional layers with the number of output channels being lower than the number of input channels. In deep ResNets [16], He et al. inserted 1×1 layers before and after 3×3 convolutional layers, thus allowing for lower input and output channel dimensionality of the 3×3 layers. In SqueezeNet [48], Iandola et al. introduced modules consisting of 1×1 layers that reduce feature map channel dimensions (squeezing the network), and combinations of 1×1 and 3×3 layers that later increase channel dimensions (expanding the network).



(a) depthwise convolution



(b) pointwise convolution

Figure 1-4: Diagram of a depthwise separable layer. This type of layer consists of a depthwise convolution shown in (a) and a pointwise convolution shown in (b).

Group convolutions divide filters within the convolution into multiple groups that then operate on distinct subsets of the input feature map. This yields a reduction in parameters (filter weights) and MAC operations. First introduced in AlexNet [49], group convolutions were then adopted in [44, 50]. Channel-wise convolution, also referred to as depthwise convolution, is an extreme case of group convolution where the number of groups equals the number of input channels such that each group has only one filter. Depthwise convolution can be directly followed by pointwise convolution in what forms a depthwise separable layer, as described earlier in Section 1.2.1. Howard et al. [45] leveraged this concept to develop a family of MobileNets — highly efficient networks suitable for mobile applications. Sandler et al. further improved on this by introducing bottleneck layers into the networks in MobileNetV2 [51].

Automated Network Architecture Search

A significant difficulty in manual network design is its tedious process, involving many design steps, e.g., selecting the number of layers, the types of layers, the ordering and connections between layers, as well as a multitude of hyper-parameter settings during training. Recent research efforts have sought to automate this process through Network Architecture Search (NAS). At the core of NAS is an optimization algorithm that samples network architectures from a search space, evaluates them, and subsequently decides which network architectures to sample next. The algorithm is iterative and continues until a network design meeting specified criteria is discovered.

Some NAS frameworks have sought to take target hardware platforms into account alongside neural networks. In NetAdapt [4], Yang et al. proposed an algorithm that progressively simplifies a pretrained network while maximizing accuracy until a resource budget is met; to guide the simplification process, NetAdapt incorporates empirically-evaluated metrics, e.g., measured latency or energy consumption, into its algorithm. In MNasNet [52], Tan et al. presented an approach for designing resource-efficient mobile CNN models using reinforcement learning; their approach incorporated platform-aware latency measurements into the search process and used a hierarchical search space that encouraged layer diversity to explore trade-offs be-

tween accuracy and latency. MNasNet was subsequently used in the development of MobileNetV3 [53] that had fewer MACs and saw improvements in the accuracy vs. latency tradeoff curve over MobileNetV2 [51].

1.2.3 Network Pruning

Hand-crafted networks tend to be over-parameterized during training, which results in a large network footprint and reduces network efficiency. To address this, network pruning [54–59] has emerged as a technique to identify and remove redundant parameters. The pruning process can be applied to different aspects of convolutional layers, e.g., filter weights can be pruned by being set to zero, and channels can be pruned away by reducing channel dimensions in filters. This results in layers with fewer channels or sparser filters, making the neural network more compact overall. The pruning process can also be guided by different target metrics, e.g. an energy consumption estimate of a weight [60]. Lastly, as proposed in NetAdapt [4], pruning may take a target hardware platform into account by incorporating metrics such as simulated or empirically-measured latency and energy consumption on hardware.

Network pruning is typically applied after a neural network is trained. Since pruning removes parameters from the network, it often results in some accuracy loss. As part of the pruning process, gradually-pruned networks undergo fine tuning (retraining) in an attempt to restore some of the accuracy and to ensure that each pruning step degrades as little accuracy as possible. Sometimes, network pruning may even slightly increase the accuracy of the network if the originally-trained network happened to have overfitted to a training set; this is because the reduction of parameters via pruning will have lessened the effect of the overfit. Additionally, since pruning effectively changes layer shapes within a network, it can be viewed as a form of network architecture search (NAS), with the pruned network architecture being the discovered network that can be initialized and trained from scratch.

As pruning methods have developed, they have been mainly applied and tested on image classification networks. It is reasonable to expect that pruning methods will soon be applied more frequently to depth estimation DNNs to generate lightweight

networks that are more easily deployable, as in [61].

1.2.4 Network Quantization

Common deep learning frameworks such as PyTorch [42] and TensorFlow [43] predominantly support training and inference in 32-bit floating point precision. However, the desire to reduce data bandwidth and computational cost of running neural network onboard embedded devices has motivated research into inference at reduced precisions. This reduction in precision of data is referred to as quantization.

A direct benefit of network quantization is reduced bandwidth and storage requirements for inference, e.g., quantizing weights and activations from 32-bit floating point to 8-bit integer lowers both bandwidth and storage needs by $4\times$. Another benefit is compute speedup, as hardware units for integer operations tend to be faster and less area costly than those for floating point operations. Hence, quantization leads to improved area and energy efficiency [62].

The process of quantizing values involves mapping them into a smaller set of quantization levels. These levels can be evenly spaced out (uniform quantization) or they may be assigned according to a non-uniform, e.g., logarithmic, distribution (non-uniform quantization). Quantization methods fall into one of these two categories.

Over the past several years, there has been an abundance of research into network quantization at various bitwidths [63–65]. Quantizing to 16 bits or 8 bits has grown common, though a few works have explored quantization as low as 2 bits [66, 67] and 1 bit [68, 69]. A significant challenge faced with quantizing to such low precision is maintaining network accuracy, since reduced precision limits how well a quantized network can approximate the non-quantized network. It may be possible, however, to restore some lost accuracy through fine-tuning or retraining after quantization.[4]

Quantization schemes can also vary in how coarse-grained or fine-grained they

[4]One of the challenges with quantization-aware training is backpropagation through quantization functions that usually resemble a step function and have a derivative of 0 almost everywhere. One way to bypass this challenge is to use straight-through estimators [70, 71], which passes the gradient through. There is currently limited support for quantization-aware training in commonly used frameworks like PyTorch and TensorFlow.

are. Coarse-grained schemes may quantize on a tensor-wise basis, using the range across all tensor dimensions to determine quantization levels. Finer-grained schemes may quantize tensors sliced along the certain dimensions, e.g., on a channel-wise basis. Quantization schemes may use different target bitwidths for different datatypes, or even vary these datatype bitwidths on a layer-per-layer basis. Increasingly fine-grained schemes will result in lower accuracy degradation but incur a higher hardware complexity cost, as the hardware compute engine will need to flexibly support variable bitwidths. This is more feasible with custom hardware design, e.g., UNPU [72] was designed to support variable precision (at least for a single datatype).

Quantization together with pruning have great potential for reducing the size of a neural network. Both were shown to be valuable steps in compressing a neural network for efficient inference on embedded systems [73].

1.3 Accelerators for Deep Neural Networks

Embedded platforms typically have tight energy, compute, and memory constraints. For successful on-platform DNN inference in light of these constraints, efficient processing becomes paramount. There are two key factors affecting the processing of DNNs: compute parallelism and data movement. The types of hardware architectures that are used to run DNN inference can be analysed with respect to these factors. More general-purpose hardware, such as CPUs and GPU, tend to have temporal architectures that employ techniques such as vector processing and multithreading to increase compute parallelism. More specialized hardware, such as FPGAs and ASICs use spatial architectures that exploit data reuse and take advantage of low-to-high cost memory hierarchies to reduce the net cost of data movement.

1.3.1 CPU and GPU Acceleration

CPUs and GPUs support high levels of compute parallelism through vector instructions (SIMD, or single instruction, multiple data), multithreading (SIMT, single instruction, multiple threads), and multiprocessing (dividing computation across mul-

tiple cores). CPUs tend to operate in the GHz frequency range and may contain up to tens of processing cores. GPUs tend to operate at lower frequencies in high MHz range but contain far more processing cores — on the order of hundreds to thousands of cores. Both have largely fixed storage hierarchies consisting of multi-leveled caches, e.g., L1, L2, L3, etc. GPUs also make use of local memory blocks within streaming multiprocessors as well as shared memory between multiprocessors.

When running DNNs on CPUs and GPUs, acceleration primarily comes from optimized matrix multiplications for computing MACs. There are many software libraries that perform these optimizations. Convolutions in DNN layers can undergo transformations such as tiling, unrolling, etc, to generate large matrix-matrix multiplications. Additional computational transforms that can be applied for further acceleration include FFT [74], Strassen [75], and Winograd [76, 77] transforms. The final matrix operations can then be optimized through software libraries, e.g., OpenBLAS [78] for CPUs, cuBLAS [79] and cuDNN [80] for GPUs. However, a drawback of relying on these libraries is that they are general-purpose and therefore less likely or more difficult to adapt for operators or convolutions emerging in newer deep learning models; that is to say, as the rapid development of DNNs in research and academia continues, general-purpose compilation libraries fall behind [81]. This is motivating work on dedicated neural network compilers, as will be discussed in Section 1.3.3.

CPUs and GPUs, ranging from server-level to embedded-level, have evolved to support deep learning applications. Intel's line of latest Xeon Phi processors feature Advanced Vector Extensions (AVX) that benefit floating-point multiply-and-add operations. Deep learning oriented GPUs from NVIDIA include the Tesla brand incorporating architectures such as the Maxwell K80, the Pascal P100, and the Volta V100. Similar offerings from AMD include the Radeon Instinct brand. Larger server systems like NVIDIA's DGX [82] have been designed entirely for deep learning acceleration, notably for DNN training. On the other spectrum, smaller systems like NVIDIA's Tegra SoCs[5] have been designed to support DNN inference, allowing for

[5]The Jetson TX1 [83] incorporates 4 ARM CPU cores and 256 Maxwell GPU cores. The Jetson TX2 [84] incorporates 6 ARM cores and 256 Pascal cores. Both are aimed at DNN inference only. Power consumption is around 10-20 W when busy, orders of magnitude less than GPU server systems.

deployment of DNNs on embedded platforms with much lower power consumption.

1.3.2 FPGA and ASIC Acceleration

In addition to supporting high compute parallelism, FPGA (field-programmable gate array) and ASIC (application-specific integrated circuit) accelerators allow for flexibility in memory hierarchy design and in dataflow design, i.e., specifying how data flows through the compute elements. This distinguishes FPGAs and ASICs from CPUs and GPUs where the dataflow and memory hierarchy are not customizable. Since data movement significantly contributes to energy consumption during DNN processing [1], this factors into FPGA and ASIC accelerators generally achieving lower energy consumption during than CPUs or GPUs.

There are several aspects that further distinguish between FPGAs and ASICs. It is typically cheaper to program a prototype accelerator on an FPGA than to design and tape-out a prototype ASIC. FPGA designs are more easily adjustable due to their programmable interconnect of logic nets, while ASIC designs are fixed after tape-out and can only be reconfigured if reconfigurability was built in. However, ASICs allow for more customizable clock tree design and placement of logic nets (in contrast to the already-placed fabric on FPGAs). Hence, ASIC designs may achieve higher speeds with lower power consumption than equivalent FPGA designs.

FPGAs and ASICs have become appealing choices for datacenters offering DNN acceleration as a service in the cloud. Google's Tensor Processing Unit (TPU) [85] and Microsoft's Brainwave Neural Processing Unit (NPU) [86] are examples of accelerators designed for datacenter applications. ASICs dedicated for deep learning tasks are also becoming increasingly commonplace in consumer produces such as mobile phones, e.g., Apple's Neural Engine (ANE) [87] and Qualcomm's Artificial Intelligence Engine [88]. In academia, both FPGA-based and ASIC-based accelerator design for DNN processing has grown in popularity, with numerous hardware architectures for various deep learning domains being proposed every year [72, 89–97].

1.3.3 Neural Network Compilers

As mentioned earlier in Section 1.3.1, the general-purpose nature of CPU and GPU compilers makes it more difficult for them to support an increasing number and variety of operator types (e.g., different convolutional layers) in deep learning models. This has motivated the development of compilers dedicated to neural network acceleration. Neural network compilers can be described in two parts: the compiler frontend and the backend. The frontend takes a DNN model from a deep learning framework and transforms it into a computation graph. Each node in such a graph represents a layer operation that takes in one or more tensors and produces one or more tensors; connections between nodes represent dataflow dependencies between operations [98]. This computation graph is also referred to as a high-level intermediate representation (IR) and is hardware-agnostic. The backend then takes this high-level IR and converts it into what is then called a low-level IR. Hardware-specific optimizations, e.g. layer and tensor fusion, kernel transformations, tiling, loop unrolling, etc, are performed at this low-level stage. The low-level IR reflects hardware-specific characteristics and may be further converted to be compatible with existing compilation toolchains such as LLVM for CPUs and CUDA for NVIDIA GPUs. The optimized low-level IR is finally compiled into an executable that can be directly run on a hardware target [81].

Current deep learning frameworks mostly implement DNN optimizations at the graph level. However, this is too high-level to be able to handle hardware-specific operator implementations. In some cases, the underlying operator implementations are optimized for runtime and efficiency on server-class systems; in other cases, frameworks rely on operator libraries that require manual tuning (exploring and selecting optimally-performing design knobs for each operator). These predefined operator libraries can then limit how well the DNN computation graph is optimized, e.g., if graph-level changes yield new operators not present in the libraries. All in all, this precludes operator implementations from being portable across a diverse set of hardware targets and may result in under-optimized operator or layer execution on hardware [98]. One way in which unoptimized execution manifests itself is in layers

not speeding up when computation is reduced. This has been observed with depthwise separable layers: although depthwise decomposition results in fewer parameters and MACs than in a standard convolutional layer, that has not translated to a runtime reduction. Using a compiler that performs both graph- and operator-level optimizations (instead of relying on a predefined operator library) helped to resolve this [98, 99].

The increase in target platform types (e.g., CPUs, GPUs, TPUs, FPGAs, etc.), has spurred research into compilers for deep learning applications across all of these platforms. TensorFlow's XLA (Accelerated Linear Algebra) [100] optimizes TensorFlow models by generating computation kernels unique to a given model and fusing them, instead of relying on precompiled GPU kernel implementations. NVIDIA's TensorRT [101] is compatible with a variety of deep learning frameworks, performs graph-level optimizations, and supports precision calibration for lower-precision inference; it primarily targets deployment on CUDA-compatible GPUs. The AutoTVM project [98] offers a compiler stack that too supports multiple deep learning frontends as well as multiple backends targeting both CPUs and GPUs. The TVM compiler uses a learning-based optimization approach for hardware-specific operator tuning. Xilinx's Vitis AI development kit [102] incorporates a compiler that targets deployment of DNN models on newer Xilinx FPGAs. Additional examples of DNN compilers include Tensor Comprehensions [103], nGraph [104], and Glow [105].

1.3.4 Dataflow-Based Accelerator Design

Developing a dataflow is a key step in designing accelerator hardware for DNNs. In the context of DNN processing, a dataflow refers to how MAC operations within a DNN layer are ordered and how inputs and outputs to and from the layer are transferred. Dataflow design involves exploring how temporal and spatial reuse of data can be leveraged to reduce energy costs of data movement and to improve the energy efficiency of the DNN accelerator built to enable that dataflow.

Leveraging Data Reuse

There are two aspects to consider in leveraging data reuse: what data is reused, and how the data reuse is exploited. As part of a convolutional operation, there is potential for various operands to be reused, e.g., input activations, or filter weights, or both. Input feature map reuse arises from different filters being applied to the same input feature map channels to produce different output feature map channels. Filter reuse arises from batching, where the same filters are applied to all input feature maps within a batch. Convolutional reuse arises from the sliding window action when performing a convolution — as a filter slides across an input feature map, both the filter weights and input activations are reused.

Data reuse may be exploited in a temporal or a spatial manner. Temporal data reuse refers to a particular data value being used multiple times by the same compute element. This reuse is often exploited via a memory hierarchy, where highly-reused data is stored in smaller memory blocks closer to the compute element; this allows for faster and less costly memory accesses for that highly-reused data. Data with less reuse is stored in larger memory blocks farther away from the compute element. Data that cannot fit into the limited storage capacity within the memory hierarchy is stored off-chip in external memory, e.g., in DRAM. Spatial data reuse refers to a particular data value being used by multiple compute elements at the same time. This reuse is often exploited via multicasting, where the highly-reused data is read once from memory and sent to many compute elements. Multicasting requires a one-to-many network to be established between memory and compute elements.

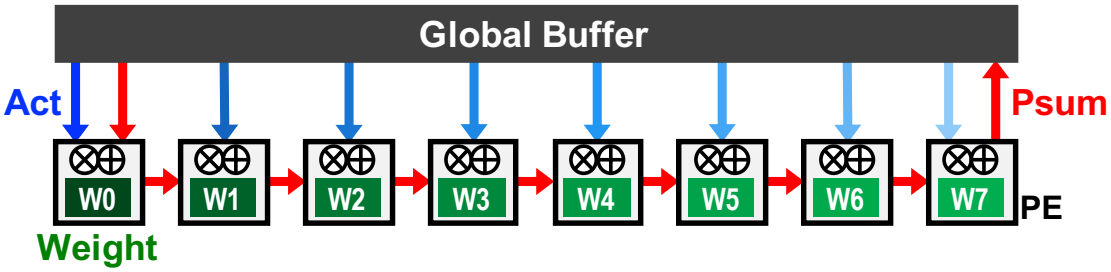
As with any major design choice, leveraging data reuse comes with design trade-offs. For instance, temporal data reuse may motivate complex multi-level memory hierarchies, where additional memory levels may increase area and access latency. Spatial data reuse may motivate a complex network-on-chip connecting memory blocks to large arrays of compute elements, resulting in more interconnect overhead.

Dataflow Taxonomy

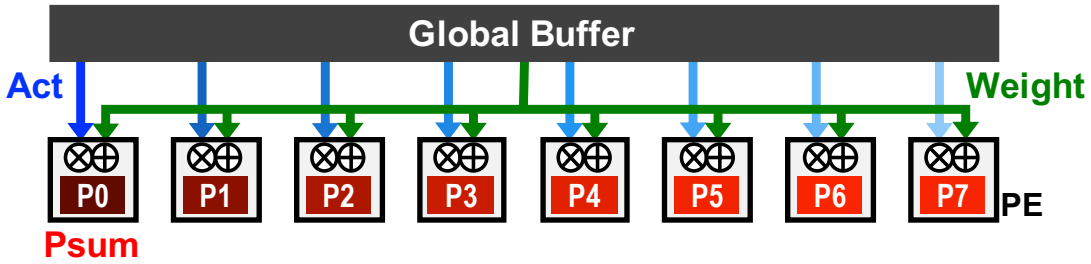
This section summarizes different dataflows that have emerged in recent DNN accelerators [106]. Each dataflow variant seeks to exploit a different data reuse pattern.

The **weight stationary dataflow** (Figure 1-5(a)) aims to minimize the energy cost of reading filter weights by exploiting filter reuse; weights are stored and kept stationary within the processing elements (PEs) that perform MAC operations. Computation of MACs is ordered such that weight values within a PE are reused as much as possible before getting overwritten with new values. An example accelerator that implements the weight stationary dataflow is NVIDIA's Deep Learning Accelerator (NVDLA) [107], where weights are stored within the convolution engine in a dedicated buffer. Microsoft's BrainWave [86] also follows a weight stationary dataflow, using a pinning strategy that keeps model weights in on-chip memory for high read bandwidth. The **output stationary dataflow** (Figure 1-5(b)) aims to minimize the energy cost of partial sum movement. These partial sums are generated as MACs are being computed and aggregated across spatial and channel-wise dimensions. The **input stationary dataflow** (Figure 1-5(c)) exploits input feature map reuse; input activations are stored and kept stationary within PEs, and computation of MACs is ordered such that each activation is maximally reused.

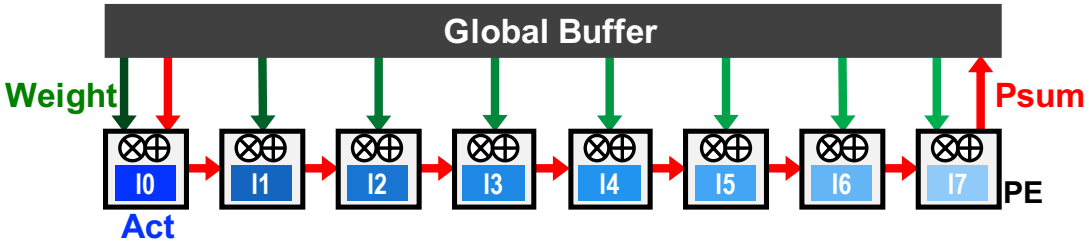
Unlike the above dataflows that cater to a specific datatype, the **row stationary dataflow** instead aims to maximize overall convolutional reuse of all datatypes. Introduced in Eyeriss [106], this dataflow keeps a row of filter weights stationary within a PE and streams a row of input activations through. The PE performs 1D convolution; it computes multiplications over the stored rows and accumulates them locally within the PE. Due to the sliding window nature of convolutions, input activations get reused along with the stationary row of weights. Upon sliding through the entire row of inputs, the PE completes the partial sums for this row. This dataflow therefore maximizes both input feature map and filter reuse as well as localized accumulation of partial sums. It was shown to be more energy-efficient on convolutional layers than the previously described dataflows [106]. Figure 1-6 illustrates convolutional reuse in



(a) weight stationary



(b) output stationary



(c) input stationary

Figure 1-5: Diagrams showing different dataflows. Each dataflow variant aims to exploit data reuse of a different datatype. Figures taken from [1].

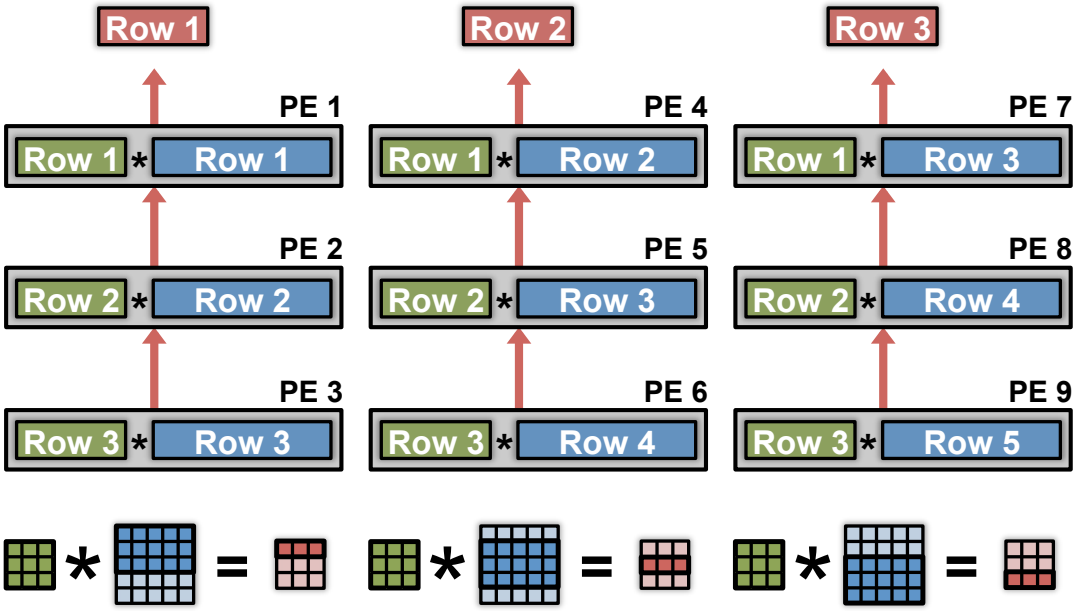


Figure 1-6: The row stationary dataflow aims to maximize overall convolutional reuse of all datatypes. Every processing element (PE) operates on one row of filter weights (reused horizontally through the array) and one row of input activations (reused diagonally through the array). Figure taken from [1].

the row stationary dataflow on a 2D array of processing elements.

Earlier on, accelerators were designed to primarily support a single dataflow. More recently, accelerators have been designed to support multiple dataflows through flexible on-chip networks and configurable switches [108–110]. This allows for an accelerator to toggle between dataflow patterns to better support a particular layer or set of layers within a DNN. However, this often comes at the cost of increased chip complexity and area needed for reconfigurability.

1.4 Thesis Contributions

This thesis explores fast and energy efficient monocular depth estimation on embedded platforms. We particularly focus on learning-based depth estimation algorithms, and investigate techniques to simplify them for real-time low-power inference. The work presented in this thesis can be divided into three contributions.

Our first contribution is a compact DNN design for monocular depth estimation.

We develop a lightweight encoder-decoder architecture incorporating *depthwise separable convolutions* throughout the network as well as additive skip connections passing feature maps from encoding layers to decoding layers. Our fully-convolutional network, FastDepth, is easy to train and achieves close to state-of-the-art accuracy rates on the NYU Depth v2 dataset. It does so while being a fraction of the size of models presented in prior works. This work is discussed in Chapter 2 and appears in [111].

Our second contribution extends the first one to focus on deployment, targeting real-time inference on NVIDIA's Jetson TX2 embedded platform. We demonstrate that to achieve this, further steps in model reduction are necessary. Our methodology for deployment includes *hardware-specific compilation* of network layers for the CPU and GPU onboard the TX2, as well as *channel pruning* to reduce model size with negligible accuracy loss. We show that with these steps, FastDepth can achieve real-time inference at 178 fps on the TX2 GPU and at 27 fps when running only on the CPU, with power consumption of $10-12$ W in both scenarios. This puts FastDepth as being over an order of magnitude faster than prior works yet with comparable accuracy. This work is discussed in Chapter 3 and appears in [111]. Trained models and evaluation code are available at `http://fastdepth.mit.edu/` and `https://github.com/dwofk/fast-depth`.

Our third contribution explores custom hardware design to further push the energy efficiency of FastDepth by aiming to lower power consumption at inference time. We employ a hardware-algorithm co-design approach in which we design an FPGA-based accelerator in conjunction with modifying the FastDepth topology to make it more accelerator-friendly. Our design choices when developing our dataflow and accelerator architecture inform the modifications we make to the FastDepth DNN, and those modifications then motivate additional accelerator features. This co-design approach results in a 21% reduction in data movement of feature maps and parameters and enables high spatial utilization of our accelerator. Our accelerator natively runs depthwise separable layers using a reconfigurable compute engine that supports a heterogeneous dataflow for depthwise and pointwise convolutions. We deploy the accelerator on the Ultra96 SoC and demonstrate end-to-end inference. Processing

time of all layers on the FPGA takes 29 ms with power consumption of around 6 W, pointing to higher energy efficiency than what the original FastDepth network achieved on the TX2 CPU. This work is discussed in Chapter 4.

Chapter 2

FastDepth, a Compact DNN for Monocular Depth Estimation

This chapter introduces our first contribution: a compact deep neural network design for monocular depth estimation.[1] This work is motivated by a rising interest in deploying DNNs and performing inference on edge devices, e.g., on mobile systems, embedded platforms, etc. A more specific example in a robotics context is that of miniaturized robotic vehicles that can traverse narrow spaces and be used in applications such as disaster relief, exploration, and environmental monitoring. Such systems are not only limited in onboard compute resources but are also subject to latency and power constraints. While state-of-the-art learning-based depth estimation algorithms achieve significant improvement in accuracy, they do so at the cost of increased computational complexity and runtime, which makes them unsuitable for small robotic systems. This highlights a key challenge in balancing the computation and runtime cost with the accuracy of the depth estimation algorithm.

In this chapter, we describe our proposed DNN architecture for low-latency depth estimation. We demonstrate that by taking latency into account throughout different DNN design stages, we can achieve depth inference at an accuracy comparable to prior works with a network that is over an order of magnitude smaller and faster.

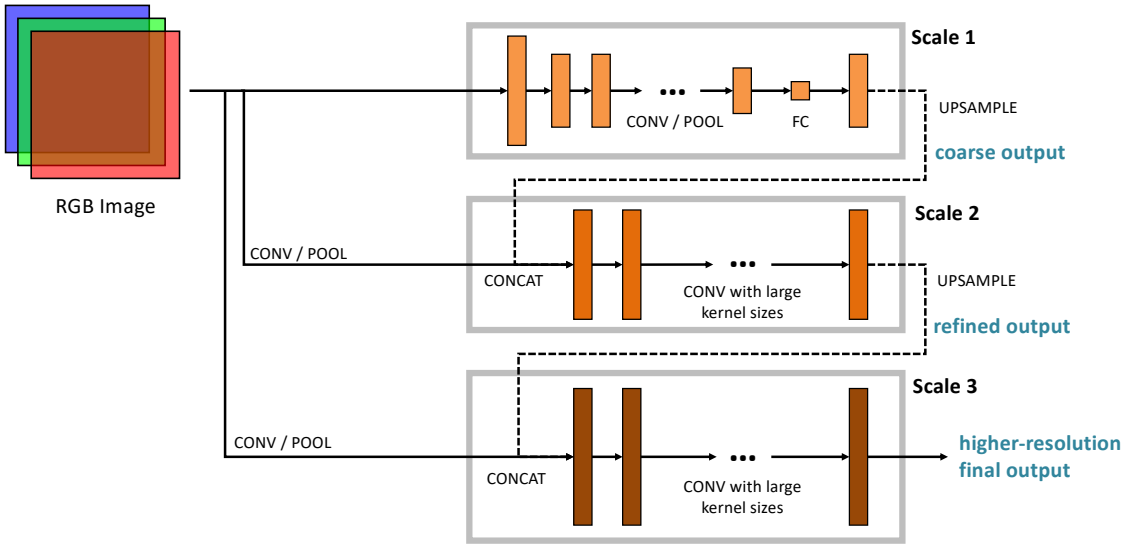
[1] This work was done in collaboration with Fangchang Ma and Tien-Ju Yang, and was supervised by Sertac Karaman and Vivienne Sze. It has been published in [111]. Material presented in this chapter has been adapted from that publication. Project website: `http://fastdepth.mit.edu/`

2.1 Related Work

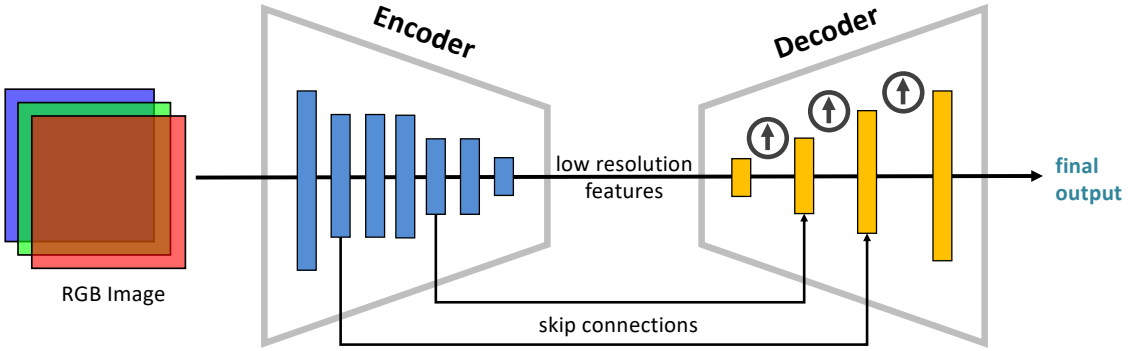
Over the years, a multitude of DNNs have been designed for monocular depth estimation; several of these are surveyed in Section 1.1.2. Here, we narrow down our literature review to the works most relevant to ours and the ones against which our DNN design is evaluated. Eigen et al. [13] proposed one of the earlier monocular depth estimation DNNs, using two stages of convolutional neural networks, with one network predicting depth on a coarse (global) scale and the second network refining depth within local regions. This approach relied on convolutional layers with large kernel sizes and pooling to extract features over large regions of the input image as well as fully-connected layers to extend the field of view to the entire input image. The coarse depth generated by the first stage network was fed into the second stage network as an additional feature map. Eigen et al. improved on this multi-scale convolutional approach in [14] by deepening the network, introducing a third stage to increase output resolution, and allowing multi-channel feature maps to pass between scales instead of a single-channel coarse depth map. The authors explored two options for their first stage network: AlexNet [49] and the much-deeper VGG [46], with VGG contributing to a higher accuracy as well as a larger model size. Figure 2-1(a) illustrates their multi-stage network predicting outputs at multiple scales.

More recent depth estimation DNNs have adopted an encoder-decoder structure [2, 112], shown in Figure 2-1(b). In such architectures, the encoder is responsible for extracting low resolution features from the input, while the decoder upsamples these features and gradually merges them via high-dimensional convolutions into a high-resolution output. The encoder output directly feeds into the decoder, and the two may be interconnected via skip connections between encoding and decoding layers [32, 113, 114]. It is possible for the encoder and decoder to be balanced in terms of complexity, or for one of the two to dominate a model's size or runtime.

Laina et al. [2] developed an encoder-decoder depth estimation DNN based on ResNet-50 [16] that achieved higher accuracy than [13, 14]. ResNet-50 served as their encoder, and the fully-connected layer at the end of the ResNet was replaced with



(a) Multi-stage DNN structure predicting outputs at different scales. Each stage/scale consists of several convolutional layers. The initial stage extracts low resolution features and applies a fully-connected layer to generate a coarse output. Later stages maintain resolution and refine local details such as boundary edges. The output from a given stage is a depth map at varying resolutions that is concatenated with RGB input and fed into the subsequent stage. Figure adapted from [14]. This style of depth estimation DNN was used in [13, 14].



(b) Encoder-decoder DNN structure. The encoder extracts low resolution features from the input, while the decoder gradually upsamples and merges these features through convolutional layers to directly produce a high resolution output. Unlike the multi-stage DNN structure above, the intermediate feature map between the encoder and decoder is not a coarse depth map. However, like the structure above, this one also incorporates multi-scaleness in the form of skip connections, which pass feature maps of varying resolutions from the encoder to the decoder to help refine details in the depth output. This style of depth estimation DNN was used in [2, 112] and is used in our FastDepth work.

Figure 2-1: Examples of two depth estimation DNN structures.

a decoder consisting of cascaded upsampling blocks. The authors defined two types of up-sampling blocks: UpProj and UpConv, with UpProj containing three times as many convolutions but yielding a higher accuracy (see Section 2.5.2 for a discussion of decoder upsampling blocks). Xian et al. [112] also used ResNet-50 as an encoder and incorporated skip connections from the ResNet to their decoder, where feature maps passed along the connections were upsampled and fused.

One trend that can be noticed in these works is using complex encoders (e.g., based on VGG, ResNet) as well as complex decoders (e.g., comprised of blocks with several convolutional layers each) to improve accuracy at the expense of runtime. How both of these can be simplified for lower latency and higher efficiency is still an active research question. Prior research on designing fast and efficient networks has primarily focused on encoder networks for tasks such as image classification and object detection [1]. In these applications, the input is an image (pixel-based), and the output is reduced to a label (an object class and position). To the best of our knowledge, less effort has been put into the efficient design of *both* encoder and decoder networks (i.e., auto-encoder networks) for tasks such as depth estimation, where the output is a dense image of similar resolution as the input. In particular, reducing decoder complexity poses a challenge since there is less information reduction at each of the decoding layers and the decoder's output is high dimensional.

2.2 FastDepth DNN Architecture

Our compact and fast monocular depth estimation DNN — FastDepth — has a fully convolutional architecture with an encoder-decoder structure shown in Figure 2-2. The encoder extracts high-level low-resolution features from the input image. These features are then fed into the decoder, where they are gradually upsampled, refined, and merged to form the final high-resolution output depth map. In developing a depth estimation DNN that can run in real-time, we seek low-latency network designs for both the encoder and the decoder.

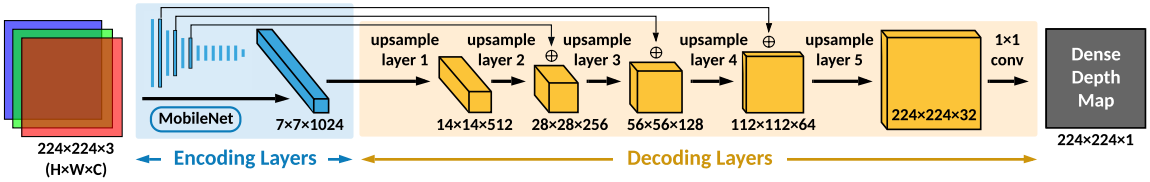


Figure 2-2: Our FastDepth network architecture. The encoder is shown in blue; decoder is shown in yellow. Dimensions of intermediate feature maps are given as height × width × channels. Arrows from encoding layers to decoding layers denote additive (rather than concatenative) skip connections.

2.2.1 Encoder Network

The encoder used in depth estimation DNNs is commonly a network designed for image classification. Commonly used encoder networks include VGG-16 [46] and ResNet-50 [16] for their strong expressive power and high accuracy. However, such networks are also very large. For example, VGG-16 uses 138M weights and performs 15.5G multiply-accumulate operations (MACs), while ResNet-50 uses 25.5M weights and performs 3.9G MACs [1]. The high computational complexity of these networks increases inference latency, thus making them unsuitable for applications running in real-time on embedded systems.

Since our work targets low inference latency, we employ the MobileNet [45] as our encoder of choice. MobileNet makes use of depthwise decomposition, which factorizes an $R{\times}S{\times}C{\times}M$ standard convolutional layer into a depthwise layer with C distinct $R{\times}S{\times}1$ filters and a pointwise layer with M distinct $1{\times}1{\times}C$ filters (illustrated earlier in Figure 1-3). Since each filter in a depthwise layer only convolves with a *single* input channel, the complexity of a depthwise layer is much lower than that of a standard convolutional layer, where each filter convolves with *all* input channels. Moreover, each pointwise filter is just a $1{\times}1$ kernel, so the number of MACs performed by a pointwise layer is $R{\times}S$ times smaller than that of the original standard convolution. Therefore, depthwise decomposition significantly reduces the complexity of a convolutional layer, making MobileNet much smaller overall. MobileNet uses 4.2M parameters and just 0.569G MACs [45]; this translates to reduced inference latency. However, MobileNet still achieves competitive accuracy rates on image classification

tasks (e.g., 70.6% top-1 accuracy on ImageNet [115] compared to 71.5% top-1 accuracy achieved with VGG-16). Its significantly smaller size makes it more efficient than networks with standard convolutional layers like ResNet and VGG. We therefore use MobileNet as the encoder backbone of our depth estimation DNN.

