Low-precision CNN Inference and Training for Edge Applications on FPGAs

Philip Leong | Computer Engineering Laboratory
School of Electrical and Information Engineering,
The University of Sydney

THE UNIVERSITY OF
SYDNEY

THE UNIVERSITY OF SYDNEY

Computer Engineering Laboratory

- Focuses on how to use parallelism to solve demanding problems
  - Novel architectures, applications and design techniques using VLSI, FPGA and parallel computing technology
- Research
  - Reconfigurable computing
  - Machine learning
  - Signal processing
- Collaborations
  - Xilinx, Exablaze (now Cisco)
  - Defence and DSTG
  - clustertech.com

Overview

- Implementation of *small* CNNs
  - Unrolling Ternary Networks
  - Training deep neural networks in low-precision with high accuracy using FPGAs

Unrolling Ternary Networks

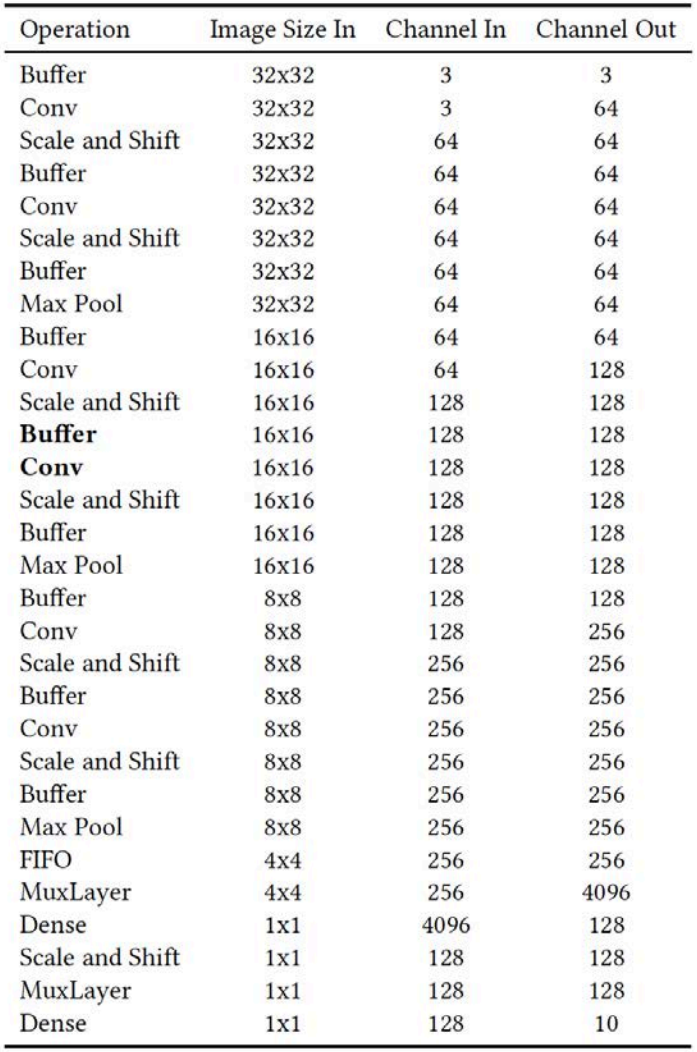
Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland, Duncan Moss, Peter Zipf, Philip H.W. Leong

Introduction

- **A fully pipelined DNN** implementation with ternary coefficients
- Difficult to make fully parallel implementations of a NN on contemporary FPGA due to size
- Fit entire DNN on FPGA by exploiting unstructured sparsity and the following techniques:
  1. Buffering of streaming inputs in a pipelined manner
  2. Ternary weights implemented as pruned adder trees
  3. Common subexpression merging
  4. 16-bit bit serial arithmetic to minimize accuracy loss with low area
  5. Sparsity control

Network Studied

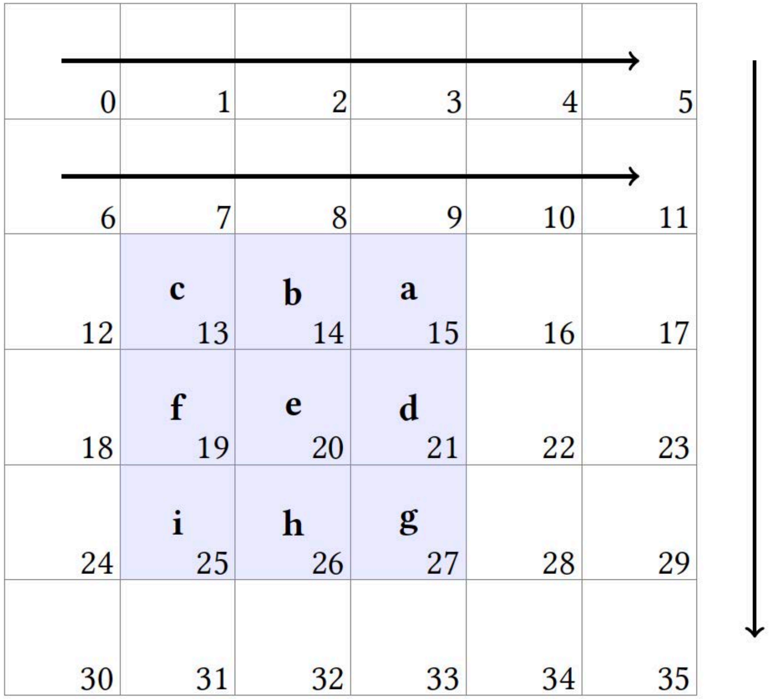
- VGG-7 network
- Ternary weights
- 16-bit activations
- Accept a single pixel every cycle (p=1)
  - W*W image takes W*W cycles



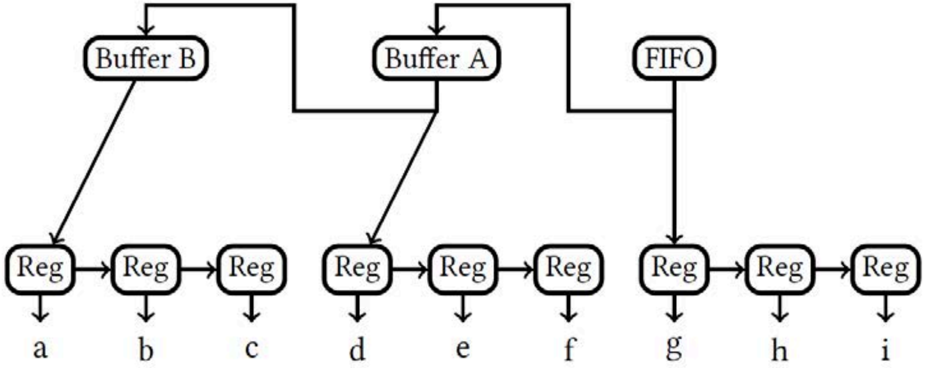
Operation	Image Size In	Channel In	Channel Out
Buffer	32x32	3	3
Conv	32x32	3	64
Scale and Shift	32x32	64	64
Buffer	32x32	64	64
Conv	32x32	64	64
Scale and Shift	32x32	64	64
Buffer	32x32	64	64
Max Pool	32x32	64	64
Buffer	16x16	64	64
Conv	16x16	64	128
Scale and Shift	16x16	128	128
Buffer	16x16	128	128
Conv	16x16	128	128
Scale and Shift	16x16	128	128
Buffer	16x16	128	128
Max Pool	16x16	128	128
Buffer	8x8	128	128
Conv	8x8	128	256
Scale and Shift	8x8	256	256
Buffer	8x8	256	256
Conv	8x8	256	256
Scale and Shift	8x8	256	256
Buffer	8x8	256	256
Max Pool	8x8	256	256
FIFO	4x4	256	256
MuxLayer	4x4	256	4096
Dense	1x1	4096	128
Scale and Shift	1x1	128	128
MuxLayer	1x1	128	128
Dense	1x1	128	10

Buffering of Streaming Inputs

Implement Pipelined 3x3 Convolution



0	1	2	3	4	5
6	7	8	9	10	11
12	c 13	b 14	a 15	16	17
18	f 19	e 20	d 21	22	23
24	i 25	h 26	g 27	28	29
30	31	32	33	34	35



Input FIFO outputs the
pixel each cycle to both Buffer A and the
first stage of a shift register.
Buffer A and Buffer B delay the output by
the image width

Ternary Weights as Pruned Adder Trees

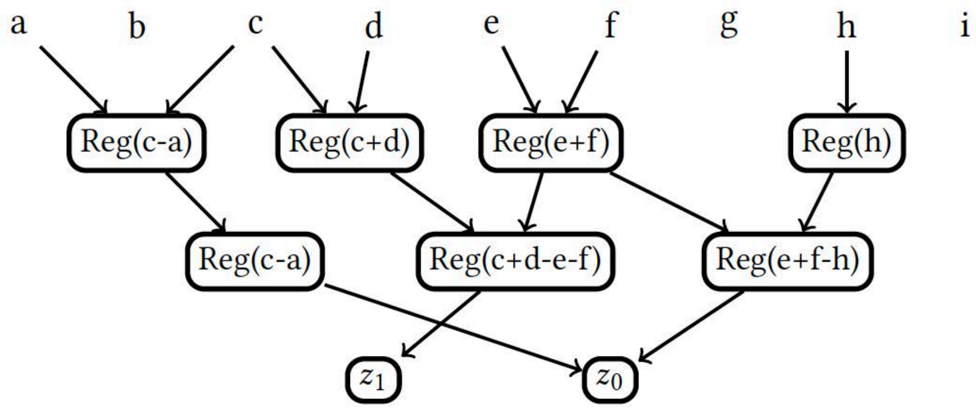
- Weights are ternary
  - So multiplication with ± 1 is either addition or subtraction
  - Multiplication with 0 makes matrix sparse

$$\begin{array}{ccc}
a \times (-1) & b \times 0 & c \times 1 \\
d \times 0 & e \times 1 & f \times 1 \\
g \times 0 & h \times (-1) & i \times 0
\end{array}$$

Common Subexpression Elimination (CSE)

- Weights are ternary
  - Reduces convolution to constructing adder tree
  - Subexpression merged to reduce implementation

$$\begin{array}{ccc} a \times (-1) & b \times 0 & c \times 1 \\ d \times 0 & e \times 1 & f \times 1 \\ g \times 0 & h \times (-1) & i \times 0 \end{array}$$



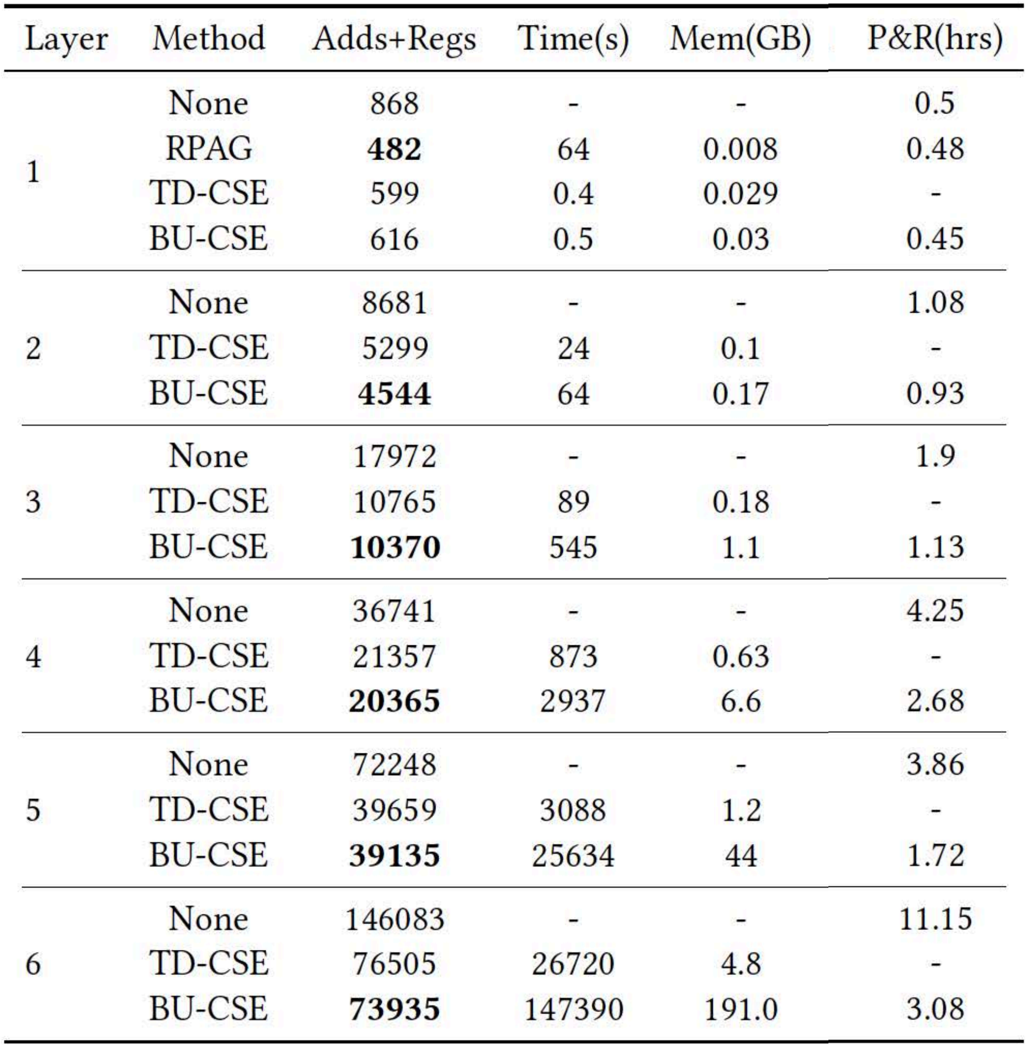
Computing $z_0 = c + e + f - (a + h)$ and $z_1 = c + d - e - f$

CSE Techniques

- None – doesn’t fit
- Reduced Pipelined Adder Graph (RPAG) – too slow
- Top down CSE (TD-CSE) – doesn’t find good solutions
- Bottom up CSE (BU-CSE) – worst of both worlds, runs out of memory

Comparison of CSE Techniques for all Layers

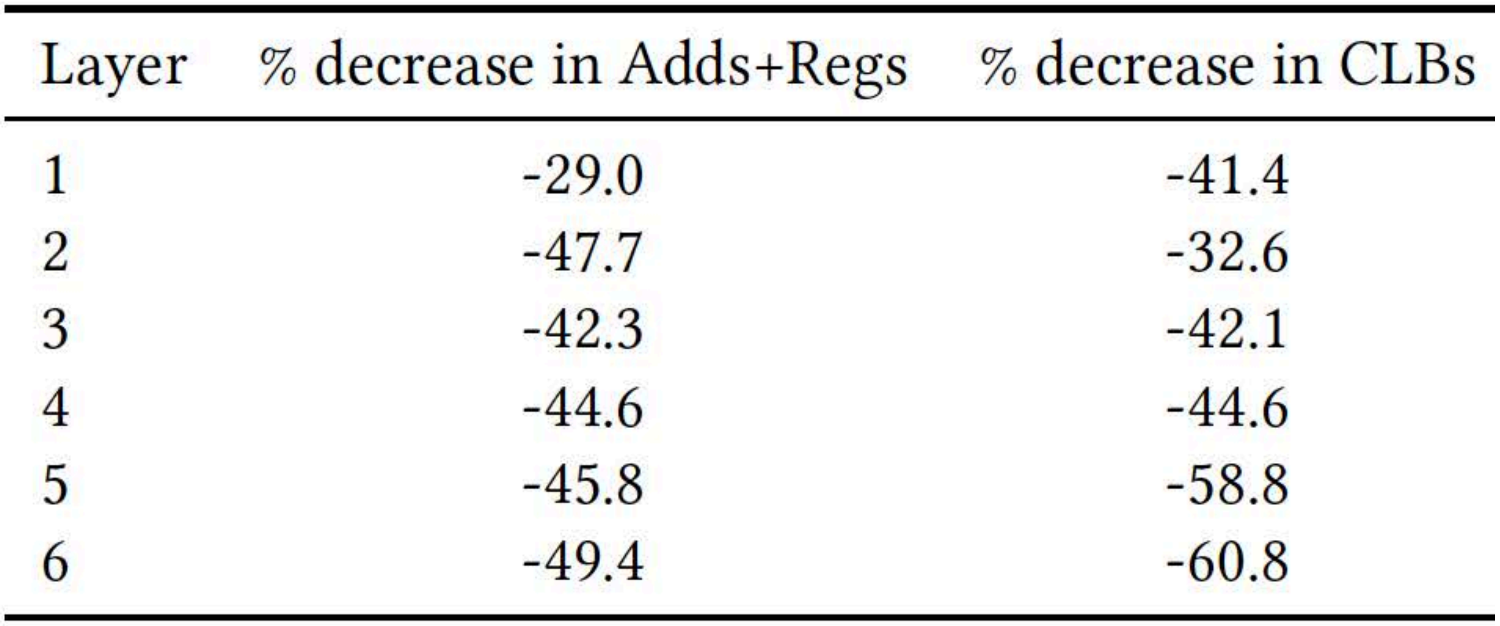
- RPAG too computationally expensive for layers 2-6
- BU-CSE a bit better than TD-CSE



