

Large-Scale FPGA implementations of Machine Learning Algorithms

Philip Leong (梁恆惠) | Computer Engineering Laboratory
School of Electrical and Information Engineering,
The University of Sydney

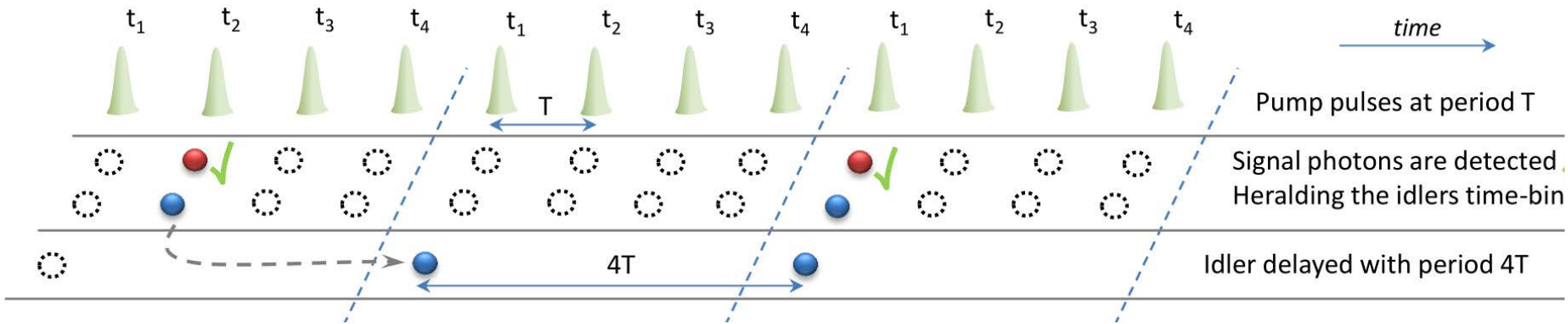


THE UNIVERSITY OF
SYDNEY

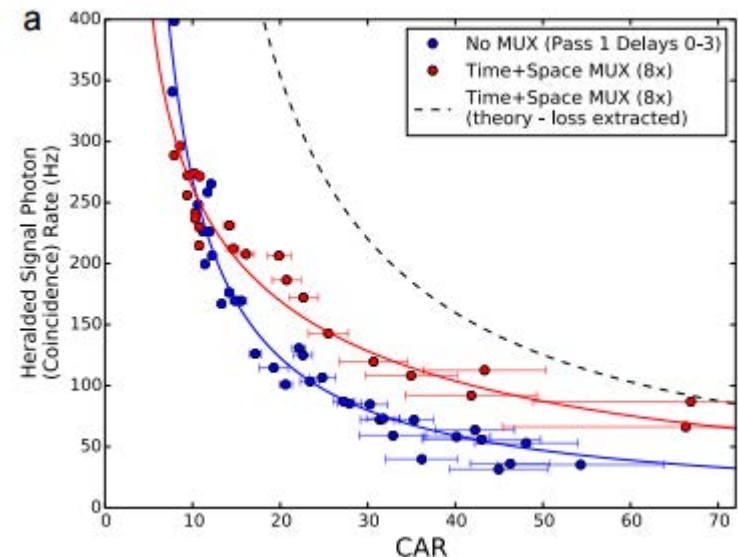
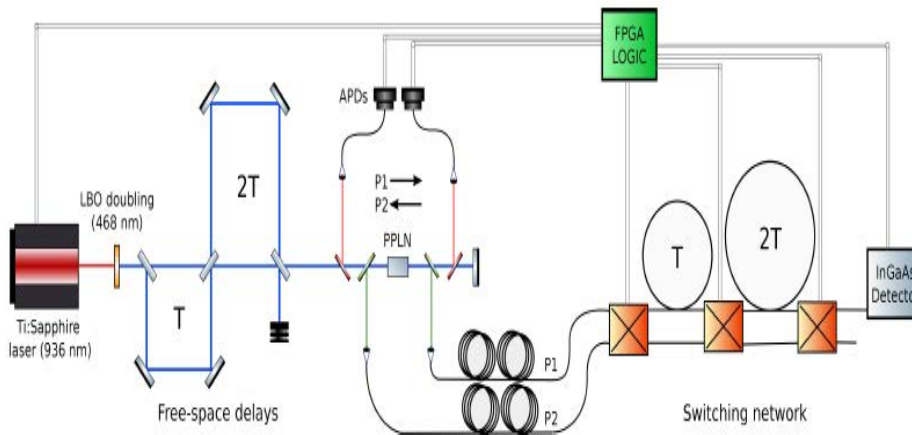
- › Focuses on how to use parallelism to solve demanding problems
 - Novel architectures, applications and design techniques using VLSI, FPGA and parallel computing technology
- › Research
 - Reconfigurable computing
 - Machine learning
 - Nanoscale interfaces



Time domain multiplexing of single photons



Initially expectation : Heralded single photon rate should enhance significantly without degrading coincidence to accidental ratio (CAR)





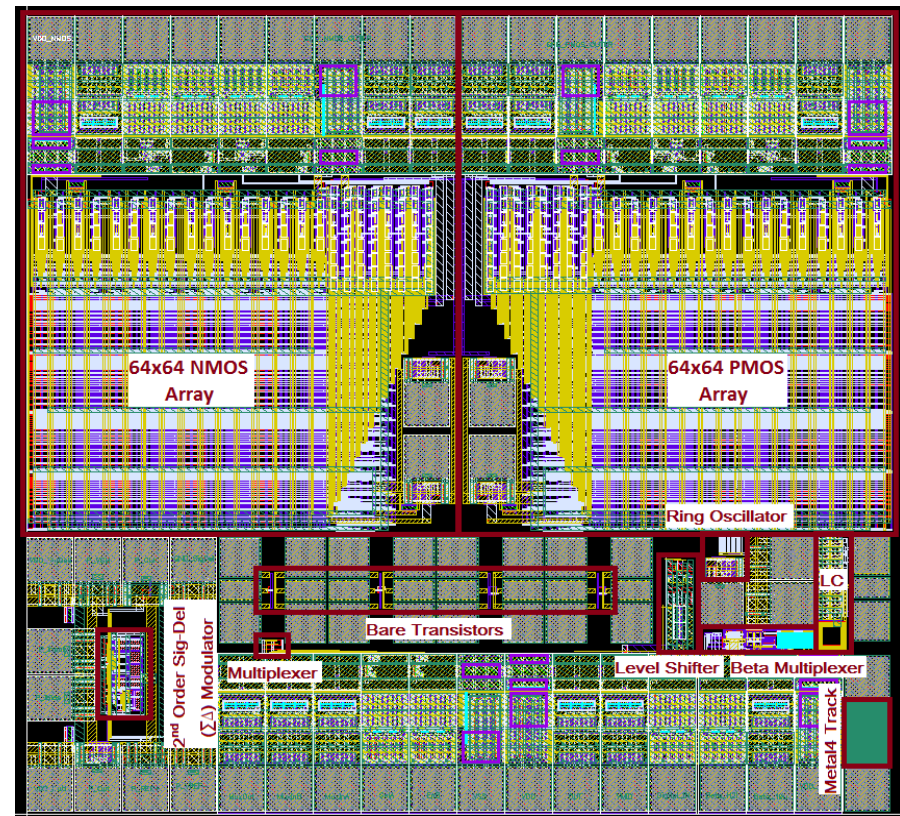
Time Multiplexing of Single Photons



Cool Transistors (0.35u CMOS C35B4C3)

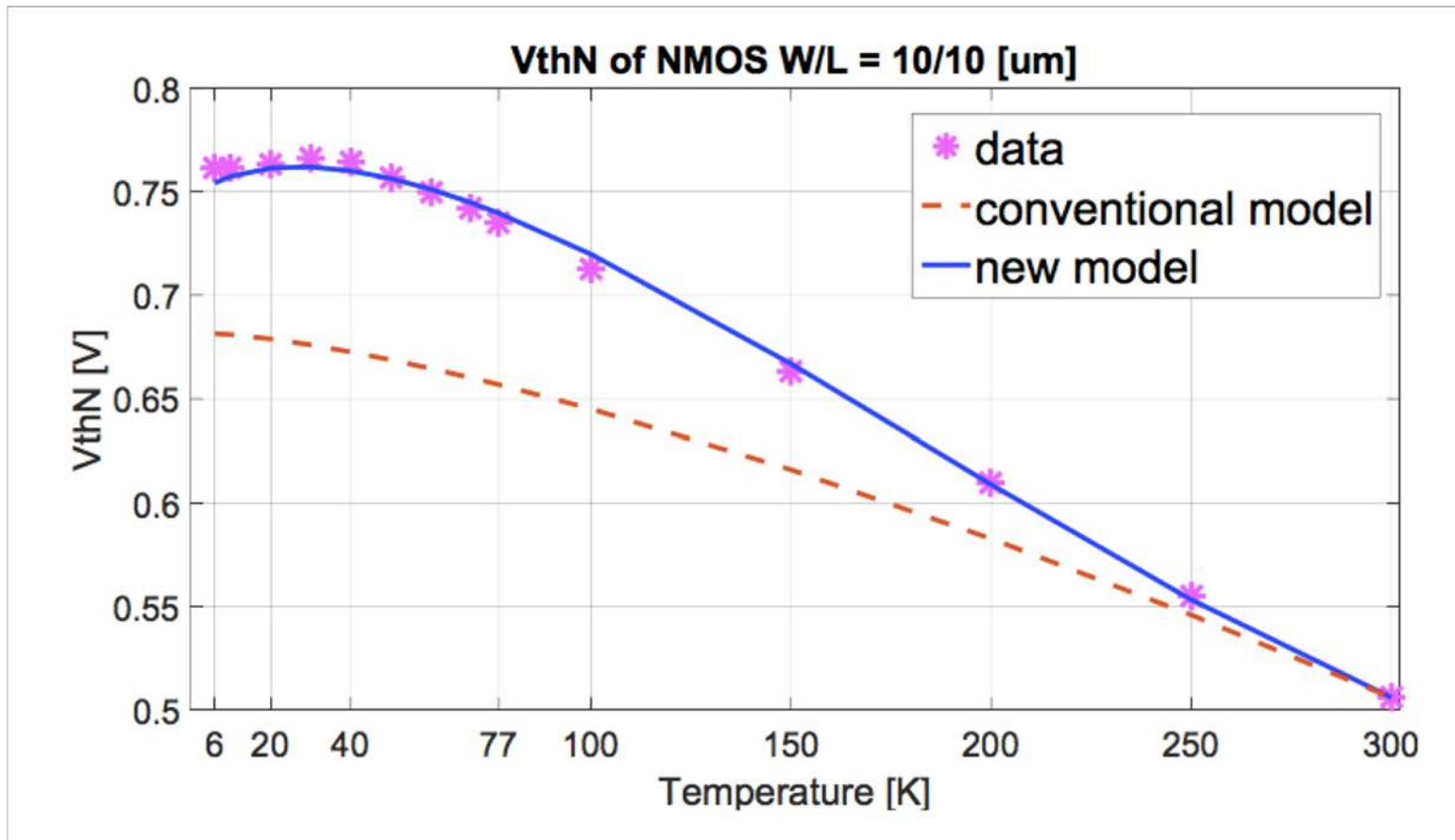
Purposes:

- To characterize CMOS transistors
- Evaluate matching property of CMOS transistors
- Test analog circuits: ADC, Level Shifter, Ring Oscillator, Beta Multiplier, Passive LC circuit, Metal tracks, ...



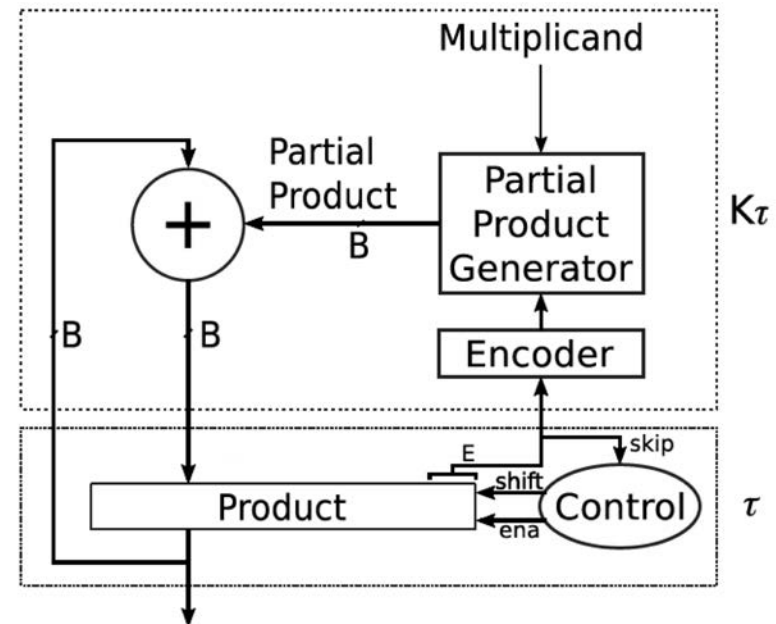
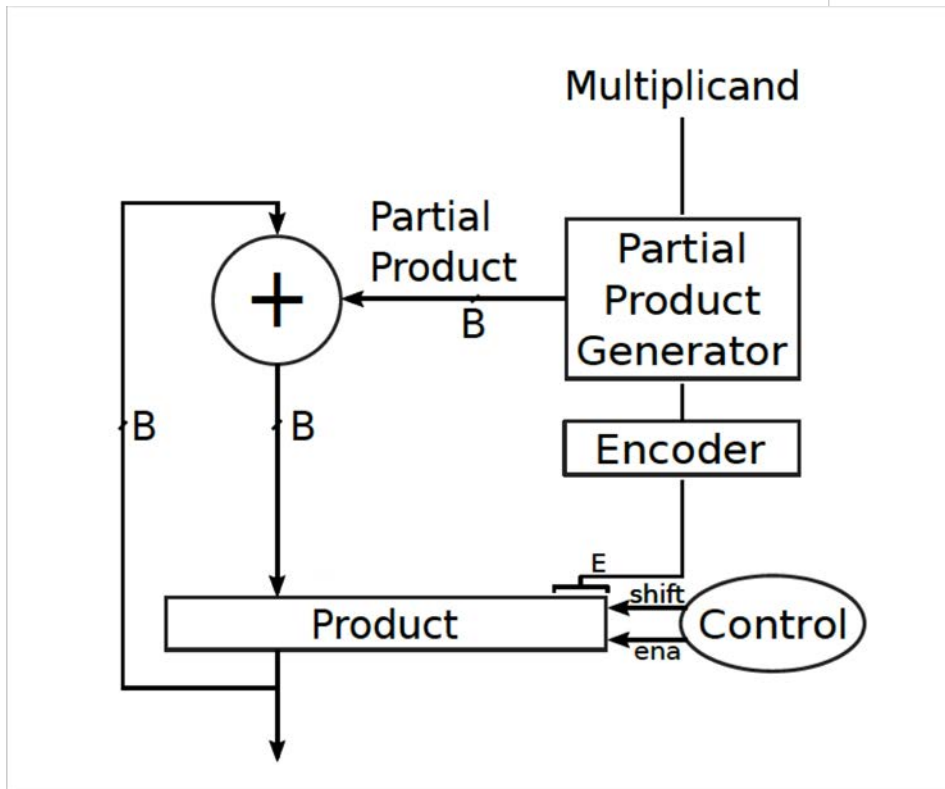
Layout of QNL2_CMOS