2.2.2 Decoder Network

In an encoder-decoder network structure, the typical objective of the decoder is to merge and upsample the output of the encoder to form a dense prediction. A key design aspect of the decoder is the upsample operation used, e.g., unpooling, transpose convolution, interpolation combined with convolution. Different upsample operations are explored more in the decoder ablation study in Section 2.5.2. The channel reduction factor of each decoding layer, i.e., how much the channel dimension is reduced as the spatial dimensions expand, is another design aspect. These decoder characteristics all factor into the accuracy, computational complexity, and inference latency of the overall neural network.

Our decoder network, which we refer to as NNConv5, consists of five cascading upsample layers and a single pointwise layer at the end. Each upsample layer performs 5×5 convolution and reduces the number of output channels by half relative to the number of input channels. Convolution is then followed by nearest-neighbor interpolation that doubles the spatial resolution of intermediate feature maps. Interpolating *after* convolution instead of before quarters the resolution of feature maps processed by the convolutional layers; this reduces computation in the decoding layers by 1/4 as when interpolating before convolution. We use depthwise separable convolutions to lower the complexity of all convolutional layers, resulting in a fast decoder that has a runtime comparable to the MobileNet encoder.

2.2.3 Skip Connections

Encoder networks tend to be deep and typically contain many layers to gradually reduce spatial resolution, which allows the network to extract higher-level features

from the input. The output of the encoder into the decoder becomes a set of low resolution features in which many image details can be lost, making it more difficult for the decoder to recover pixel-wise (dense) data. Skip connections that carry residual feature maps from encoding layers to decoding layers allow image details from high resolution feature maps in the encoder to be merged into features within the decoder. This helps the decoding layers to reconstruct a more detailed dense output, e.g., with sharper edges. Skip connections have been previously been used in networks for image segmentation such as U-Net [32] and DeeperLab [114], showing that they can be beneficial in improving accuracy of networks producing dense outputs.

In the FastDepth DNN, we incorporate skip connections from the MobileNet encoder to the outputs of the middle three layers in the decoder, as depicted by the arrows in Figure 2-2. Feature maps at the terminating end of the skip connections are combined through addition rather than concatenation; this is to avoid increasing the number of feature map channels processed by the decoding layers.

2.2.4 Layer Types Used

FastDepth is a fully-convolutional neural network, meaning it does not contain any fully-connected layers or pooling layers. Most of its convolutional layers use depthwise separable convolutions. Exceptions to this include the very first layer of the MobileNet encoder (that is a standard convolutional layer) and the very last layer of the decoder (that is simply a pointwise convolution followed by interpolation).

Each convolutional layer in FastDepth is followed by a batch normalization layer and a ReLU function. After training, batch normalization parameters are folded into preceding convolutions, resulting in an neural network topology consisting solely of convolutional layers, ReLU functions, and addition operations for skip connections.

2.3 Training Environment

We implement the FastDepth network in PyTorch [42] and train on the NYU Depth v2 dataset [25] using the official train/test data split. Encoding layers are initialized

with weights from models that have been pretrained on ImageNet [115]. The network is then trained as a whole for 20 epochs with a batch size of 16 and an initial learning rate of 0.01. The learning rate is reduced by a factor of 2 every 5 epochs.

Data Augmentation We follow similar data augmentation steps as the training procedure in [23]. After center-cropping RGB frames to 304×228, we additionally resize the frames to 224×224 to match the input size to our MobileNet encoder.

Loss Function and Optimizer Our loss function is L1 (mean absolute error). Our optimizer is SGD with a momentum of 0.9 and a weight decay of 0.0005.

Error Metrics We evaluate accuracy using two metrics: (1) RMSE, the root mean squared error and (2) δ_1, the percentage of predicted pixels where the relative error is within 25%. Lower RMSE and higher δ_1 values indicate better predictions.

2.4 Post-Training Evaluation and Analysis

This section presents an initial evaluation of our FastDepth network. Evaluated metrics are summarized in Table 2.1. Our evaluation here focuses on accuracy metrics (δ_1 and RMSE) and complexity metrics (number of MACs). However, the MACs count only serves as a coarse first-order estimate of complexity, as it is hardware-agnostic and does not take into account additional complexity factors like the amount of data movement taking place. Latency, power consumption, and energy efficiency are more holistic metrics. While we focus on evaluating those metrics in Chapter 3, we include a preliminary runtime comparison in our evaluation here.

Our target hardware platform is the NVIDIA Jetson TX2 [84]. All models evaluated on the TX2 are run with a batch size of 1 and at 32-bit floating point precision. The TX2 is set to operate in the max-N power mode (see section 3.3.1 on TX2 power consumption for more details).

on NYU Depth v2	**Input Size**	**MACs [G]**	**RMSE [m]**	δ_1	**GPU [ms]**	**CPU [ms]**
Eigen *et al.* [13]	228×304	2.06	0.907	0.611	23	**307**
Eigen *et al.* [14] (w/ AlexNet)	228×304	8.39	0.753	0.697	96	1391
Eigen *et al.* [14] (w/ VGG)	228×304	23.4	0.641	0.769	195	2797
Laina *et al.* [2] (w/ UpConv)	228×304	22.9	0.604	0.789	237	2384
Laina *et al.* [2] (w/ UpProj)	228×304	42.7	**0.573**	**0.811**	319	3298
Xian *et al.* [112]	384×384	61.8	0.660	0.781	283	4429
Ours (FastDepth)	224×224	**0.74**	0.599	0.775	**19**	5100

Table 2.1: Comparing FastDepth against prior work. For δ_1, higher is better. For all others, lower is better. Statistics for cited works come from our re-implemented models. Reported runtimes are measured in PyTorch on an NVIDIA Jetson TX2. Our network design achieves close to state-of-the-art accuracy with a significant reduction in MACs and GPU runtime.

Accuracy and Complexity Our FastDepth DNN achieves both δ_1 and RMSE metrics that are comparable to prior works. Although our δ_1 accuracy is almost 4% lower than that achieved by Laina et al. [2] with their UpProj decoder, our RMSE (arguably a more intuitive metric) is within only 3 cm of theirs. At the same time, FastDepth is far less computationally costly, with a reduction in MACs of $30-80\times$ when compared with models yielding similar or higher accuracy rates. This reduction is made possible through our exploration of the encoder and decoder design spaces as described by our ablation studies in Section 2.5.

Runtime on the Jetson TX2 The runtimes evaluated here are all obtained by running models directly in PyTorch. FastDepth surpasses real-time inference speeds on the TX2 GPU and runs faster than prior works. There are, however, observed inefficiencies in speed on the TX2 CPU. This is largely due to PyTorch's under-optimized CPU execution of depthwise separable layers that are used heavily throughout FastDepth.[2] During inference on the GPU, we enable cuDNN (a library of CUDA-compatible GPU-accelerated operation primitives) in PyTorch; this is successfully

[2]Depthwise separable layers offer a reduction in MACs over standard convolutional layers. However, MACs are not always representative of real-world performance, e.g., latency. What we observe here is an example of that — a reduction in MACs achieved by using depthwise separable layers is not translating to lower runtime on the TX2 CPU. This highlights the importance of using direct metrics such as latency measurements to gauge the impact of DNN design choices.

Encoder	Weights [M]	MACs [G]	RMSE [meters]	δ_1	CPU [ms]	GPU [ms]
ResNet-50	25.6	4.19	**0.568**	**0.800**	610	35.0
ResNet-18	11.7	1.84	**0.568**	0.782	**220**	15.2
MobileNet	**3.19**	**0.57**	0.579	0.772	3700	**8.7**

Table 2.2: Comparison of encoder variants in our ablation study. RMSE and δ_1 are for encoder-decoder networks with the decoder fixed as NNConv5. All other metrics are for the encoder in isolation. Runtimes are measured in PyTorch on a TX2. We select MobileNet as the best encoder option.

accelerating FastDepth layers. However, during inference on the CPU, no CPU-equivalent to cuDNN is enabled. To remedy the observed inefficiencies, we need to perform hardware-specific compilation; this is discussed further in Chapter 3.

2.5 Ablation Studies

This section presents ablation studies for the major components that make up the FastDepth network design. We discuss the encoder and decoder design spaces and how our design choices impact the latency of our depth estimation network.

2.5.1 Encoder Design Space

A common encoder used in existing high-accuracy DNNs [2, 112] is ResNet-50 [16]. Targeting lower encoder latency, we consider the smaller ResNet-18 and MobileNet [45] as alternatives to ResNet-50. The last average pooling layer and fully connected layers are removed from the MobileNet and ResNet architectures, since the output from the last convolutional layer in those networks will feed into the decoder. Furthermore, to make the encoders compatible with a fixed decoder structure, we append a 1×1 convolutional layer to the end of both ResNet encoders, so that the output from all encoder variants has a consistent shape of 7×7 with 1024 channels.

We compare all three encoder options against each other in Table 2.2. We pair each encoder with a fixed NNConv5 decoder and train that network as a whole. The reported runtimes are obtained by running the networks in PyTorch. Runtimes for

ResNet-50 and ResNet-18 are too high, even on the TX2 GPU, to achieve real-time speeds above 25 fps if these encoders are paired with decoders of similar latency. In comparison, MobileNet efficiently trades off between accuracy and latency, and has a noticeably lower GPU runtime. We therefore select MobileNet as our encoder.

We note that despite its lower complexity, MobileNet is an order of magnitude slower on the TX2 CPU than ResNet-18. This can be attributed to as-of-yet unoptimized low-level CPU execution of depthwise layers in deep learning frameworks and can be remedied through hardware-specific compilation [99].

2.5.2 Decoder Design Space

While encoders have been well characterized in deep learning research, decoders have been less extensively explored, especially in the context of efficient DNN design. We consider two decoder aspects: upsample operation and depthwise decomposition.

Upsample Operation

We survey four ways of upsampling in the decoder. Their characteristics are listed below, and visual representations are shown in Figure 2-3:

1. **UpProj** is explored as a decoder building block in [2]. It consists of 2×2 unpooling (zero-insertion) followed by a two-branched residual structure that computes a total of three convolutions (two 5×5 and one 3×3).

2. **UpConv** is also explored in [2]. It consists of 2×2 unpooling (zero-insertion) followed by a single 5×5 convolution.

3. **DeConv5** refers to transpose convolution using a 5×5 kernel.[3]

4. **NNConv5** refers to 5×5 convolution followed by nearest-neighbor interpolation[4] with a scale factor of 2.

[3]Sometimes also called deconvolution. We use a kernel size of 5 to fairly compare against UpConv.

[4]An alternate option would be using bilinear interpolation. However, we select nearest-neighbor interpolation as is it a simpler operation with more consistent implementations across different deep learning frameworks and compilers.

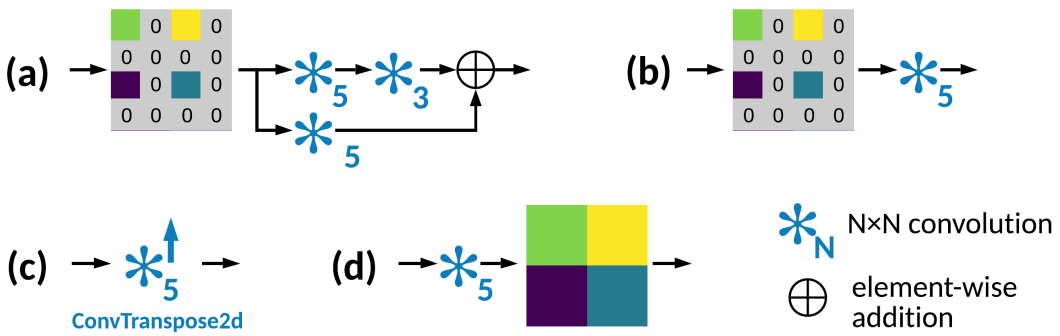


Figure 2-3: Visual representations of different upsample operations we consider for decoders: (a) UpProj [2], (b) UpConv [2], (c) DeConv5, (d) NNConv5.

Decoder	Weights [M]	MACs [G]	RMSE [meters]	δ_1	CPU [ms]	GPU [ms]
(a) UpProj [2]	38.1	28.0	0.599	**0.774**	3300	325
(b) UpConv [2]	**17.5**	12.9	0.591	0.771	1600	238
(c) DeConv5	**17.5**	12.9	0.596	0.766	**290**	31.0
(d) NNConv5	**17.5**	**3.21**	**0.579**	0.772	410	**26.2**

Table 2.3: Comparison of decoder variants in our ablation study. RMSE and δ_1 are for encoder-decoder networks with a MobileNet encoder. All other metrics are for the decoder in isolation. Runtimes are measured in PyTorch on a TX2. We select NNConv5 as the best decoder option.

We implement four decoder variants using these upsample operations, keeping the structure fixed at 5 decoding layers with 1×1 convolution at the end. Table 2.3 compares the four decoders. UpProj is most complex, due to its larger number of convolutions per upsample layer. It achieves the highest δ_1 accuracy but is the slowest. UpConv is less complex and faster than UpProj, but its CPU and GPU runtimes are still too slow for real-time processing. DeConv5 has an identical number of weights and MACs as UpConv and is noticeably faster on both the CPU and GPU. However, it can be prone to introducing checkerboard artifacts in its outputs [116], which helps explain its lower accuracy. NNConv5 achieves higher δ_1 accuracy and lower RMSE than both UpConv and DeConv5, with a slightly lower GPU runtime. We therefore select NNConv5 as our decoder.

MobileNet-NNConv5	Weights [M]	MACs [G]	RMSE [meters]	δ_1	CPU [ms]	GPU [ms]
with standard decoder	20.6	3.78	**0.579**	0.772	**4100**	34.9
with depthwise decomposition	**3.93**	**0.74**	0.584	0.767	5200	**18.6**
with depthwise decomposition & concatenative skip connections	3.99	0.85	0.601	**0.776**	5500	26.8
with depthwise decomposition & additive skip connections	**3.93**	**0.74**	0.599	0.775	5100	19.1

Table 2.4: Impact of depthwise decomposition and skip connections in the decoder on network complexity and TX2 runtime.

Depthwise Separable Convolution

After selecting MobileNet as our encoder and NNConv5 as our decoder, we observe that the runtime of our network is dominated by the decoder. From Table 2.2 and Table 2.3, we see that a MobileNet encoder takes 8.7 ms to run on the TX2 GPU, while the NNConv5 decoder takes 26.2 ms — 3 times as long as the encoder. This motivates us to simplify our decoder even further. Similar to how depthwise decomposition lowers the complexity and latency in MobileNet, we now replace all convolutions within the decoder with depthwise separable convolutions.

Table 2.4 shows that depthwise decomposition in the decoder lowers inference runtime on the GPU by almost half. It also reduces the runtime contribution of the decoder to the entire model runtime. In our encoder ablation study, we report that the MobileNet encoder runs in 8.7 ms on the GPU. Here, this implies that a standard NNConv5 decoder accounts for 75% of entire model runtime. However, with depthwise decomposition, the decoder accounts for just 53% of entire model runtime. This shows that incorporating depthwise decomposition is instrumental in helping lowering the decoder runtime to better balance with the encoder runtime.

In contrast to the runtime reduction on the GPU, runtime on the CPU increases, despite the reduced number of MACs; as mentioned earlier, this is due to the inefficient CPU execution of depthwise separable layers in PyTorch. Like with MobileNet, depthwise decomposition in the decoder results in a slight accuracy loss, due to the reduction in trainable parameters and computation.

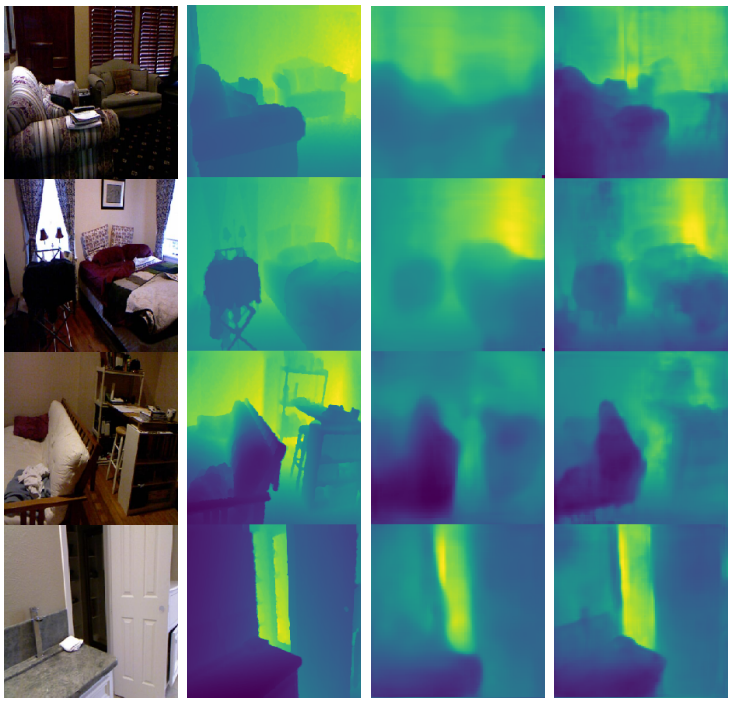


Figure 2-4: Visualized results of depth estimation on the NYU Depth v2 dataset after training. (a) input RGB image; (b) ground truth; (c) our model, FastDepth, without skip connections; (d) our model, FastDepth, with skip connections.

2.5.3 Skip Connections

We consider both additive and concatenative skip connections. Concatenative skip connections increase the computational complexity of the decoder since decoding layers need to process feature maps with more channels. Table 2.4 shows that this improves the δ_1 accuracy but also noticeably increases CPU and GPU runtimes. In contrast, using additive skip connections leaves the number of channels in the decoder unchanged and has a negligible impact on inference runtime while achieving almost the same accuracy boost. We therefore use additive skip connections in our final network design. As shown in Figure 2-4(d), skip connections noticeably improve the sharpness and visual clarity of the depth maps output by our network design.

2.6 Summary

This chapter introduces our first contribution, a compact deep neural network for monocular depth estimation. Our FastDepth DNN uses a lightweight encoder-decoder architecture that achieves depth inference accuracy on par with prior work at only a fraction of the model size and runtime. As explained in our ablation studies, FastDepth uses a low-complexity and low-latency decoder that does not dominate computation or runtime even when combined with a small MobileNet encoder. This balanced network design contrasts with prior work [2] featuring a deep network with a complex decoder that dominates computation and runtime. Key design choices that allow us to achieve low decoder latency include performing fewer convolutions per upsampling operation as well as incorporating depthwise separable convolutions. Additive skip connections from the encoder to the decoder help boost network accuracy without increasing decoder computation or runtime.

The results and analysis in this chapter have focused on evaluating FastDepth on basis of its computational cost (MACs) and accuracy. We additionally offer preliminary inference runtime estimates on an NVIDIA Jetson TX2. Though our FastDepth DNN achieves over 50 fps on the TX2 GPU — easily surpassing real-time inference, its performance on the CPU is observed to be orders of magnitude slower than an

acceptable real-time range of 25−30 fps. This is due to execution inefficiencies on specific hardware (in this case, an ARM CPU) rather than the DNN architecture itself, which motivates additional DNN deployment steps that will be described next.

Chapter 3

Real-Time Depth Inference on an Embedded CPU/GPU

This chapter extends our work on FastDepth described in the previous chapter, now with a focus on deployment and attaining real-time inference speeds on an embedded GPU as well as CPU.[1] Two challenges in achieving fast inference speeds with our neural network have been simplifying the network sufficiently enough without sacrificing accuracy as well as ensuring that those simplifications translate to reduced runtime. In this chapter, we discuss the steps we take in addressing these challenges and reducing FastDepth inference latency: (1) hardware-specific compilation to optimize FastDepth layers for our target platform, and (2) network simplification (pruning) to reduce overall network computation without degrading accuracy. We analyse the impact of these steps and present an updated evaluation against prior works.

3.1 Hardware-Specific DNN Compilation

Our proposed network architecture is fully convolutional and makes use of depthwise decomposition in both the encoder and the decoder. In commonly-used deep learning

[1]This work was done in collaboration with Fangchang Ma and Tien-Ju Yang, and was supervised by Sertac Karaman and Vivienne Sze. It has been published in [111]. Material presented in this chapter has been adapted from that publication. Project website: `http://fastdepth.mit.edu/` Trained models and evaluation code available at: `https://github.com/dwofk/fast-depth`

frameworks, depthwise separable layers have not yet been fully optimized for fast runtime on edge devices, e.g., on embedded ARM CPUs [98, 99]. As a result, although depthwise decomposition significantly reduces the number of MACs in a network, a similar reduction may not be observed in inference latency. The left portion of Table 3.1 highlights exactly this: the TX2 CPU runtime of MobileNet-NNConv5 in PyTorch is high, due to the prevalence of depthwise layers in MobileNet, and it increases when we incorporates depthwise layers in the decoder. To address the observed runtime inefficiencies of depthwise layers, we use the TVM compiler stack [98]. TVM performs hardware-specific scheduling and operator tuning that allows the impact of reduced operations to be translated into reduced processing time.

	in PyTorch		**using TVM**	
MobileNet-NNConv5	**CPU [ms]**	**GPU [ms]**	**CPU [ms]**	**GPU [ms]**
with standard decoder	**4100**	34.9	176	20.9
with depthwise decomposition	5200	**18.6**	**50**	8.3
with depthwise decomposition & additive skip connections	5100	19.1	66	**8.2**

Table 3.1: Hardware-specific compilation enables inference speedup on both the CPU and GPU when incorporating depthwise separable layers in our network. Additive skip connections do not add noticeable runtime overhead after compilation, as is expected. All runtimes are measured on the Jetson TX2.

During compilation of FastDepth with TVM, every layer in the network is individually tuned for optimal performance on the specified target platform — in our case, the TX2 GPU or CPU. Tuning a layer involves searching and optimizing within a configuration space defining the execution of the operations within that layer; such configurations include tiling factors, vectorization, unrolling, etc. Layer tuning also involves additional optimization steps such as operator fusion to reduce memory accesses between operations, constant folding to avoid static computations that can be precomputed, and memory latency hiding. The length of the tuning process impacts how well a layer is tuned, i.e., tuning for longer allows for a larger search and optimization space, potentially resulting in a better optimization. We find that for FastDepth layers, 1000 trials are enough for optimizations to converge. The result of the tuning

process is a log with tuned settings for the different operators (e.g., multiplications, additions, interpolations) present in each layer. This log is then queried at inference time, resulting in faster layer execution. Both standard and depthwise convolutional layers benefit from being tuned. The right portion of Table 3.1 reports TX2 runtimes after compilation with TVM. Depthwise decomposition in the decoder now reduces GPU runtime by 2.5× and CPU runtime by 3.5×.

3.2 DNN Simplification through Pruning

To reduce network latency even further, we perform post-training network pruning using NetAdapt [4]. Starting from a trained network, NetAdapt automatically and iteratively identifies and removes redundant channels from feature maps to reduce parameters and computation within layers. In each iteration, NetAdapt generates a set of network proposals simplified from a reference network. Each network proposal is retrained for a short period of time (i.e., short-term fine tuning), and the network proposal with the best accuracy-complexity trade-off is then chosen as the reference network for the next iteration. This process continues until the target accuracy or complexity is achieved. Network complexity can be gauged by indirect metrics (e.g., number of MACs) or direct metrics (e.g., latency on a target hardware platform).

When applying NetAdapt to FastDepth, we use the number of MACs as the guiding complexity metric. Our pruning process begins with an initial reduction of 0.01G MACs and decays at a rate of 0.98 with each pruning iteration. When performing short-term fine tuning, we use a learning rate of 0.001. Figure 3-1 shows our FastDepth architecture shape after pruning. The colored bars illustrate the effect of channel pruning. Compared to the original FastDepth shape shown in the background in grey, channels in the middle of the network (i.e., the second half of the MobileNet encoder, and the encoder-decoder boundary) are pruned away the most. Two bottleneck regions, where layer parameters and MACs are fewer than in neighboring layers, appear in the pruned network: one in the encoder (around layer `mobilenet.9`) and one in the decoder (around layer `decoder.2`).

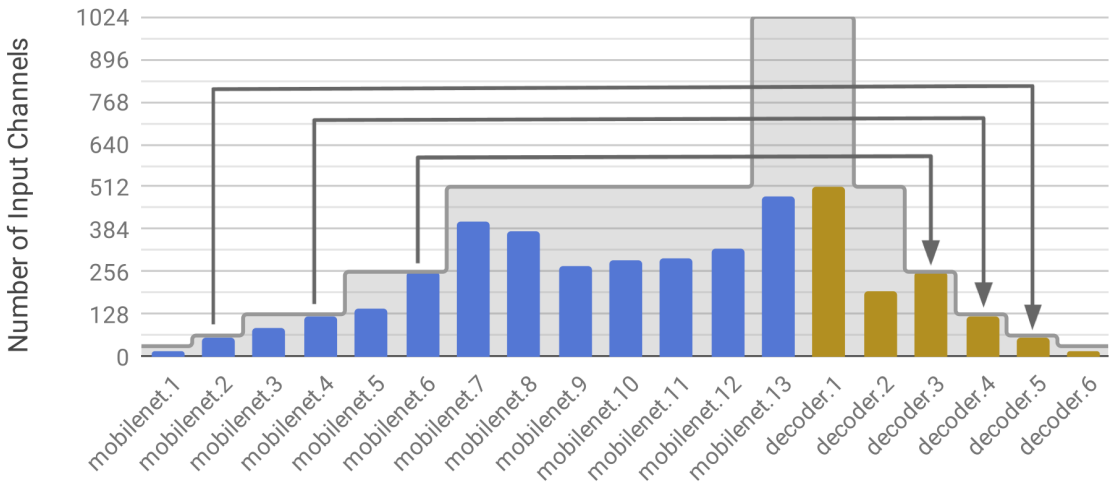


Figure 3-1: Number of input channels to each layer in our network architecture after pruning. The shaded part represents the architecture before pruning. The very first layer to the network (`mobilenet.0`) is not shown since the channel size of the input fed into the network remains fixed at 3 channels (RGB).

Prior to pruning, our compiled FastDepth network already surpasses real-time throughput on the TX2 GPU but does not yet achieve real-time throughput on the TX2 CPU. Pruning with NetAdapt lowers the FastDepth model runtime and increases CPU framerate to between 25 and 30 fps, which is far more suitable for real-time inference. As shown in Table 3.2, pruning achieves a 2× reduction in MACs, a 1.5× reduction in GPU runtime, and a 1.8× reduction in GPU runtime, with almost the same accuracy. Figure 3-2(e) shows that the pruning process preserves the sharpness and visual clarity of output depth maps.

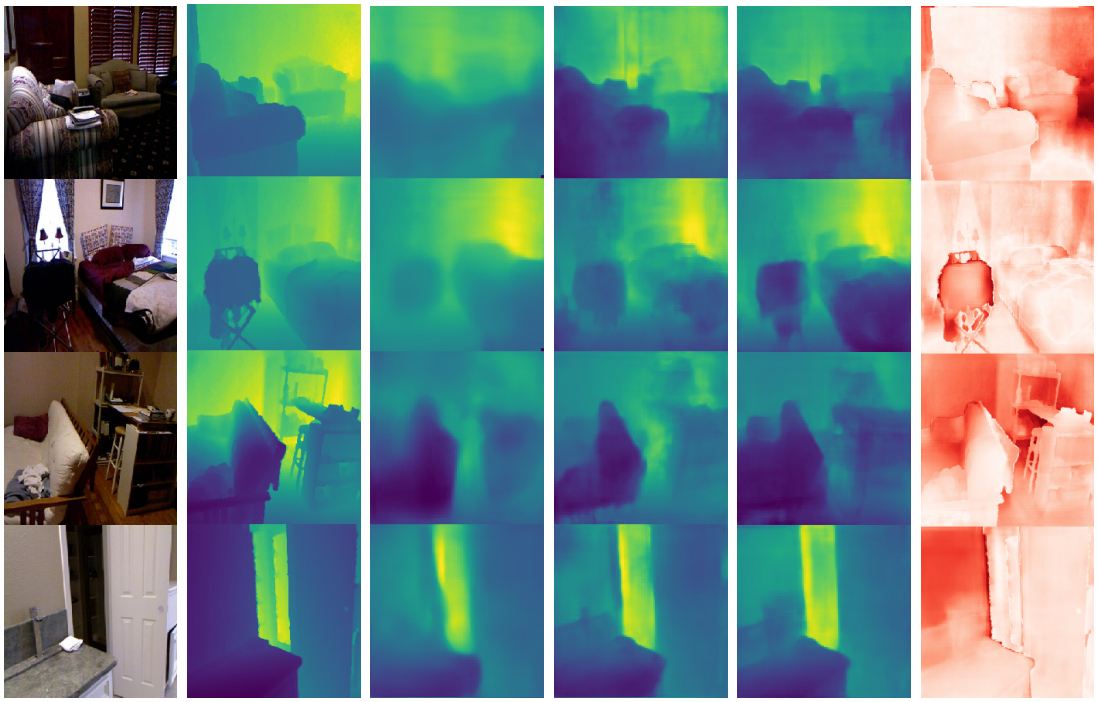
3.3 Post-Compilation Evaluation on the Jetson TX2

This section presents an updated evaluation of our FastDepth network after compilation and pruning. Updated metrics are summarized in Table 3.3.

Accuracy The steps we take in optimizing FastDepth for deployment on the TX2 have negligible effect on model accuracy. Network pruning does result in a slight accuracy loss that is mostly restored in retraining; the pruned FastDepth model

	Before Pruning	After Pruning	Reduction
Weights	3.93M	1.34M	2.9×
MACs	0.74G	0.37G	2.0×
RMSE	0.599	0.604	-
δ_1	0.775	0.771	-
CPU [ms]	66	37	1.8×
GPU [ms]	8.2	5.6	1.5×

Table 3.2: Impact of pruning on our encoder-decoder network. Pruning together with compilation enable real-time inference throughput on the CPU at 27 fps and further increase throughput on the GPU to 178 fps. Reported runtimes are measured after compilation for the Jetson TX2.



(a) (b) (c) (d) (e) (f)

Figure 3-2: Visualized results of depth estimation on the NYU Depth v2 dataset, now including results from FastDepth after compilation and pruning. (a) input RGB image; (b) ground truth; (c) our model, without skip connections, unpruned; (d) our model, with skip connections, unpruned; (e) our model, with skip connections, pruned; (f) error map between the output of our final pruned model and ground truth, where redder regions indicate higher error.

on NYU Depth v2	**Input Size**	**MACs [G]**	**RMSE [m]**	δ_1	**GPU [ms]**	**CPU [ms]**
Eigen *et al.* [13]	228×304	2.06	0.907	0.611	23	307
Eigen *et al.* [14] (w/ AlexNet)	228×304	8.39	0.753	0.697	96	1391
Eigen *et al.* [14] (w/ VGG)	228×304	23.4	0.641	0.769	195	2797
Laina *et al.* [2] (w/ UpConv)	228×304	22.9	0.604	0.789	237	2384
Laina *et al.* [2] (w/ UpProj)	228×304	42.7	**0.573**	**0.811**	319	3298
Xian *et al.* [112]	384×384	61.8	0.660	0.781	283	4429
Ours (FastDepth)	224×224	**0.37**	0.604	0.771	**5.6**	**37**

Table 3.3: Comparing our pruned and compiled FastDepth network against prior work. For δ_1, higher is better. For all others, lower is better. Statistics for cited works come from our re-implemented models. Reported runtimes are measured in PyTorch on an NVIDIA Jetson TX2 in max-N mode. Our final network design surpasses real-time inference speeds on both the GPU and CPU. Overall, FastDepth achieves close to state-of-the-art accuracy while running an order of magnitude faster.

achieves δ_1 and RMSE metrics that are within 1% of those achieved prior to pruning. Hardware-specific compilation yields a model that produces depth outputs identical to those obtained prior to compilation; there is no observed accuracy loss with compilation. With these two steps, our pruned and compiled FastDepth model still achieves competitive accuracy rates when compared to prior works.

Complexity As shown earlier in Table 3.2, network pruning allows us to reduce the number of MACs in FastDepth by half. This drives our model complexity even lower, and when compared against prior works, our model can now be up to $1-2$ orders of magnitude less complex yet with comparable accuracy.

Runtime Network pruning and compilation both significantly contribute to a reduction in FastDepth runtime on the TX2. The reduction is more prominent for the CPU, owing to the high inefficiencies of depthwise separable layers originally observed when running in PyTorch on the TX2 CPU. Our pruned and compiled model now achieves real-time inference on *both* the TX2 GPU and CPU and runs over an order of magnitude faster than prior works. Figure 3-3 shows our model on the far right of an accuracy vs. framerate curve, indicating a better tradeoff than prior works.

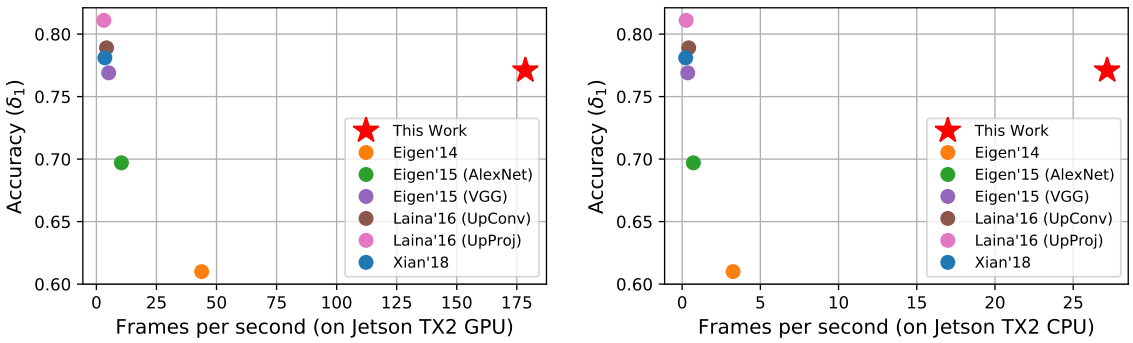


Figure 3-3: Accuracy vs. framerate plot comparing FastDepth against prior works. Our network is to the far right of the curve, indicating a better performance tradeoff.

TX2 Mode	**GPU Frequency**	**Denver 2 Cores in Use (at Frequency)**	**ARM A57 Cores in Use (at Frequency)**
Max-N	1.30 GHz	2 (at 2.0 GHz)	4 (at 2.0 GHz)
Max-Q	0.85 GHz	None	4 (at 1.2 GHz)
Max-P Core-All	1.12 GHz	2 (at 1.4 GHz)	4 (at 1.4 GHz)
Max-P ARM	1.12 GHz	None	4 (at 2.0 GHz)
Max-P Denver	1.12 GHz	1 (at 2.0 GHz)	1 (at 2.0 GHz)

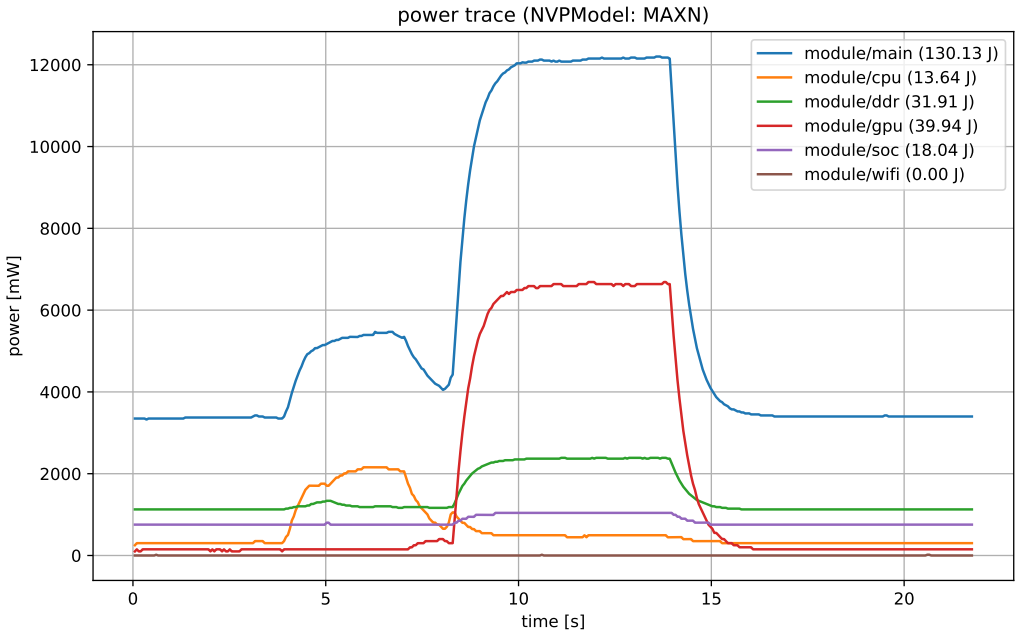
Table 3.4: Summary of NVIDIA Jetson TX2 power modes, taken from [6]. Max-N mode allows for the highest performance (throughput) at the cost of higher power consumption. Max-Q mode aims to provide the best power-throughput tradeoff.

