Reconfigurable Convolution Implementation for CNNs in FPGA

Jesse Bannon

A thesis
submited in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2018

Reading Committee:

Matthew Tolentino, Chair

Jie Sheng

Program Authorized to Offer Degree:
Computer Science and Systems

University of Washington

Abstract

Reconfigurable Convolution Implementation
for CNNs in FPGA

Jesse Bannon

Chair of the Supervisory Committee:
Assistant Professor Matthew Tolentino
Center for Data Science

Deep learning continues to be the revolutionary method used in pattern recognition applications including image, video, and speech processing. Convolutional Neural Networks (CNNs) in particular have outperformed every competitor in image classification benchmarks, but suffer from high computation and storage complexities. It is becoming more apparent to extend this breakthrough technology to embedded applications that demand low power and mission critical response times. Consequently, embedded CNNs deployed on the edge require compact platforms capable of accelerated computing. Previous works have explored methods to optimize convolution computation within Field Programmable Gate Arrays (FPGAs). Many of which only consider supporting a single CNN architecture. While this approach allows precise optimizations structured around a specific CNN, it restricts the FPGA from updating its model without tremendous compile times upwards to hours. In this work, we explore state-of-the-art CNN-FPGA architectures and implement our own reconfigurable convolution computation unit (CCU) using the Intel High Level Synthesis (HLS) Compiler using a sliding window-based implementation. Results show our CCU does not suffice for real-time computations on the edge.

TABLE OF CONTENTS

Page

Glossary ii

Chapter 1: Introduction 1

Chapter 2: Background 3
2.1 Field Programmable Gate Arrays 3
2.2 Convolutional Neural Networks 4
2.3 High Level Synthesis 4
2.4 Benchmarking 5

Chapter 3: Related Work 6
3.1 Frameworks 6
3.2 Convolution Computation Units 7
3.3 Memory Access Patterns 9

Chapter 4: System Design 10
4.1 Architecture 10
4.2 Environment and Experimental Results 11

Chapter 5: Conclusion 14

Bibliography 15

GLOSSARY

HDL: Hardware definition language, syntax used to describe a hardware design.

FPGA: Field programmable gate array, an integrated circuit designed for consumers to reconfigure with custom ASIC designs via HDL.

HLS: High level synthesis, an abstraction layer for designing HDL in high level languages such as C.

CNN: Convolutional neural network, a class of feed-forward neural networks that uses convolution to detect spacial features.

CCU: Convolutional computation unit, a component within an FPGA that computes convolution layers of a CNN.

MAC: Multiply-accumulate operation.

BRAM: Limited on-chip random access memory within an FPGA.

DRAM: Off-chip random access memory accessible by an FPGA.

DSP: Digital signal processor, an FPGA resource notably used to compute floating point operations.

LE: Logic element, an FPGA resource that perform logic functions, including logic gates, registers, and multiplexers.

Chapter 1

INTRODUCTION

Convolutional Neural Networks (CNNs) have been adopted as the de-facto method for image processing. However, their high computational complexity and memory usage imposes a burden when operating in embedded environments. Standard embedded platforms lack sufficient GOP/s for practical CNN applications [8]. In recent years, many FPGA implementations of CNNs have been explored to exploit the parallel nature of convolution to obtain satisfactory performance that can support real time inference applications [27]. Most research is focused on convolution computation units (CCUs) as they account for nearly 90% of compute time and energy usage. Kernel sizes, strides, other layer parameters, and FPGA resources play an important role in the design of CCUs. Many CNN-FPGA implementations only consider hard-coded CCUs for specific CNN architectures, allowing granular optimizations to fully maximize performance [17, 27, 11, 2, 12]. Only a few consider reconfigurable designs which allow various CNN compatibility without having to recompile[28, 5, 18].

Recompiling an FPGA has significant implications when using one as an edge device. The place and route stage maps HDL onto the FPGA itself, and routes the connections between components. This stage alone can take multiple hours. When considering a CNN-FPGA application that is performing mission-critical computations, that much downtime for an update can hinder its usage altogether. Thus, we consider reconfigurable designs that allow compatibility for various layers found in a CNN. This allows a CNN-FPGA device to be updated with a new model by only updating its weights and layer configuration stored in external memory of the FPGA. If extended downtime is permissible, a reconfigurable design on an FPGA can update with the latest breakthrough methods, unlike an ASIC device. The penalty of recompilation would only be necessary for modifications to the layer compatibility

itself.