Wide-range Threshold Voltage Model



Modified Booth Radix-4 datapath is split into 2 sections, each with its own critical path

Non-zero encodings take $\bar{K}\tau$ and zero take τ



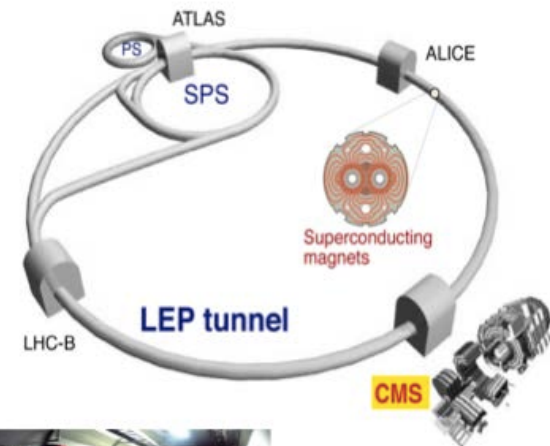
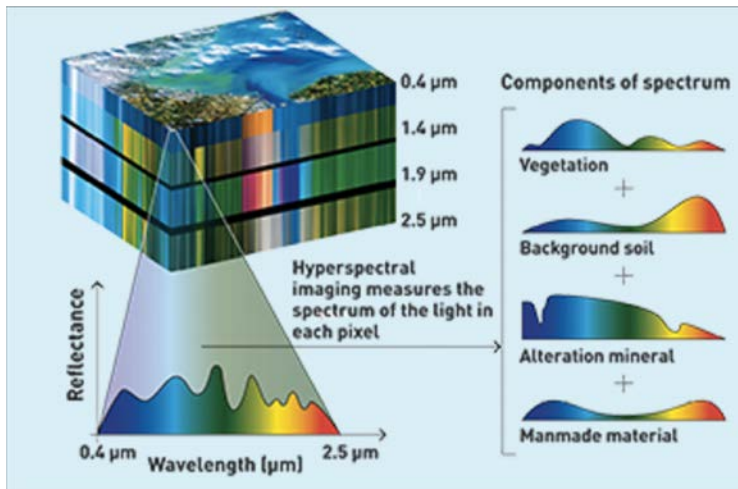
TVLSI (to appear)

- › FPGAs offer an opportunity to provide ML algorithms with higher throughput and lower latency through
 - **E**xploration– easily try different ideas to arrive at a good solution
 - **P**arallelism – so we can arrive at an answer faster
 - **I**ntegration – so interfaces are not a bottleneck
 - **C**ustomisation – problem-specific designs to improve efficiency
- › **Describe our work on implementations of ML that use these ideas**

- › **Exploration (Online kernel methods)**
- › **Parallelisation**
- › **Integration**
- › **Customisation**

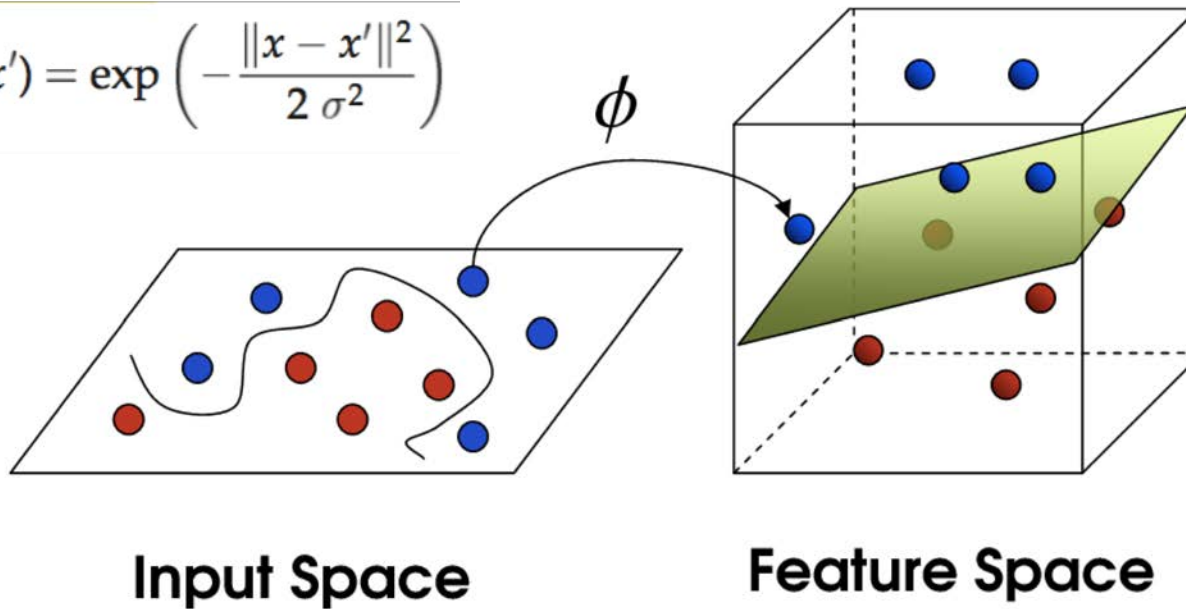
Challenges in measurement and control are becoming feasible

- › Significant improvements in ML algorithms but cannot keep up with sources e.g. hyperspectral imager or wireless transceiver
- › Need extremely high throughput
- › In control applications we need low latency e.g. triggering data collection in Large Hadron Collider
- › Need very low latency



Improvements in throughput and latency enable new applications!

$$k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$$



- › Choose high dimensional feature space (so easily separable)
- › Use kernel trick to avoid computing the mapping (fast)
- › Do regression/classification using

$$f(x_i) = \sum_{j=1}^N \alpha_j \kappa(x_i, v_j)$$

› Kernel is a similarity function

- defined by an implicit mapping ϕ , (original space to feature space)

$$\kappa(x, x') = \phi(x)^T \phi(x') = \langle \phi(x), \phi(x') \rangle$$

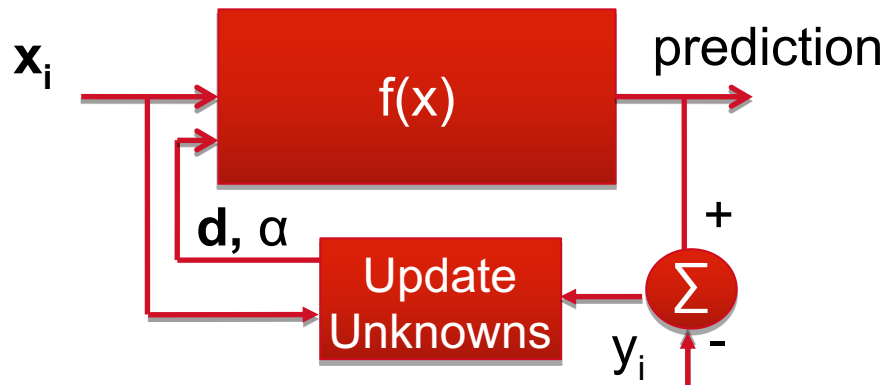
- e.g. Linear kernel $\kappa(x, x') = \langle x, x' \rangle$

- e.g. Polynomial kernel $\kappa(x, x') = (1 + \langle x, x' \rangle)^d$ for $d=2$: $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$


- e.g. Gaussian kernel (universal approximator) $k(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right)$
 - $\Phi(x)$ infinite in dimension!

› **Modify linear ML techniques to kernel ones by replacing dot products with the kernel function (kernel trick)**

- e.g. linear discriminant analysis, logistic regression, perceptron, SOM, K-means, PCA, ICA, LMS, RLS, ...
- **While we only describe prediction here, also applied to training equations**



$$f(x) = \sum_{i=1}^N \alpha_i \kappa(x, d_i)$$


 Dictionary Entry

› “Kernel Method” $\rightarrow \kappa(x, x') : \mathbb{R}^d \rightarrow \mathbb{R}^D$, where $D \gg d$

- › Dictionary \rightarrow subset of the input data of length \mathbf{N}
- › Computation and Memory scale $O(\mathbf{N}d)$
- › **BUT...** \mathbf{N} scales linearly with the dataset size

Exact Kernel Methods

$$f(x) = \sum_{i=1}^N \alpha_i \kappa(x, d_i)$$

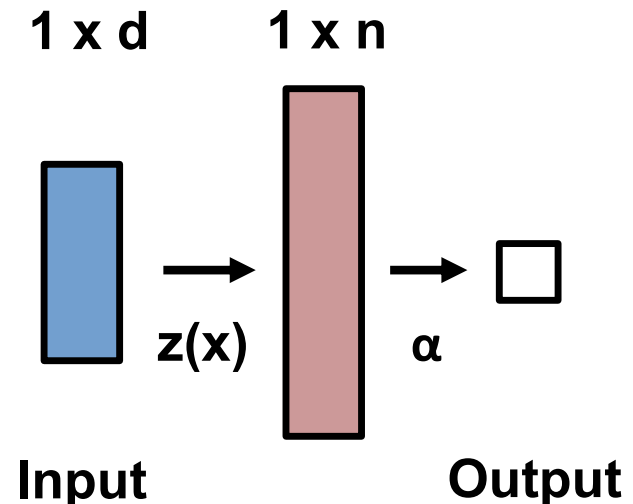
Random Kernel Expansion

$$f(x) = \sum_{i=1}^n \alpha_i z(x)$$
$$z(x) = \frac{1}{\sqrt{n}} \cos(\mathbf{W}x)$$

** Only for $k(x, x') = k(x - x', 0)$

Define $z(x)$:

- › Approximates $\kappa(x, x')$
- › MV + Non-Linear Activation (i.e. like Multilayer Perceptron)
- › W is **fixed** and **random**



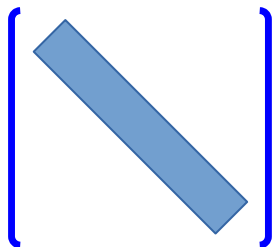
- › Computes $\mathbf{z}(\mathbf{x})$ efficiently by replacing $\mathbf{W}\mathbf{x}$ with combinations of random diagonal matrices and Hadamard transforms

$$z(x) = \frac{1}{\sqrt{n}} \cos(\mathbf{V}x), \quad \text{where } \mathbf{V}x = [\mathbf{Q}_1x, \mathbf{Q}_2x, \dots, \mathbf{Q}_hx]$$

$$\mathbf{Q}_jx = \mathbf{S}\mathbf{H}\mathbf{G}\mathbf{P}\mathbf{H}\mathbf{B}x$$

** Each \mathbf{Q}_jx is an independent $d \times d$ transform

B, G, S



Memory = $O(3n)$

P

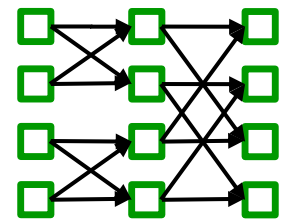
$$\begin{bmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

H

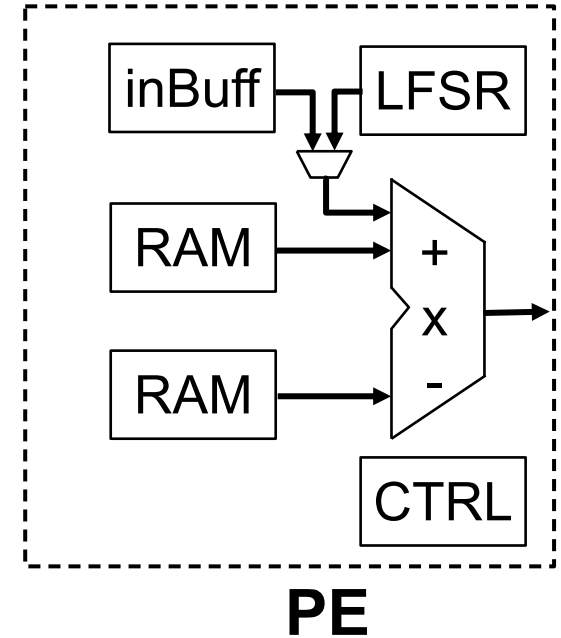
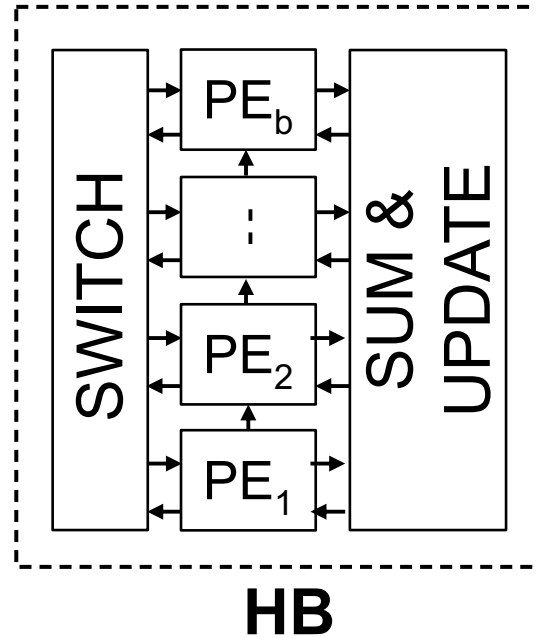
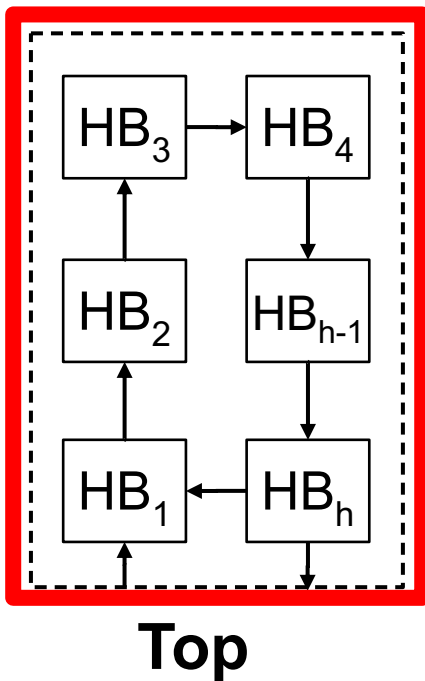
$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

