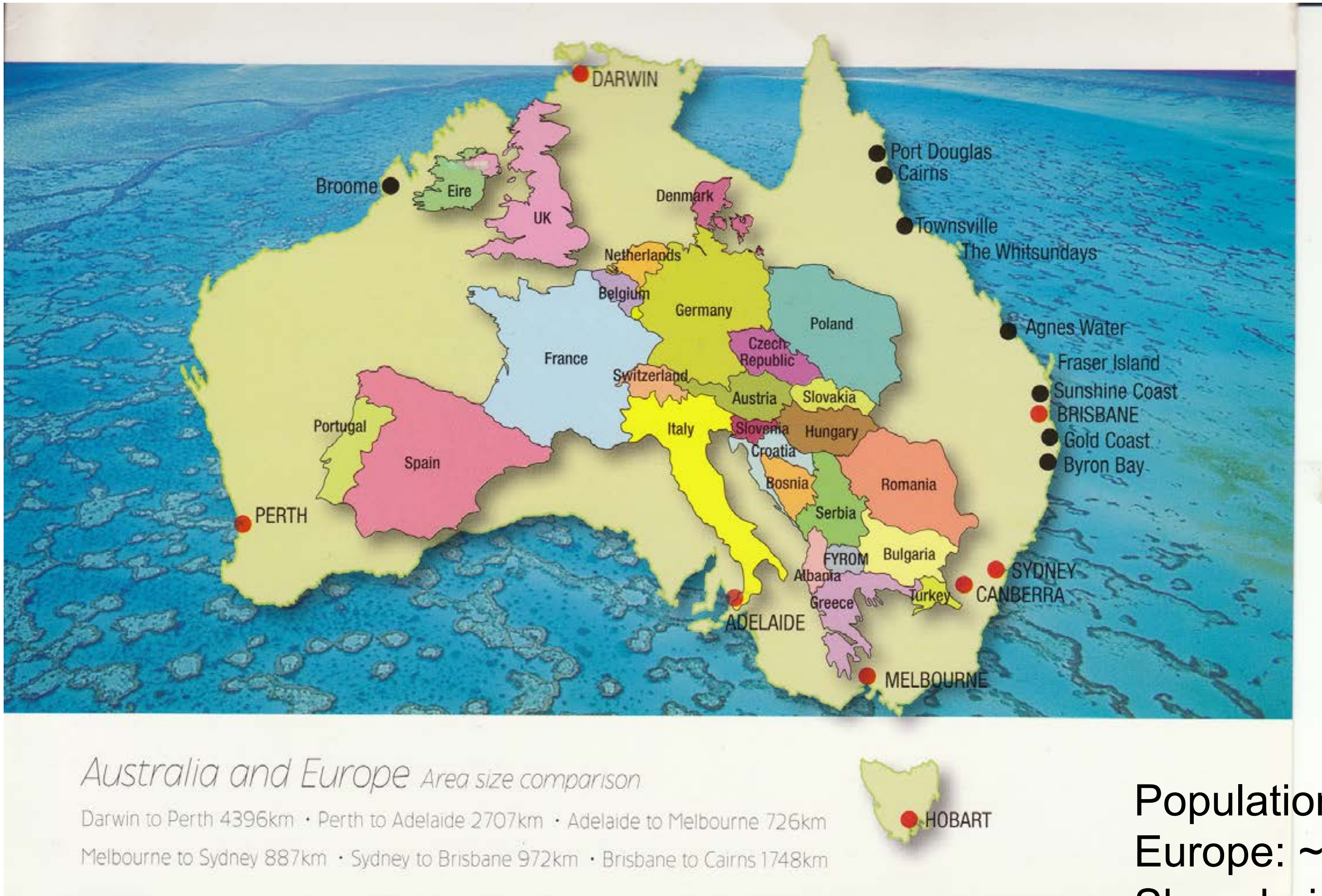
# Multipliers for FPGA Machine Learning Applications

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

THE UNIVERSITY OF SYDNEY

# Australia

Population density: 3.20 people per km$^2$ (Shanghai 2059)




Population: ~25M (2017)
Europe: ~743M (2018)
Shanghai: ~24M (2018)

