



Title	Accelerating machine learning applications with FPGAs
Advisor(s)	So, HKH; Lam, EYM
Author(s)	Ho, Man-ho; 何文灝
Citation	Ho, M. [何文灝]. (2018). Accelerating machine learning applications with FPGAs. (Thesis). University of Hong Kong, Pokfulam, Hong Kong SAR.
Issued Date	2018
URL	http://hdl.handle.net/10722/263209
Rights	The author retains all proprietary rights, (such as patent rights) and the right to use in future works.

ACCELERATING MACHINE LEARNING APPLICATIONS
WITH FPGAS

SAM M.H. HO

PH.D. THESIS

THE UNIVERSITY OF HONG KONG

2018



Abstract of thesis entitled

“Accelerating Machine Learning Applications with FPGAs”

Submitted by

Sam M.H. Ho

for the degree of Doctor of Philosophy

at the University of Hong Kong

in August 2018

As the demand for a more power efficient device to handle machine learning tasks rises, FPGAs serve as a platform with great potential, due to their flexibility to adapt to novel algorithms and designs. However, FPGA users face many unique difficulties when implementing such algorithms, for instances the lack of algorithmic libraries, and the lack of custom-precision math operator libraries, or the lack of reusable common interfaces for host-software integration.

In this thesis, we start by presenting a case study on accelerating the Support Vector Machine (SVM) training on an Apache Spark cluster equipped with FPGAs, demonstrating a consistent speed-up of about $1.6\times$ against CPUs as the cluster size increases up to 8-nodes.

Then, to address the problem of the lack of math operators, we propose an open source function generator, NnCore, for floating-point non-linear operator cores built using fixed-point piecewise polynomial segments. The proposed work takes advantage of properties such as oddness/evenness and intercept-at-origin, often found in numerical functions commonly used in machine learning applications, and applies an improved segmentation algorithm that specifically handles “outlier” segments to reduce memory size. Experiments show that at single-precision, generated cores use up to 65% fewer BRAMs and runs at up to $2.2\times$ the clock speed, compared with cores generated from a



previous generic function generator. At half-precision, cores can run at $1.2\times$ higher clock speed while requiring more resource, or use a comparable number of resource but run at 12% to 45% lower clock speed.

At last, we present hDNN, a software-hardware integrated research platform for deep learning, which can serve as both the building blocks for further research, or as a baseline comparison target for benchmarks. It consists of a collection of hardware IP modules, written in HLS C++ for the Xilinx SDAccel platform, and a modified Caffe that enables support of quantized arithmetic in individual layers with user-defined quantization scheme, that can target to run on CPUs/GPUs/FPGAs. The hardware designs include a 32×32 systolic array that runs at 200 MHz for 16-bit integer/fixed-point types on a Virtex-7 FPGA, which translates to a 409.6 GOPS theoretical throughput. Together with an “Im2col” module provided, convolution operations can also be performed. Experiments show that speed-up in the range of $1.8\times$ to $32.7\times$ is achieved against an optimized CPU implementation, for the matrix multiplication part of the convolution layers in networks like Lenet, Cifar10 and Caffenet.

An abstract of exactly 391 words.

Accelerating Machine Learning Applications with FPGAs

by

Sam M.H. Ho

B.Eng., M.Phil. *CUHK*

A thesis submitted in partial fulfilment of the requirements for the degree of
Doctor of Philosophy at the University of Hong Kong.

August 2018



Declaration

I declare that this thesis represents my own work, except where due acknowledgement is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institutions for a degree, diploma or other qualifications.

Signed _____

Sam M.H. Ho

This page is intentionally left blank.

Acknowledgements

First, I want to express my sincere gratitude to my supervisor Dr. Hayden Kwok-Hay So. He has always been open-minded and gave me significant freedom to explore on research topics and ideas. I would like to thank him for his patient guidance.

I also want to thank my groupmates at HKU, who helped me along the way in many aspects. Specifically, Mr. Yuk-Ming Choi, Mr. Ho-Cheung Ng, Dr. Cheng Liu, Dr. B. Sharat Chandra Varma, Dr. Manish Kumar Jaiswal, Mr. Dominic Hung, Ms. Nina Engelhardt, Mr. Maolin Wang and Mr. Runbin Shi.

Last but not the least, I would like to thank my parents for supporting me spiritually on the PhD study.

This page is intentionally left blank.

Contents

Declaration	i
Acknowledgements	iii
Table of Contents	v
List of Figures	vii
List of Tables	ix
List of Code Listing	xi
Abbreviations	xiii
1 Introduction	1
1.1 Motivations	1
1.2 Thesis Organization	2
2 Case Study: Accelerating SVM Training on Apache Spark	7
2.1 Introduction	7
2.2 Overview	9
2.2.1 Resilient distributed datasets (RDDs) in Apache Spark .	9
2.2.2 Support Vector Machines (SVMs)	10
2.3 Accelerator Core Design	13
2.4 System Architecture	16
2.5 Benchmarks	19
2.6 Conclusion	21
3 NnCore: A Parameterized Non-linear Function Generator	23
3.1 Introduction	23



3.2	Background	25
3.3	Polynomial Approximation of Activation Functions	28
3.3.1	Function Evaluation by Polynomial Approximation	28
3.3.2	Activation Functions	29
3.4	Segment Search Algorithm	31
3.4.1	Terminologies	31
3.4.2	Algorithm	31
3.5	Core designs	38
3.6	Experiment results	40
3.7	Conclusion	44
4	hDNN: A Soft-Hard Integrated Platform for Deep Learning Research	45
4.1	Introduction	45
4.2	Related Works	47
4.3	Caffe Integration	53
4.3.1	Overview of Caffe	54
4.3.2	Extending Caffe	55
4.4	Hardware Core Designs	59
4.5	Benchmarks	64
4.5.1	Original Results	65
4.5.2	Revised Design and Results	67
4.6	Conclusion	68
5	Conclusion and Future Work	71
5.1	Future Work	72

List of Figures

2.1	Formulation of the SVM optimization problem in MLlib	10
2.2	Pseudo code of mapper process for SGD in SVM	11
2.3	Hardware block diagram of the proposed SVM mapper	12
2.4	System architecture of the FPGA design	17
2.5	Run time against number of iterations for 3T3-OAC	20
3.1	An example showing how the Tanh function is segmented after step 5	35
3.2	Evaluator core design	38
4.1	Block diagram for the DSA and the programmable region	60
4.2	Data flow in classical 2-D systolic array. Recreated from [1]. . .	61
4.3	The HLS C++ systolic array system design.	63
4.4	HLS design block diagram illustrating the deadlock issue	67
4.5	Block diagram of revised HLS design	67

This page is intentionally left blank.

List of Tables

2.1	Utilization of implementations after synthesis	14
2.2	Resource utilization of the system design, post-implementation .	18
2.3	Run time of SVM for CPUs and FPGAs against number of nodes, data size = 2400	19
2.4	Training time of SVM on CPUs and FPGAs, dataset = 3T3- OAC, size = 4800, on 8 nodes	21
3.1	Availability of various math functions from Vivado HLS	26
3.2	Number of segments after each algorithm stage. “%i”, “%d” being percentage of increase and decrease respectively.	36
3.3	Maximum no. of bits in the integer part of all fixed-point coefficients in all segments	37
3.4	Resource usage of ReLU	40
3.5	Post-Place and Route Results for single-precision operators in Virtex-7	40
3.6	Post-Place and Route Results for half-precision operators in Virtex-7.	43
4.1	Summary of DNN accelerators in recent years	53
4.2	Resource utilization in different configurations for 16-bit fixed- point.	65
4.3	Performance on matrix multiplications.	66
4.4	Convolution layer throughput.	66
4.5	Throughput in the MMULT part of CONV for the revised design.	68

4.6 Speedup of FPGA against naive CPU MMULT implementation. 69

List of Code Listing

4.1	Net instantiation	55
4.2	Modified Net declaration	56
4.3	Extended interfaces in Layer class	57
4.4	New interfaces added to Layer class	57
4.5	Original interface for forward calls	58
4.6	HLS style systolic array	62

This page is intentionally left blank.

Abbreviations

AI	A rtificial I ntelligence
ASIC	A pplication S pecific I ntegrated C ircuit
AXI	A dvanced E Xtensible I nterface
BRAM	B lock R andom A ccess M emory
CAD	C omputer- A ided D esign
CNN	C onvolutional N eural N etwork
CPU	C entral P rocessing U nit
DDR	D ouble D ata R ate
DMA	D irect M emory A ccess
DNN	D eep N eural N etwork
FPGA	F ield P rogrammable G ate A rray
GPU	G raphics P rocessing U nit
GRU	G ated R ecurrent U nit
HDFS	H adoop D istributed F ile S ystem
HLS	H igh- L evel S ynthesis
IP	I ntellectual P roperty
MKL	M ath K ernel L ibrary
PCIe	P eripheral C omponent I nterconnect e xpress
RDD	R esilient D istributed D ataset
RTL	R egister- T ransfer L evel
SVM	S upport V ector M achine
TPU	T ensor P rocessing U nit
VHDL	V ery H igh-speed H ardware D escription L anguage

This page is intentionally left blank.

Chapter 1

Introduction

1.1 Motivations

Machine learning has become ubiquitous in almost every category of computing applications, from e-mail spam filters, image/voice recognition to machine translation, or even chess/real-time strategy game AIs. The acceleration of this trend was partly due to the breakthroughs on deep-learning techniques since the 2000s, and also the exploding amount of data generated since the age of mobile internet, providing massive sample data for training. As this trend continues, the industry looked to accelerate the processing of these data on GPUs, and the use of GPU clusters with as many as 8 GPUs per node has become the mainstream in in-house clusters of companies like Facebook. There are also similar instances with up to 8 or 16 GPUs available from public cloud providers like Amazon and Google.

However, GPUs are designed to handle graphics processing tasks like video games or CAD tools acceleration. As such, they contain structures designed for those tasks that might not be needed in machine learning. Their power consumption tend to be high too, at up to 300 Watts per device for the latest Nvidia Volta GPU for example. Hence, there is a trend in the industry to look for alternative solutions that are more power efficient. Google developed its own ASICs called the TPUs (1st-gen [2] for inference and 2nd-gen [3] for

training too), for both in-house use and Cloud TPU service for the public.

FPGAs also serve as a platform with great potential, due to their flexibility to adapt to novel algorithms and designs. Microsoft deployed the project Catapult [4] in its data centres, for accelerating its own services like Bing search with FPGAs. In their latest project Brainwave [5], they reported 39.5 Teraflops sustained performance on a GRU model running on a Stratix 10 FPGA with its custom 8-bit floating-point format.

Yet much work left to be done if FPGAs are to be popularized for machine learning deployments. For CPUs or GPUs, a rich number of libraries and solvers for machine learning problems exist. Problem specific examples include LibLinear and LibSVM for Support Vector Machines. For deep learning, frameworks like Caffe, TensorFlow, Torch, MXNet, and Theano, etc., are available.

For FPGAs, no common host-accelerator programming interface were available until the adoption of OpenCL from both major providers Xilinx and Altera in recent years, and hence no machine learning software frameworks provided FPGAs support. Works from the academic community tend to focus on implementing particular techniques for a specific application, demonstrating what could be done on FPGA platforms but provided no portability and made design reuse infeasible.

In this thesis, we study various aspects of difficulties that users face when implementing machine learning applications on FPGAs, and try to provide solutions that show some form of limited portability, such that the deliverables can serve as a common base for the community to build further research upon.

1.2 Thesis Organization

In chapter 2, we start by presenting a case study on accelerating the Support Vector Machine (SVM) training on an Apache Spark cluster equipped with FPGAs.

Apache Spark is the de facto standard platform for big-data cluster computation. Although it does not include deep-learning support, it is often used for data collection and pre-processing of the sample data. There are also works [6] [7] that combine the distributed batch-processing capability of Spark with the deep-learning on GPUs capability of Caffe to perform DNN training tasks on clusters.

In the case study, we demonstrate what level of acceleration can be achieved, by using FPGAs, for a machine learning problem in a distributed computing setup, using a SVM training task as a sample application.

In the case study in chapter 2, we solved a simple linear SVM problem, with standard single-precision floating-point arithmetic on the FPGAs. Such workflow is fine for such a simple sample application, but what problems will a developer face if he wants to tackle a more complex problem, for example, neural networks, at some custom floating-point precision, to save memory space, bandwidth and computations?

The first problem a FPGA developer will face is the lack of math IP cores, which in the case of CPU programming would have been provided by standard libraries like the GNU C library. While for standard IEEE754 single-/double-precisions or even half-precision, FPGA vendors do provide IP cores for many functions, support for customized floating-point precisions is usually absent. Developers could look to research projects like FloPoCo [8] or VFloat [9] for basic operators with custom precisions, like adder, multiplier, divider, etc., but still functions like sigmoid, hyperbolic-tangent, etc., are often not provided.

As such, in chapter 3, we present a non-linear function generator we built, that can produce these frequently used maths functions at arbitrary floating-point precision the user desires. The generator's algorithm was based on previous work in [10], but with added optimizations for the class of functions that we target. Also, the generated operators are in HLS C++ format, which would allow them to be integrated into designs in modern high-level work-flow

such as Xilinx SDAccel.

With the work in chapter 3 providing the required math function cores, we can study the implementations of more complex machine learning applications on FPGAs, like neural networks. Although there is a large amount of literatures on neural network implementations on the FPGAs, almost all of them are about applying specific techniques on particular network types, and usually problem-specific software was written to drive the proposed hardware. The hardware designs are also typically tightly bounded to the targeted platform, e.g. Xilinx Zynq series, providing no portability and design reuse to the research community.

In chapter 4, we present an integrated deep-learning research platform, which is a collection of hardware modules for the Xilinx SDAccel platform, and a modified version of Caffe, extended with support for using arbitrary C/C++ data type instead of float/double only, such that it can allow reduced precision arithmetic in individual layers without breaking compatibility with the original layers.

Since the hardware design is coded in HLS C++ for the SDAccel platform, different arithmetic schemes like integer/fixed-point/floating-point and also the rounding strategy can be easily adapted to ease further research. The use of SDAccel also allows seamless design porting over different series of FPGAs from Virtex-7, Kintex UltraScale to Virtex UltraScale. The hardware designs include a 32×32 systolic array at 200 MHz for integer/fixed-point, which is the maximum target frequency achievable on the targeted Alpha Data 7v3 board in SDAccel without using RTL flow. The systolic array can be used in both fully-connected layers and convolution layers, and so it is commonly found in neural network processors from the industry, e.g. Google TPU, but it is frequently omitted from the benchmark comparisons in many research works. It would be valuable to provide it in a research platform like this, to serve as a baseline comparison.

At last, we conclude this thesis in chapter 5, with discussions on possible future work directions.

The content to be presented in chapter 2 has been published in [11] as a conference paper, and the work in chapter 3 was accepted for full-paper presentation at FPT 2017 conference, December 2017.

This page is intentionally left blank.

Chapter 2

Case Study: Accelerating SVM Training on Apache Spark

2.1 Introduction

Developing FPGA-accelerated applications for execution on large-scale distributed clusters with reasonable speed-up is a multi-faceted challenge. First, application code must be parallelized for distributed execution in the cluster, which is already challenging for many software developers. Furthermore, designers must also engage in the tedious hardware-software co-design and low-level hardware implementation flow, to achieve additional speed-up with FPGA acceleration. The combination of these challenges has made this process prohibitively difficult for most but a handful of highly trained specialized computer engineers.

A number of attempts have been made by researchers to address these challenges, that combine FPGA accelerator generation with programming models that are familiar to most application designers. For instances, a number of mixed hardware-software frameworks for executing map-reduce tasks on hybrid FPGA-CPU clusters [12, 13, 14] have been developed. On the other hand, researchers have also tried to address such usability issues through integrated software-hardware generation frameworks. For example, in the

LEAP project [15], a simple computing model based on latency-insensitive communication channels among hardware/software modules was defined. Similarly, the Lime language and run-time environment [16] integrates both hardware and software computation within a unified Java environment. While these frameworks have all demonstrated promising early results, in practice, the lack of integration with existing mainstream software frameworks such as Apache Hadoop remains a major usability challenge for most software-inclined application developers.

In this chapter, we propose a new integrated environment with the popular Apache Spark data processing framework to facilitate deployment and management of mixed hardware-software applications executing on distributed FPGA-accelerated clusters. Specifically, using the training of a Support-Vector-Machine (SVM) classifier for biological cell images [17] as a case study, we report our initial feasibility and performance trade-offs study of the proposed framework, and propose engineering practices for better management of data communication between the host CPU and FPGA accelerators to improve overall performance. In our case study, managing partition size and restricting data access to facilitate on-board data reuse played a crucial role to the resulting overall application performance. When compared to the equivalent processing on using only the host CPU cluster, our FPGA-accelerated Spark implementation of the SVM training sustained a $1.6\times$ speed-up with strong scaling as the number of cluster node increases from 1 to 8.

As such, the contribution of the work described in this chapter rests on the following aspects:

- We demonstrated the feasibility, usability and performance advantages of using FPGAs to accelerate Spark applications in a distributed cluster.
- We demonstrated engineering practices for achieving good performance when accelerating Spark applications at the level of RDD transformations.

- We demonstrated $1.6\times$ performance improvement with strong scaling by using FPGAs to accelerate an SVM cell image classifier training with minimal change to the existing software implementation.

In the next section, we will review the basic operation of the Spark application framework and SVM operation. We will then elaborate on our FPGA accelerated SVM classifier training in Section 2.3 and the target system architecture in Section 2.4. Experiment results will be reported in Section 2.5 and we will conclude in Section 2.6.

2.2 Overview

2.2.1 Resilient distributed datasets (RDDs) in Apache Spark

The RDD [18] is a programming abstraction in Spark for a partitioned, distributed list of records. They are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them with operators. They are read-only, and can only be created from either 1) data in stable storage, e.g. Hadoop HDFS, or 2) operating on other RDDs. Such operations are called transformations, with examples like *map*, *filter*, and *join*.

Another class of operations is called actions, which return a value to the application or export data to a storage system. Examples include *count*, *collect* and *save*.