3.3.1 TX2 Power Consumption Modes

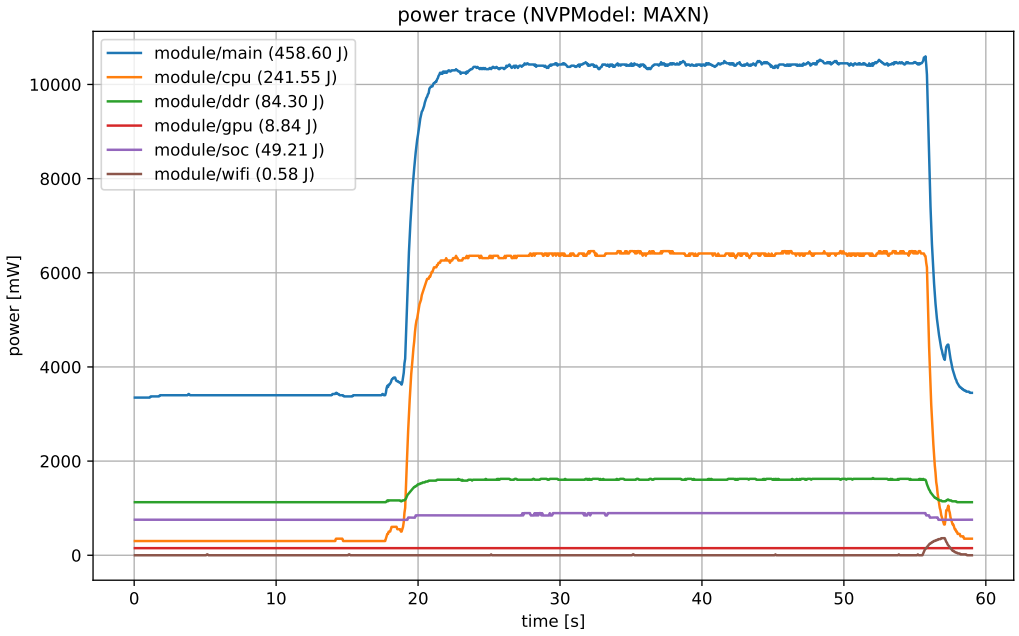
The Jetson TX2 supports several different power configurations. The GPU and the six CPU cores onboard the TX2 can be clocked at different frequencies depending on which power mode the TX2 is set to operate in [6]. These modes are summarized in Table 3.4. When evaluating FastDepth, we set the TX2 to run in max-N mode.

Figure 3-4 shows power consumption traces for FastDepth on the TX2 GPU and CPU, recorded during a test run of 1000 inference trials. The idle power consumption of the TX2 is observed to be around 3.8 W. When running FastDepth on the GPU, power rises to 12.2 W. When running only on the CPU, power rises to a bit less, 10.5 W. If we consider active power consumption, i.e., subtracting away idle power from total power, we estimate that FastDepth requires under 10 W of power.

The max-N mode used for evaluation is the highest performance mode and thus



(a) Power consumption measured over 1000 inference trials on the TX2 GPU



(b) Power consumption measured over 1000 inference trials on the TX2 CPU

Figure 3-4: Power consumption over time when running FastDepth inference on the Jeston TX2. Code used to generate power traces sourced from [3].

Platform	Runtime	Max Framerate	Power Consumption
TX2 GPU (max-N)	5.6 ms	178 fps	12.2 W (3.4 W idle)
TX2 GPU (max-Q)	8.2 ms	120 fps	6.5 W (1.9 W idle)
TX2 CPU (max-N)	37 ms	27 fps	10.5 W (3.4 W idle)
TX2 CPU (max-Q)	64 ms	15 fps	3.8 W (1.9 W idle)

Table 3.5: Inference runtime and peak power consumption when deploying FastDepth on the Jetson TX2 in high performance (max-N) and high efficiency (max-Q) modes. Active power consumption can be estimated by subtracting the idle power consumption from the reported total power consumption. In both power modes, FastDepth consumes less than 10 W of active power.

likely to consume the most power of the various TX2 power modes. As an alternative, we also consider the max-Q mode, which tries to balance throughput and power. It clocks the GPU at a slower clock frequency and disables 2 of the 6 available CPU cores. Table 3.5 compares FastDepth performance across these two power modes. As expected, running in max-Q mode consumes less power, roughly half of that consumed when running in max-N mode. However, inference speed is also lowered; while GPU framerates still exceed real-time at 120 fps, CPU framerates drop to 15 fps.

3.4 Live Depth Inference on an Apple iPhone

In addition to deploying FastDepth on the Jetson TX2, we develop a live demonstration on a mobile phone. In recent years, mobile phone systems have been developed with capabilities to run deep learning models locally. Examples of on-device hardware dedicated to neural network processing include Qualcomm's AI Engine on Snapdragon [88] and Apple's Neural Engine on iPhones [87]. We deploy FastDepth on two iPhone devices: an older iPhone 6S and a newer iPhone X.

We use CoreML, the machine learning framework used across Apple products, to run FastDepth on the iPhones. Since our DNN is defined and trained in PyTorch, we need to first convert it into a CoreML model; we do so using the ONNX open source format for interoperability across frameworks. The FastDepth CoreML model amounts to 5.5 MB, and we we then integrate it into an iOS application that can be downloaded and run on the iPhone. The application connects to the iPhone's

camera that captures live 1080p color video; each frame is center-cropped and resized to produce a stream of 224×224 RGB images. The images are then sequentially run through the FastDepth model stored within the app. Output depth is visualized on-screen in grayscale, as shown in Figure 3-5. Depth could also be visualized using a color gradient by integrating OpenCV functions into the app; however, we found that the functions incurs significant runtime overhead.

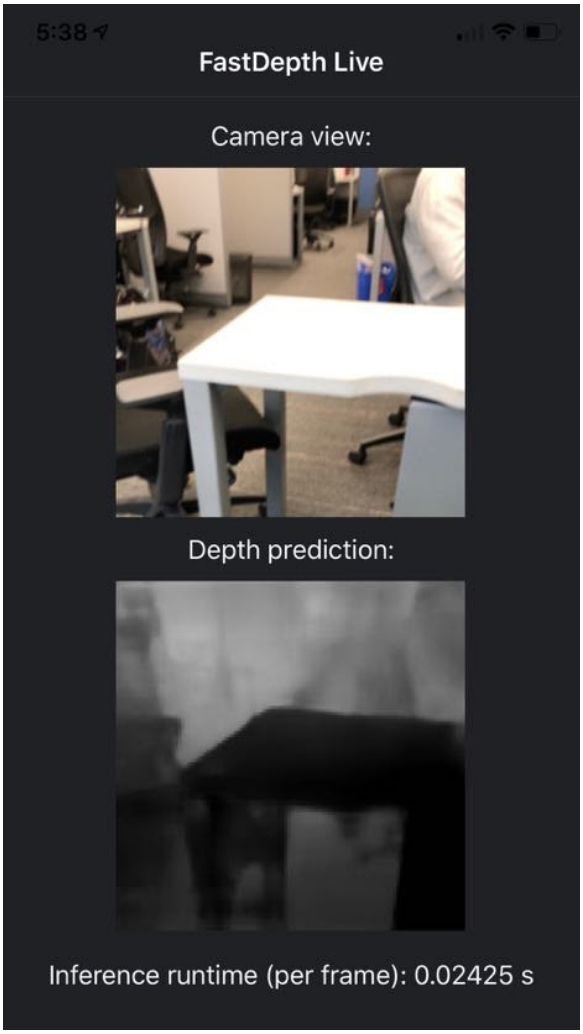


Figure 3-5: Our FastDepth CoreML model running live at 40 fps on iPhone X. Demo video available at `http://fastdepth.mit.edu/`.

Table 3.6 compares the hardware in our two iPhone devices as well as the performance of FastDepth on those devices. We achieve real-time inference at or above 25 fps on both devices. The iPhone X sees an inference speedup of 1.6× relative

to the iPhone 6S. It is reasonable to assume that both devices are operating under 5W, though power consumption estimates or proxies, e.g., the thermal design point (TDP), are not easily ascertained for iPhone devices.[2]

Device	Hardware Description	FastDepth Runtime
iPhone 6S (A9 SoC)	16−14 nm process 1.85 GHz dual-core CPU custom 650 MHz PowerVR GPU 2GB LPDDR RAM	∼0.040 s per image (up to 25 fps)
iPhone X (A11 Bionic SoC)	10 nm FinFET process 2.39 GHz hexa-core CPU Apple 3-core GPU and Neural Engine 3GB LPDDR4X RAM	∼0.025 s per image (up to 40 fps)

Table 3.6: Inference runtime of our FastDepth CoreML model on Apple iPhone.

3.5 Summary

This chapter discusses our second contribution, mainly concerning the deployment of FastDepth for real-time inference on the Jetson TX2. In order to address the runtime inefficiencies of running FastDepth directly in PyTorch onboard the TX2, we apply two state-of-the-art techniques to realize inference speedup. One focuses on hardware-specific compilation to ensure that reductions in MACs, e.g., in depthwise separable layers, translate to reduced runtime on hardware [98]. The other performs iterative channel pruning within network layers to reduce model size at minimum accuracy loss. With these two deployment steps, FastDepth successfully achieves a real-time throughput of 178 fps on the TX2 GPU and 27 fps on the TX2 CPU, with depth accuracy still on par with prior works.

To summarize this second contribution alongside our first one discussed in Chapter 2, we illustrate the GPU runtime progression across our design steps along with an encoder-decoder breakdown in Figure 3-6. We treat ResNet-50 with UpProj [2] as

[2]Online sources have not been definitive or explicit. Perhaps a more reliable option would be to use the Instruments app in the Xcode iOS development environment to profile energy consumption of the FastDepth demo application.

a baseline network, since that work has been well-cited and achieves the highest accuracy out of the works we evaluate against. However, we slightly modify and retrain this baseline network with a 224×224 (instead of 228×304) input and five (instead of four) upsample layers in the decoder. This is done for the baseline to better match our FastDepth DNN topology and allow for a more fair comparison. Consequently, the accuracy and runtime reported in this figure differ from those reported in our evaluation tables.

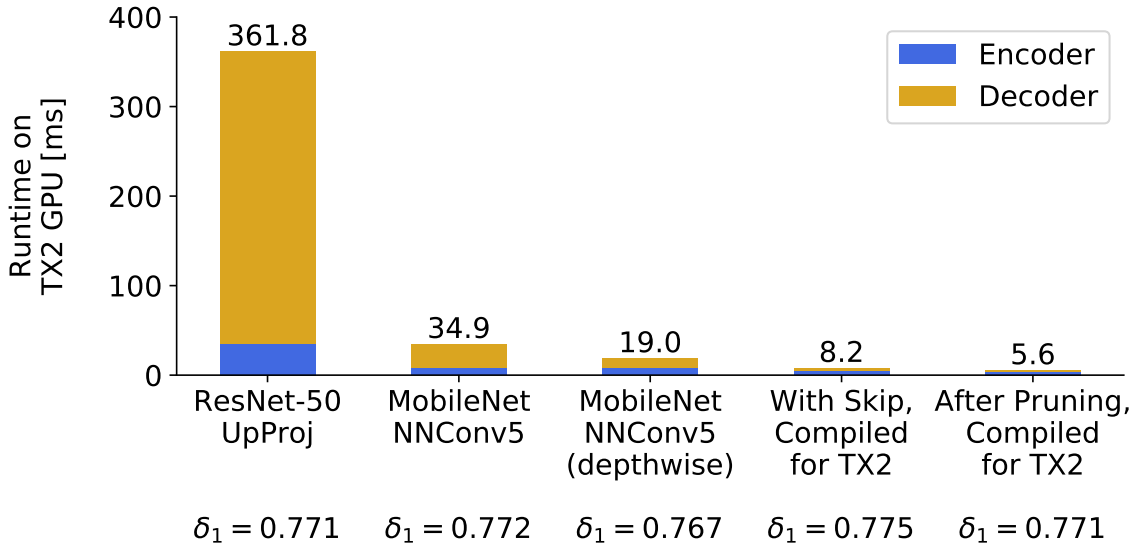


Figure 3-6: Reduction in inference runtime on the TX2 achieved with different steps our approach. Stacked bars represent encoder-decoder breakdown; total runtimes are listed above the bars. The row of δ_1 accuracies listed at the bottom shows the impact of individual steps in our approach on accuracy. Relative to ResNet-50 with UpProj, our final model achieves 65× speedup while maintaining accuracy.

One immediate observation that can be made is the imblanace in encoder vs. decoder runtime; the decoder dominates runtime in the baseline network. It also dominates in our own initial MobileNet-NNConv5 network. It is only after we incorporate depthwise decomposition in the decoding layers that the decoder runtime begins to match that of the encoder. Furthermore, the two deployment steps we peform — compilation and pruning — yield a 2.3× and 1.5× reduction in runtime, respectively. In our final FastDepth model, after compilation and pruning, the encoder has runs in 3.4 ms, while the decoder runs in 2.2 ms. This emphasizes that our

decoder latency has been lowered even further, making the MobileNet encoder now the dominant component. FastDepth ultimately runs 65× faster than the baseline, with similar accuracy. This concludes our successful design and deployment of a fast monocular depth estimation DNN on an embedded GPU and CPU.

Chapter 4

Energy-Efficient Acceleration on an Embedded FPGA

This chapter presents our third contribution, a custom hardware design that aims to improve the energy efficiency of FastDepth. Our motivation for designing hardware dedicated to running FastDepth is twofold:

While we successfully achieved real-time inference on an embedded CPU/GPU platform as described in the previous chapter, these CPU/GPU systems tend to consume roughly on the order 10 W. The Jetson TX2, for instance, can consume 5−20 W. It can be fairly difficult to lower power consumption on these systems without also negatively impacting performance. As a case in point, we observed that setting the TX2 CPU to run in a more energy-efficient power mode reduced FastDepth inference speeds by half. Furthermore, several applications may call for more stringent power consumption constraints. This is where custom-designed hardware comes into play; one of its key strengths is that it can be designed from the ground up and tailored for whatever task it is to run. This makes a custom hardware design likely to be smaller, with lower power consumption, than a general-purpose processor — while also allowing for more opportunities through which to accelerate the task for higher performance. This chapter will discuss the design opportunities we take advantage of to accelerate FastDepth inference on a low-power embedded FPGA.

When designing hardware for a task, we also have the flexibility to adapt the task

to better utilize the hardware. The ability to revisit the algorithm we are trying to run on hardware while the hardware is still being developed is advantageous in that it increases the degrees of freedom in the design process. Algorithm-hardware co-design allows for joint optimizations in both the algorithm and the hardware to achieve what would not have been possible with optimizations in just one or the other. We employ this strategy in our own effort to accelerate FastDepth.

4.1 Algorithm-Hardware Co-Design Strategy

The FastDepth neural network described in Chapter 2 is fully convolutional; there are no pooling or fully connected layers. Furthermore, most layers, except the first and last, are depthwise separable layers (previously defined in Section 1.2.1). The network follows an encoder-decoder structure: encoding layers use kernel sizes of three, while decoding layers use kernel sizes of five. The neural network also incorporates additive skip connections between the encoder and the decoder; these are integral for improved clarity of predicted depth along high-resolution components, e.g., boundary edges.

Widespread usage of depthwise separable convolution throughout the network motivates us to first design an accelerator for these depthwise separable layers. We then envision how various layers in FastDepth would run on the proposed accelerator and reassess aspects of the network that are not natively supported by the accelerator design. This allows us to discover modifications we can make to the original FastDepth network that not only preserve accuracy but also result in an accelerator-friendly network. Upon modifying the DNN and retraining it, all layers of the accelerator-friendly network can natively run on the accelerator — resulting in successful end-to-end inference when deployed on an embedded CPU-FPGA system.

4.1.1 Design Considerations

Design considerations relevant to designing specialized hardware for FastDepth can be grouped into three levels: processing element design, dataflow design, and memory hierarchy design. Design decisions made at one level affect those made at others.

A **processing element** (PE) refers to the lowest-level building block of many neural network accelerator designs, including ours. A PE is often designed to perform a particular computation, e.g., a multiply-and-accumulate (MAC) operation. PEs can be instantiated many times to form a PE array of some size; in this manner, they can be used to significantly increase compute parallelism. PEs typically contain one or more arithmetic unit(s) along with a small amount of fixed storage, e.g., a register file, to store data on which it is performing a computation.

A **dataflow** refers to the manner in which data flows to, through, and from processing elements. Dataflow design determines how processing elements are arrayed, how data flows in between them, and how data may held stationary within them. If there are opportunities to re-use data amongst or within processing elements, a good dataflow design will aim to do so, to improve processing speed and energy-efficiency.

Together, the processing elements and the dataflow constitute the compute core of the accelerator. A **memory hierarchy** is designed around the compute core and involves different levels of memory, each with different storage sizes and different access latencies. On an FPGA, for example, a multi-level memory hierarchy could consist of small on-chip buffers implemented with block RAM and larger off-chip storage in DRAM. Read and write accesses to higher memory levels (e.g., off-chip DRAM) incur significantly higher latency and energy costs than accesses to on-chip memory. Therefore, an appropriate memory design goal would be to store highly re-used data in local on-chip buffers in order to reduce accesses to off-chip DRAM.

The processing element, dataflow, and memory hierarchy are meant to be designed in conjunction with each other to exploit data reuse, increase processing speed and throughput, and improve energy-efficiency. A critical factor to consider during the design process is whether the compute core or the memory hierarchy could be a bottleneck. There are various ways by which one can speed up compute logic on an FPGA, e.g., by clocking the compute core faster, utilizing specialized DSP blocks for arithmetic operations, etc. However, it is more difficult to speed up memory logic, since on-chip and off-chip RAM typically have a lower clock ceiling than arithmetic units. Thus, it is usually the memory hierarchy design and its corresponding bandwidth lim-

its that are bottlenecks in an accelerator design. In our case, since the purpose of the memory hierarchy is to enable our dataflow and interface with processing elements, an additional design consideration is ensuring that the memory hierarchy can provide sufficient read/write bandwidth to keep all the PEs in the compute core busy.

4.2 Dataflow Design

In designing an accelerator for FastDepth, we develop a dataflow that can support the type of layer most prevalent throughout the network, i.e., the depthwise separable layer. Our dataflow design seeks to minimize off-chip memory accesses for model parameters and feature maps. As described in Section 1.2.1, depthwise separable layers consist of two convolutions: depthwise and pointwise. These two convolutions primarily differ in how values are accumulated, which motivates a heterogeneous dataflow design that can offer dedicated support for both convolution types.

4.2.1 Heterogeneous Dataflow for Depthwise Separable Layers

The rise of deep learning — and hardware for deep learning — has spurred much research on dataflow design for accelerating neural networks. Several dataflow variants have since emerged: weight-stationary [86, 107], output-stationary [91], input-stationary [96], and row-stationary [106]. As discussed in Section 1.3.4, these dataflow variants keep different datatypes stationary within processing elements to enable data reuse. Since data movement often constitutes a considerable, if not dominant, energy and latency cost, a good dataflow choice will aim to exploit data reuse and reduce movement across the memory hierarchy, especially accesses to off-chip memory.

We use the row-stationary dataflow for depthwise convolution and an output-stationary dataflow for pointwise convolution. This heterogeneous dataflow design maximizes convolutional reuse and minimizes partial sum movement.

Row-Stationary Dataflow for Depthwise Convolution

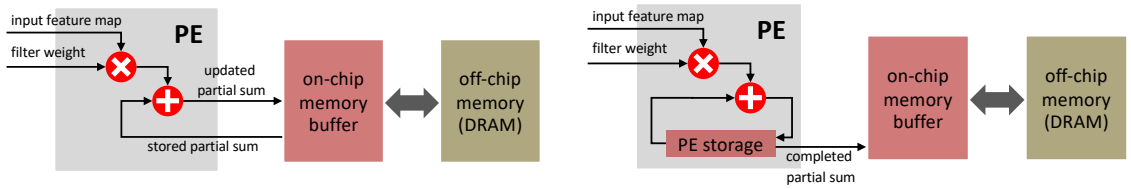
Depthwise convolution involves a kernel size typically greater than 1, which implies spatial accumulation. However, depthwise convolution takes place on a per-channel basis, meaning that the number of output channels equals the number of input channels, and the values of each output channel depend only on the inputs from a single input channel. There is no channel-wise accumulation, as illustrated in Figure 1-4(a).

This suggests that the main opportunity for data reuse is in spatial accumulation and convolutional reuse. The row-stationary dataflow [106] has been shown to successfully exploit convolutional reuse and optimize for best overall energy-efficiency. Hence, we adopt the row-stationary dataflow for depthwise convolution.

Output-Stationary Dataflow for Pointwise Convolution

Pointwise convolution involves a kernel size equal to 1, meaning that there is no spatial accumulation. For a pointwise convolution with C input channels and M output channels, there are C 1×1 filters that effectively scale each of the C input channels. Afterwards, these channels are all added element-wise to produce a single output channel. This is repeated for M output channels, with each output channel having its own set of C filters. In this manner, pointwise convolution performs scaled channel-wise accumulation, as illustrated in Figure 1-4(b).

Since there is no spatial accumulation as part of pointwise convolution, the row-stationary dataflow is less suited for this type of convolution. Instead, our dataflow choices are again informed by the types of data that are reused heavily throughout the convolution. There is a total of $C\times M$ weights for each pointwise convolution; these are known and fixed at inference-time and are reused as the input feature map is iterated through. Thus, a weight-stationary dataflow that holds weights within the processing elements to be reused during the corresponding pointwise convolution(s) is one potential dataflow choice. Alternatively, another data type that is reused often is the set of partial sums when accumulating across channels. Since a processing element would still be operating on a row of a feature map at a time, it would need to



(a) writing out and reading in partial sums (b) holding partial sums stationary in PE

Figure 4-1: Motivation for an output stationary dataflow. In (a), partial sum outputs are written out to on-chip memory (and potentially to off-chip memory), then read back in so they can be accumulated or updates as the PE continues to compute MACs. In (b), partial sums are held stationary in PE storage until accumulation is done, and the completed partial sum output is written out once. Since data movement across memory levels (PE $\leftrightarrow$ on-chip buffer $\leftrightarrow$ off-chip DRAM) gets increasingly more expensive, both in terms of latency and energy [1], option (b) is a desirable choice.

continuously add to a row of partial sums until channel-wise accumulation is complete. Holding this row of partial sums within the processing elements would avoid the need to continuously write out the partial sums to a local buffer and then read the values back in, as highlighted in Figure 4-1. Hence, an output-stationary dataflow that holds partial sums within the processing elements during channel-wise accumulation is another potential dataflow choice.

Deciding between weight-stationary and output-stationary dataflows for pointwise convolution comes down to determining which dataflow yields a greater reduction in data movement across the memory hierarchy. Table 4.1 compares data movement in each of the two dataflow choices. In the weight stationary dataflow, weights are held locally while partial sums are written out and then read back in. In the output-stationary dataflow, partial sums are held locally, while weights are only read in. By computing read and write statistics across all convolutional layers in FastDepth, we determine there to be roughly 10$\times$ as much data movement of partial sums as of weights. Hence, to minimize pointwise partial sum movement and reduce data movement across the memory hierarchy, we select the output-stationary dataflow for pointwise convolution. We also seek to minimize weight movement by caching weights in on-chip buffers, such that every weight is read in from off-chip memory just once.

Dataflow	Datatype Moved Across Memory Hierarchy	Total Values and Bitwidth[1]	Reads or Writes[2]	Data Moved
weight-stationary	pointwise partial sums	∼2.6M (24b per psum)	W, then R	15.6MB
output-stationary	pointwise weights	∼1.3M (10b per weight)	R only	1.6MB

[1] Assuming bitwidths that we will be using in our accelerator design.

[2] Assuming each weight is read once and each partial sum is written out and read once.

Table 4.1: Selecting a dataflow for pointwise convolution: choosing between weight-stationary and output-stationary. The weight-stationary dataflow suffers from high overhead of writing and reading high-bitwidth pointwise partial sums. The output-stationary dataflow avoids this overhead by holding partial sums within processing elements, thus achieving roughly 10× reduction in data movement. This makes it a more appealing choice for pointwise convolution.

Dataflow Visualization by Example

Figures 4-2 and 4-3 provide a visualization of how our dataflow processes depthwise separable layers. This visualization uses a toy example consisting of a depthwise convolution of a $9\times9\times C$ input feature map with a $3\times3\times C$ kernel, followed by a pointwise convolution with a $1\times1\times C\times M$ kernel, producing a $7\times7\times M$ output feature map. It accounts for depthwise and pointwise bias addition as well.

We use a 3×7 array of PEs as a basis. The rationale for this is as follows: Every column of the array will work on a single row of the output feature map. Every row of the array will compute either a 1D convolution with single row of the weight kernel (in depthwise convolution) or a different output channel (in pointwise convolution).

Arrows in these figures represent data reuse, e.g., depthwise input feature maps are reused diagonally across PEs, while weights are reused horizontally. Analogously, pointwise input feature maps (that are just the depthwise outputs) are reused vertically, while weights are again reused horizontally. Boxes shown within the PEs represent storage of completed output feature map rows.

4.2.2 On Serializing vs. Pipelining the Dataflow Design

In our dataflow design, all PEs are envisioned to be reconfigurable, i.e., they support both the row-stationary and output-stationary dataflows and toggle between the two when computing a depthwise separable layer. This approach **serializes** the process-

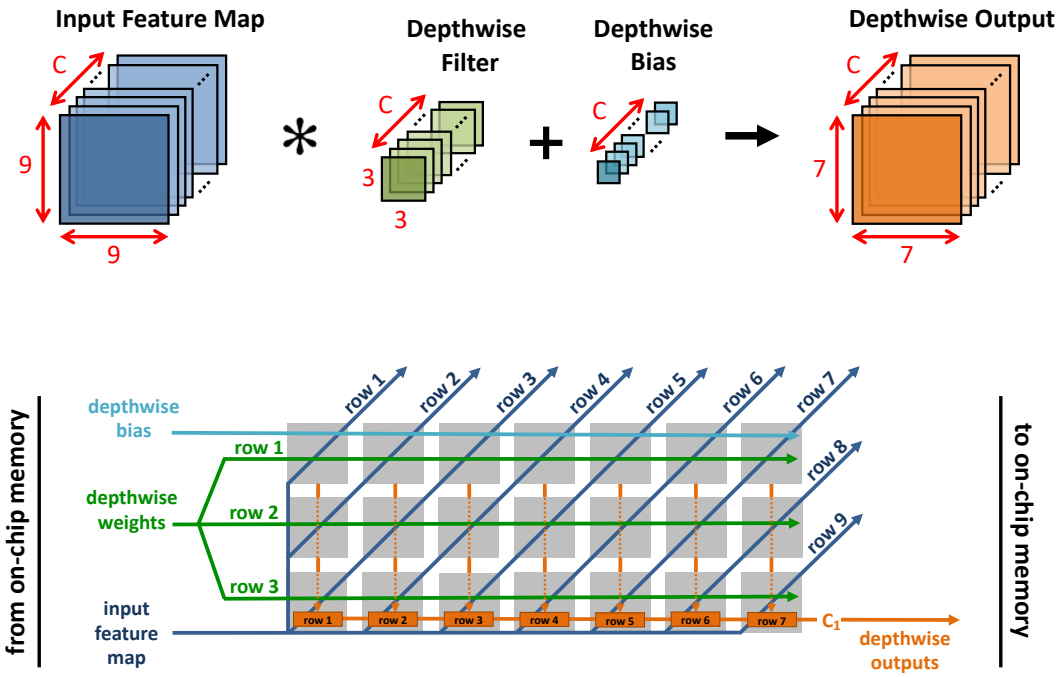


Figure 4-2: Row-stationary dataflow for depthwise convolution. Each processing element (PE), depicted as a gray box, takes a row of input feature map values and a row of depthwise filter weights as input; after convolving the rows, the PE sends its result to the PE below it for spatial accumulation. The bottom-most PEs contain completed results from the 3×3 convolution and can send those out to memory. This dataflow computes up to 7 rows from a single channel of depthwise outputs at once.

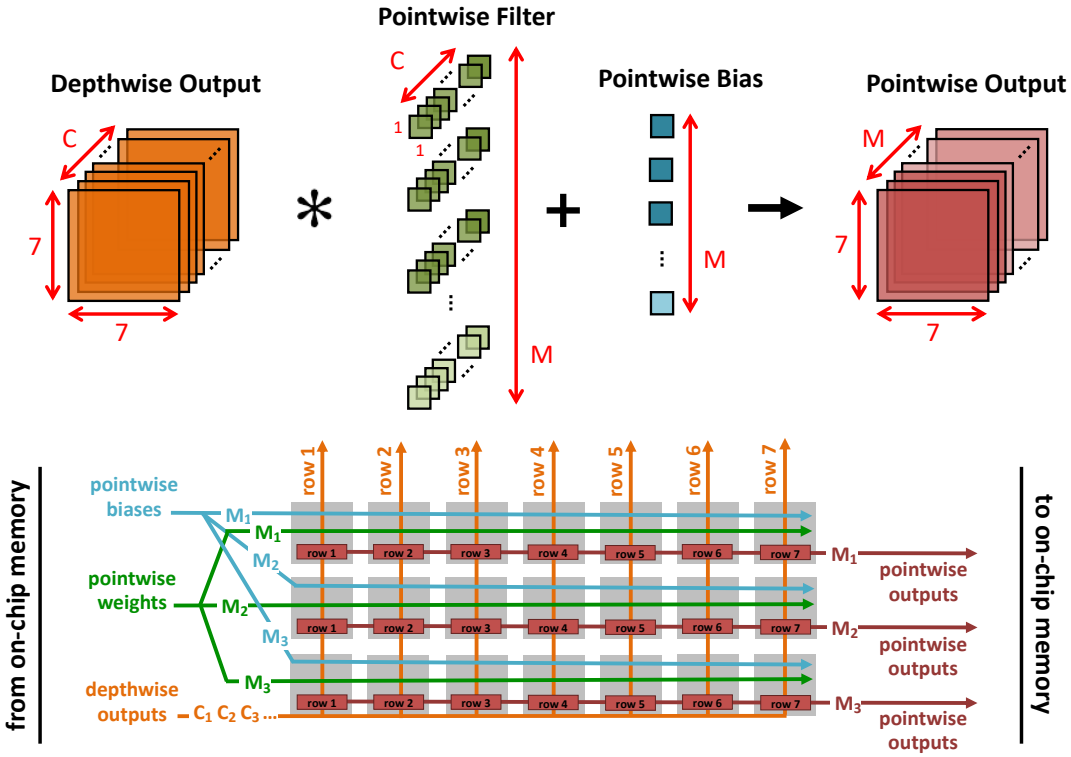


Figure 4-3: Output-stationary dataflow for pointwise convolution. This begins after the row-stationary dataflow has completed *all* depthwise channel outputs. These depthwise outputs are then streamed channel-by-channel back into the processing elements. Each row of PEs in the array receives pointwise weights for all channels from one filter. This allows every PE in the row to complete channel-wise aggregation for a unique row of a single pointwise output channel. Different rows of PEs work on different pointwise output channels. Every PE will hold its row of pointwise partial sums stationary until channel-wise accumulation is complete. This dataflow computes up to 7 rows from 3 channels of pointwise outputs at once.

ing of depthwise and pointwise convolutions. An alternate approach would have been to **pipeline** the two convolutions, e.g., by having a smaller array work on the depthwise operation and a larger array work on the more dominant pointwise operation. Figure 4-4 depicts how the PE array could be partitioned into smaller dedicated arrays. It also points to the key advantages and disadvantages of the two approaches. While a serialized approach allows for greater flexibility in utilizing the PE array, it necessitates a reconfigurable PE design that can be larger and more complex than an otherwise more dedicated PE. A pipelined approach, on the other hand, allows for a simpler PE design but complicates load balancing and potential data transfer between the two pipelined operations — load balancing becomes especially critical to keeping PEs working on different tasks busy.

We perform a preliminary analysis with analytical computations comparing these serialized and pipelined approaches. In this analysis, we assume that the partitioned depthwise PE array has a height of at least 3 rows to natively support 3×3 convolution using the row-stationary dataflow. We also assume that the width of the depthwise PE array will match one of the dimensions of the pointwise array so that the outputs from the PEs in depthwise array could be immediately transferred to those in pointwise array (to avoid caching them). We then experiment with the remaining flexible dimensions to gauge compute speed and PE utilization in the pipelined approach as compared to the serialized approach. Our analysis results are shown in Figure 4-5 and can be summarized as follows:

- For a small PE array (e.g., less than 100 PEs), the analysis shows that a serialized approach will be faster. Pointwise convolution is known to be the dominant operation in a depthwise separable layer. When applying a pipelined approach with a small number of PEs, partitioning a separate array for the depthwise operation effectively takes resources away from the dominant pointwise operation. This slows down layer computation as a whole.

- For a large PE array (e.g., more than 500 PEs), a serialized approach will allow for greater parallelism than a pipelined approach. Since both the depthwise

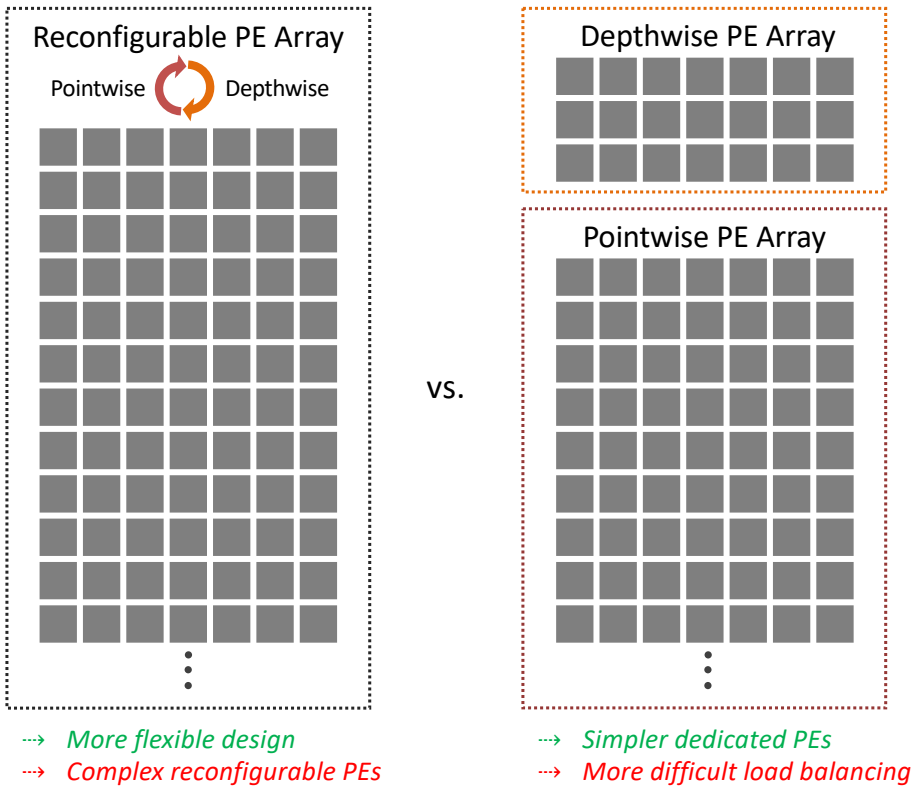


Figure 4-4: Partitioning the PE array for serialized vs. pipelined dataflow approaches. A serialized approach requires a reconfigurable PE array that can toggle, e.g., between depthwise and pointwise convolution dataflows. This allows for more flexible allocation of PE resources but increases the complexity of PE design and control logic. A pipelined approach sets aside subsets of the PE array dedicated to each of the pipelined operations, which in this case are the depthwise and pointwise convolutions. This allows for simpler PE designs in each of the sub-arrays; however, since pointwise computations dominate in quantity over depthwise computations, a pipelined approach makes load balancing across dedicated PE arrays more difficult.

and pointwise operations will get to benefit from this greater parallelism, the serialized approach will again be faster overall.

- For a medium-sized PE array (e.g., between 100 and 500 PEs), the two approaches are much closer to each other performance-wise, yet the serialized approach still edges out the pipelined approach since it always utilizes the full PE array for the dominant pointwise operation.

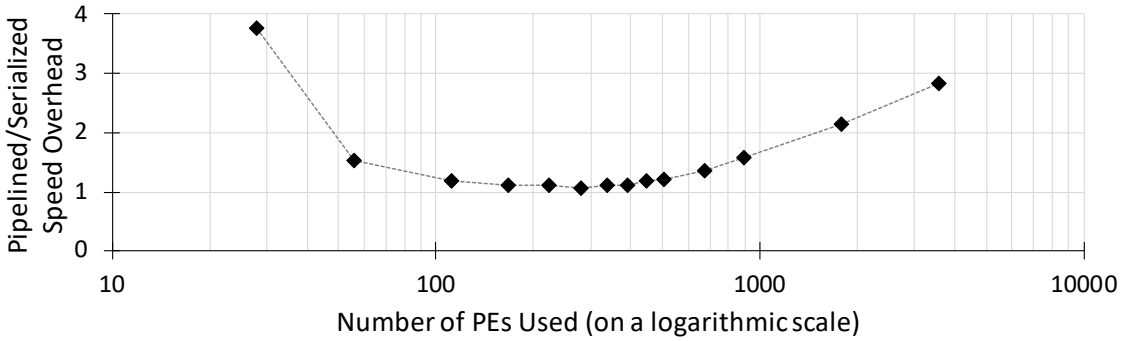


Figure 4-5: Comparing serialized vs. pipelined dataflow approaches. Our analysis shows that a pipelined approach will be at least slightly slower than a serialized approach over a wide range of PE array sizes. This figure plots the speed overhead of a pipelined approach relative to a serialized one, where speeds are proxied by analytically computing clock cycles for all depthwise separable convolutional layers in FastDepth's MobileNet encoder.