# 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, Intel, Exablaze
  - Defence and DSTG
  - clustertech.com

# Overview

- Multipliers (and adders) play a key role in the implementation of DNNs
- This talk
  - Two speed multiplier with different critical paths for zero and non-zero recodings
  - PIR-DSP block to support a range of precisions
  - AddNet which uses k-levels of shifted values as multipliers
  - A fully pipelined DNN implementation with ternary coefficients

- These slides are available at https://phwl.github.io/talks

# A Two Speed Multiplier

*D. J. M. Moss, D. Boland, and P. H. W. Leong*

THE UNIVERSITY OF SYDNEY

# Overview

- Multipliers (and adders) play a key role in the implementation of DNNs
- This talk
  - **Two speed multiplier** with different critical paths for zero and non-zero recodings
  - PIR-DSP block to support a range of precisions
  - AddNet which uses k-levels of shifted values as multipliers
  - A fully pipelined DNN implementation with ternary coefficients

# Unsigned Multiplication

Example: Multiply 118d by 99d

| | | |
|---|---|---|
| Step1) Initialize | Multiplicand | 118d |
| | Multiplier | 99d |
| Step2) Find partial products | | 1062d |
| | | 1062 d |
| Step3) Sum up the shifted partial products | | 11682d |

- Shift-and-Add Algorithm

*Two's Complement Method*

Step1) Initialize

118d = 01110110b
99d = 01100011b

Step2) Find partial products

```
        01110110b
       01110110 b
      00000000  b
     00000000   b
    00000000    b
   01110110     b
  01110110      b
 00000000       b
010110110100010 b
```

Step3) Sum up the shifted partial products

Convert 2's-Comp back to decimal:
0010 1101 1010 0010 = 11682d

# Signed Multiplication

› How can we handle signed multiplication?

› Could

  - multiply absolute values
  - separately calculate the sign
  - negate if necessary

› But …

# Signed Multiplication using Booth Recoding

- Booth Recoding
  - Reduce the number of partial products by recoding the multiplier operand
  - Works for signed numbers

Example: Multiply -118 by -99

Recall, 99 = 0110 0011b

-99 = 1001 1101b

Radix-2 Booth Recoding

$-99 = \bar{1}010\ \ 0\bar{1}1\bar{1}$

Low-order Bit

Last Bit Shifted Out

| $A_n$ | $A_{n-1}$ | Partial Product |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | +B |
| 1 | 0 | -B |
| 1 | 1 | 0 |

Source: Shawn Nicholl, Xilinx

# Example of Booth Radix-2 Recoding

Multiply -118 by -99

B = -118 = 1000 1010b
-B = 118 = 0111 0110b

A = -99 = 1001 1101b
-99 = $\bar{1}010\ \ 0\bar{1}1\bar{1}$

- $-99 = (-2^7 + 2^5 - 2^2 + 2^1 - 2^0)$

*Radix-2 Booth*

Step1) Initialize

-118 = 0111 0110b
-99 = $\bar{1}010\ \ 0\bar{1}1\bar{1}$

Step2) Find partial products

```
        01110110b  -B
       110001010 b  B
       01110110  b -B
      00000000   b  0
     00000000    b  0
  1110001010     b  B
 000000000       b  0
 01110110        b -B
0010110110100010b
```

Sign Extension

Step3) Sum up the shifted partial products

Convert 2's-Comp back to decimal:
0010 1101 1010 0010 = 11682d

# Booth Radix-4 Multiplication

› Similar to Radix-2, but uses looks at two low-order bits at a time (instead of 1)

Recall, 99d = 0110 0011b

$$
\begin{array}{r}
1001\ 1100b \\
\underline{1b} \\
-99d = 1001\ 1101b
\end{array}
$$

Radix-4 Booth Recoding

$-99d = \bar{2}2\bar{1}1$

Low-order Bits

Last Bit Shifted Out

| $Y_{i+2}$ | $Y_{i+1}$ | $Y_i$ | $e_i$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | +B |
| 0 | 1 | 0 | +B |
| 0 | 1 | 1 | +2B |
| 1 | 0 | 0 | -2B |
| 1 | 0 | 1 | -B |
| 1 | 1 | 0 | -B |
| 1 | 1 | 1 | 0 |