Time = $O(n \log d)$

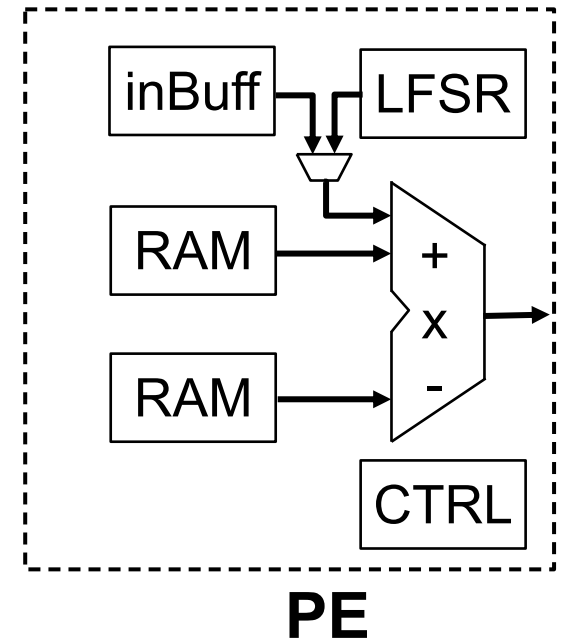
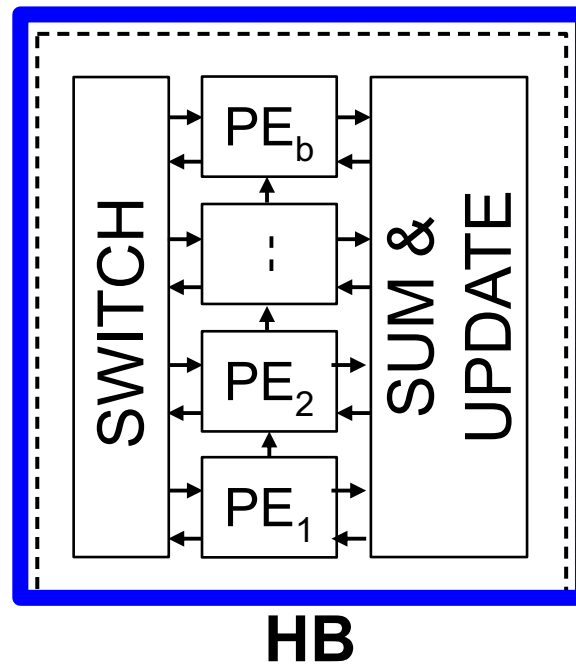
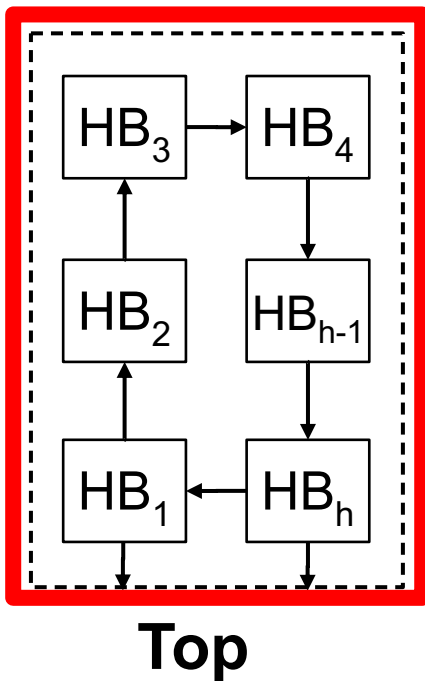
Fast Hadamard Transform ($d \times d$)



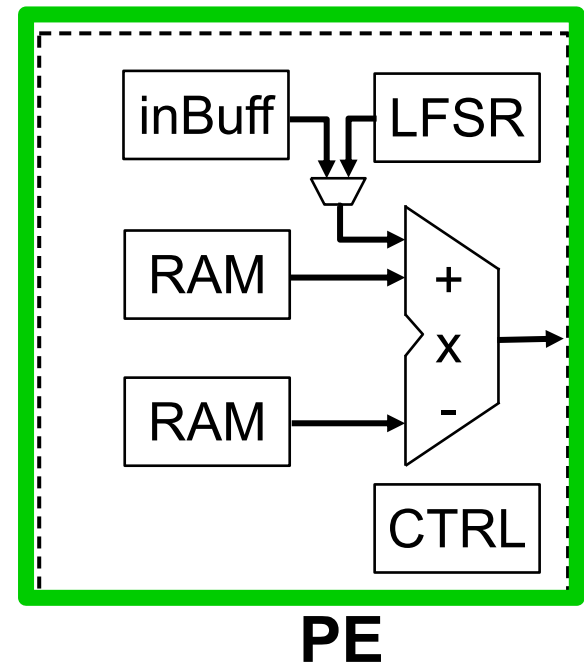
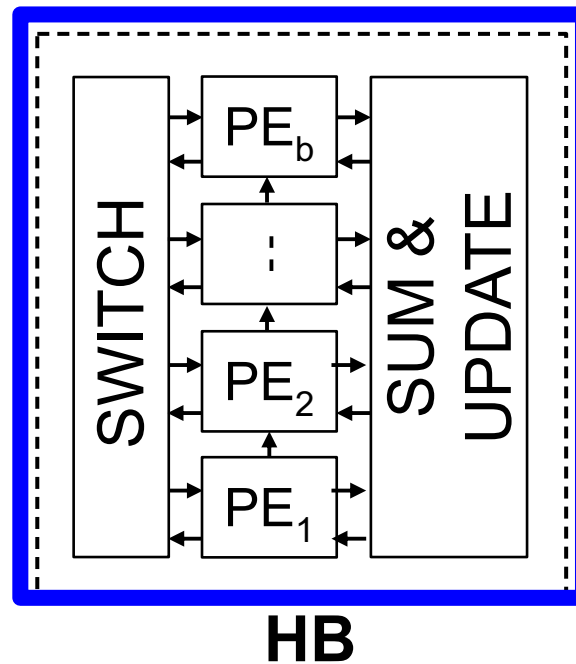
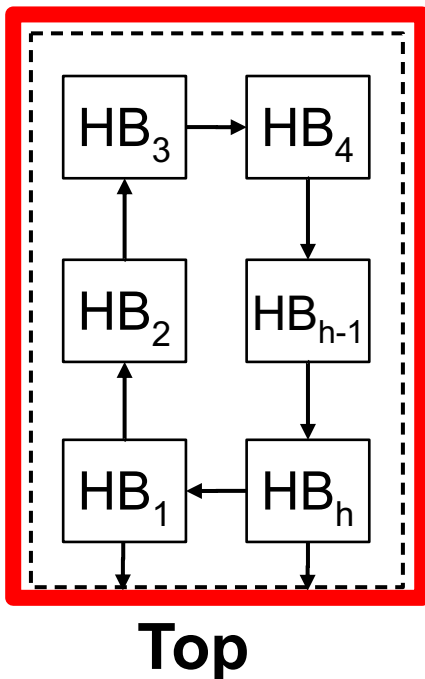
> $V_X = [Q_1x, Q_2x, \dots, Q_hx]$



- > $V_X = [Q_1x, Q_2x, \dots, Q_hx]$
- > Block of **b** PEs (i.e. Q_qx)



- › $V_x = [Q_1x, Q_2x, \dots, Q_hx]$
- › Block of b PEs (i.e. Q_qx)
- › General PE: 18-bit ALU, RAMs, Control Unit, LFSR



Impl.	dim.	n	bw	Lat. (cyc)	Fmax (MHz)	Exec (ns)	Th.put (Gb/s)
NORMA (V7, '15)	8	200	18	10	127	7.87	18.3
KNLMS (V7, '15)	8	16	32	207	314	3.18	80.4
CPU (Le et. '13)	1024	16.4k	32			58e4	0.06
FASTFOOD (V7)	1024	16.4k	18	1893	432	23.7e3	7.77
FASTFOOD (KU035)	8192	90.1k	18	16930	508	17.2e3	8.57

Impl.	dim.	n	bw	Lat. (cyc)	Fmax (MHz)	Exec (ns)	Th.put (Gb/s)
NORMA (V7, '15)	8	200	18	10	127	7.87	18.3
KNLMS (V7, '15)	8	16	32	207	314	3.18	80.4
CPU (Le et. '13)	1024	16.4k	32			58e4	0.06
FASTFOOD (V7)	1024	16.4k	18	1893	432	23.7e3	7.77
FASTFOOD (KU035)	8192	90.1k	18	16930	508	17.2e3	8.57

- › Supports much larger problems

Impl.	dim.	n	bw	Lat. (cyc)	Fmax (MHz)	Exec (ns)	Th.put (Gb/s)
Braiding (V7, '15)	8	200	18	10	127	7.87	18.3
KNLMS (V7, '15)	8	16	32	207	314	3.18	80.4
CPU (Le et. '13)	1024	16.4k	32			58e4	0.06
FASTFOOD (V7)	1024	16.4k	18	1893	432	23.7e3	7.77
FASTFOOD (KU035)	8192	90.1k	18	16930	508	17.2e3	8.57

- › Supports much larger problems
- › High speed design

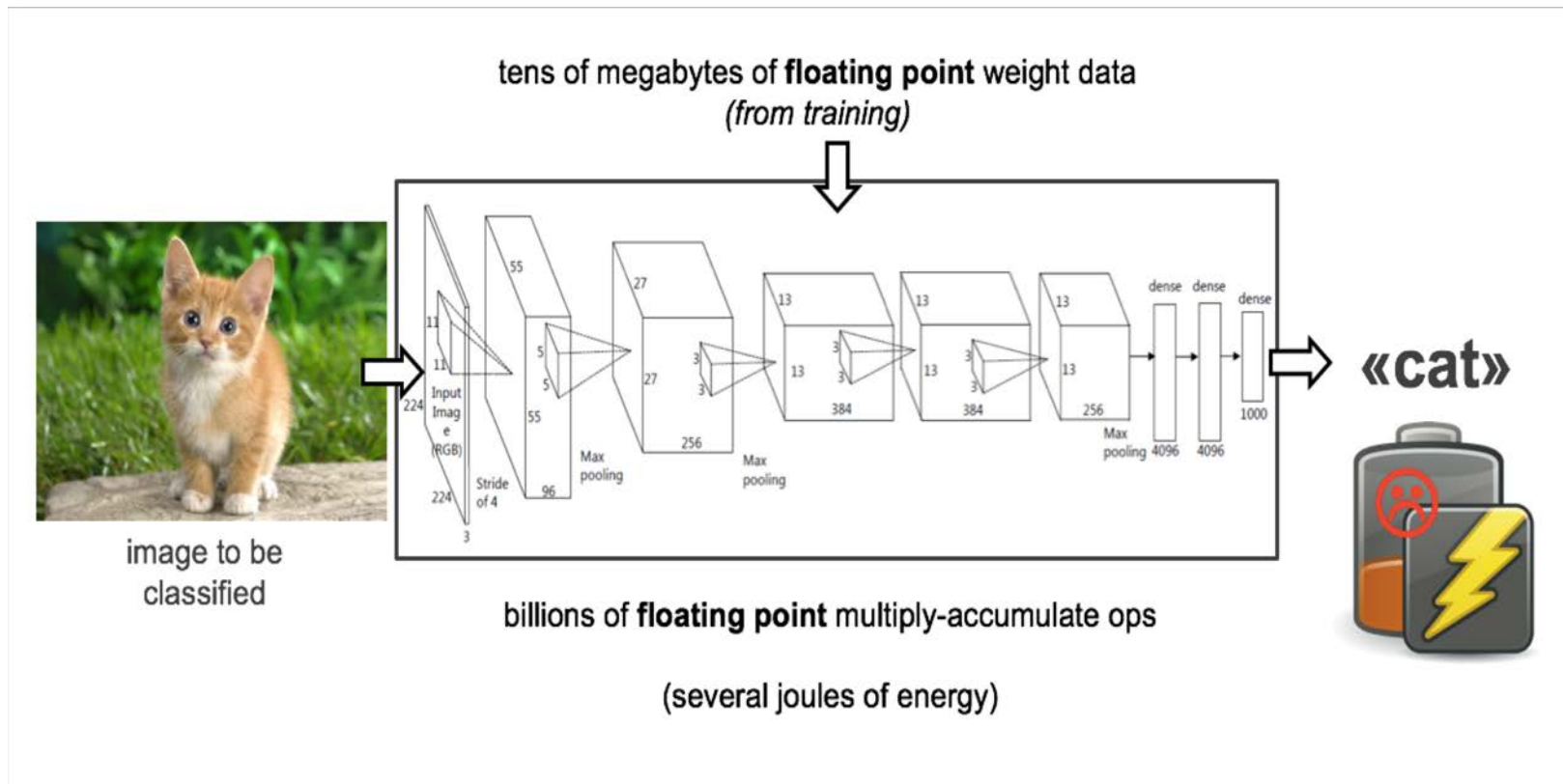
Impl.	dim.	n	bw	Lat. (cyc)	Fmax (MHz)	Exec (ns)	Th.put (Gb/s)
Braiding (V7, '15)	8	200	18	10	127	7.87	18.3
KNLMS (V7, '15)	8	16	32	207	314	3.18	80.4
CPU (Le et. '13)	1024	16.4k	32			58e4	0.06
FASTFOOD (V7)	1024	16.4k	18	1893	432	23.7e3	7.77
FASTFOOD (KU035)	8192	90.1k	18	16930	508	17.2e3	8.57

- › Supports much larger problems
- › High speed design
- › **245x** speed-up over a CPU

- › **Exploration**
- › **Parallelisation (Low Precision Neural Network)**
- › **Integration**
- › **Customisation**

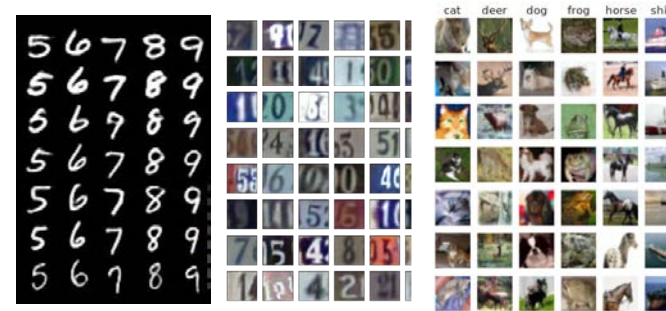
Inference with Convolutional Neural Networks

Slides from Yaman Umuroglu et. al., “FINN: A framework for fast, scalable binarized neural network inference,” FPGA’17



- > The extreme case of quantization
 - Permit only two values: +1 and -1
 - Binary weights, binary activations
 - Trained from scratch, not truncated FP

- > Courbariaux and Hubara et al. (NIPS 2016)
 - Competitive results on three smaller benchmarks
 - Open source training flow
 - Standard “deep learning” layers
 - Convolutions, max pooling, batch norm, fully connected...