RDDs are not materialized at all times, instead their *lineage* is kept. These are the transformations an RDD was derived from, such that its partitions can be computed in the first place, re-computed when there is a fault, or when a partition is evicted from memory base on a Least-Recently-Used (LRU) policy due to limited memory available. Thus, lost data can be recovered without requiring costly replications of RDD partitions.

```

1: for  $t = 1 \dots T$  do
2:   Choose  $S \subseteq N$ , where  $|S| = fraction \cdot n$ 
3:   Let  $L'_{w,i} = \begin{cases} -y_i x_i & \text{if } 1 - y_i(w \cdot x_i) > 0 \\ 0 & \text{otherwise} \end{cases}$ 
4:   Set  $f'_w := \frac{1}{|S|} \sum_{i \in S} L'_{w,i} + \lambda w^{(t)}$ 
5:   Set  $\gamma := \frac{step}{\sqrt{t}}$ 
6:   Set  $w^{(t+1)} := w^{(t)} - \gamma f'_w$ 
7: end for

```

Figure 2.1: Formulation of the SVM optimization problem in MLlib

2.2.2 Support Vector Machines (SVMs)

A support vector machine (SVM) is a binary classifier that finds a maximum margin separating hyperplane. The optimization problem is given as:

$$\text{minimize } \frac{\lambda}{2} \|w\|^2 + \frac{1}{n} \sum_{i=1}^n [1 - y_i(w \cdot x_i)]_+ \quad (2.1)$$

where $[\cdot]_+$ denotes the hinge-loss:

$$[x]_+ = \max(0, x)$$

w denotes the normal to the hyperplane, x_i the i -th training data and y_i its label. The above formula is also called the Primal problem for SVM, and can be thought of as minimizing the hinge-loss with a l_2 regularization term. Since both the terms are convex, the Lagrangian dual of the above problem can be found and is given as:

$$\begin{aligned} \text{maximize } & \sum_i \alpha_i - \frac{1}{2} \sum_i \sum_j \alpha_i \alpha_j y_i y_j \langle x_i \cdot x_j \rangle, \\ \text{s.t. } & 0 \leq \alpha_i \leq \frac{1}{\lambda n} \end{aligned} \quad (2.2)$$

This is called the Dual form of the SVM problem. Classical SVM solvers, like

```

1:  $loss \leftarrow 0$ 
2: for  $i \leftarrow 1 \dots n$  do
3:    $dotp \leftarrow 0$ 
4:   for  $j \leftarrow 1 \dots rank$  do
5:      $dotp \leftarrow dotp + w_i[j] \cdot x_i[j]$ 
6:   end for
7:    $cndt \leftarrow 1 - y_i \cdot dotp$ 
8:   if  $cndt > 0$  then
9:     for  $j \leftarrow 1 \dots rank$  do
10:       $grdt[j] \leftarrow grdt[j] - y_i \cdot x_i[j]$ 
11:    end for
12:     $loss \leftarrow loss + cndt$ 
13:   end if
14: end for

```

Figure 2.2: Pseudo code of mapper process for SGD in SVM

SVM-Light, Sequential Minimal Optimization (SMO), etc., start with this Dual problem and apply a solver on a subset of variables. However, a problem with the aforementioned classical SVM solvers is that, the runtime of these algorithms tends to scale with square of the size of the datasets, making them infeasible for large-scale datasets. For example, the run-time for SMO is $\Omega(n^2d)$, based on empirical analysis. As such, for large-scale datasets, solvers like Pegasos [19] and LibLinear [20] were proposed. The former uses “Stochastic sub-Gradient Descent”(SGD)/ “Stochastic sub-Gradient Projection”(SGP) optimizations on the Primal problem, while the latter proposes a Dual Coordinate Descent method on the Dual form.

Pegasos guarantees the number of iterations required to obtain a solution of accuracy ϵ to be $\tilde{O}(1/\epsilon)$, and total run-time to be $\tilde{O}(s/\lambda\rho)$, where s is the bound on number of non-zero features in each sample, λ the SVM regularization parameter, and ρ the optimization tolerance. For LibLinear, the number of iterations needed is $O(\log(1/\epsilon))$, and the run-time is $O(nd \cdot \log(1/\rho))$. Therefore,

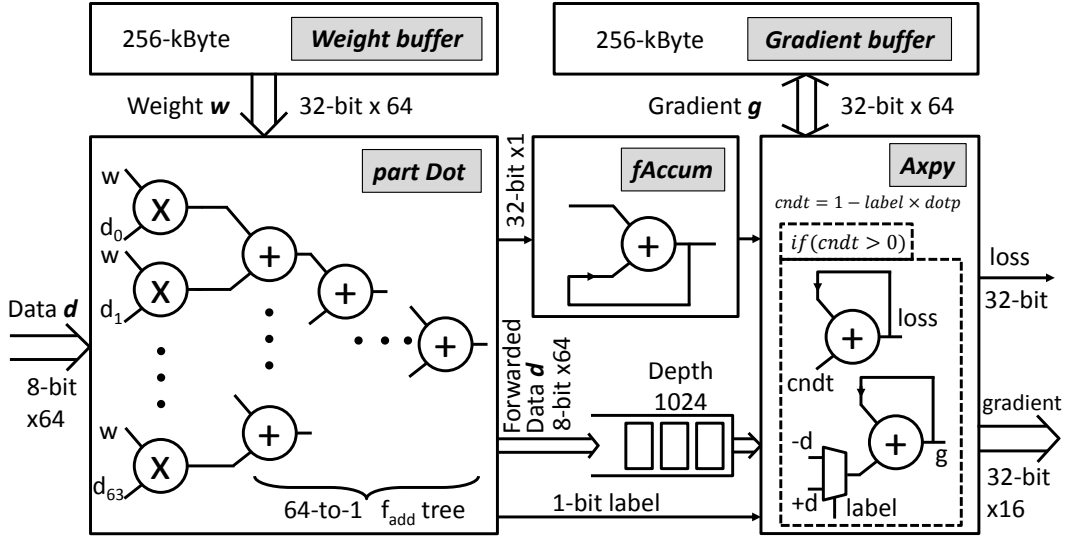


Figure 2.3: Hardware block diagram of the proposed SVM mapper

in big data frameworks like Apache Spark and Apache Flink, SGD and Dual Coordinate Descent are used respectively, instead of SMO in the popular LIBSVM.

In this chapter, the SVM algorithm being studied is the mini-batch SGD version found in Spark’s MLlib, since we are targeting big-data analytics platform for large datasets. Pseudo code of the algorithm is given in figure 2.1.

The $L'_{w,i}$ in line 3 is the part of a sub-gradient of the loss function determined by the i -th data-point, with respect to w . f'_w in line 4 is the sub-gradient of the objective function. The *fraction* variable in line 2 stands for the mini-batch fraction, which is the portion of the full training dataset to train on. When set to 1, the resulting step in each iteration is exact sub-gradient descent, whereas when chosen very small, such that only a single point is sampled, then the algorithm is equivalent to standard SGD. The γ variable is called the learning rate, and is set to be dividing a *stepSize* parameter by the square root of current iteration number.

When Spark executes the algorithm, the set of all training points N is represented as an RDD, as explained in Section 2.2.2. The sampling of set S , the (sub)gradient calculation in line 3 of figure 2.1, and part of the summation

of (sub)gradients in line 4 are executed in parallel on the cluster, by calling the “treeAggregation” function of Spark. When finished, the rest of line 4 to line 6 are calculated on a single driver node. This process repeats until the target number of iterations T is reached, or the convergence check: $\|w^{(t)} - w^{(t-1)}\| < convergenceTol * \max(1, \|w^{(t)}\|)$ returns true, where *convergenceTol* is a convergence tolerance parameter default to 0.001.

2.3 Accelerator Core Design

The cell images dataset we are working with consists of 256×256 images of 3 types of cells, 2500 for each type. This means the inputs for our SVM implementation are dense vectors of rank 65,536, which is indeed quite large if 32-bit or even 64-bit floating-point were used for each element, considering the Kintex-7 FPGA we are using offers only a total of $890 \times 18\text{k-bit} \approx 2\text{MByte}$ of on-chip BRAMs, and a double-precision vector of $65,536 \times 8\text{-Byte} = 512\text{kByte}$. So, the first design choice we made was to allow the input training vectors to stay as 8-bit integers for each element, since they are natively 256-level Grey-scale images. Any intermediate vector and scalar values are stored at 32-bit single-precision.

As discussed in section 2.2.2, the computations that are scheduled to run in parallel over the cluster includes line 3 and part of line 4 in figure 2.1. These are also known as the “mappers” in map-reduce frameworks. Although not shown in the pseudo code, the function submitted to the “treeAggregate” call of Spark also returns the accumulated loss of $L = \sum_i 1 - y_i(w \cdot x_i), \forall_i \in \{(y_i, x_i) : 1 - y_i(w \cdot x_i) > 0\}$, to keep track of the actual value of the optimization target as it shrinks over iterations. As such, our implementation follows, and the logical computation in one iteration of the mapper that we aim to accelerate on FPGA is shown in the pseudo code in figure 2.2.

Intuitively, to implement the code in figure 2.2 at lowest latency, one could combine the dot product loop in line 4-6 with the one in line 9-11. This can be

done by predictively doing the gradient accumulation on line 10, writing it into a temporary buffer, and only committing it if the condition ($cndt > 0$) turns out to be true after finishing the combined loop. In hardware, committing the temporary buffer could be implemented simply by switching the control of a ping-pong buffer. We prototyped this design and call it Ver1.

After building the Ver1, we try to integrate it into the System On-Chip, and run back-end implementation flows with EDK tool. However, this design is hard to achieve timing closure. Analyzing Ver1 we found that, the design needs significant amount of BRAMs to store the weight vector w and two gradient vectors g and g_{temp} . The three vectors alone add up to $65,536 \times 4\text{Bytes} \times 3 = 768\text{kBytes}$, and is using 43% of on-chip BRAMs. Table 2.1 shows the resource utilizations of Ver1 and Ver2 after synthesis in Vivado HLS.

Table 2.1: Utilization of implementations after synthesis

Designs	BRAM (%)	DSP (%)	FF (%)	LUT (%)
Ver1	384 (43)	466 (55)	130,473 (32)	111,030 (54)
Ver2	285 (32)	464 (55)	129,677 (31)	112,289 (55)
Available	890	840	407,600	203,800

To avoid this problem, we changed to a design with longer latency, that we call Ver2. Since we cannot start the gradient accumulation until we determine the $cndt$ from the dot product, the data vector had to be stored in case later the condition ($cndt > 0$) evaluates to true. Storing a data vector instead of a temporary gradient vector in Ver1 saved us $256\text{ kBytes} - 64\text{ kBytes} = 192\text{ kBytes}$ of BRAMs, a 11% decrease. We built a fully pipelined design, which consumes all the input every cycle ($II=1$), and a 64 kByte FIFO is used to forward the data vector to the “Axy” module for gradient calculation, as shown in figure 2.3.

When the design is kick-started, it sends command to a Xilinx “DataMover” core, and the initial weight vector is streamed-in from on-board DDR3-RAM through the 512-bit stream interface, followed by all the training data vectors

in the RAM. Since the amount of data in a RDD partition might not fit into the DDR3-RAM, it is possible that the core is started and run multiple times within a single iteration of the algorithm.

Since each feature of our input vector is 1Byte, the 512-bit interface supplies 64 out of the 65,536 dimensions every cycle. To take advantage of this, both the buffers of the weight and gradient vectors were partitioned by 64, such that the data vectors can be processed at initialization interval $(II) = 1$.

At every cycle, the “PartDot” module remaps 64 dimensions of the input vector into 32-bit floating-point, multiplies them with the corresponding dimensions of the weight vector read from BRAM, and sends the results into a 64-to-1 floating-point adder tree. This output 1 floating-point number per cycle, which is a 64-dimension-part of the dot product between w and d (x_i in earlier formulation), to the floating-point accumulator module “FAccum”. The data vector is also forwarded into a 512-bit 1024-depth FIFO, as mentioned above.

The “FAccum” module accumulates the partial dot product in 1024 cycles, and outputs the dot product to the “Axy” module for $cndt$ calculation. The “Axy” module starts by fetching a 1-bit “label”, which is essentially the y_i in previous formulations, and then fetches the dot product and calculates $cndt$. If $cndt$ is greater than zero, it adds the (sub)gradient $(-y_i \cdot x_i)$ to the cumulative gradient in the buffer. Since we are saving calculations and wires by using 1-bit for the label, the actual implementation is a mux selecting the correspondingly-signed data, as shown in the figure 2.3. After processing all the training data vectors in the DDR3, the core outputs the loss and the cumulative (sub)gradient back to the DDR3.

The core was originally designed to fully exhaust the DDR3 memory controller’s bandwidth, which is 512-bit at 200MHz, with an efficiency around $80 \sim 90\%$, as estimated in section 2.4. However, due to the need of adding debug cores during development, we were only able to deploy a debug version

of the core at 160MHz, while the core itself is believed to be capable of even higher clock rate (timing closure problems were usually due to the PCIe and DMA core at 250MHz). At 160MHz, both the DDR3 bandwidth and the SVM core could be the bottleneck, depending on the memory controller’s efficiency. It should be noted that, since the SVM core is fully-pipelined, modules earlier in the pipeline (“partDot”) could be processing the next data vector $d_{(i+1)}$, while modules deeper in the pipeline like (“Axy”) is still processing $d_{(i)}$, and the input bandwidth efficiency of the SVM core itself is 100% in this regard.

2.4 System Architecture

Our cluster consists of 8 PC nodes, each equipped with an Intel Core i5-4570S CPU at 2.9GHz, 8GB DDR3 RAM, and a Xilinx KC705 development board featuring a xc7k325tffg900-2 FPGA with 512MB DDR3 RAM. Clustering frameworks used include Apache Hadoop, for resource management (YARN) and distributed data storage (HDFS), Apache Spark for the actual in-memory data processing, and its machine learning library (MLlib) for benchmark comparisons.

For the hardware accelerated cases, we developed our own Spark application, in which we provide a FPGA driver function to the “mapPartitions” call of Spark. The driver function iterates over the RDD partition, stores each input entries into a Java “ByteBuffer”, starts a RIFFA [21] Direct Memory Access (DMA) call (which avoids copying between kernel and user space memory) to transfer it from host memory to FPGA on-board DDR3 memory, and at last transfers and returns the completed result when FPGA finishes.

Here a few tricks had to be applied, and the effects will be discussed in section 2.5. First, it is possible for Spark to schedule multiple processes to run on the same machine, but this is not supported for the hardware-accelerated cases, so we simply set the number of “vCores” in Yarn to 1 for these cases.

Second, when possible, we only transfer the training data from host to

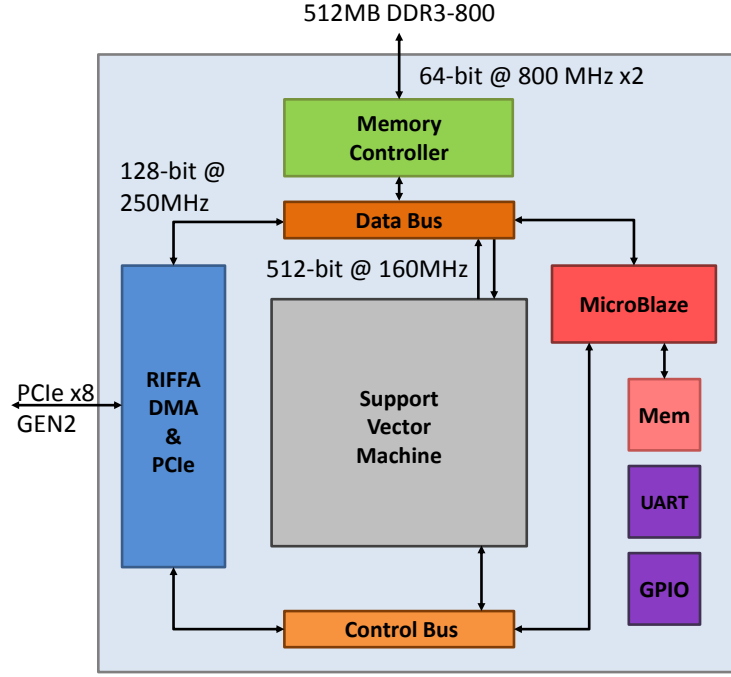


Figure 2.4: System architecture of the FPGA design

FPGA on-board memory during the first iteration of the algorithm. We call these cases “in-mem” in section 2.5, and is only feasible when:

$$\frac{TotalDataSize}{NumberOfNodes} \leq fpgaOnboardMemorySize$$

Since it is also possible for Spark to schedule computations differently across multiple iterations, the “in-mem” trick implies that the FPGAs could well be calculating on a different partition of training data, rather than which Spark had scheduled it to. Nevertheless, in our use cases there is always at most one Spark application running on the cluster, and it uses all the assigned nodes in every iteration, so the trick does not affect the correctness of the algorithm.

Although setting a partition size ($TotalDataSize/NumberOfNodes$) smaller than on-board memory size may not seem convincing at first, FPGA boards supporting a large amount of memories are actually available (e.g.16GB in AlphaData ADM-PCIE-7V3), and the scheme matches well with Spark’s notion of in-memory processing as well, so we think this is a technique that can be useful in future systems.

The third thing to note is that, since the data on HDFS were stored in the LIBSVM format (which is in plain-text), the file size is a lot larger than the actual memory footprint when data is in binary instead. Hence, the number of partitions of a file on HDFS is larger than we expected, and so we had to apply the Spark call “coalesce”, with number of nodes as argument, to reduce the partition count of the RDD before we check whether “in-mem” processing should be applied on FPGAs.

Figure 2.4 shows the overall architecture of our system on-chip, which was assembled using the Xilinx Vivado IP Integrator environment. The KC705 has a PCIe physical connection of x8 gen2, offering a theoretical bandwidth of 3.2GB/s. The RIFFA v2 core [21] is used for DMA between PC host memory and on-board memory. The RIFFA core runs at 250MHz, parameterized into 3 channels, and one of them was bridged onto the data bus for DDR3 access through a Xilinx DataMover core run at 240MHz. The remaining two channels offer respectively register read/write access to the control bus, and streaming access for applications like 10Gbps Ethernet, but the latter was not in-use in this context and hence not shown in the figure.

Table 2.2: Resource utilization of the system design, post-implementation

Resource	Utilization	Available	Utilization %
LUT	124,000	203,800	60.84
LUTRAM	13,446	64,000	21.01
FF	175,229	407,600	42.99
BRAM	324	445	72.81
DSP	464	840	55.24

The off-chip physical interface is 64-bit at 800MHz, or 1,600M transactions per second (double data rate). This offers a 12.8GB/s maximum bandwidth. The actual achievable bandwidth depends on various factors. The overall efficiency was estimated to be 81%, when access is through a 64-bit port with burst length of 128, 10 cycles read to write overhead and 5% efficiency overhead for refresh. This gives a practical bandwidth of ~ 10.37 GB/s. The on-chip

interface of the memory controller on the data bus is 512-bit at 200MHz.