› $(-99=-2.4^3+2.4^2-1.4^1+1.4^0)$

# Example of Booth Radix-4 Multiplication

Example: Multiply -118d by -99d

B = -118d = 1000 1010b
-B = 118d = 0111 0110b
2B = -236d = 1 0001 0100b
-2B = 236d = 0 1110 1100b

A = -99d = 1001 1101b
-99d = $\bar{2}2\bar{1}1$

Sign Extension

*Radix-4 Booth*

Step1) Initialize

-118d = 0111 0110b
-99d = $\bar{2}\ 2\ \bar{1}\ 1$

Step2) Find partial products

Step3) Sum up the shifted partial products

| | | |
|---|---|---|
| 1111111110001010 | b | B |
| 01110110 | b | -B |
| 111100010100 | b | 2B |
| 011101100 | b | -2B |
| 0010110110100010 | b | |

Convert 2's-Comp back to decimal:
0010 1101 1010 0010 = 11682d

- Reduces number of partial products by half!

Source: Shawn Nicholl, Xilinx

# Booth Radix-4 Multiplier Implementation

TABLE I: Booth Encoding

| $Y_{i+2}$ | $Y_{i+1}$ | $Y_i$ | $e_i$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 2 |
| 1 | 0 | 0 | $\bar{2}$ |
| 1 | 0 | 1 | $\bar{1}$ |
| 1 | 1 | 0 | $\bar{1}$ |
| 1 | 1 | 1 | 0 |

$\bar{2}$ and $\bar{1}$ represent $-2$ and $-1$ respectively.

**Algorithm:** Booth Radix-4 Multiplication

**Data:** $y$: Multiplier, $x$: Multiplicand
**Result:** $p$: Product

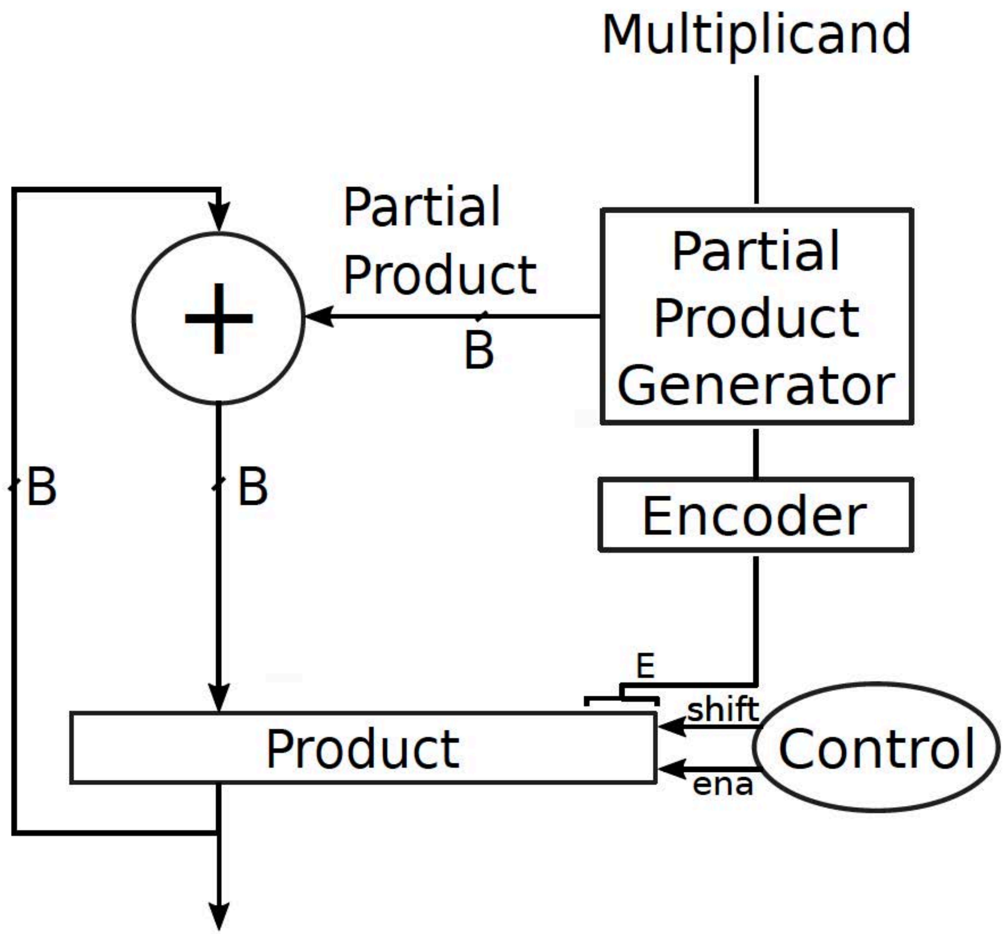
$p = y$;
$e = (P[0] - 2P[1])$;
**for** $count = 1$ **to** $N$ **do**
    $PartialProduct = e * x$;
    $p =$ sra($p$,2);
    $P[2 * B - 1 : B] += PartialProduct$;
    $e = (P[1] + P[0] - 2P[2])$;
**end**