	MNIST	SVHN	CIFAR-10
Binary weights & activations	0.96%	2.53%	10.15%
FP weights & activations	0.94%	1.69%	7.62%
BNN accuracy loss	-0.2%	-0.84%	-2.53%

% classification error (lower is better)

Vivado HLS estimates on Xilinx UltraScale+ MPSoC ZU19EG

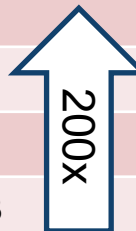
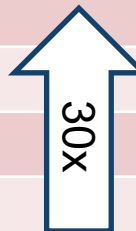
› *Much* smaller datapaths

- Multiply becomes XNOR, addition becomes popcount
- No DSPs needed, everything in LUTs
- Lower cost per op = more ops every cycle

› *Much* smaller weights

- Large networks can fit entirely into on-chip memory (OCM)
- More bandwidth, less energy compared to off-chip

Precision	Peak TOPS	On-chip weights
1b	~66	~70 M
8b	~4	~10 M
16b	~1	~5 M
32b	~0.3	~2 M

› **fast** inference with **large BNNs**

	Accuracy	FPS	Power (chip)	Power (wall)	kFPS / Watt (chip)	kFPS / Watt (wall)	Precision	
FINN	MNIST, SFC-max	95.8%	12.3 M	7.3 W	21.2 W	1693	583	1
	MNIST, LFC-max	98.4%	1.5 M	8.8 W	22.6 W	177	269	1
	CIFAR-10, CNV-max	80.1%	21.9 k	3.6 W	11.7 W	6	2	1
	SVHN, CNV-max	94.9%	21.9 k	3.6 W	11.7 W	6	2	1
Prior Work	MNIST, Alemdar et al.	97.8%	255.1 k	0.3 W	-	806	-	2
	CIFAR-10, TrueNorth	83.4%	1.2 k	0.2 W	-	6	-	1
	SVHN, TrueNorth	96.7%	2.5 k	0.3 W	-	10	-	1

Max accuracy loss: ~3%

10 – 100x better performance

CIFAR-10/SVHN energy efficiency comparable to TrueNorth ASIC

- › Who would be willing to incur a loss in accuracy?
- › Can we get better accuracy with a little more hardware?

- To compute quantised weights from FP weights

$$\mathbf{Q}_l = \text{sign}(\mathbf{W}_l) \odot \mathbf{M}_l$$

with,

$$M_{l,i,j} = \begin{cases} 1 & \text{if } |W_{l,i,j}| \geq \eta_l \\ 0 & \text{if } -\eta_l < W_{l,i,j} < \eta_l \end{cases}$$

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

where \mathbf{M} represents a masking matrix, η is the quantization threshold hyperparameter (0 for binarised)

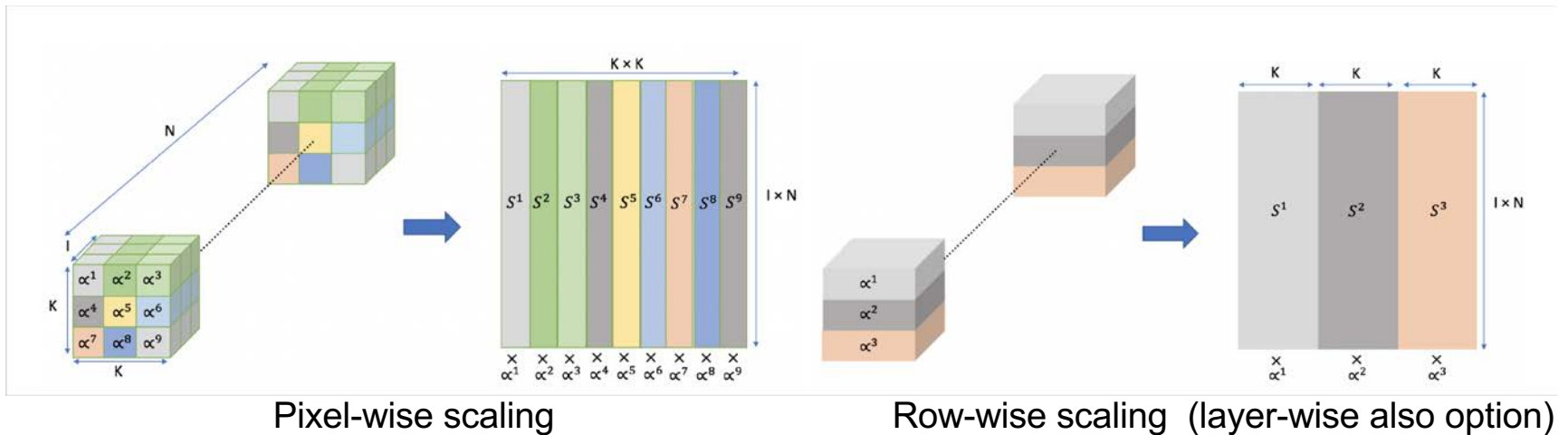
- Make approximation $W_l \approx \alpha_l Q_l$, $Q_l \in \mathcal{C}$
- \mathcal{C} is the codebook, $\mathcal{C} \in \{\mathcal{C}_1, \mathcal{C}_2, \dots\}$ e.g. $\mathcal{C} = \{-1, +1\}$ for binary, $\mathcal{C} = \{-1, 0, +1\}$ for ternary
- A diagonal matrix α_l is defined by the vector $\alpha_l = [\alpha_l^1, \dots, \alpha_l^m]$:

$$\alpha = \text{diag}(\alpha) := \begin{bmatrix} \alpha^1 & 0 & .. & 0 & 0 \\ 0 & \alpha^2 & .. & : & 0 \\ : & : & .. & \alpha^{m-1} & : \\ 0 & 0 & .. & 0 & \alpha^m \end{bmatrix}$$

- Train by solving

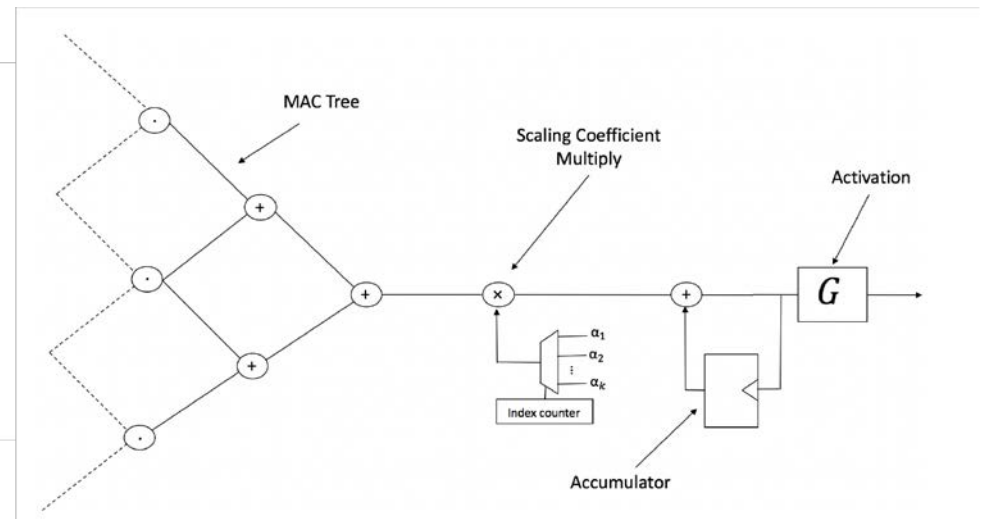
$$\alpha_l = \underset{\alpha}{\operatorname{argmin}} E(\alpha, \mathbf{Q}) \quad \text{s.t.} \quad \alpha \geq 0, \mathbf{Q}_{l_{i,j}} \in \mathcal{C}$$

- More fine-grained quantisation can improve approximation of weights



- › For K filters, I Input feature maps of dimension $F \times F$, N output feature maps
- › $P = K^2 I N F^2$

Method	Scalars	Ops
Layer (DoReFa)	1	P
Row (SYQ)	K	P
Pixel (SYQ)	K^2	P
Asymmetric (TTQ)	2	$P + Z$
Grouping (FGQ)	$K^2 N / 4$	P
Channel (HWGQ/BWN)	N	P



- › Full precision for 1st and last layers, CONV layers pixel-wise, FC layer-wise

Model		1-8	2-8	Baseline	Reference
AlexNet	Top-1	56.6	58.1	56.6	57.1
	Top-5	79.4	80.8	80.2	80.2
VGG	Top-1	66.2	68.7	69.4	-
	Top-5	87.0	88.5	89.1	-
ResNet-18	Top-1	62.9	67.7	69.1	69.6
	Top-5	84.6	87.8	89.0	89.2
ResNet-34	Top-1	67.0	70.8	71.3	73.3
	Top-5	87.6	89.8	89.1	91.3
ResNet-50	Top-1	70.6	72.3	76.0	76.0
	Top-5	89.6	90.9	93.0	93.0

Baseline is floating-point, reference <https://github.com/facebook/fb.resnet.torch> (ResNet) and <https://github.com/BVLC/caffe> (AlexNet)

Model	Weights	Act.	Top-1	Top-5
DoReFa-Net [33]	1	2	49.8	-
QNN [15]	1	2	51.0	73.7
HWGQ [2]	1	2	52.7	76.3
SYQ	1	2	55.4	78.6
DoReFa-Net [33]	1	4	53.0	-
SYQ	1	4	56.2	79.4
BWN [24]	1	32	56.8	79.4
SYQ	1	8	56.6	79.4
SYQ	2	2	55.8	79.2
FGQ [21]	2	8	49.04	-
TTQ [34]	2	32	57.5	79.7
SYQ	2	8	58.1	80.8

Model	Weights	Act.	Top-1	Top-5
BWN [24]	1	32	60.8	83.0
SYQ	1	8	62.9	84.6
TWN [19]	2	32	65.3	86.2
INQ [32]	2	32	66.0	87.1
TTQ [34]	2	32	66.6	87.2
SYQ	2	8	67.7	87.8

ResNet-18

Model	Weights	Act.	Top-1	Top-5
HWGQ [2]	1	2	64.6	85.9
SYQ	1	4	68.8	88.7
SYQ	1	8	70.6	89.6
FGQ [21]	2	4	68.4	-
SYQ	2	4	70.9	90.2
FGQ [21]	2	8	70.8	-
SYQ	2	8	72.3	90.9

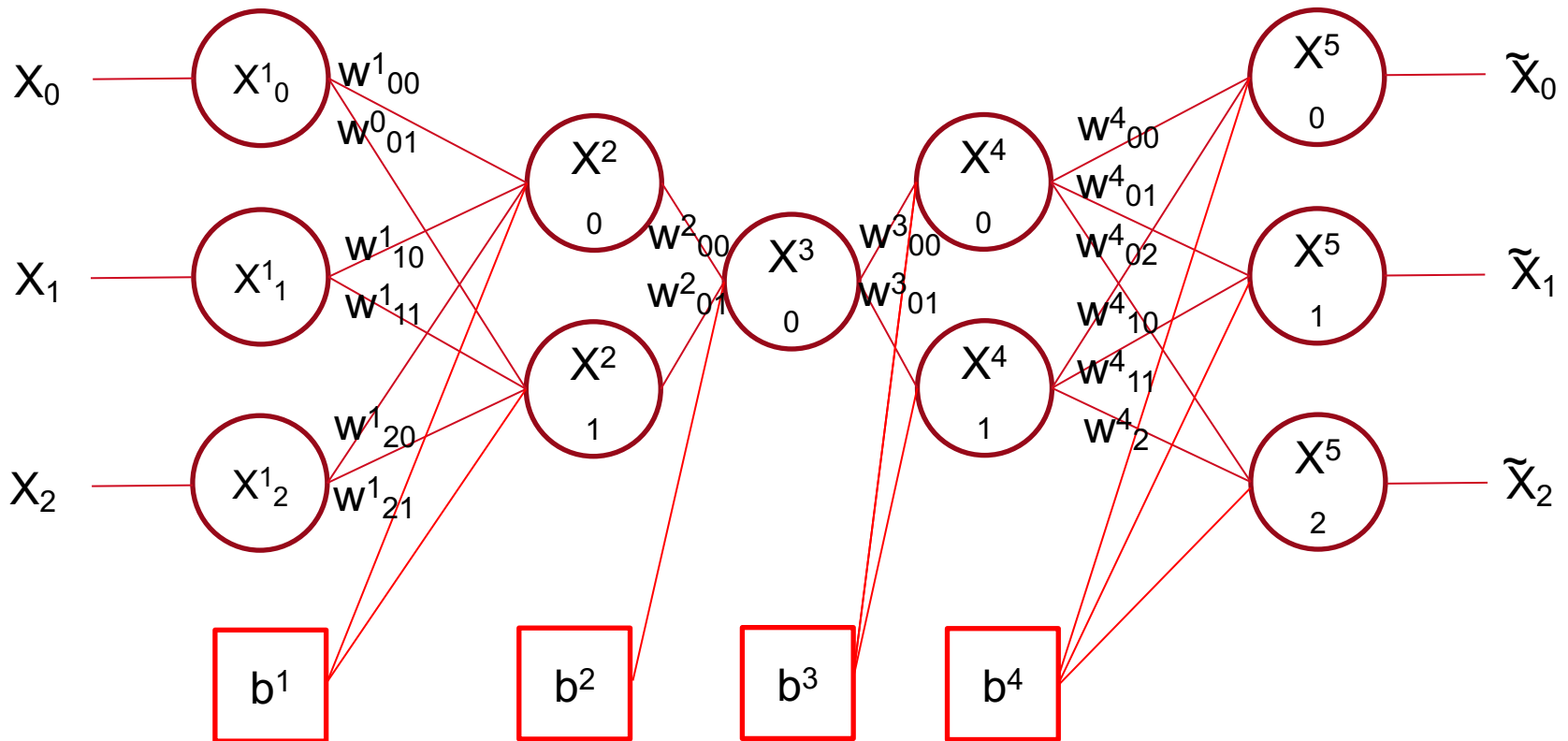
ResNet-50

- › **Exploration**
- › **Parallelisation**
- › **Integration (radio frequency machine learning)**
- › **Customisation**

- › Processing radio frequency signals remains a challenge
 - high bandwidth and low latency difficult to achieve
- › Autoencoder to do anomaly detection

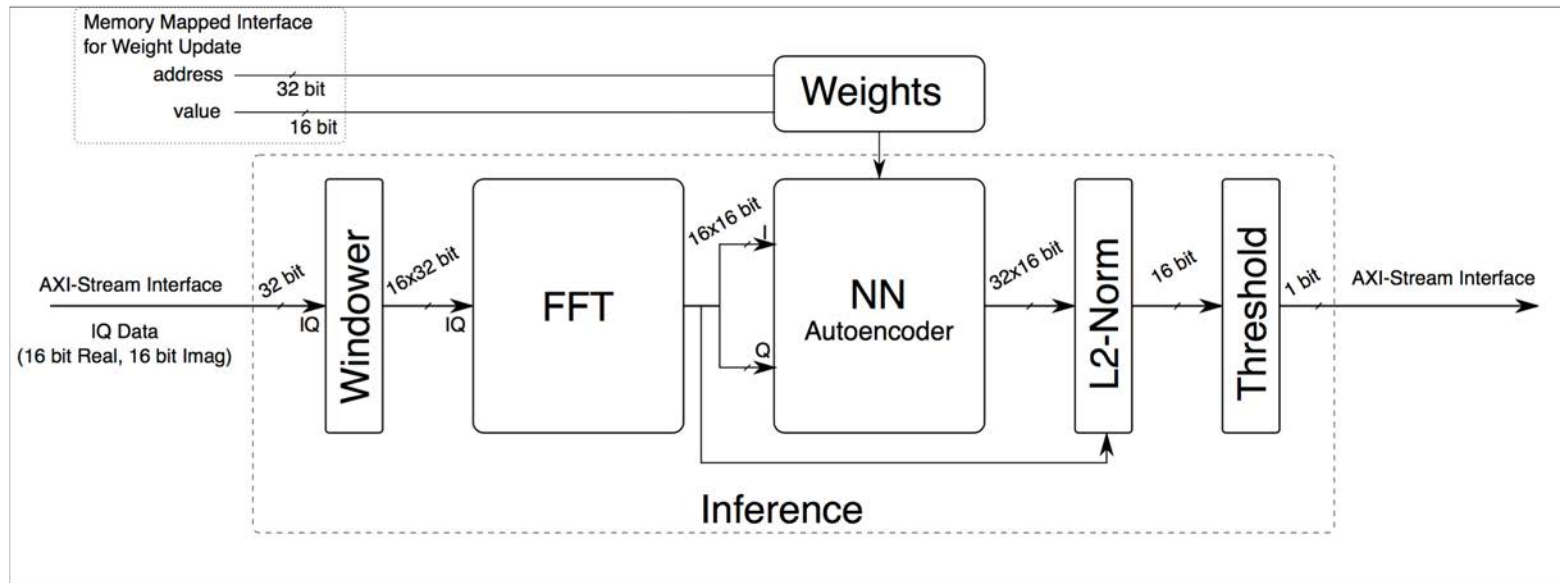


Train so $\tilde{x} \approx x$ (done in an unsupervised manner)