Layer	Method	Adds+Regs	Time(s)	Mem(GB)	P&R(hrs)
1	None	868	-	-	0.5
	RPAG	**482**	64	0.008	0.48
	TD-CSE	599	0.4	0.029	-
	BU-CSE	616	0.5	0.03	0.45
2	None	8681	-	-	1.08
	TD-CSE	5299	24	0.1	-
	BU-CSE	**4544**	64	0.17	0.93
3	None	17972	-	-	1.9
	TD-CSE	10765	89	0.18	-
	BU-CSE	**10370**	545	1.1	1.13
4	None	36741	-	-	4.25
	TD-CSE	21357	873	0.63	-
	BU-CSE	**20365**	2937	6.6	2.68
5	None	72248	-	-	3.86
	TD-CSE	39659	3088	1.2	-
	BU-CSE	**39135**	25634	44	1.72
6	None	146083	-	-	11.15
	TD-CSE	76505	26720	4.8	-
	BU-CSE	**73935**	147390	191.0	3.08

THE UNIVERSITY OF SYDNEY

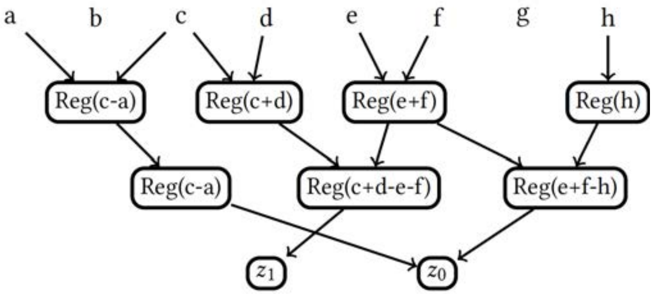
Improvement in using CSE

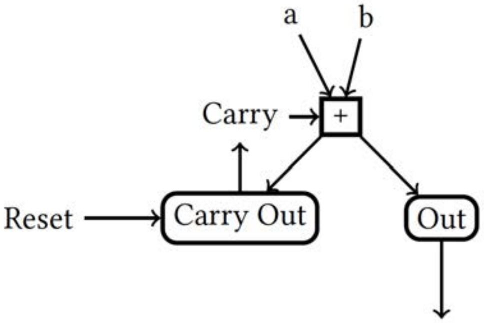


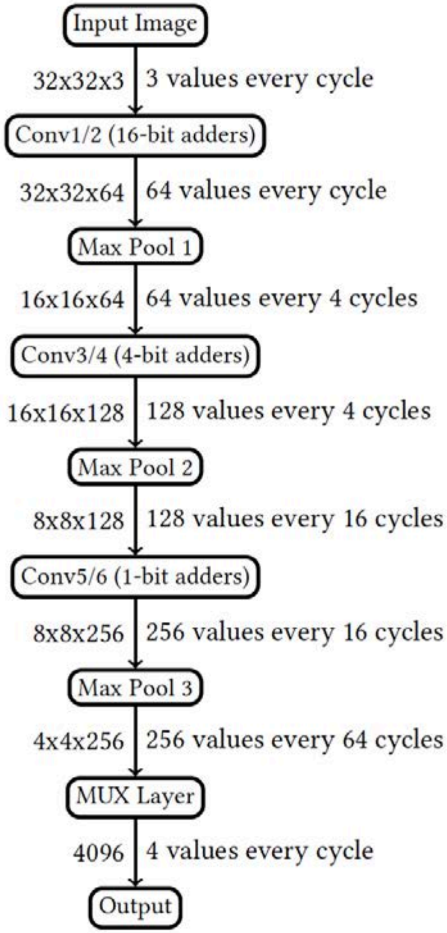
Layer	% decrease in Adds+Regs	% decrease in CLBs
1	-29.0	-41.4
2	-47.7	-32.6
3	-42.3	-42.1
4	-44.6	-44.6
5	-45.8	-58.8
6	-49.4	-60.8

Digit Serial Arithmetic

- Used 16-bit fixed point
- Each layer followed by batch normalization with floating point scaling factor
- Suppose that for a given layer, p pixels arrive at the same time
  - For p ≥ 1 have p adder trees in parallel
  - For p < 1 word or bit-serial adders can match input rate with hardware resources
  - 4-bit digit serial has 1/4 area
  - 1-bit bit serial has 1/16 area
- Avoids idle adders

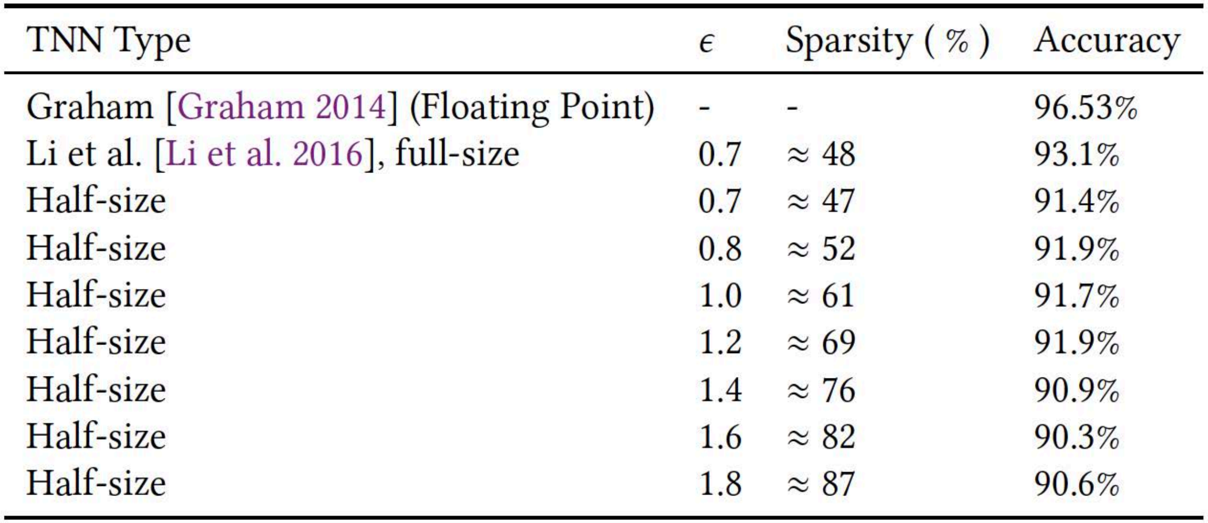






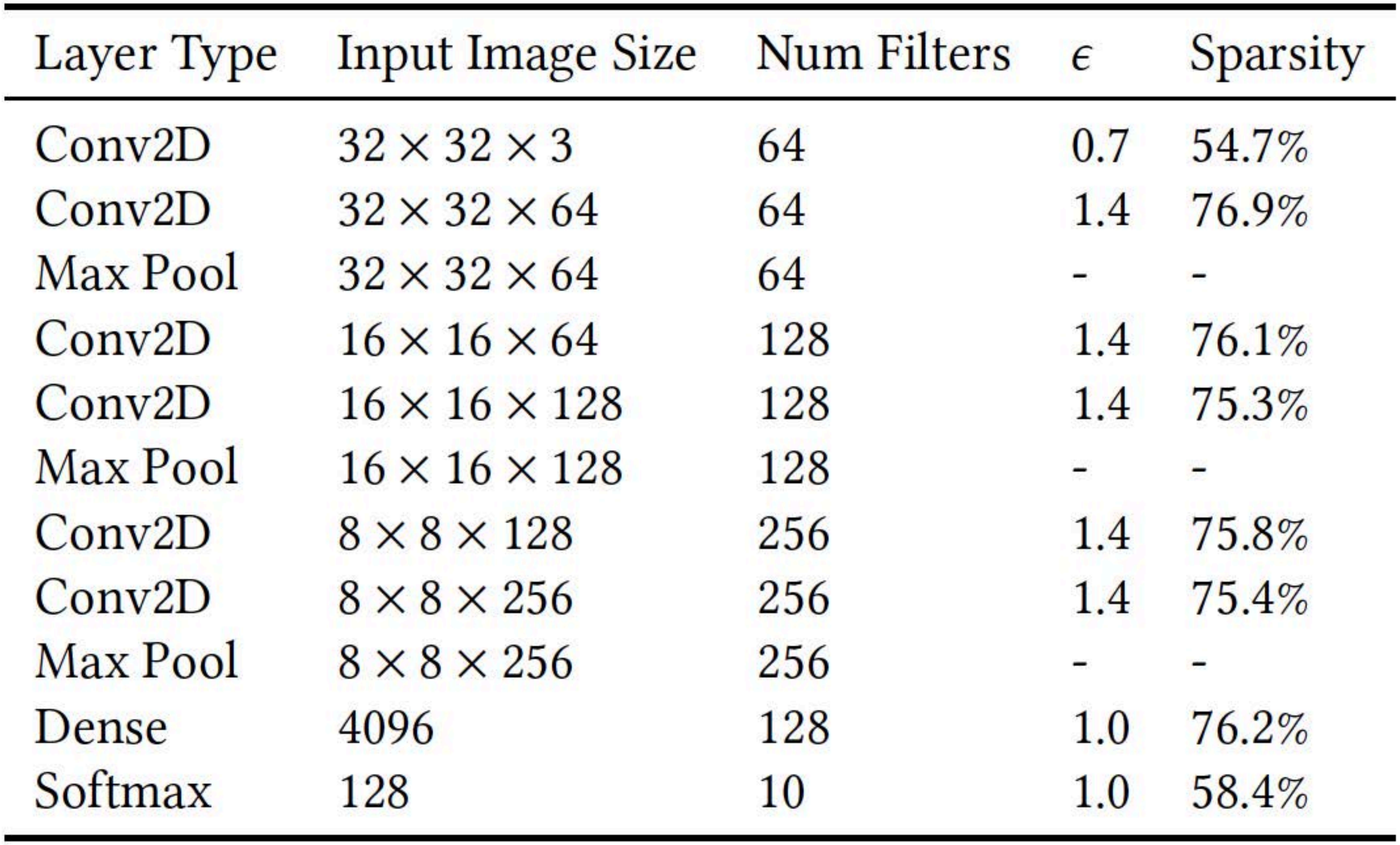
Sparsity Control

- CIFAR10 dataset
- Image padded with 4 pixels each side and randomly cropped back to 32x32
- Weights are compared with threshold $\Delta^* \approx \epsilon \cdot E(|W|)$
  - 0 if less than threshold, $s(\pm 1)$ otherwise (s is a scaling factor)
- We introduce the idea of changing ϵ to control sparsity



TNN Type	ϵ	Sparsity (%)	Accuracy
Graham [Graham 2014] (Floating Point)	-	-	96.53%
Li et al. [Li et al. 2016], full-size	0.7	$\approx$ 48	93.1%
Half-size	0.7	$\approx$ 47	91.4%
Half-size	0.8	$\approx$ 52	91.9%
Half-size	1.0	$\approx$ 61	91.7%
Half-size	1.2	$\approx$ 69	91.9%
Half-size	1.4	$\approx$ 76	90.9%
Half-size	1.6	$\approx$ 82	90.3%
Half-size	1.8	$\approx$ 87	90.6%

Breakdown of Layer Sparsity

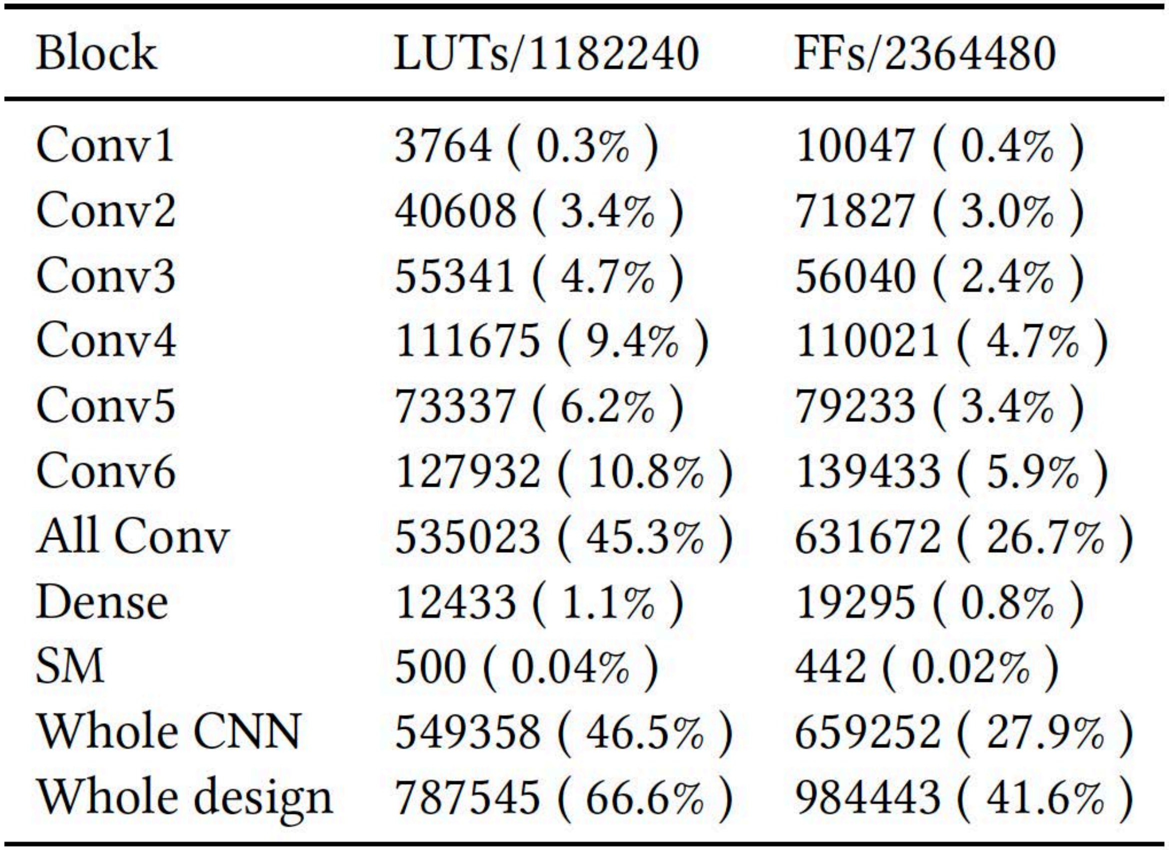


Layer Type	Input Image Size	Num Filters	ϵ	Sparsity
Conv2D	$32 \times 32 \times 3$	64	0.7	54.7%
Conv2D	$32 \times 32 \times 64$	64	1.4	76.9%
Max Pool	$32 \times 32 \times 64$	64	-	-
Conv2D	$16 \times 16 \times 64$	128	1.4	76.1%
Conv2D	$16 \times 16 \times 128$	128	1.4	75.3%
Max Pool	$16 \times 16 \times 128$	128	-	-
Conv2D	$8 \times 8 \times 128$	256	1.4	75.8%
Conv2D	$8 \times 8 \times 256$	256	1.4	75.4%
Max Pool	$8 \times 8 \times 256$	256	-	-
Dense	4096	128	1.0	76.2%
Softmax	128	10	1.0	58.4%

Implementation

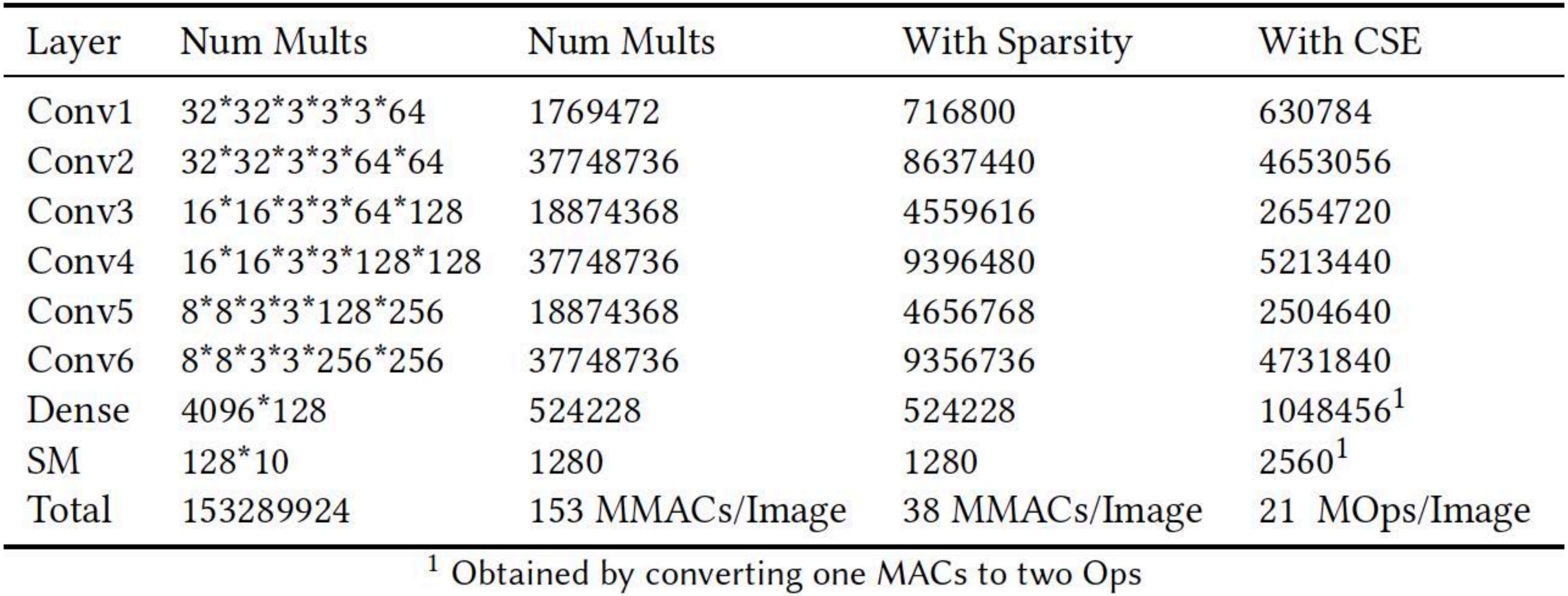
- System implemented on Ultrascale+ VU9P @ 125 MHz
- Open Source Verilog generator
  - https://github.com/da-steve101/binary_connect_cifar
- Generated code using in AWS F1 implementation
  - https://github.com/da-steve101/aws-fpga