# Booth Radix-4 Multiplier Datapath

**Algorithm:** Booth Radix-4 Multiplication

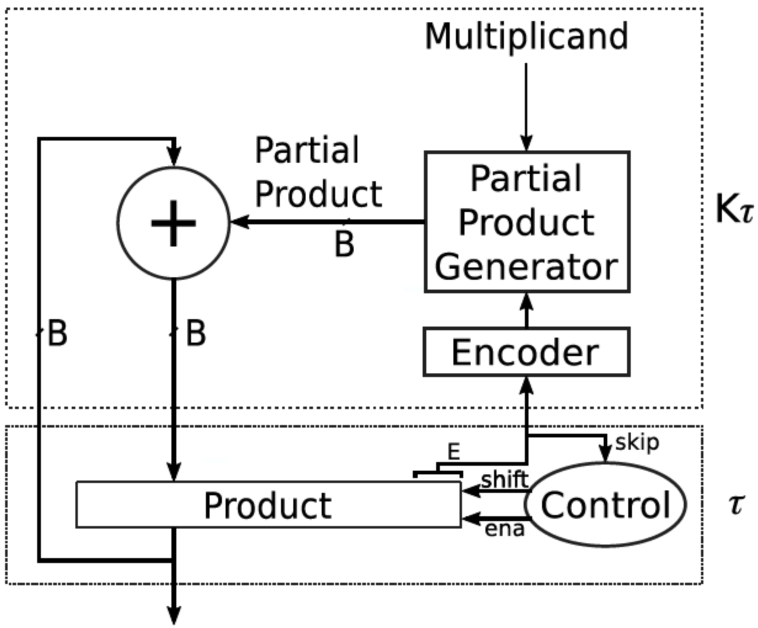
**Data:** $y$: Multiplier, $x$: Multiplicand
**Result:** $p$: Product

$p = y$;
$e = (P[0] - 2P[1])$;
**for** $count = 1$ **to** $N$ **do**
  $PartialProduct = e * x$;
  $p = \text{sra}(p,2)$;
  $P[2*B-1 : B] += PartialProduct$;
  $e = (P[1] + P[0] - 2P[2])$;
**end**



# Two-Speed Multiplier

- Booth Radix-4 datapath split into 2 sections, each with own critical path
- Non-zero encodings take $\bar{K}\tau$ (add) and zero take $\tau$ (skip)
- Naturally supports sparse problems




**Algorithm:** Two Speed Booth Radix-4 Multiplication

**Data:** $y$: Multiplier, $x$: Multiplicand
**Result:** $p$: Product

$p = y$;
$e = (P[0] - 2P[1])$;
**for** $count = 1$ **to** $N$ **do**
  $p = \text{sra}(p,2)$;
  `// If non-zero encoding, take the` $K\tau$ `path, otherwise the` $\tau$ `path`
  **if** $e \neq 0$ **then**
    `// this path is clocked` $\bar{K}$ `times`
    $PartialProduct = e * x$;
    $P[2 * B - 1 : B] += PartialProduct$;
  **end**
  $e = (P[1] + P[0] - 2P[2])$;
**end**