Since this system was designed to adapt different applications, a MicroBlaze processor core running at 160MHz is also placed, and can be used to assist debugging or handle register read/writes. Table 2.2 shows the resource utilization from backend implementation of the system together with the accelerator core.

2.5 Benchmarks

For the benchmarks, images of 3 types of cells, namely 3T3, OAC and OST, were obtained from ATOM imaging [22], and are stored in HDFS of the cluster in LIBSVM format. For the sake of simplicity, in the benchmarks here we only demonstrate 3 pairs of binary classifications.

Table 2.3: Run time of SVM for CPUs and FPGAs against number of nodes, data size = 2400

Cell Types	Iterations	Number of Nodes		
		2	4	8
3T3-OAC	CPU	379,141	399,016	383,630
	FPGA	1,212,213	708,175	561,779
	FPGA in-mem	194,068	216,049	244,256
	Speedup [†]	1.95×	1.85×	1.57×
3T3-OST	CPU	83,917	83,917	85,355
	FPGA	269,129	162,108	126,630
	FPGA in-mem	41,180	46,554	56,523
	Speedup [†]	2.04×	1.69×	1.51×
OAC-OST	CPU	167,898	142,636	149,764
	FPGA	496,221	284,817	229,132
	FPGA in-mem	78,066	88,580	107,572
	Speedup [†]	2.15×	1.61×	1.39×

[†] FPGA in-mem vs CPU

The original dataset consists of 2,400 images for each type of cells, so for each pair of classification the dataset size is 4,800, 80% of which is used as

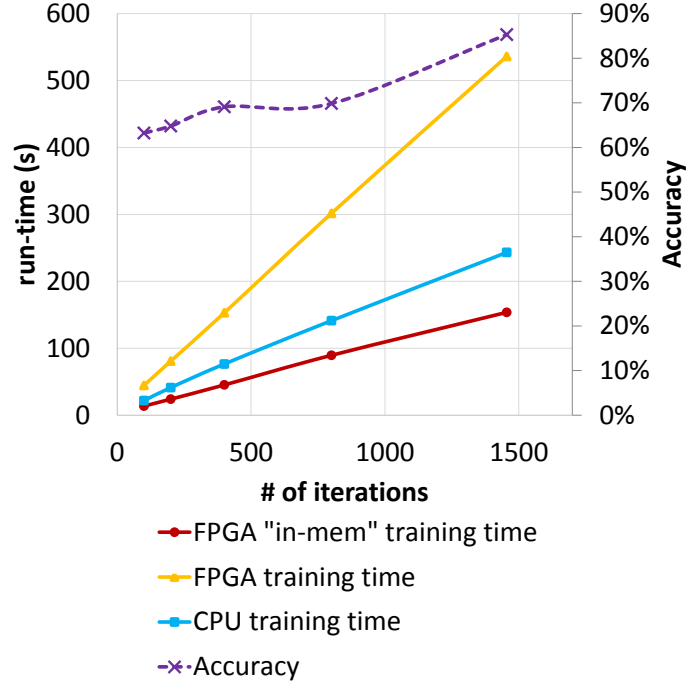


Figure 2.5: Run time against number of iterations for 3T3-OAC

training data, 20% as test data during run-time. As mentioned in section 2.4, the “in-mem” trick for FPGAs only works when partitions can fit into the on-board DRAM. In order to demonstrate the “in-mem” mode’s performance, we also included a size-reduced set, where the dataset size is 2,400, so that “in-mem” also works for 2- and 4-node configurations. These results are given in table 2.3.

Figure 2.5 shows the run-time plotted against number of iterations for the CPU, FPGA and FPGA “in-mem” cases, with 4,800 data on a 8-node configuration. Also overlaid onto the figure is the accuracy of the algorithm on the dataset against the number of iterations run. The figure shows that, the run-time of the SGD algorithm is almost linear with the number of iterations run, and the speed-up is roughly constant with a value of $\sim 1.6\times$ as shown in table 2.4.

On the other hand, the speed-up for FPGA “in-mem” versus CPU drops as the system scales out to more number of nodes. The values decrease from $2\times$ to $1.49\times$ on average.

Table 2.4: Training time of SVM on CPUs and FPGAs, dataset = 3T3-OAC, size = 4800, on 8 nodes

Iterations	Training Time (ms)				
	100	200	400	800	1455
CPU	21,771	41,265	76,449	141,175	243,153
FPGA	44,488	80,809	152,970	301,573	536,144
FPGA in-mem	13,586	23,936	45,187	89,420	153,668
Speedup [†]	1.60×	1.72×	1.69×	1.58×	1.58×
Accuracy (%)	63.24	64.81	69.12	69.85	85.29

[†] FPGA in-mem vs CPU

The reason for the diminished speed-up is as follows: as the number of nodes scales up, the workload scheduled onto each node decreases, but the setup cost for FPGA accelerations, including the DMA transfers, register read/writes and function launching, etc., remains unchanged. Therefore, the ratio between computation time versus setup time decreases.

2.6 Conclusion

In this chapter, we have presented initial results of our proposed integrated FPGA-assisted Spark application framework. In the studied case of training an SVM cell image classifier, scalable speed-up of up to 1.6× over the host CPU cluster has been demonstrated with FPGA accelerators. The speed-up, however, is achievable only with very careful tuning of the Spark environment specific to the target application to ensure the best possible data reuse on the FPGA board. Although we have achieved the intended goal of acceleration, they require laborious investigation on the application data I/O run-time behaviour and significant limitation on the amount of data we can efficiently process. These limitations, unfortunately counteract the benefits of the original goal to improve usability of the target FPGA accelerated cluster with commonly used framework like Spark. In the future, we plan to streamline this optimization process through automated accelerator generation with data I/O taken into account, and through improved communication channels between FPGAs and

the storage subsystem of the cluster.

Chapter 3

NnCore: A Parameterized Non-linear Function Generator

In chapter 2, we have demonstrated the feasibility of accelerating the training of a support vector machine model with FPGAs. However, for more complex machine learning problems, one of the first obstacles a developer will face upon working on is the lack of IP cores for some maths functions. In this chapter, we introduce NnCore, a non-linear function generator, that generates quality IP cores to facilitate machine learning model implementations such as neural networks.

3.1 Introduction

Recent years have witnessed a tremendous growth in interest to offload deep convolutional neural network inference and training on FPGA based custom computing machines for performance and power-efficiency reasons. Despite this widespread interest and the relatively regular nature of this application domain, however, efforts that would have allowed researchers to efficiently share, reuse and compare their results remain limited. Very often, researchers find themselves reinventing common features in hardware, such as a sigmoid activation function for a neural network, even when a very similar design

has already been implemented in a related work. The need to reinvent may be a result of different requirements in data precision, function accuracy, or performance, while in many cases, it was merely because the source of the related work is not readily available. This lack of design reuse has a significant impact on the designers' productivity and greatly hinders further innovation in the use of custom computing machines in this application domain, especially when application specific designs are needed to fully exploit the potential of custom computing machines.

The work in this chapter aims to help bridge this gap by introducing an open-source floating-point activation function generator that is parameterizable and extensible. The proposed framework, NnCore, is capable of generating a family of non-linear functions that are commonly used as activation functions in a range of neural network training and inference applications, and can be extended to incorporate custom functions with relative ease. Not only are these functions commonly employed in many applications, as complex non-linear functions, they are also not immediately trivial for efficient implementation in hardware. A common framework thus facilitates optimised hardware implementation of these functions be easily reused and re-targeted for different applications.

From a technical point of view, the proposed framework constructs floating-point operators using piecewise minimax-polynomials with fixed-point coefficients. To accommodate different application requirements, the framework allows users to make a trade-off between resource consumption and operator accuracy by specifying the maximum degree of the underlying polynomial, and the bit-width of the operator as input. It is, therefore, able to generate cores with standard half/single-precision, or custom-precision with variable non-standard mantissa and exponent width, which standard commercial tools do not support but are important to develop custom-made machine learning computing systems. In the current version, the proposed framework is capable of generating the hyperbolic tangent (\tanh) and its

derivative ($d'tanh$), sigmoid and its derivative ($d'sigmoid$), arc tangent ($atan$) and its derivative ($d'atan$), rectified linear unit (ReLU) and its variant ReLU6.

We thus consider the main contributions of this work are in the following areas:

- We have developed an open-source floating-point core generator for neural network activation functions and their derivatives, which is more optimised for the target function category than previous generic function generators;
- The proposed generator provides HLS C++ outputs, allowing more flexible resource/frequency trade-off during logic synthesis, and seamless integration into modern high-level tool-chain such as Xilinx SDAccel;
- We demonstrate the flexibility of the proposed framework in producing high-quality non-linear arithmetic cores with custom non-standard data precision for use in custom computing machines, which common commercial tools fail to achieve. Generated operators are faithfully rounded.

In the next section, related work in FPGA implementation for neural network will first be introduced. The design of the proposed framework will then be detailed in Section 3.3, Section 3.4 and Section 3.5. Experimental results that evaluate the quality of the generated operators will be shown in Section 3.6. At last, we conclude this work in Section 3.7.

3.2 Background

Part of the purpose of the work in this chapter is to fill in the empty IP core space where factory provided IP cores do not exist for certain target functions, or the accuracy or flexibility of such IP cores do not meet the application's requirement. Table 3.1 illustrates the availability of some math cores from Xilinx.

Table 3.1: Availability of various math functions from Vivado HLS

	Data Type	Avail.	Acc.(ULP)
atan	half	Yes, since 2017.1	n/a
	single/double	Yes	2
	custom	No	--
cosh	half/double	Yes, since 2017.1	n/a
	single	Yes	4
	custom	No	--
exp	half	Yes, since 2017.1	n/a
	single/double	Yes	Exact
	custom	No	--
sigmoid	--	No	--
sinh	half/double	Yes, since 2017.1	n/a
	single	Yes	6
	custom	No	--
tanh	half/single/double	Yes, since 2017.1	n/a
	custom	No	--
d'atan	--	No	--
d'sigmoid	--	No	--
d'tanh	--	No	--

For most functions, half-precision was only introduced until version 2017.1 of Vivado HLS, where no accuracy information was given in the documentation [23]. Accuracy information provided in the table are hence obtained from the 2016.4 version documentation [24]. This also reflects another problem, where users only with a license for older tool-chains would not be able to use the newly added cores.

It can be seen from the table that for all the functions included, only standard half/single/double-precision are supported, if available. No custom data-width is supported. Also, the accuracy differs between different operations. if a user is not aware of such difference while composing more complex operations, e.g. composing $\tanh(x)$ from $\sinh(x)/\cosh(x)$ instead of $\exp(x)$, the composed operator may suffer from suboptimal accuracy and resource usage, which we will show in section 3.6.

Since the activation functions' implementation is not the focus for many neural network application research works, in these works the activation functions tend to be approximated very roughly, e.g. piecewise linear approximate with number of segments from empirical experience [25][26][27][28], without quantifying the consequence on the networks'

accuracy. Some [29] also apply segmented minimax polynomial, which gives better accuracy.

On the other hand, as there is a long history of research on neural networks hardware, there are a large amount of previous works on implementing the individual activation functions in hardware, particularly $\tanh(x)$ and $\text{sigmoid}(x)$ as these were used since the early age of neural networks [30][31][32][33][34]. However, these works tend to focus on the resource reduction of a single function base of piecewise linear methods, compromising on the function's accuracy.

Work on more generic function evaluation, which is more similar to this work, includes [35], [36] and [10].

In [35] a hierarchical segmentation method was proposed to produce faithfully rounded segments, in an effort to reduce table size compared to uniform segmentation approaches. Fixed-point polynomials were used for evaluation, same as in this work. However, the method targets fixed-point inputs and was applied only to restricted input domains in the benchmarks with degree-1 and degree-2 polynomials.

The work [36] and [10] used similar binade partitioning approach and fixed-point polynomial approximation. However, in [36] function accuracy was not a focus, and an acceptable absolute error was used as one of the inputs to the generator. It also did not support multi-modal functions generation.

In this chapter, we based our algorithm on the work of [10], but added optimisations that are applicable to the category of activation functions used in neural networks, which we will discuss in section 3.4.

3.3 Polynomial Approximation of Activation Functions

3.3.1 Function Evaluation by Polynomial Approximation

In general, function evaluation, usually studied for elementary transcendental functions for implementing math libraries, includes the three steps: 1) range reduction, 2) evaluation, and 3) reconstruction. This is a well-studied topic with a long history and rich amount of literature. Readers are referred to materials like Muller’s book [37] for more information. Here we only give a brief introduction.

Since it is hard to approximate a function over a large range at a high accuracy, range reduction is usually used so that the approximation would only need to be done in a relatively small range. There are two kinds of reduction, including additive reduction, where the reduced argument x^* is equal to $x - kC$, k being integer and C a constant, and the multiplicative reduction, where x^* equals x/C^k .

As an example, to implement the exponential function, Tang [38] used the additive reduction with $C = \ln(2)/32$, such that the reduced argument x^* has a small range of $[-\ln(2)/64, +\ln(2)/64]$.

The reconstruction step is to deduce $f(x)$ back from $f(x^*)$. Take the above example, after we have an approximation $p(r)$ of e^{x^*} , we then construct

$$\begin{aligned} e^x &= e^{x^* + k \cdot \ln(2)/32} \\ &= 2^{k/32} \cdot p(r) \end{aligned}$$

This is a simplified version of what was given in [38], but illustrates the idea of reconstruction.

The core of the three steps, the “evaluation” can be done by using polynomial (or rational) approximations, with or without tabulation, or by using shift-and-add algorithms like CORDIC.

In terms of hardware implementations, CORDIC based methods are usually iterative, require less resource usage at the cost of performance. Polynomial estimations usually provide a higher throughput of 1 output per cycle, or at a configurable cycle-per-operation if resource sharing is built into the design.

The simplest form of a polynomial approximation that most people would have learnt is the Taylor series, but it does not give good approximations since it only approximates the target function around a particular point x . In fact, due to a theory by Chebychev, a special polynomial that minimizes the maximal error for approximation exist, known as the minimax polynomial. The Remez Exchange Algorithm is used to find minimax polynomials and is available in math packages like Mathematica and Maple.

However, it should be noted that, there is no guarantee on polynomial approximation being the most accurate, and that for some functions in some domains, a rational approximation can be superior over minimax approximation, like for the \sqrt{x} function.

3.3.2 Activation Functions

Traditionally, the hyperbolic tangent and the sigmoid function, defined respectively as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

and

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

have been used as activation functions in neural networks. They share the feature of having a limited output range, at $[-1, 1]$ and $[0, 1]$ respectively. The arc tangent function $\tan^{-1}(x)$ also has a similar curve and serves as an activation function.

Recent works in the image processing area tend to use the rectified linear

unit (ReLU) function, especially in convolutional neural networks (CNNs),

$$ReLU(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$$

otherwise, there is no general rule, other than the experience of previous works, on which activation functions should be used in what applications. A variation of the ReLU, the ReLU6, which equals $\min(\max(x, 0), 6)$, is commonly found in neural network packages like TensorFlow [39] and Torch [40], and is also available in our generator.

There are also relatively new activation functions being proposed like the Maxout [41] (which is not a simple math function that takes a single numeric input, but more like an extra layer) and the ELU [42], and are shown to perform well in various applications.

Other than the activation functions, when training a network using error back propagation, the derivatives of these activation functions are needed, and so a processor hardware for the training task will also need to have them implemented. The 1st order derivative of the hyperbolic tangent is

$$\tanh'(x) = 1 - \tanh^2(x)$$

, and that for sigmoid is

$$\text{sigmoid}'(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$

One can see that these derivatives are complex functions made up of the original activation functions, and so an increase in resource usage is expected when implemented by assembling from multiple elementary functions.

From these formulations of activation functions and the respective derivatives, one should be able to see that range reduction is hard to apply

since there is no obvious way to do reconstruction.

3.4 Segment Search Algorithm

3.4.1 Terminologies

Here we introduce some terminologies before we go further into our algorithm. The floating-point numbers format used follows that of FloPoCo [8], where the first 2-bits represents the special numbers ($\pm 0, \pm \infty, \text{NaN}$), and regular numbers are of the form:

$$s \times 2^e \times (1 + f) \text{ where } 0 \leq f < 1, s \in \{-1, 1\}$$

and the bit of one in front of the significand is implicit, as in IEEE floating-point numbers. We also refer to the binade as the region of floating-point numbers that have the same exponent value:

$$\text{bin}(x) = (\text{sign}(x), \text{exponent}(x))$$

The input domain is to be divided into segments S_i , each of which can be identified by its lower- and upper-bounds $\lfloor S_i \rfloor$ and $\lceil S_i \rceil$.

3.4.2 Algorithm

In this work, we implemented an algorithm similar to Thomas's work [10], and applied various optimisations that are feasible for our target functions. The basic idea behind the algorithm is to divide the input domain into segments based on the input and output binade, such that each resulting segment has a fixed input- and output- exponent pair, and so fixed-point polynomial approximation can be applied on the significand.

Our overall approach is as follows:

1. Make single-binade segments based on the input domain.

2. Walk along segment domain and split segments if their output crosses a binade.
3. Merge segments that have constant outputs of the same value.
4. Split segments until they can be approximated by a polynomial of degree d .
5. Split segments until they can be approximated by a fixed-point polynomial of degree at most d .
6. Detect and split “outlier” segments that have larger-than-average polynomial coefficients recursively.
7. Merge neighbouring segments if they have the same input-output binade, and are given the same approximation polynomial.
8. Merge neighbouring segments that are assigned the identity function, e.g. $y = x$, for approximation.

In the above flow, steps 3, 6, 7 and 8 are our optimisations methods, while steps 1, 2, 4 and 5 are mostly adapted from [10]. In step 1 we took advantage of the fact that many of our target functions are odd or even, so we only need to build the segments for the positive input domain in those cases, and map the sign bit of the output in the generated hardware accordingly.

In the original work of [10], after step 1 segments are also split according to their monotonicity, but since our target functions are mostly monotonic after taking advantage of their odd/even properties, we did not adapt this splitting.

In step 2), we walk along the segment domain, check if the output binade of the boundaries of segment S are equal, e.g.,

$$\text{bin}(f(\lfloor S \rfloor)) \equiv \text{bin}(f(\lceil S \rceil))$$

and split the segment if they are not. A binary search is applied here to find

the maximum x where

$$r \leftarrow \max\{x : \lfloor S \rfloor \leq x \leq \lceil S \rceil \wedge \text{bin}(f(x)) \equiv \text{bin}(\lfloor S \rfloor)\}$$

is set as the upper bound of the newly split segment.