Area Breakdown



Block	LUTs/1182240	FFs/2364480
Conv1	3764 (0.3%)	10047 (0.4%)
Conv2	40608 (3.4%)	71827 (3.0%)
Conv3	55341 (4.7%)	56040 (2.4%)
Conv4	111675 (9.4%)	110021 (4.7%)
Conv5	73337 (6.2%)	79233 (3.4%)
Conv6	127932 (10.8%)	139433 (5.9%)
All Conv	535023 (45.3%)	631672 (26.7%)
Dense	12433 (1.1%)	19295 (0.8%)
SM	500 (0.04%)	442 (0.02%)
Whole CNN	549358 (46.5%)	659252 (27.9%)
Whole design	787545 (66.6%)	984443 (41.6%)

Summary of Sparsity and CSE Improvement

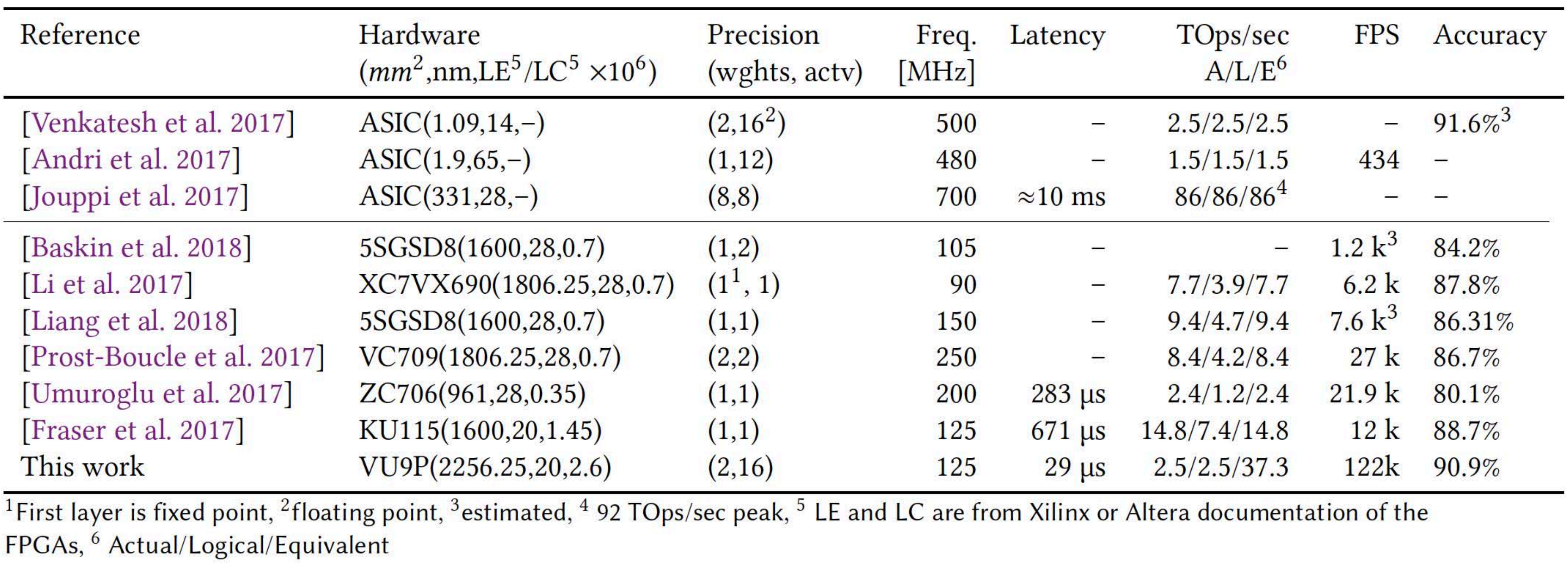


Layer	Num Mults	Num Mults	With Sparsity	With CSE
Conv1	32*32*3*3*3*64	1769472	716800	630784
Conv2	32*32*3*3*64*64	37748736	8637440	4653056
Conv3	16*16*3*3*64*128	18874368	4559616	2654720
Conv4	16*16*3*3*128*128	37748736	9396480	5213440
Conv5	8*8*3*3*128*256	18874368	4656768	2504640
Conv6	8*8*3*3*256*256	37748736	9356736	4731840
Dense	4096*128	524228	524228	1048456[1]
SM	128*10	1280	1280	2560[1]
Total	153289924	153 MMACs/Image	38 MMACs/Image	21 MOps/Image

[1] Obtained by converting one MACs to two Ops

Accuracy

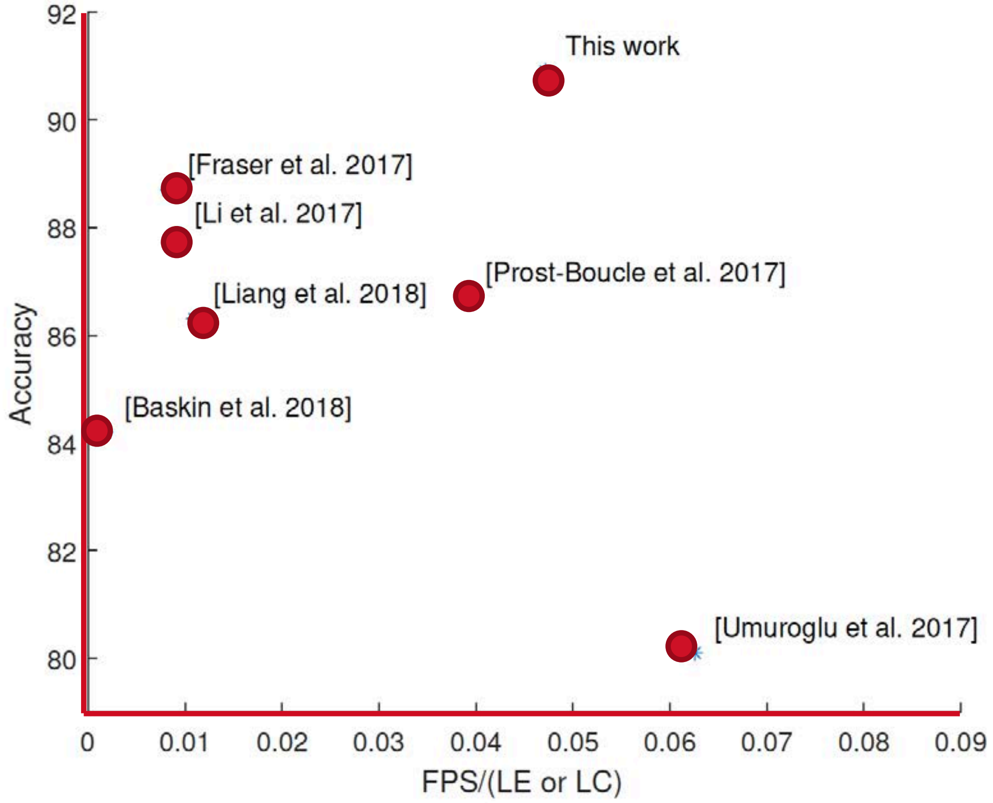
Comparison with ASIC and FPGA implementations (CIFAR10)



Reference	Hardware (mm^2,nm,LE[5]/LC[5] ×10^6)	Precision (wghts, actv)	Freq. [MHz]	Latency	TOps/sec A/L/E[6]	FPS	Accuracy
[Venkatesh et al. 2017]	ASIC(1.09,14,–)	(2,16[2])	500	–	2.5/2.5/2.5	–	91.6%[3]
[Andri et al. 2017]	ASIC(1.9,65,–)	(1,12)	480	–	1.5/1.5/1.5	434	–
[Jouppi et al. 2017]	ASIC(331,28,–)	(8,8)	700	≈10 ms	86/86/86[4]	–	–
[Baskin et al. 2018]	5SGSD8(1600,28,0.7)	(1,2)	105	–	–	1.2 k[3]	84.2%
[Li et al. 2017]	XC7VX690(1806.25,28,0.7)	(1[1], 1)	90	–	7.7/3.9/7.7	6.2 k	87.8%
[Liang et al. 2018]	5SGSD8(1600,28,0.7)	(1,1)	150	–	9.4/4.7/9.4	7.6 k[3]	86.31%
[Prost-Boucle et al. 2017]	VC709(1806.25,28,0.7)	(2,2)	250	–	8.4/4.2/8.4	27 k	86.7%
[Umuroglu et al. 2017]	ZC706(961,28,0.35)	(1,1)	200	283 µs	2.4/1.2/2.4	21.9 k	80.1%
[Fraser et al. 2017]	KU115(1600,20,1.45)	(1,1)	125	671 µs	14.8/7.4/14.8	12 k	88.7%
This work	VU9P(2256.25,20,2.6)	(2,16)	125	29 µs	2.5/2.5/37.3	122k	90.9%

[1] First layer is fixed point, [2] floating point, [3] estimated, [4] 92 TOps/sec peak, [5] LE and LC are from Xilinx or Altera documentation of the FPGAs, [6] Actual/Logical/Equivalent

Accuracy vs Speed (FPGA Implementations)



Summary

- Presented method to unroll convolution with ternary weights and make parallel implementation
  - Exploits unstructured sparsity with no overhead
  - Uses CSE, sparsity control and digit serial adders to further reduce area
  - Limited amount of buffering and only loosely dependent on image size
- As larger FPGAs become available this technique may become more favourable

Training deep neural networks in low-precision with high accuracy using FPGAs

Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip H.W. Leong

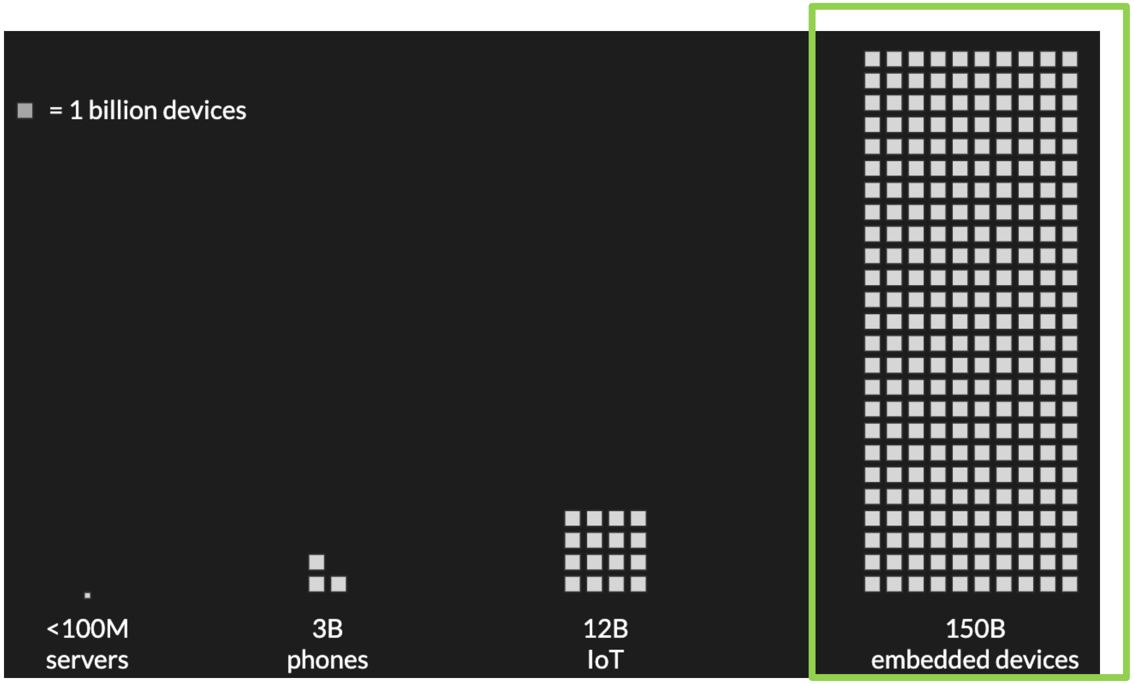
THE UNIVERSITY OF SYDNEY

Deep Learning on the Edge

- How do we get deep learning to the edge?
  - Train on GPUs
  - Move trained model to the edge

(and make use of fast, low power FPGA inference engines?)

- .



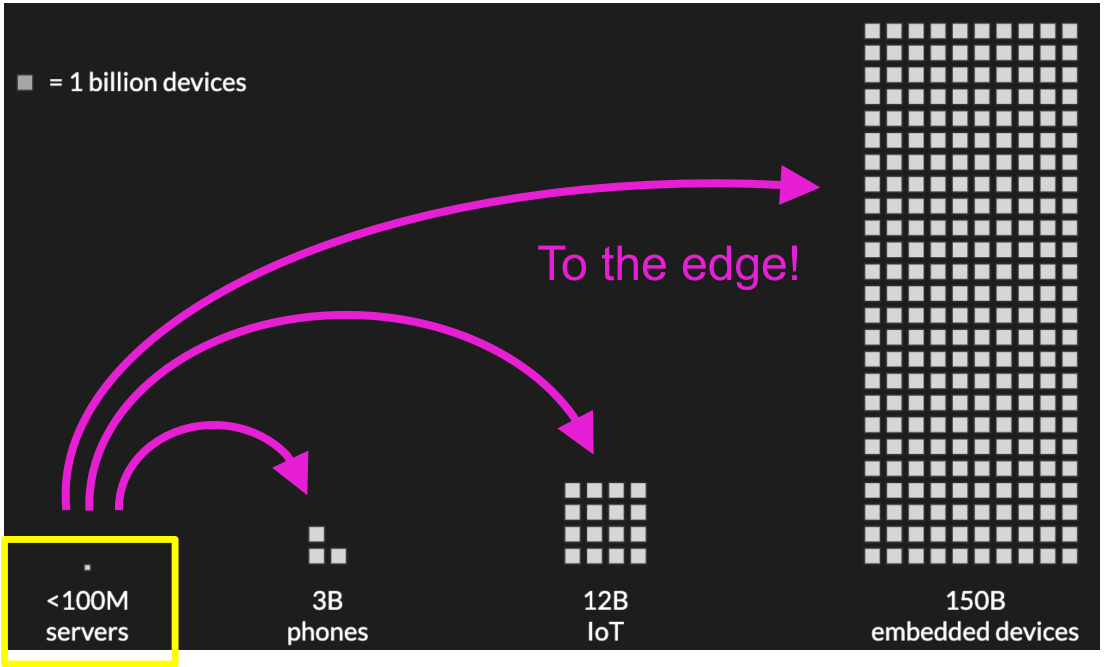
source: https://heartbeat.fritz.ai/

Deep Learning

- How do we get deep learning to the edge?
  - Train on GPUs
  - Move trained model to the edge

What about changing environments?

- .