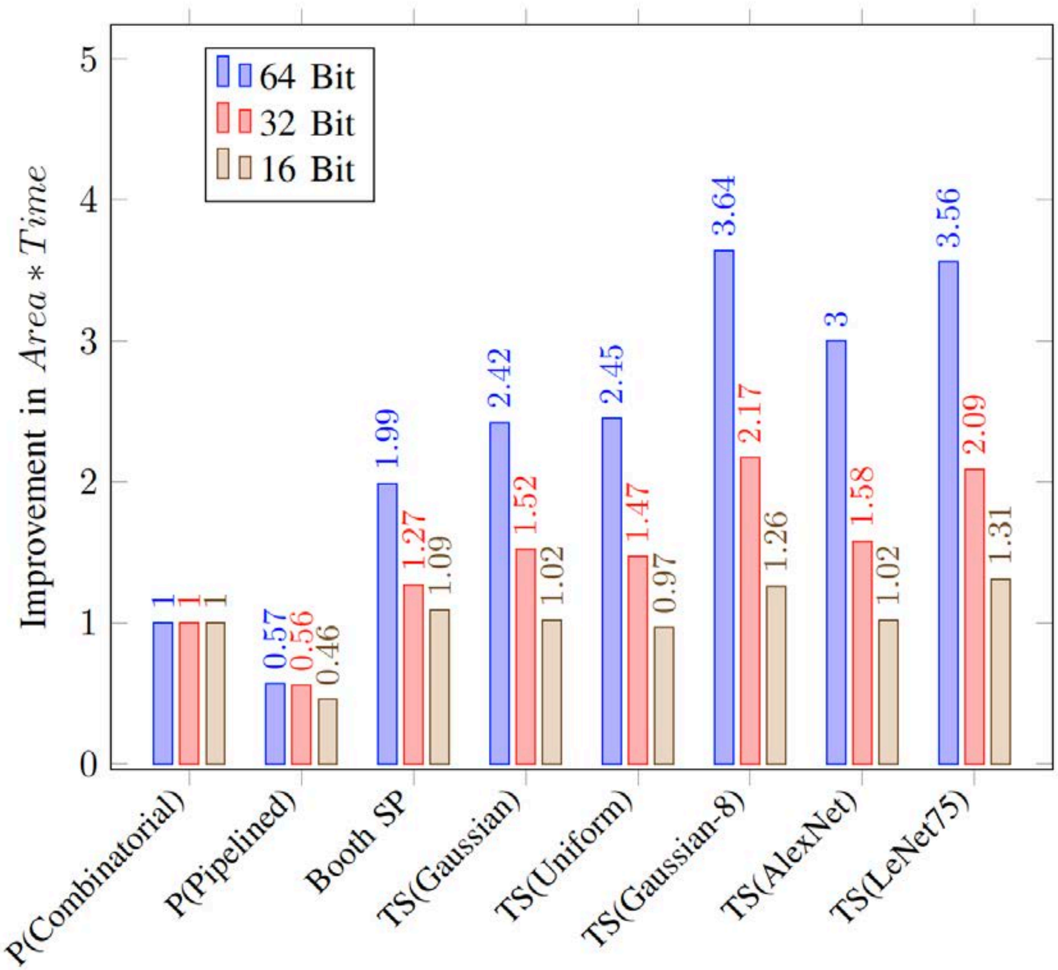
# Two-speed multiplier Execution

| Bit Representation | Action | Time | PartialProduct |
|---|---|---|---|
| 1 1 1 1 0 1 0 0 0 1 0 0 0 | skip | $\tau$ | $0x$ X $2^{0}$ |
| 1 1 1 1 0 1 0 0 0 1 0 | add | $\tau + \bar{K}\tau$ | $1x$ X $2^{2}$ |
| 1 1 1 1 0 1 0 0 0 | skip | $2\tau + \bar{K}\tau$ | $0x$ X $2^{4}$ |
| 1 1 1 1 0 1 0 | add | $2\tau + 2\bar{K}\tau$ | $1x$ X $2^{6}$ |
| 1 1 1 1 0 | add | $2\tau + 3\bar{K}\tau$ | $-1x$ X $2^{8}$ |
| 1 1 1 | skip | $3\tau + 3\bar{K}\tau$ | $0x$ X $2^{10}$ |