We introduced step 3) according to the properties of our target functions. It is observed that many of the functions described approaches a limit at the ends of the input domain, and so a constant output can be set for a certain range of x . We can merge the already-split segments, as long as they can share the same output sign, exponent and the polynomial coefficients in the lookup table, in this case being a single constant value. This also saves runtime for the following steps of polynomial approximations.

After these steps, the resulting segments are all flat in both their domain and image, and we are ready to run fixed-point approximations on them. However, the runtime will be very slow if we do this directly at this stage, so approximations with real-coefficient polynomials are applied in step 4.

Given a segment S , a transformation on the target function f_t is applied, such that the transformed function maps from input significand to output significand, i.e. $f_s : [0, 1) \mapsto [0, 1) :$

$$f_s(x) = s_r \times 2^{-e_r} \times f_t(s_d \times 2^{e_d} \times (1 + x)) - 1$$

where $(s_d, e_d) = \text{bin}(\lfloor S \rfloor)$, $(s_r, e_r) = \text{bin}(f_t(\lfloor S \rfloor))$. This transformed function is then used as the approximation target for that segment.

Segments which have an approximation error $\text{err}_{\text{approx}} > 2^{-\omega_F-3}$ is split, where ω_F is the width of the significand, until all segments can be approximated with such error bound. This target serves as a heuristic to pre-split more segments, such that the fixed-point approximation in the next step will be easier. Since the purpose of this step is to speed up the overall runtime, we only perform the approximation at the maximum degree constraint by the user. The

`remez` command from Sollya [43] is used for this step, and the `dirtyinfnorm` command for quick error estimation.

Then at step 5, we try to approximate the remaining segments with fixed-point polynomials, as shown in the algorithm below. The while-loop on line 5 serves as a heuristic, such that a lower-powered polynomial has a higher priority to be chosen first if it can reach the approximation error requirement. The checking on line 14 induces another splitting of segments, in the case where no polynomials were able to reach the error constraint. Although not shown here, the “fixapprox” method on line 6 also takes as input a list of formats, which is the number of fractional bits in the coefficients of the requested fixed-point minimax polynomial. The `fpminimax` and the `supnorm` command of Sollya are used for finding the fixed-point polynomial.

Algorithm 1 SegByFixPoly

```

1: while  $S_4 \neq \emptyset$  do
2:    $S \leftarrow \{s | s \in S_4\}$ 
3:    $S_4 \leftarrow S_4 / S$ 
4:    $i \leftarrow 0$ 
5:   while  $i \leq d \wedge S \neq \emptyset$  do
6:      $(poly, err) \leftarrow fixapprox(f_s, S, i)$ 
7:     if  $err \leq 2^{-\omega_F - 2}$  then
8:        $S_5 \leftarrow S_5 \cup \{(S, poly, err)\}$ 
9:        $S \leftarrow \emptyset$ 
10:    else
11:       $i \leftarrow i + 1$ 
12:    end if
13:  end while
14:  if  $S \neq \emptyset$  then
15:     $m \leftarrow round((\lfloor S \rfloor + \lceil S \rceil) / 2)$ 
16:     $S_4 \leftarrow S_4 \cup \{\{\lfloor S \rfloor .. m\}\}$ 
17:     $S_4 \leftarrow S_4 \cup \{\{up(m) .. \lceil S \rceil\}\}$ 
18:  end if
19: end while

```

Figure 3.1 shows how an example function, in this case *Tanh*, is segmented after step 5. The range between each consecutive vertical line-pairs is one segment. It can be seen that there is one red line on every grid point on the x-axis due to step 1. The vertical lines in green are due to splitting according

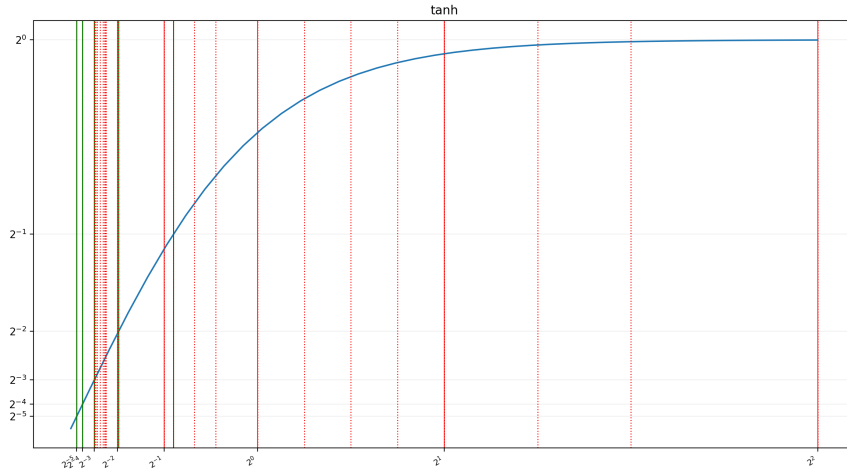


Figure 3.1: An example showing how the Tanh function is segmented after step 5

to output binades in step 2. The other vertical red lines are hence due to step 4 and 5. The figure shows that generally, the part of the target function that is steeper requires more segments, whereas the part that is more flat needs fewer segments.

Step 6 of the flow is one of the most critical steps we proposed, for generating polynomials that requires a reasonable amount of memory for the coefficients lookup table. During the development process, we found that after finishing the first 5 steps, the fixed-point polynomial of some resulting segments may contain coefficient of very large value above the integer part. This, in turn, requires significantly wider bit-width in each row in the coefficient ROM. We suspect the reason behind this is that the said segment may contain a very short sub-segment that is difficult to be approximated. This leads to the remaining parts of the segment being approximated with such a polynomial altogether. We call these segment “outliers”, and our solution for this is to filter every segment after step 5, and recursively split these outlier segments, whose polynomial coefficients have a maximum width of integer part that is two times larger than the average width.

Table 3.2: Number of segments after each algorithm stage. “%i”, “%d” being percentage of increase and decrease respectively.

Func.	Impl.	S5	SplitOutliers		MergeByPoly		MergeByX	
		segs.	segs.	%i	segs.	%d	segs.	%d
tanh	wf10d3	61	66	8.2	65	1.5	57	12.3
	wf10d4	52	58	11.5	57	1.7	49	14.0
	wf23d3	358	368	2.8	364	1.1	250	31.3
	wf23d4	307	322	4.9	318	1.2	204	35.8
d'tanh	wf10d3	49	49	0	49	0	49	0
	wf10d4	36	40	11.1	40	0	40	0
	wf23d3	2044	2044	0	2044	0	2044	0
	wf23d4	542	542	0	542	0	542	0
atan	wf10d3	67	72	7.5	71	1.4	63	11.3
	wf10d4	62	69	11.3	68	1.4	60	11.8
	wf23d3	448	459	2.5	455	0.9	341	25.1
	wf23d4	357	373	4.5	369	1.1	255	30.9
d'atan	wf10d3	67	70	4.5	70	0	70	0
	wf10d4	59	62	5.1	62	0	62	0
	wf23d3	2937	2945	0.3	2945	0	2945	0
	wf23d4	1160	1170	0.9	1170	0	1170	0
sigmoid	wf10d3	69	77	11.6	77	0	77	0
	wf10d4	57	124	117.5	124	0	124	0
	wf23d3	2109	2109	0	2109	0	2109	0
	wf23d4	598	598	0	598	0	598	0
d'sigmoid	wf10d3	45	45	0	45	0	45	0
	wf10d4	34	36	5.9	36	0	36	0
	wf23d3	2013	2013	0	2013	0	2013	0
	wf23d4	535	535	0	535	0	535	0

In step 7, we try with the least effort to merge any neighbouring segments that have the same binade and the same polynomial assigned. We do this because the binary splitting at steps 4, 5 and 6 may well be too pessimistic, since the boundary created from splitting a segment will never be removed once committed. But this is necessary for a fast runtime and good user experience, and so in this step, we pay the minimum effort trying to rescue some of these obvious cases.

In step 8, we perform an “identity function merging”, as we see that for functions that pass through the origin, lots of segments around $x = 0$ would be assigned the approximation $y = x$ when x is very small. Since the segments that will most likely be merged due to this technique is often clustered around 0, usually only one resulting segment is left on the lookup-table, and so we can hard-code these addresses range information into the control logic of the polynomial evaluator, without increasing the bits to be stored in the coefficient lookup-table.

Table 3.3: Maximum no. of bits in the integer part of all fixed-point coefficients in all segments

Func.	Impl.	SegByFixPoly		SplitOutliers	
		avg p	max p	max p	%d
tanh	wf10d3	1	9	0	100
	wf10d4	2	22	4	81.8
	wf23d3	1	38	1	97.4
	wf23d4	2	72	4	94.4
d'tanh	wf10d3	4	7	7	0
	wf10d4	3	9	7	22.2
	wf23d3	13	16	16	0
	wf23d4	16	20	20	0
atan	wf10d3	1	9	1	88.9
	wf10d4	1	17	3	82.4
	wf23d3	1	38	2	94.7
	wf23d4	1	72	3	95.8
d'atan	wf10d3	2	14	2	85.7
	wf10d4	2	14	2	85.7
	wf23d3	3	37	4	89.2
	wf23d4	3	37	6	83.8
sigmoid	wf10d3	2	7	5	28.6
	wf10d4	2	8	4	50
	wf23d3	13	16	16	0
	wf23d4	14	20	20	0
d'sigmoid	wf10d3	3	7	7	0
	wf10d4	3	8	7	12.5
	wf23d3	13	16	16	0
	wf23d4	16	20	20	0

Table 3.2 shows the number of segments remaining after each of the steps 5 ("SegByFixPoly"), 6 ("SplitOutliers"), 7 ("MergeByPoly") and 8 ("MergeByX"). Table 3.3 shows the average and the maximum number of integer bits in the fixed-point coefficients of all polynomial segments before and after step 6 ("SplitOutliers").

From table 3.2 it is clear that the steps "MergeByPoly" and "MergeByX" are effective only for functions that pass through the origin $y = x = 0$, which is as expected. The "MergeByX" step is capable of reducing segment counts by up to 35.8%, which is quite significant.

Looking at both tables it can be seen that except for the sigmoid function at half-precision and degree 4, the "SplitOutliers" step only increased the number of segments by less than 12%, and reduced maximum coefficient bit-width of integer part by up to 100%. This is important because the polynomial coefficients are stored in ROM, with each segment occupying for one row, and the bit-width decided by the one segment with the highest power of polynomial

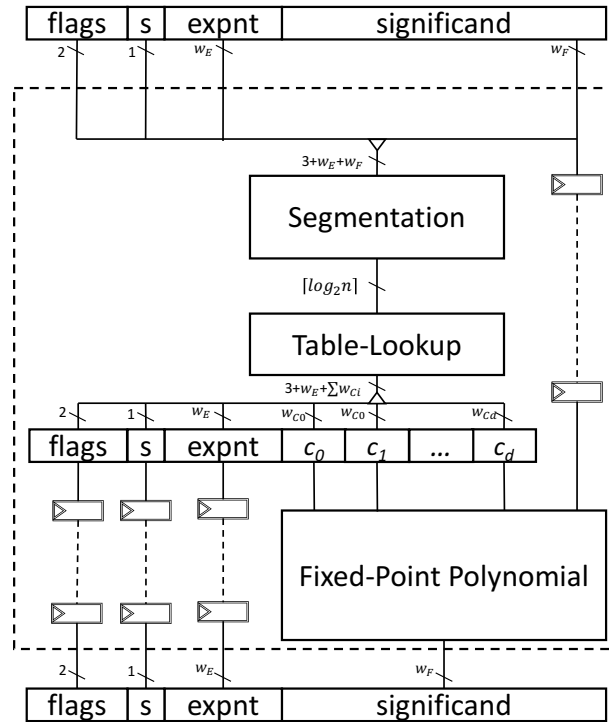


Figure 3.2: Evaluator core design

and widest coefficient. Reducing this maximum integer bit-width allows all segments to be stored in a ROM with narrower bit-width.

3.5 Core designs

Our floating-point function approximator is similar to that in [10], consisting of "segmentation", "table-lookup" and "fixed-point polynomial evaluation" stages, as can be seen in figure 3.2. The difference in ours, in particular, is that the evaluator reads also the address to be used for table-lookup, compares its value against a number of compile-time hard-coded values, to determine whether the evaluation can be skipped, as mentioned in step 8 of section 3.4.

The segmentation module takes the whole floating-point number as input, and returns the address of the corresponding segment in the coefficient lookup table. The module is built with $k = \lceil \log_2(n) \rceil$ comparators, each compares the input x with a segment boundary read from register, effectively performing a binary segment search, with each stage recovering one bit of the segment

address output. It also checks the boundary value against hard-coded maximum, such that it can accept any segment count n .

The bit width in the LSBs of coefficients in the ROM follows the heuristic suggested in [10], which is:

$$l_i = 2^{-\omega_f + i - 1}$$

To achieve faithful rounding, the requirement would be:

$$\epsilon_{approx} + \epsilon_{eval} + \epsilon_{round} \leq 2^{-\omega_f}$$

Since we set the approximation error bound in line 7 of algorithm 1 to be $2^{-\omega_f - 2}$, and the final rounding error is $2^{-\omega_f - 1}$, our evaluation error budget would be $2^{-\omega_f - 1}$. To satisfy this goal, the bit width of the output of each stage of the Horner form evaluation is set to:

$$\begin{aligned} a_d &= c_d \\ a_i &= round(c_i + x \times a_{i+1}, \quad 2^{-\omega_f - i - g}) \\ a_0 &= round(c_0 + x \times a_1, \quad 2^{-\omega_f}) \end{aligned}$$

where $round(x, r)$, and g is a predetermined number of guard bits to achieve faithful rounding.

Currently, generated operators are output in Xilinx HLS C++ form factor, for the ease of integrating the cores into other projects as IPs. However, there is no limit in the generator for supporting other output formats, e.g. to use FloPoCo operators for the fixed-point evaluator. Along with the generated core, we also generate the corresponding test vector file. A C++ test bench was written to take these test vector files and verify against the generated cores for correctness in code generation. The operators are faithfully rounded (ULP = 1) by construction, so testing for accuracy is not the main purpose of this test bench.

Table 3.4: Resource usage of ReLU

	Prec.	LUT	FF	BRAM	DSP
ReLU	5,10	9	0	0	0
	8,23	17	0	0	0
ReLU6	5,10	19	0	0	0
	8,23	35	0	0	0

Table 3.5: Post-Place and Route Results for single-precision operators in Virtex-7

Func.	Impl.	Spec.	Lat.	Slice	LUT	FF	DSP	BRAM	SRL	Clk.
tanh	HLS comp.	exp	75 --	1098 --	2955 --	4123 --	14 --	0	143 --	517.9 --
		sinh, cosh	117 (1.56)	2841 (2.59)	9210 (3.12)	8953 (2.17)	30 (2.14)	0	233 (1.63)	342.7 (0.66)
	NnCore	d = 3	63 (0.84)	555 (0.51)	1082 (0.37)	2282 (0.55)	5 (0.36)	4	243 (1.70)	359.6 (0.69)
		d = 4	84 (1.12)	702 (0.64)	1109 (0.38)	3061 (0.74)	8 (0.57)	3	264 (1.85)	566.6 (1.09)
d'tanh	HLS comp.	exp	97 --	1284 --	3342 --	4894 --	17 --	0	159 --	501.5 --
		d = 3	66 (0.68)	495 (0.39)	1076 (0.32)	1808 (0.37)	8 (0.47)	28	350 (2.20)	337.6 (0.67)
	NnCore	d = 4	88 (0.91)	955 (0.74)	2740 (0.82)	2553 (0.52)	12 (0.71)	8	330 (2.08)	458.5 (0.91)
atan	HLS synth.	--	124 --	2817 --	9414 --	8979 --	2 --	0	413 --	370.2 --
	FloatApprox	d = 3	40 (0.32)	562 (0.20)	1471 (0.16)	1590 (0.18)	6 (3.00)	3	362 (0.88)	244.4 (0.66)
		d = 4	49 (0.40)	640 (0.23)	1753 (0.19)	1904 (0.21)	8 (4.00)	4	542 (1.31)	247.8 (0.67)
	NnCore	d = 3	79 (0.64)	758 (0.27)	1486 (0.16)	3065 (0.34)	6 (3.00)	2	270 (0.65)	554.3 (1.50)
		d = 4	56 (0.45)	595 (0.21)	957 (0.10)	2677 (0.30)	7 (3.50)	4	203 (0.49)	393.1 (1.06)
d'atan	HLS comp.	mul, div	51 --	479 --	1178 --	2195 --	3 --	0	53 --	548.5 --
		d = 3	43 (0.84)	709 (1.48)	1918 (1.63)	1953 (0.89)	6 (2.00)	32	373 (7.04)	230.9 (0.42)
	FloatApprox	d = 4	51 (1.00)	679 (1.42)	1604 (1.36)	2295 (1.05)	8 (2.67)	22	576 (10.87)	232.1 (0.42)
		d = 3	66 (1.29)	1911 (3.99)	6466 (5.49)	2098 (0.96)	6 (2.00)	11	358 (6.75)	402.1 (0.73)
	NnCore	d = 4	90 (1.76)	802 (1.67)	2143 (1.82)	2159 (0.98)	8 (2.67)	19	357 (6.74)	468.6 (0.85)
sigmoid	HLS comp.	exp	75 --	722 --	1840 --	2771 --	7 --	0	90 --	511.5 --
		d = 3	88 (1.17)	1373 (1.90)	3812 (2.07)	2978 (1.07)	9 (1.29)	30	367 (4.08)	419.1 (0.82)
	NnCore	d = 4	89 (1.19)	944 (1.31)	2423 (1.32)	3069 (1.11)	12 (1.71)	11	334 (3.71)	454.3 (0.89)
d'sigmoid	HLS comp.	exp	83 --	764 --	1962 --	2986 --	10 --	0	121 --	523.6 --
		d = 3	83 (1)	942 (1.23)	2338 (1.19)	2657 (0.89)	9 (0.90)	23	358 (2.96)	467.3 (0.89)
	NnCore	d = 4	88 (1.06)	795 (1.04)	1884 (0.96)	2426 (0.81)	12 (1.20)	13	335 (2.77)	465.3 (0.89)

3.6 Experiment results

In our experiment setups, the target device was set to the Alpha-Data ADM-PCIE-7V3 board, featuring a xc7vx690tffg1157-2 chip from Xilinx.