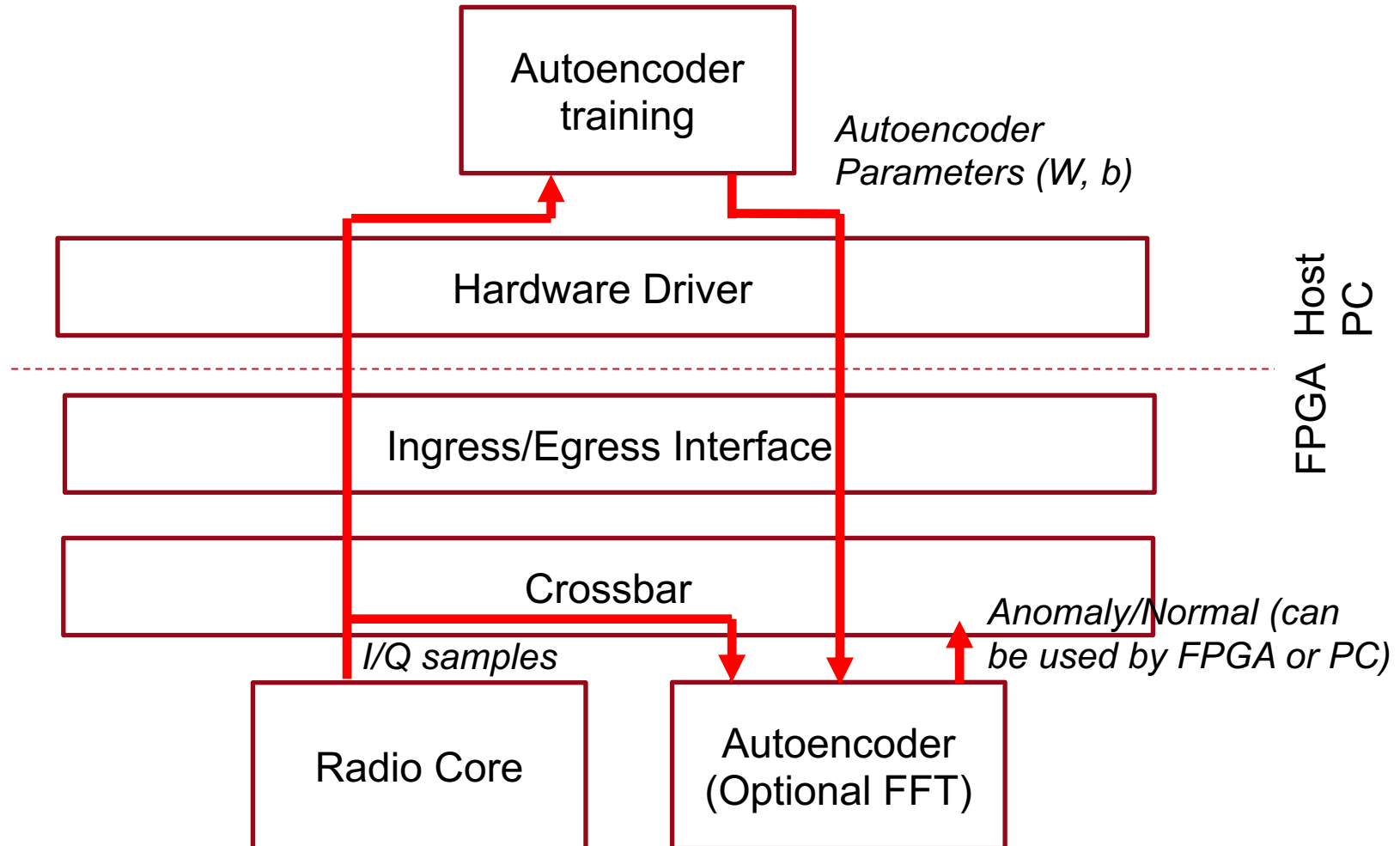


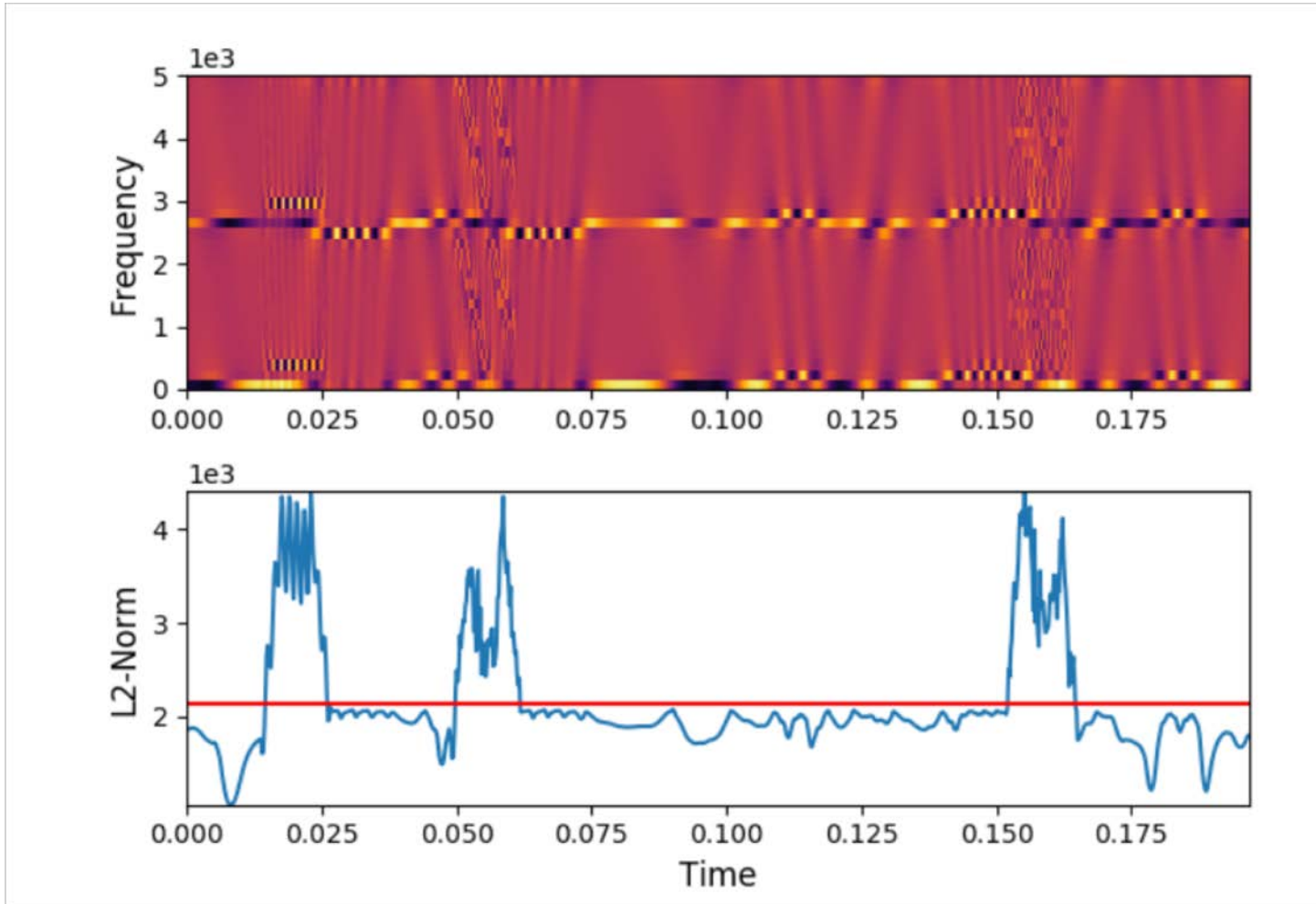
Autoencoder learns “normal” representation

- › Anomaly if distance between autoencoder output and input large
- › FPGA has sufficiently high performance to process each sample of waveform at 200 MHz!
 - This minimises latency and maximises throughput
 - Weights trained on uP and updated on FPGA without affecting inference



Implemented on Ettus X310 platform





Typical SDR latency >> 1 ms

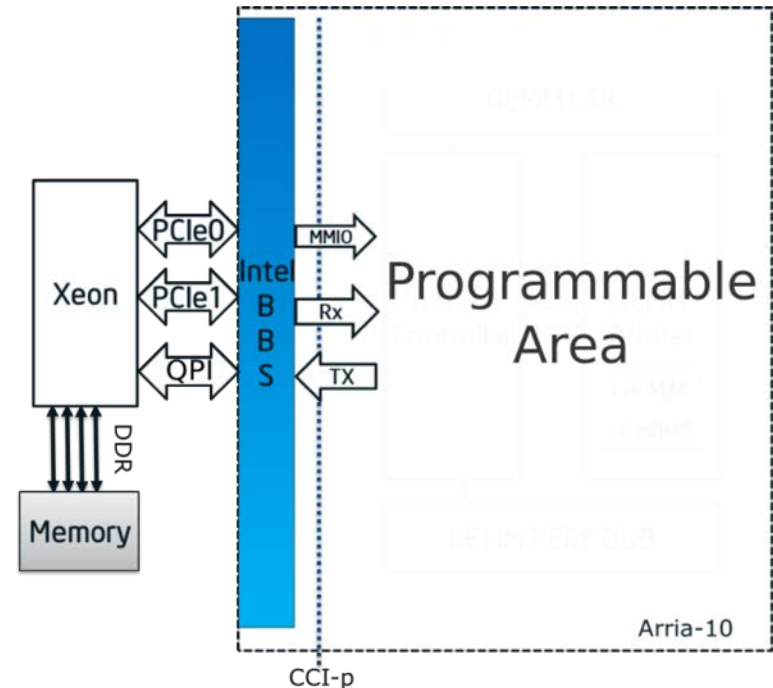
Module	II	Latency (cycles)	BRAM	DSP	FF	LUT
Windower	1	0	0	0	1511	996
FFT	1	8	0	48	4698	2796
NN	1	17	4	1280	213436	13044
L_2 -Norm	1	4	0	32	1482	873
Thres	1	0	0	0	3	21
Weight Update	258	257	0	0	21955	4528
Inference (FFT+NN)	1	37	1068	1360	241522	45448
Inference (NN)	1	29	1068	1312	236824	42652
Total	N/A	N/A	1068	1360	263477	49976
Total Util.	N/A	N/A	67%	88%	51%	19%

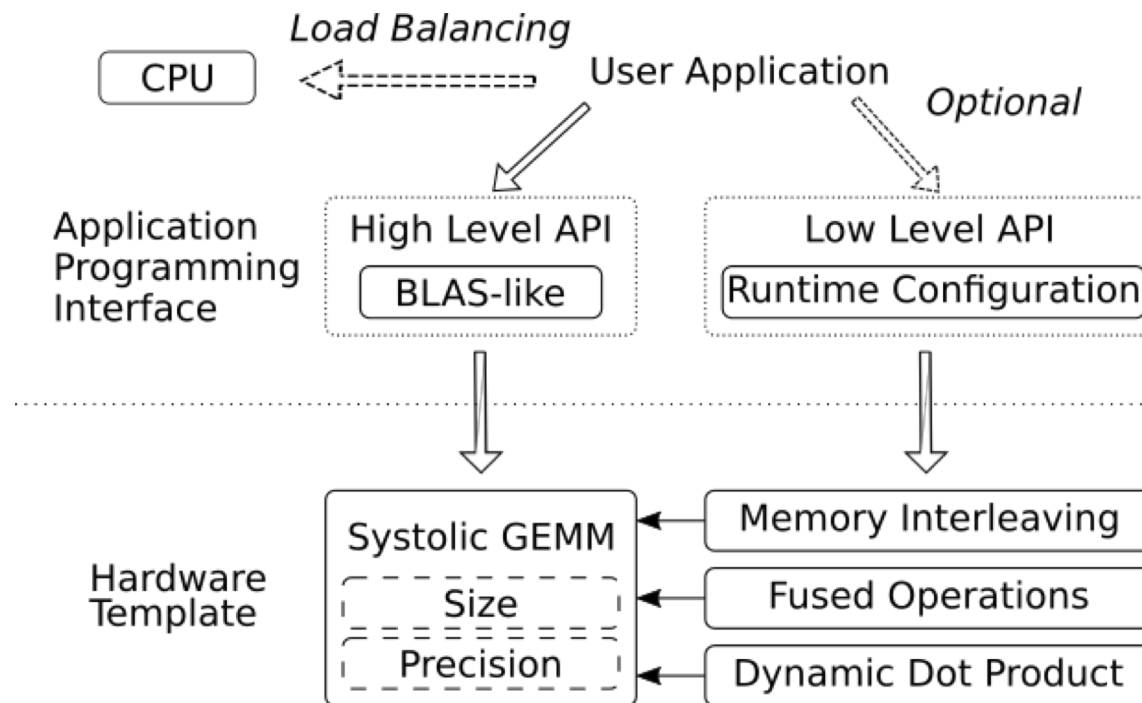
Operation	Throughput	Latency
Inference(FFT+NN)	5ns	185ns
Inference(NN)	5ns	105ns
Weight Update	1290ns	1285ns