## Results

| B | Type | *Area* (LEs) | Max Delay (ns) | Latency (Cycles) | *Power* (mW) |
|---|---|---|---|---|---|
| 64 | Parallel(Combinatorial) | 5104 | 14.7 | 1 | 2.23 |
| | Parallel(Pipelined) | 4695 | 6.99 | 4** | 9.62 |
| | Booth Serial-Parallel | 292 | 3.9 | 33 | 2.23 |
| | Two Speed | 304 | 1.83 ($\tau$) | 45.2* | 5.2 |
| 32 | Parallel(Combinatorial) | 1255 | 10.2 | 1 | 1.33 |
| | Parallel(Pipelined) | 1232 | 4.6 | 4** | 5.07 |
| | Booth Serial-Parallel | 156 | 3.8 | 17 | 1.78 |
| | Two Speed | 159 | 1.76 ($\tau$) | 25.6* | 3.18 |
| 16 | Parallel(Combinatorial) | 319 | 6.8 | 1 | 0.94 |
| | Parallel(Pipelined) | 368 | 3.2 | 4** | 3.49 |
| | Booth Serial-Parallel | 81 | 2.72 | 9 | 1.67 |
| | Two Speed | 87 | 1.52 ($\tau$) | 14* | 4.35 |

# Area * Time Improvement of TSM



# Summary

- Variant of the serial-parallel modified radix-4 Booth multiplier
- Adds only the non-zero Booth encodings and skips over the zero operations
- Two sub-circuits with different critical paths are utilised so that throughput and latency are improved for a subset of multiplier values
- For bit widths of 32 and 64, our optimisations can result in a 1.42-3.36x improvement over the standard parallel Booth multiplier
- Future work: explore training NN with weights to minimise execution time on TSM

# PIR-DSP: An FPGA DSP block Architecture for Multi-Precision Deep Neural Networks

*SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H.W. Leong*

THE UNIVERSITY OF SYDNEY

# Overview

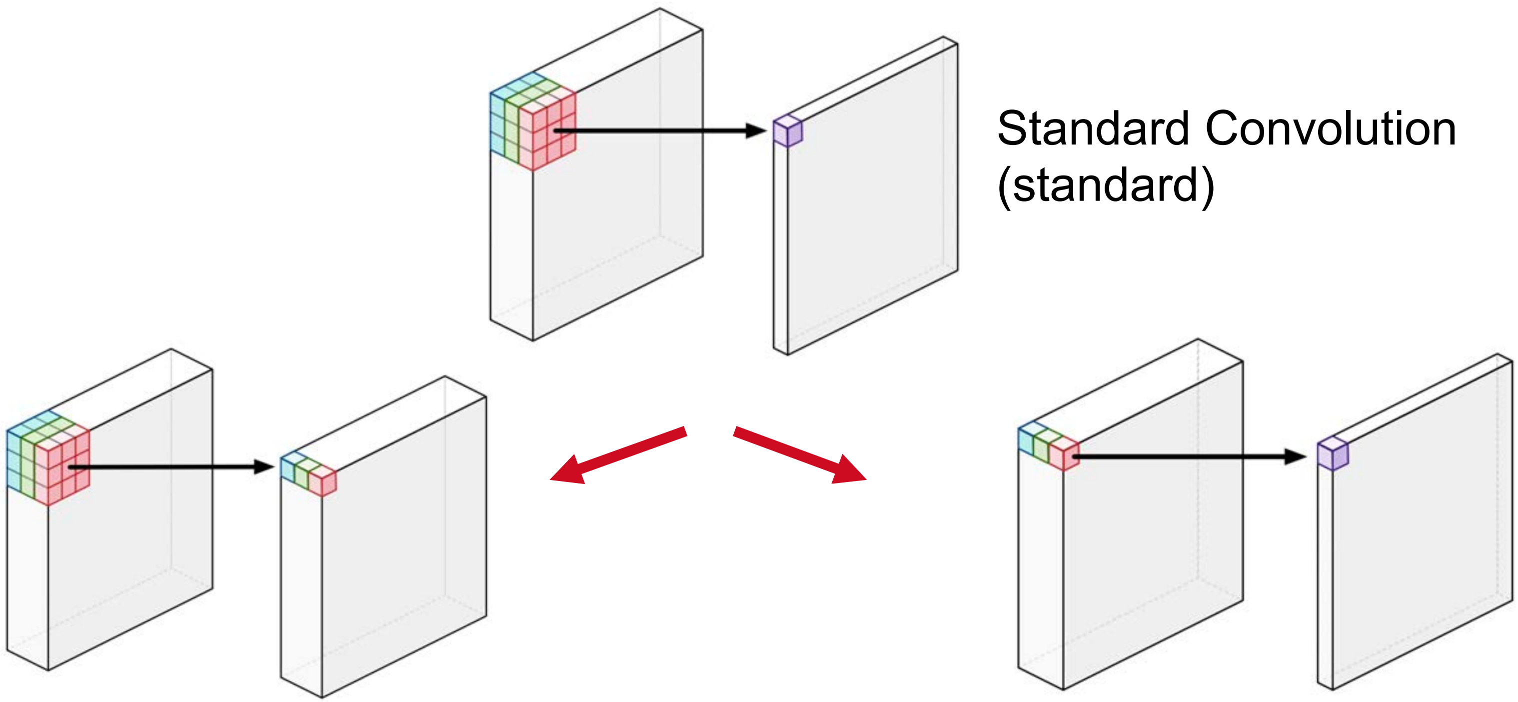
- Multipliers (and adders) play a key role in the implementation of DNNs
- This talk
  - Two speed multiplier with different critical paths for zero and non-zero recodings
  - **PIR-DSP** block to support a range of precisions
  - AddNet which uses k-levels of shifted values as multipliers
  - A fully pipelined DNN implementation with ternary coefficients