Unlike other generated functions, the ReLU and ReLU6 are generated from our older code base published in [44]. Generated codes are in Verilog RTL format, and the post-placement and route results are shown here in table 3.4 for the sake of completeness. These are functionally just constant comparators, therefore require only LUTs to implement and no BRAMs are needed.

The tool version used in the experiments was Xilinx Vivado HLS 2016.2. NnCore is written in Python 3, while all the math operations used within the algorithm were passed to Sollya 6.0 [43] for processing. Due to the lack of license for the newer version of the Vivado tool (2017.1 and up), we were not able to provide benchmark results for the newer half-precision operators introduced since version 2017.1, as listed in table 3.1 of section 3.2. Hence, other than the $\text{atan}(x)$ function which is available from the tool, other HLS implementations used as a baseline for comparisons were composed using primitive operators as

follows.

$\tanh(x)$ is given by either $(e^x - e^{-x})/(e^x + e^{-x})$, or $\sinh(x)/\cosh(x)$. It's derivative given by $d\tanh/dx = 1 - \tanh^2(x)$; $\operatorname{datan}(x)/dx = 1/(1 + x^2)$; $\operatorname{sigmoid}(x) = 1/(1 + e^{-x})$ and $d\operatorname{sigmoid}(x)/dx = e^{-x}/(1 + e^{-x})^2$.

Since part of our algorithm is based on the work of [10], comparison with the operators generated by it is also provided, referred to as "FloatApprox" in the tables. The source code was downloaded from an obscure branch of FloPoCo [8] named "random", at [45]. Since the version of FloPoCo on that branch was marked as 2.5.0, we followed the installation instruction of FloPoCo 2.5.0 and used FPLLL 3.0.12 library and Sollya 3.0 as dependencies.

Resource usage and clock achieved after out-of-context placement and route flow is presented. To find the maximum achievable clock, we set an initial target clock period, decrease the clock period at a 0.1 ns step and re-iterate starting from high-level synthesis, until the post-route timing target was not met. The result with the highest clock speed achieved is reported.

For the operators generated by "FloatApprox", which were in VHDL RTL, we extracted the auto-generated out-of-context synthesis and placement-and-route scripts from the HLS projects, and applied to the generated VHDL.

Typically, generated cores give more all-rounded results on both resource usage, design latency and clock speed when the highest degree of polynomial is set to 3 or 4, hence results of only these settings are provided in the experiments, but such assumption may not hold true for every target functions. Table 3.5 shows the design latency and post-placement and route results of generated functions in single-precision. Numbers in brackets are the results normalised by those of the reference HLS design. The reader should bear in mind that, both "FloatApprox" and "NnCore" generated cores are faithfully rounded (ULP = 1), while the reference composed cores mostly have lower accuracies, so it would be natural if generated cores involve higher resource usage.

Note that for the single-precision configuration, "FloatApprox" failed to

generate a design in 8 out of 12 setups, which is significantly worse than the original results presented in [10]. We tried our best effort to reach the author of [10], to see if any specific dependencies might be needed, but without success as of the time of writing. Yet we consider the results presented here valid, since anyone following the same steps as us would produce the same results, if no additional knowledge on how to replicate the results in [10] is given.

From Table 3.5 we can see that composing the $\tanh(x)$ with $\sinh(x)/\cosh(x)$ could involve more than $2\times$ the resource required compared to using $\exp(x)$.

For $\tanh(x)$, $d'\tanh(x)$ and $\operatorname{atan}(x)$, both "NnCore" and "FloatApprox" generated designs use fewer slices, LUTs and flip-flops than references, but more DSPs for $\operatorname{atan}(x)$ and more BRAMs for all 3.

For $d'\operatorname{atan}(x)$, $\operatorname{sigmoid}(x)$ and $d'\operatorname{sigmoid}(x)$, slices and LUTs used are about under $2\times$ and flip-flops about $1\times$, with the exception of $d'\operatorname{atan}(x)$ at degree = 3 for "NnCore".

In all functions except $\operatorname{atan}(x)$, generated cores use between $1.7\times$ to $6.75\times$ the shift-registers for "NnCore", between $1.63\times$ to $10.87\times$ for "FloatApprox". On achievable clock frequencies, most "NnCore" generated designs run at about $0.7\times$ to $1.5\times$ the clock of reference designs.

Compared with "FloatApprox", "NnCore" generated designs use up to 65% fewer BRAMs, 63% fewer shift-registers, and run at up to $2.2\times$ the clock speed at the same setting. The clock speed advantage may be due to the fact that "NnCore" generates HLS C++ code as output, enjoying better optimisations during high-level synthesis.

Table 3.6 shows the comparison between "FloatApprox" and "NnCore" generated functions at half-precision setting. As mentioned above since no composite implementation is possible before Vivado HLS 2017.1, no reference implementation is provided. In general, "NnCore" generated cores required more resources than that of "FloatApprox" except for DSPs, while being able to run at about $1.2\times$ higher clock speed.

Table 3.6: Post-Place and Route Results for half-precision operators in Virtex-7.

Func.	Impl.	Deg.	Lat.	Slice	LUT	FF	DSP	BRAM	SRL	Clk.	Runtime
tanh	FloatApprox	3	22	105	252	375	3	0	87	542.0	2434 m 1 s
		4	27	109	263	407	4	0	102	555.2	2476 m 16 s
	NnCore	3	56	274	433	1108	3	0	121	656.2	1 m 18 s
		4	62	315	508	1399	4	0	141	663.6	1 m 41 s
	NnCore-min	3	26	157	263	716	3	0	63	479.2	--
		4	27	180	325	770	4	0	84	348.9	--
d'tanh	FloatApprox	3	22	130	302	487	3	0	108	541.7	3652 m 27 s
		4	27	135	334	522	4	0	140	561.2	3643 m 46 s
	NnCore	3	56	227	468	894	3	0	137	644.7	1 m 51 s
		4	66	278	573	1117	4	0	158	648.1	1 m 51 s
	NnCore-min	3	24	145	329	529	3	0	79	345.1	--
		4	26	156	416	570	4	0	97	335.5	--
atan	FloatApprox	3	22	99	241	370	3	0	81	550.1	22 s
		4	27	113	270	419	4	0	108	554.6	40 s
	NnCore	3	53	252	431	1012	3	0	118	634.5	2 m 2 s
		4	60	295	504	1247	4	0	137	638.2	2 m 32 s
	NnCore-min	3	26	157	277	719	3	0	68	470.1	--
		4	29	194	344	819	4	0	82	336.9	--
d'atan	FloatApprox	3	22	105	270	428	3	0	84	539.1	9 s
		4	27	132	320	492	4	0	124	544.7	13 s
	NnCore	3	59	218	454	815	3	0	135	644.3	17 m 37 s
		4	67	257	530	997	4	0	154	635.7	54 m 41 s
	NnCore-min	3	26	148	327	527	3	0	85	362.2	--
		4	29	178	386	590	4	0	100	325.8	--
sigmoid	FloatApprox	3	22	99	210	328	3	0	76	578.7	30 s
		4	27	99	230	363	4	0	92	586.2	19 s
	NnCore	3	66	320	653	1225	3	0	157	634.1	2 m 33 s
		4	68	328	744	1258	4	0	165	628.5	3 m 23 s
	NnCore-min	3	26	165	444	602	3	0	84	317.1	--
		4	32	199	470	668	4	2	110	334.9	--
d'sigmoid	FloatApprox	3	22	128	307	486	3	0	108	535.3	10 s
		4	27	133	341	519	4	0	140	525.8	1 m 0 s
	NnCore	3	53	199	418	833	3	0	133	644.7	6 m 7 s
		4	63	256	500	1035	4	0	147	634.5	7 m 27 s
	NnCore-min	3	24	127	292	496	3	0	79	354.6	--
		4	26	146	324	533	4	0	95	337.6	--

The higher resource usage is partly due to that when we iterate for the highest clock achievable, we start from the high-level synthesis step, which in turn tries to generate faster circuits at a compromise of resource usage. To prove this, we also provide a setup named "NnCore-min" in table 3.6, where we set the target clock to 250MHz and report the actual clock and resource usage achieved.

At this setting "NnCore" generated cores use comparable number of slices and LUTs to "FloatApprox" cores, up to 30% fewer number of shift-registers in most cases, about $2\times$ the number of flip-flops in $\tanh(x)$, $\text{atan}(x)$ and $\text{sigmoid}(x)$, or otherwise similar number of flip-flops in $d'\tanh(x)$, $d'\text{atan}(x)$ and $d'\text{sigmoid}(x)$. The clock speed achieved range from 12% to 45% lower than that of "FloatApprox" cores.

In addition to the latency and resource usage metrics same as in table 3.5, here we also listed the generators' runtime on our machine with an Intel Core i7-3820 CPU at 3.6GHz. For "NnCore-min", since re-run of core generations is

not needed, generator runtime for these entries was skipped in the table.

Normally this metric is not relevant as long as the generator finishes in a reasonable amount of time, say within several hours. However, during our experiments, we found that the "FloatApprox" sometimes required an extensive amount of time to run. For the functions $\tanh(x)$ and $d'\tanh(x)$, it took up to 2.5 days to produce the generated operators. This, together with the issue of failing to generate most cores at single-precision, seriously hindered the use of the "FloatApprox" generator. Comparatively, "NnCore" generated cores within a reasonable amount of time across all target functions.

3.7 Conclusion

In this chapter, we present NnCore, an open-source, parameterizable and extensible activation function generator for FPGA based machine learning applications. NnCore allows easy generation of non-linear functions that are common to many machine learning applications, with a goal to further facilitate design reuse among the research community. NnCore constructs operators using piecewise fixed-point minimax-polynomials and allows users to specify the maximum polynomial power, bit widths of the exponent and the significand as parameters.

Experimental results show that NnCore is capable of generating functions with a comparable number of slices, LUTs and flip-flops usage, a fewer number of block RAMs and shift-registers, and significantly higher clock frequencies compare to a previous generic function generator. Generated cores are faithfully rounded ($ULP = 1$), which is not available from composited functions. NnCore also showed stability across all generation settings, which is not shown in the previous generator. The use of HLS C++ as output format also allows integration of cores into modern high-level work-flow such as Xilinx SDAccel, which is a unique feature.

Chapter 4

hDNN: A Soft-Hard Integrated Platform for Deep Learning Research

4.1 Introduction

In recent years, many breakthroughs in deep learning techniques have been achieved, driven by a large community of both academia and the industry. For software developers, a long list of deep learning frameworks are now available, like Caffe [46] from BVLC, Microsoft's Cognitive Toolkit (CNTK), TensorFlow [47] from Google, Theano, Torch, etc., providing capabilities like GPU processing or even distributed computation on cluster of servers, and also the ease of development for application deployment or neural network researches.

For hardware developers of ASICs or FPGAs, we have also seen a surge in the amount of publications on yet more novel architectures and techniques for neural network processing in conference proceedings and journals.

However, most of these publications present specific techniques for a particular kind of neural networks, e.g. convolutional neural network (CNN),

and are typically bounded to some target platform, e.g. Xilinx Zynq series, etc. These together make them more of a demonstration on what and how such neural network implementations could be done on those platforms, but not a shared work that the community can reuse for further researches, as in the software alternatives like Caffe.

Also, the lack of a common framework for hardware developers makes direct benchmark comparisons impossible. Often, researchers could only present comparisons with number provided in other publications, which is probably implemented on a different FPGA model, different host CPU and with custom host software code.

We can use software CNN as an example. In the baseline convolution layer in Caffe, an “Im2col + Gemm” implementation was used. “Im2col” is a image-to-column-vector sub-routine originally found in Matlab, and “Gemm” the well-known matrix multiply function in BLAS linear algebra libraries. This is considered the classical standard implementation nowadays, which was popularized through Caffe.

As more research effort were being put on improving the performance of convolution layers, researchers have applied methods like “Winograd”[48] or “FFTs” that can out-perform “Im2col + Gemm” in many cases.

In the hardware world, however, in many publications only a single implementation of their own, be it “direct”, “Winograd” or “FFT”, is presented, without providing a baseline, e.g. “Im2col + Gemm” for comparison. This is understandable, as implementing multiple methods could involve an extensive amount of work.

As such, in this chapter, we present a software-hardware integrated research platform, hDNN, which can serve as both the building blocks for further research, or as a baseline implementation for benchmark comparisons. It consists of a collection of hardware IP modules, written in HLS C/C++ for the Xilinx SDAccel platform, and a modified version of Caffe.

The hardware modules include a systolic array for matrix multiplication, that can be configured for fixed-point/integer with user-defined bit-width, or standard half/single/double floating-point types.

For INT16 data-type, at a 32×32 setup, it can achieve 200 MHz clock, which is the maximum target clock frequency in SDAccel for the Alpha Data 7v3 board, giving a 409 Gops theoretical computation throughput.

Hardware modules for input caching and “Im2col” are also included, such that the systolic array can be used to handle both convolution (CONV) layer and fully-connected (FC) layer computations.

The modified version of Caffe that we introduce can support the use of quantized, limited precision arithmetic for network inference. We added generic interfaces such that the scheme of quantization can be programmed by the developer of each layer. The modification we made to Caffe is largely backward-compatible, such that the large number of existing layers can still be used and run on CPU, while user’s custom low-precision layers can be added to run on CPU/GPU/FPGA as the developer wishes.

The rest of this chapter is organised as follows. In Section 4.2 we review the related works on DNN accelerators in ASICs and FPGAs, and approaches to use quantized arithmetic in neural network computations. In Section 4.3 we describe how we modified and extended the Caffe framework, to allow quantized arithmetic in individual layers, regardless of whether an accelerator or the CPU is to be used for the computation. In Section 4.4 we describe the hardware modules we provide, written in HLS C++ for the Xilinx SDAccel platform. In Section 4.5 we present benchmark results of the proposed software-hardware platform, and finally we conclude this chapter in Section 4.6.

4.2 Related Works

As we mentioned in Section 4.1, there are a huge amount of DNN/CNN hardware accelerator literatures. Here we introduce some of the more significant

and recent works. As we are providing a modified version of Caffe, we also review some related works on quantized arithmetic for neural networks, that are software implementations/emulations of modes intended to be applied on hardware designs.

One such work is Ristretto, an open-source (available at [49]) extension on Caffe by Gysel et al. [50]. Ristretto can analyse pre-trained networks and condense models by using fixed-point arithmetic and representation instead of floating-point. It can also fine-tune the resulting fixed-point network. Quantization formats provided include “dynamic fixed-point”, “minifloat” and “power-of-two”.

One thing to note about the work is that, since the authors took the approach of only-extending the original Caffe, the actual C language data-type used to hold the quantized data under the hood is still 32-bit float. Therefore, the aim of the work was not to provide acceleration, but to provide a tool to produce quantized models for hardware accelerated use cases.

Another framework for quantization of neural networks is Gemmlowp [51], an open-source library owned by Google, that is also used within TensorFlow [47]. The low-precision used is 8-bit unsigned integer. To put it simple, for a floating-point range with known minimum and maximum values, the framework tries to represent the range with integers 0 through 255, but with adjustment such that one of the quantized integer value exactly represents the real value “0”, since zero-padding is often used in neural networks.

In [52] Han et al. introduced “Deep Compression”. They demonstrated with the use of pruning, trained quantization and Huffman coding, storage required by AlexNet/VGG-16 can be reduced by $35\times/49\times$ respectively. Later in [53], Han et al. proposed a processor called the Energy Efficient Engine (EIE) that can perform inference on the compressed network model from “Deep Compression” directly, accelerating the resulting sparse matrix-vector multiplication with weight sharing. The EIE was said to provide 102 GOPS for

compressed networks, corresponding to 3 TOPS equivalent for uncompressed networks, being $24,000\times/3,400\times$ more energy efficient than CPU and GPU respectively.

In another work [54], Han et al. proposed a load-balance-aware pruning method to compress Long Short-Term Memory (LSTM) networks, together with an Efficient Speech Recognition Engine (ESE) that can work directly on the compressed model. They implemented the ESE on a Xilinx XCKU060 FPGA at 200 MHz clock, yielding a performance of 282 GOPS, corresponding to 2.52 TOPS on an uncompressed network, at a power dissipation of 41 Watts. A $43\times/3\times$ speed-up and $40\times/11.5\times$ energy efficiency against Core i7 5930k CPU and Pascal Titan X GPU was reported.

In [55], Gupta et al. studied the effect of low-precision fixed-point for network training, and observed that by using stochastic rounding, deep networks can be trained using only 16-bit fixed-point number representation, with little to no degradation in classification accuracy. They also demonstrated an systolic array based accelerator, that implements low-precision fixed-point arithmetic with stochastic rounding. The 28×28 systolic array running at 166 MHz clock on a Xilinx Kintex K325T FPGA yields a throughput of 260 GOPS at 7 Watt power consumption.

For neural network accelerator architectures, we can roughly categorize them into two main styles: 1) single-accelerator, like a systolic array, and 2) multiple processing elements, a.k.a. GPU-like style. On the industry's side, it was disclosed (forum post [56]) that in Xilinx's "AlexNet on Caffe platform on Nimbix" cloud, a 8-bit/16-bit integer systolic array (binary available from Xilinx's SDAccel Examples Github repository [57]) with 2048 DSP running at 400 MHz clock (logic at 300 MHz) was used, achieving a throughput of over 4.2 TOPS and about 2000 Int8 images/s CNN inference, on a Xilinx KCU1500 board with Kintex KU115 FPGA.

In Google's 1st generation Tensor Processing Unit (TPU) [2], a 256×256

8-bit systolic array, with a 28 MiB software-managed on-chip memory, was included for matrix multiplications and convolutions. They achieved $30\times/80\times$ higher TOPS/Watt compared to an Intel Haswell CPU and a Nvidia K80 GPU respectively. Although not disclosed in the same paper, one can find in a related patent of Google [58], how a convolution can be computed with the systolic array inside TPU, by rotating the weight matrix to be convolved and also using selection multiplexors within each systolic element.

On the other hand, Microsoft recently unveiled a deep learning acceleration platform codenamed Project BrainWave [5][59][60], that runs on Intel Stratix 10 FPGAs. With the use of a custom 8-bit floating-point format (“ms-fp8”), they reported a 39.5 TFLOPS sustained performance for a large gated recurrent unit (GRU) model. Currently the framework supports running with Microsoft’s CNTK and Google’s TensorFlow. The architecture contains a sea of soft DNN Processing Units (DPUs), each with a matrix-vector-multiply unit, runs a custom instruction set, and connected through a network-on-chip (NOC) interconnect and broadcast trees.