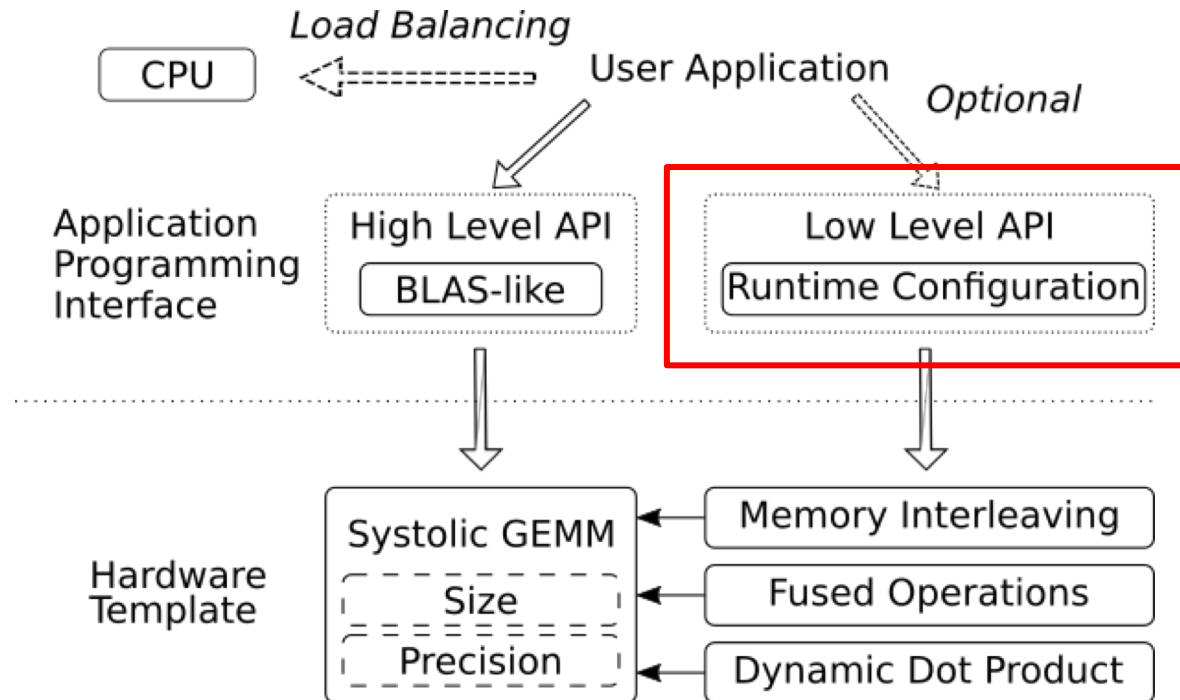
- › Exploration
- › Parallelisation
- › Integration
- › **Customisation (Matrix Multiplication on Intel Harp v2)**

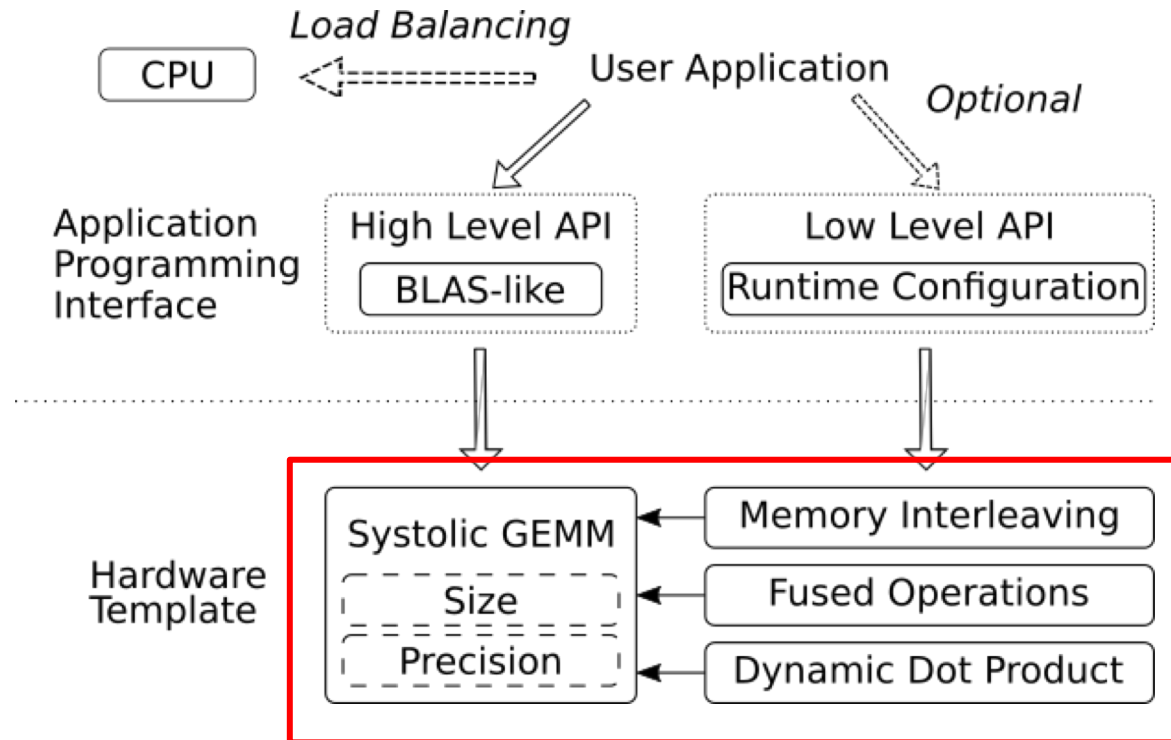
$$C = \alpha * op(A) * op(B) + \beta * C$$

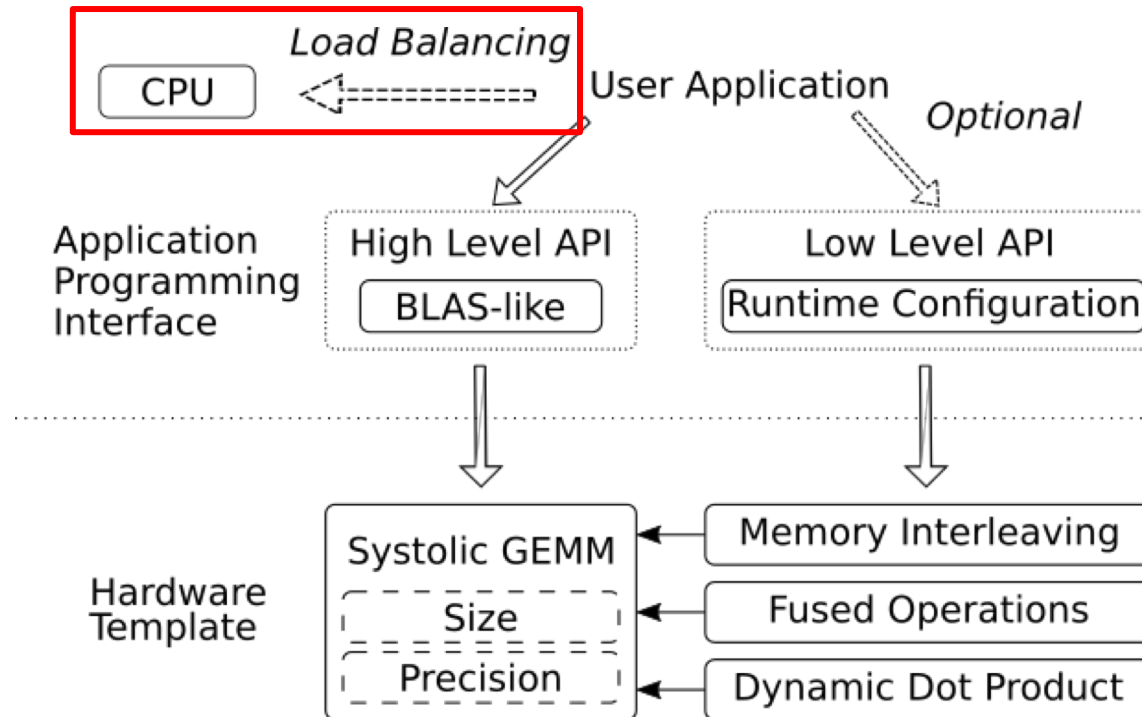
- › Xeon+FPGA
- › Simple, software-based interface
- › Extensions to efficiently support Machine Learning

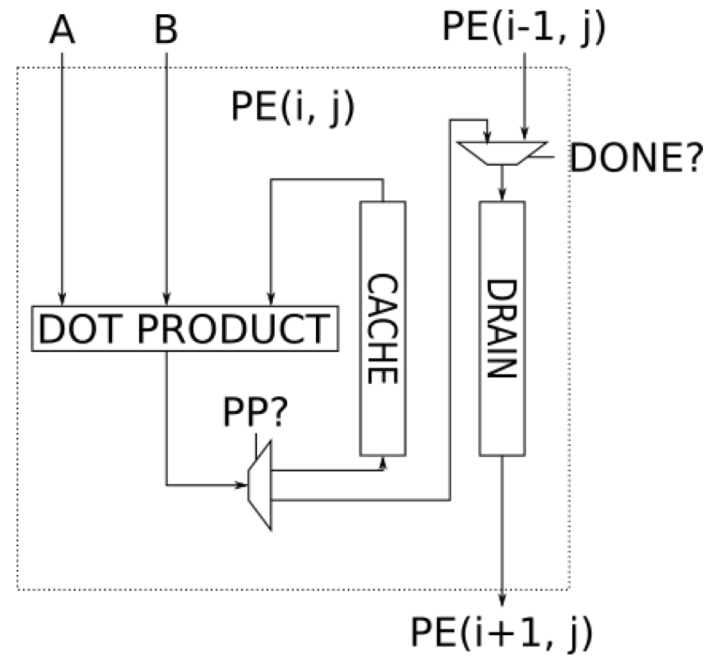
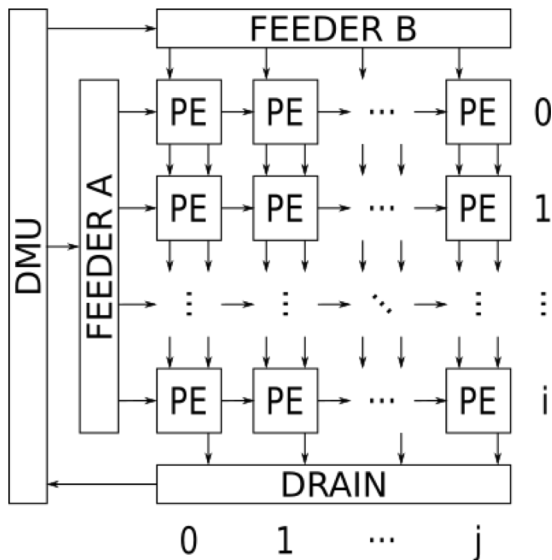




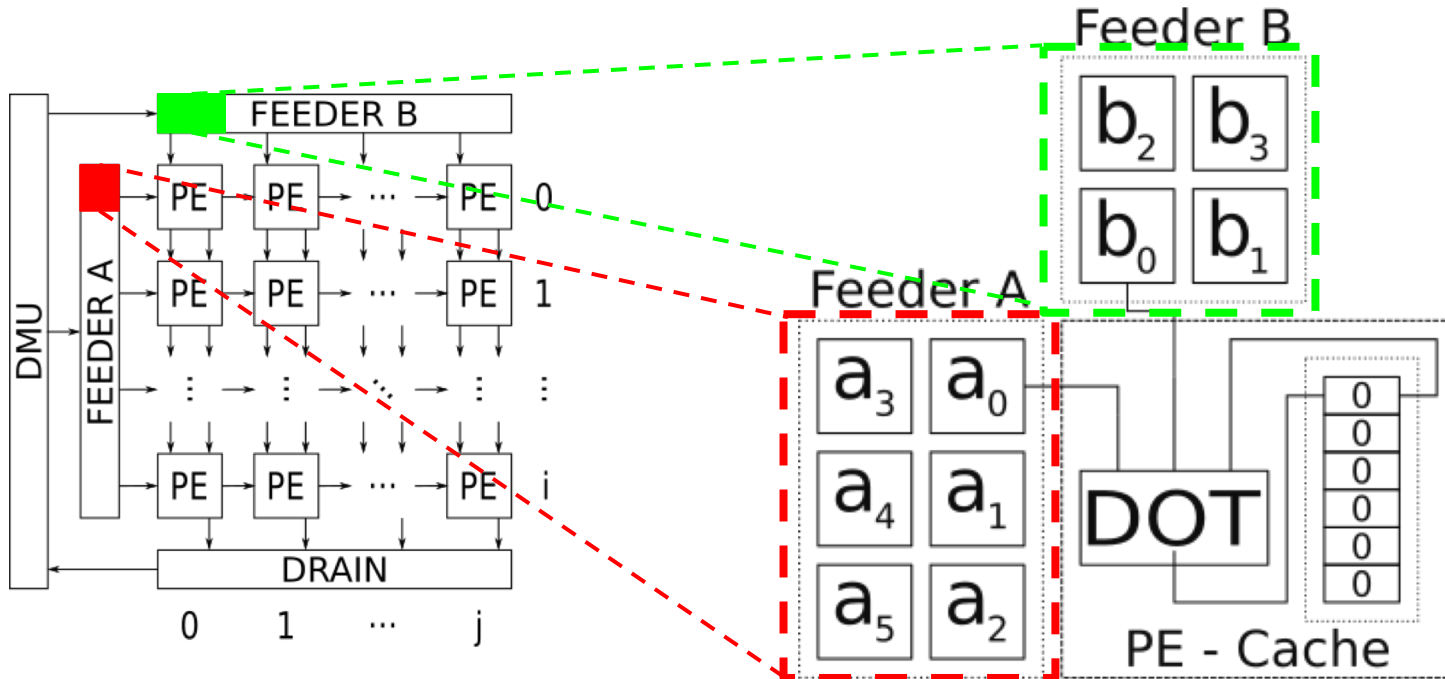








FP32, INT16, INT8, INT4, Ternary, Binary





Memory Interleaving

$$\begin{array}{cccc|cccc}
 a_{00} & a_{01} & a_{02} & a_{03} & 0 & 0 & b_{00} & b_{01} & b_{02} & b_{03} & 0 & 0 \\
 a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 & b_{10} & b_{11} & b_{12} & b_{13} & 0 & 0 \\
 a_{20} & a_{21} & a_{22} & a_{23} & 0 & 0 & b_{20} & b_{21} & b_{22} & b_{23} & 0 & 0 \\
 \hline
 a_{30} & a_{31} & a_{32} & a_{33} & 0 & 0 & b_{30} & b_{31} & b_{32} & b_{33} & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 \times$$

Inefficient with Static Partitioning

$$\begin{array}{cc|cc}
 a_{00} & a_{01} & a_{02} & a_{03} \\
 a_{10} & a_{11} & a_{12} & a_{13} \\
 \hline
 a_{20} & a_{21} & a_{22} & a_{23} \\
 a_{30} & a_{31} & a_{32} & a_{33}
 \end{array}
 \times
 \begin{array}{cc|cc}
 b_{00} & b_{01} & b_{02} & b_{03} \\
 b_{10} & b_{11} & b_{12} & b_{13} \\
 \hline
 b_{20} & b_{21} & b_{22} & b_{23} \\
 b_{30} & b_{31} & b_{32} & b_{33}
 \end{array}$$