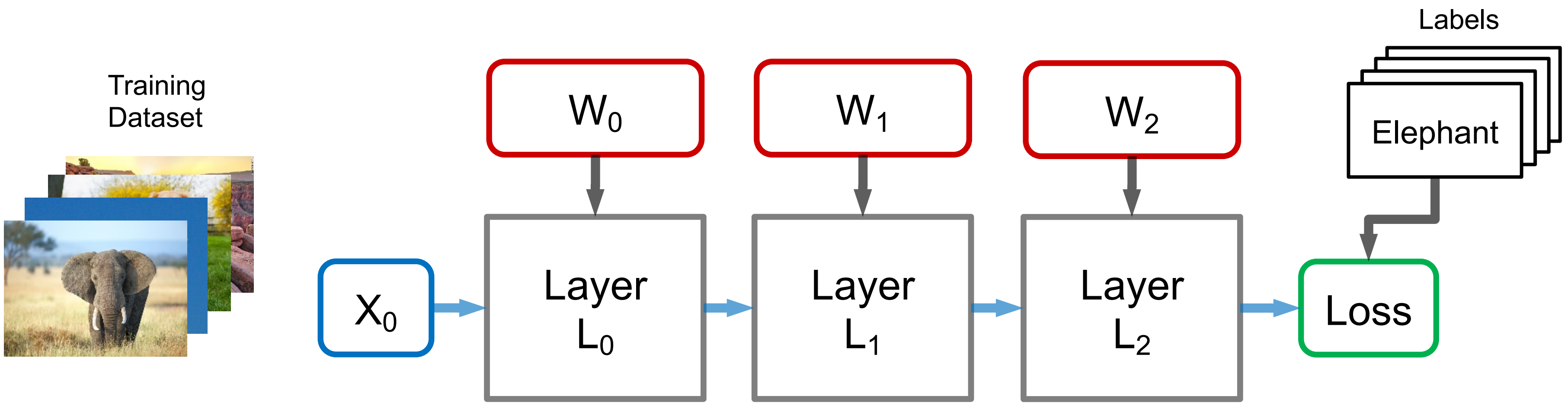


We need to be able to train at the edge

source: https://heartbeat.fritz.ai/

DNN Inference and Training

- Forward path (Inference)



- Each layer computes:
  - Linear transform (Matrix operation)
  - Non-linear activation function

$$f(\boldsymbol{x}) = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{b})$$

DNN Inference and Training

- Backward path:

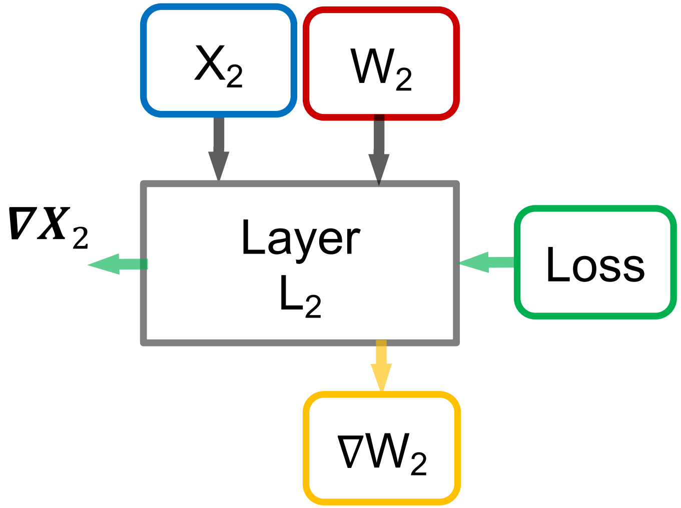
$$\boldsymbol{\nabla L}(.) = \frac{\partial Loss}{\partial y_{pred}}$$

- Step 1: Weight gradient

$$\boldsymbol{\nabla W_2} = \frac{\partial Loss}{\partial \boldsymbol{W}_2} = \boldsymbol{X}_2^T.\boldsymbol{\nabla L}(.)$$

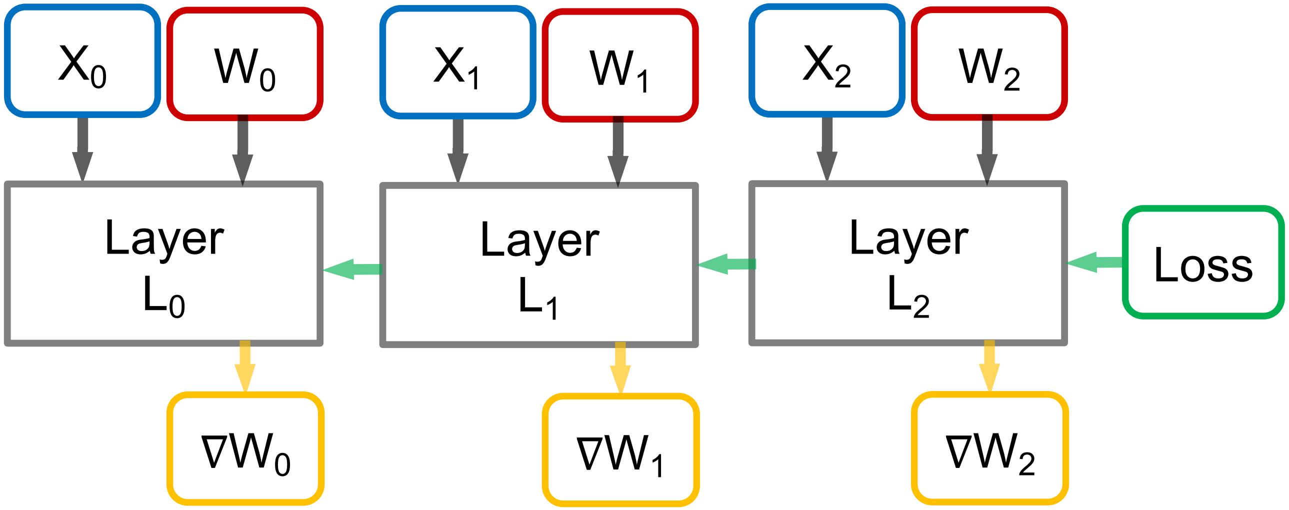
- Step 2: Activation gradient

$$\boldsymbol{\nabla X}_2 = \frac{\partial Loss}{\partial \boldsymbol{X}_2} = \boldsymbol{W}_2.\boldsymbol{\nabla L}(.)$$



DNN Inference and Training

- Backward path



- Two additional Matrix Mults at each layer:
  - Weight gradients
  - Activation gradients (or loss back-propagated)

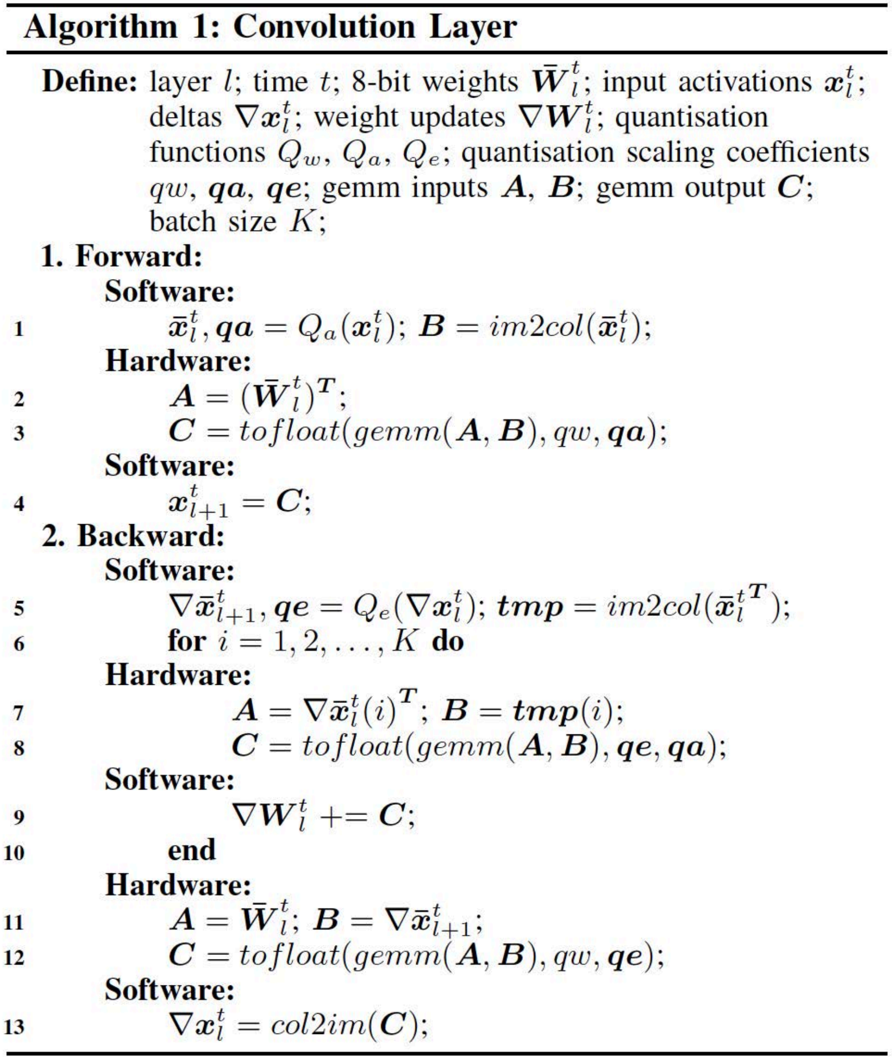
$$\boldsymbol{\nabla W}_l = \frac{\partial Loss}{\partial \boldsymbol{W}_l} = \boldsymbol{X}_l^T . \boldsymbol{\nabla X}_{l+1}$$

$$\boldsymbol{\nabla X}_l = \frac{\partial Loss}{\partial \boldsymbol{X}_l} = \boldsymbol{W}_l . \boldsymbol{\nabla X}_{l+1}$$

DNN Training

- DNN training is dominated by matrix multiplication
- GPUs good at floating-point matrix operations
  - Large dynamic range and avoids excessive quantisation errors critical for DNN training

Mixed-Precision Algorithm



```
Algorithm 1: Convolution Layer
  Define: layer $l$; time $t$; 8-bit weights $\bar{\boldsymbol{W}}_l^t$; input activations $\boldsymbol{x}_l^t$;
          deltas $\nabla\boldsymbol{x}_l^t$; weight updates $\nabla\boldsymbol{W}_l^t$; quantisation
          functions $Q_w$, $Q_a$, $Q_e$; quantisation scaling coefficients
          $qw$, $\boldsymbol{qa}$, $\boldsymbol{qe}$; gemm inputs $\boldsymbol{A}$, $\boldsymbol{B}$; gemm output $\boldsymbol{C}$;
          batch size $K$;
  1. Forward:
      Software:
          $\bar{\boldsymbol{x}}_l^t, \boldsymbol{qa} = Q_a(\boldsymbol{x}_l^t)$; $\boldsymbol{B} = im2col(\bar{\boldsymbol{x}}_l^t)$;
      Hardware:
          $\boldsymbol{A} = (\bar{\boldsymbol{W}}_l^t)^{\boldsymbol{T}}$;
          $\boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), qw, \boldsymbol{qa})$;
      Software:
          $\boldsymbol{x}_{l+1}^t = \boldsymbol{C}$;
  2. Backward:
      Software:
          $\nabla\bar{\boldsymbol{x}}_{l+1}^t, \boldsymbol{qe} = Q_e(\nabla\boldsymbol{x}_l^t)$; $\boldsymbol{tmp} = im2col(\bar{\boldsymbol{x}}_l^{t\,\boldsymbol{T}})$;
          for $i = 1, 2, \ldots, K$ do
      Hardware:
              $\boldsymbol{A} = \nabla\bar{\boldsymbol{x}}_l^t(i)^{\boldsymbol{T}}$; $\boldsymbol{B} = \boldsymbol{tmp}(i)$;
              $\boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), \boldsymbol{qe}, \boldsymbol{qa})$;
      Software:
              $\nabla\boldsymbol{W}_l^t \mathrel{+}= \boldsymbol{C}$;
          end
      Hardware:
          $\boldsymbol{A} = \bar{\boldsymbol{W}}_l^t$; $\boldsymbol{B} = \nabla\bar{\boldsymbol{x}}_{l+1}^t$;
          $\boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), qw, \boldsymbol{qe})$;
      Software:
          $\nabla\boldsymbol{x}_l^t = col2im(\boldsymbol{C})$;
```

FPGA

- Low-Precision (8-bit)
  - All matrix multiplications
  - >95% of DNN operations

ARM

- High-Precision
  - Everything else!
  - Of particular importance is the **weight update and gradient accumulator**
- Suits a Zynq platform
  - Fast DDR, shared between PL and floating-point

Mixed-Precision Algorithm

- Weight Update

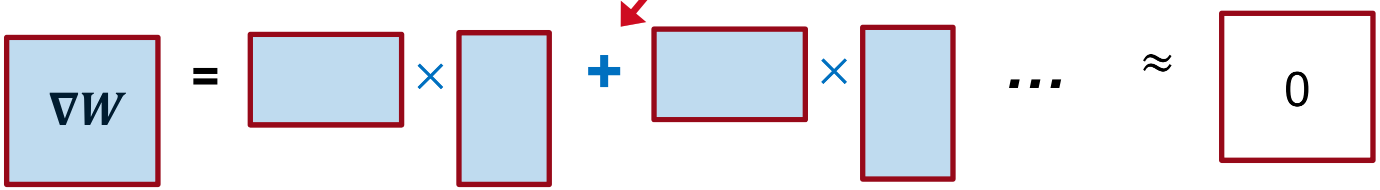
$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^{t} - \alpha\, \boldsymbol{\nabla W}^{t}$$

If learning rate α and weight gradient $\boldsymbol{\nabla W}^{t}$ are small, then the update will be rounded to 0

- Weight Gradient Accumulator

$$\boldsymbol{\nabla W} = \sum_{\boldsymbol{i=1}}^{\boldsymbol{batch\_size}} \boldsymbol{x}^{T} . \boldsymbol{\nabla x}$$

If accumulator is low precision, then small numbers are rounded to 0



Mixed-Precision Algorithm

- Weight Update

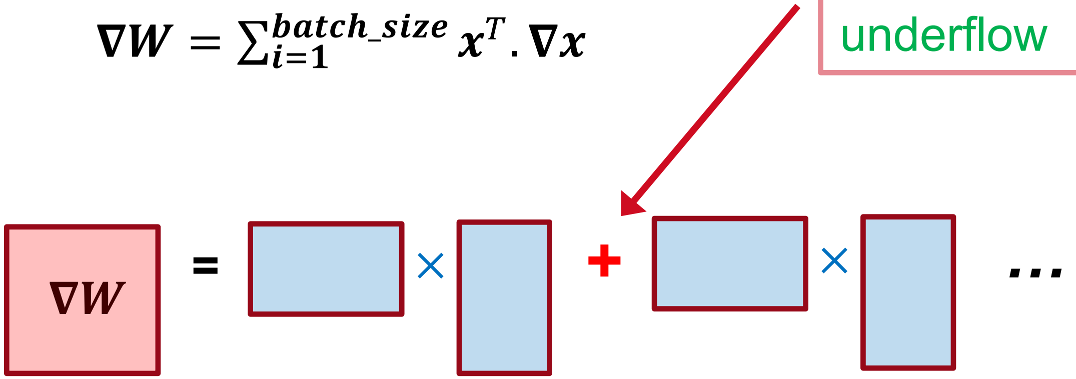
$$\boldsymbol{W}^{t+1} = \boldsymbol{W}^{t} - \alpha\, \nabla\boldsymbol{W}^{t}$$

If learning rate α and weight gradient $\nabla\boldsymbol{W}^{t}$ are small, then the update will be rounded to 0

- Weight Gradient Accumulator

$$\nabla\boldsymbol{W} = \sum_{i=1}^{batch\_size} \boldsymbol{x}^{T}.\nabla\boldsymbol{x}$$

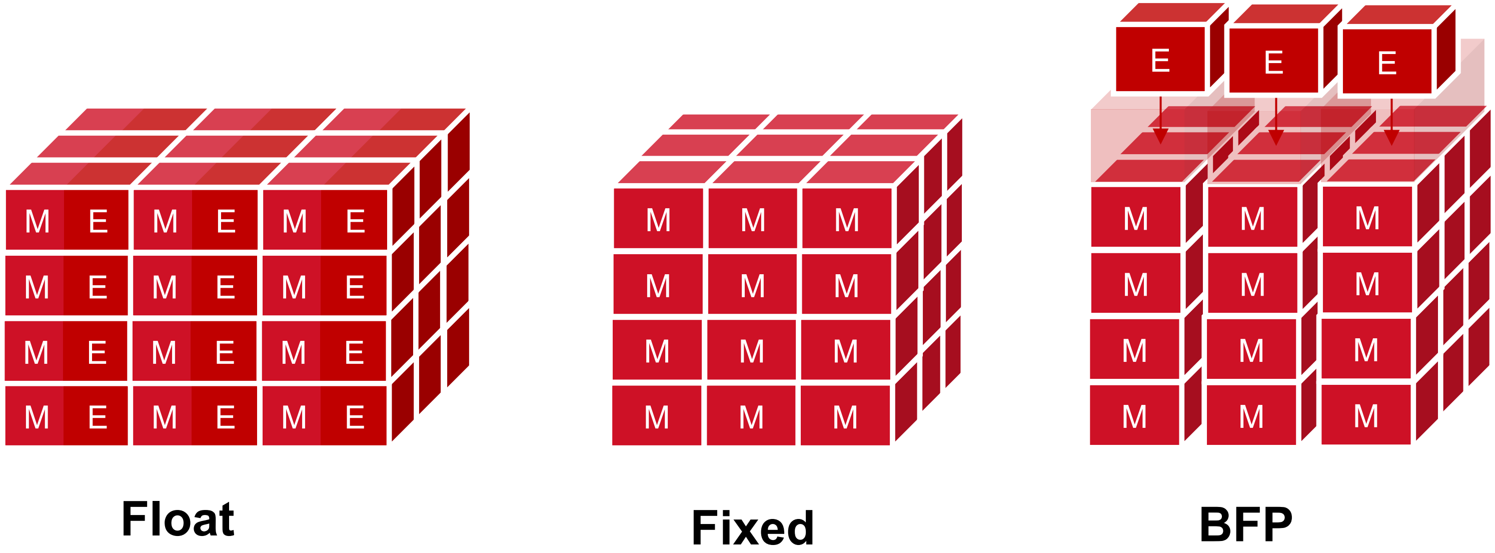
High-precision accumulator can avoid underflow



Block Floating Point (BFP)

- A group of numbers that share the same exponent

- Outcome: Dynamic Range
  (at a cost of some accuracy degradation)