In this work, we implement a reconfigurable CCU using the Intel HLS Compiler and benchmark it using the layer sizes from VGG-16. While our results shows weak performance, it can serve as an evaluation of the Intel HLS Compiler and performance of a sliding-window based CCU.

Chapter 2

BACKGROUND

2.1 Field Programmable Gate Arrays

FPGAs are integrated circuits designed to be reconfigurable. They are made up of programmable logic blocks and interconnects that allow the designer to create programs made up of modular blocks. For our work, we are interested in the amounts of digital signal processors (DSPs), logic elements (LEs), block RAM (BRAM), and dynamic RAM (DRAM) equipped in FPGAs. DSPs are specialized microprocessor designed to perform primitive signal processing operations including multiply-accumulate (MAC). Given enough DSPs, they can be used to accelerate embarrassingly parallel tasks including matrix-multiply, convolution, and Fast Fourier transform (FFT) much faster than a traditional processer. LEs are resources on an FPGA that can perform logic functions, including logic gates, registers, and multiplexers. BRAM is limited on-chip random access memory. Lastly, DRAM is off-chip random access memory capable of storing much more data than on-chip at the cost of access latency. It is imperative to access DRAM such that data bursts. Burst mode refers to sending data repeatedly without performing any additional steps including indexing. DRAM implements burst mode by automatically fetching the next memory contents before they are requested. Thus, if memory is accessed in a contiguous manner, the number if memory access requests are minimized. Quantization is the process of constraining floating points to a discrete set of bits for precision. The radix point refers to the separation of the exponent and mantissa of a floating point. It can be placed anywhere relative to the significant digits of a number. FPGAs are often equipped with DSPs that support variable precisions that scale with respect to precision.

Name	Description
Activation	Applies a monotonic function on all input values
Add	Takes same sized parallel tensors and adds them together by value
AveragePooling2D	Strides along feature maps with k-by-k kernel and outputs average of all elements
BatchNormalization	Normalizes all inputs to be between 0 and 1
Concatenate	Concatenates two tensors along some axis
Conv2D	Performs k convolutions on a 3d tensor depth-wise
Cropping2D	Consider only a rectangular subset of an input feature map, disregard the rest
Dense	Multiplies elements in input vector by n weights
Dropout	Multiplies n random elements by 0
Flatten	Flattens a tensor to a 1 dimensional vector, used to feed to dense layers
MaxPooling2D	Reduces an output feature map by striding across each channel taking the max of k-by-k values

Figure 2.1: Common CNN Layers

2.2 Convolutional Neural Networks

CNNs are a class of feed-forward artificial neural networks that excel in image classification. Notably because their shared, trained weights are able to identify patterns irrespective of their location within input images. Convolutional kernels, which contain these weights, stride along the axes and perform convolution to map inputs into feature maps, where it is easier to identify desired patterns to classify. CNN architectures are comprised of many different types of layers, as shown in 2.2. Our work is focused only on convolution and padding as they account for nearly 90% of computation and energy consumption (NEED REF).

2.3 High Level Synthesis

HLS is a compiler that interprets an algorithm and compiles it into a digital hardware representation. Its purpose is to design hardware at a much faster pace. Similar to high level languages, HLS limits users from accessing low-level constructs, which can potentially hinder performance for algorithms that require meticulous designs. The Intel HLS compiler claims it generates high-quality code that is orders of magnitude faster than register-transfer level (RTL), requires 80% less lines of code, meets performance, and is within 10-15% of the area of hand- coded RTL. This is accomplished using existing logic units (IPs); the intellectual property of a party.

In many cases, requiring the FPGA on-hand is unfeasible for testing code. ModelSim is a tool that allows you to simulate RTL within software. Our work will use both the Intel HLS Compiler for faster development and more performant code compared to writing our own RTL. Additionally, we will use ModelSim to conduct experiments that analyze performance on a variety of FPGA architectures.

2.4 Benchmarking

Giga operations per second (GOP/s) is a standard metric for analyzing accelerator performance. It is simpler than runtime because it does not rely on experiments using the same inputs for a comparison between two accelerators, and is hardware agnostic. Floating point operations per second (FLOP/s) is not considered because it abstracts the complexities of primitive operations including divide, multiply, add, etc. With FPGAs targeting granular acceleration techniques, it is best not to abstract any performance characterizations. GOPs per watt (GOP/W) considers performance with respect to energy usage. It is an important metric to consider for minimizing long term energy costs and efficiency.