We note that under the assumptions we make in partitioning our PE array, our analysis does not consider all degrees of freedom when selecting dimensions for the depthwise and pointwise arrays. We also do not consider caching to facilitate load balancing between the two pipelined convolutions. As a result, our analysis is not exhaustive. However, it serves as a guiding factor in our selection of a serialized approach for our PE array design.

4.3 Accelerator Design

A high-level architecture diagram of our accelerator design is shown in Figure 4-6. The accelerator is built up of blocks of processing elements (PEs) and a memory

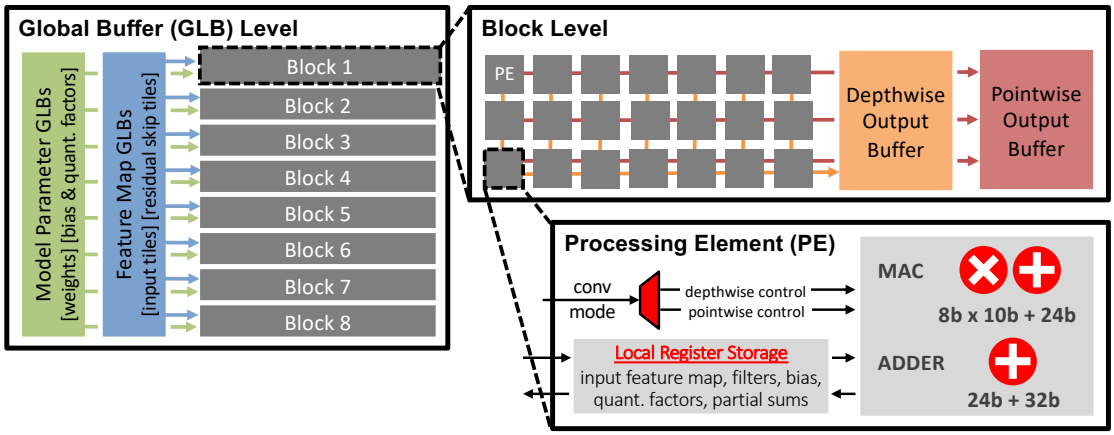


Figure 4-6: High-level accelerator diagram. The innermost module is the processing element (PE) that computes multiply-accumulate operations. PEs are arranged in blocks that compute depthwise and pointwise convolutions. Blocks can be replicated for more compute parallelism. The resulting PE array interfaces with on-chip memory consisting of local PE storage, block-level storage, and larger global buffers (GLBs).

hierarchy consisting of local PE storage, block-level storage, and larger global buffers (GLBs) that are banked to provide sufficient data bandwidth for all blocks of PEs.

The previous section describes the dataflows used in our accelerator and depicts how processing elements are interconnected via these dataflows. Here, Sections 4.3.1 and 4.3.2 describe the remaining key aspects of the compute core design: the processing element itself, and how the PE array is constructed. Sections 4.3.3 and 4.3.4 then describe the memory hierarchy designed around the compute core.

4.3.1 Compute Core

Processing Element Design

A processing element is the innermost module of the compute core. It is responsible for computing the multiply-and-accumulate operations (MACs) that make up the convolutions in the FastDepth network. A typical MAC operation looks like:

$$\textbf{feature map value} \times \textbf{weight value} + \textbf{accumulated value}$$

We assume that after quantizing and transforming FastDepth layers to run on the accelerator, feature map values do not exceed 8 bits and weight values do not

exceed 10 bits.[1] Multiplying values of these bitwidths results in a product no larger than 8+10=18 bits. Supposing that such 18-bit values need to be accumulated across channels (as in pointwise convolution), the bitwidth required for accumulation depends on the largest channel dimension in FastDepth. Since all channel dimensions in the pruned FastDepth network are under 1024 (up to 10 bits), the largest possible accumulation in this case is 18+10=28 bits. However, we find that the full bit range of feature map and weight values is not used in any of the layers and the 24 bits is enough for accumulation. Hence, the PE is designed to perform 8-bit by 10-bit multiplication with 24-bit accumulation.

Each of the layers in FastDepth also has a bias that needs to be added to the completed convolution. We assume these biases to be 32-bit values that need to be added to the 24-bit accumulated values mentioned previously. Hence, the PE also has an adder to perform 24-bit and 32-bit addition.

This MAC and addition arithmetic is depicted in the PE diagram shown in Figure 4-7. The inputs and outputs of these operations, along with their bitwidths, are tabulated in Table 4.2. These values are all stored in registers local to the PE. They are held for the duration of computation over a single feature map row, after which they may be overwritten with new values or kept for reuse. New values are prefetched as necessary to keep PEs as continuously busy as possible.

Following MAC computation and bias addition, a nonlinear ReLU activation function may applied to the completed partial sums. Since partial sums are signed values, ReLU is easily performed by checking the most significant bit (effectively the sign bit) and zero-ing out the value if the bit is a 1. Afterwards, the partial sum is quantized to an 8-bit unsigned value using a predetermined quantization factor. Every PE has the capability to perform ReLU and quantization on streaming data on-the-fly. Whether this capability is enabled depends on the type of convolution being performed. During depthwise convolution, a PE may send its outputs to a neighboring PE for spatial accumulation; in this case, the PE does not perform ReLU or quantization. During pointwise convolution, however, every PE will generate complete outputs that must

[1] Quantization of FastDepth layers is described in Section 4.4.2.

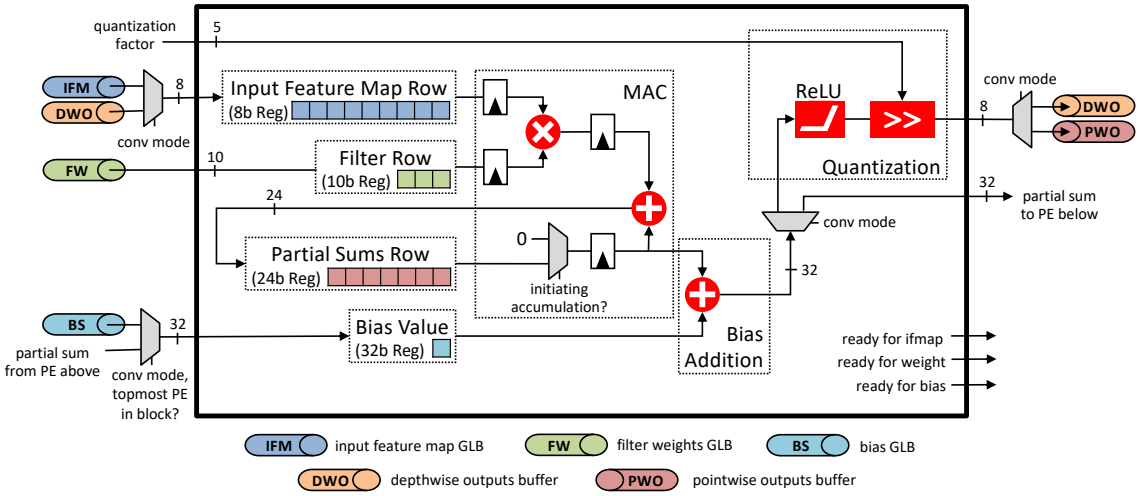


Figure 4-7: Diagram of a processing element. The PE performs multiply-accumulate operations (MACs) and bias addition; it also applies ReLU and quantization functions to values being written out to on-chip buffers. The PE performs row-wise processing (e.g., 1D convolutions of rows, element-wise addition of rows) and stores a row of an input feature map, a filter, and partial psums at a time. Some datapaths and control logic are reconfigurable based on the convolution mode (depthwise vs. pointwise).

Datatype	Bitwidth
input feature map	8-bit unsigned
depthwise layer weights	10-bit signed
depthwise layer bias	32-bit signed
depthwise accumulated values	24-bit signed
depthwise outputs	8-bit unsigned
pointwise layer weights	10-bit signed
pointwise layer bias	32-bit signed
pointwise accumulated values	24-bit signed
pointwise outputs	8-bit unsigned
quantization factors	5-bit unsigned

Table 4.2: Datatypes and bitwidths processed within a processing element.

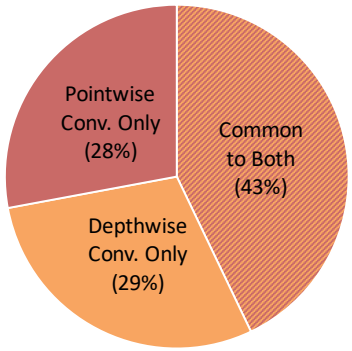


Figure 4-8: Logic breakdown comparing depthwise- and pointwise-specific logic within the PE. Logic here refers primarily to LUTs and registers found in the PE netlist after synthesis. Common logic mostly includes shared registers. Depthwise- and pointwise-specific logic includes counters and control logic for those convolutions.

undergo ReLU and quantization prior to being sent out to an output buffer.

In order to support both depthwise and pointwise convolutions, each PE contains control logic to enable both the row-stationary and output-stationary dataflows, and can toggle between them depending on the type of convolution being performed. This reconfigurability allows for a single homogeneous PE design but incurs a logic overhead. To reduce this overhead, we avoid replicating register storage, e.g. the same registers used to store weights, biases, and partial sums during depthwise convolution are used during pointwise convolution. Similarly, the MAC unit and the adder are also reused. Figure 4-8 illustrates the partition of logic nets within the reconfigurable PE into those needed for depthwise convolution, needed for pointwise convolution, and shared between the two convolution modes. Depthwise-specific and pointwise-specific logic take up roughly 29% and 28%, respectively, of all the PE logic, with the remaining 43% of logic being reused for both operations. Based on this, PEs dedicated solely for depthwise or pointwise convolution would use around 72% or 71% of logic nets compared to the reconfigurable PE. This gives an estimate for the logic overhead of reconfigurability in the PE: when compared to a PE design dedicated solely for depthwise convolution or one dedicated solely for pointwise convolution, the reconfigurable PE design incurs roughly $100\%/71\% = 1.4$ or 40% logic overhead.

This analysis excludes logic overhead that is datapath related: e.g., multiplexing

the buses carrying depthwise vs. pointwise input feature maps, and demultiplexing the bus carrying partial sums (to flow between PEs in the depthwise case or to be held stationary within PEs for accumulation in the pointwise case). This is due to the breakdown in Figure 4-8 being calculated based on logic nets found in just the PE, which does not include the far more distributed logic comprising the network-on-chip that delivers data to the PE. However, the datapath or network-on-chip logic is not replicated nearly as much as the PEs are; hence, we use the PE logic breakdown as a first-order estimate of reconfigurability overhead.

Processing Element Array

A single processing element performs computation on a single feature map row at a time. For depthwise convolution, an individual PE processes a row of input feature map values and a row of depthwise weights, producing a row of depthwise partial sums before moving onto the next row. For pointwise convolution, an individual PE processes a row of depthwise partial sums and a single pointwise weight, producing a row of pointwise partial sums before moving onto the next row.

We now explain how the PE array is built up in a modular fashion. We use depthwise convolution as a case scenario first and then extend our discussion to how the PE array processes pointwise convolution. Since our PE array is design to directly support our dataflow design, it may help to refer back to Figure 4-2.

PE Column. In order to implement the row-stationary dataflow for depthwise convolution, we create a column of PEs, where the height of the column equals the height of the convolution kernel. Most depthwise convolutions in FastDepth use 3×3 kernels, so we define a column to have three PEs. Each of these PEs computes a 1D convolution on a row of input feature map values and a row of depthwise weights. This **row-wise filter parallelism** allows for the results of these three 1D convolutions to stay local within the PE interconnect: they flow down through the PE column and are aggregated such that the 1D convolution results of $\text{PE}_{(i,j)}$ serve as the bias input to $\text{PE}_{(i,j+1)}$, where i and j refer respectively to the column and row indices of the PE location in the array. This becomes equivalent to computing a 2D

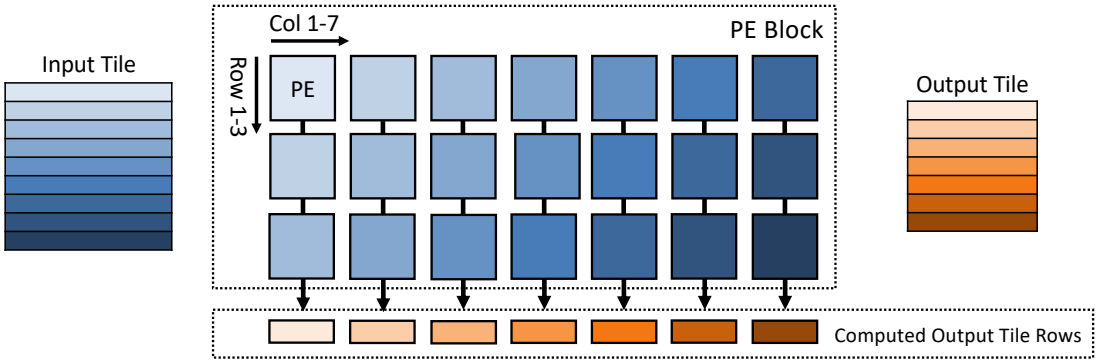


Figure 4-9: Row-wise output parallelism in the PE block. Each column of PEs works on a different row of convolution outputs. The number of columns equals the number of rows in an output tile, which is selected to be 7, the greatest common factor of output feature map dimensions in FastDepth layers.

3×3 convolution. Thus, a column of PEs will produce a single row of completed 3×3 depthwise convolution outputs that are ready to be read out to a buffer.

PE Block. In order to enable **row-wise output parallelism**, we group several columns of PEs together so that each column can work on a different row of depthwise convolution outputs. This approach exploits diagonal reuse of input feature map rows and horizontal reuse of depthwise weights. The number of columns is selected to be 7, the greatest common factor of output feature map dimensions P and Q (defined in Table 1.1) across all FastDepth layers. This is to ensure that all PE columns in the array are adequately utilized. Spatial and temporal utilization of PEs will be discussed more in Sections 4.5.3 and 4.5.3.

The 3×7 array of PEs discussed so far forms what we call a PE block. Figure 4-9 illustrates this concept of a PE block and the row-wise parallelism it exhibits. A block can work on 7 rows of a single-channel depthwise convolution output at a time. The bottom-most PEs in the block will contain completed rows of depthwise partial sums that will be quantized prior to being stored in the block's depthwise psums buffer.

PE Array. The overall PE array simply consists of multiple PE blocks. Given that a PE block works on a single depthwise channel at a time, and that depthwise convolutions often involve many channels, there is an opportunity for **channel-wise**

parallelism. PE blocks are replicated so that each one works on a different feature map channel. This level of parallelism is more coarse-grained than the row-wise filter and output parallelism discussed earlier. The number of blocks is primarily constrained by the number of arithmetic units or the amount of logic you have available. In our design, we use an array of 8 PE blocks, as an array of that size fits onto our target FPGA hardware. It merits to note that 8 is also the greatest common factor of channel dimensions across all FastDepth layers, meaning that an array of 8 PE blocks will achieve full spatial utilization during depthwise convolution.

We extend the PE array's functionality from depthwise convolution to pointwise convolution in a straightforward manner. PE blocks still exhibit row-wise and channel-wise parallelism. The main difference now is that there is no need for spatial accumulation, so every row of PEs in a given PE block can be treated independently. To maximize channel-wise parallelism, we assign each row of PEs to process a single output channel of the pointwise convolution. Every PE then computes a distinct feature map row within that output channel. Since our design uses an output-stationary dataflow for pointwise operations, every PE will hold the row for local accumulation until the pointwise convolution is done. At that point, every individual PE will have a completed row of the output feature map ready to be read out to an output buffer.

To summarize, our accelerator is built up of 8 PE blocks, each of which contains a 3×7 sub-array of PEs. This results in a PE array with a total PE count of 168 PEs. Figure 4-10 shows how the PE array can process up to 8 input channels in parallel during depthwise convolution and up to 24 output channels in parallel during pointwise convolution. The array can compute up to 7 rows of an output feature map at once (i.e., an output tile of height 7) during both types of convolutions.

Processing Element Interconnect

PE interconnect refers to how PEs communicate with each other. PEs in a given PE block are effectively isolated from PEs in a different block, i.e., there is no direct communication between PEs across blocks. Within a PE block, each column of PEs operates in a standalone manner; different columns work to compute different output

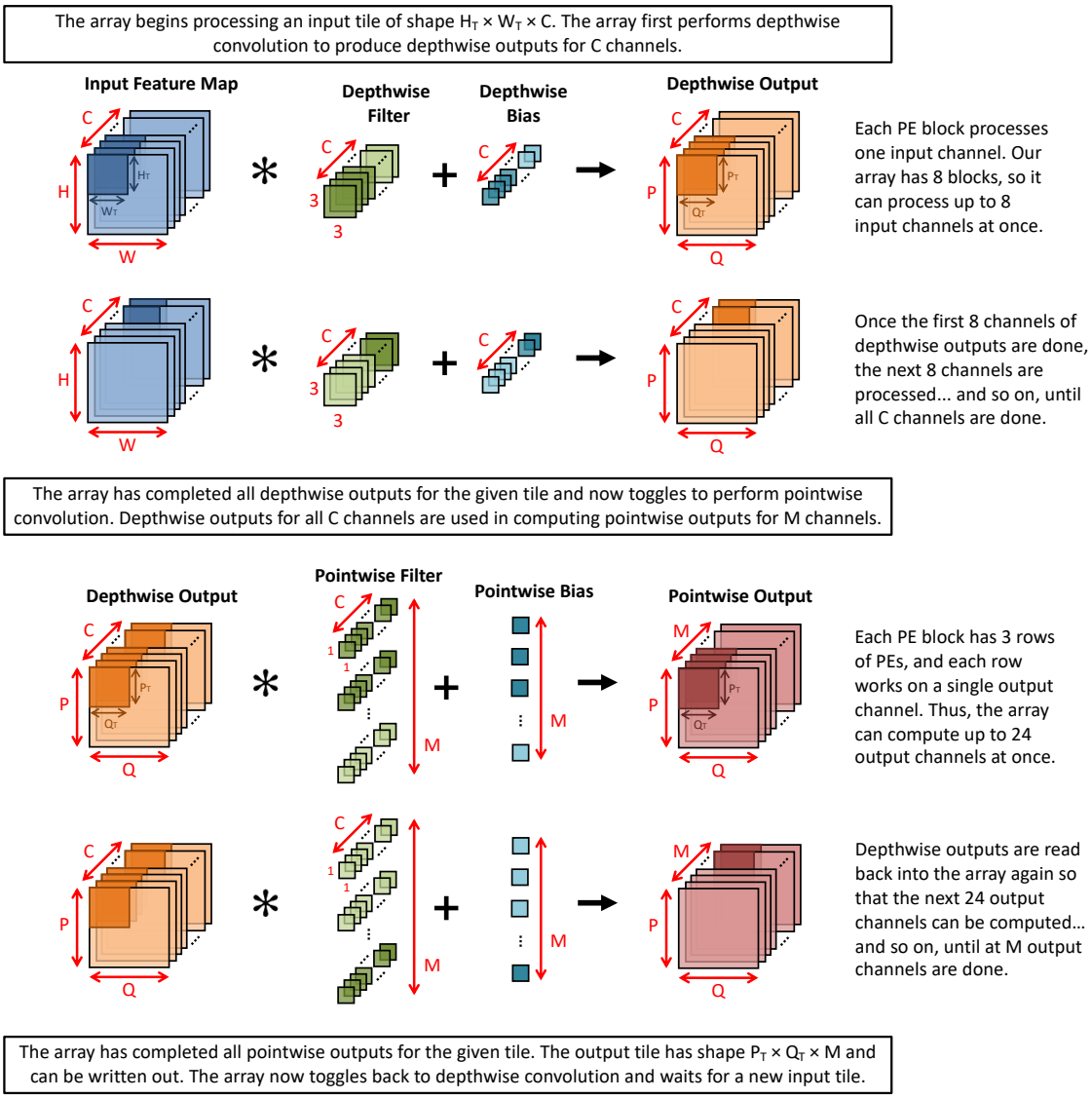


Figure 4-10: Diagram illustrating how the PE array processes channels during depthwise and pointwise convolution. Shown for a single tile that may be smaller than the input or output feature map (e.g., as shown here by the darker-shaded portions within feature maps). To cover the entire feature map, tiles are iterated through in a raster scan — first horizontally, then vertically.

feature map rows and therefore do not depend on results from each other. The only direct connections present between PEs are within a column. As described in Section 4.3.1, to perform spatial accumulation during depthwise convolution, each PE in a column sends its partial sums down the column to the PE below it; in other words, a PE within a column will depend on results from PEs above it. During pointwise convolution, however, there is no spatial accumulation and each PE in a column operates independently. PE interconnect forms part of the network-on-chip (NoC).

4.3.2 Network-on-Chip (NoC)

The network-on-chip manages data delivery to individual PEs as well as between PEs. It directly interfaces with on-chip memory and with the PE array.

In our design, the NoC side interfacing with PEs is local to a PE block, i.e., it is solely responsible for connecting with the 3×7=21 PEs in the block. Replicating a PE block replicates this NoC within it. The NoC side interfacing with on-chip memory is, on the contrary, not local to a PE block, as it is multiplexed to facilitate reading from on-chip buffers in different blocks. Here, we focus on the NoC side interfacing with PEs, while on-chip memory is discussed in the following Section 4.3.3.

Inside a PE block, the NoC implements the data delivery patterns seen in row-stationary and output-stationary dataflows, as illustrated in Figures 4-2 and 4-3 respectively. Communication via the NoC is *multicast*:

- Weights read in from on-chip memory are multicast to PEs across rows during both depthwise and pointwise convolution. All PEs in any given row always receive the same depthwise and pointwise weight values.

- Incoming feature map values are multicast to PEs across diagonals during depthwise convolution and across columns during pointwise convolution.

- Incoming bias values are multicast across the topmost PE row during depthwise convolution and across all PE rows during pointwise convolution.

Outgoing values from a PE are routed depending on convolution type and PE location. In depthwise convolution, PEs produce depthwise partial sums that get communicated from PE-to-PE within a column: output partial sums from $\text{PE}_{(i,j)}$ directly feed as an input to $\text{PE}_{(i,j+1)}$, where i and j refer respectively to the column and row indices of the PE in the array. The bottom-most PE row contains completed convolution outputs that can be quantized and sent out into a depthwise output buffer. In pointwise convolution, every row of PEs produces pointwise partial sums independently of each other; these partial sums are held within the PEs themselves until pointwise computation is done. Afterwards, every PE row contains completed convolution outputs that can be quantized and sent out into a pointwise output buffer.

To summarize, the NoC uses multicast communication to deliver inputs being read in from on-chip memory. For data that can be immediately reused within the PE network (e.g., depthwise partial sums that are accumulated in registers within individual PEs), the NoC keeps this data local to PEs through PE-to-PE communication. Lastly, for delivering outputs to be written out to on-chip memory, the NoC connects a row of PEs to an appropriate output buffer.

NoC Bandwidth and Datapath Widths

A critical goal of the NoC is to provide enough bandwidth for data delivery to support the highly parallelized processing in the PE array. Our design seeks to keep all PEs in sync with each other. This affects the datapath widths within the NoC:

- During both depthwise and pointwise convolution, weights need to be delivered along 3 PE rows in parallel. Each weight is 10 bits, which corresponds to a 30-bit datapath in the weights NoC.

- During depthwise convolution, input feature map values need to be delivered along 9 PE diagonals in parallel. During pointwise convolution, feature map values need to be delivered along 7 PE columns in parallel. Since each feature map value is 8 bits, this corresponds to a 72-bit and a 56-bit datapath, respectively. The input feature map NoC toggles between these two datapaths.

- During depthwise convolution, biases need to be delivered along a single PE row. During pointwise convolution, biases need to be delivered along 3 PE rows in parallel. Since each bias is 32 bits, this corresponds to a 32-bit and a 96-bit datapath, respectively. The bias NoC toggles between the two datapaths.

- Incomplete depthwise partial sums are 32 bits and need to be delivered along PE columns. The NoC includes 32-bit connections between PEs in every column.

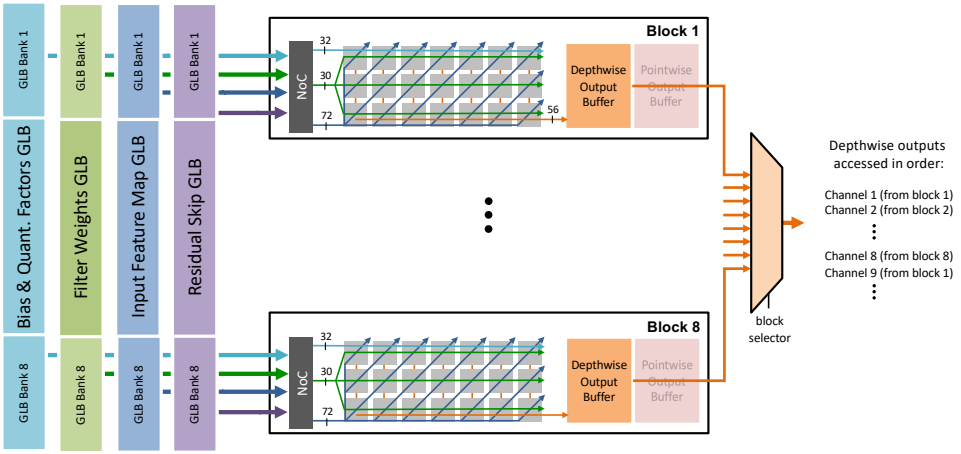
- Completed depthwise outputs are quantized to 8-bit values and need to be written out from all 7 PEs in the bottom-most row at once. This necessitates a 56-bit datapath from the bottom-most PE row to the depthwise output buffer.

- Completed pointwise outputs are quantized to 8-bit values and need to be written out from all 7 PEs in each of 3 rows at once. This necessitates a 56-bit datapath from each PE row to the pointwise output buffer.

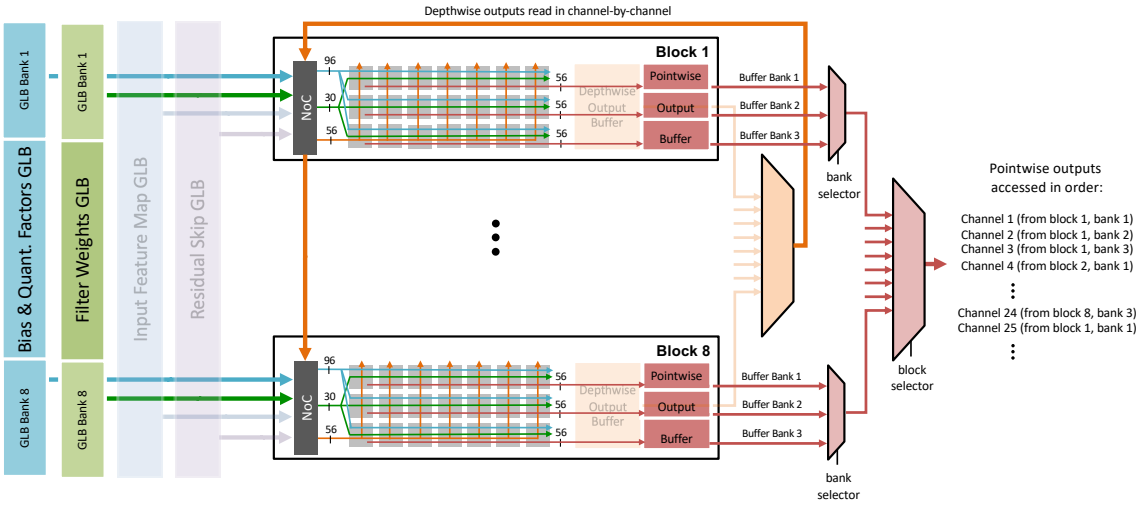
The NoC and its various datapath widths are illustrated alongside on-chip memory in Figure 4-11. The on-chip memory hierarchy is discussed more in Section 4.3.3.

Handling Strides and Skip Connections in the Input Feature Map NoC

One design challenge concerning the input feature map NoC is in handling different convolution strides. While most convolutions in FastDepth have a stride of 1, several layers in the MobileNet encoder have a stride of 2. When supporting different convolution strides in our design, we keep the output feature map tile size fixed, i.e. we design so that the PE array produces the same-sized output tile. As compared to convolving with a stride of 1, convolving with a stride of 2 covers 2× as much input in each of the height and width dimensions (equaling to 4× as much input overall). Hence, in order to compute the same-sized output tile when convolving with a stride 2, our design needs to read in 4× as many input values for that convolution. We achieve this by setting aside enough on-chip memory to be able to store up to four input feature map tiles at once. The tiles are loaded on-chip sequentially. Afterwards, when reading from on-chip memory into the PE array, we buffer individual channels



(a) during depthwise convolution



(b) during pointwise convolution

Figure 4-11: Network-on-Chip (NoC) connecting PE blocks to on-chip memory. Input feature maps, weights, and biases are delivered to PE blocks from on-chip global buffers (GLBs). There is a separate GLB for each of the different datatypes, and all GLBs are banked to provide parallel read access to all PE blocks. After convolution, outputs are held within PE blocks; each PE block has dedicated storage for the depthwise and pointwise output channels computed by that block. In-order read access for those outputs is controlled via block selectors (multiplexers).

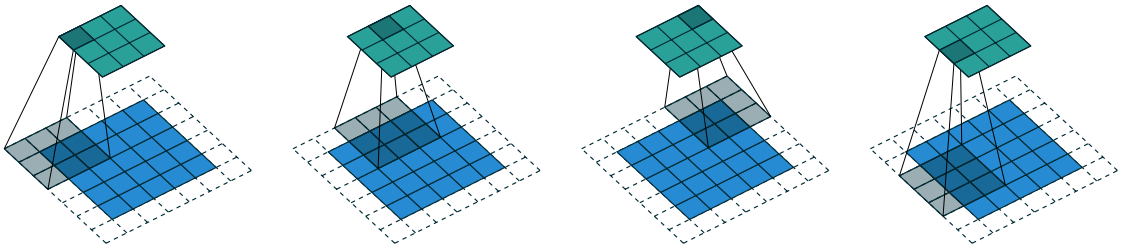
within the tiles and align them in a 2×2 window. This allows us to generate a longer row of input feature map values that we can stream into a PE block. The NoC within the PE block then performs vertical striding, skipping rows as necessary. The PEs that process the rows perform horizontal striding, skipping over values as necessary. This handling of convolution strides is illustrated in Figure 4-12.

Another design challenge is supporting the addition of residual feature maps that have been passed along skip connections. We perform this addition as part of the logic for input feature map delivery. First, we read in the residual feature map tiles just as we read in the input feature map tiles; the two sets of tiles are stored in disjoint on-chip buffers. We then pass one or more tiles (depending on the convolution stride) through the buffer and alignment logic described above. Following that, we optionally add the residual tile to the input tile on a row-by-row basis prior as the feature map rows are delivered to PE blocks. This is visualized in Figure 4-13.

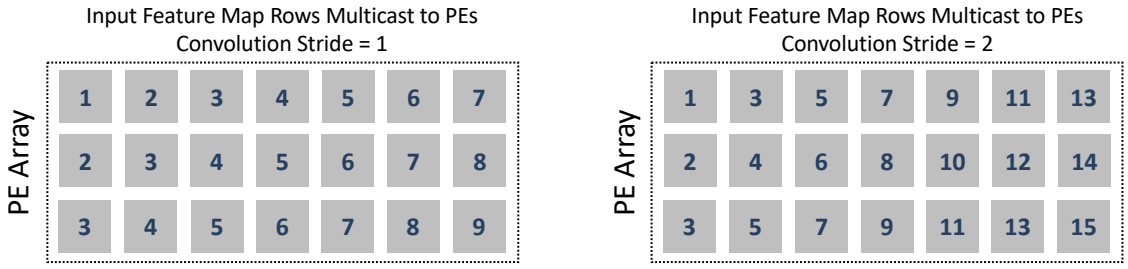
4.3.3 On-chip Memory Hierarchy

On-chip memory serves several key purposes: on-chip buffers can hold data that is reused during processing, thereby reducing time- and energy-costly off-chip memory accesses. Buffers can also store pre-fetched data, which helps to hide memory read latency. Lastly, buffers can cache outputs from a previous computation to allow the next computation to begin before all previous outputs are completely copied out; this helps to hide memory write latency. On-chip memories are typically more limited in size than off-chip memories, requiring more deliberation when deciding how to partition these memories and which data to store in them.

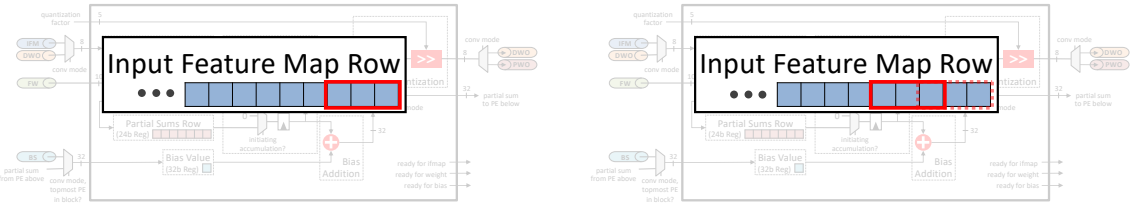
On-chip memory on our target platform — the ZU3EG MPSoC — comes in the form of Block RAM (BRAM) and distributed RAM. There is 7.6Mb of BRAM available on the FPGA, consisting of 216 blocks of 36Kb RAM (that can be reconfigured to 432 blocks of 18Kb RAM). Additionally, there is up to 1.8Mb of distributed RAM that can be implemented via lookup tables (LUTs). In general, BRAM operations are slower than distributed RAM ones, e.g. a read operations to BRAM has a latency



(a) Convolution with a horizontal and vertical stride of 2. The input feature map is shown in blue, the filter in gray, and the output feature map in teal. Figure taken from [5].



(b) Vertical striding performed by the NoC. The delivery pattern that the NoC uses to multicast input feature map rows to PEs is set based on the convolution stride.



(c) Horizontal striding performed by the PE. When convolving an input feature map row with a filter row, the PE computes MACs over a window of the same length as the filter; in our case, this length is 3. When the convolution stride is 1, the PE iterates through the input feature map by shifting this window one element at a time. When the convolution stride is 2, the PE shifts the window two elements over, as shown above.

Figure 4-12: Handling convolution strides in the NoC and the PE.

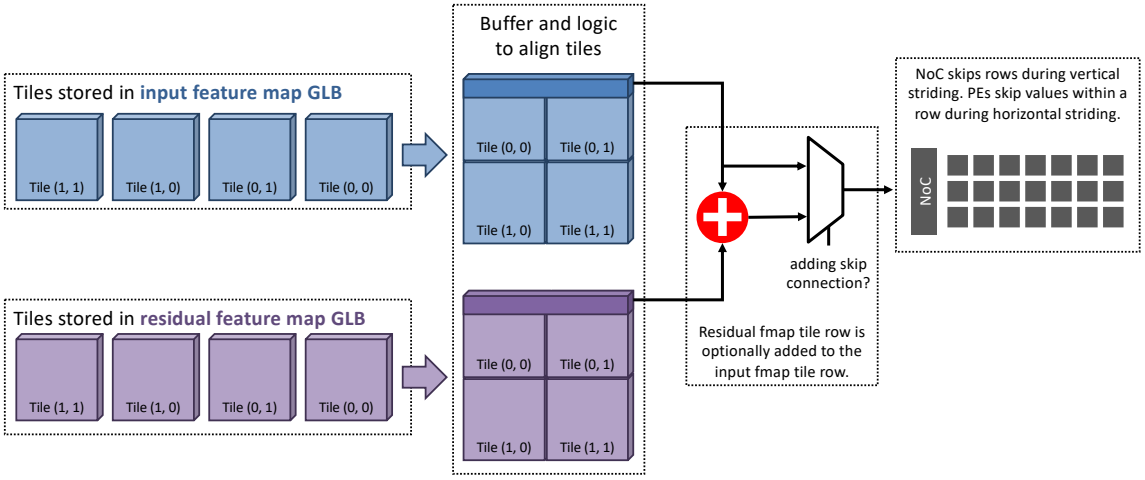


Figure 4-13: Buffer and alignment logic to facilitate convolution striding and adding skip connections. In both cases, four feature map tiles are buffered on-chip at once.

of 2 clock cycles,[2] whereas with distributed RAM, the latency can be made to be 1 or even 0 clock cycles. However, distributed RAMs are much generally smaller and contribute to LUT usage. BRAMs allow for coarse-grained partitioning of memory, where the smallest possible partition is the size of a single BRAM. If one is instantiated and only a small subset of its address space is used, the BRAM as a whole will be underutilized. For buffers that are small and/or instantiated many times, it is therefore advantageous to use memory that is more fine-grained, e.g., distributed RAM, as using block RAM for these will result in under-utilized BRAMs. For larger buffers, however, BRAMs are an appropriate choice.

Our accelerator utilizes both BRAM and distributed RAM depending on storage needs at the PE, block, and array levels of the design:

Storage at the PE and NoC Level

Each PE contains storage for data it is actively processing, i.e., a row of input feature map values, a row of filter weights, a row of accumulated partial sums, and a bias value. PE storage uses distributed RAM and is replicated a large number of times as the PE array is built up. Distributed RAM is also used to implement line buffers for

[2]Assuming that output registers have been set, which allows the BRAM to be clocked faster.

stitching tiles during striding and adding residual tiles in skip connections as part of the input feature map NoC.

Buffers at the Block Level

There are two buffers in each PE block: one that holds outputs of depthwise convolution, and one that holds outputs of pointwise convolution.