In the academia, [61] Suda et al. presented a systematic design space exploration methodology for OpenCL-based FPGA accelerator for CNN. On an Altera Stratix-V P395-D8 board, they achieved 136.5 GOPS for convolution, and 117.8 GOPS for the entire VGG network. Convolutions were done using matrix multiplication, with a “implicit im2col” style known in GPU programming, where input features were rearranged on-the-fly in the on-chip memory of FPGA such that each receptive field to be convolved becomes a column vector.

Zhang and Li [62] proposed an analytical performance model to analyse CNN kernels for OpenCL based FPGA implementations. They identified that the key bottleneck is the on-chip memory bandwidth, and proposed a new kernel design to address it. On an Altera Arria 10 GX1150 board, their proposed design achieved 866 GFLOPS floating-point at 370 MHz clock, and 1.79 TOPS

16-bit fixed-point at 385 MHz clock, for the VGG network model. The key design components include a 16×16 PE array, 2-D dispatcher for work-items, and multicast 2-D PE-to-local memory interconnects to improve data reuse. For convolutions, data flattening and rearrangement similar to [61] was used to convert 2-D convolution into matrix multiplication. Weights and input feature maps were stored in DRAM.

In [63] Zhang et al. proposed a roofline model to identify solution of CNN with the best performance and lowest FPGA resource requirement. As a case study, they implemented a CNN accelerator on a Xilinx VC707 board and achieved performance of 61.62 GFLOPS at 100 MHz clock.

In [64] Qiu et al. proposed a dynamic-precision quantization method and an efficient convolver design, targeting CNN acceleration on embedded FPGAs. For the VGG16n model with 8/4-bit quantization, the accuracy loss was only 0.4%. For a case study the VGG16-SVD model was implemented on the Xilinx Zynq ZC706 board, achieving 187.8 GOPS for convolution layers and 137 GOPS for the full CNN, at 150 MHz clock, using 16-bit quantization.

In [65] Li et al. proposed an end-to-end CNN accelerator, with all layers working concurrently in a pipelined style. For FC layers batch-based computing was applied, and two different computing access patterns were used to reduce required on-chip buffers. For the AlexNet implemented on a Xilinx VC709 board, they achieved performance of 565.94 GOPS and 391 FPS at 156 MHz clock.

Ma et al. [66] proposed a CNN acceleration scheme and architecture after searching the design configurations for minimized memory access and maximized resource utilization. They demonstrated a VGG-16 CNN model on an Altera Arria 10 GX 1150 FPGA, achieving 645.25 GOPS throughput and 47.97 ms latency.

In [67] Zhang and Prasanna exploited the use of Fast Fourier Transform (FFT) and Overlap-and-Add (OaA) to reduce computations for CNNs on Intel

QuickAssist, a CPU-FPGA platform with coherent shared memory. A 2D parallel convolver and concurrent processing on CPU were used to improve overall system performance. For the VGG-16, AlexNet and GoogLeNet models, their design sustained 123.48 GFLOPS, 83.00 GFLOPS and 96.60 GFLOPS of performance. The main advantage of the design was that around $3.3 \sim 3.4 \times$ fewer multipliers were used. For GoogLeNet their design was $5.56 \times$ better in performance compare to a 10-core Intel Xeon at 2.8 GHz running 16 threads.

Aydonat et al. [68] introduced a novel architecture called a Deep Learning Accelerator (DLA), written in OpenCL for FPGAs, that maximizes data reuse and minimizes external memory bandwidth usage. They also used the Winograd transform to boost CNN performance. They reported a performance of 1020 img/s, or 23 img/s/Watt, for the AlexNet model, on an Intel Arria 10 device, which translates to 1382 GFLOPS. The key components of the DLA architecture were PEs arranged in a daisy-chain, each consisting of dot-product units, accumulators and caches.

In [69] Lu et al. proposed an architecture for implementing Winograd algorithm for CNNs on FPGAs. Their design employed line buffer to reuse feature map data among different tiles. Multiple pipelined Winograd PE engines were used for parallel processing. They achieved 1006.4 GOPS / 3044.7 GOPS for convolutional layers and 854.6 GOPS / 2940.7 GOPS for the overall AlexNet and VGG-16 respectively, on a Xilinx ZCU102 platform.

All the aforementioned previous works on hardware accelerators for DNNs are summarized in table 4.1. It can be seen that, except for the case of commercial products like TPU of Google and DPU of Microsoft, none of the works provided any software framework integration. Most of them required custom host software code to run, if any existed. This makes it difficult for the research community to reuse designs and modularly improve upon previous published designs.

Hence, in our work, we implemented a modified Caffe, such that further

Table 4.1: Summary of DNN accelerators in recent years

Author	In	Type	Conv	ASIC / FPGA	Prec.	Host sw.	Perf.
Gupta et al.[55]	CoRR 2015	DNN	--	Xilinx K325T	16bit fixed stoch. rnd.	custom	260 Gops
Zhang et al.[63]	FPGA 2015	CNN	direct	Xilinx VC707	32bit float	none	61.6 Gflops
Suda et al.[61]	FPGA 2016	CNN	gemm	Altera Stratix V	8-16bit fixed	custom	117.8 Gops
Qiu et al.[64]	FPGA 2016	CNN	direct	Xilinx ZC706	16bit dyn. fixed	none	137 Gops
Li et al.[65]	FPL 2016	CNN	direct	Xilinx VC709	16bit fixed	none	565.9 Gops
Ma et al.[66]	FPGA 2017	CNN	direct	Altera Arria 10	8-16bit fixed	custom	645.2 Gops
Zhang and Li [62]	FPGA 2017	CNN	gemm	Altera Arria 10	16bit fixed / 32bit float	custom	866 Gops / 1790 Gops
Zhang and Prasanna [67]	FPGA 2017	CNN	FFT	Intel QuickAssist	32bit float	custom	83 Gflops
Aydonat et al.[68]	FPGA 2017	CNN	Winograd	Altera Arria 10	16bit float	custom	1382 Gflops
Lu et al.[69]	FCCM 2017	CNN	Winograd	Xilinx ZCU102	16bit fixed	custom	854.6 Gops / 2940.7 Gops
Jouppi et al.[2]	ISCA 2017	--	gemm	TPU	8bit int	Tensor- Flow	92 Tops

research can be done on individual layers, without the need of re-implementing the corresponding host control software, or other common layers that may not be the topic of interest in the further research work.

4.3 Caffe Integration

As we mentioned in previous sections, most academic work on DNN acceleration provided no framework integration, therefore it is hard for the community to reuse. The only exception that we were able to find is the work of DiCecco et al. [70]. However, their work used the 32-bit float data type, which is the default in Caffe and so no modifications were needed in that regard. And that makes the work not of interest to us, as we are building a research platform where the data precision, representation and bit-width are definitely research topics that many would focus on.

We started by modifying upon the OpenCL branch [71] of Caffe, which can work on Intel and AMD OpenCL platforms. It is mostly similar to the original Caffe, but with added OpenCL device memory calls to perform the same task

originally done with Cuda calls for GPU processing, and also infrastructures for compiling and launching OpenCL kernels. Our modifications are generic, in that quantized arithmetic can be used on both CPU or FPGA, and are not bound to the Xilinx SDAccel platform that we use.

4.3.1 Overview of Caffe

In Caffe, a “Blob” is a wrapper object of the actual data being processed. It provides synchronization capability between CPU and GPU under the hood, and provides a unified memory interface for different data, e.g. images, parameters, etc. The abstraction is basically an N-dimensional array, so for images it would typically be a 4-D blob with dimensions equal number $N \times$ channel $K \times$ height $H \times$ width W .

To access the actual data on CPU, the programmer use the calls `cpu_data()` and `mutable_cpu_data()`, and on GPU `gpu_data()` and `mutable_gpu_data()` correspondingly. This design is such that Caffe can do memory synchronization lazily behind the scene, to minimize data transfer between host and GPU. When using GPUs, as long as all layers have GPU implementations, all the intermediate data will remain in the GPU.

After Caffe is started for an inference or a training task, a “Net” object is created, and “Layer” objects are being created for each layer described in a “prototxt” file. The input and output blobs of the layers, called the “bottom” and the “top”, are being created and connected to the respective layers. A layer type has the following major methods that the developer needs to implement: `LayerSetUp`, `Reshape`, `Forward_cpu` and `Backward_cpu`.

The `LayerSetUp` is for one-time initializations, like reading parameters, fixed-size allocations, etc., during model initialization, and is optionally implemented. The `Reshape` is for computing the sizes of top blobs, allocating buffers, and other work that depends on the shapes of the bottom blobs. The `Forward_cpu` and `Backward_cpu` are the actual functions the layer computes

during the forward- and backward-pass, respectively, where the latter is optionally implemented. If corresponding calls named `Forward_gpu` and `Backward_gpu` are implemented, and during runtime a GPU is available, these will be called instead of the “_cpu” variants. Yet for layers without GPU implementation, Caffe automatically falls back to using the “_cpu” calls.

4.3.2 Extending Caffe

One obstacle to adding low-precision arithmetic support to Caffe is how the data type of any variable within an object of the aforementioned “Net”, “Layer” and “Blob” classes is defined. The code snippet in listing 4.1 below shows how the “Net” instance was created within the train mode top level function.

Listing 4.1: Net instantiation

```
// Instantiate the caffe net.
Net<float> caffe_net(FLAGS_model, caffe::TEST,
                    Caffe::GetDefaultDevice(), FLAGS_level, &stages);
caffe_net.CopyTrainedLayersFrom(FLAGS_weights);
```

The data type “float” is passed to the constructor of the “Net” class via a template variable, and this is in turn passed to each of the “Layer” and “Blob” objects during their construction. Hence, the same data type is used throughout the whole network, in each and every layer’s input features, parameters (weight, bias, etc.) and output activations, and also all the intermediate variables used.

Although there is corresponding code such that changing the C++ “float” type to “double” will still allow Caffe to work, the same does not hold true for integer types like “int”, “short” and “cl_half” (which is a place holder type for OpenCL host code for half-precision that is aliased to “short”), due to no equivalent arithmetic operations exist for some maths functions originally done in “float” type.

As was mentioned in section Section 4.2, one possible solution is to keep the “float” data type, but to treat the stored value differently according to the quantization scheme needed, as was done in Ristretto [50] Caffe. But this

is not a good approach for hardware integration, since the on-board memory required would stick to the original for “float”, instead of decreased to half or one forth, when a smaller data type like “half” or “int8” is used.

In some study on using low precision (fp16) format for recurrent neural network (RNN) training to improve computation throughput, it was proposed that two copies of the weights can be kept, one in higher precision (fp32) and one in low precision (fp16), such that update is done to the high precision copy, and that copy is rounded to low precision to perform computations.

Inspired by this, our approach to extending Caffe for low precision support involves creating an extra low precision data path, and to only calculate and use the low precision copy when a layer is “lowp”-capable. To maintain compatibility with all the existing layers code, we have to keep the “float” type in the “Net” object, and so we have to add a new “Net” template class such that it takes one more template variable, for the data type used in the “lowp” data path.

Listing 4.2: Modified Net declaration

```
template<typename Dtype, typename Ldtype>
class Net<Dtype, Ldtype> {
    vector<shared_ptr<Blob<Ldtype> > > blobs_lowp_;
    vector<vector<Blob<Ldtype>*> > bottom_vecs_lowp_;
    vector<vector<Blob<Ldtype>*> > top_vecs_lowp_;
```

The code in listing 4.2 illustrates the major addition we made. Since the new “Net” template class has different template parameters, we have to copy all the source code of the original “Net” class, and modify the template parameter list for all the class methods. Other than that, we added “lowp” version of lists of blobs correspondingly. These are basically initialised the same way as the original non-“lowp” versions, so is not shown here.

On the other hand, for the “Layer” class we cannot add new template variable to the parameter list, since that will break compatibility with current layers inheriting the “Layer” class. Instead, we created interfaces similar to

the original ones mentioned in Section 4.3.1 for the “lowp” data path. Part of them is shown in listing 4.3.

In addition to these, we also need to add new interfaces to support “lowp”, and to let quantization and de-quantization take place automatically, when there are subsequent layers such that only one of them has “lowp” implementation. These are shown in listing 4.4.

Listing 4.3: Extended interfaces in Layer class

```
template<typename Lptype>
void SetUp_lowp(const vector<Blob<Dtype>*>& bottom,
               const vector<Blob<Lptype>*>& bottom_lowp,
               const vector<Blob<Lptype>*>& top_lowp) {
    LayerSetUp_lowp(bottom, bottom_lowp, top_lowp);
    Reshape_lowp(bottom, bottom_lowp, top_lowp);
}

template<typename Lptype>
inline Dtype Forward_lowp(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top,
                        const vector<Blob<Lptype>*>& bottom_lowp,
                        const vector<Blob<Lptype>*>& top_lowp,
                        const vector<bool*>& bottom_need_dequantize,
                        const vector<bool*>& top_need_dequantize);
```

Listing 4.4: New interfaces added to Layer class

```
virtual void CopyParam_lowp() {
}

virtual bool has_lowp_cpu(){ return false; }
virtual bool has_lowp_gpu(){ return false; }
virtual bool has_lowp(){ return has_lowp_cpu() || has_lowp_gpu(); }
virtual inline void QuantizeBottom_lowp(
    const vector<Blob<Dtype>*>& bottom,
    const vector<Blob<int16_t>*>& bottom_lowp,
    const vector<bool*>& bottom_need_dequantize) {
}

virtual inline void DequantizeTop_lowp(
    const vector<Blob<int16_t>*>& top_lowp,
    const vector<Blob<Dtype>*>& top,
    const vector<bool*>& top_need_dequantize) {
}
```

Here although for QuantizeBottom_lowp and DequantizeTop_lowp their

“lowp” input/output were hard-coded to type “int16_t”, the corresponding declaration for “int8_t” type also exists in the source code, therefore allowing both 8-bit and 16-bit data types to be used to hold the data in the host and on the device memory.

The `QuantizeBottom_lowp` and `DequantizeTop_lowp` methods were meant to be overloaded by the user, allowing them to choose whatever quantization scheme they want in their layers, e.g. half-precision (fp16), `Gemmlowp` int8, stochastic rounding, etc., as discussed in Section 4.2. This would also allow different fractional width in fixed-point arithmetic to be used in individual layers of the same network model.

With these added interfaces, there are still problems to be solved. During inference, the `ForwardFromTo` method of the “Net” object is called, which in turn calls the `Forward` method of the “Layer” class with each layer instance, which are child instances of “Layer”. The `Forward` method then calls into the `Forward_cpu` and `Forward_gpu` methods implemented within each individual layer class as needed. The function prototype of them is shown in listing 4.5.

Listing 4.5: Original interface for forward calls

```
/** @brief Using the CPU device, compute the layer output. */
virtual void Forward_cpu(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top) = 0;

/**
 * @brief Using the GPU device, compute the layer output.
 *      Fall back to Forward_cpu() if unavailable.
 */
virtual void Forward_gpu(const vector<Blob<Dtype>*>& bottom,
                        const vector<Blob<Dtype>*>& top) {
    // LOG(WARNING) << "Using CPU code as backup.";
    Forward_cpu(bottom, top);
}
```

We can see that, for individual layer instances, these calls only provide them with access to blobs that are the layers’ direct input or output. But as mentioned above we intentionally defined the `DequantizeTop_lowp` interface to be implemented by each “lowp” layer. So when a “lowp” layer has its

output connected to a “normal” layer, either 1) the “lowp” layer needs to run de-quantize regardless of its subsequent layer, which is wasteful, or 2) the subsequent “normal” layer needs to call the de-quantize method of its previous layer, which it has no access to.

To solve this, we instead added checking to conditionally run the de-quantize of a subsequent layer inside the `ForwardFromTo` method of the “Net” class. If two subsequent layers are both “lowp” layers, the de-quantize of the earlier layer would not be run, to avoid extra DMA transactions since the de-quantize is intended to be run on CPU.

4.4 Hardware Core Designs

As mentioned in previous sections, our hardware designs are written in Xilinx HLS C++, to be used in the SDAccel platform. The SDAccel platform supports the OpenCL programming model, with host C/C++ code running on CPU for application control, and kernel code written in OpenCL/HLS C++/RTL are compiled into hardware compute units for application deployment. The development board is physically connected to the host PC through a PCIe connection.

SDAccel supports a range of development boards from several vendors with different FPGA models from Xilinx, ranging from Virtex-7, Kintex UltraScale to Virtex UltraScale. These development boards and the development tools are also available from cloud service providers like Amazon, IBM and Nimble, making SDAccel an ideal target platform for this work.

To achieve support on such a broad range of platforms, SDAccel provides pre-built sub-systems, called the design support archives (DSAs), for the supported boards. These DSAs include system IPs such as PCIe DMA, DDR controller, debug cores, etc., and are built into the static region of a partial re-configuration flow, so that they can be pre-programmed onto the FPGA boards. The user’s kernel designs are synthesised and implemented during

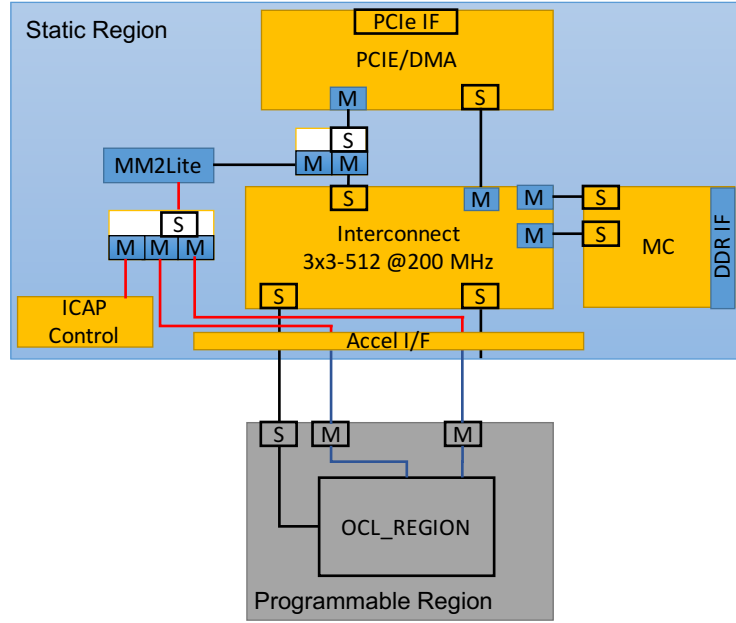


Figure 4.1: Block diagram for the DSA and the programmable region

compile time, and then partially re-configured onto the FPGA at run-time. Figure 4.1 shows the block diagram for the DSA and the programmable region.