Chapter 3

RELATED WORK

3.1 Frameworks

Many mature frameworks exist to implement CNNs on CPU and GPU computing platforms. For FPGA, frameworks are still relatively elementary. Lui et al. [17] implements a Verilog source code generator using a domain-specific language consisting of parameters describing each layer in a CNN configuration. Each layer is capable of being analytically modeled to estimate computation complexity, execution time, and resource usage based on their input parameters. For compute, they use the convolver CCU, which is a pipelined convolutional kernel with data reuse via registers. Guo et al. [8] implements Angel-Eye, a complete design flow for mapping CNNs onto FPGAs. They first collect statistics on the feature maps and network parameters to get a histogram of their logarithm value, which inspires how they can choose the radix point for fixed point floats layer-by-layer. For their CCU, they adopt a 3x3 convolution kernel based on the reconfigurable line buffer design [33] that supports any kernel size. Data is arranged as row-major tiles. They require a single row in a tile to have enough elements to maximize the DRAM burst length. The number of rows in a single tile is chosen by a set of rules that minimizing DRAM memory accesses. DiCecco et al. [5] implements Caffeinated, a modification of Caffe for FPGA support alongside a host processor. The FPGA is used as a dedicated accelerator for 3x3 convolution layers using a Winograd-based CCU. Lu et al. [18] implements another Winograd- based approach with code generation. They too employ a line buffer structure similar to Angel- Eye. They model their resource usages which is used to maximize Winograd CCU parallelism within a given FPGA, and generate respective HLS code. Tu et al. [28] implements Deep Neural Architecture (DNA), an acceleration chip with reconfigurable computation patterns for different models. DNA

reconfigures its data path to maximize data reuse based on the model to minimize energy usage. Data reuse is accomplished by tiling input and output feature maps with different configurations per layer that minimize off-chip memory accesses. While DNA is not designed for FPGAs, many of its hardware optimizations can be considered in FPGA implementations.

3.2 Convolution Computation Units

3.2.1 Convolver

The convolver [17] exploits pipelined parallelism by using a sliding window when scanning a single channel, and is connected to a reduction tree to get a single output element per cycle. Convolvers are fixed to the size of a kernel. It is assumed feature maps are stored row-wise for coalesced data accesses to feed the convolver.

Their experimental results only consider two sequential models, AlexNet [13] and LeNet [15]. Their FPGA is large enough to map each layer onto the FPGA. Within AlexNet are kernel sizes of 11x11, 7x7, and 3x3. Three independent convolvers of each size are needed to support all convolution layers. In the case where FPGA resources are limited, there may not be enough DSPs to account for all CCUs. In addition, each CCU requires sufficient memory for its buffer to avoid stalling. Otherwise, bottlenecks will occur in the pipeline, causing convolvers to not be fully utilized. They do not elaborate on how their framework can be used to mitigate this problem. Lastly, the convolver is not optimal for convolutions with stride greater than one. The computation is not coalesced, and requires overhead to orchestrate either the data access pattern or convolver input, both of which can stall the pipeline.

3.2.2 Output Oriented Mapping

Previous works consider Output Oriented Mapping (OOM) [34, 4, 25], which loads an input map into its CCU and performs convolution using a single weight in all processing elements (PEs). Both OOM and convolver exploit data reusability via sliding window interconnects

between PE registers. The benefit of OOM is it can support various kernel sizes with a fixed number of DSPs without having to allocate k-by-k DSPs for each weight in a kernel. However, OOM too suffers from stride greater than one for the same reasons as convolver.

3.2.3 Parallel OOM

DNA from [28] extends OOM using parallel (POOM), a modification that parallel computes P feature maps on the PE array. In the example above, if stride were equal to two, the PE array's middle column would be unutilized. POOM will use that column to compute the convolution of an additional feature map.

3.2.4 Winograd CCU

Winograd convolution [31] has been used to accelerate convolutions of small kernels in other CNN implementations [5, 14, 18]. It is an algorithm that exploits the multiplicative structure of FFT and transforms it into a cyclic convolution computation [30]. The end result is a convolution algorithm that requires less multiplications and more additions via matrix multiplications (GEMMs). Most hardware platforms, including FPGAs, contain optimizations specifically for GEMM. The Winograd convolution exploits this to accelerate convolution computation. Winograd output is referred to as F(mxm, rxr), where mxm represents the number of output values and rxr is the filter size. [5] implements a F(2x2, 3x3) Winograd CCU with unity stride. An output size of 2x2 with 3x3 kernels requires 4x4 input size. Feature maps are tiled using 4x2 blocks with replicated values in the rows to store overlapping data in successive Winograd convolutions along a column. The tile width of two results in no overlaps in Winograd convolutions along a row. Tiles are fed into the CCU row-wise and compute partial outputs. Partial outputs of the transformed data are computed in parallel along multiple columns of a row, allowing successive reuse for neighboring tiles. Following that, each 4x2 tile and its neighboring tile are fed into a pipelined CU that handles all subsequent computations after the initial transformation.