The goal of the **depthwise output buffer** is to completely avoid writing depthwise convolution results out to off-chip memory. Since the pointwise convolution that follows depthwise convolution is an element-wise operation that scales and aggregates along the channel dimension, it can be performed to completion on any-sized feature map tile as long as all depthwise channels are available. To facilitate this, the depthwise output buffer stores all channels output by the PEs during the depthwise operation — and then immediately feed these channels back to the PEs for the pointwise operation. This exploits temporal reuse of depthwise outputs on-chip.

The goal of the **pointwise output buffer** is to cache pointwise convolution outputs. This serves two purposes: (1) to lower PE idle time arising from stalled computation due to bandwidth limitations when directly sending pointwise outputs to off-chip memory, and (2) to allow reordering of pointwise outputs so that they can be accessed in the correct order (e.g., consecutive channel-by-channel, as illustrated with the selectors/multiplexers in Figure 4-11).

Both of these output buffers are implemented in block RAM. Every block has its own set of dedicated buffers; this is to ensure that there is enough bandwidth to grab outputs from the PE network in order to keep PEs busy and prevent having one PE to wait for an adjacent one to write its outputs out. While this simplifies writing to buffers, it makes reading slightly more complicated. Since every PE block processes a disjoint set of input and output channels, their corresponding output buffers store a disjoint set of outputs. For instance, in our array with 8 PE blocks, PE block 1 stores depthwise outputs for channels $C = 1, 8, 16, ...$ and pointwise outputs for channels $M = 1, 2, 3, 25, 26, 27, 49, 50, 51, ...$

In order for these outputs to be read out in the correct order, buffers from all of

the blocks need to be iterated through. Depthwise output buffers receive values from just one PE row in the block and thus contain only one bank; iterating through these is sequential — first, the buffer from block 1 is read from, then the buffer from block 2, and so on, looping through the blocks until all channels stored in the buffers have been read out. Reads from pointwise output buffers function similarly. However, these buffers receive values from three PE rows in parallel and thus contain three banks, one for each PE row. Hence, when iterating through these buffers, the banks need to be iterated through before a subsequent block's buffer is read from. This is visualized in Figure 4-11, where the multiplexers to the right of the buffers represent selector-based iteration through all the blocks' output buffers.

Global Buffers at the Array Level

There are four global buffers (GLBs) present at the array level: **a feature map GLB** that stores up to four input feature map tiles at a time, **another feature map GLB** that stores up to four residual feature map tiles when skip connections are being added on-chip, **a weights GLB** that stores depthwise and pointwise weights for an entire layer, and **a bias GLB** that stores depthwise and pointwise biases along with channel-wise quantization factors. All GLBs are implemented in BRAM.

GLBs are populated with values streaming into the accelerator. They directly interface with PE blocks in the compute core via the blocks' NoC. The compute core begins processing a layer after all GLBs have been initially loaded. Since the weights and bias GLBs are large enough to store all the weights and biases for a given layer, they are simply re-read from during computation until the layer has been fully processed. The input feature map GLB, however, is only large enough to store a tile (or, for strided convolutions, a subset of tiles) of the whole feature map at a time. While the compute core works on one tile, the next tile begins is loaded; this continues until all tiles have been iterated through. Once the compute core is ready to process the next layer in the network, all GLBs are overwritten with parameters and the first feature map tile for that layer.

In order for GLBs to provide sufficient bandwidth to the NoC within each of the

Datatype	Width per Bank	Depth per Bank	# of Banks	Total Size	BRAM Used
bias & quant. factors GLB	37 bits	132	8	0.04Mb	0.29Mb
filter weights GLB	30 bits	11814	8	2.84Mb	3.02Mb
input fmap tile(s) GLB	72 bits	2376	8	1.37Mb	1.44Mb
residual fmap tile(s) GLB	72 bits	2376	8	1.37Mb	1.44Mb
depthwise output buffer	56 bits	462	8	0.21Mb	0.29Mb
pointwise output buffer	56 bits	154	24	0.21Mb	0.87Mb
aggregated over all GLBs and output buffers:				**6.04Mb**	**7.35Mb**

Table 4.3: Block RAM usage for on-chip GLBs and output buffers. Total size refers to how much memory our design actively uses for each datatype. BRAM used refers to how many 18Kb and 36Kb block RAMs are placed-and-routed in our design by the synthesis and implementation process.

PE blocks in the PE array, the GLBs are banked; there is a bank dedicated to each PE block. This allows GLBs to feed all PE blocks simultaneously in parallel, as opposed to having a single un-banked memory block that serially sends to each of the PE blocks. This further enforces independent operation of PE blocks, keeps them in sync with each other, and maximizes parallelism and speed within the compute core.

Table 4.3 lists the sizes and parameters of these GLBs as well as the block-level output buffers. The total amount of BRAM used is larger than the size of each particular GLB or buffer. This is due to BRAM being available in fixed sizes (18Kb or 36Kb on our target hardware, the Xilinx Zynq UltraScale+ MPSoC). Individual GLB or buffer banks will not necessarily use the entirety of the implemented BRAM.[3]

BRAMs can be configured with a variety of widths, which determines whether 18Kb or 36Kb RAMs are instantiated. Since we use simple dual-port (SDP) BRAM with a single read port and a single write port, our 18Kb BRAMs support word widths of up to 36 bits and 36Kb BRAMs support word widths of up to 72 bits [117]. This allows our large filter weights GLBs, using 30-bit word widths, to be more densely built up out of a combination of 18Kb and 36Kb RAMs, resulting in 94% utilization. Meanwhile, GLBs for biases and quantization factors use 37-bit word widths and are

[3]If this were an ASIC design with flexibility in how much or where to place on-chip memory, such under-utilization of memory could be better handled. However, since this is a design targeting an FPGA, we accept this under-utilization as long as there is enough BRAM available for use.

built up of 36Kb RAMs; they are not very populated (as there are so few biases compared to all other datatypes), which contributes to a low utilization of only 14%.

In total, our design uses 7.35Mb of BRAM, approximately 97% of available on-chip memory. While this is enough for our accelerator to process and compute a single 7×7 output feature map tile in a single layer at a time, it is far too small to store entire feature maps and parameters for all layers during inference. Hence, the accelerator must also interface with the larger off-chip memory.

4.3.4 Off-chip Memory Interface

Our accelerator design targets the Ultra96 board that comes with a Zynq UltraScale+ MPSoC and 2GB LPDDR4 (512M×32) off-chip memory [118]. The MPSoC consists of a Processing System (PS) unit and a Programmable Logic (PL) unit.[4] The PL, where our accelerator is implemented, has the ability to access the PS-side DRAM via dedicated high-performance ports on the PS-PL interface. This section describes how I/O in our design is set up, as well as the DMA (direct memory access) interface that enables these DRAM reads and writes to and from our accelerator.

Top Level Design Wrapper

Figure 4-14 shows a block diagram illustrating the hierarchy of modules providing I/O connectivity to our accelerator. The `top_level_control` module directly interfaces with our accelerator logic (i.e. the GLBs and the PE array). I/O to this module primarily consists of `data` and `valid` signals for different datatypes as well as output control signals for monitoring or debugging. For compatibility with the AXI-Stream protocol that will be necessary for PS-PL interfacing, we add the `io_stream` module. This module handles input and output streams that are 64 bits wide and follow standard `valid-ready` handshaking. It performs time-multiplexing of inputs as necessary to feed the correct datatypes into the `top_level_control` module.[5]

[4]The PS contains the ARM processor on the chip. The PL refers to the FPGA fabric.

[5]An input stream to a layer consists of bias values and quantization factors, followed by weight values, and lastly, the input feature map tiles. The `io_stream` module determines when to assert the `valid` signal for each datatype as these values stream in.

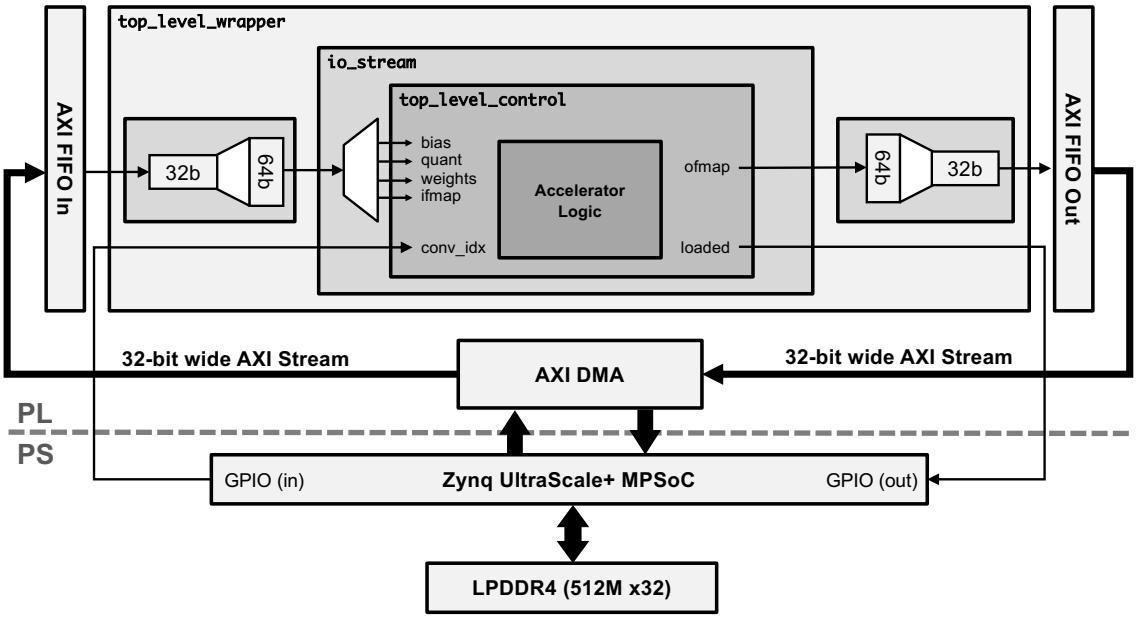


Figure 4-14: Top-level design wrapper and the DMA interface.

The DRAM onboard the Ultra96 supports a bus width of 32 bits. Hence, the input and output streams to/from the `io_stream` module need to be up-sized (from 32 bits to 64 bits) and down-sized (from 64 bits to 32 bits), respectively. We add custom modules to perform these functions. Together with the `io_stream` module, they are packaged into the `top_level_wrapper`. This wrapper provides 32-bit I/O connectivity that is compatible with the AXI protocol.

In addition to 32-bit I/O data streams, the `top_level_wrapper` exposes several ports that connect to GPIO pins on the Zynq PS. Output GPIO pins from PS to the PL carry the `conv_idx` bits to specify which layer is being run on the accelerator. Input GPIO pins from the PL to the PS carry a bus of `loaded` signals specifying whether GLBs within the accelerator have been loaded.

DMA Interface

For access to off-chip DRAM, we use AXI-based DMA. Our design integrates the AXI DMA IP [119]; this core provides a high-bandwidth interface between AXI4 memory-mapped reads/writes and data streams adhering to the AXI4-Stream protocol. The

AXI DMA IP operates on a clock frequency of 100 MHz and provides a total throughput of 400 MBps on the memory-mapped-to-stream channel and 300 MBps on the stream-to-memory-mapped channel [119].

The AXI DMA IP offers two modes of operation: simple mode, where DMA reads from and writes to contiguous memory buffers; and Scatter/Gather mode, where an optional Scatter/Gather Engine fetches and updates a set of buffer descriptors (that may point to non-contiguous memory). Although Scatter/Gather mode offers more flexibility, we use the simple mode for its easy setup and compatibility with the PYNQ framework [120]. We set the Memory Map Data Width and the Stream Data Width to 32 bits to match the DRAM bus width. Lastly, we set the Max Burst Size to 16.[6]

The AXI DMA IP connects to our accelerator design through FIFOs on the input and output ends. These are smaller FIFOs (with depths of just a few elements) that we add in addition to the DMA FIFOs already within the IP core. The purpose of these smaller secondary FIFOs is to handle the `tlast` signal indicating when a DMA transfer is complete. On the input end, this signal is simply stripped as our accelerator automatically waits for new incoming data. On the output end, however, this `tlast` signal must be manually asserted after the accelerator outputs all elements of all tiles of an output feature map. Since this element count will vary between layers, we specify it via an external input signal to the output-side FIFO.

4.4 Accelerator-Friendly FastDepth DNN

The dataflow and accelerator architecture that we have designed so far have been tailored for depthwise separable layers with 3×3 and 1×1 kernels. Our goal is to deploy all FastDepth layers onto this accelerator. However, not all of them use 3×3 kernels; notably, the spatial convolutions throughout in the decoder use 5×5 kernels. This presents a challenge in mapping the decoder onto our proposed accelerator.

This motivates our algorithm-hardware co-design strategy. Instead of expanding

[6]We note that the AXI4-compliant DMA IP can support burst sizes up to 256; however, when interfacing with the Zynq PS, there is a conversion to AXI3 in the AXI FIFO interface that limits the transaction burst length to a maximum of 16 [121].

the versatility of our accelerator, which would incur inefficiencies, we keep our accelerator dedicated to 3×3 convolutions and instead re-examine the FastDepth network itself. We experiment with cascaded convolutions and kernel decompositions to develop an accelerator-friendly FastDepth network that achieves the same accuracy as our original FastDepth DNN yet natively maps onto our accelerator. At this stage, we also develop a quantization scheme for a lower-precision FastDepth DNN with 8-bit integer activations and 10-bit integer weights.

4.4.1 Network Modifications

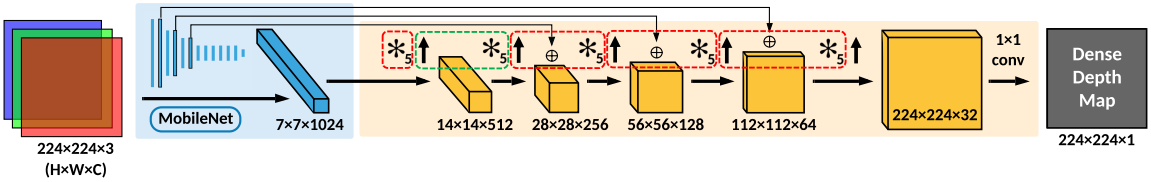
Modifications made to the original FastDepth network — for it to be better compatible with our accelerator — largely focus on enabling the decomposition of 5×5 convolutions into 3×3 convolutions. As will be shown later in this section, the decomposition is possible if: (1) interpolation and convolution operations can be grouped such that interpolation takes place immediately before convolution, and (2) the type of interpolation used is either zero-insertion or nearest-neighbors interpolation.[7]

A diagram of the original FastDepth network is shown again in Figure 4-15(a), now with dotted boxes depicting possible groupings of nearest-neighbor interpolation and 5×5 convolution operations. However, not all of these groupings are decomposable: in several groupings, skip connections are added in between the interpolation and the convolution, thus breaking the structure that would enable decomposition. Hence, the first modification to FastDepth that we explore is shifting these skip connections.

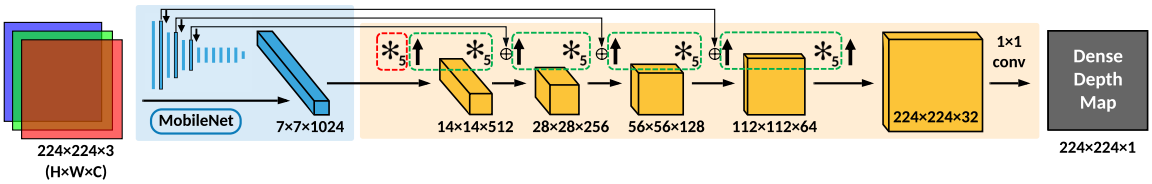
Shifting and Downsampling Skip Connections

Skip connections carry feature maps layers from earlier in the network to later layers in the network. This allows those later layers to incorporate features extracted in earlier layers, which has been shown to improve accuracy of networks performing

[7]This is because zero-insertion introduces sparsity from the zeros being inserted between the rows and columns of input feature map values. Nearest-neighbors interpolation introduces structure from a pixel value being copied into adjacent pixel locations, resulting in windows of pixels known to have identical values. Performing a convolution on a feature map with such sparsity or structure leads to data reuse that can be exploited.



(a) original FastDepth network



(b) after shifting skip connections in the decoder

Figure 4-15: Shifting skip connections in the decoder so that they terminate *before* interpolation allows us to ensure that nearest-neighbor interpolation is always followed by a 5×5 convolution; this enables the decomposition of the 5×5 convolution into smaller 3×3 ones. Additionally, this shift allows us to downsample feature maps from the encoder prior to them being passed along the skip connection, which contributes to a reduction in overall feature map data movement.

dense prediction tasks where the output is of similar dimensionality as the input. FastDepth uses additive skip connections, meaning that feature maps passed along the skip connection are added to the feature maps at the end of the connection.

In the original FastDepth network, skip connections originate from the second, fourth, and sixth layers of the MobileNet encoder and terminate at the inputs to the last three 5×5 convolutions in the decoder. This means that the addition of feature map values passed via the skip connections takes place right before each of those convolutions but after the outputs of the previous convolutions have already been upsampled through nearest-neighbors interpolation. As a result, it is not possible to group the interpolation and convolution operations such that interpolation directly precedes convolution. To enable such groupings, it becomes necessary to shift the skip connections so that they terminate either before interpolation or after convolution.

We find that moving skip connections before interpolation versus after convolution does not result in any appreciable difference in overall network accuracy. Conceptu-

ally, this is reasonable to expect, as the majority of the network structure remains unchanged, and additive skip connections are still present even if they terminate in slightly offset locations within the network. We ultimately decide to use skip connections that have been moved before interpolations. Doing so constrains the skip connections to carry feature maps with heights and widths each reduced by a factor of 2, since the shapes of the carried feature maps must match the shapes of the inputs to the interpolation in order for element-wise addition to be valid. This allows feature maps to be downsampled prior to being passed along the skip connection, which reduces the total amount of data moved along skip connections by a factor of 4.

The shifted and downsampled skip connections in our modified FastDepth network are shown in Figure 4-15(b). All dotted boxes now contain decomposable groupings of nearest-neighbor interpolation and 5×5 convolution operations. Decomposition will be described in greater detail in the following section.

Decomposing Nearest-Neighbors Interpolation with 5×5 Convolution into a Set of Smaller 3×3 Convolutions

Some types of interpolation, e.g., zero-insertion and nearest-neighbors interpolation, produce outputs that are either sparser or more structured than the inputs. Zero-insertion introduces sparsity from the zeros being inserted between the rows and columns of input feature map values. Nearest-neighbors interpolation introduces structure from a pixel value being copied into adjacent pixel locations, resulting in windows of pixels known to have identical values. Performing a convolution on a feature map with such sparsity or structure leads to computational patterns or data reuse that can be exploited.

Suppose that an input feature map is first upsampled through zero-insertion and then fed into a 5×5 convolution. The upsampled feature map will be 75% sparse. As the 5×5 convolution kernel is slid across the upsampled feature map, 75% of the MACs will involve multiplications by zero. This pattern allows the 5×5 kernel to be decomposed into four smaller kernels that, when convolved with the original non-upsampled feature map, will yield four distinct output feature maps. These

decomposed outputs can be interleaved to produce the same output as the one from the 5×5 convolution performed after upsampling. Such an approach was explored by Laina et al. [2] as part of their up-convolution operations; their work found that using such a decomposition made up-convolution operations more efficient and decreased training time. Conceptually, this could be attributed to the decomposition resulting in fewer MACs (due to smaller kernels acting on smaller input feature map sizes) and lower data movement (since upsampling is avoided). Yazdanbakhsh et al. [122] also explored a variant of this decomposition in their work on FlexiGAN, where they used filter and row reordering to make the convolution following zero-insertion more compact and dense for better hardware resource utilization.

Inspired by these approaches, we are interested in determining whether a decomposition can be performed when the mode of upsampling is not zero-insertion but rather nearest-neighbor interpolation. In this case, the interpolation feature map will have replicated pixel values — every 2×2 window of pixels will have identical values, which can be thought of as a source of known structure and redundancy in the feature map. Suppose that this interpolated feature map is fed into a 5×5 convolution, as shown in Figure 4-16. As the 5×5 kernel (i.e., the filter) is slid across, several of the MACs will involve multiplying kernel values by the same pixel values. More precisely, one can visualize 2×2 windows in which kernel values align over a patch of identical pixel values. Within such windows, instead of performing 4 multiplications and 4 additions, the kernel values can first be pre-added and then multiplied once by the pixel value. This takes advantage of redundancy in the pixels and yields a 75% reduction in computation for that window.[8] In this manner, the 5×5 filter can be decomposed into four smaller 3×3 filters that, when convolved with the original non-interpolated feature map, produce correctly valued interleave-able outputs. Figure 4-17 indicates that upon interleaving the four individual outputs, the resulting output feature map will match the one coming from the original 5×5 convolution performed after nearest-neighbor interpolation.

[8]Kernel values (i.e., filter weights) can be pre-added prior to inference time, removing the need to add weights on the fly.

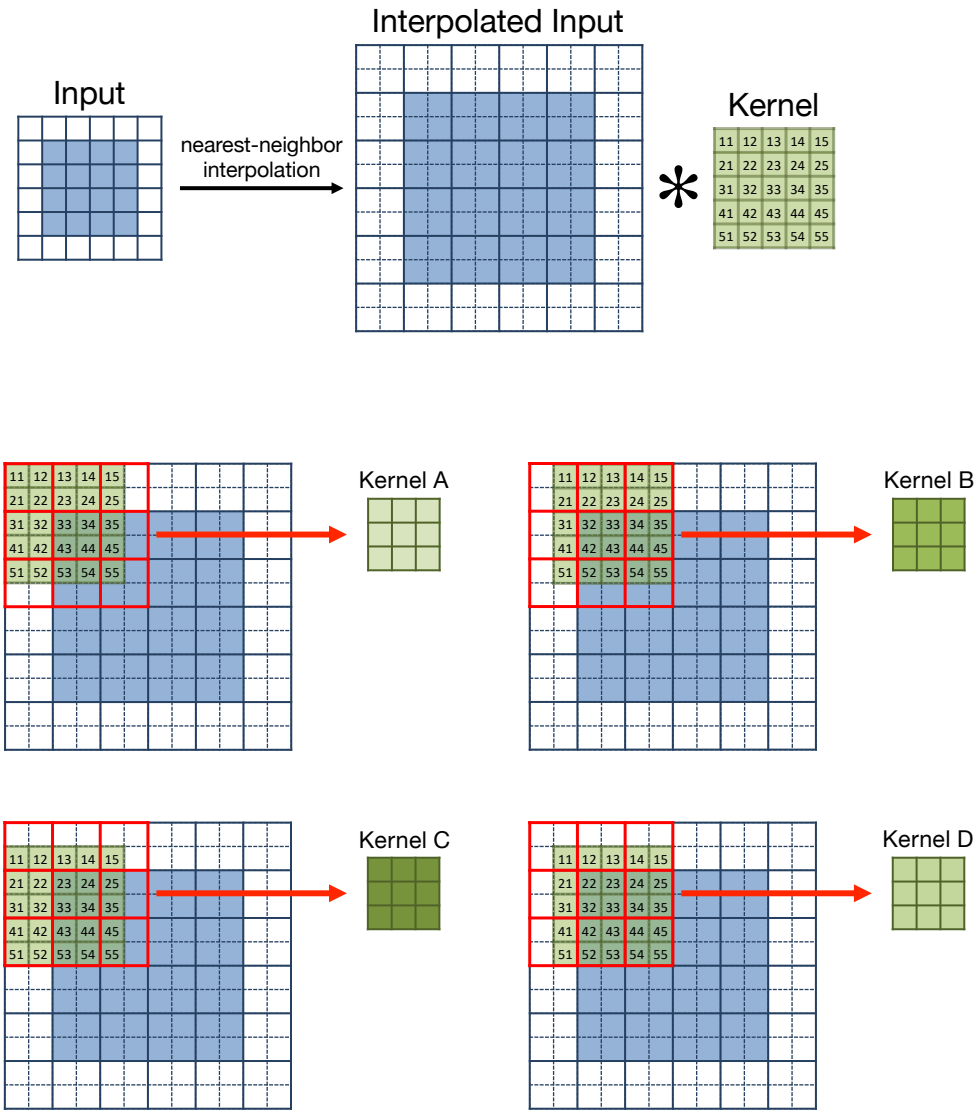


Figure 4-16: Decomposition of a 5×5 kernel into 4 smaller 3×3 kernel. This decomposition is valid when the 5×5 convolution is preceded by nearest-neighbor interpolation. The red boxes here indicates windows of pixels in the interpolated input feature map that have identical values. As the 5×5 kernel slides across, kernel values inside the 2×2 windows get multiplied by identical feature map values. instead of performing 4 multiplications and 4 additions, the kernel values can first be pre-added and then multiplied once by the shared pixel value.

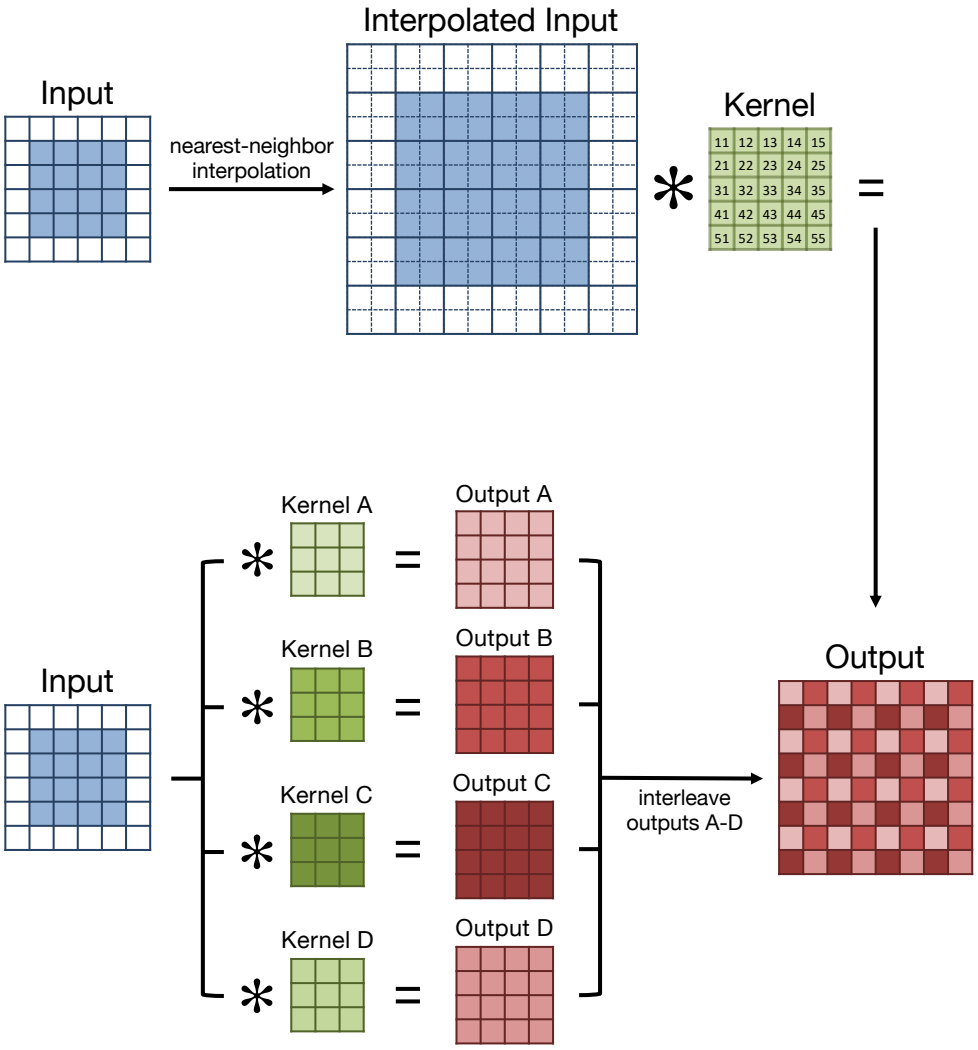


Figure 4-17: After filter decomposition, each of the four smaller 3×3 filters can be convolved with the *non-interpolated* input feature map. This produces four outputs that can be interleaved. The resulting output feature map with match the one coming from the original 5×5 convolution performed after nearest-neighbor interpolation.

In the original FastDepth network, we design each decoding layer so that interpolation happens after convolution. This reduces the number MACs by a factor of 4× for each of the decoding layers, when compared to decoders in existing cited encoder-decoder models [2]. Now, in the modified FastDepth network, we have nearest-neighbor interpolation in layer L directly followed by the 5×5 convolution in layer $L+1$. We can decompose this set of operations into four 3×3 convolutions and remove the interpolation function, thereby reducing the number of MACs in each decoding layer even further by a factor of 2.8×.[9]

Replacing 5×5 Convolution with Cascaded 3×3 Convolutions

The decomposition described above is applicable to all but one of the 5×5 convolution in the FastDepth decoder. The remaining 5×5 convolution in question is the one in the first decoding layer. This layer interfaces directly with the output of the encoder, and there is no interpolation preceding the 5×5 convolution. Keeping this layer as is would result in inefficiencies when mapping it onto our 3×3 convolutional accelerator; e.g., we would first need to map the initial 3 rows of the 5×5 convolution, followed by a second pass for the remaining 2 rows — resulting in a 50% utilization hit for 1/3 of the PEs in the array left idle during this second pass. Furthermore, adding control logic for when and how to support this partitioned 5×5 convolution would further complicate and increase the size of every individual PE. As such, alternate ways of handling the 5×5 convolution need to be considered.

One way is to break apart the 5×5 convolution into a series of 3×3 convolutions, as described by Du et al. in [123]. In this approach, the 5×5 kernel is first zero-padded to create a 6×6 kernel that is then broken up into four smaller 3×3 kernels. This decomposition differs from the one enabled by a preceding nearest-neighbor interpolation in two ways:

1. In the decomposition enabled by nearest-neighbor interpolation, decomposed 3×3 kernels convolve with a feature map that is smaller than the feature map

[9] Compare $5\times5 = 25$ MACs for each pixel in a single channel with the original 5×5 convolution to $4\times(3\times3\times\frac{1}{4}) = 9$ MACs after decomposition into four 3×3 convolutions.

in the original 5×5 convolution. This is not applicable in the decomposition considered here, as there is no interpolation present. Here, the four decomposed 3×3 kernels each convolve with a feature map that is the same size as the feature map in the original 5×5 convolution. Thus, there is no reduction in MACs; on the contrary, unless the extra MACs containing multiplies by zero (due to the kernel padding) are gated or skipped, there will be an increase in total MACs.

2. In the decomposition enabled by nearest-neighbor interpolation, decomposed outputs can be recombined by simple interleaving of rows and columns; this does not involve any extra computation and can be done at the datapath level. In the decomposition considered here, however, recombination of decomposed outputs is more complex. The four outputs need to be overlaid such that they are shifted relative to one another, and then added element-wise. This not only requires extra logic for overlay and addition but also necessitates that all four decomposed outputs be cached on-chip at once.

The computation and storage cost overhead of supporting such a decomposition makes it less ideal of a choice. Another way of handling the standalone 5×5 convolution is by replacing it altogether with a set of 3×3 convolutions. Figure 4-18 shows how two 3×3 convolutions will yield the same receptive field as a 5×5 convolution. **Receptive field** refers to the window of the input that is visible to an element of a filter. This means that each element in the output of the cascaded 3×3 convolutions will depend on just as many input values as each element in the output of a single 5×5 convolution would have depended on. This idea was explored in VGGNet [46].

We focus on maintaining the receptive field of the first decoding layer as doing so helps maintain the receptive field of the whole decoder; this makes it less likely for our network modification to result in accuracy loss. However, this comes at the cost of increased parameter count and MACs due to the additional convolutions that are introduced. Recall that in depthwise separable layers, the pointwise convolution contributes more parameters and more MACs than the depthwise convolution. When replacing a single 5×5 depthwise layer with two cascaded 3×3 depthwise layers, we

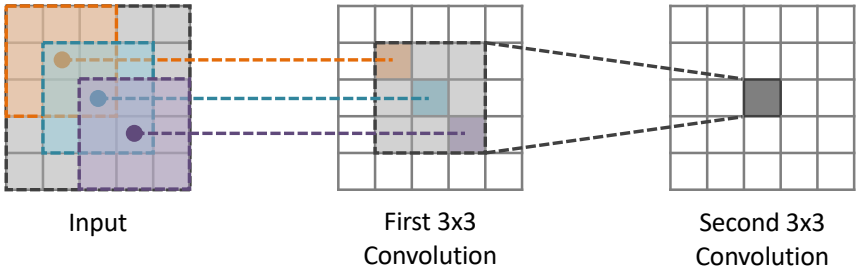


Figure 4-18: Receptive field refers to the window of the input that is visible to an element of a filter in a convolution (shown here by the dashed squares). The receptive field of two cascaded 3×3 convolutions matches that of a single 5×5 convolution.

introduce an extra depthwise and pointwise convolution, the latter of which can drive the number of weights and MACs up. This differs from the VGGNet that used standard convolutional layers. For instance, consider a standard convolutional layer with $5\times5\times C\times M$ filters. These can be replaced with two $3\times3\times C\times M$ filters, resulting in a reduction of $\frac{5\times5\times C\times M}{2\times3\times3\times C\times M} = 1.4\times$. Now, consider a depthwise separable layer with $5\times5\times1\times C$ depthwise filters and $1\times1\times C\times M$ pointwise filters. If we replace this with two depthwise separable layers, each with $3\times3\times1\times C$ depthwise filters and $1\times1\times C\times M$ pointwise filters, there is a net overhead of $\frac{5\times5\times1\times C+1\times1\times C\times M}{2\times(3\times3\times1\times C+1\times1\times C\times M)}$. While there is still a 1.4× reduction in depthwise parameters and MACs, it is offset by the 2× increase in pointwise parameters and MACs. It may be possible to decrease C and M in the cascaded 3×3 depthwise separable layers in order to lower the this overhead (at a potential cost of accuracy); however, we did not experiment with that. We instead reason that the decomposition of subsequent 5×5 layers in the decoder, as described earlier in this section, will lead to a reduction in MACs there, thus mitigating the overall change in MACs due to the overhead incurred here.

Impact of DNN Modifications

Figure 4-19 shows the topology of the FastDepth DNN after the aforementioned modifications are applied. We retrain the modified network using the same experimental settings as described in Section 2.3. We also reapply the NetAdapt pruning algorithm to the retrained network and select the pruning iteration that results in a model with

the lowest accuracy degradation and a MACs count most closely equal to that in the original pruned FastDepth DNN.

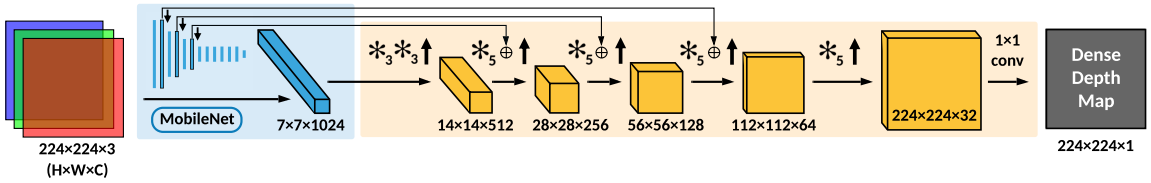
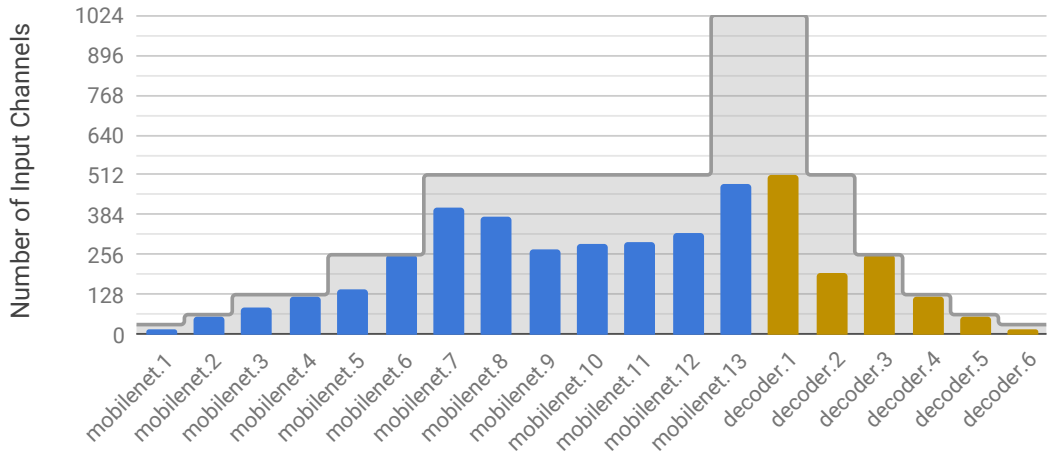


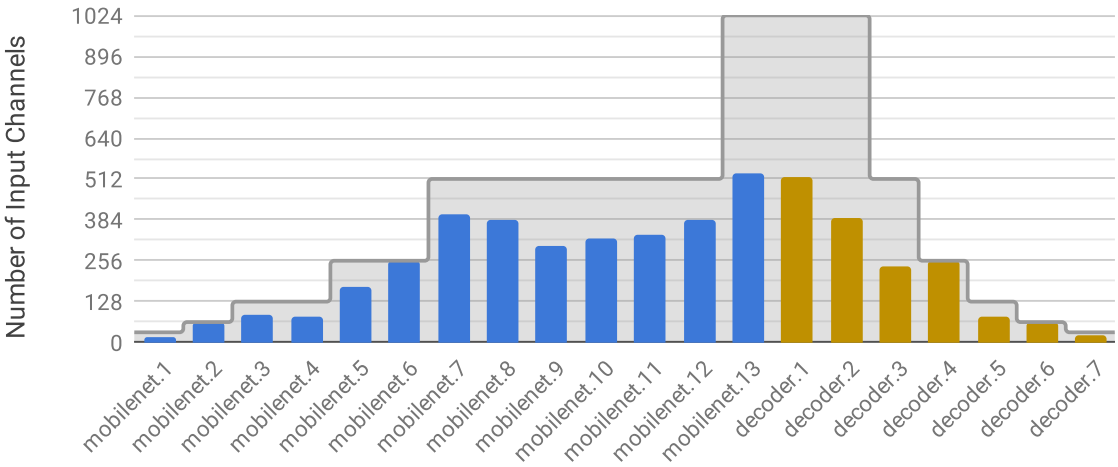
Figure 4-19: Accelerator-friendly FastDepth DNN topology. The two key modifications are in the decoder: (1) shifting skip connections to terminate *before* nearest-neighbor interpolation and downsampling feature maps passed along the connections, and (2) replacing the first 5×5 convolution with a cascade of two 3×3 convolutions.