Fewer Computations Increase Bandwidth

Memory Sharing between Feeder A and Feeder B

$$\begin{array}{cccc|cc}
 a_{00} & a_{01} & a_{02} & a_{03} & 0 & 0 \\
 a_{10} & a_{11} & a_{12} & a_{13} & 0 & 0 \\
 a_{20} & a_{21} & a_{22} & a_{23} & 0 & 0 \\
 \hline
 a_{30} & a_{31} & a_{32} & a_{33} & 0 & 0 \\
 a_{40} & a_{41} & a_{42} & a_{43} & 0 & 0 \\
 0 & 0 & 0 & 0 & 0 & 0
 \end{array}
 \times
 \begin{array}{ccc}
 b_{00} & 0 & 0 \\
 b_{10} & 0 & 0 \\
 b_{20} & 0 & 0 \\
 \hline
 b_{30} & 0 & 0 \\
 0 & 0 & 0
 \end{array}$$

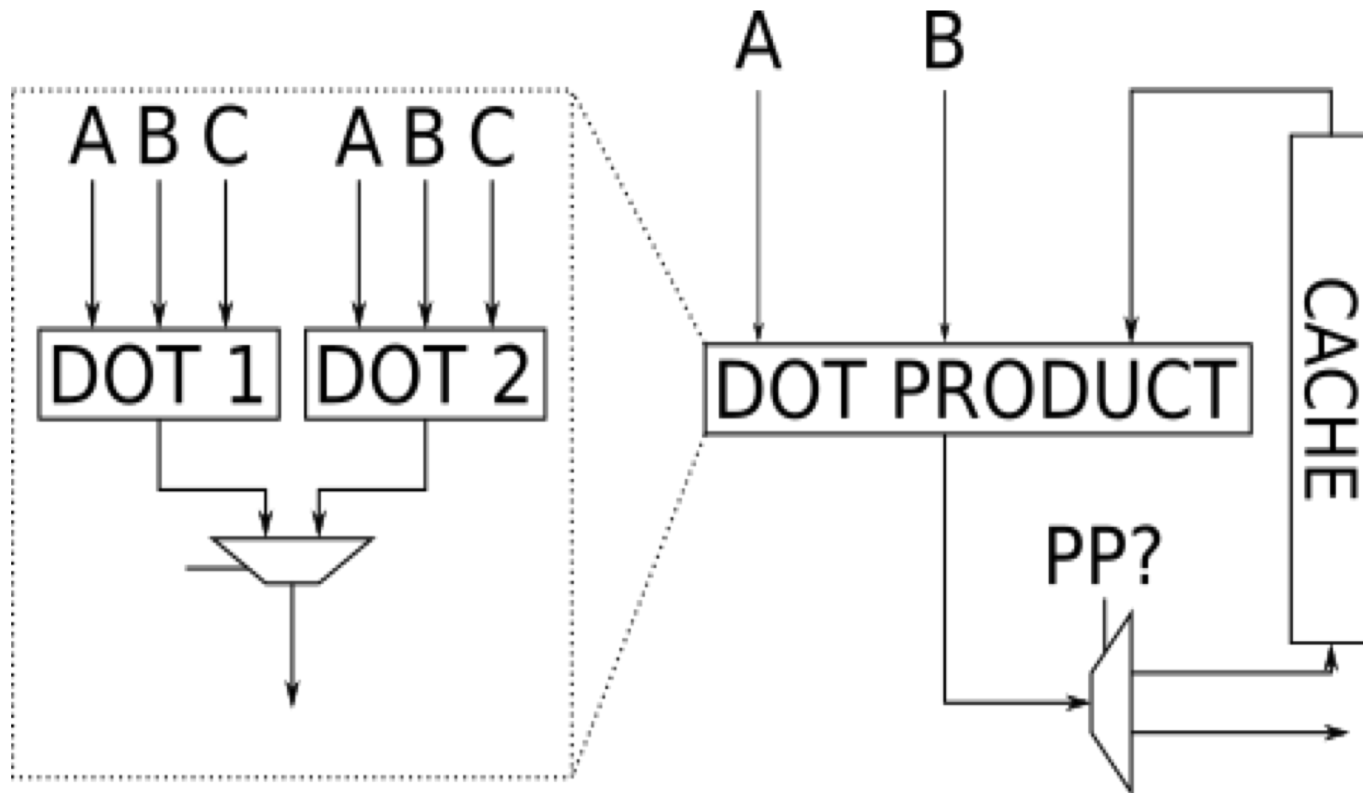
Inefficient with Static Partitioning

$$\begin{array}{ccc|cc}
 a_{00} & a_{01} & a_{02} & a_{03} & \\
 a_{10} & a_{11} & a_{12} & a_{13} & \\
 a_{20} & a_{21} & a_{22} & a_{23} & \\
 a_{30} & a_{31} & a_{32} & a_{33} & \\
 a_{40} & a_{41} & a_{42} & a_{43} &
 \end{array}
 \times
 \begin{array}{c}
 b_{00} \\
 b_{10} \\
 \hline
 b_{20} \\
 b_{30}
 \end{array}$$

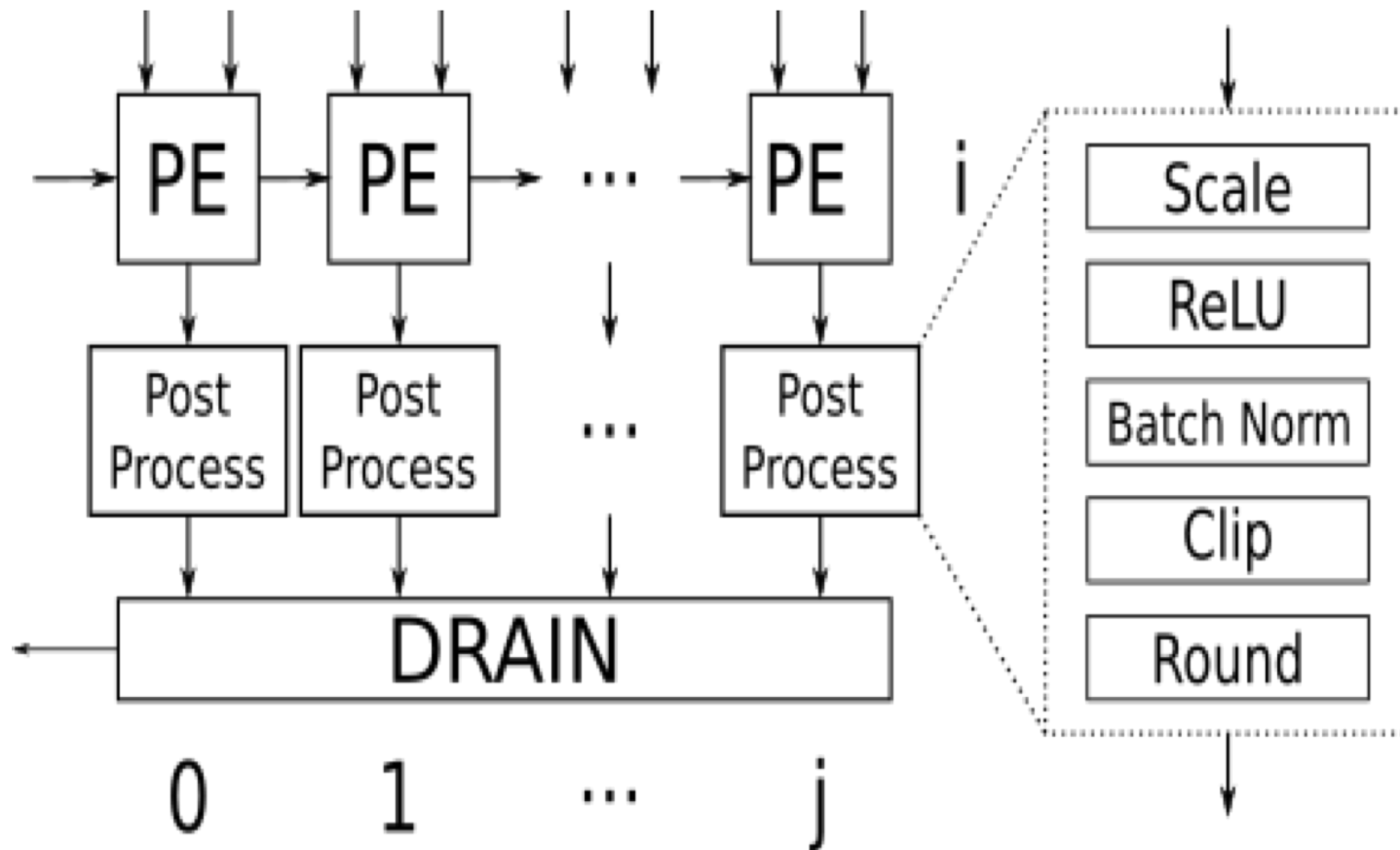
Minimising Bandwidth
Increase Maximum block size restriction

Layer	Type	
	Forward	Backward
...
conv	BINxBIN	FPxBIN
c&r	INT	STE
relu	INT	FP
norm	FP	FP
pool	FP	FP
...
fc	FPxFP	FPxFP
prob	FP	FP

STE=straight through estimator

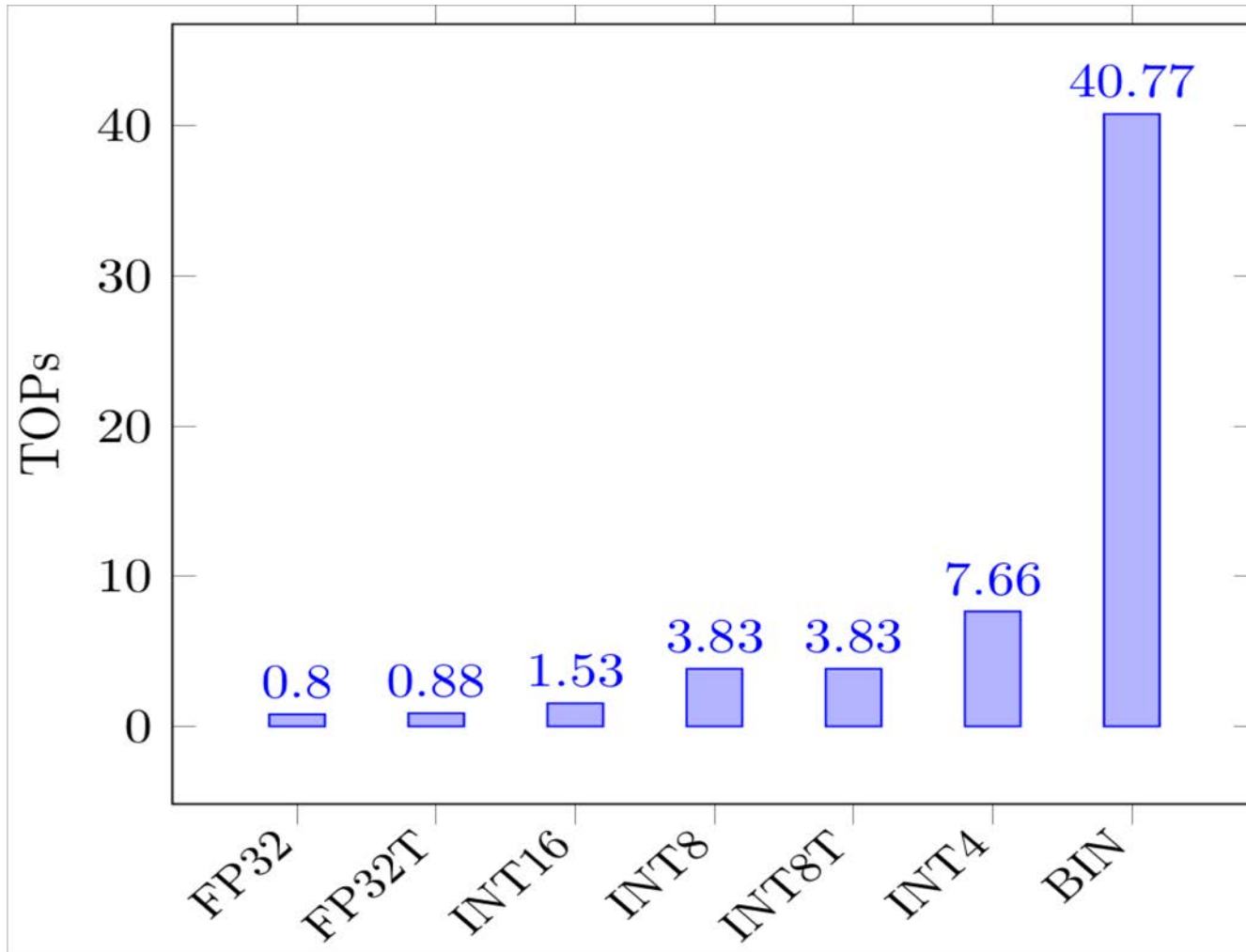


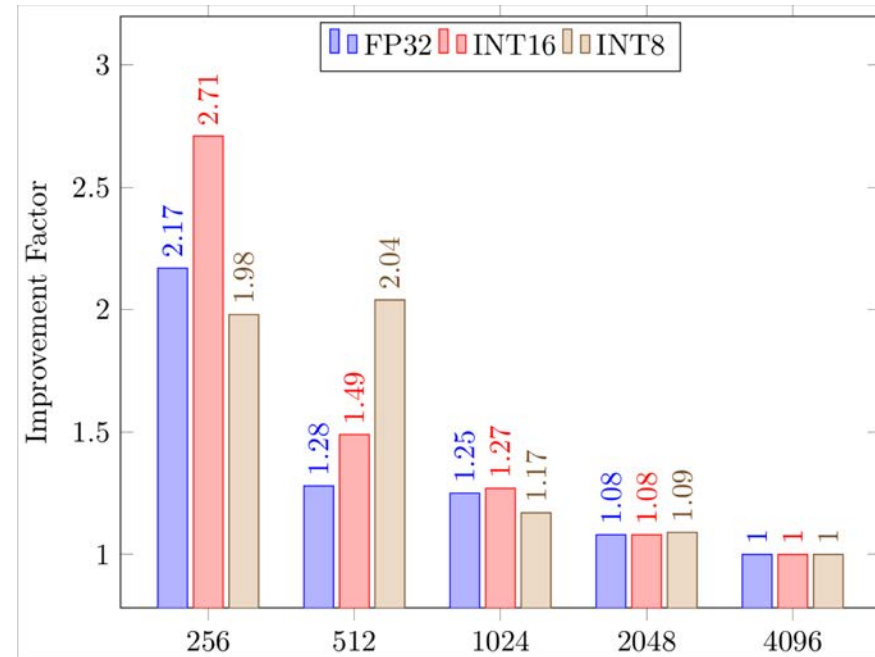
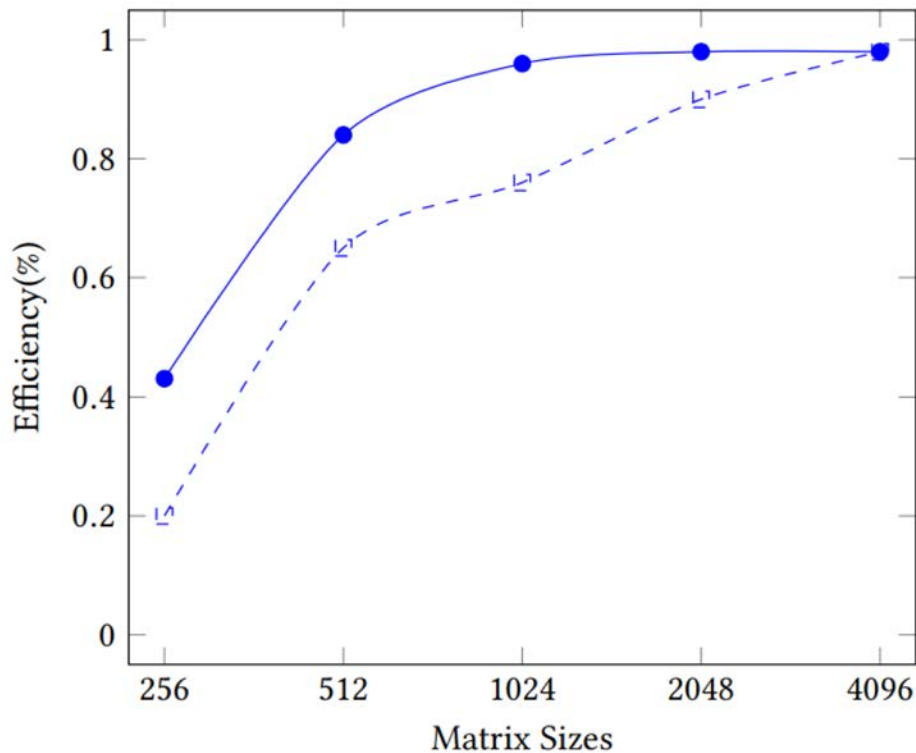
Supports two different precisions to avoid reconfiguration at runtime