3.3 Memory Access Patterns

3.3.1 Row/Column Major

The most basic form of storing kernel weights and input/output feature maps is depth x row x column or depth x column x row, such that columns or rows are operated on first. For the convolver and OOM-like CCUs, they can be feed rows of data that reuse adjacent columns within registers. Several memory access penalties can occur if multiples rows cannot be stored in BRAM. If a row is sufficiently large, multiple data accesses to DRAM must be made on successive rows. Additionally, DRAM access can only exploit burst-mode for a single row of data. When accessing a subset of multiple rows, each row within the subset is not contiguous, causing additional memory requests for each row.

3.3.2 Tiling

Many CNN-FPGA implementations tile data to avoid the pitfalls of row or column major formatting [28, 5, 2, 8]. Tiles often contain duplicated data in their overlapped regions such that a single tile is only ever accessed once during layer computation. It is typical to only tile the feature maps and not the channels, such that multiple CCUs can operate on independent channels in parallel. Tile sizes are often chosen with respect to data burst lengths, kernel sizes, number of channels, and BRAM buffer sizes for inputs, outputs, and weights.

Chapter 4

SYSTEM DESIGN

4.1 Architecture

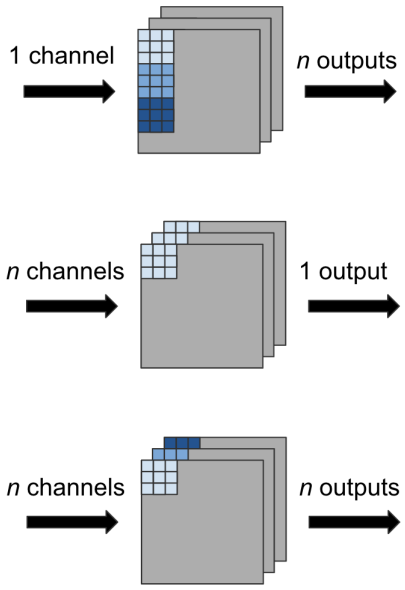

Figure 4.1: Channel-wise convolution oriented to either reuse inputs, outputs, or stay independent.

Our sliding window CCU component computes a single channel's worth of convolution. 4.1 shows how channel-wise parallelism can be oriented to either reuse inputs, outputs, or be entirely independent. In the case of reusing inputs, a mechanism must be in place to insure atomic updates are made to the output layer to prevent concurrent CCUs from overwriting values. In conjunction with one of these orientations, channel computation can be iterated to reuse existing input channels for different kernels altogether, continue on the kernel's next channel to complete the current output layer, or reuse the same kernel on a different input within a mini-batch. Input and kernel sizes could drastically impact memory latencies for each orientation. Our work focuses primarily on the acceleration of a single reconfigurable CCU. Thus, we leave the exploration of CCU parallelism slated for future work.

The logic of our reconfigurable CCU component is depicted in 4.2. Note that subsequent diagrams do not have a one-to-one correspondence with their actual hardware implementation due to the HLS translation. We use a sliding window mechanism similar to [17] with the exception that ours is reconfigurable for any sized input, with a fixed kernel size of three. We choose three because its one of the more popular kernel