Due to this retraining and repruning process, the channel dimensions in hidden layers change. Figure 4-20 compares the shape of the original FastDepth and the modified accelerator-friendly FastDepth. The shaded area represents the original topology of both networks prior to pruning. The topologies are very similar with the single major difference being a wider bottleneck in the accelerator-friendly network; this is due to our expansion of the first decoding layer with a 5×5 convolution into two cascaded layers with 3×3 convolutions. The blue and yellow columns represent encoder and decoder layer shapes after channel pruning. A key observation is that to compensate for an extra layer with a large channel dimension in the bottleneck region of the modified network, several other layers get pruned more.

Table 4.4 summarizes the impact of the model modifications we make in the accelerator-friendly FastDepth network. The number of MACs, the δ_1 accuracy, and the RMSE all remain similar. The number of weights in the DNN increases by about 28% — this is largely due to both having extra decoding layer for cascaded 3×3 convolutions as well as having larger channel dimensions in some layers after pruning. However, the expected number of memory accesses for feature maps is lower for the modified network. Sources of reduction include: (1) the downsampling of feature maps passed along skip connections, which reduces reads and writes done for these feature maps, and (2) the decomposition of interpolated 5×5 convolutions into non-interpolated 3×3 convolutions, which removes the interpolation operation and reduces the feature map sizes being read in from memory.



(a) original FastDepth network topology



(b) modified FastDepth network topology

Figure 4-20: Comparing FastDepth network topologies after channel pruning with NetAdapt [4]. Figures show the number of input channels to each layer in the network. The shaded part represents the topology before pruning. The very first layer to the network (`mobilenet.0`) is not shown since the channel size of the input fed into the network remains fixed at 3 channels (RGB). Overall, the shapes of the two topologies look similar. The modified network contains an extra layer at the beginning of its decoder; to compensate for this, several other layers get pruned more.

Metric	Original DNN (32b `float`)	Modified DNN (32b `float`)	Quantized DNN (8b-10b `int`)
# **Weights** [10^6]	1.34	1.71	1.71
# **MACs** [10^9]	0.37	0.38	0.38
Accuracy (δ_1 [%])	77.1	77.2	77.0
RMSE [cm]	60.4	59.5	59.5
Skip Connection R/W	10.4 MB	2.4 MB	0.6 MB
Input and Output R/W	27.2 MB	24.8 MB	6.2 MB
Total Feature Map R/W	37.6 MB	27.2 MB	6.8 MB
Weights R/W	5.4 MB	6.8 MB	2.2 MB
Total Data R/W	43 MB	34 MB	9 MB

Table 4.4: Impact of FastDepth network modifications on accuracy and data movement (reads and writes to off-chip DRAM). Compared to the original DNN, our modified accelerator-friendly DNN achieves equivalent accuracy with a 21% reduction in DRAM accesses. Quantization to 8-bit activations and 10-bit weights results in an additional 75% reduction in data movement, with negligible degradation of accuracy.

4.4.2 Integer Quantization

As first discussed in Section 1.2.4, quantization refers to the reduction of bitwidths used to represent weights and activations of a neural network. This bitwidth reduction translates to reductions in data movement (fewer bits need to be transferred), in storage cost (fewer bits need to be stored), and in compute time and energy per MAC (fewer bits need to be multiplied and accumulated, allowing for smaller arithmetic blocks to be used). However, reduced bitwidth — also termed as reduced precision — effectively limits how many unique values a weight or activation can take on. This has implications on network accuracy. For example, when comparing a quantized model against an original non-quantized model, reduced precision will limit how well the quantized model can approximate the non-quantized model, which will translate to accuracy degradation. Hence, the tradeoff between reducing precision and incurring accuracy loss is a significant design consideration when quantizing neural networks.

Quantization for image classification networks has been extensively explored, with reasonably high post-quantization network accuracy being achieved with 2-bit and 1-bit quantization [66–69]. Successful quantization with such low bitwidths is partly due to the relatively simple nature of the image classification task, where the output is 1D

(a vector of class label predictions). Conceptually, while the quantization of activation values to lower bitwidths may be more lossy, as long as the output class label ordering remains similar, the overall network accuracy will not degrade significantly.

The same cannot be said for networks that perform dense prediction tasks; such networks produce outputs that are multidimensional feature maps where individual pixel values have meaning (e.g., in depth estimation networks, each pixel in the output feature map corresponds to a predicted depth measurement). If feature map activations and filter weights in the network are quantized at inference-time, the output feature map will need to be de-quantized for output pixels to preserve meaning (e.g., for a depth estimation network, an output feature map value that is a quantized integer needs to be de-quantized into a floating point value representing a depth measurement in specific units). This suggests that quantization impacts output values and overall accuracy more directly for dense prediction networks.

When training and evaluating FastDepth in PyTorch, we use 32-bit floating-point precision for all tensors (i.e., weights, biases, and activations). When quantizing the network for deployment onto an FPGA, we target 8-bit integer precision for weights and activations.[10] Since every hidden layer in FastDepth is preceded by a ReLU activation function that zeroes out negative values, activations will always be greater than or equal to zero; we therefore represent activations with 8-bit unsigned values. Weights, however, can be positive or negative and thus are represented with 8-bit signed values. We do not target a particular bitwidth for biases, since there are relatively few of them per layer, and they do not add significantly to compute or data movement. We ultimately use 32-bit signed integer precision for biases; this bias precision is informed by our quantization methodology.

Quantization Methodology — Introduction

Quantizing a set of values can be interpreted as mapping the range of those values to a target number of levels. For example, 8-bit unsigned precision corresponds to 256 target levels (ranging in value from 0 to 255). If we were to quantize a set of

[10] Several weight values will need to be 10 bits, as explained at the end of this section.

values ranging from 0 to 0.5 to this precision, we would divide the range of 0.5 into 256 subranges and map each subrange to a level (e.g. values between 0.498 and 0.5 would correspond to a quantized value of 255).

An alternative interpretation of quantization is determining a numerical factor that, when multiplied with the values to be quantized and potentially rounded, results in values of the desired precision. In the example described above, this factor — the quantization factor — would be $255/0.5 = 510$. In this case, 0.498-0.499 would correspond to a quantized value of 254, while 0.499-0.5 would correspond to 255. Since this approach involves multiplication by a fixed factor, it lends itself better to a hardware-friendly implementation; the quantization factor can be constrained to be a power of two for efficient bit shift-based multiplication. This approach forms the basis of our quantization methodology.

Tensor-wise vs. Channel-wise Quantization

A key design consideration when quantizing FastDepth weights and activations is whether to quantize over an entire tensor or over a slice of the tensor along a specific dimension. The range of values in a particular tensor slice is likely to be smaller than the range of values across the entire tensor; this makes quantization over a tensor slice more fine-grained, which reduces accuracy degradation due to quantization. However, such a quantization scheme requires an implementation that is more adaptive and can signal unique quantization factors for every tensor slice.

In our experiments, weights appear to be the datatype more sensitive to quantization; this is reasonable to expect, as each individual weight value is used far more often than any individual activation value when computing MACs in a layer. We find that tensor-wise quantization of weights lowers network accuracy to sub-1% and therefore use channel-wise quantization instead, where every output channel of the weight tensor is quantized separately. This is consistent with observations made in [124]. For activations, we use tensor-wise quantization. Applying this quantization scheme to each layer in FastDepth results in less than 1% overall accuracy loss when compared to the unquantized network. We do not perform any retraining after quantization.

Quantization Methodology — Walkthrough

We now explain our quantization methodology more formulaically. In developing this methodology, we reference [64, 124] and draw inspiration from the quantization algorithms in the Neural Network Distiller [125]. The equations describing quantization of weights and activations are given in equations 4.1a and 4.1b. Q_a and Q_w are quantization factors for activations and weights, respectively. We use tensor-wise quantization for activations, so Q_a is a single-element vector. It differs from layer to layer and is determined prior to inference-time based on an average of activation ranges over a select validation set. We use channel-wise quantization for weights, so Q_w is a vector of C elements, where C equals the number of output channels in the weight tensor. This vector also differs from layer to layer and is based on channel-wise ranges in model weights (that are known and fixed post-training).

$$\text{activation}_{\text{8b-int}} = Q_a \times \text{activation}_{\text{32b-fp}} \tag{4.1a}$$

$$\text{weight}_{\text{8b-int}} = Q_w \times \text{weight}_{\text{32b-fp}} \tag{4.1b}$$

The products of these quantized weights and activations will be accumulated until MACs for a given layer are done. The resulting values will need to be added with a bias before passing through a ReLU activation function to complete the computation of the layer. Without quantization, we would have:

$$\text{output}_{\text{32b-fp}} = \left[\sum \text{activation}_{\text{32b-fp}} \times \text{weight}_{\text{32b-fp}}\right] + \text{bias}_{\text{32b-fp}} \tag{4.2}$$

We need to maintain this expression when quantizing values. This means that when multiplying activations and weights by their respective quantization factors, we also need to multiply the bias by the product of those quantization factors:

$$Q_a \times Q_w \times \text{output}_{\text{32b-fp}} = \Big[\sum (Q_a \times \text{activation}_{\text{32b-fp}}) \times (Q_w \times \text{weight}_{\text{32b-fp}})\Big] + Q_a \times Q_w \times \text{bias}_{\text{32b-fp}} \tag{4.3}$$

Empirically, we find the product $(Q_a \times Q_w \times \text{bias}_{\text{32b-fp}})$ to produce values less than $2^{32} - 1$. These values can be rounded to yield 32-bit integers. We therefore have the following calculation for an integer bias to be used in our quantization scheme:

$$\text{bias}_{\text{32b-int}} = Q_a \times Q_w \times \text{bias}_{\text{32b-fp}} \tag{4.4}$$

Substituting into equation 4.3, we now have:

$$\begin{aligned} Q_a \times Q_w \times \text{output}_{\text{32b-fp}} &= \Big[\sum \text{activation}_{\text{8b-int}} \times \text{weight}_{\text{8b-int}}\Big] + \text{bias}_{\text{32b-int}} \\ &= \text{output}_{\text{32b-int}} \end{aligned} \tag{4.5}$$

The right-hand side expression corresponds to the 32-bit integer-valued output feature map computed by the accelerator for a certain layer, say layer L. If this is the final layer in the network and we wish to obtain the de-quantized output feature map, we simply divide the output feature map by $Q_a \times Q_w$ for that layer:

$$\text{output}_{\text{32b-fp}} \text{ for layer } L = \frac{\text{output}_{\text{32b-int}} \text{ for layer } L}{Q_a(L) \times Q_w(L)} \tag{4.6}$$

However, if this layer is followed by some layer $L + 1$, there is an additional step, since the output from layer L will be fed as an input to layer $L + 1$. We need to first de-quantize the output feature map using quantization factors for layer L, and then quantize the resulting activations using Q_a for layer $L + 1$.

$$\begin{aligned} \text{activation}_{\text{8b-int}} \text{ for layer } L+1 &= \frac{\text{output}_{\text{32b-int}} \text{ for layer } L}{Q_a(L) \times Q_w(L)} \times Q_a(L+1) \\ &= \text{output}_{\text{32b-int}} \text{ for layer } L \times \frac{Q_a(L+1)}{Q_a(L) \times Q_w(L)} \end{aligned} \tag{4.7}$$

The expression $\frac{Q_a(L+1)}{Q_a(L)\times Q_w(L)}$ represents the intermediate quantization factor between hidden layers in the network. It is a vector of C elements, where C is the number of channels in the activation between the layers. Since the Q_a and Q_w factors for all layers are known at inference-time, these intermediate quantization factors can be precomputed and read in alongside weights and biases.

Hardware-friendly Quantization Factors Our quantization scheme is designed to be performed on the accelerator on-the-fly as output feature map values stream from the compute core. Multiplying streaming outputs by quantization factors can become area-costly as well as a computational bottleneck if additional multipliers are used, especially since these factors can be large integer values. Hence, we constrain the quantization factors Q_a and Q_w to be powers of 2. This simplifies multiplications to efficient bit-shit operations. We find that this constraint still allows us to successfully quantize all layers while maintaining network accuracy. Furthermore, instead of loading, storing, and passing around large quantization factors on-chip, we can simply use the correponding smaller bit-shift values instead.

Handling Decomposed Weights As discussed in Section 4.4.1, several of the FastDepth layers with 5×5 convolutions will be decomposed when being mapped onto the FPGA-based accelerator. This decomposition creates four 3×3 kernels from the 5×5 kernel by pre-adding values within 2×2 windows. Since the values in the 5×5 kernel are initially quantized to 8 bits, the values in each of the decomposed 3×3 kernels will be sums of four 8-bit values. To avoid overflow, these decomposed kernels must be allocated 10 bits per value. Hence, all datapaths and buffers in our FastDepth accelerator assume that incoming weights have a bitwidth of 10.

4.5 Mapping FastDepth onto the Accelerator

4.5.1 Tiling Feature Maps

Accelerators typically have limited onboard compute and storage resources, meaning that there is a limit to the size of an input that can be processed at once. The FastDepth accelerator is designed to handle feature maps that are $7\times7\times C$ in shape, where C is the number of input feature map channels. The largest C supported by the on-chip buffers exceeds 520, the largest channel dimension in the FastDepth DNN.

The feature maps involved in most FastDepth layers are larger than 7×7 in the height and width dimensions. For them to be processed within the accelerator, they are first partitioned into a set of $7\times7\times C$ feature map chunks; this set of chunks (referred to as tiles) is then iterated through to cover the entire larger feature map.

The motivation to select 7 as the tile height and tile width comes from the observation that all feature maps within the FastDepth network have dimensions divisible by 7. In other words, the tile size is chosen to be the greatest common factor of feature map sizes. When bringing in feature maps on-chip for computation, we do not tile along the channel dimension, as this would unnecessarily complicate channel-wise aggregation during pointwise convolutions. Our design prioritizes the ability to bring in all channels necessary for complete computation of both depthwise and pointwise convolution over a single tile at a time, with the major limiting factor here being on-chip storage. If our selected tile size multiplied by the largest channel dimension had been too large for on-chip storage, the tile size would have been made smaller.

Padding the Input Feature Map Tiles

A convolution with a kernel size greater than 1 will result in reduced dimensions in the output feature map, unless the input feature map is padded prior to convolution. As illustrated in Figure 4-21, for feature map sizes to be preserved during 3×3 convolution, the height and width of the input feature map must be padded with a single row or column of elements on each side. Therefore, since the FastDepth accelerator computes tiles of shape $P_T \times Q_T \times C$ with $P_T = Q_T = 7$, the input tiles to the

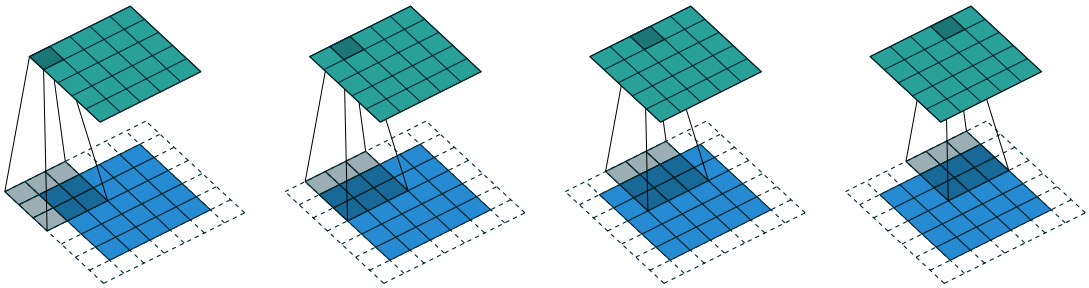


Figure 4-21: For feature map sizes to be preserved during 3×3 convolution, the height and width of the input feature map must be padded with a single row or column of elements on each side. Here, the input feature map is shown in blue, the kernel is shown in grey, and the output feature map is shown in teal. Figure taken from [5].

accelerator must be of shape $H_T \times W_T \times C$ with $H_T = W_T = 9$.

Padding is closely intertwined with the tiling process described above — an input feature map of shape $H \times W \times C$ must first be padded (with zeros) to a shape of $(H+2) \times (W+2) \times C$ and then tiled along the height and width dimension to produce $9 \times 9 \times C$ tiles. This padding and tiling process is depicted in Figure 4-22.

Suppose that we have an input feature map of shape $14 \times 14 \times C$. This feature map is first padded to a shape of $16 \times 16 \times C$ and then tiled into four tiles, each of shape $9 \times 9 \times C$. Note that tiling here does not produce tiles of shape $8 \times 8 \times C$, as would have been expected if one were to simply partition the padded feature map into four distinct chunks. Such partitioning, however, will not work well with the sliding nature of convolutions, and will introduce inter-tile dependencies, called halos, along tile edges [96]. To handle these halos, the tiles are made slightly larger — in our case, by overlapping with one column and one row of an adjacent tile. These overlapping regions are marked with a diagonal pattern in Figure 4-22.

Padding and tiling does not involve computation and is largely a memory-bound task that requires reorganization of how a feature map is stored in memory. In our current design, we offload the padding and tiling of feature maps to off-chip processing.

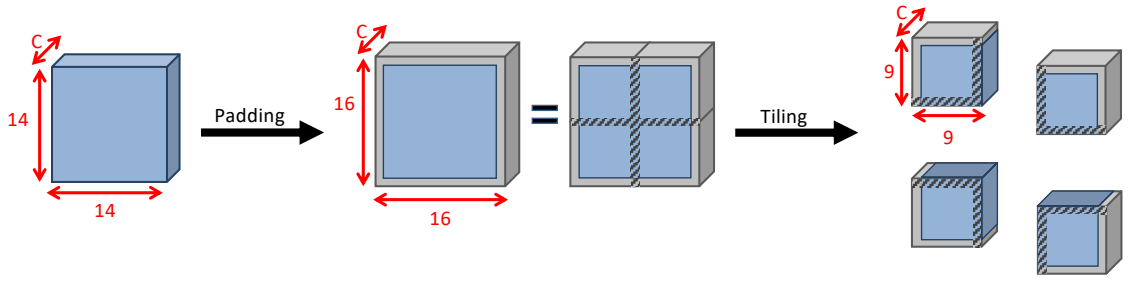


Figure 4-22: Padding and tiling input feature maps for 3×3 convolutions. In the FastDepth accelerator, the output feature map tile height and width are set to be 7×7, meaning that input feature map tiles must have a height and width of 9×9. This figure illustrates an example of how a 14×14×C input feature map is padded and then tiled into four 9×9×C tiles. Overlapping regions (called halos) amongst adjacent tiles are depicted with a diagonal pattern.

4.5.2 Mapping FastDepth Layers

This section summarizes the various layer types present in our modified accelerator-friendly FastDepth DNN, as well as how the FastDepth accelerator supports them.

Depthwise Separable Layers with 3×3 Convolutions and Stride of 1

These convolutions are prevalent throughout the FastDepth network: they are found in the MobileNet encoder as well as in the first two layers of the decoder. These convolutions are natively supported by our dataflow and accelerator design, described extensively throughout Sections 4.2 and 4.3, respectively.

Depthwise Separable Layers with 3×3 Convolutions and Stride of 2

These convolutions are interspersed throughout the MobileNet encoder where they gradually reduce the spatial (height and width) of intermediate feature maps. The primary difference between supporting a stride of 2 versus a stride of 1 is that more input feature map data needs to be read in for the same-sized output tile to be computed. Our accelerator supports this by offering enough on-chip buffer storage to cache up to four input feature map tiles that, with striding by 2, would yield a single output feature map tile. This is discussed in greater detail in Section 4.3.2.

Decomposable 5×5 Depthwise Separable Layers

These layers exhibit a specific structure: they consist of 5×5 convolutions that immediately follow nearest-neighbor interpolation. Such layers make up the majority of the FastDepth decoder. We handle these by observing that the interpolation with 5×5 convolution can be grouped and decomposed into four smaller 3×3 convolutions. The latter are then natively supported by our accelerator. This decomposition is explained in Section 4.4.1.

Interleaving of Decomposed Outputs

Each of the four 3×3 convolutions resulting from the decomposition mentioned above generate a distinct output feature map. These need to be interleaved to produce a valid input for the subsequent layer in the network. We aim to add interleaving capability within the accelerator to avoid performing this off-chip. We note that interleaving requires handling all four feature maps at once; this is in fact similar to bringing in four feature map tiles for convolutions with a stride of 2. Hence, we reuse the buffering and tile alignment logic that is described in Section 4.3.2. This completes our handling of decomposable 5×5 depthwise separable layers.

Additive Skip Connections

The skip connections present in FastDepth all terminate at decomposable 5×5 depthwise separable layers. They need to be added after the decomposed outputs are interleaved. Our accelerator supports this in a straightforward manner, bringing residual feature map tiles on-chip in alongside the feature map tiles being interleaved; addition then takes place on a row-by-row tile-by-tile basis, as described in Section 4.3.2.

Standalone Pointwise Layers

The last layer in the FastDepth networks consists solely of a pointwise convolution, i.e., there is no depthwise convolution preceding it. The purpose of this layer is to perform channel-wise aggregation at the end of the network to produce a single-

channel depth map output. Our FastDepth accelerator can easily support this layer by first performing an identity depthwise convolution (with an identity kernel). However, this does add some computation overhead as the input feature map to this layer has high height and width dimensionality, meaning it involves processing a large number of tiles. We observe that although this last layer computes just 1% of all MACs in the network, it accounts for around 3% of total network runtime in our implementation.

Standard Non-Depthwise-Separable Layers

The first layer in the FastDepth network cannot be expressed as a depthwise separable layer. This layer consists of a standard 3×3 convolution with 3 input channels and 16 output channels. As this type of layer is not natively supported by the accelerator, we consider two ways of handling it: (1) offloading the entire layer computation to off-chip, e.g. on a CPU/GPU, or (2) representing this layer via a pseudo-depthwise-separable operation. The first option is less desirable as it involves off-chip compute that may introduce scheduling delays, e.g. if the CPU/GPU are switching between other tasks in a system. We therefore implement the latter option — by replicating the 3-channel input 16 times, we can compute the 48 intermediate feature maps and aggregate them as part of a pseudo-depthwise operation. Afterwards, an identity pointwise operation can be applied to yield the correct 16-channel output from the layer. This requires adding a control flag for the first layer in our depthwise convolution logic but otherwise makes use of our existing dataflow and accelerator design. However, the expanded input channel dimension and a non-optimal mapping of this layer type results in inefficient use of the accelerator, which incurs a runtime overhead. We observe that this initial layer is the slowest and accounts for around 10% of the entire network runtime in our implementation.

4.5.3 Utilization of the PE Array

The PE array that forms the compute core of our FastDepth accelerator design is described in detail in Section 4.3.1. This section discusses utilization of that PE

array, namely how many — and how often — PEs in the array are active during runtime. We first analyse spatial utilization, then temporal utilization, and finally combine the two to obtain an overall PE array utilization rate.

Spatial Utilization of PEs

In our analysis, spatial utilization refers to *how many* of the PEs within the array are active during runtime. This is one measure of how well individual layers in the FastDepth network map onto our accelerator design. This mapping is influenced by (1) the PE array size (e.g., how many rows or blocks of PEs there are), and (2) layer shapes (e.g., how many input or output channels are computed within a layer).

The PE array in the FastDepth accelerator is designed to work on multiple channels in parallel. As described in Section 4.3.1, channel-wise parallelism served as motivation for building up the array via PE blocks for depthwise computation. Specifically, each PE block is meant to work on a single input channel at a time. For an array that consists of B blocks, B input channels can be processed at once. This means that during depthwise convolution, sets of B input channels are iterated through until computation for all channels is complete — the total number of passes through the array can thus be expressed as $\lceil C/B \rceil$, where C refers to the number of input channels in the layer. If C is divisible by B, then the depthwise convolution in that layer can map onto the array with perfect spatial utilization, since all PE blocks will be used. If, however, C not divisible by B, then the final pass will only use $C \bmod B$ blocks; the remaining blocks will be idle for that last pass. From this we can estimate spatial utilization for depthwise convolutions using the following equation:

$$\text{spatial util}_{\text{DW}} = \begin{cases} 100\%, & \text{if } C \bmod B = 0 \\ 100\% \times \left(\frac{\lceil C/B \rceil - 1}{\lceil C/B \rceil} + \frac{C \bmod B}{B} \times \frac{1}{\lceil C/B \rceil} \right), & \text{otherwise} \end{cases} \tag{4.8}$$

where $\frac{\lceil C/B \rceil - 1}{\lceil C/B \rceil}$ refers to the number of passes in which all PE blocks are active, $\frac{1}{\lceil C/B \rceil}$ refers to the final pass, and $\frac{C \bmod B}{B}$ refers to the fraction of PE blocks active

on that final pass. This equation is to be applied on a per-layer basis as C will change from layer to layer. B is fixed to be 8 in our PE array design. Figure 4-23 visualizes spatial utilization rates for every layer in FastDepth. Since input channel dimensions in all layers are perfectly divisible by $B = 8$, the utilization rate for depthwise convolution is 100% across all layers.

Spatial utilization for pointwise convolutions can be calculated in a similar way. Instead of considering input channels, we now consider the number of output channels, M. In the PE array, each of the B blocks consists of 3 rows of PEs, and each of those rows works on a distinct output channel. Analogous to how we map C input channels onto B blocks, we now map M output channels onto $3B$ rows. This yields an analogous equation to compute spatial utilization for pointwise convolutions:

$$\text{spatial util}_{\text{PW}} = \begin{cases} 100\%, & \text{if } M \bmod 3B = 0 \\ 100\% \times \left(\frac{\lceil M/3B \rceil - 1}{\lceil M/3B \rceil} + \frac{M \bmod 3B}{3B} \times \frac{1}{\lceil M/3B \rceil} \right), & \text{otherwise} \end{cases} \tag{4.9}$$

Not all output channel dimensions in FastDepth layers are perfectly divisible by $3B = 24$, resulting in several layers with $< 100\%$ utilization. Larger oC correspond to more passes through the array during pointwise convolution, which mitigates the utilization hit of inactive PEs on the final pass. Hence, utilization percentage rates for layers in the middle of the network tend to be in the high 90s. The most inefficient layer is the last decoding layer as it produces a single output channel and therefore uses just one of the 24 rows throughout the duration of that pointwise convolution.

This utilization analysis so far focused on spatial parameters such as array size and layer shape. It has not taken into account any temporal statistics, e.g., how long depthwise convolutions take relative to pointwise convolutions, or how long each layer takes relative to the others. Those statistics factor into temporal utilization.

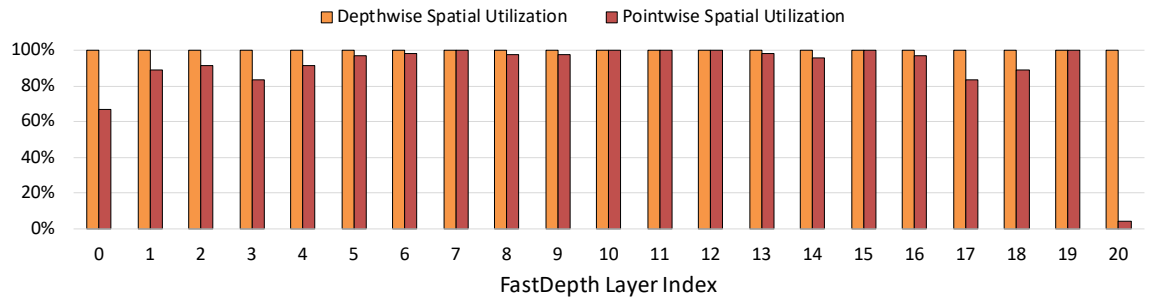


Figure 4-23: Layer-by-layer spatial utilization of the PE array. Our accelerator achieves high spatial utilization for almost all FastDepth layers. The only significant exception is the final layer that produces just a single output channel.

Temporal Utilization of PEs

Whereas spatial utilization estimates how many PEs are active, temporal utilization estimates *how often* PEs are active. One straightforward approach to computing this is by counting clock cycles when PEs are active and dividing those by the total cycle count for depthwise or pointwise computation:

$$\text{temporal util}_{\text{DW}} = \frac{\#\text{ clock cycles when PEs are active}}{\#\text{ clock cycles for depthwise convolution}} \tag{4.10}$$

$$\text{temporal util}_{\text{PW}} = \frac{\#\text{ clock cycles when PEs are active}}{\#\text{ clock cycles for pointwise convolution}} \tag{4.11}$$

Figure 4-24 visualizes temporal utilization rates for every layer in FastDepth. Sub-100% utilization is due to PE idleness during data read-out and read-in.[11] During depthwise computation, idleness occurs if more time is needed for input feature map GLBs to fill up. During pointwise computation, idleness occurs if more time is needed for output feature map buffers to be fully read out. Challenges in hiding memory latency and reducing idleness are further discussed in Section 4.6.5.

[11]Some, but not all, of this memory latency is hidden through data pre-fetching or pre-loading.

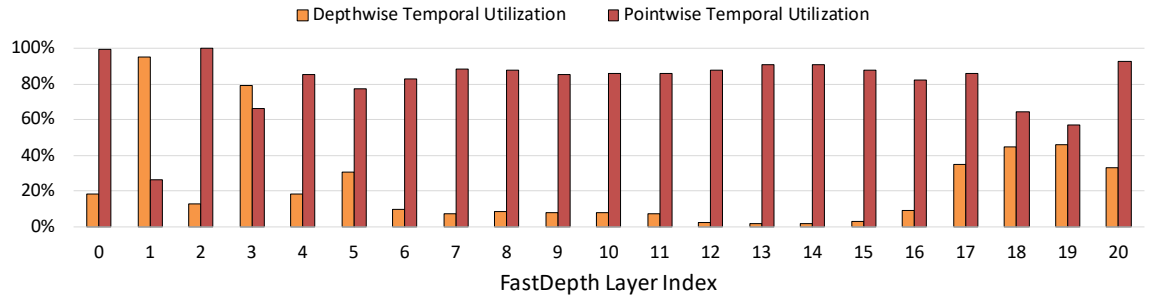


Figure 4-24: Layer-by-layer temporal utilization of the PE array. There are two sources of temporal utilization hits: PE idleness while feature map GLBs are filled up with data from DRAM (this impacts depthwise temporal utilization) and PE idleness while output feature maps are written out to DRAM (this impacts pointwise temporal utilization). Since there is far less depthwise computation in FastDepth than there is pointwise computation, the depthwise temporal utilization hit is far more noticeable.

Overall Utilization

The spatial and temporal utilization rates analysed above can be combined to determine a more comprehensive utilization rate for the depthwise and pointwise convolutions in each individual layer:

$$\text{util}_{\text{DW}} = \text{spatial util}_{\text{DW}} \times \text{temporal util}_{\text{DW}} \tag{4.12}$$

$$\text{util}_{\text{PW}} = \text{spatial util}_{\text{PW}} \times \text{temporal util}_{\text{PW}} \tag{4.13}$$

The layer utilization rate is then a weighted sum of these two utilizations, with the weighing factor being the fraction of clock cycles spent on depthwise vs. pointwise convolution within a layer:

$$\text{layer utilization} = \frac{\#\text{ clock cycles}_{\text{DW}}}{\#\text{ cycles for layer}} \times \text{util}_{\text{DW}} + \frac{\#\text{ clock cycles}_{\text{PW}}}{\#\text{ cycles for layer}} \times \text{util}_{\text{PW}} \tag{4.14}$$

Layer utilization rates can be aggregated over all layers, where the weighing factor is fraction of the network that each layer constitutes time-wise. This yields an effective utilization rate for the network as a whole:

$$\text{network utilization} = \sum_{\text{layers}} \frac{\#\text{ clock cycles for layer}}{\#\text{ clock cycles for network}} \times \text{layer utilization} \quad (4.15)$$

This yields an overall PE array utilization rate of 46.1%. In our discussion of mapping FastDepth layers in Section 4.5.2, we establish that the first layer and final layer of the network do not map well onto our accelerator design. If we remove these two layers from consideration, we obtain an overall PE array utilization rate of 50.2%. Hence, the utilization hit from these two layers is not overly significant; this is reasonable to expect since they account for only a small fraction of the entire network. Instead, the primary utilization hit comes from idleness during on-chip memory read-in and read-out (i.e., when filling up GLBs with inputs from DRAM, or emptying out PE block-level buffers by sending outputs to DRAM). As mentioned earlier, this causes a noticeable temporal utilization hit, which then factors into the overall utilization rate.

4.6 Implementation Results

When implementing our accelerator design, we target deployment on an embedded FPGA. We select the Ultra96 development board [118] for its small form factor, low power consumption, and easy interfacing through the on-board ARM processor running Linux. We use Xilinx's Vivado Design Suite 2018.1 for all accelerator logic simulation, synthesis, and implementation.

4.6.1 System Overview

Our target platform — the Ultra96 board — has an ARM-based Zynq UltraScale+ MPSoC. This MPSoc incorporates a Processing System (PS) unit that runs a Linux kernel and a Programmable Logic (PL) unit that runs user logic designs. Our system for FastDepth inference uses both the PS and the PL: while our accelerator logic is implemented on the PL, our I/O to the accelerator is handled by the PS. We make

	LUT	LUTRAM	FF	BRAM (36K)	DSP
Available	70560	28800	141120	216	360
Accelerator Logic	54691	8656	32004	205	353
AXI Stream FIFOs	130	24	211	0.5	0
AXI DMA	1527	121	2054	2	0
AXI Misc.	5216	796	6226	0	0
Zynq PS Reset	19	1	37	0	0
Total Used (Utilization %)	61583 (87%)	9598 (33%)	40532 (29%)	207.5 (96%)	353 (98%)

Table 4.5: Logic utilization of the accelerator deployed on the Ultra96 FPGA.

use of the PYNQ API [120] to interface with DMA to off-chip DRAM and with GPIO communicating control signals directly to/from our accelerator. Additionally, while all FastDepth layers run on the accelerator on the PL, feature map transforms (for padding and tiling) between layers takes place on the CPU in the PS.

Logic Utilization

Our design is inherently scalable through our concept of independent PE blocks. Scaling up will increase parallelism, which will generally increase speed at the cost of area and power consumption. We scale our design to fit on the Ultra96 FPGA. Table 4.5 reports utilization of logic resources for various components of the system, while Figure 4-25 shows a logic breakdown across modules within the accelerator design.

Our design uses BRAM for on-chip buffers to store model parameters and feature maps. We design buffers to be large enough to store all biases and weights for a given layer at once. In addition, each PE block has a dedicated depthwise and pointwise output buffer; BRAM usage therefore increases with every additional PE block that is instantiated. This contributes to our design's high BRAM utilization.

Our design also relies on using DSP slices within the PE array: each PE uses one DSP slice for computing MACs and another for performing bias addition. Separating these two operations and dedicating a DSP slice to each allows them to be performed in a pipelined fashion, which speeds up computation overall. However, having two DSP slices per PE rapidly depletes DSP resources since PEs are instantiated numerous

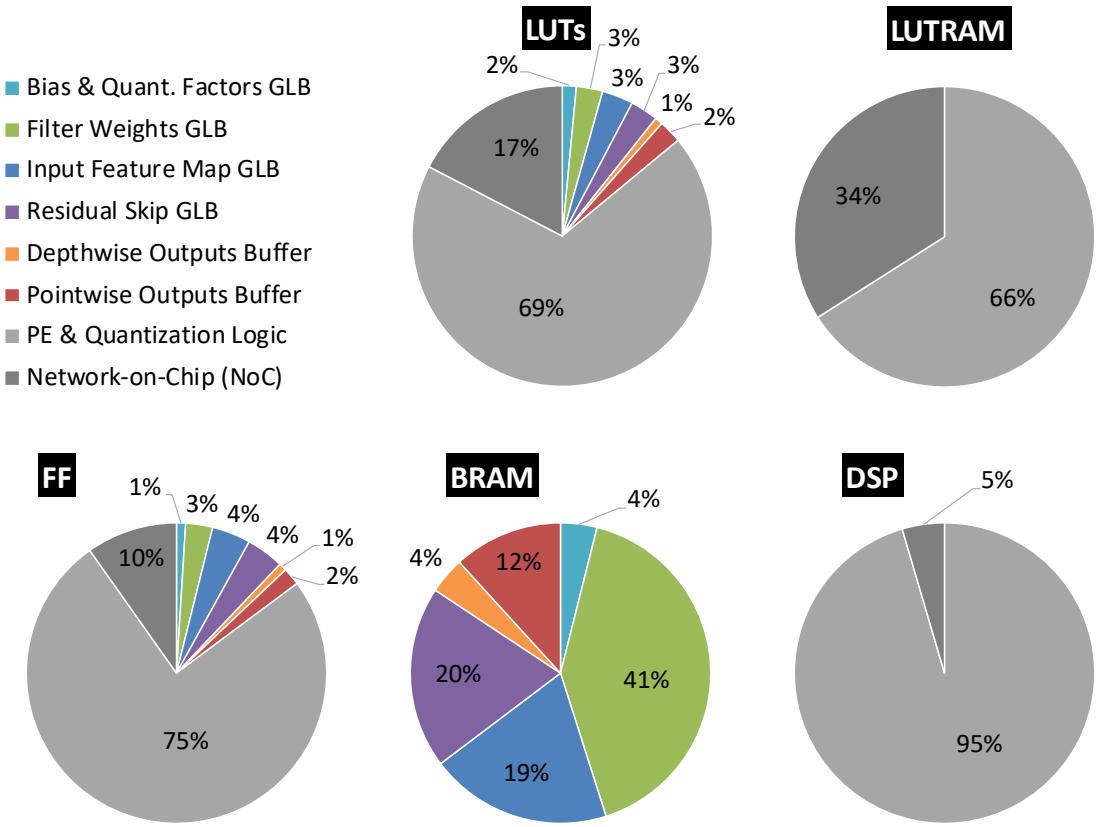


Figure 4-25: Accelerator logic breakdown by module type.

times. If needed, DSP usage could be halved by serializing all computation in a PE to go through a single DSP slice or, if there are enough LUTs available, by switching simpler computations to LUT-based resources.