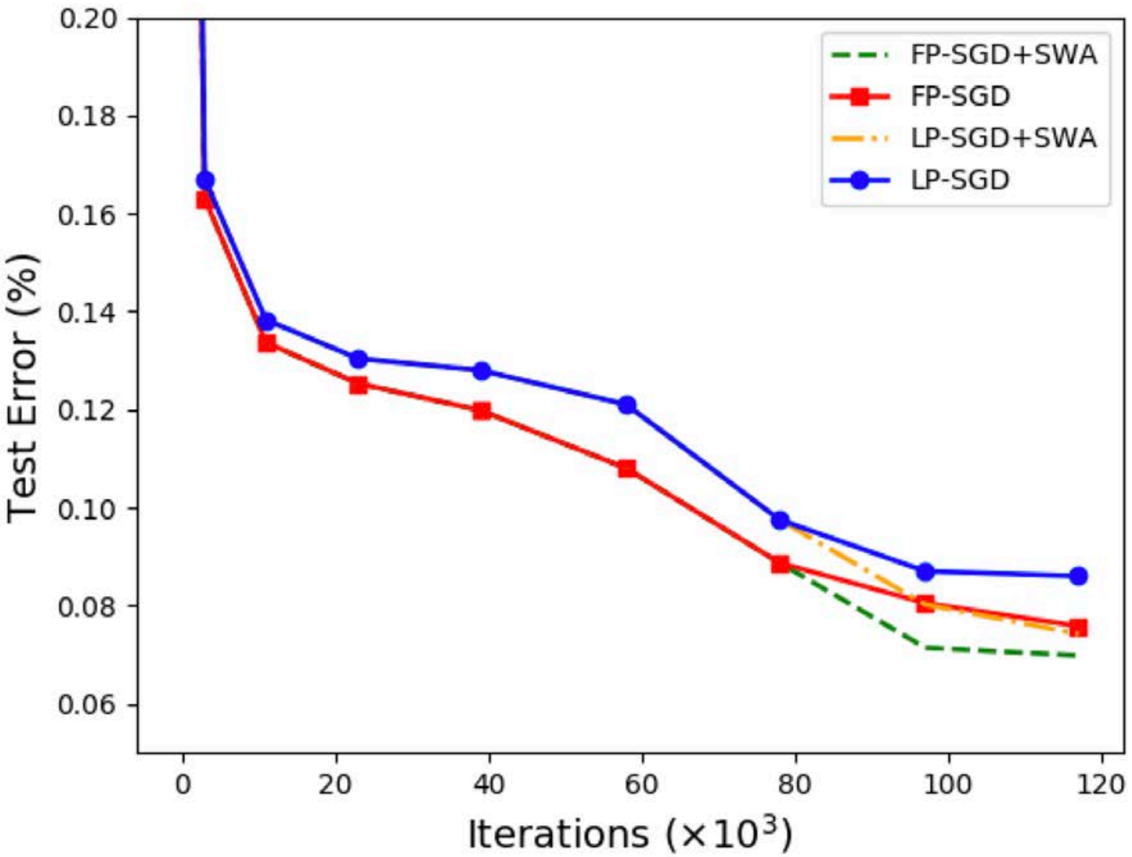


We use 1 block for weights, b for activations and deltas (where b is batch size)

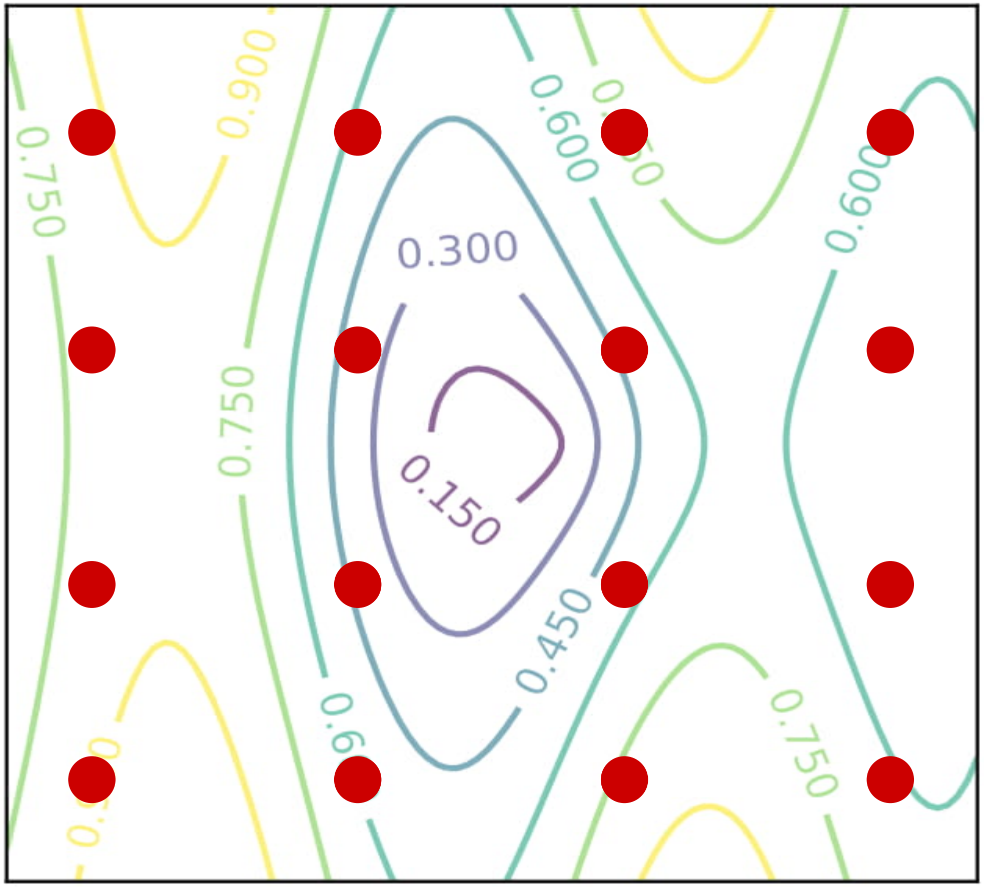
THE UNIVERSITY OF SYDNEY

Stochastic Weight Averaging Low Precision (SWALP)

- SWALP - averaging SGD iterates of the weights with a higher learning rate can recover quantisation errors

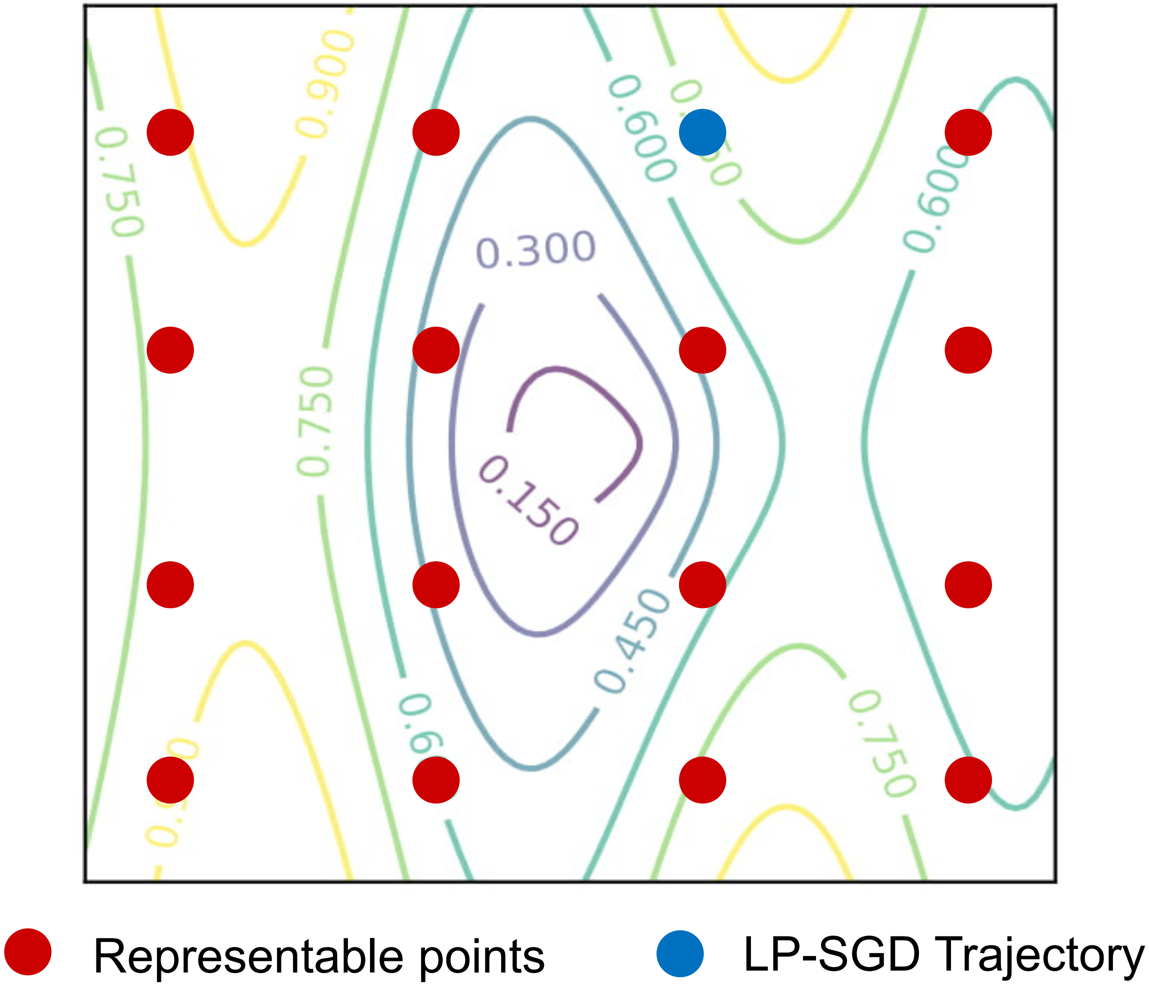


Stochastic Weight Averaging (SWA)

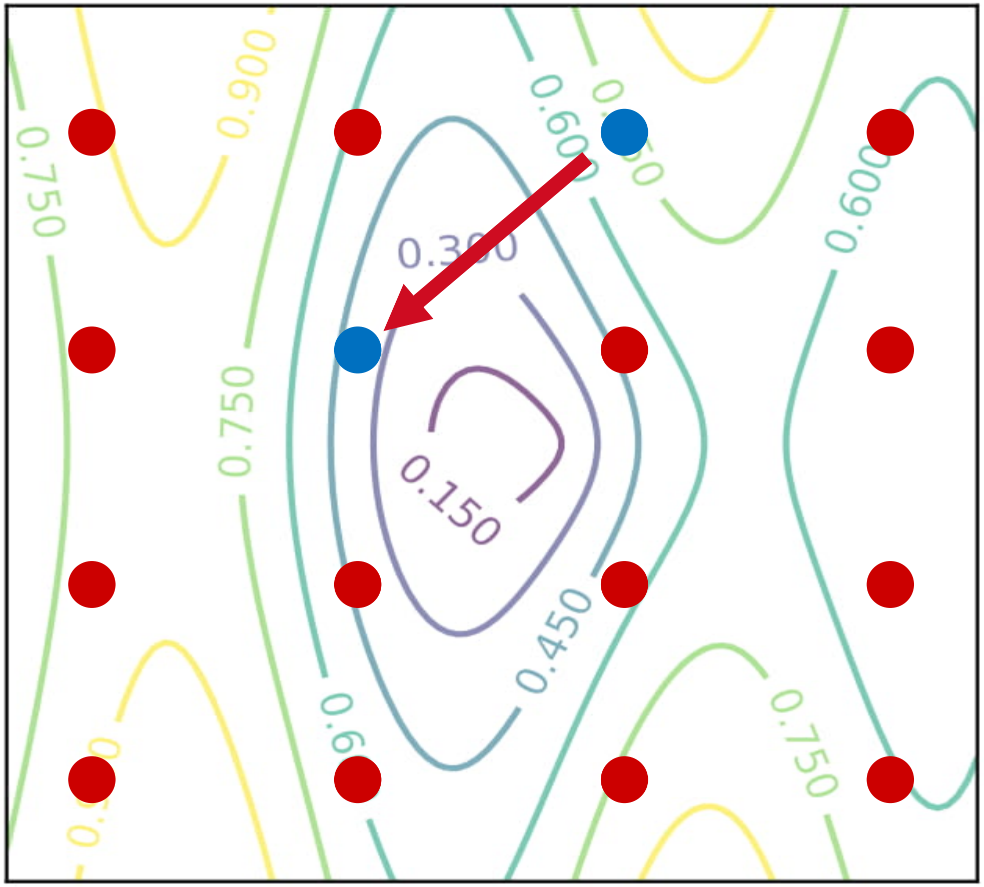


● Representable points

Stochastic Weight Averaging (SWA)

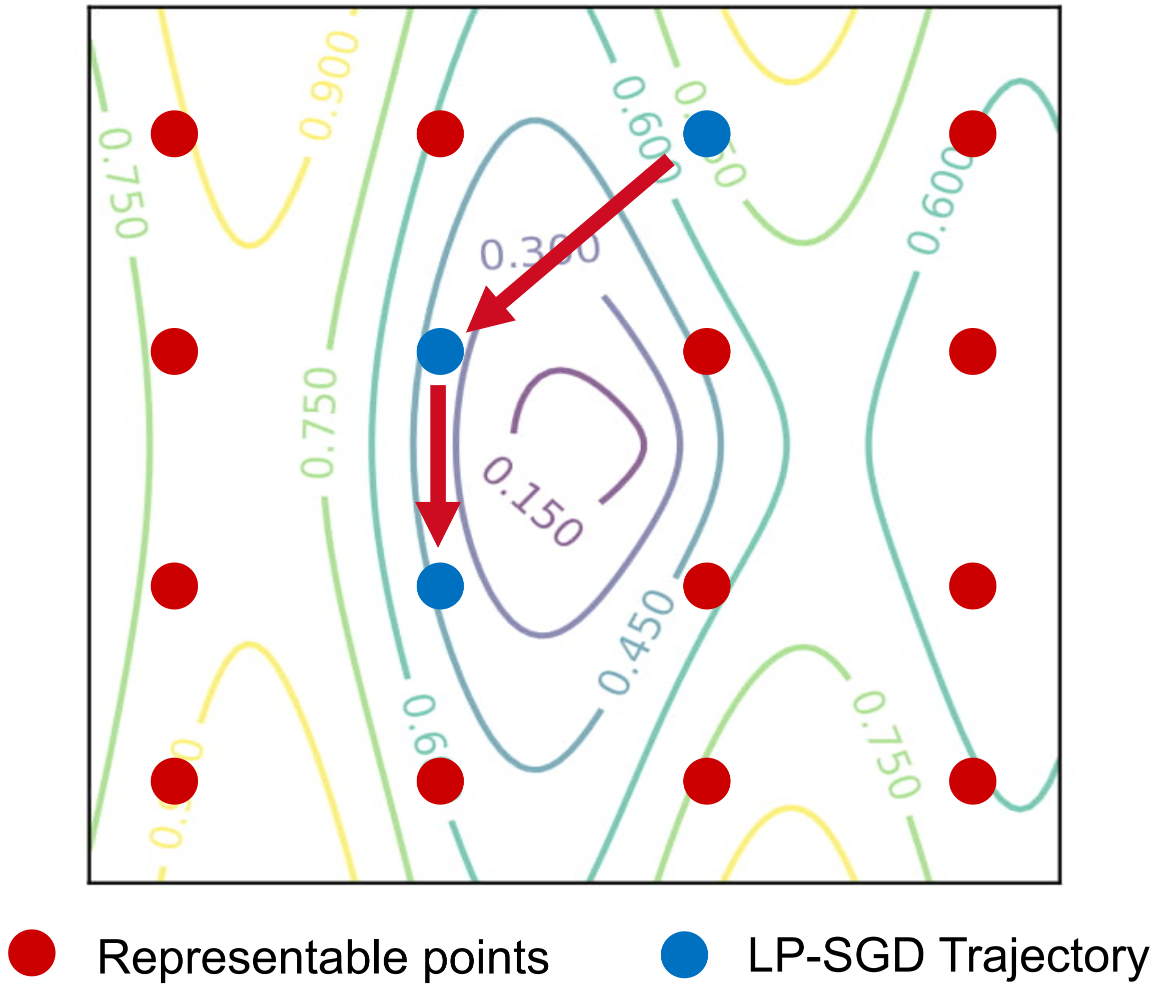


Stochastic Weight Averaging (SWA)

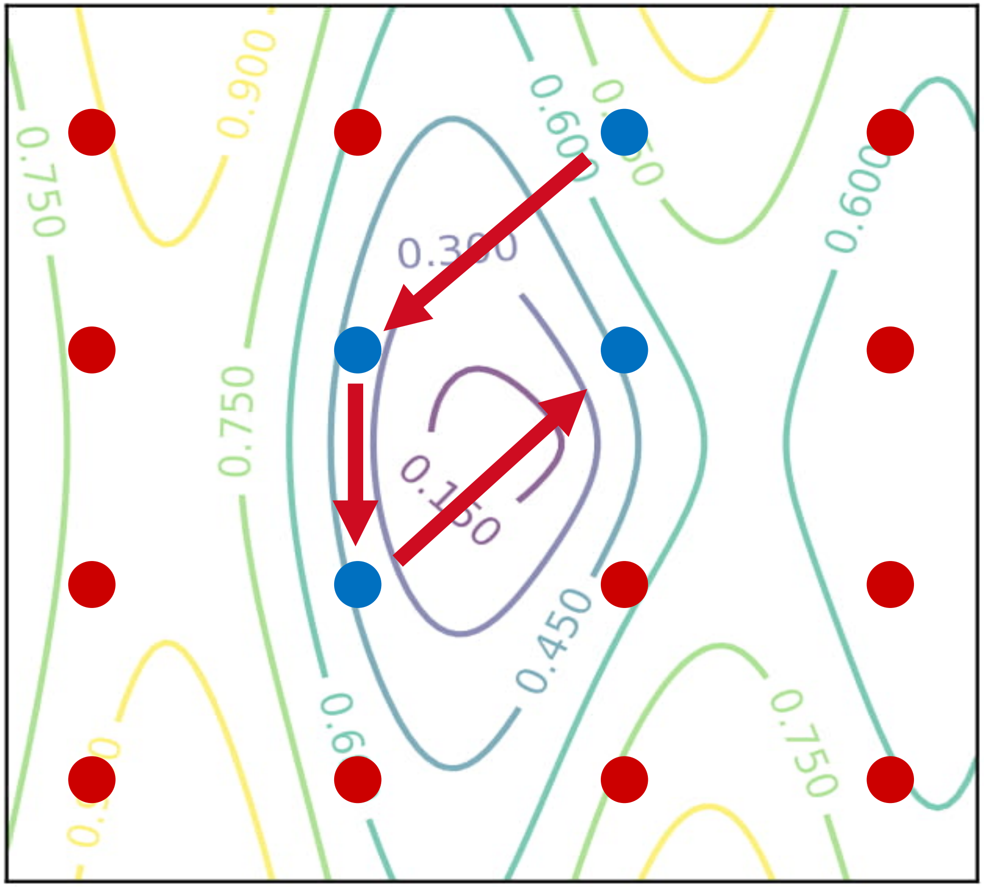


● Representable points ● LP-SGD Trajectory

Stochastic Weight Averaging (SWA)