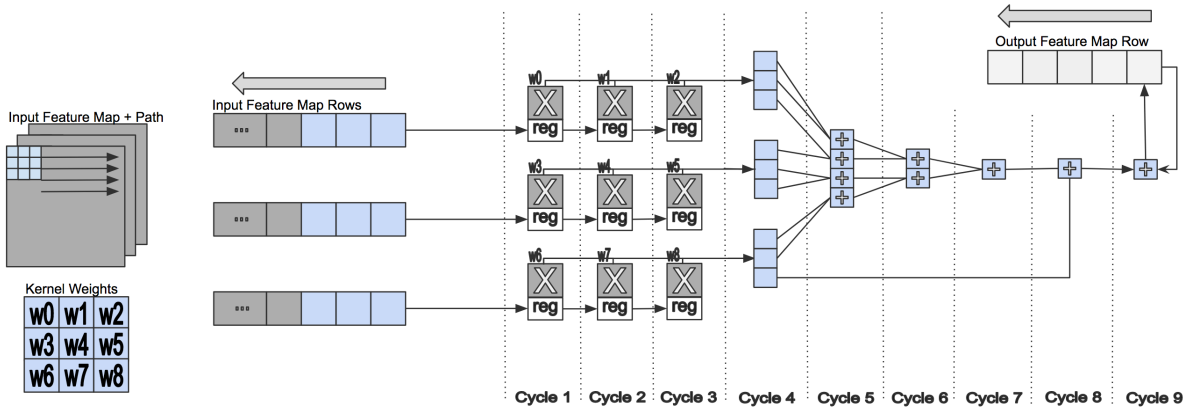

Figure 4.2: Sliding window CCU with a 3x3 kernel. Loads three rows into RAM and performs a sliding window to load values into registers and compute the convolution.

sizes. We can compute convolution for different sized kernels using a technique similar to [8] where they pad outer weights with zero for smaller kernels and perform multiple convolutions using subsets and zero-padding for larger kernels.

We first load three rows from external RAM into the input data buffer, implemented in on-chip RAM. Each row in the data buffer can hold 32 elements. This number was chosen based upon making the resource usage of BRAM and DSPs relatively equal so when placing multiple CCUs onto the FPGA, we can fully utilize all resources. The sliding 3x3 window then strides along the rows, loading the current values within the window into registers to multiply and accumulate them with the kernel weights into the output buffer, which is also of size 32. Once the three rows are complete, we transfer the output buffer data to external RAM, and repeat the process until all output rows are computed.

4.2 Environment and Experimental Results

We implement our reconfigurable CCU using the Intel HLS Compiler, and compile for the Cyclone V FPGA. Benchmarks and correctness tests are obtained and verified using the ModelSim FPGA simulator. Each benchmark is ran using artificial data with input dimensionality, padding, and kernel sizes set to respective layer sizes from the VGG-16 architecture.

3x3 Convolution, Single Channel, Padding=1, Stride=1, 100 MHz			
Size	Cycles	OPs	GOP/s
7x7	1433	882	0.062
14x14	3938	3528	0.09
28x28	10174	14112	0.139
56x56	42568	56448	0.133
112x112	172815	225792	0.131
224x224	812774	903168	0.111

Figure 4.3: Benchmarks for a single sliding window CCU

Floating point quantization is set to 16 bit precision throughout all benchmarks. For OPs, we consider only MACs and take into account the padding. We compute GOP/s by fixing the clock speed to 100 MHz.

Results shown in 4.2 show our CCU's performance scales near-consistent for input sizes greater than 28. Smaller input sizes pay the penalty of refilling the input data buffer more frequently, shortening the phase of pipelined convolution. In addition, larger input sizes suffer in performance by reloading large rows into the input data buffer multiple times.

Comparison to related works in 4.2 shows our results are significantly slower. We exclude GOPs/W since we only consider benchmarks from simulation. We compute our total GOP/s by taking the average of all GOP/s in 4.2 and multiplying by six - the total amount of sliding window CCUs that can fit onto a Cyclone V FPGA based on the amounts in 4.2. Additionally, we show GOPs/DSP to better understand the performance of our CCU regardless of FPGA resources used in related works' benchmarks. Based on this, results conclude that our approach is more than 25x slower than the state-of-the-art FPGA-based CCU.

Resource	Amount p/CCU	Total
ALUTs	17.5k	110k
FFs	27.3k	219k
BRAM (18kb)	81	514
DSPs	9.5	112

Figure 4.4: Resource usage of a sliding window CCU

Ref	Design	FPGA	DSPs	GOP/s	GOPs/W	GOPs/DSP
Ours*	Sliding Window	Cyclone V	112	0.67	-	0.006
[17]	Convolver	Virtex7 VX690T	3600	222.1	8.96	0.061
[8]	Angel Eye	Zynq XC7Z045	900	137	14.2	0.152
[5]*	Winograd	Virtex7 VX690T	50	-	0.014	0.006
[28]	POOM	ASIC	-	194.4	109.2	-

Figure 4.5: Benchmark Comparisons to Related Works. (*) denotes CCUs that only compute convolution

Chapter 5

CONCLUSION

In this work, we evaluated multiple CNN-FPGA architectures, their CCUs, and implemented our own reconfigurable CCU using the Intel HLS compiler. We benchmarked our design in simulation using the same layer sizes as the VGG-16 architecture. Results show our implementation is roughly 25x slower than the state-of-the-art CCUs.