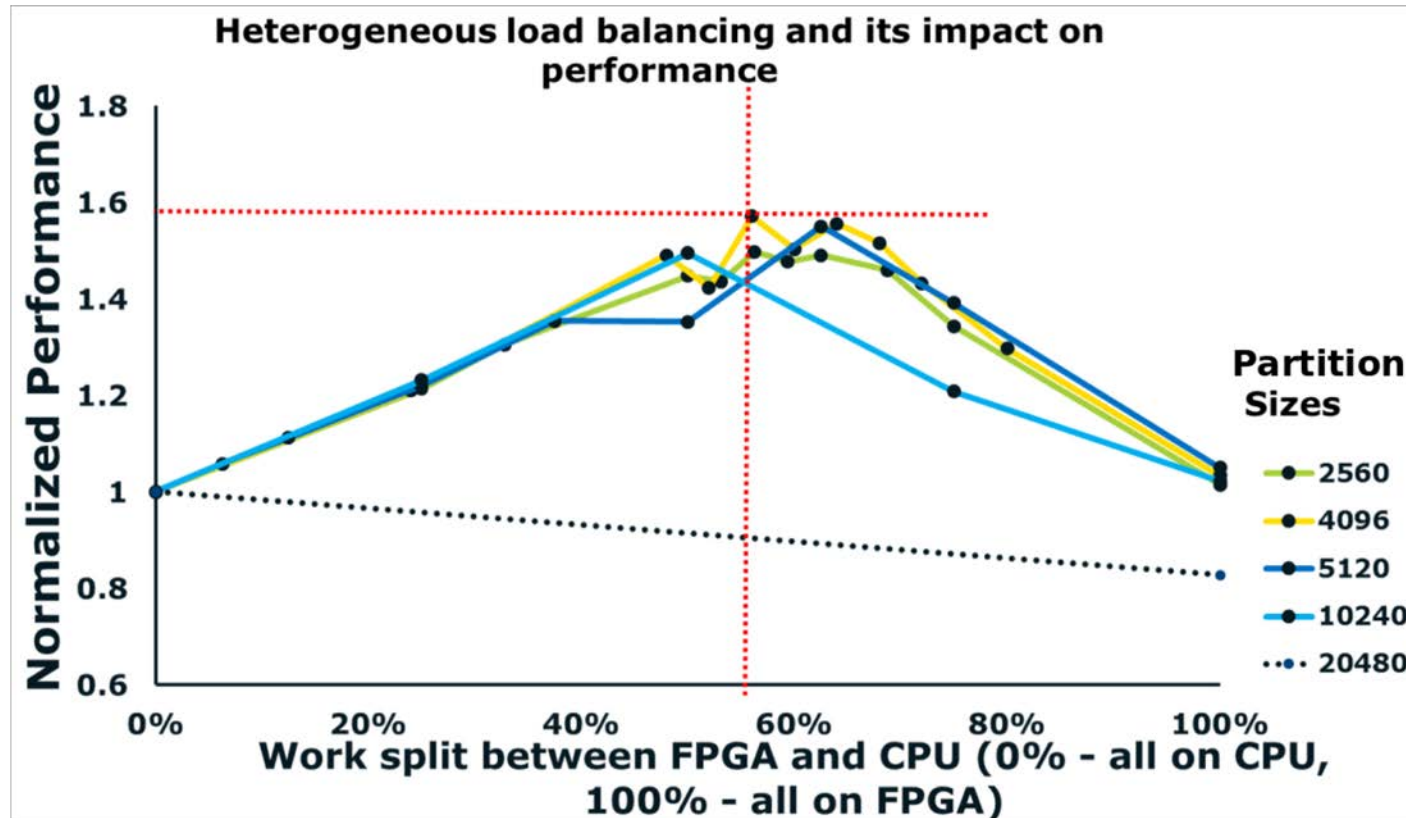




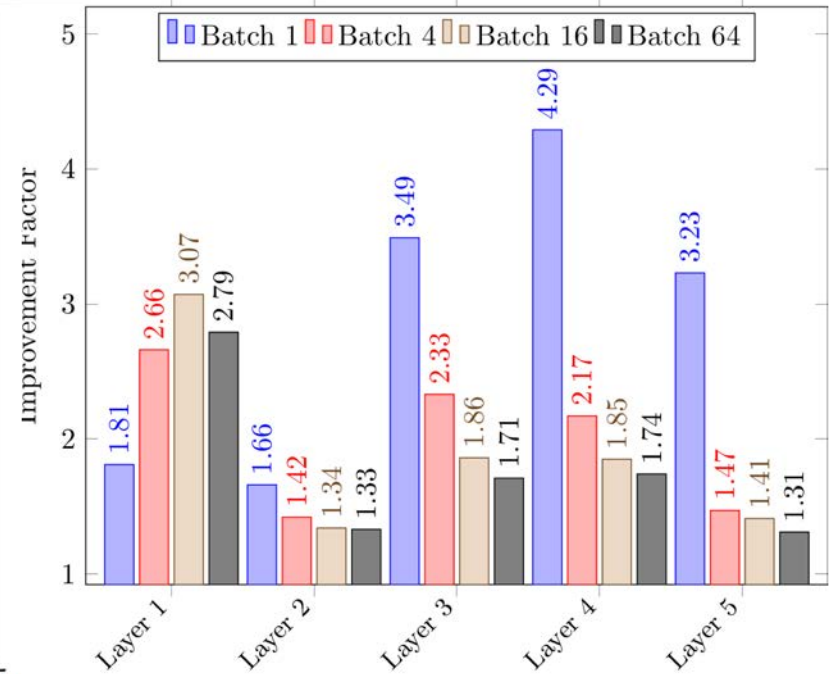
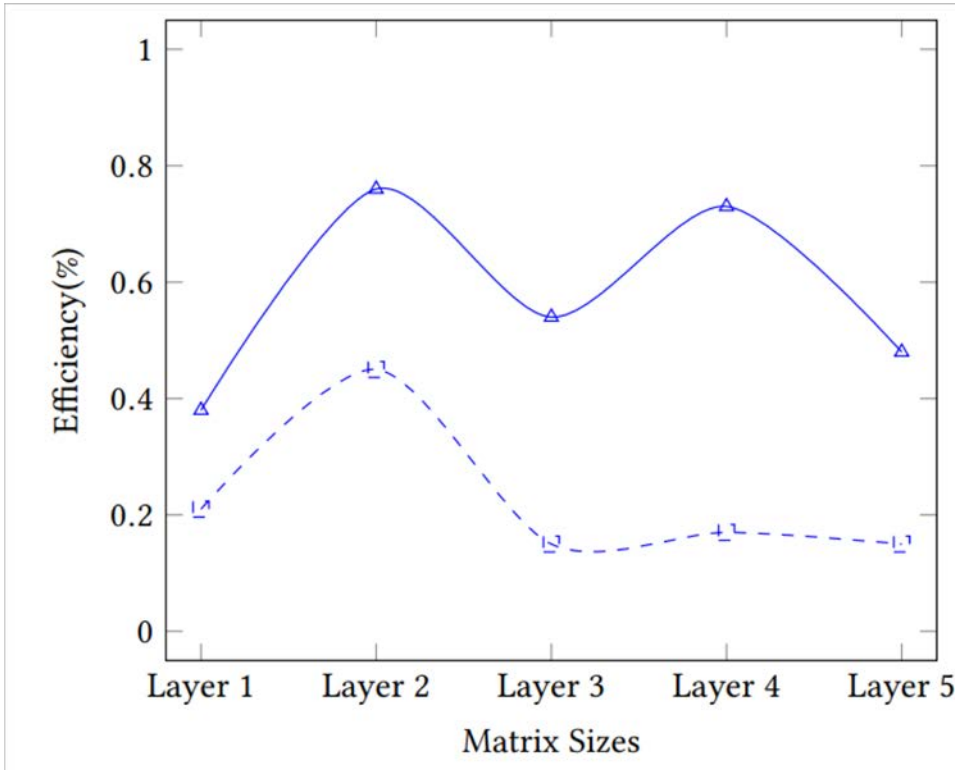
Measured Peak Performance







Neural Network Memory Interleaving Results



Device	FPGA				GPU			
	TOPs	GOPS/W	IPS	IPS/W	TOPS	GOPS/W	IPS	IPS/W
AlexNet	31.54	657.27	1610	33.54	37.60	568.09	1626	25.02
VGGNet	31.18	649.67	114	2.39	35.85	522.59	121	1.78

	[26]	[11]	[21]	Our Work
Platform	Zynq z7045	Kintex US KU115	Arria 10 GX1150	Arria 10 GX1150
Logic Elements (LEs)	350K	1,451K	1,150K	1,150K
Power (W)	11.3	41	-	48
TOPs (Peak)	11.612	14.8	25	40.77
MOPs / LE	33.17	10.19	-	35.45
GOPs / Watt	1027.68	360.97	-	849.38

- › **Exploration (Online kernel methods)**
- › **Parallelisation**
- › **Integration**
- › **Customisation**

› Exploration

- › Kernel methods optimised using different algorithms, mathematical techniques, computer architectures, arithmetic

› Parallelism

- › Increase parallelism by reducing precision
- › Keep weights on-chip to devote more hardware to arithmetic

› Integration

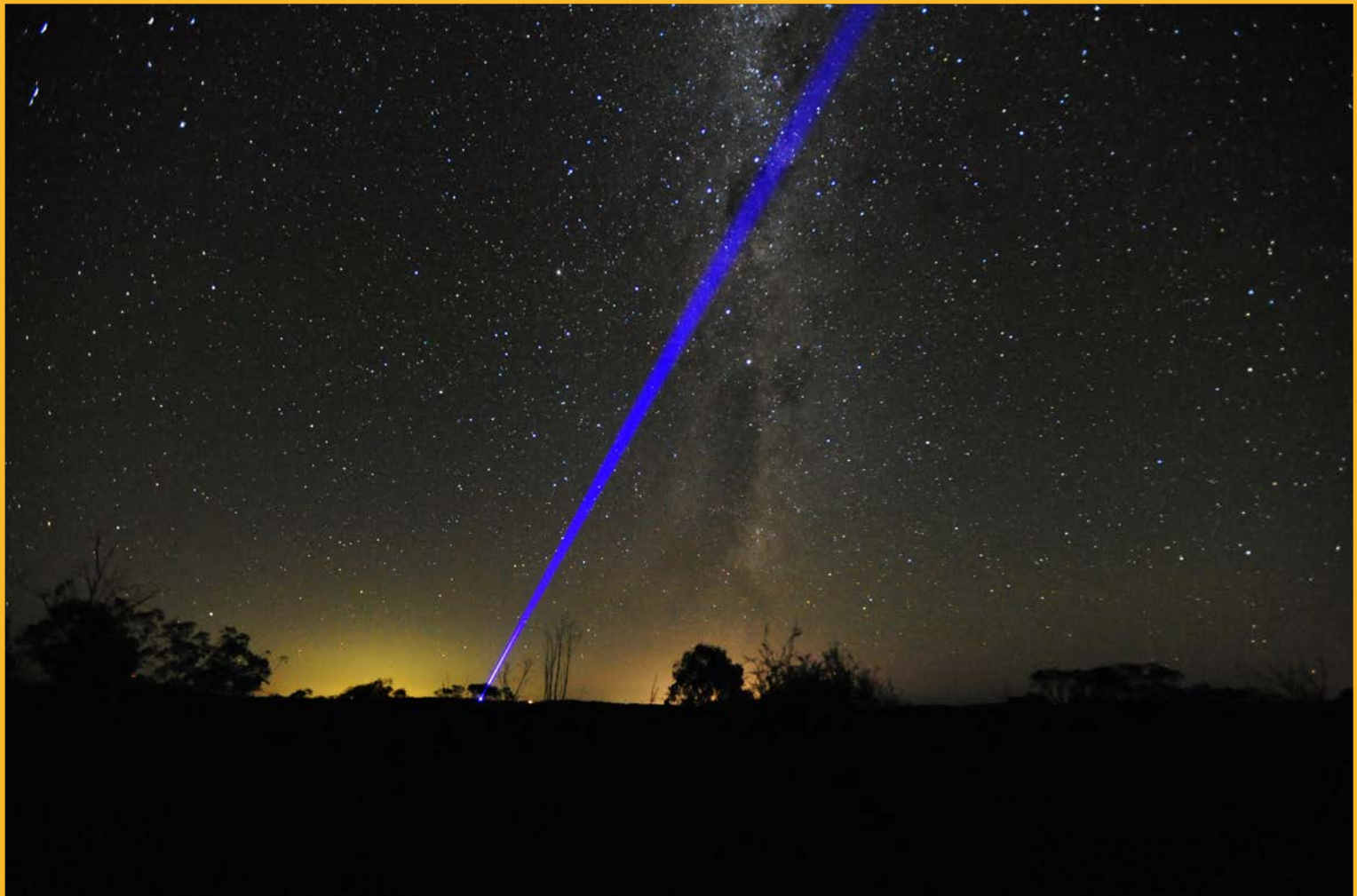
- › In radio frequency, this allows latency to be reduced by 4 orders of magnitude

› Customisation

- › Supplement conventional matrix multiplication to support DNN implementation

› FPGAs can greatly assist with the implementation of intelligent sensing

- › Learning & inference at 70 Gbps
 - › Learning & inference with 100 ns latency
 - › Image processing @ 12.3 Mfps
 - › Multimodal measurements
- ## › Radio frequency anomaly detector
- › We are using this to predict physical and media access layer protocols
 - › Could also be used as a novel diagnostic instrument - monitor RF output of electronic equipment, detect anomalies



Thank you!

Philip Leong (philip.leong@sydney.edu.au)
<http://phwl.org>