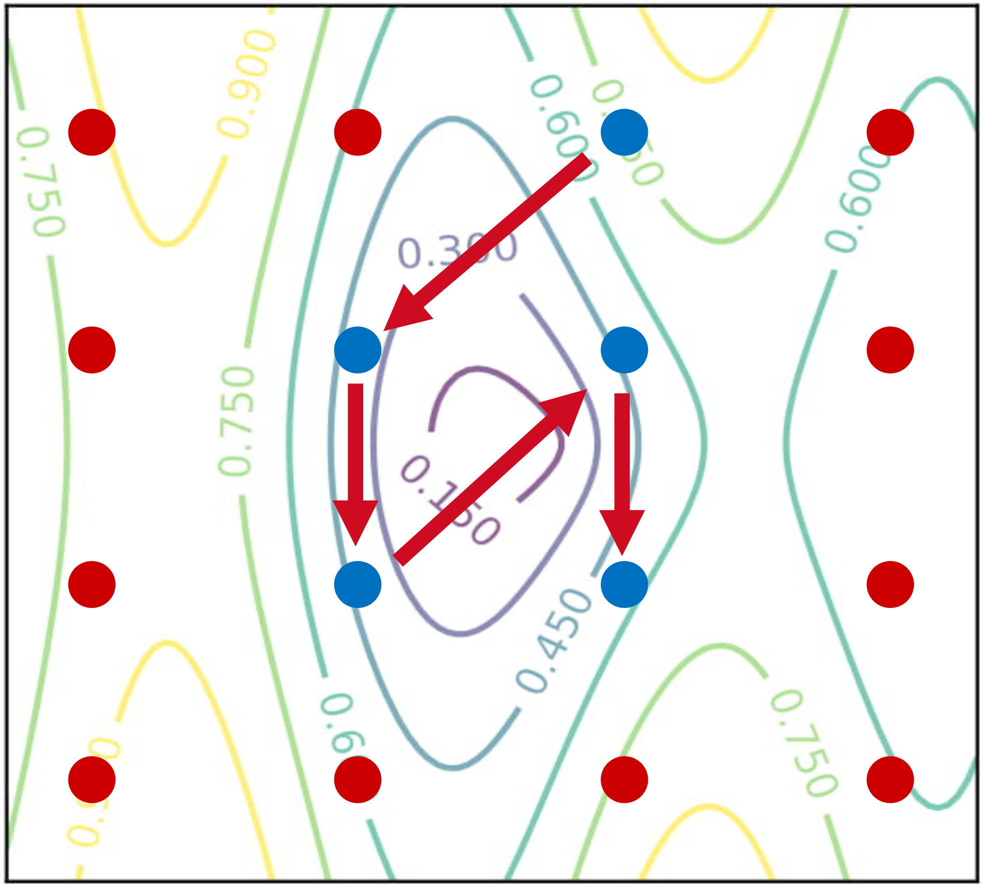


Stochastic Weight Averaging (SWA)



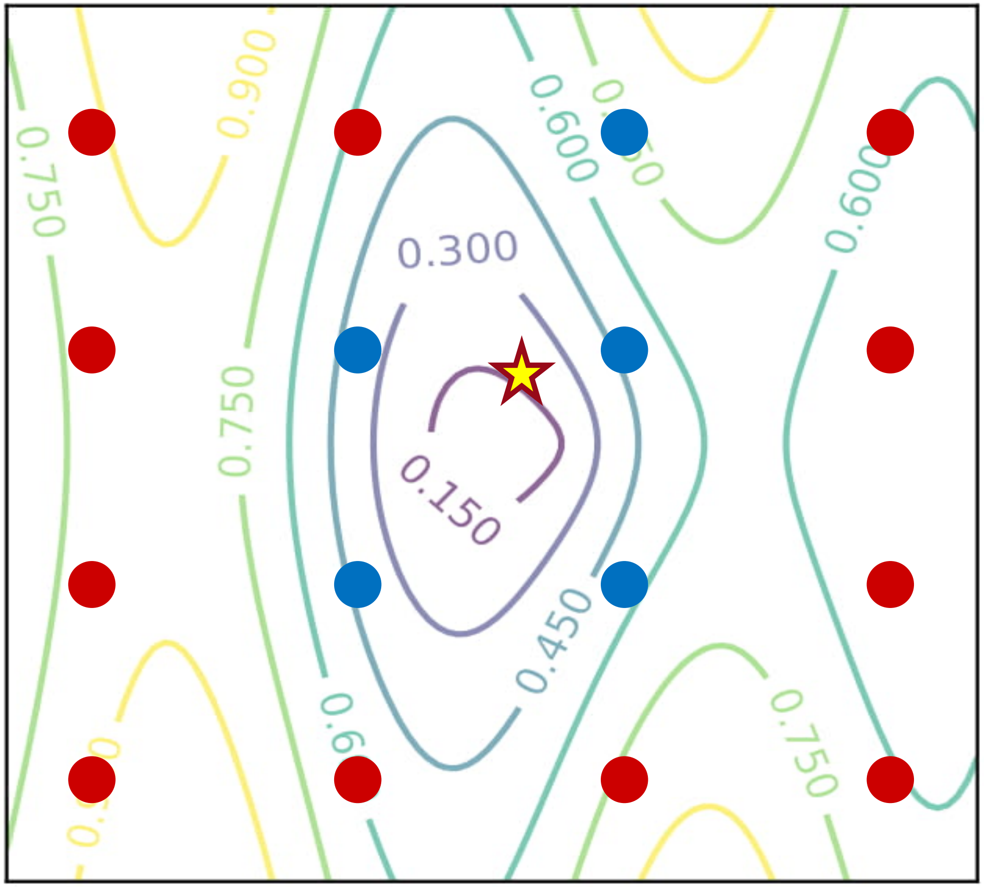
🔴 Representable points 🔵 LP-SGD Trajectory

Stochastic Weight Averaging (SWA)



● Representable points ● LP-SGD Trajectory

Stochastic Weight Averaging (SWA)

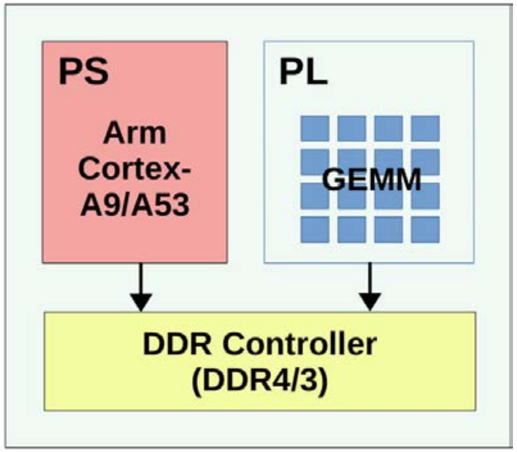


● Representable points ● LP-SGD Trajectory ★ SWA Solution

Zynq APSoC

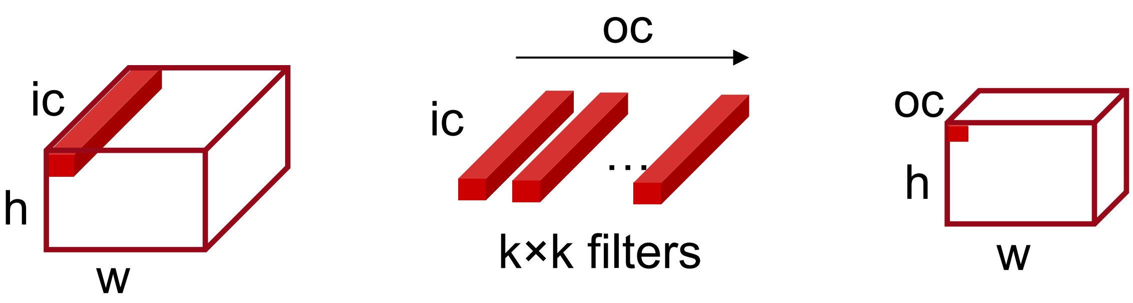
- Heterogenous computing platform:
  1. Processing System (ARM processor)
  2. Programmable Logic (8-bit GEMM)
  3. Shared DRAM

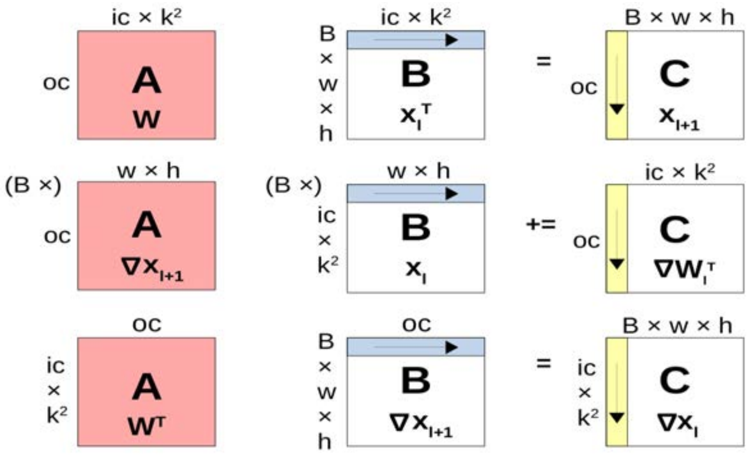
- Well suited to standalone embedded applications



Example: Convolution

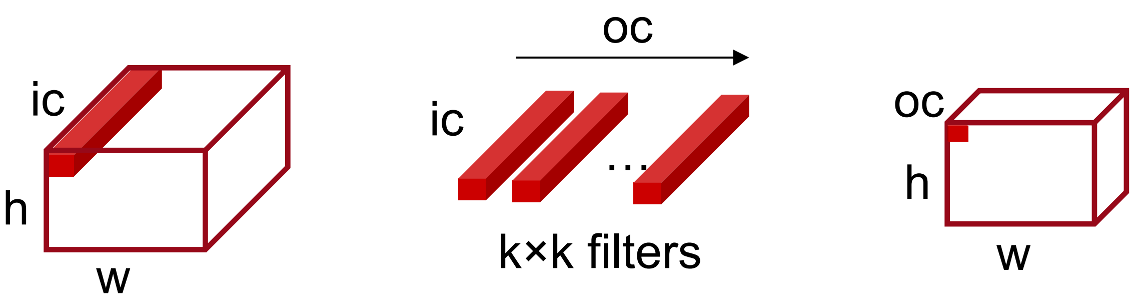
- Convolution must be transformed into Matrix Multiplication

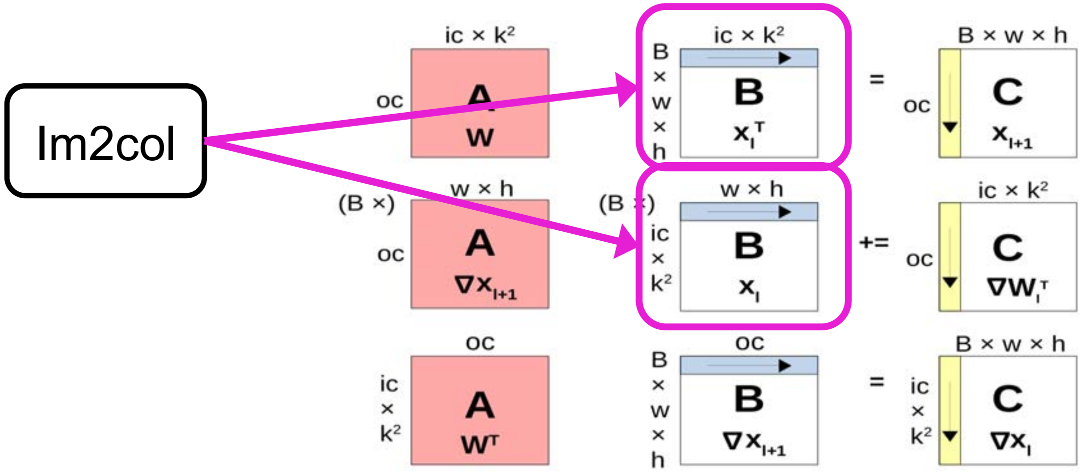




Example: Convolution

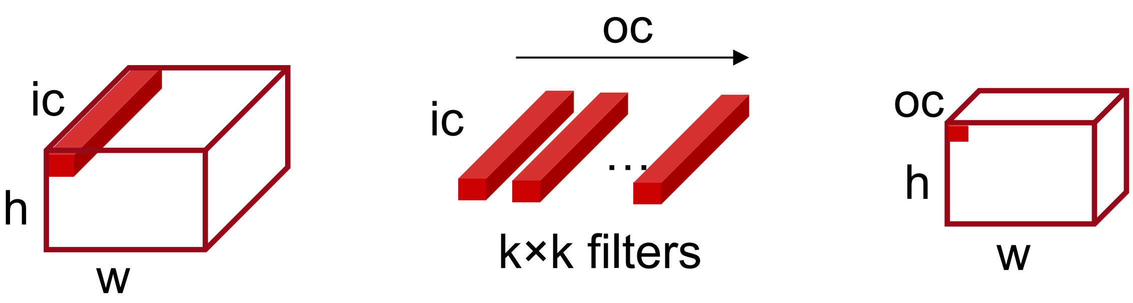
- Convolution must be transformed into Matrix Multiplication