BIBLIOGRAPHY

[1] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C Ling, and Gordon R Chiu. An opencl™ deep learning accelerator on arria 10. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 55–64. ACM, 2017.

[2] Marco Bettoni, Gianvito Urgese, Yuki Kobayashi, Enrico Macii, and Andrea Acquaviva. A convolutional neural network fully implemented on fpga for embedded platforms. In *CAS (NGCAS), 2017 New Generation of*, pages 49–52. IEEE, 2017.

[3] Bernard Bosi, Guy Bois, and Yvon Savaria. Reconfigurable pipelined 2-d convolvers for fast digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(3):299–308, 1999.

[4] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 52(1):127–138, 2017.

[5] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Graham Taylor, and Shawki Areibi. Caffeinated fpgas: Fpga framework for convolutional neural networks. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 265–268. IEEE, 2016.

[6] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 152–159. IEEE, 2017.

[7] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 629–634. IEEE, 2017.

[8] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2018.

[9] Kaiyuan Guo, Shulin Zeng, Jincheng Yu, Yu Wang, and Huazhong Yang. A survey of fpga based neural network accelerator. *arXiv preprint arXiv:1712.08934*, 2017.

[10] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–84. ACM, 2017.

[11] Wen-Jyi Hwang, Yun-Jie Jhang, and Tsung-Ming Tai. An efficient fpga-based architecture for convolutional neural networks. In *Telecommunications and Signal Processing (TSP), 2017 40th International Conference on*, pages 582–588. IEEE, 2017.

[12] Li Jiao, Cheng Luo, Wei Cao, Xuegong Zhou, and Lingli Wang. Accelerating low bit-width convolutional neural networks with embedded fpga. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[13] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[14] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[15] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[16] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–9. IEEE, 2016.

[17] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic code generation of convolutional neural networks in fpga implementation. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 61–68. IEEE, 2016.

[18] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. Evaluating fast algorithms for convolutional neural networks on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*, pages 101–108. IEEE, 2017.

[19] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 45–54. ACM, 2017.

[20] Duncan JM Moss, Eriko Nurvitadhi, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. High performance binary neural networks on the xeon+ fpga^{TM} platform. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[21] Mohammad Motamedi, Philipp Gysel, Venkatesh Akella, and Soheil Ghiasi. Design space exploration of fpga-based deep convolutional neural networks. In *ASP-DAC*, pages 575–580, 2016.

[22] Hiroki Nakahara, Tomoya Fujii, and Shimpei Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*, pages 1–4. IEEE, 2017.

[23] Abhinav Podili, Chi Zhang, and Viktor Prasanna. Fast and efficient implementation of convolutional neural networks on fpga. In *Application-specific Systems, Architectures and Processors (ASAP), 2017 IEEE 28th International Conference on*, pages 11–18. IEEE, 2017.

[24] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM, 2016.

[25] Atul Rahman, Jongeun Lee, and Kiyoung Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pages 1393–1398. IEEE, 2016.

[26] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 16–25. ACM, 2016.

[27] F Javier Toledo-Moreo, J Javier Martínez-Alvarez, Javier Garrigos-Guerrero, and J Manuel Ferrandez-Vicente. Fpga-based architecture for the real-time computation of 2-d convolution with large kernel size. *Journal of Systems Architecture*, 58(8):277–285, 2012.

[28] Fengbin Tu, Shouyi Yin, Peng Ouyang, Shibin Tang, Leibo Liu, and Shaojun Wei. Deep convolutional neural network architecture with reconfigurable computation patterns. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(8):2220–2233, 2017.

[29] Stylianos I Venieris and Christos-Savvas Bouganis. fpgaconvnet: Automated mapping of convolutional neural networks on fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 291–292. ACM, 2017.

[30] Shmuel Winograd. On computing the discrete fourier transform. *Proceedings of the National Academy of Sciences*, 73(4):1005–1006, 1976.

[31] Shmuel Winograd. *Arithmetic complexity of computations*, volume 33. Siam, 1980.

[32] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 62:1–62:6, New York, NY, USA, 2017. ACM.

[33] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the 35th International Conference on Computer-Aided Design*, page 12. ACM, 2016.

[34] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[35] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. Energy-efficient cnn implementation on a deeply pipelined fpga cluster. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 326–331. ACM, 2016.

[36] Chi Zhang and Viktor Prasanna. Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 35–44. ACM, 2017.

[37] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 25–34. ACM, 2017.