From a utilization standpoint, the PE is the most critical design component as utilization will scale with the number of PEs instantiated. We therefore aim to make our reconfigurable PEs as compact as possible, without sacrificing PE compute speed. We reuse control logic and registers throughout both depthwise and pointwise convolution and rely on distributed RAM for efficient PE storage. A single PE requires around 150 registers, 34 LUTs as memory, 2 DSP slices, and 220 LUTs in total.

Timing Summary

Our accelerator operates in a single clock domain, i.e., all PEs, blocks, and buffers are clocked at the same frequency and designed to work in sync with each other. In our implemented design, the accelerator is clocked at 250 MHz. This clock is generated at the PS-PL interface and routed to all key components of the design — the DMA interface, FIFOs, and the top level wrapper for our accelerator.

When implemented at 250 MHz, our design is reported to have a worst negative slack (WNS) of 0.043 ns. We can use this to gauge the maximum frequency our design could support: $f_{\text{max}} = 1/(T_{\text{clk}} - \text{WNS})$ where $T = 1/f_{\text{clk}} = 4$ ns. From this, we estimate f_{max} to be around 252 MHz.

The critical path in a logic design is a limiting factor to how fast the design can be clocked; in our implemented design, the critical path lies within the input feature map NoC. More specifically, it is a path coming from a depthwise output buffer to a PE when inputs are being delivered to the PE array for pointwise convolution.

4.6.2 Logic Performance Analysis

This section discusses performance of the register-transfer logic for our accelerator design — namely everything implemented on the programmable logic fabric, and excluding the Zynq processing system. The following results come from a post-

implementation functional simulation of the design across all FastDepth layers.

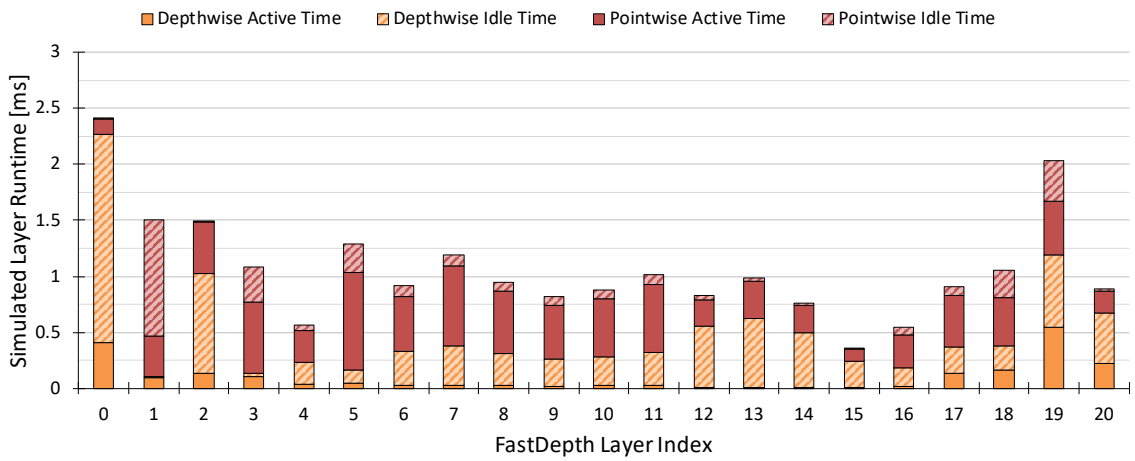


Figure 4-26: Layer-by-layer runtime in simulation (clocked at 250 MHz). Pointwise computation dominates active time, as is to be expected since there are more pointwise MACs than depthwise MACs in FastDepth. Depthwise idle time is due to PEs waiting for the input feature map GLB to fill up (which is why layers 0 and 2, requiring 4× as many input activations due to convolution strides of 2, experience high depthwise idle times). Pointwise idle time is due to PEs waiting for output buffers to clear out.

Layer-by-Layer Runtime

Figure 4-26 shows simulated runtimes across layers in FastDepth. The runtimes are broken down into active time spent on depthwise and pointwise computation as well as idle time where PEs wait for input/output data movement to complete. Depthwise idle time mainly comes from PEs waiting for input feature map tiles to finish streaming in, while pointwise idle time comes from PEs waiting for output feature map tiles to finish streaming out. Idle time varies by layer since it depends on how many tiles as well as how many channels are being processed within that layer. Overall, idle time accounts for half of the total runtime aggregated across all layers. This highlights that memory latency, the source of idle time here, is a remaining challenge that would need to be addressed in order to improve the performance of the accelerator. We discuss memory latency more in Section 4.6.5.

Layer-by-Layer Power Consumption

Figure 4-27 shows the simulated power consumption across layers, taking the switching activity of logic into account. Layers tend to consume between 1.5 and 2.5 W of power, with an observed slight dip in power consumption within the bottleneck region at the encoder-decoder interface (layers 12−15). Taking a weighted average based on what fraction of the network the layer constitutes, we estimate the overall power consumption of our accelerator to be around 1.9 W.

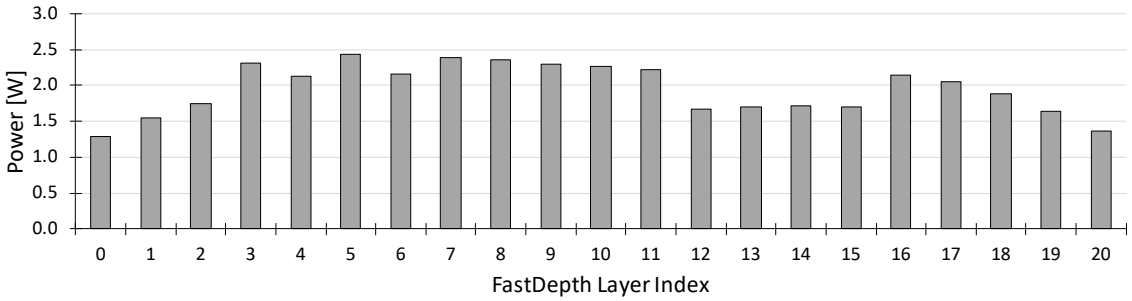


Figure 4-27: Layer-by-layer power consumption in simulation.

Figure 4-28 shows the breakdown of power consumption into dynamic and static power, as well as amongst the different sources of dynamic power. These power estimates were gathered on a per-layer basis from simulation, similar to the above analysis, and were then aggregated across layers.

From this breakdown, we observe that BRAM power consumption dominates; this is due to data movement of feature maps and parameters to and from BRAMs that make up most of our on-chip memory (input data GLBs and output data buffers). Furthermore, in our design, the read and write ports of BRAMs are set to always be enabled; however, these ports are not always both in use, e.g., after a BRAM is loaded or emptied. Thus, there is potential to reduce BRAM power consumption in our design by disabling read or write BRAM ports when they are not in use.

The next two highest sources of power consumption comes from logic elements and signals transmitted on internal wires. These are indicative of the power consumption of the network-on-chip that delivers data to and from processing elements, as well as

datapaths between GLBs and external DRAM. A potential way to reduce this power consumption could be to lower wire switching activity by gating signal paths that are not actively delivering data at a particular moment in time.

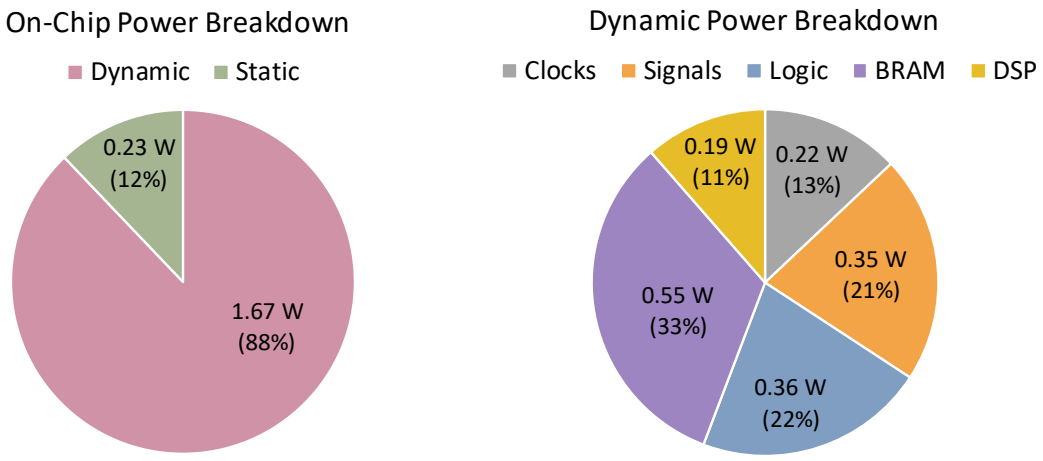


Figure 4-28: Power consumption breakdown from simulation. BRAM power consumption dominates, followed by logic power and signal power.

4.6.3 System Performance Analysis

This section now considers a complete functioning system that includes the Zynq processing system and DRAM. Results presented here come from experiments running end-to-end FastDepth inference on the Ultra96 board.

System Runtime

Figure 4-29 shows a layer-wise breakdown of FastDepth runtime, reporting data from both simulation as well as on physical hardware. Runtimes on hardware are largely in accordance with those observed in simulation, though there does appear to be some overhead. The singular difference between our design in simulation and our design on hardware is the inclusion of the AXI DMA and Zynq PS interface in the latter. This may be contributing to the observed overhead, though this is difficult to verify as both components – especially the Zynq PS interface – are difficult to simulate. The slightly more noticeable overhead on hardware (relative to in simulation) for layers

such as layer 0 and layer 2 may due to the large number of tiles being processed with a convolution stride of 2. Since tiles are fed into the accelerator in a sequential manner, reading from the DMA channel will happen more frequently for those channels.

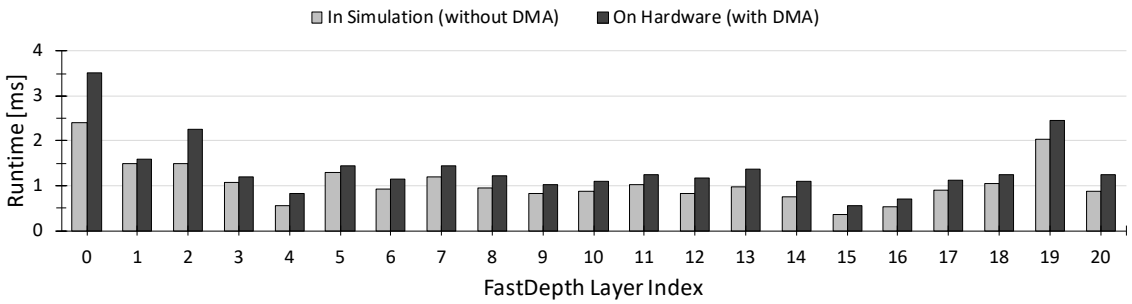


Figure 4-29: Layer-by-layer runtime on hardware (clocked at 250 MHz), compared with the previously reported layer-by-layer runtime from simulation.

FastDepth Accelerator	**Inference Runtime**	**Maximum Framerate**	**Energy Efficiency**
In Simulation	22.5 ms	44.4 fps	7.3 fps/W
On Hardware	29.0 ms	34.5 fps	5.7 fps/W

Table 4.6: FastDepth accelerator runtime and energy efficiency given average power consumption of 6.1 W. The accelerator achieves real-time inference at over 30 fps.

Summing across all layers, we get the total inference runtime for the FastDepth accelerator. Table 4.6 reports this alongside maximum supported framerates and estimated energy effiency. Our observed hardware runtime is 29% higher than simulated runtime; however, both still achieve real-time inference speeds at over 30 fps.

Up until now, we have been considering layers in isolation of any processing that happens on the CPU in between layers. This was to isolate the performance of our accelerator within the system from the system at large. Now, we take CPU processing into account as well. Figure 4-30 shows a layer-wise breakdown of system runtime, including overheads for PS-side buffer allocation as well as feature map transformations taking place between layers. The Layer Processing runtimes reported in this figure include not the accelerator runtimes that were previously reported in

Figure 4-29, but also the overhead coming from allocating PS-side DMA buffers[12] as well as copying data arrays to those buffers prior to them being streamed onto the accelerator. The Feature Map Transform runtimes encapsulate the overhead of un-tiling an output feature map from the accelerator, padding it, and then re-tiling it to be fed back into the accelerator as input for a subsequent layer. These transforms present a system-level challenge and are discussed more in Section 4.6.5.

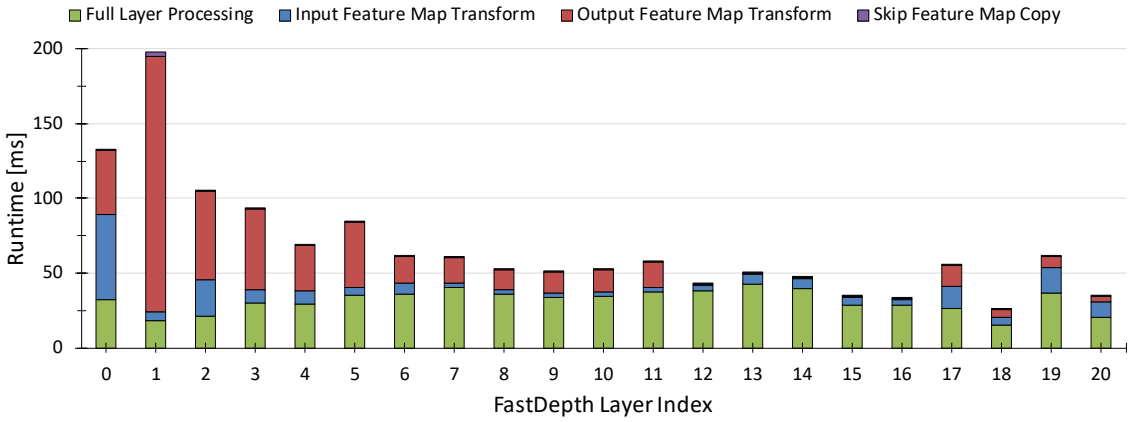


Figure 4-30: System runtime including PYNQ API calls and feature map transformations between layers. Feature map transforms involve aligning and merging output tiles, which incurs significant runtime overhead (to be discussed in Section 4.6.5).

System Power Consumption

We measure system power consumption through sensors on the Ultra96 using the PMBus protocol that is supported by the Linux kernel running on the Zynq PS. Since we use the PYNQ framework as the frontend to our system, we take advantage of the `pynq.pmbus` class that can read and record sensor values during processing. A power profile of our system during FastDepth inference is shown in Figure 4-31. At resting state, prior to the bitfile with our accelerator logic being loaded onto the FPGA, the system consumes around 4.75 W of power. Upon loading our design, power consumption rises to just above 6 W and remains steady throughout the processing of all FastDepth layers. On average, the system consumes around 6.1 W of power

[12]By calling the Contiguous Memory Allocator in the AXI DMA class of the PYNQ API.

during inference. This is about 1.2−1.4 W higher than the power consumption of the system in its idle state, which is slightly lower than the power levels that we observed on a layer-per-layer basis in simulation.[13]

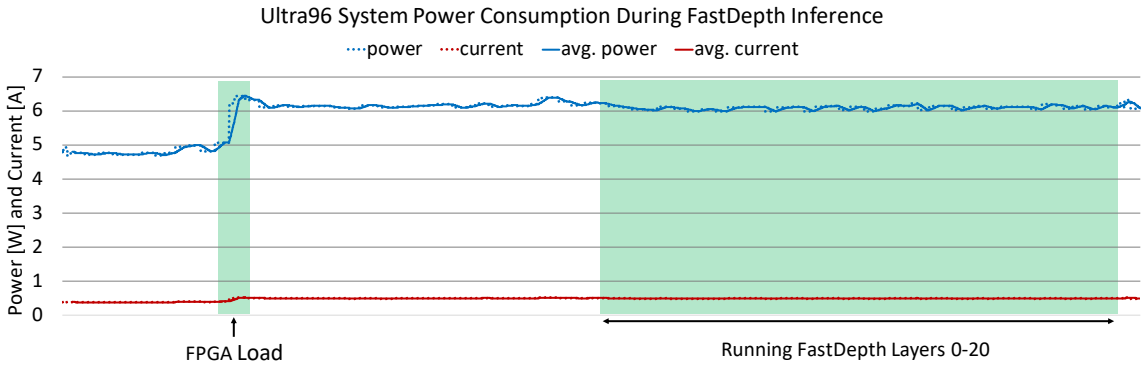


Figure 4-31: System power consumption during end-to-end FastDepth inference on the Ultra96. In its idle state, the system consumes around 4.75 W of power. During inference, the system consumes around 6.1 W.

4.6.4 External Memory Accesses

One of our contributions in this accelerator design is a dataflow and supporting architecture that seeks to minimize how many DRAM accesses are performed during layer computation. Since we are targeting an embedded FPGA system with limited on-chip memory (less than 1MB), it is infeasible to store intermediate feature maps (between layers) in their entirety on-chip. We instead focus on minimizing how many times an input feature map value, weight, or bias is read from DRAM as well as how many times an output feature map value is written to DRAM. In this context, the minimum corresponds to the size of a feature map, weight, or bias tensor, i.e. all input values will need to be brought on-chip *once* during inference and all output values will need to be written out *once*.

We compare the number of DRAM accesses in our design to these minimums in two stages. In the first stage, we assume our default design bitwidths for datatypes (e.g. feature map activations are 8 bits, weights are 10 bits). Any DRAM access overhead in

[13]It is likely that the resting state system power included idle FPGA power consumption.

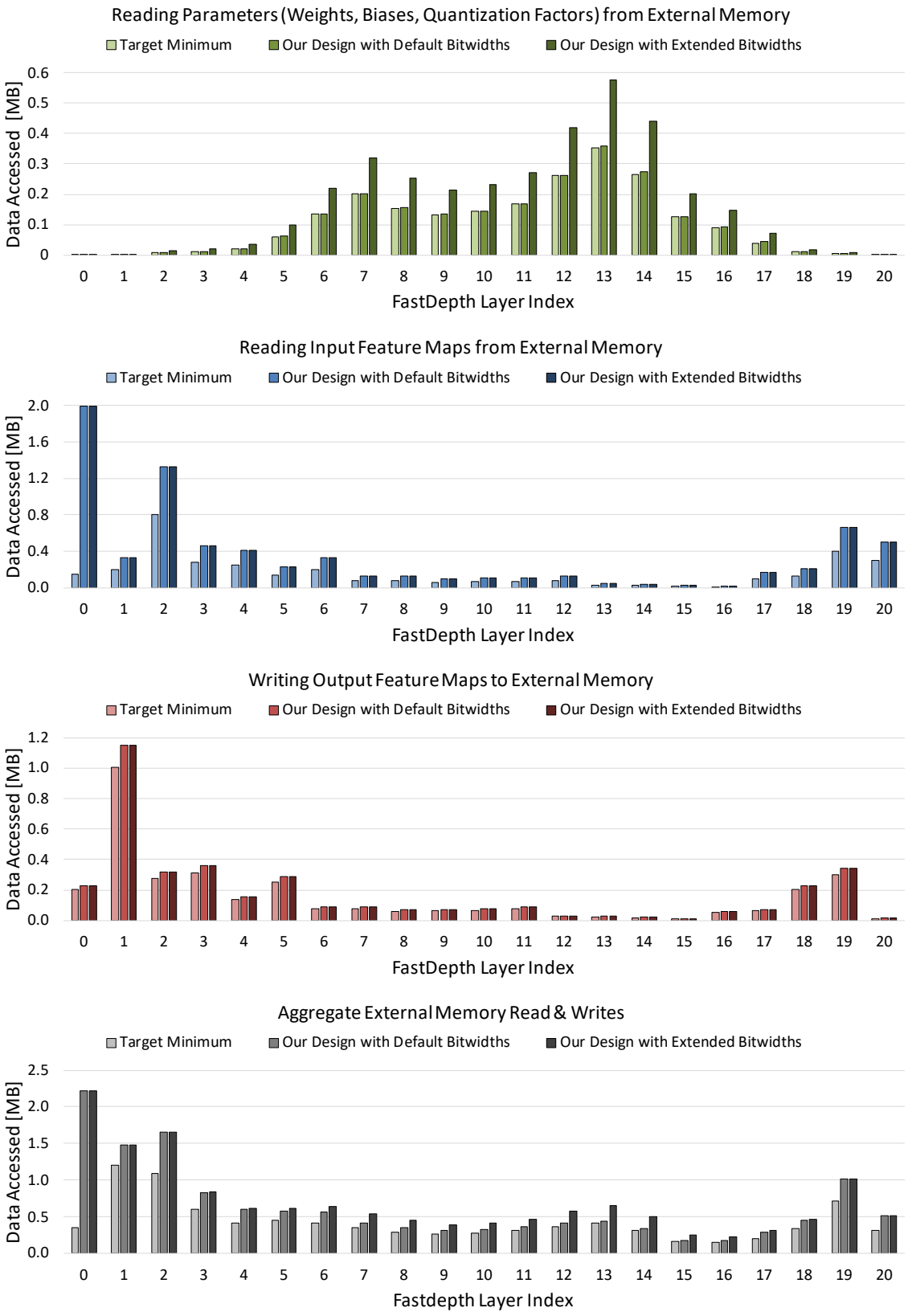


Figure 4-32: External memory accesses in our design vs. target minimums.

Datatype	Default Bitwidth	Extended Bitwidth for Alignment in DRAM
bias values	32 bits	32 bits (packed ×1 into 32-bit word)
quantization factors	5 bits	32 bits (packed ×1 into 32-bit word)
filter weights	10 bits	16 bits (packed ×2 into 32-bit word)
feature map values	8 bits	8 bits (packed ×4 into 32-bit word)

Table 4.7: Datatypes being streamed into or out of the accelerator, listed alongside their default and extended bitwidths. DRAM onboard the Ultra96 has a width of 32 bits, thus limiting our stream word size to 32 bits. We extend datatype bitwidths as necessary to facilitate packing into 32-bit words.

this analysis stage reflects redundant reads or writes. In the second analysis stage, we assume bitwidths have been extended for better alignment in DRAM. For instance, 10-bit weights do not fit well into a 32-bit word — instead, we set aside 16 bits for weights, so that two weights can then be packed into a 32-bit word. Table 4.7 summarizes the default bitwidths along with the extended bitwidths for the various datatypes in our design. Under-utilization of memory due to these extended bitwidths will therefore also factor into DRAM access overhead.

The four graphs in Figure 4-32 quantify DRAM accesses in our design relative to the minimums we define earlier. The graphs report statistics for different datatypes, both with default bitwidths as well as extended bitwidths.

- Our design achieves the target minimum in reading parameters — every individual weight, bias, and quantization factor is read from external memory only once throughout inference. This is enabled by our on-chip weight and bias GLBs that have been sized to store all parameters for any given layer at once. The only overhead in reading parameters comes from extending bitwidths due to weights being 10-bits and not fitting well into 32-bit words.

- Our design incurs overhead in reading input feature maps. This is due to padding done off-chip and padded input feature maps being stored in DRAM prior to being read in. Since our design works on 7×7 tiles and pads for 3×3 convolution, tiling and padding will introduce overhead of $(9\times9)/(7\times7) = 1.65\times$.

The impact of this overhead will be more noticeable for layers with many feature map tiles, e.g., at the beginning and end of the network.

- Our design incurs much less overhead in writing output feature maps than in reading input feature maps. The only source of overhead here is actually under-utilization of the 32-bit bandwidth of the output stream. Our accelerator outputs rows of 8-bit values at a time; these rows are 7 elements long (corresponding to our tile size of 7). This amounts to 56 bits that are padding to 64 bits prior to being serialized into a 32-bit stream. Hence, this under-utilization by 12.5% manifests itself as memory access overhead.

Overall, our design closely follows the target minimum trends for external memory accesses. Primary sources of overhead include padding in input feature maps and packing data into fixed-width 32-bit streams. Possible steps for improvement include re-evaluating whether padding off-chip can be avoided as well as performing more fine-grained packing, e.g., of weights.[14]

4.6.5 Challenges

Memory Latency Hiding

In our accelerator, there are two key sources of memory latency, both involving on-chip buffers: (1) read latency when waiting for data stored in buffers to be read out after a request for it is made, and (2) the delay in waiting for buffers (specifically our input GLBs) to be filled up with data before any can be read out.

Our design seeks to hide read latency by prefetching inputs and parameters whenever possible. Individual PEs within the PE array assert control signals to indicate that they are ready for a subsequent input or parameter. Since PEs all operate in sync, it is straightforward to predict when the PE will finish computing a given row and be ready for new incoming data. This allows us to prefetch from GLBs and keep all PEs continuously busy once computation begins.

[14]Not all weights in our design use their full bitwidth, e.g., pointwise weights that have been quantized to 8 bits. These can be packed twice as compactly into every 32-bit word as our current approach, thus lowering memory access overhead.

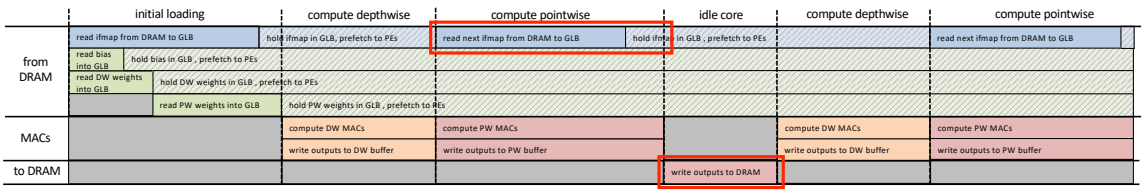


Figure 4-33: Timing diagram illustrating memory accesses overlapped with depthwise and pointwise computation. The red boxes highlight two potential sources of idle time in the PE array. Both represent challenges in hiding memory access latency.

However, computation within the PE can only begin once all GLBs are loaded. This constraint was put in place due to how rapidly depthwise or pointwise convolution can complete in certain layers. Indeed, the highly parallelized processing in our accelerator shifts the bottleneck to the memory hierarchy. We attempt to prefetch data into the GLBs just as we prefetch data into PEs, e.g., by prefetching the next layer's input feature map while the current layer's pointwise operation is being computed. For many layers, this is infeasible as GLB loading time still exceeds compute time. A similar challenge is faced with unloading pointwise output buffers at the end of layer processing. GLB load-in and output buffer read-out are the two main sources of idle time in our PE array, as highlighted in Figure 4-33. We envision that a potential way of addressing this could be by using a separate clock domain for GLB writes and output buffer reads, in hopes of clocking those ports faster to hide more of the inherent memory latency.

Feature Map Transformations

Transformations of feature maps between the output of one layer and input to the next are performed by the quad-core ARM Cortex-A53 CPU onboard the Ultra96. Since our system uses the Python frontend of the PYNQ framework [120] to interface with implemented logic, we use NumPy [126] for these transformations.

Transforming Output Feature Maps from a Given Layer. Output feature map values are read out from on-chip memory into DRAM in a continuous stream of

32-bit words. Transforming this 1-dimensional vector into a 4-dimensional[15] feature map tensor requires the following steps:

1. ***Unflattening*** the vector. Given the expected output feature map dimensions and settings (e.g., number of channels, total number of output tiles, the stride with which output tiles are computed, whether decomposition has taken place), the 1-D vector is reshaped into a multi-dimensional tensor. This allows us to easily swap axes later on to transpose dimensions and consolidate tiles.

2. ***Unpacking*** 32-bit words to 8-bit values. In the output stream, every 32-bit word packs four 8-bit feature map values. This conversion unpacks individual feature map activations. Unpacked activations are in *column-major* order.

3. ***Transposing*** to rearrange activations so that they are in *row-major* order. This can be done efficiently by creating a new view of how tensor data is read in memory[16] without creating a new copy of the entire tensor.

4. ***Rearranging*** dimensions and merging tiles. This step effectively reshapes the activations into $NCHW$ format. To achieve this, it merges output tiles in the horizontal and vertical directions. Unlike the steps above, merging dimensions from shape $(\frac{H}{H_T}, \frac{W}{W_T}, ..., C, H_T, W_T)$ to (N, C, H, W) cannot be expressed as simply a different view of tensor data in memory. It requires moving data (for the tiles) within memory. As a result, this step creates a new copy of the entire tensor, which incurs a time cost. For layers with high channel dimensionality and many feature map tiles, e.g., layers 0 to 11 shown in Figure 4-30, this step accounts for over 95% of all the time spend on output feature map transform. Layer 1 in particular suffers the most, as it has both a large output channel dimension (64) and a large tile number (16×16 tiles).

Transforming Input Feature Maps for the Next Layer. After the output feature map has been transformed via the steps described above, it is transformed

[15]Following the $NCHW$ (batch, channel, height, width) format.
[16]This uses NumPy's `stride_tricks.as_strided` method.

into an input stream for the subsequent layer. This input feature map transform assumes a starting tensor shape of (N, C, H, W) and performs the following steps:

1. ***Padding*** the height and width. This is done by creating an array of zeroes of shape $(N, C, H+2, W+2)$ and setting center elements equal to tensor values.

2. ***Tiling*** the tensor into $\frac{H}{P_T} \times \frac{W}{Q_T}$ tiles, each of shape (N, C, H_T, W_T). In our design, $P_T = Q_T = 7$ and $H_T = W_T = 9$. This can be done efficiently by modifying how tensor data is viewed in memory, without re-copying the tensor.

3. ***Transposing*** tile height and width dimensions so that the activations are in *column-major* order, i.e., of shape (N, C, W_T, H_T). This enables activations from multiple rows to be fed in parallel to PEs withing a PE block, to minimize loading times as well as to keep PEs in sync.

4. ***Parallelizing*** tiles for parallel feeding into PE blocks. There are $B = 8$ blocks, and each block works on a distinct input channel. Thus, the parallelizing step adds a new dimension, reorganizes tile data from shape (N, C, W_T, H_T) to $(N, \frac{C}{B}, B, W_T, H_T)$ and then moves the new dimension so that the final shape is $(N, \frac{C}{B}, H_T, W_T, B)$. Again, this can be done by re-viewing tensor data in memory without creating a copy.

5. ***Packing*** 8-bit values into 32-bit words. This step slices the innermost tensor dimension into sets of four values that are then packed into 32-bit words.

6. ***Flattening*** the tensor into a single vector of 32-bit values that can be streamed onto the chip for layer processing.

A flow diagram of these transforms is shown in Figure 4-34. Several of the output feature map transforms appear to be inverses of the input feature map transforms (unflattening $\rightarrow$ unpacking $\rightarrow$ merging tiles vs. tiling $\rightarrow$ packing $\rightarrow$ flattening). This leads one to question why these transforms are even performed. The transforms are made necessary by the padding and parallelization steps for input feature maps. Padding introduces new values (zeros) into the tensor, while parallelization changes

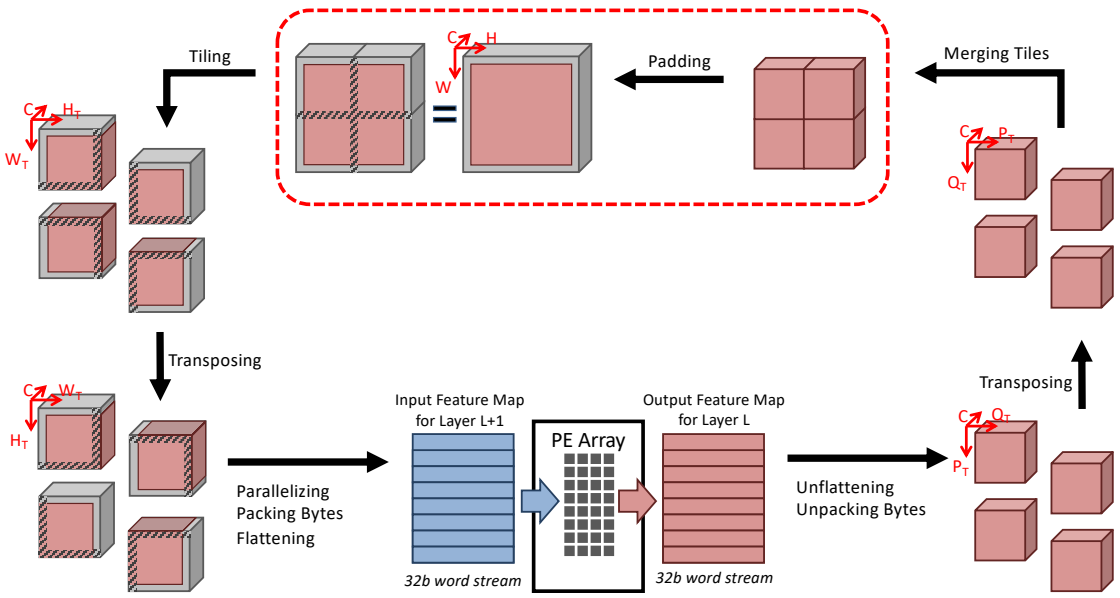


Figure 4-34: Flow diagram of feature map transformations taking place between layers. These transformations convert the output stream coming from the accelerator into a high-dimensional tensor that is padded and tiled before being fed back into the accelerator via an input stream. In our system, these transformations are done by the CPU onboard the Ultra96, while layer processing is done on the FPGA.

how tensor values are ordered. Both present a challenge when trying to avoid transforming the entire activation tensor between one layer and the next.

The parallelization step could be integrated into implementable logic by creating an extra B (=8) FIFOs and feeding output feature map tiles of every 8th channel into the appropriate FIFO. Synchronously popping the first element in all the FIFOs would be equivalent to performing the parallelization transform.

However, padding remains a challenge. In fact, padding complicates tiling: when merging tiles in the output feature map, the tiles are of shape (N, C, P_T, Q_T), but when tiling the input feature map, the tiles are of shape (N, C, H_T, W_T) and have overlapping values. The two steps are therefore not true inverses of each other. That alone necessitates transforming the entire activation tensor in our approach.

Designing specialized logic to handle padding on-the-fly as feature map values stream through could allow the output stream to directly feed back as input without any transformations. Though this would incur a logic utilization cost, we expect it

would reduce overall system runtime when compared to our current approach. We leave this as part of future work to improve the system.

4.7 Evaluation of FastDepth on the Ultra96 SoC

This section offers an evaluation of our accelerator design against our own previous work on FastDepth as well as against comparable works in literature.

4.7.1 Against FastDepth on the Jetson TX2

We first evaluate the performance of our accelerator-friendly FastDepth network on the Ultra96 board against the performance of our original FastDepth DNN on the NVIDIA Jetson TX2. Table 4.8 summarizes the key evaluation metrics used to compare the different platforms.

Platform	**NVIDIA Jetson TX2 GPU**		**NVIDIA Jetson TX2 CPU**		**Ultra96**
	max-N mode	max-Q mode	max-N mode	max-Q mode	
Hardware	NVIDIA Pascal CPU		NVIDIA Denver 2 ARM Cortex-A57		ZU3EG Xilinx Zynq UltraScale+ MPSoc w/ ARM Cortex-A53
Process	16 nm		16 nm		16 nm
Memory	8GB 128-bit LPDDR4		8GB 128-bit LPDDR4		2GB 32-bit LPDDR4
Operating Frequency	1.30 GHz	0.85 GHz	2.0 GHz	1.2 GHz	1.5GHz (CPU) 250 MHz (FPGA)
FastDepth δ_1 Accuracy	77.1%		77.1%		77.0%
FastDepth Runtime	5.6 ms (178 fps)	8.2 ms (120 fps)	37 ms (27 fps)	64 ms (15 fps)	29 ms on PL (34.5 fps)
Power (Idle)	3.4 W	1.9 W	3.4 W	1.9 W	4.7 W
Power (Busy)	12.2 W	6.5 W	10.5 W	3.8 W	6.1 W
Efficiency	14.5 fps/W	18.5 fps/W	2.6 fps/W	3.9 fps/W	5.7 fps/W

Table 4.8: Evaluation of our accelerator against FastDepth on Jetson TX2. When compared to the TX2 CPU, our accelerator achieves about 1.5−2× improvement in energy-efficiency. FastDepth on the TX2 GPU, however, still achieves a higher energy-efficiency due to its much higher supported framerate.

Of these platforms, the Jetson TX2 GPU achieves significantly higher framerates; despite the relatively high power consumption, the GPU still achieves the highest

frames per Watt efficiency. The TX2 CPU, however, achieves significantly lower framerates, and our FastDepth accelerator on the Ultra96 is more competitive against it. Although the CPU power consumption drops to 3.8 W in the energy-efficient max-Q mode, the framerate also drops to 15 fps — below real-time inference. In comparison, the FastDepth accelerator can achieve over double that framerate at just around 6 W. This results in our accelerator design achieving a higher efficiency than the TX2 CPU in either mode configuration.