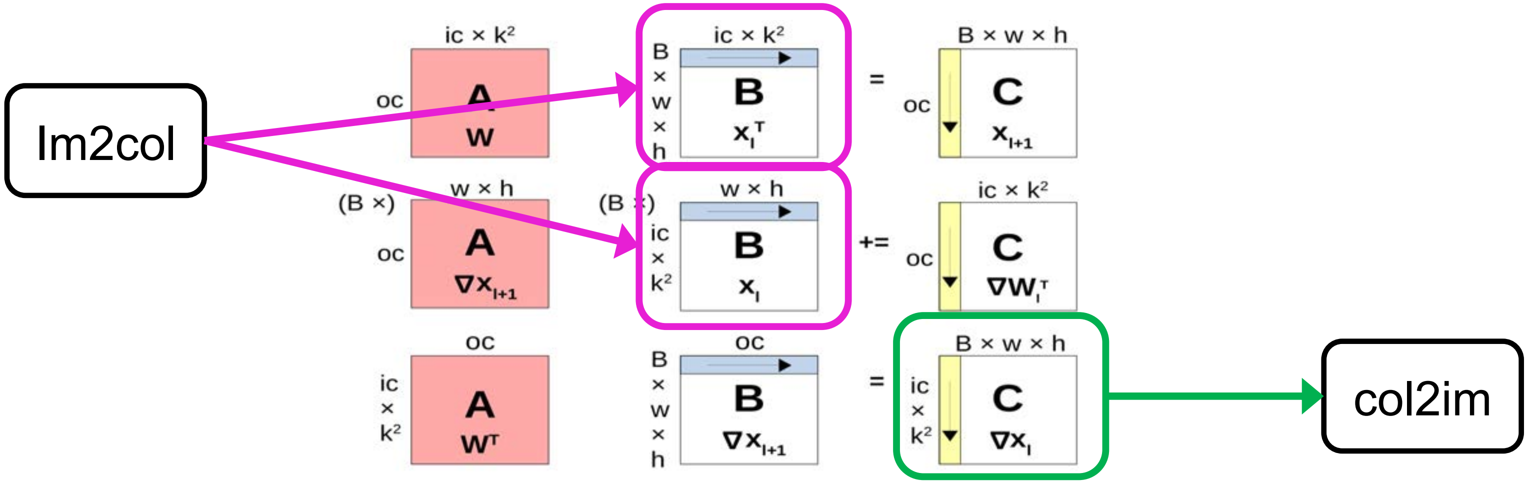




Example: Convolution

- Convolution must be transformed into Matrix Multiplication

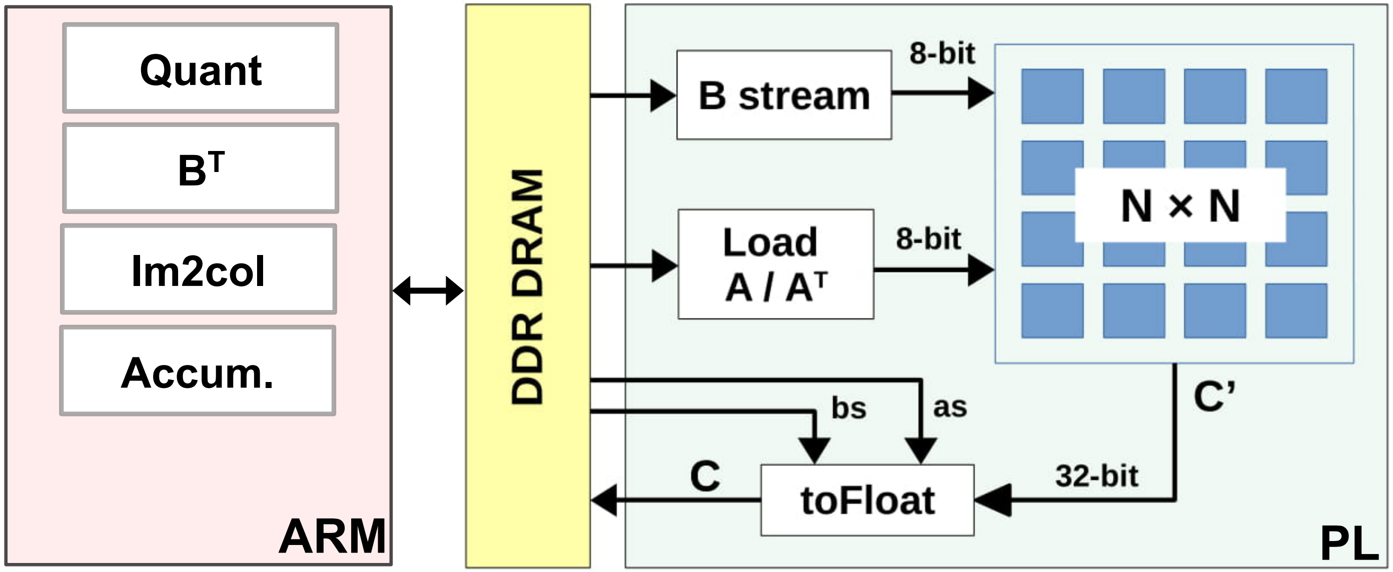




THE UNIVERSITY OF SYDNEY

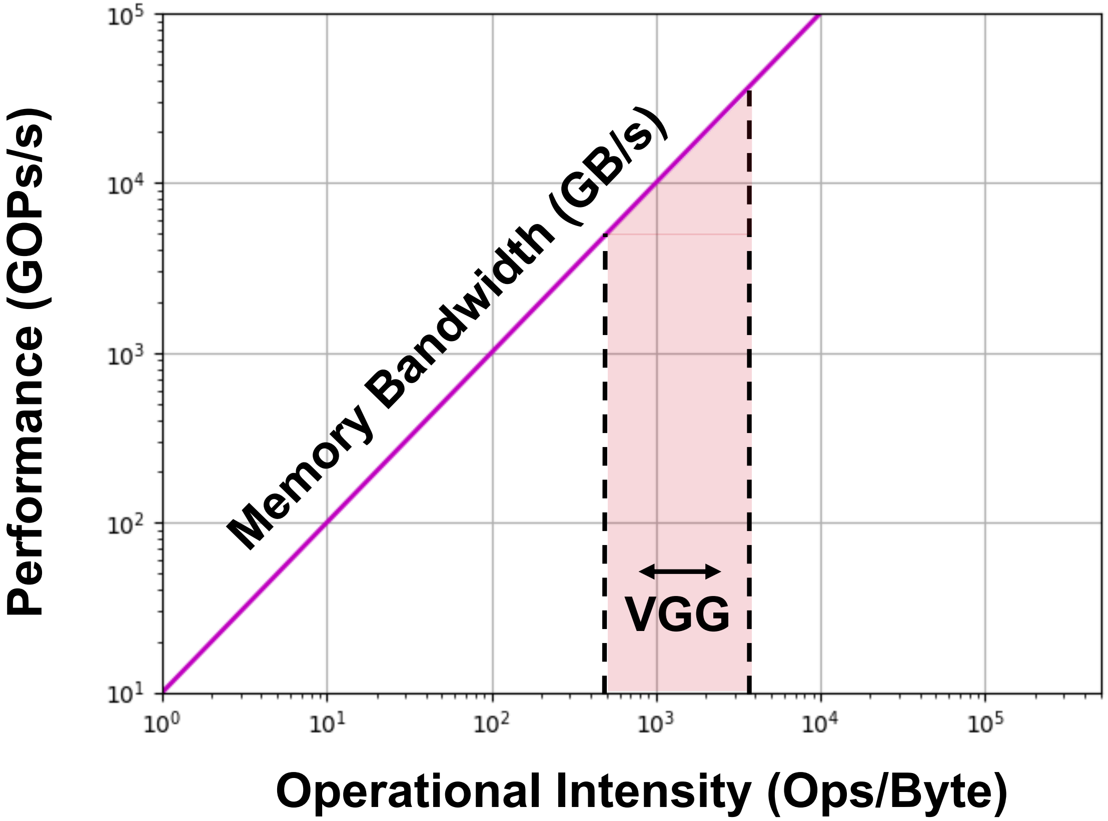
Hardware Accelerator

- 8-bit GEMM (C = A.B or C=A^T.B)
- Block Floating Point (BFP) rescaling



Hardware Accelerator

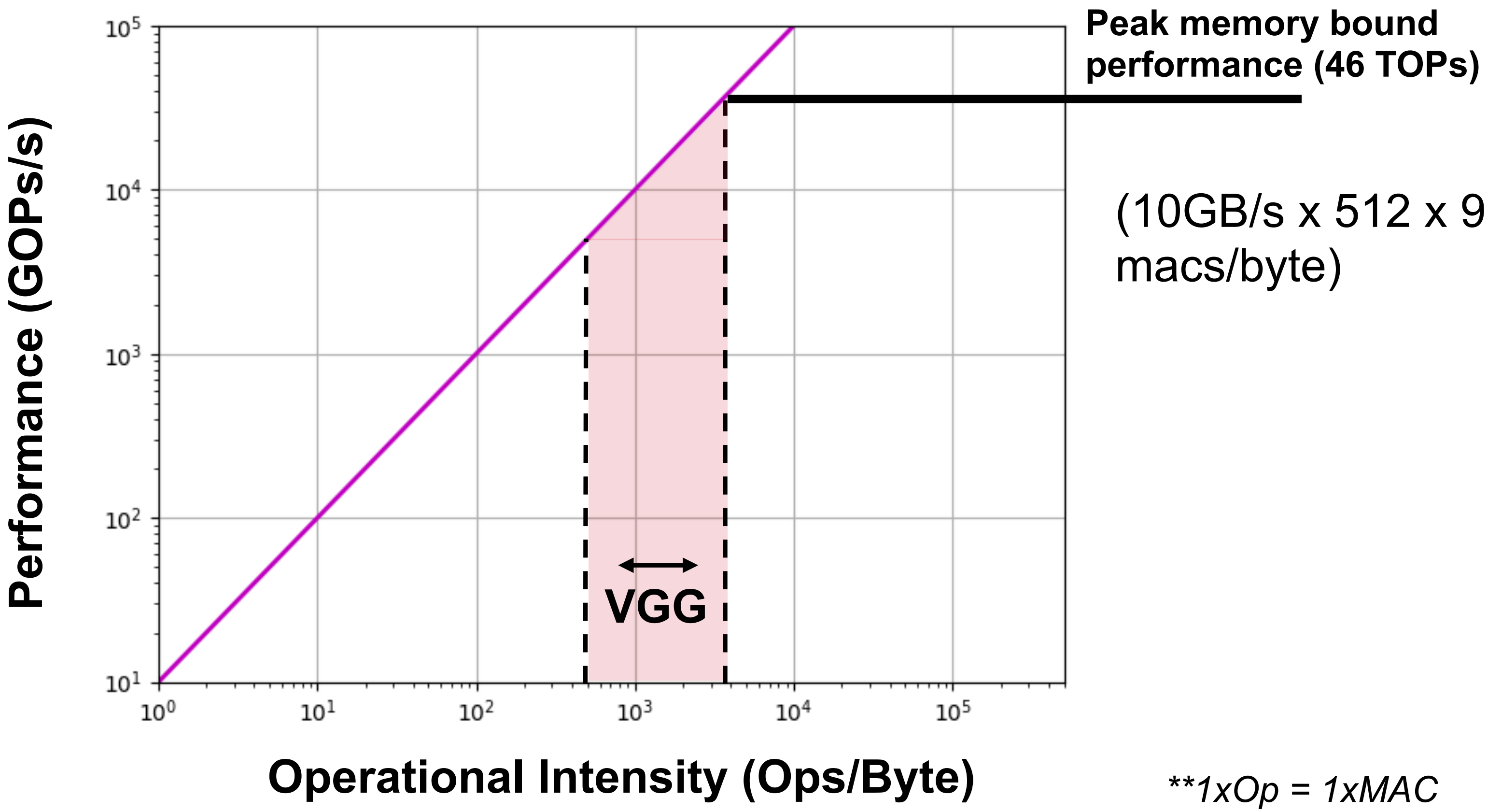
- Red line: 32x32 Array @ 200Mhz on ZCU111



***1xOp = 1xMAC*

Hardware Accelerator

- Red line: 32x32 Array @ 200Mhz on ZCU111



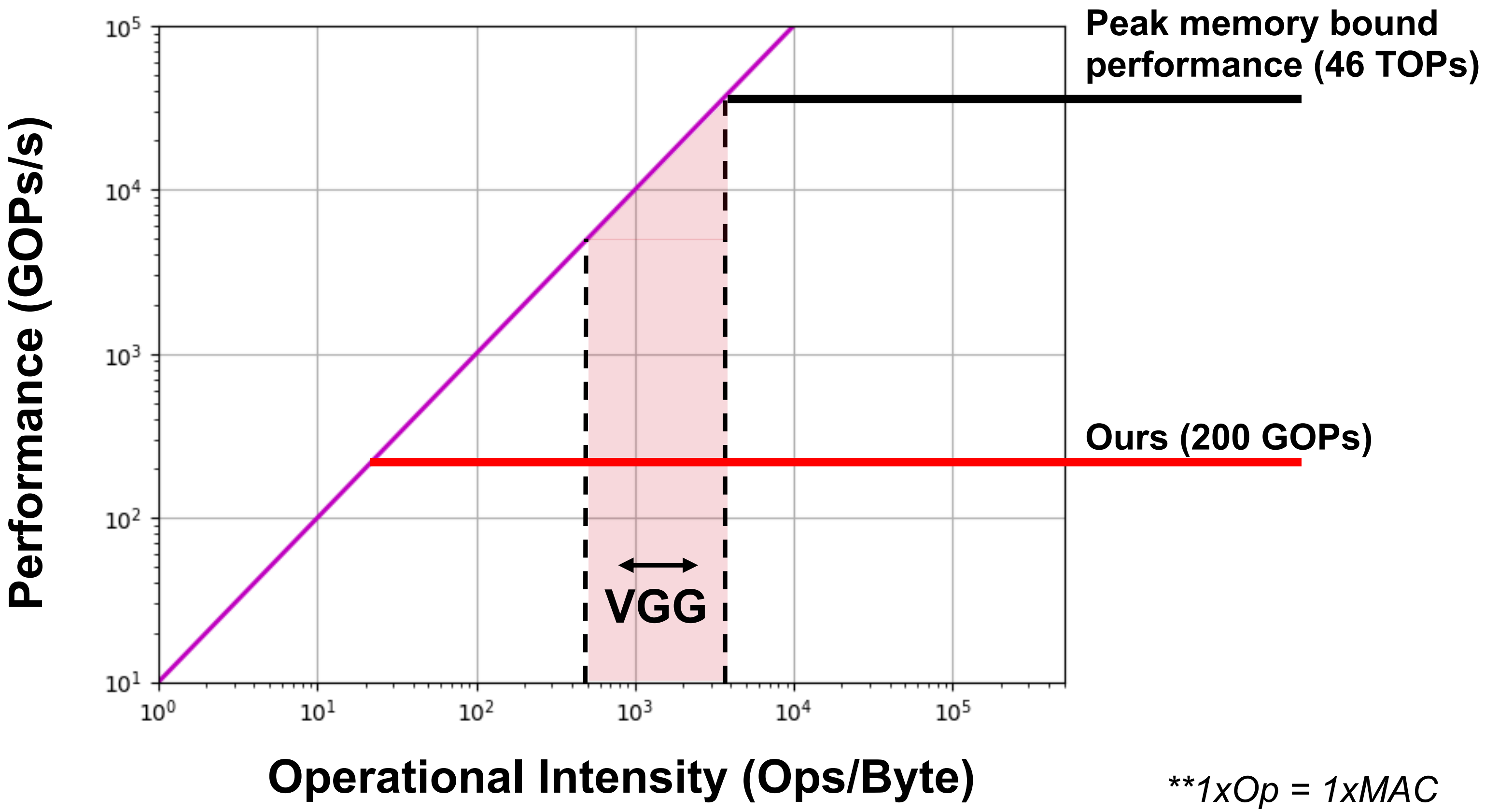
Peak memory bound performance (46 TOPs)

(10GB/s x 512 x 9 macs/byte)

***1xOp = 1xMAC*

Hardware Accelerator

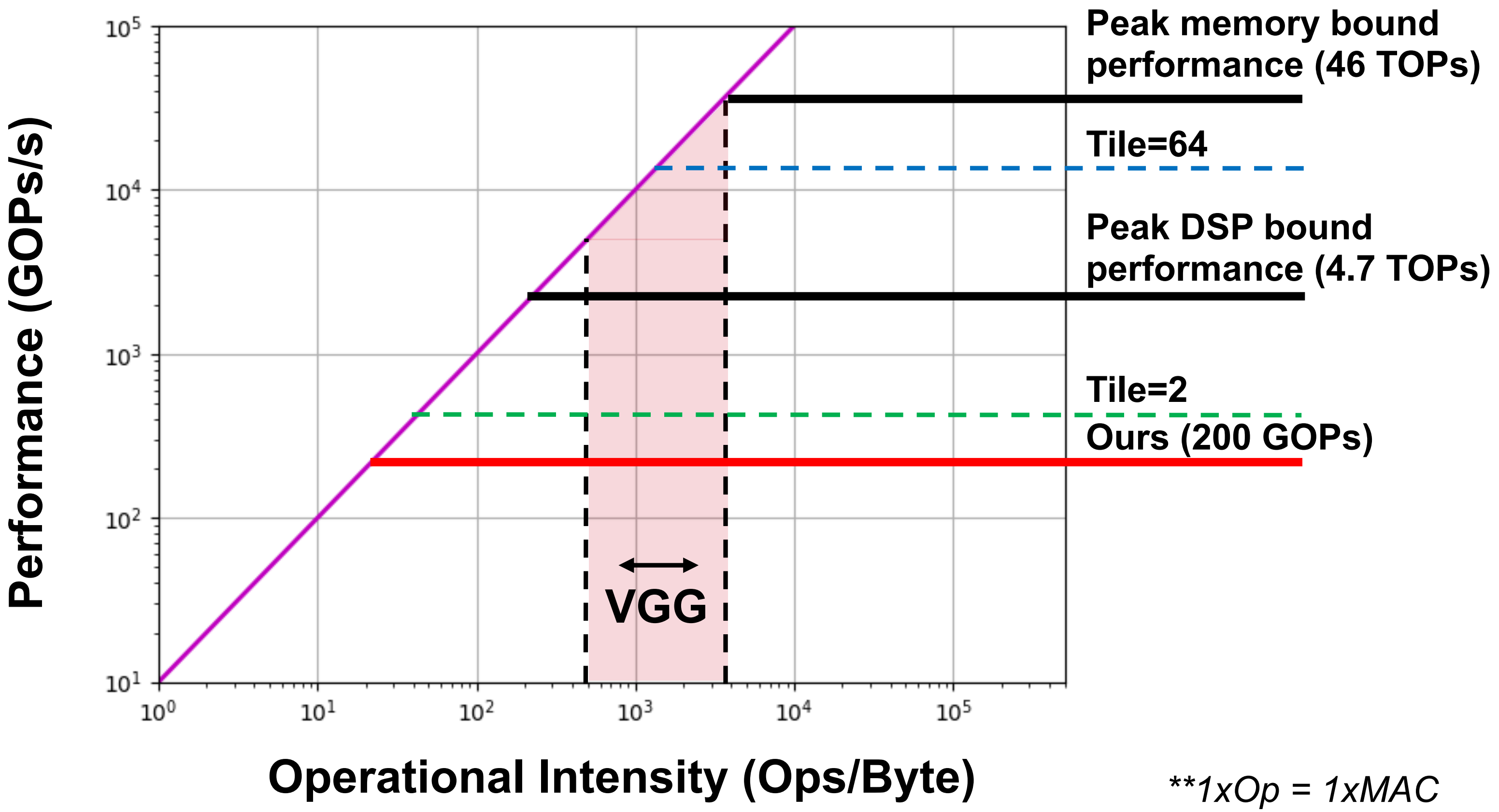
- Red line: 32x32 Array @ 200Mhz on ZCU111



***1xOp = 1xMAC*

Hardware Accelerator

- Red line: 32x32 Array @ 200Mhz on ZCU111



***1xOp = 1xMAC*

Design and Runtime

- Software integration with Darknet (Yolo backend)
- Available: https://github.com/sfox14/darknet-zynq

Bitstream (.bit) and C/C++ runtime library (.so)