The major hardware design we included is a 2-D systolic array, that can be used for both matrix multiplications and convolution operations. Systolic arrays are well-established design pattern that can be found in many research and commodity designs, some of them [55, 2] we have discussed in Section 4.2. The main advantage of such pattern is that it achieves high computation throughput at a relatively low external memory bandwidth requirement, and also maximizes data reuse. For example, for matrix multiplications, a 2-D systolic array may take two $O(n^2)$ inputs and perform $O(n^3)$ multiply and accumulate operations in $O(n)$ cycles.

Classically, data are flowed into each processing element (PE) from two directions, and then each PE forwards the inputs of the current cycle to its adjacent PEs in the next cycle. This data flow sequence is illustrated in the example in figure 4.2, for the multiplication of two 3×3 matrices A and B with a 2-D systolic array of the same size. One can notice the highest input bandwidth is achieved when 3 elements were read from each of A and B in the

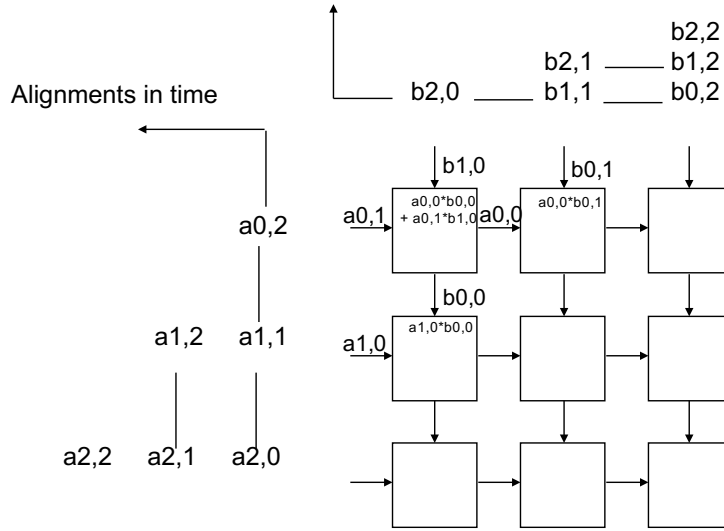


Figure 4.2: Data flow in classical 2-D systolic array. Recreated from [1].

same cycle, and the highest computation throughput in a cycle is $3 \times 3 = 9$ multiply-accumulates, which needs a longer input sequence from A and B to be achieved.

However, as we are developing using HLS C++, which is at a higher abstraction level than RTL designs, it is hard to describe the exact same behaviour in HLS C++ code in an efficient way. On the other hand, HLS is good for using simplified code to generate functionally equivalent RTL. As such, it was suggested in Xilinx’s SDAccel example design that the following simple nested for-loops be used to generate a functionally equivalent systolic array, as shown in listing 4.6.

The multiply-accumulate (MAC) within each PE is realized in line 9 of the code. By using the PIPELINE pragma in line 2, the 2 for-loops on lines 3 and 4 are fully unrolled, thereby creating a 2-D array of MACs that run in parallel every cycle. The key difference between this style and the classical version above is that, the elements of input matrices A and B are now broadcast to every PE of the corresponding row/column, instead of being forwarded from a neighbouring PE. But they are logically equivalent in that, they can do $O(n^2)$ MACs every cycle.

One drawback for the HLS style is that, since broadcasting was used for

Listing 4.6: HLS style systolic array

```

1  systolic1: for(int k = 0; k < a_col; k++) {
2      #pragma HLS PIPELINE
3          systolic2: for(int i = 0; i < MAX_SIZE; i++) {
4              systolic3: for(int j = 0; j < MAX_SIZE; j++) {
5                  int last = (k==0) ? 0 : localC[i][j];
6
7                  int a_val = (i < a_row && k < a_col)? localA[i][k] : 0;
8                  int b_val = (k < b_row && j < b_col)? localB[k][j] : 0;
9                  int result = last + a_val*b_val;
10
11                 localC[i][j] = result;
12             }
13         }
14     }

```

the PE communications, more high-fanout wiring would be created during the physical implementation flow, and so the physical design does not scale-up as well as the classical style. To compensate this effect and to achieve the maximum clock target of 200 MHz on our target Alpha Data 7v3 board, we had to divide a 32×32 systolic array into 4 tiles of 16×16 arrays. Also, the input matrices A and B are fetched from FIFOs streaming from off-chip DDR reads, instead of being in local block RAMs as shown in listing 4.6.

The 32×32 systolic array is targeted for 16-bit fixed-point/integer/floating-point data types, with $32 \times 16 - \text{bit} = 512 - \text{bit}$ matching the widest data-width configurable in SDAccel for global DDR memory access through each AXI master interface. However, as can be seen in figure 4.2, the input matrix A needs to be read column-by-column, while the input matrix B needs to be read row-by-row.

But in C/C++ programming for the host CPU, arrays are physically stored in row-major order, where all data within a row are stored in consecutive locations followed by data within the next row. If this data layout is to be kept on the FPGA’s global DDR, extra logic would be needed to rearrange input data before passing the data to the systolic array, because the address of data within each of the 512-bit word fetched from on-board DDR must be continuous. These extra rearrangement logics will be a waste of FPGA resources.

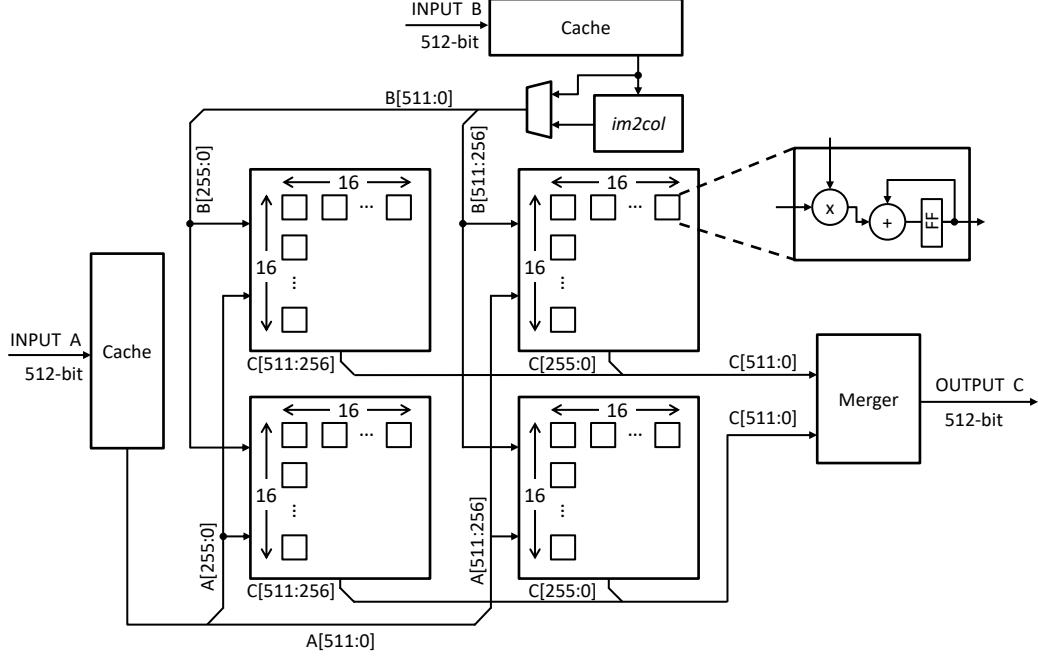


Figure 4.3: The HLS C++ systolic array system design.

To avoid this, on the FPGA’s global DDR we store the input A in a tile-wise column-major order, whereas for input B a tile-wise row-major order is used, with the tile height/width set to 32, respectively. E.g. for A , address 32-63 would store the first 32 elements of column 1, instead of the 32nd through the 64th elements of row 0. Depending on the use of tile rows of A or tile columns of B as the outer loop bound, the output C can either be tile-wise column major or tile-wise row major. As such, we have implementations for both of the configurations.

Other than matrix multiplications, we also use the systolic array for convolutions. Our systolic array does not feature infrastructures that facilitate filter rotations and input selection MUXs, required for the approach in [58] as discussed in Section 4.2. Hence, we provided the “im2col” module, that can be optionally enabled to replace the input source matrix B for the systolic array, to rearrange input blob into column vectors.

The rearranged column vector contains $C_{in} \times k_h \times k_w$ elements of the original input blob in a single column, where C_{in} is the number of channels in the

input, k_h and k_w the kernel height and kernel width of the convolution filter respectively. The number of column vectors after rearrangement equals to the $Output_h \times Output_w$, which is the spatial dimensions for one convoluted filter output. During matrix multiplication, the input A would be the convolution filter weights, with C_{out} rows equal the number of output channels.

Since neighbouring column vectors represent the field of the convolution filter after a stride, they usually share some common input pixels/activations. Storing the whole rearranged column matrix in DDR memory would be space consuming, and so our implementation used the “implicit” style, where the column matrix is expanded on-the-fly, and its output stream to the systolic array directly without being stored in any block RAM. There are also caches for both the matrix A and matrix $B/im2col$ input, such that for input A that has total size smaller than $2048 \text{ entries} \times 64 \text{ Bytes}$, or input $B/im2col$ input with size of each tile column smaller than $6144 \text{ entries} \times 64 \text{ Bytes}$, the input would only be read from DDR for the first time. These cache sizes were set according to some commonly used CONV layers, that are small enough to be fully cached, selected from popular CNNs like AlexNet. The overall system design is shown in figure 4.3.

4.5 Benchmarks

In this section, we present the benchmark results of the systolic array design described in Section 4.4. In the experiment setup, the host machine features an Intel Xeon E5-2650 v3 CPU at 2.3 GHz clock, 64 GB of RAM, and two Alpha Data 7v3 development boards, each with a Xilinx Virtex-7 XC7VX690T-2 FPGA and two 8GB ECC-SODIMM memory. In the experiments, we are only using one of the two FPGA boards.

As mentioned, the systolic array can be configured in compile time to generate either tile-wise column major or tile-wise row major result matrix. As an early effort, we implemented the column major version, with a 4096-entry

Table 4.2: Resource utilization in different configurations for 16-bit fixed-point.

	Available	Utilization	
		Ver1	Ver2
LUT	429600	112095 (26%)	130871 (30%)
LUTRAM	172800	16132 (9%)	19682 (11%)
FF	859200	102565 (12%)	127143 (15%)
BRAM	1470	154.5 (11%)	266.5 (18%)
DSP	3600	1093 (30%)	1056 (29%)
CacheA	--	4096 x 512-bit	2048 x 512-bit
CacheB	--	0	6144 x 512-bit
Im2col	--	No	Yes
CLK	--	200 MHz	200 MHz

cache only available on the path of input A , which we refer to in table 4.2 as “Ver1”. Table 4.2 shows the resource utilization of both “Ver1” and “Ver2” designs. One can note that the “Im2col” module and the extra amount of cache in the “Ver2” design only contributed to mostly BRAMs in the resource usage. Both designs achieved 200 MHz, which is the maximum target for the Alpha Data 7v3 development board in SDAccel. This gives a theoretical throughput of $200 \text{ MHz} \times 32 \times 32 \times 2 = 409.6 \text{ GOPS}$. Also note that we are only using less than 30% of resources in every aspect, and so a significant headroom is left for users to add extra kernel cores for other layers or to improve performance.

For basic functionality verifications, we performed matrix multiplications with a custom OpenCL host code on the “Ver1” design, and reported the computation throughput achievable in table 4.3. The input matrices A and B have the same dimensions equal to $dim \times dim$. From the table, we can see that generally the larger the input data, the higher the computation throughput. The design achieved the highest efficiency of 281.87 GOPS at input dimension 4096, and after that the throughput dropped, due to the fact that one tile row of the input matrix now cannot be fully cached and so caching is disabled.

4.5.1 Original Results

To test the systolic array for convolution operations, we used the convolution layer settings from several well-known CNNs, such as LeNet, Cifar10 and

Table 4.3: Performance on matrix multiplications.

Dim	Size	NumOps	Time(Secs)	Throughput
32	2 kB	0.07 M	0.000077	0.85 GOPS
64	8 kB	0.52 M	0.000063	8.31 GOPS
128	32 kB	4.19 M	0.000087	48.03 GOPS
256	128 kB	33.55 M	0.013698	2.45 GOPS
512	512 kB	268.44 M	0.013428	19.99 GOPS
1024	2 MB	2147.48 M	0.013499	159.08 GOPS
2048	8 MB	17.18 G	0.064096	268.04 GOPS
4096	32 MB	137.44 G	0.487605	281.87 GOPS
8192	128 MB	1099.51 G	10.770000	102.07 GOPS

Table 4.4: Convolution layer throughput.

	Input				Stride	Pad	Kernel			#Operations	Time(ms)	GOPS
	N	C	H	W			C	H	W			
Lenet-conv1	100	1	28	28	1	0	20	5	5	57,600,000	14.1458	4.07
Cifar10-conv1	100	3	32	32	1	2	32	5	5	491,520,000	16.2402	30.27
Caffenet-conv1	50	3	227	227	4	0	96	11	11	10,541,520,000	182.58	57.74

Caffenet, which is a modified version of the AlexNet. The result is shown in table 4.4. During inference, each batch is processed in an iteration, and such batch size is listed in the table as “N” under “Input”. The “C” under “Input” is the number of input/activation channels, and the “C” under “Kernel” is the number of output features.

Only 3 result entries are available in table 4.4, due to a deadlock issue that occurs for all the other settings on the 3 networks, which we will explain and propose a solution below.

Consider the block diagram in figure 4.4, where the HLS design has 2 independent AXI master bus ports for reading the matrix A and matrix B inputs from the off-chip DDR through the DSA. This design is in itself valid, however since the DSA for the 7v3 board supports only a single off-chip DDR port, the two AXI master ports are connected to the same AXI “interconnect” IP, sharing the same AXI slave input of the DDR controller IP.

Our experiments show that, this AXI “interconnect” setup cannot handle the two simultaneous bus requests from the independent “readA” and “readB” modules correctly. When the “readA” module holds the AXI bus during a read, and a back-pressure by the systolic array occurs, it cannot clear the incoming

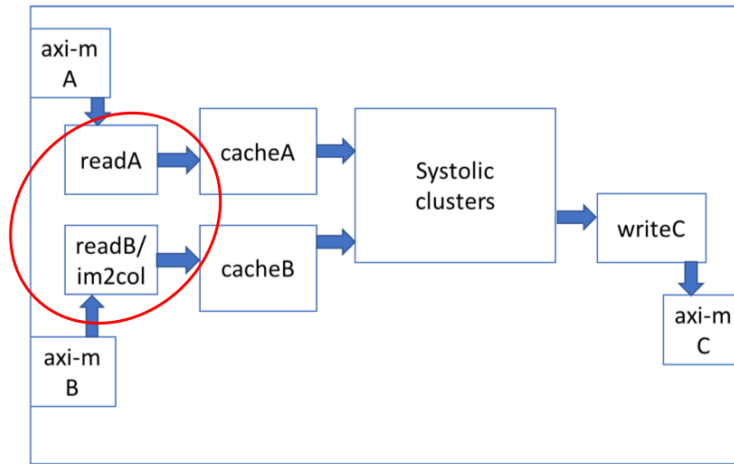


Figure 4.4: HLS design block diagram illustrating the deadlock issue

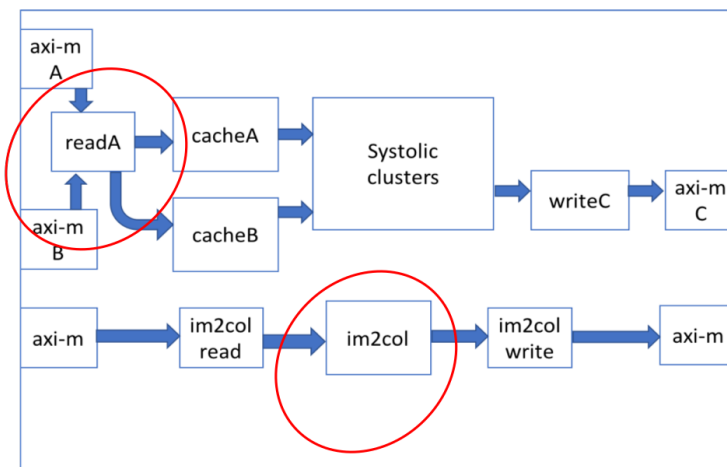


Figure 4.5: Block diagram of revised HLS design

data from the bus. The systolic array needs data from port B in order to consume data from both ports together, but then “readB” module would be blocked to wait for the AXI bus, hence forming a deadlock.

4.5.2 Revised Design and Results

To solve the deadlock problem, we revised the design to instead use a single module for reading both the matrix A and matrix B, so that the read requests on the AXI interconnect would appear in specific orders instead of random orders. Another change is that, the read requests are now guaranteed to have enough local buffering space to finish one transfer, so that a back-pressure from the systolic array does not affect the outstanding AXI bus read. Also,

Table 4.5: Throughput in the MMULT part of CONV for the revised design.

	Input				Stride	Pad	Kernel			#Operations	Cache		Time(ms)	GOPS
	N	C	H	W			C	H	W		A	B		
Lenet-conv1	100	1	28	28	1	0	20	5	5	57,600,000	y	y	0.679	84.8
Lenet-conv2	100	20	12	12	1	0	50	5	5	320,000,000	y	y	1.093	292.7
Cifar10-conv1	100	3	32	32	1	2	32	5	5	491,520,000	y	y	2.606	188.6
Cifar10-conv2	100	32	16	16	1	2	32	5	5	1,310,720,000	y	y	4.933	265.7
Cifar10-conv3	100	32	8	8	1	2	64	5	5	655,360,000	y	y	3.817	171.7
Caffenet-conv1	50	3	227	227	4	0	96	11	11	10,541,520,000	y	y	26.830	392.9
Caffenet-conv2	50	96	27	27	1	2	256	5	5	44,789,760,000	n	y	177.370	252.5
Caffenet-conv3	50	256	13	13	1	1	384	3	3	14,952,038,400	n	y	63.945	233.8
Caffenet-conv4	50	384	13	13	1	1	384	3	3	22,428,057,600	n	y	95.271	235.4

the “im2col” module now exist on a separate datapath. The revised design is shown in figure 4.5.

With this revised design, the matrix multiplication(MMULT) part of the convolution(CONV) can now complete under all settings. Their results are shown in table 4.5. From the table we can see that generally, the higher is the number of operations needed in a layer, the higher is the achieved throughput, with a maximum throughput of 392.9 GOPS achieved in “conv1” layer of Caffenet, a variant of AlexNet from the Caffe package.