4.7.2 Against Other Workloads on the Ultra96

There have been many implementations of depth estimation algorithms on FPGAs; however, they have been based on more traditional computer vision approaches, e.g. stereo matching [127, 128]. Since our FastDepth work falls into the deep learning domain, we narrow down our evaluation to learning-based approaches and consider workloads for tasks such as image classification and object detection. Furthermore, we compare against works that have been deployed onto the same hardware platform as our work, i.e., on the Ultra96 SoC system. Table 4.9 summarizes this evaluation.

Work	**Task**	**# of Params**	**Precision (A/W)**	**Accuracy**	**Frequency (MHz)**	**Batch Size**	**Framerate (fps)**	**Power (W)**	**Efficiency (fps/W)**
Synetgy [129]	IC	3.3M	4/4b	68.3% (top-1)	250	16	96.5	5.5	19.3
FINN-R DoReFa-Net/PF [130]	IC	60.2M	2/1b	50.3% (top-1)	220	—	—	10.2	—
MobileNet [131]	IC	—	4/3b	68.1% (top-1)	215	—	>18-27	6.9	>2.6-3.9
SkyNet [132]	OD	0.44M	9/11b	71.6% (IoU)	—	4	25.05	7.26	3.5
FINN-R Tincy YOLO [130]	OD	6.4M	3/1b	50.1% (mAP)	220	—	—	9.7	—
MobileNet-SSD [131]	OD	—	4/3b	66.4% (mAP)	215	—	18-27	6.9	2.6-3.9
Ours (FastDepth)	DE	1.71M	8/10b	77.0% (δ_1)	250	1	34.5 on PL	6.1	5.7
Ours (MobileNet only)	—	1.29M	8/10b	—	250	1	48.5 on PL	6.1	7.9

Table 4.9: Evaluation against related accelerators on the Ultra96 SoC. Definition of task abbreviations: IC = image classification, OD = object detection, DE = depth estimation. Our accelerator design consumes less power than many of the cited works, while performing inference at higher precision on a similar or more complex task.

Evaluation Against Image Classification Accelerators

The first three works shown in this table deploy image classification networks onto the Ultra96. In Synetgy [129], Yang et al. present a compact convolution neural network,

DiracDeltaNet, that uses 1×1 convolutions and shift operations to replace spatial convolutions. They also co-design an accelerator that supports high batch sizes and performs ImageNet classification inference at 96.5 fps on the Ultra96. Their usage of smaller convolutions and shift operators results in simpler logic and lower resource utilization, which ultimately factors into a relatively low power consumption.

The FINN-R framework [130] offers end-to-end design space exploration and automated creation of inference architectures on FPGAs. FINN-R supports arbitrary precision in weights and activations as well as two options for architecture design: a dataflow architecture that is customize-able for a specific neural network topology to avoid "one-size-fits-all" inefficiencies, as well as a multilayer offload architecture that helps alleviate fragmentation overhead or handle cases where an unrolled dataflow architecture exceeds device resources constraints. This flexibility, along with layer cost models and a quantization-aware layer transformation, allows FINN-R to further adapt networks and generate executable hardware designs for them. As part of their evaluation, the authors apply FINN-R to various reduced-precision neural networks. Their modified DoReFa-Net model and architecture are very low-precision but reportedly have high power consumption.

In the third listed work, Li et al. [131] present a system for low-power object detection on the Ultra96. They use a customized MobileNet-SSD network, which itself uses MobileNet as a backbone, leading to similar design challenges that we faced in our FastDepth accelerator design. Indeed, their accelerator shares a key design aspect with ours: a hybrid dataflow for depthwise separable convolutions. However, their dataflow choice depends on the position of the layer in the network, not whether the layer has a depthwise or pointwise convolution. They opt to use an output-stationary dataflow for earlier layers in MobileNet but argue against using it for all layers since deeper layers contain more parameters and require larger weight buffers. Hence, they switch entirely to weight-stationary for later layers. Furthermore, they use dedicated PEs for 3×3 and 1×1 convolutions. This contrasts with our use of a row-stationary and output-stationary dataflows with reconfigurable PEs, where we toggle between depthwise- and pointwise-dedicated dataflows within *every* layer.

As part of their evaluation, Li et al. isolate MobileNet from within their customized network to compare against other image classification accelerators. We report these statistics as they are more directly comparable to the MobileNet encoder we use in our own FastDepth work. Our MobileNet encoder can support similar if not higher framerates and runs at slightly less power than the cited work. Furthermore, our accelerator supports a higher precision for weights and activations, which may lead to higher accuracy if the encoder were used to run classification tasks.

Evaluation Against Object Detection Accelerators

Image classification tends to be a simpler task than prediction tasks requiring not only the encoding of features within an image but also decoding to produce pixel-specific results, e.g. bounding boxes in object detection or dense depth maps in depth estimation. Since FastDepth is a depth estimation DNN, it presents a more complex workload to map onto an accelerator than many of the image classification workloads that have been explored in recent years. To present an evaluation against more comparable workloads, we include several works on object detection accelerators. One such work is SkyNet [132], where Zhang et al. present a hardware-accelerator co-design approach to object detection. Their SkyNet architecture consists of bundles combining 3×3 depthwise convolutions with 1×1 pointwise convolutions. Stacks of these bundles form the backbone of the network, which is then augmented with bypasses (resembling skip connections) and an adapted YOLO detector head for bounding box regression. Their accelerator exploits data reuse through tiling and batching feature maps and implements a 5-stage pipeline for bundle processing. On the Ultra96, their accelerator achieves inference speeds of 25 fps while consuming 7.26 W.

Table 4.9 also reports other object detection networks deployed on the Ultra96, including a Tincy YOLO accelerated using the FINN-R framework [130] described earlier, as well as the customized MobileNet-SSD by Li et al. [131]. The latter, whose MobileNet backbone we previously compared against, achieves $18-27$ fps on the object detection task, consuming around 6.9 W of power. In comparison, our FastDepth accelerator consumes slightly less power and achieves a slightly higher framerate on

its depth estimation task, while performing inference at a higher precision.

4.8 Summary

In this part of our work on FastDepth, we develop a dataflow design and an accelerator architecture to deploy the FastDepth DNN onboard an embedded CPU-FPGA SoC. We apply an algorithm-hardware co-design approach that re-evaluates the original FastDepth DNN topology and modifies it to be more accelerator-friendly. Our contributions are listed below, alongside the sections in which they are described:

- **Heterogeneous Dataflow Design.** We develop a heterogeneous dataflow design to accelerate processing of depthwise separable layers. We use a row-stationary dataflow for spatial accumulation in depthwise convolutions, and an output-stationary dataflow for channel-wise accumulation in pointwise convolutions. Our heterogeneous dataflow avoids writing *any* intermediate depthwise feature maps out to external memory. [Section 4.2.1]

- **Modular and Scalable PE Array with Reconfigurable PEs.** We design a PE array and on-chip memory hierarchy to exploit row-wise and channel-wise parallelism in depthwise and pointwise convolutions. Our PE array is built up of functionally-independent PE blocks that allows the PE array to scale up or down as necessary. PEs are made reconfigurable to toggle between depthwise and pointwise operations while reusing as much control logic as possible. The PE array achieves high spatial utilization for both depthwise and pointwise convolutions. [Sections 4.3.1, 4.3.2, 4.3.3]

- **Decoder Modifications for Improved Mapping.** In our re-evaluation of the original FastDepth DNN, we decompose upsampling and convolution operations in the decoder to reduce computation and ensure that all decoding layers can easily map onto our proposed accelerator. This decomposition also leads to a reduction in memory accesses for feature maps. [Section 4.4.1]

Chapter 5

Conclusion

In this thesis, we explore fast and energy-efficient monocular depth estimation on embedded platforms. Our work is motivated by recent trends in research on learning-based depth estimation, where emphasis is placed on improving accuracy at the cost of increased model size and computational complexity. The practicality of learning-based depth estimation methods in the real world depends on how well these methods can be deployed on mobile or embedded platforms, e.g., performing depth estimation as part of an autonomous navigation system onboard a micro aerial vehicle. Many state-of-the-art depth estimation methods are deep neural networks that require high computation power and are just too complex to run in real-time on embedded systems. Throughout this thesis, we investigate techniques to simplify these methods for real-time low-power inference.

Compact Depth Estimation DNN In Chapter 2, we present FastDepth [111], our encoder-decoder DNN architecture for monocular depth estimation. We use a MobileNet encoder and a lightweight decoder consisting of alternating convolutional and interpolation layers; we additionally incorporate additive skip connections between encoding and decoding layers to produce sharper depth maps. Our DNN heavily makes use of *depthwise separable convolutions in both the encoder and decoder*, resulting in a compact DNN that has $30-80$ less MACs than prior works yielding similar accuracy rates. In addition, our DNN is balanced in that *neither the encoder or decoder*

dominate computation or runtime; this is an improvement over prior state-of-the-art work [2] with complex decoder structures dominating runtime.

DNN Compilation and Simplification In Chapter 3, we describe the steps we take in achieving real-time inference on an embedded CPU/CPU — namely, the NVIDIA Jetson TX2. We find that the depthwise separable layers prevalent throughout FastDepth incur runtime penalties despite a reduction in MACs due to unoptimized layer implementations being used. To resolve this, we perform *hardware-specific compilation* [98] that tunes every FastDepth layer for faster runtime on our selected hardware; this tuning process speeds up processing of depthwise separable layers by almost 2× on the TX2 GPU and by over two orders of magnitude on the TX2 CPU. We additionally apply *channel pruning* [4] to all layers to remove filter channels that have least impact on model accuracy, thereby simplifying FastDepth even further; this step enables an additional $1.5-1.8\times$ speedup in runtime. After compilation and pruning, our model performs depth inference at 178 fps on the TX2 GPU and at 27 fps on just the TX2 CPU, with total power consumption ranging $10-12$ W and active power consumption under 10 W. FastDepth achieves accuracy rates on par with prior work, while running over an order of magnitude faster.

Custom Dataflow and Accelerator Design In Chapter 4, we explain our approach for accelerating FastDepth on an FPGA. We are motivated by the design flexibility enabled through custom hardware design, which allows us to explore trade-offs in area, speed, and power consumption to develop an energy-efficient accelerator dedicated for our target task. We employ an algorithm-hardware co-design strategy, in which we design our accelerator in conjunction with modifying the workload itself (the FastDepth DNN) to make it more accelerator-friendly. This approach results in a 21% reduction in data movement of feature maps and parameters and enables high spatial utilization of the accelerator hardware. Our accelerator natively runs depthwise separable layers using a reconfigurable engine that toggles between a row-stationary dataflow for depthwise convolutions and an output-stationary dataflow for

pointwise convolutions. The accelerator exploits row-wise and channel-wise parallelism in its compute engine and relies on banked on-chip buffers for high throughput in the network-on-chip. We deploy our accelerator onto the Ultra96 SoC, where it runs FastDepth layers in 29 ms with a total system power consumption of 6.1 W and an estimated active power consumption of under 2 W. When compared to our deployment of FastDepth on the Jetson TX2 CPU, our custom-designed accelerator achieves $1.5-2\times$ improvement in energy efficiency. FastDepth on the TX2 GPU, however, still boasts a higher energy-efficiency due to its higher supported framerate.

With these contributions, we show different ways in which learning-based depth estimation methods can be designed and deployed for fast and energy-efficient inference. All of the platforms on which FastDepth is evaluated in this thesis have small form factors and can be easily carried by robotic vehicles out in the field. Our work demonstrates that depth estimation DNNs can run with reasonably high accuracy in real-time while consuming on the order of several watts, further paving the way for learning-based depth processing live on the edge.

5.1 Key Takeaways

In looking at all of our work on FastDepth, we summarize the following takeaways:

In depth estimation DNN design, complexity is not necessarily better, while simplicity is not necessarily worse. This concerns the tradeoff between DNN complexity and accuracy. A common trend in computer vision research is to explore more complex DNNs with potential pre- and post-processing steps in order to improve accuracy. However, an accuracy boost may not be not very meaningful if the algorithm cannot be deployed in a practical setting due to it surpassing compute limitations. In our development of the FastDepth DNN we adopt an encoder-decoder DNN structure similar to those used in state-of-the-art networks. However, we focus on selecting efficient low-latency building blocks for each. Instead of seeking to

increase depth inference accuracy, we aim to maintain accuracy while significantly simplifying the DNN. Our FastDepth model succeeds in achieving accuracy rates similar to prior works but at a fraction of the size and running over an order of magnitude faster. This shows that simplicity in depth estimation DNNs does not necessarily translate to degraded accuracy.[1]

Hardware-in-loop optimizations are critical in meeting performance specifications, such as latency, energy consumption, or utilization. How efficiently an algorithm runs on a given hardware platform largely depends on how well the algorithm maps to that hardware's primitives (e.g., binary representation, instructions, logic units, registers, memory). Compilers are responsible for optimizing programs and algorithms to better map and utilize hardware primitives. This extends to DNNs as well, as our steps in deploying FastDepth onto an embedded CPU/GPU confirm. Yet hardware-specific compilation alone may not be enough to meet performance specifications, and hardware-in-loop design optimizations become necessary as well. For instance, when we applied network pruning to FastDepth, we used an indirect metric (MACs) to guide the pruning process. Though this was sufficient for our model to achieve our target inference speeds, we could have achieved even better efficiency had we incorporated live runtime measurements into the process. Another example of hardware-in-loop design optimization happens as part of our algorithm-hardware co-design strategy when deploying FastDepth on our custom accelerator. Here, our performance specification is the utilization of processing elements within the accelerator. We notice that several FastDepth layers do not map well onto our PE array design, motivating us to modify the FastDepth DNN — in doing so, we optimize it for high utilization on our accelerator.

[1]This concept has been well explored in the realm of image classification DNNs, with research in pruning, quantization, and compression all showing that it is possible to preserve accuracy while significantly reducing model size and computation. However, at the time of our work on FastDepth, this concept had been less well explored for depth estimation DNNs performing an arguably more difficult dense regression task.

There are often tradeoffs between hardware reconfigurability and performance, and similar tradeoffs may be addressed in different ways at different design stages. We observe two such tradeoffs when designing our FastDepth accelerator. One tradeoff concerns the reconfigurability of a processing element to support different dataflows for different convolution types. Non-reconfigurable PEs allow for a simpler and smaller compute core but complicates load balancing between PEs dedicated to a single dataflow/convolution. Reconfigurable PEs incur complexity (logic) overhead but offer greater flexibility in mapping layers and result in a faster compute core overall. Here, our analysis argues in favor of reconfigurability. At a later point in the design stage, we encounter a different tradeoff — whether to keep our compute core dedicated to 3×3 convolutions or to extend support to 5×5 convolutions found in the second half of our FastDepth DNN. In this case, reconfigurability would again incur logic overhead, and we explore whether we can avoid that overhead by taking advantage of our hardware-algorithm co-design strategy. We are able to modify the FastDepth DNN for it to map better onto a unified accelerator compute core, with negligible impact on DNN accuracy. We decide that the hardware cost of reconfigurability is not worth it if we can address the underlying issue at an algorithmic level. Therefore, in a field where the flexibility to support a variety of workloads (e.g., different DNN layers) is often a very desirable hardware design goal, it is important to consider how similar tradeoffs in a single design can be addressed in different ways across different design stages.[2]

5.2 Future Work

Our work on FastDepth faces several limitations and challenges that point to ways in this work could be extended or improved:

[2]This is particularly relevant for an FPGA design, where the FPGA can be configured from DNN to DNN (workload to workload), but some cases may call for reconfigurability within the DNN model (workload).

Extending FastDepth to work in different environments One limitation of our FastDepth DNN is its interoperability across environments. Since it was trained on a single dataset (NYU Depth v2), it performs well (i.e., accurately) on indoor scenes with depth ranging in several meters. However, it will not maintain accuracy on outdoor scenes with larger depth ranges. As discussed in Section 1.1.2, this limitation pertains to many depth estimation DNNs, not just FastDepth in particular. However, it motivates investigating whether techniques being explored to improve cross-environment depth accuracy could be used on FastDepth.

Improving temporal consistency of FastDepth Another limitation of our FastDepth DNN is its robustness across consecutive images, e.g., frames in a video feed. When performing inference on live video, we observe flickering in the output depth maps; this indicates that predicated pixel-wise depth values in similar regions across frames may not always have similar depth estimates. This is also discussed in Section 1.1.2 as a general limitation of depth estimation DNNs. However, directly incorporating memory-like or feedback-like elements into the FastDepth DNN design to enforce temporal consistency would increase model complexity and incur runtime costs. Investigating this tradeoff more carefully and exploring alternate solutions could be worthwhile next steps in improving FastDepth.

Hiding more memory latency in the accelerator In designing our FPGA-based accelerator for FastDepth, we put emphasis on compute parallelism, as is evident from our structuring of PEs into PE blocks as well as banking all on-chip memory to provide sufficient bandwidth to all of these PE blocks. Increasing both row-wise and channel-wise parallelism in our compute engine naturally sped up the processing of depthwise and pointwise convolutions but also exacerbated the bottleneck arising from data movement between on-chip and off-chip memory. This is discussed in Section 4.6.5 on hiding memory access latency. Thus, one way in which the FastDepth accelerator could be sped up is by reducing the effect of this memory access latency — perhaps through a more flexible clock tree or through a more adaptive prefetching strategy.

Improving on-chip memory utilization in the accelerator A related aspect is on-chip memory utilization. Since we designed an accelerator targeting deployment on an FPGA, we assumed coarse-grained memory in the form of block RAM. As this memory is already part of the FPGA fabric and comes in fixed block sizes, there is less flexibility in defining how exactly on-chip memory in our accelerator is organized. This leads to under-utilized on-chip memory, especially given that layers have different amounts parameter and feature map data. It could help to improve utilization and perhaps reduce total on-chip memory within the accelerator by redesigning it with more fine-grained control, as would be possible in an ASIC design flow.

Bibliography

[1] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel Emer. Efficient Processing of Deep Neural Networks: A Tutorial and Survey, 2017. arXiv:1703.09039.

[2] Iro Laina, Christian Rupprecht, Vasileios Belagiannis, Federico Tombari, and Nassir Navab. Deeper depth prediction with fully convolutional residual networks. In *International Conference on 3D Vision (3DV)*, pages 239–248, 2016.

[3] Convenient Power Measurements on the Jetson TX2 Board. `https://embeddeddl.wordpress.com/2018/04/25/convenient-power-measurements-on-the-jetson-tx2-tegra-x2-board/`, Apr 2018.

[4] Tien-Ju Yang, Andrew Howard, Bo Chen, Xiao Zhang, Alec Go, Mark Sandler, Vivienne Sze, and Hartwig Adam. NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications. In *The European Conference on Computer Vision (ECCV)*, September 2018.

[5] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning, March 2016. arXiv:1603.07285.

[6] NVPModel. `https://www.jetsonhacks.com/2017/03/25/nvpmodel-nvidia-jetson-tx2-development-kit/`, Dec 2018.

[7] Christopher Poulton, Ami Yaacobi, David Cole, Matthew Byrd, Manan Raval, Diedrik Vermeulen, and M.R. Watts. Coherent solid-state LIDAR with silicon photonic optical phased arrays. *Optics Letters*, 42:4091, 10 2017.

[8] Ashutosh Saxena, Sung H Chung, and Andrew Y Ng. Learning depth from single monocular images. In *Advances in Neural Information Processing Systems (NIPS)*, pages 1161–1168, 2006.

[9] Kevin Karsch, Ce Liu, and Sing Bing Kang. Depth extraction from video using non-parametric sampling. In *European Conference on Computer Vision (ECCV)*, pages 775–788, 2012.

[10] Janusz Konrad, Meng Wang, and Prakash Ishwar. 2d-to-3d image conversion by learning depth from examples. In *Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 16–22, 2012.

[11] Kevin Karsch, Ce Liu, and Sing Bing Kang. Depth Transfer: Depth Extraction from Video Using Non-Parametric Sampling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36:2144–2158, 2014.

[12] Miaomiao Liu, Mathieu Salzmann, and Xuming He. Discrete-continuous depth estimation from a single image. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 716–723, 2014.

[13] David Eigen, Christian Puhrsch, and Rob Fergus. Depth map prediction from a single image using a multi-scale deep network. In *Advances in Neural Information Processing Systems (NIPS)*, pages 2366–2374, 2014.

[14] David Eigen and Rob Fergus. Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture. In *International Conference on Computer Vision (ICCV)*, pages 2650–2658, 2015.

[15] Fayao Liu, Chunhua Shen, and Guosheng Lin. Deep convolutional neural fields for depth estimation from a single image. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5162–5170, 2015.

[16] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[17] Xiaojuan Qi, Renjie Liao, Zhengzhe Liu, Raquel Urtasun, and Jiaya Jia. GeoNet: Geometric Neural Network for Joint Depth and Surface Normal Estimation. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 283–291, 2018.

[18] Yevhen Kuznietsov, Jörg Stückler, and Bastian Leibe. Semi-supervised deep learning for monocular depth map prediction. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6647–6655, 2017.

[19] Tinghui Zhou, Matthew Brown, Noah Snavely, and David G Lowe. Unsupervised learning of depth and ego-motion from video. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[20] Ravi Garg, Gustavo Carneiro, and Ian Reid. Unsupervised CNN for single view depth estimation: Geometry to the rescue. In *European Conference on Computer Vision (ECCV)*, pages 740–756, 2016.

[21] Clément Godard, Oisin Mac Aodha, and Gabriel J Brostow. Unsupervised monocular depth estimation with left-right consistency. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

[22] Michele Mancini, Gabriele Costante, Paolo Valigi, and Thomas A Ciarfuglia. Fast robust monocular depth estimation for obstacle detection with fully convolutional networks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4296–4303, 2016.

[23] Fangchang Ma and Sertac Karaman. Sparse-to-dense: depth prediction from sparse depth samples and a single image. *IEEE International Conference on Robotics and Automation (ICRA)*, 2018.

[24] Fangchang Ma, Guilherme Venturelli Cavalheiro, and Sertac Karaman. Self-supervised Sparse-to-Dense: Self-supervised Depth Completion from LiDAR

and Monocular Camera. *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[25] Nathan Silberman, Derek Hoiem, Pushmeet Kohli, and Rob Fergus. Indoor segmentation and support inference from RGBD images. In *European Conference on Computer Vision (ECCV)*, pages 746–760, 2012.

[26] Zhengyou Zhang. Microsoft Kinect Sensor and Its Effect. *IEEE MultiMedia*, 19:4–12, April 2012.

[27] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets Robotics: The KITTI Dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[28] HDL-64E Velodyne Lidar. `https://velodynelidar.com/products/hdl-64e/`.

[29] Faisal Khan, Saqib Salahuddin, and Hossein Javidnia. Deep Learning-Based Monocular Depth Estimation Methods - A State-of-the-Art Review. *Sensors*, 20:2272, April 2020.

[30] Matteo Poggi, Filippo Aleotti, Fabio Tosi, and Stefano Mattoccia. Towards real-time unsupervised monocular depth estimation on CPU. In *IEEE/JRS Conference on Intelligent Robots and Systems (IROS)*, 2018.

[31] Y. Wang, Z. Lai, G. Huang, B. H. Wang, L. van der Maaten, M. Campbell, and K. Q. Weinberger. Anytime Stereo Image Depth Estimation on Mobile Devices. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 5893–5900, 2019.

[32] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[33] Linda Wang, Mahmoud Famouri, and Alexander Wong. Depthnet nano: A highly compact self-normalizing neural network for monocular depth estimation, 2020. arXiv:2004.08008.

[34] Ibraheem Alhashim and Peter Wonka. High Quality Monocular Depth Estimation via Transfer Learning, 2018. arXiv:1812.11941.

[35] Michele Mancini, Gabriele Costante, Paolo Valigi, Thomas Ciarfuglia, Jeffrey Delmerico, and Davide Scaramuzza. Towards Domain Independence for Learning-Based Monocular Depth Estimation. *IEEE Robotics and Automation Letters*, PP:1–1, 01 2017.

[36] René Ranftl, Katrin Lasinger, David Hafner, Konrad Schindler, and Vladlen Koltun. Towards Robust Monocular Depth Estimation: Mixing Datasets for Zero-shot Cross-dataset Transfer, 2019. arXiv:1907.01341.

[37] Tananaev, Denis and Zhou, Huizhong and Ummenhofer, Benjamin and Brox, Thomas. Temporally Consistent Depth Estimation in Videos with Recurrent Architectures. In *The European Conference on Computer Vision (ECCV) Workshops*, September 2018.

[38] Haokui Zhang, Chunhua Shen, Ying Li, Yuanzhouhan Cao, Yu Liu, and Youliang Yan. Exploiting temporal consistency for real-time video depth estimation. In *International Conference on Computer Vision (ICCV'19)*, 2019.

[39] Keisuke Tateno, Federico Tombari, Iro Laina, and Nassir Navab. CNN-SLAM: Real-Time Dense Monocular SLAM With Learned Depth Prediction. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[40] Ashutosh Saxena, Sung H. Chung, and Andrew Y. Ng. 3-D Depth Reconstruction from a Single Still Image. *International Journal of Computer Vision*, 76:53–69, 2007.

[41] Xingtong Liu, Ayushi Sinha, Masaru Ishii, Gregory D. Hager, Austin Reiter, Russell H. Taylor, and Mathias Unberath. Dense Depth Estimation in Monocular Endoscopy with Self-supervised Learning Methods, 2019. arXiv:1902.07766.

[42] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[43] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. Software available from tensorflow.org.

[44] F. Chollet. Xception: Deep Learning with Depthwise Separable Convolutions. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1800–1807, 2017.

[45] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets:

Efficient Convolutional Neural Networks for Mobile Vision Applications, 2017. arXiv:1704.04861.

[46] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, 2015.

[47] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2818–2826, 2016.

[48] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, 2016. arXiv:1602.07360.

[49] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, page 1097âĂŞ1105, 2012.

[50] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated Residual Transformations for Deep Neural Networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5987–5995, 2017.

[51] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks, 2018. arXiv:1801.04381.

[52] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2815–2823, 2019.

[53] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. Searching for MobileNetV3, 2019. arXiv:1905.02244.

[54] Yann LeCun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In D. S. Touretzky, editor, *Advances in Neural Information Processing Systems 2*, pages 598–605. Morgan-Kaufmann, 1990.

[55] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'15, pages 1135–1143, 2015.

[56] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. Pruning Convolutional Neural Networks for Resource Efficient Inference, 2016. arXiv:1611.06440.

[57] Suraj Srinivas and R. Venkatesh Babu. Data-free Parameter Pruning for Deep Neural Networks. In *BMVC*, 2015.

[58] Hengyuan Hu, Rui Peng, Yu-Wing Tai, and Chi-Keung Tang. Network Trimming: A Data-Driven Neuron Pruning Approach towards Efficient Deep Architectures, 2016. arXiv:1607.03250.

[59] Ariel Gordon, Elad Eban, Ofir Nachum, Bo Chen, Hao Wu, Tien-Ju Yang, and Edward Choi. MorphNet: Fast & Simple Resource-Constrained Structure Learning of Deep Networks, 2018. arXiv:1711.06798.

[60] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. Designing Energy-Efficient Convolutional Neural Networks Using Energy-Aware Pruning. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6071–6079, 2017.

[61] Sara Elkerdawy, Hong Zhang, and Nilanjan Ray. Lightweight Monocular Depth Estimation Model by Joint End-to-End Filter pruning, 2019. arXiv:1905.05212.

[62] William Dally. High-Performance Hardware for Machine Learning. `https://media.nips.cc/Conferences/2015/tutorialslides/Dally-NIPS-Tutorial-2015.pdf`, Dec 2015.

[63] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. Quantized Convolutional Neural Networks for Mobile Devices. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[64] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[65] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. Fixed Point Quantization of Deep Convolutional Networks. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 2849–2858. JMLR.org, 2016.

[66] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained Ternary Quantization. In *5th International Conference on Learning Representations ICLR*, 2017.

[67] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients, 2016. arXiv:1606.06160.

[68] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[69] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898, 2017.

[70] Yoshua Bengio, Nicholas Léonard, and Aaron Courville. Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation, 2013. arXiv:1308.3432.

[71] Penghang Yin, Jiancheng Lyu, Shuai Zhang, Stanley Osher, Yingyong Qi, and Jack Xin. Understanding Straight-Through Estimator in Training Activation Quantized Neural Nets, 2019. arXiv:1903.05662.

[72] J. Lee, C. Kim, S. Kang, D. Shin, S. Kim, and H. Yoo. UNPU: An Energy-Efficient Deep Neural Network Accelerator With Fully Variable Weight Bit Precision. *IEEE Journal of Solid-State Circuits*, 54(1):173–185, 2019.

[73] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations ICLR*, 2016.

[74] Michael Mathieu, Mikael Henaff, and Yann LeCun. Fast Training of Convolutional Networks through FFTs, 2013. arXiv:1312.5851.

[75] Jason Cong and Bingjun Xiao. Minimizing computation in convolutional neural networks. In *International Conference on Artificial Neural Networks*, pages 281–290. Springer, 2014.

[76] S. Winograd. Fast Algorithms for Convolutional Neural Networks, 1980.

[77] Andrew Lavin and Scott Gray. Fast Algorithms for Convolutional Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[78] OpenBLAS: An optimized BLAS library. `http://www.openblas.net/`.

[79] cuBLAS. `https://docs.nvidia.com/cuda/cublas/index.html`.

[80] NVIDIA cuDNN. `https://developer.nvidia.com/cudnn`.

[81] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. The Deep Learning Compiler: A Comprehensive Survey, 2020. arXiv:2002.03794.

[82] NVIDIA DGX-1. https://www.nvidia.com/en-us/data-center/dgx-1/.

[83] Jetson TX1 Module. https://developer.nvidia.com/embedded/jetson-tx1.

[84] Jetson TX2 Module. https://developer.nvidia.com/embedded/jetson-tx2.

[85] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh K. Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-Datacenter Performance Analysis of a Tensor Processing Unit. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2017.

[86] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. Serving DNNs in Real Time at Datacenter Scale with Project Brainwave. *IEEE Micro*, 38:8–20, March 2018.

[87] Apple's Neural Engine Infuses the iPhone With AI Smarts. https://www.wired.com/story/apples-neural-engine-infuses-the-iphone-with-ai-smarts/.

[88] Accelerate Your On-device AI with the Qualcomm Artificial Intelligence (AI) Engine on Snapdragon. https://developer.qualcomm.com/blog/accelerate-your-device-ai-qualcomm-artificial-intelligence-ai-engine-snapdragon.

[89] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. DianNao: A Small-Footprint High-Throughput Accelerator for Ubiquitous Machine-Learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'14, pages 269–284. Association for Computing Machinery, 2014.

[90] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 609–622, 2014.

[91] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. ShiDianNao: Shifting Vision Processing Closer to the Sensor. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 92–104. Association for Computing Machinery, 2015.

[92] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernandez-Lobato, G. Wei, and D. Brooks. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 267–278, 2016.

[93] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA'16, pages 243–254. IEEE Press, 2016.

[94] Chen, Yu-Hsin and Krishna, Tushar and Emer, Joel and Sze, Vivienne. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. In *IEEE International Solid-State Circuits Conference, ISSCC 2016, Digest of Technical Papers*, pages 262–263, 2016.

[95] P. Judd, J. Albericio, T. Hetherington, T. M. Aamodt, and A. Moshovos. Stripes: Bit-serial deep neural network computing. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12, 2016.

[96] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA'17, page 2740. Association for Computing Machinery, 2017.

[97] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit Fusion: Bit-Level Dynamically Composable Architecture for Accelerating Deep Neural Network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, 2018.

[98] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In

13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 578–594, 2018.

[99] Ziheng Jiang, Tianqi Chen, and Mu Li. Efficient deep learning inference on edge devices. In *SysML'18*, 2018.

[100] XLA: Optimizing Compiler for Machine Learning | TensorFlow. `https://www.tensorflow.org/xla`.

[101] NVIDIA TensorRT. `https://developer.nvidia.com/tensorrt`.

[102] Xilinx Vitis AI: Adaptable and Real-Time AI Inference Acceleration. `https://www.xilinx.com/products/design-tools/vitis/vitis-ai`.

[103] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions, 2018. arXiv:1802.04730.

[104] Scott Cyphers, Arjun K. Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, Robert Kimball, Jason Knight, Nikolay Korovaiko, Varun Kumar, Yixing Lao, Christopher R. Lishka, Jaikrishnan Menon, Jennifer Myers, Sandeep Aswath Narayana, Adam Procter, and Tristan J. Webb. Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning, 2018. arXiv:1801.08058.

[105] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, Jack Montgomery, Bert Maher, Satish Nadathur, Jakob Olesen, Jongsoo Park, Artem Rakhov, Misha Smelyanskiy, and Man Wang. Glow: Graph Lowering Compiler Techniques for Neural Networks, 2018. arXiv:1805.00907.

[106] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA'16, pages 367–379. IEEE Press, 2016.

[107] NVIDIA Deep Learning Accelerator. `http://nvdla.org/`.

[108] Wenyan Lu, Guihai Yan, Jiajun Li, Shijun Gong, Yinhe Han, and Xiaowei Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 553–564. IEEE, 2017.

[109] F. Tu, S. Yin, P. Ouyang, S. Tang, L. Liu, and S. Wei. Deep Convolutional Neural Network Architecture With Reconfigurable Computation Patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017.

[110] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'18, pages 461–475. Association for Computing Machinery, 2018.

[111] Wofk, Diana and Ma, Fangchang and Yang, Tien-Ju and Karaman, Sertac and Sze, Vivienne. FastDepth: Fast Monocular Depth Estimation on Embedded Systems. In *IEEE International Conference on Robotics and Automation (ICRA)*, 2019.

[112] Ke Xian, Chunhua Shen, Zhiguo Cao, Hao Lu, Yang Xiao, Ruibo Li, and Zhenbo Luo. Monocular Relative Depth Perception With Web Stereo Data Supervision. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 311–320, 2018.

[113] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[114] Tien-Ju Yang, Maxwell D. Collins, Yukun Zhu, Jyh-Jing Hwang, Ting Liu, Xiao Zhang, Vivienne Sze, George Papandreou, and Liang-Chieh Chen. DeeperLab: Single-Shot Image Parser. *arXiv preprint arXiv:1902.05093*, 2019.

[115] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[116] Augustus Odena, Vincent Dumoulin, and Chris Olah. Deconvolution and Checkerboard Artifacts. *Distill*, 2016.

[117] UltraScale Memory Resources. `https://www.xilinx.com/support/documentation/user_guides/ug573-ultrascale-memory-resources.pdf`.

[118] Ultra96 Hardware User Guide. `http://zedboard.org/sites/default/files/documentations/Ultra96-HW-User-Guide-rev-1-0-V1_1.pdf`.

[119] AXI DMA v7.1 LogiCORE IP. `https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf`.

[120] PYNQ: Python productivity for Zynq. `http://www.pynq.io/`.

[121] Zynq UltraScale+ Technical Reference Manual. `https://www.xilinx.com/support/documentation/user_guides/ug1085-zynq-ultrascale-trm.pdf`.

[122] A. Yazdanbakhsh, M. Brzozowski, B. Khaleghi, S. Ghodrati, K. Samadi, N. S. Kim, and H. Esmaeilzadeh. FlexiGAN: An End-to-End Solution for FPGA Acceleration of Generative Adversarial Networks. In *2018 IEEE 26th Annual*

International Symposium on Field-Programmable Custom Computing Machines (FCCM), pages 65–72, 2018.

[123] L. Du, Y. Du, Y. Li, J. Su, Y. Kuan, C. Liu, and M. F. Chang. A Reconfigurable Streaming Deep Convolutional Neural Network Accelerator for Internet of Things. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65(1):198–208, 2018.

[124] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper, 2018. arXiv:1806.08342.

[125] Neta Zmora, Guy Jacob, Lev Zlotnik, Bar Elharar, and Gal Novik. Neural Network Distiller: A Python Package For DNN Compression Research, October 2019. arXiv:1910.12232.

[126] Travis Oliphant. *A guide to NumPy*. Trelgol Publishing USA, 2006.

[127] C. Banz, S. Hesselbarth, H. Flatt, H. Blume, and P. Pirsch. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and FPGA-implementation. In *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 93–101, 2010.

[128] D. Honegger, H. Oleynikova, and M. Pollefeys. Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU. In *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4930–4935, 2014.

[129] Yifan Yang, Qijing Huang, Bichen Wu, Tianjun Zhang, Liang Ma, Giulio Gambardella, Michaela Blott, Luciano Lavagno, Kees Vissers, John Wawrzynek, and et al. Synetgy. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb 2019.

[130] Michaela Blott, Thomas B. Preußer, Nicholas J. Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), December 2018.

[131] Fanrong Li, Yang Zhang, Jian Cheng, Zitao Mo, Peisong Wang, Zejian Liu, Jiayun Zhang, Gang Li, Qinghao Hu, Xiangyu He, and Cong Leng. A System-Level Solution for Low-Power Object Detection. In *2019 IEEE/CVF International Conference on Computer Vision Workshops, ICCV Workshops*, pages 2461–2468. IEEE, 2019.

[132] Xiaofan Zhang, Haoming Lu, Cong Hao, Jiachen Li, Bowen Cheng, Yuhong Li, Kyle Rupnow, Jinjun Xiong, Thomas Huang, Honghui Shi, Wen-Mei Hwu, and Deming Chen. SkyNet: a Hardware-Efficient Method for Object Detection and Tracking on Embedded Systems. In *Proceedings of Machine Learning and Systems 2020*, pages 216–229, 2020.