Open source C library for DNN training and inference

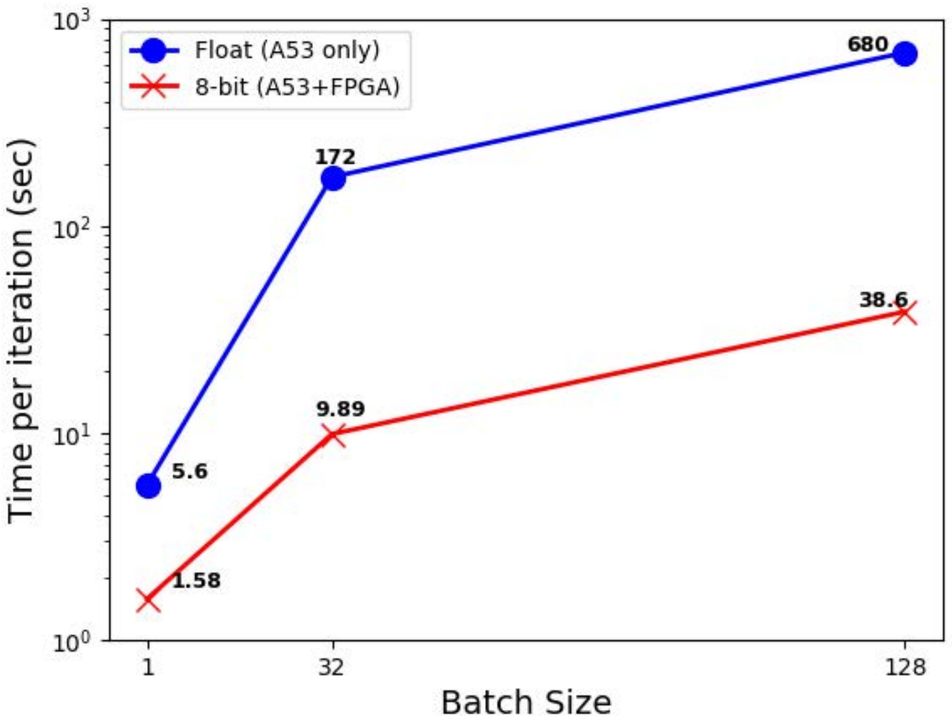
Our code can be compiled and run on ZYNQ boards with PYNQ v2.3 image

High Accuracy

Dataset	Model	Float	SWALP	Ours (8-bit)
CIFAR10	VGG16	93.02	92.47	92.93
	PreResNet-20	93.50	93.29	93.29
MNIST	Logistic Regression	92.60	92.06	92.70

- ❑ High accuracy can be sustained with mostly 8-bits
  - only requires a few more epochs than float
- ❑ Batch=128, no bias, no batchnorm
- ❑ Lr=0.01, SWA applied to last 25% of epochs

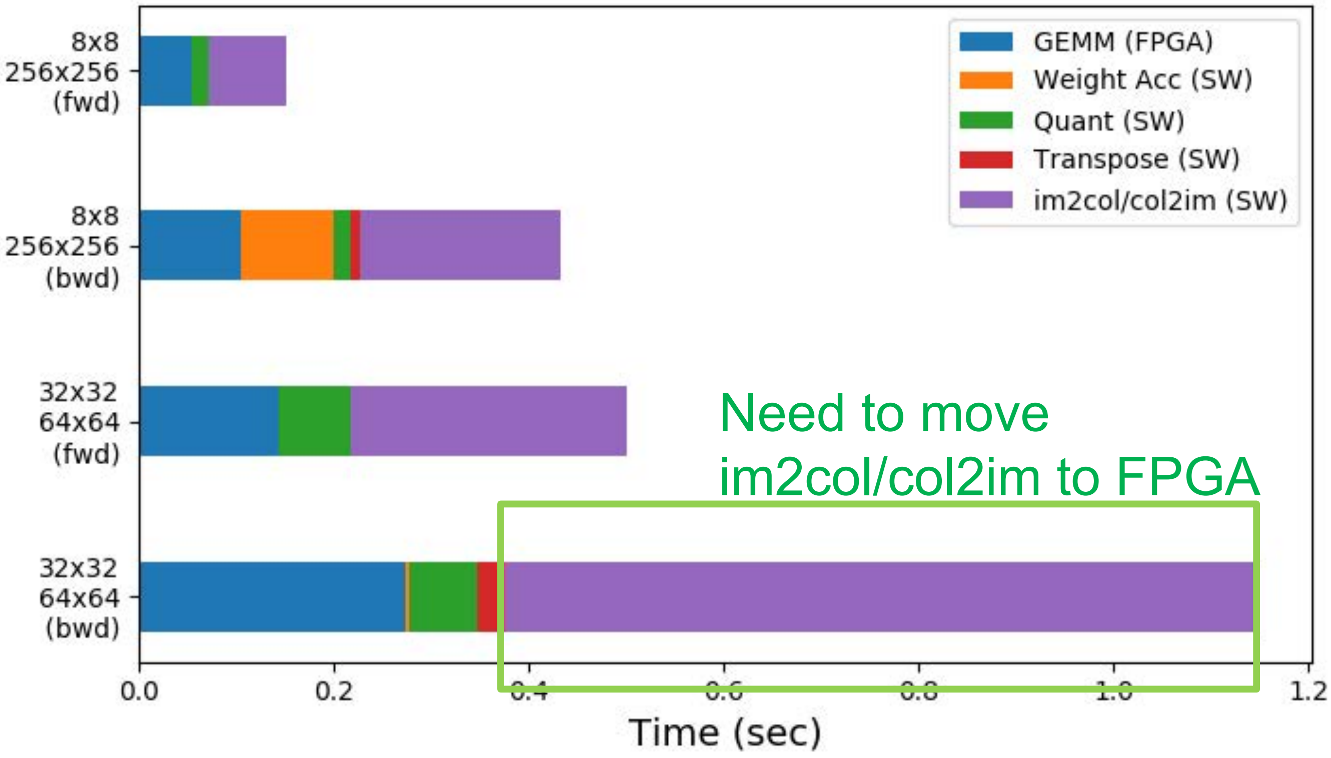
Training Time



- Good speed-up over ARM
- We don’t exploit batch-level parallelism
- Much slower than a Tesla M40 GPU (600x) …

Software Overheads

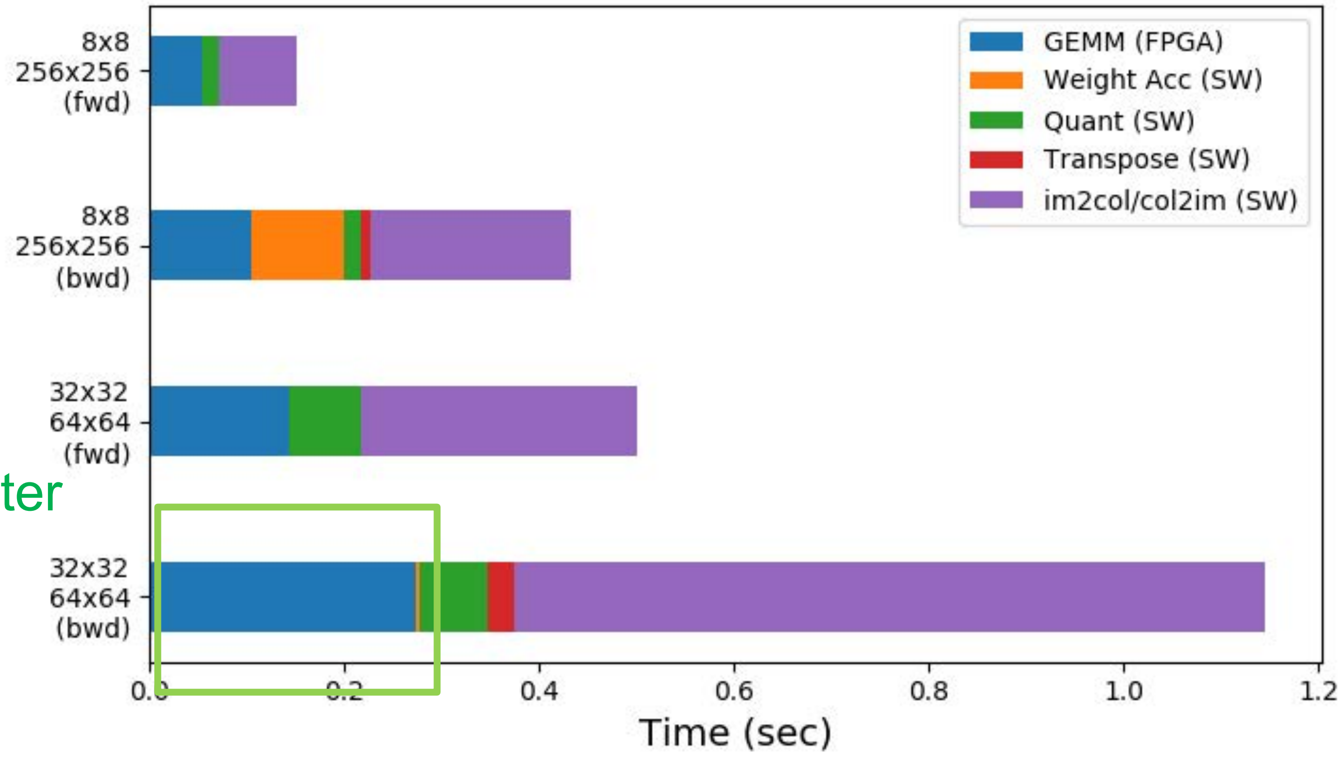
- Profiling VGG16 convolution layers (Pynq-Z1 and ZCU111)



Need to move im2col/col2im to FPGA

Software Overheads

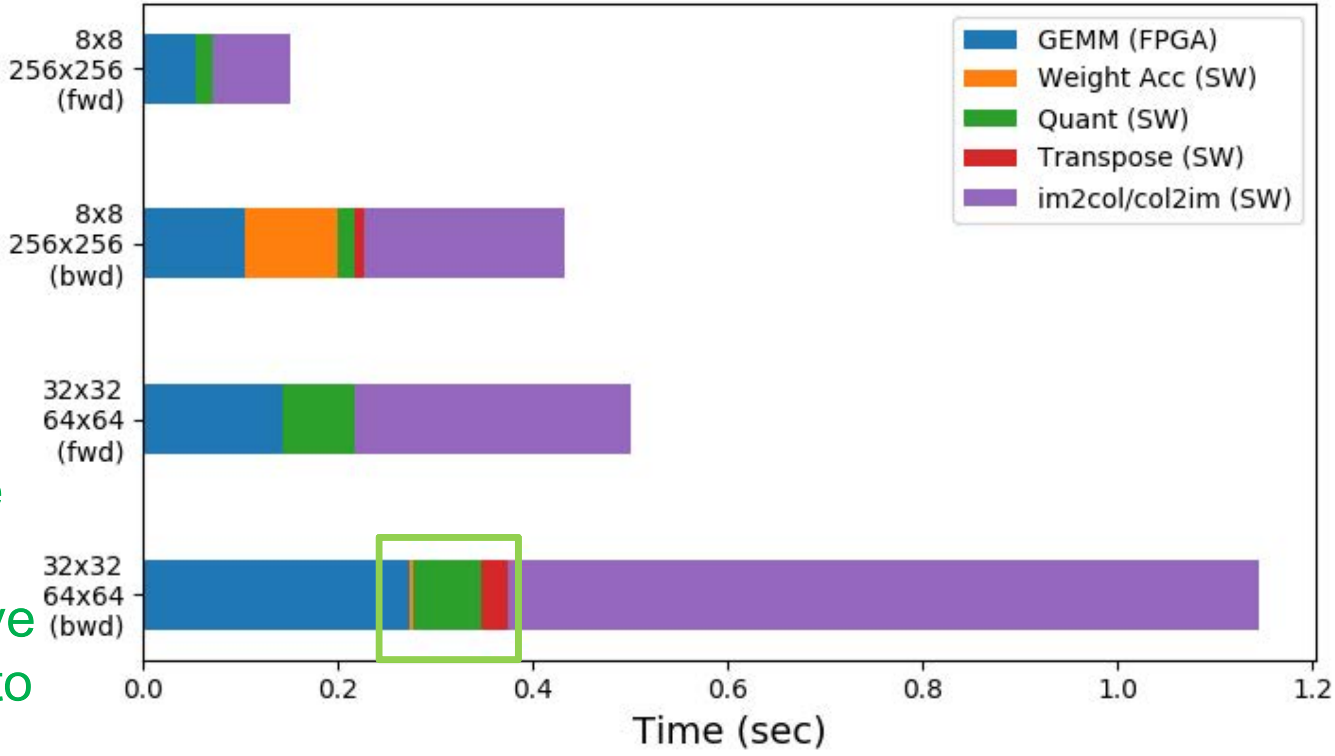
- Profiling VGG16 convolution layers



Room for better GEMM

Software Overheads

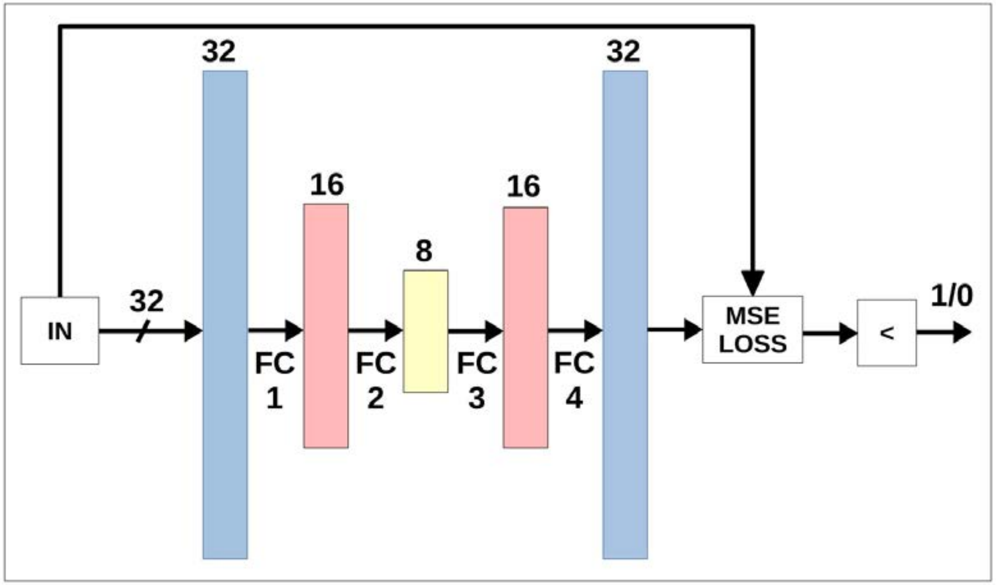
- Profiling VGG16 convolution layers



Could update BFP less regularly/move quantisation to FPGA

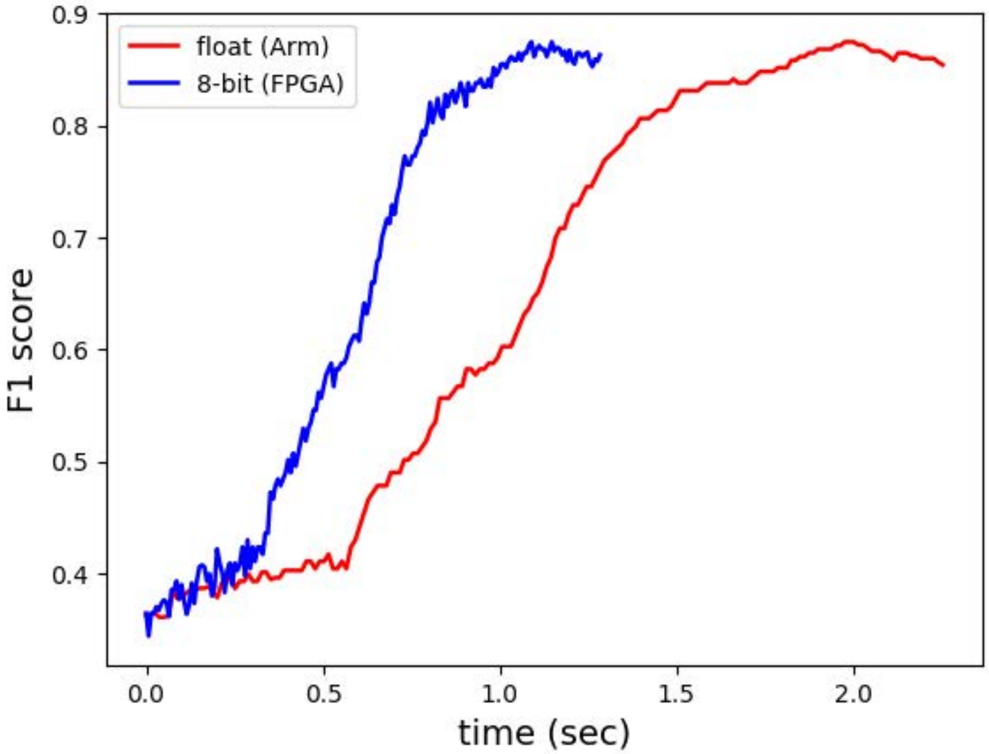
Case Study: RF Anomaly Detector

- Trained a small Autoencoder network in low-precision



Case Study: RF Anomaly Detector

- Injected known anomalies into the validation set, and observed the time taken to train the model.



Conclusion

- FPGAs can be used for training at the edge!
- We demonstrate:
- Low-precision training techniques for high accuracy
- FPGA prototype with software integration
- We have highlighted room for optimisation:
  - Move more computation to the FPGA
  - Optimise matrix multiplication
  - Lower precision
  - Versal?

https://phwl.github.io/talks

References

[1] Stephen Tridgell, Martin Kumm, Martin Hardieck, David Boland , Duncan Moss, Peter Zipf, and Philip H. W. Leong. Unrolling ternary neural networks. *ACM Transactions on Reconfigurable Technology and Systems*, page to appear (accepted 30 Aug 2019), 2019.

[2] Sean Fox, Julian Faraone, David Boland, Kees Vissers, and Philip H.W. Leong. Training deep neural networks in low-precision with high accuracy using FPGAs. In Proc. International Conference on Field Programmable Technology (FPT), to appear. 2019.