Table 4.6 shows the speed-up of the FPGA implementation against an optimized CPU implementation, the “SGEMM” function from the Intel Math Kernel Library(MKL).

Compared with table 4.5, although the highest throughput is achieved in “Caffenet-conv1”, the largest speed-up actually happens in “Lenet-conv1”, with a maximum speed-up of $32.7\times$. This can be attributed to that the CPU has different performance characteristics among different layers compared to that of the FPGA, due to difference in cache handling, memory alignment, etc. A minimum speed-up of $1.8\times$ is achieved for the “Caffenet-conv2” layer.

4.6 Conclusion

In this chapter, we have proposed the hDNN platform, which is a collection of a modified Caffe software, and hardware designs for the SDAccel platform, that can serve as building blocks for DNN implementations, or as baseline

Table 4.6: Speedup of FPGA against naive CPU MMULT implementation.

	CPU Time(ms)	FPGA Time(ms)	Speedup
Lenet-conv1	22.188	0.679	32.7
Lenet-conv2	23.956	1.093	21.9
Cifar10-conv1	29.944	2.606	11.5
Cifar10-conv2	44.739	4.933	9.1
Cifar10-conv3	39.521	3.817	10.4
Caffenet-conv1	121.810	26.830	4.5
Caffenet-conv2	323.488	177.370	1.8
Caffenet-conv3	160.814	63.945	2.5
Caffenet-conv4	197.351	95.271	2.1

comparison target for further researches, which in today’s research community is often lacking.

The modified Caffe is capable of supporting quantized arithmetic in the user’s layer implementation code, intend to be run on any CPUs, GPUs or FPGAs, such that quantized and non-quantized layers can work together interchangeably. The provided interface also allows user-defined quantization scheme in individual layers.

The provided hardware core includes a systolic array, configurable as 32×32 for 16-bit fixed-point/integer to run at 200 MHz, achieving a theoretical throughput of 409.6 GOPS. Caching and “Im2col” modules are also provided so that the systolic array can be used for convolution operations. We proposed a design that resolves the deadlock situation that can occur on the Alpha Data 7v3 platform, and demonstrated speed-up from $1.8\times$ to $32.7\times$, for the matrix multiplication part of convolution layers in networks like Lenet, Cifar10 and CaffeNet, against an optimized CPU implementation from the Intel MKL library.

This page is intentionally left blank.

Chapter 5

Conclusion and Future Work

In chapter 2 we studied the case of training an SVM cell image classifier on FPGA-assisted Spark cluster. We have shown that a speed-up of up to $1.6\times$ over the host CPU cluster can be achieved, but that requires user's special care on ensuring data reuse on the FPGA boards between iterations of the algorithm.

In chapter 3, we presented NnCore, an open-source, parameterizable non-linear function generator for FPGA based machine learning applications, to facilitate design reuse among the research community. NnCore constructs floating-point operators using piecewise fixed-point minimax-polynomials, and the generated operators are faithfully rounded by construction.

Experiments show that NnCore generated functions, in HLS C++ format, uses a comparable number of slices, LUTs and flip-flops, fewer number of BRAMs, and runs at much higher clock rate, compare to cores from a previous generic function generator, while showing run-time stability throughout generation of all functions, which is missing in the previous generic generator in comparison.

In chapter 4, we proposed the hDNN platform, which is a collection of a modified Caffe software, and hardware designs for the SDAccel platform, that can serve as building blocks for DNN implementations, or as baseline comparison target for further research. The modified Caffe enables quantized

arithmetic in user layers, for running on any of CPUs/GPUs/FPGAs. The provided hardware core includes a systolic array, that can run 200 MHz for 16-bit fixed-point/integer, at 32×32 setup, achieving 409.6 GOPS theoretical throughput. Together with the provided “Im2col” core, convolution operations can also be run on the systolic array. A speed-up range of $1.8\times$ to $32.7\times$ is achieved against an optimized CPU implementation from the Intel MKL library, on the matrix multiplication part of convolution layers in networks like Lenet, Cifar10, CaffeNet, etc.

5.1 Future Work

There are interesting possible directions, relating to the Spark work in chapter 2 and hDNN in chapter 4, like deploying the FPGA-integrated Caffe on a Spark cluster. Similar works have been done and deployed in real-world application environment, for the case of GPUs [72, 7, 6].

The NnCore generator in chapter 3 is an activation-function IP core generator that supports arbitrary floating-point bitwidths, while the hDNN platform in chapter 4 is a platform for deep learning acceleration, currently supporting fixed-point/integer arithmetics. Now that we have these two resources ready, it would be interesting to try to integrate NnCore generated operators in hDNN.

On one hand, we can vary the bitwidths of the activation functions to study how they affect the whole-network inference/training accuracy. On the other hand, we can vary the supported numeric precisions in hDNN, to study how we can maximize runtime throughput, minimize resource usage and yet maintain acceptable accuracy.

Compared to hDNN, there is a trend in the latest commodity hardware platforms to include a “graph compiler”, where the high-level network topologies are compiled into instructions to be executed on the hardware. Examples of these platforms include the Google TPU [3], Xilinx xfDNN, Intel DLA [73], etc.

These compilers provide optimizations like layer-fusing, numeric quantizations, and some also handles scheduling of data transfer and operations. However, currently there exist no such alternative in the open-source community, to the best of our knowledge. Part of the reason for this is that such compilers are closely coupled with a corresponding hardware execution platform, and yet there are almost no open-source neural network hardware platforms. Now with the introduction of hDNN, we can try to build an open-source alternative for such compilers, and introduce new innovative techniques to optimize network executions.

This page is intentionally left blank.

Bibliography

- [1] M. Shaaban, “RIT CMPE655, lecture notes: Introduction to parallel processing,” 2015, URL: <http://meseec.ce.rit.edu/cmpe655-fall2015/655-8-25-2015.ppt>. Last visited on 2017/09/28.
- [2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. Vazir Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellman, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-Datacenter Performance Analysis of a Tensor Processing Unit,” *ArXiv e-prints*, Apr. 2017.
- [3] J. Dean and U. Holzle. Build and train machine learning models on our new Google Cloud TPUs. [Online]. Available: <https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning>
- [4] A. Putnam, A. Caulfield, E. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, J. Gray, M. Haselman, S. Hauck,

- S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, E. Peterson, A. Smith, J. Thong, P. Y. Xiao, D. Burger, J. Larus, G. P. Gopal, and S. Pope, “A reconfigurable fabric for accelerating large-scale datacenter services,” in *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, June 2014, pp. 13–24. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/a-reconfigurable-fabric-for-accelerating-large-scale-datacenter-services/>
- [5] D. Burger, “Microsoft unveils project Brainwave for real-time AI,” <https://www.microsoft.com/en-us/research/blog/microsoft-unveils-project-brainwave/>, 2017.
- [6] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan, “Sparknet: Training deep networks in spark,” *arXiv preprint arXiv:1511.06051*, 2015.
- [7] H. Kim, J. Park, J. Jang, and S. Yoon, “DeepSpark: Spark-based deep learning supporting asynchronous updates and Caffe compatibility,” *CoRR*, vol. abs/1602.08191, 2016. [Online]. Available: <http://arxiv.org/abs/1602.08191>
- [8] F. de Dinechin and B. Pasca, “Designing custom arithmetic data paths with FloPoCo,” *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.
- [9] X. Fang and M. Leeser, “Open-source variable-precision floating-point library for major commercial fpgas,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 9, no. 3, pp. 20:1–20:17, Jul. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2851507>
- [10] D. B. Thomas, “A general-purpose method for faithfully rounded floating-point function approximation in FPGAs,” in *2015 IEEE 22nd Symposium on Computer Arithmetic*, June 2015, pp. 42–49.

- [11] S. M. H. Ho, M. Wang, H. C. Ng, and H. K. H. So, “Towards FPGA-assisted Spark: An SVM training acceleration case study,” in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1–6.
- [12] J. H. C. Yeung, C. C. Tsang, K. H. Tsoi, B. S. H. Kwan, C. C. C. Cheung, A. P. C. Chan, and P. H. W. Leong, “Map-reduce as a programming model for custom computing machines,” in *Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th International Symposium on*, April 2008, pp. 149–159.
- [13] Y. Shan, B. Wang, J. Yan, Y. Wang, N. Xu, and H. Yang, “FPMR: Mapreduce framework on FPGA,” in *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '10. New York, NY, USA: ACM, 2010, pp. 93–102. [Online]. Available: <http://doi.acm.org/10.1145/1723112.1723129>
- [14] Y. M. Choi and H. K. H. So, “Map-reduce processing of k-means algorithm with FPGA-accelerated computer cluster,” in *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, June 2014, pp. 9–16.
- [15] K. Fleming, H. J. Yang, M. Adler, and J. Emer, “The LEAP FPGA operating system,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, Sept 2014, pp. 1–8.
- [16] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, “Lime: A java-compatible and synthesizable language for heterogeneous architectures,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 89–108. [Online]. Available: <http://doi.acm.org/10.1145/1869459.1869469>

- [17] J. Xie, X. Niu, A. Lau, K. Tsia, and H. So, “Accelerated cell imaging and classification on FPGAs for quantitative-phase asymmetric-detection time-stretch optical microscopy,” in *Field-Programmable Technology, 2015. FPT 2015. International Conference on*, 2015, pp. 388--391.
- [18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15--28. [Online]. Available: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia>
- [19] S. Shalev-Shwartz, Y. Singer, and N. Srebro, “Pegasos: Primal estimated sub-gradient solver for SVM,” in *Proceedings of the 24th International Conference on Machine Learning*, ser. ICML '07. New York, NY, USA: ACM, 2007, pp. 807--814. [Online]. Available: <http://doi.acm.org/10.1145/1273496.1273598>
- [20] C.-J. Hsieh, K.-W. Chang, C.-J. Lin, S. S. Keerthi, and S. Sundararajan, “A dual coordinate descent method for large-scale linear SVM,” in *Proceedings of the 25th International Conference on Machine Learning*, ser. ICML '08. New York, NY, USA: ACM, 2008, pp. 408--415. [Online]. Available: <http://doi.acm.org/10.1145/1390156.1390208>
- [21] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, “RIFFA 2.1: A reusable integration framework for FPGA accelerators,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 4, pp. 22:1--22:23, Sep. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2815631>
- [22] T. T. Wong, A. K. Lau, K. K. Ho, M. Y. Tang, J. D. Robles, X. Wei, A. C. Chan, A. H. Tang, E. Y. Lam, K. K. Wong *et al.*, “Asymmetric-detection

- time-stretch optical microscopy (ATOM) for ultrafast high-contrast cellular imaging in flow,” *Scientific reports*, vol. 4, 2014.
- [23] Xilinx. (2017, Apr.) Vivado design suite user guide ug902 v2017.1. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2017_1/ug902-vivado-high-level-synthesis.pdf
- [24] ----- . (2016, Nov.) Vivado design suite user guide ug902 v2016.4. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug902-vivado-high-level-synthesis.pdf
- [25] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun, “CNP: An FPGA-based processor for convolutional networks,” in *2009 International Conference on Field Programmable Logic and Applications*, Aug 2009, pp. 32--37.
- [26] O. Temam, “A defect-tolerant accelerator for emerging high-performance applications,” in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, June 2012, pp. 356--367.
- [27] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, “A 240 G-ops/s mobile coprocessor for deep neural networks,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, June 2014.
- [28] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. 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*, ser. ASPLOS ’14. New York, NY, USA: ACM, 2014, pp. 269--284. [Online]. Available: <http://doi.acm.org/10.1145/2541940.2541967>
- [29] J. C. Ferreira and J. Fonseca, “An FPGA implementation of a long short-term memory neural network,” in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Nov 2016, pp. 1--8.

- [30] H. Faiedh, Z. Gafsi, and K. Besbes, “Digital hardware implementation of sigmoid function and its derivative for artificial neural networks,” in *ICM 2001 Proceedings. The 13th International Conference on Microelectronics*, Oct 2001, pp. 189--192.
- [31] K. Basterretxea, J. M. Tarela, and I. del Campo, “Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons,” *IEE Proceedings - Circuits, Devices and Systems*, vol. 151, no. 1, pp. 18--24, Feb 2004.
- [32] C.-W. Lin and J.-S. Wang, “A digital circuit design of hyperbolic tangent sigmoid function for neural networks,” in *2008 IEEE International Symposium on Circuits and Systems*, May 2008, pp. 856--859.
- [33] B. Zamanlooy and M. Mirhassani, “Efficient VLSI implementation of neural networks with hyperbolic tangent activation function,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 1, pp. 39--48, Jan 2014.
- [34] A. M. Abdelsalam, J. M. P. Langlois, and F. Cheriet, “A configurable FPGA implementation of the tanh function using DCT interpolation,” in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, April 2017, pp. 168--171.
- [35] D. U. Lee, R. C. C. Cheung, W. Luk, and J. D. Villasenor, “Hierarchical segmentation for hardware function evaluation,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 1, pp. 103--116, Jan 2009.
- [36] L. Ge, S. Chen, Y. Nakamura, and T. Yoshimura, “A synthesis method of general floating-point arithmetic units by aligned partition,” *IPSJ Transactions on System LSI Design Methodology*, vol. 1, pp. 67--77, 2008.

- [37] J. Muller, *Elementary Functions: Algorithms and Implementation*, ser. Computer Science. Birkhauser Boston, 2006. [Online]. Available: http://books.google.com/books?id=Mx-_kaANJBEC
- [38] P. T. P. Tang, “Table-driven implementation of the expm1 function in IEEE floating-point arithmetic,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 18, no. 2, pp. 211–222, 1992.
- [39] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” *ArXiv e-prints*, Mar. 2016.
- [40] R. Collobert, K. Kavukcuoglu, and C. Farabet, “Torch7: A matlab-like environment for machine learning,” in *BigLearn, NIPS Workshop*, no. EPFL-CONF-192376, 2011.
- [41] I. J. Goodfellow, D. Warde-Farley, M. Mirza, A. Courville, and Y. Bengio, “Maxout Networks,” *ArXiv e-prints*, Feb. 2013.
- [42] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, “Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs),” *ArXiv e-prints*, Nov. 2015.
- [43] S. Chevillard, M. Joldes, and C. Lauter, “Sollya: An environment for the development of numerical codes,” in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, K. Fukuda, J. van der Hoeven,

- M. Joswig, and N. Takayama, Eds., vol. 6327. Heidelberg, Germany: Springer, September 2010, pp. 28--31.
- [44] S. M. Ho, C.-H. D. Hung, H.-C. Ng, M. Wang, and H. K.-H. So, "A parameterizable activation function generator for FPGA-based neural network applications," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 84--84.
- [45] FloPoCo repository. [Online]. Available: <https://scm.gforge.inria.fr/anonscm/git/flopoco/flopoco.git>
- [46] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.
- [47] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265--283. [Online]. Available: <https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf>
- [48] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: <http://arxiv.org/abs/1509.09308>
- [49] P. Gysel, M. Motamedi, and S. Ghiasi. Ristretto github repo. [Online]. Available: <https://github.com/pmgysel/caffe>
- [50] -----, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.

- [51] gemmlowp: a small self-contained low-precision gemm library. [Online]. Available: <https://github.com/google/gemmlowp>
- [52] S. Han, H. Mao, and W. J. Dally, “Deep compression: Compressing deep neural network with pruning, trained quantization and Huffman coding,” *CoRR*, vol. abs/1510.00149, 2015. [Online]. Available: <http://arxiv.org/abs/1510.00149>
- [53] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: efficient inference engine on compressed deep neural network,” in *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 2016, pp. 243–254.
- [54] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, “ESE: Efficient speech recognition engine with sparse LSTM on FPGA.” in *FPGA*, 2017, pp. 75–84.
- [55] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, “Deep learning with limited numerical precision,” *CoRR*, vol. abs/1502.02551, 2015. [Online]. Available: <http://arxiv.org/abs/1502.02551>
- [56] Xilinx SDAccel forum. [Online]. Available: <https://forums.xilinx.com/xlnx/board/message?board.id=SDx&message.id=1021#M1021>
- [57] Xilinx SDAccel examples Github repository’s systolic array binary. [Online]. Available: https://github.com/Xilinx/SDAccel_Examples/tree/master/acceleration/high_perf_mat_mult
- [58] J. Ross and G. Thorson, “Rotating data for neural network computations,” Nov. 24 2016, US Patent App. 14/845,022. [Online]. Available: <http://google.com/patents/US20160342893>
- [59] M. Feldman, “Microsoft takes FPGA-powered deep learning to the next level,” <https://www.top500.org/news/microsoft-takes-fpga-powered-deep-learning-to-the-next-level/>, 2017.

- [60] T. P. Morgan, “Drilling into Microsofts BrainWave soft deep learning chip,” <https://www.nextplatform.com/2017/08/24/drilling-microsofts-brainwave-soft-deep-leaning-chip/>, 2017.
- [61] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J.-s. Seo, and Y. Cao, “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 16–25.
- [62] J. Zhang and J. Li, “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network.” in *FPGA*, 2017, pp. 25–34.
- [63] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing fpga-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2015, pp. 161–170.
- [64] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, “Going deeper with embedded FPGA platform for convolutional neural network,” in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2016, pp. 26–35.
- [65] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, “A high performance FPGA-based accelerator for large-scale convolutional neural networks,” in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, Aug 2016, pp. 1–9.
- [66] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. 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*. ACM, 2017, pp. 45–54.

- [67] C. Zhang and V. K. Prasanna, “Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system.” in *FPGA*, 2017, pp. 35–44.
- [68] U. Aydonat, S. O’Connell, D. Capalija, A. C. Ling, and G. R. Chiu, “An OpenCLTM deep learning accelerator on Arria 10,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 55–64.
- [69] L. Lu, Y. Liang, Q. Xiao, and S. Yan, “Evaluating fast algorithms for convolutional neural networks on FPGAs,” in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 101–108.
- [70] R. DiCecco, G. Lacey, J. Vasiljevic, P. Chow, G. W. Taylor, and S. Areibi, “Caffeinated FPGAs: FPGA framework for convolutional neural networks,” *CoRR*, vol. abs/1609.09671, 2016. [Online]. Available: <http://arxiv.org/abs/1609.09671>
- [71] OpenCL Caffe GitHub repository. [Online]. Available: <https://github.com/BVLC/caffe/tree/opencv>
- [72] CaffeOnSpark by Yahoo, on github. [Online]. Available: <https://github.com/yahoo/CaffeOnSpark>
- [73] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. OConnell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu, “DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration,” *ArXiv e-prints*, Jul. 2018.