# Overview

- Introduction
- PIR-DSP Architecture
- Results
- Conclusion

# Embedded Deep Neural Networks

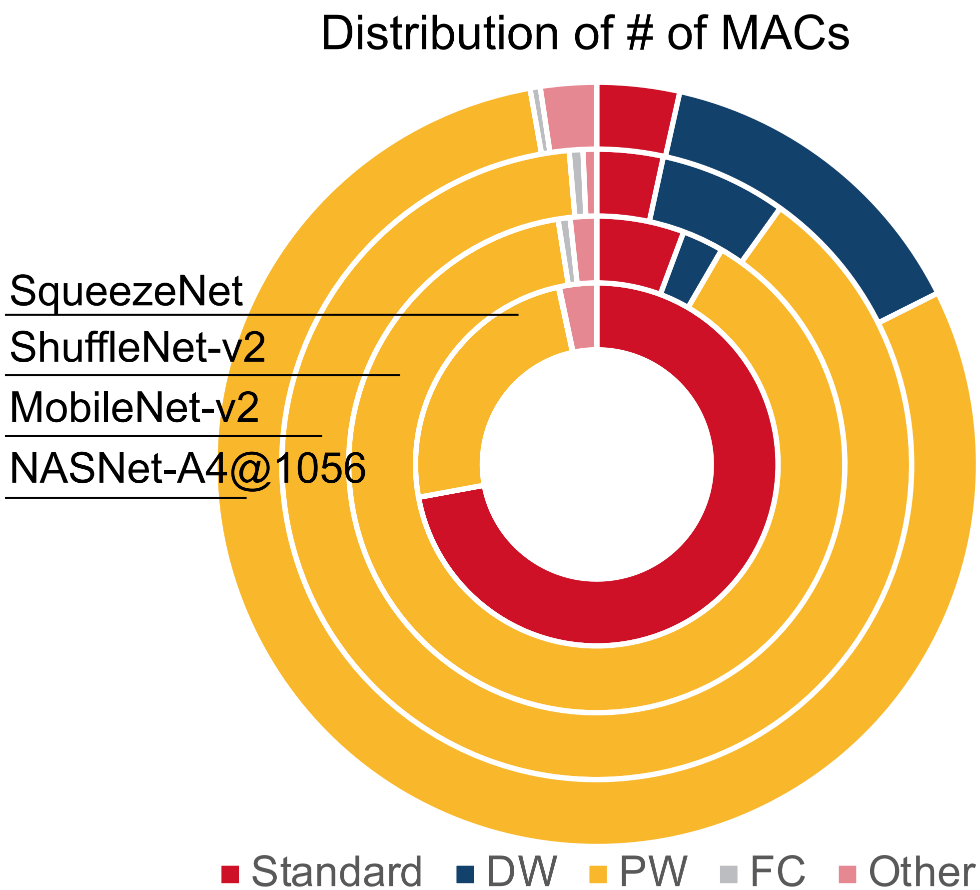
- DNNs for embedded applications share two features to reduce computation and storage requirements
  - Low precision (from 1-16 bits)
  - Depthwise separable convolutions

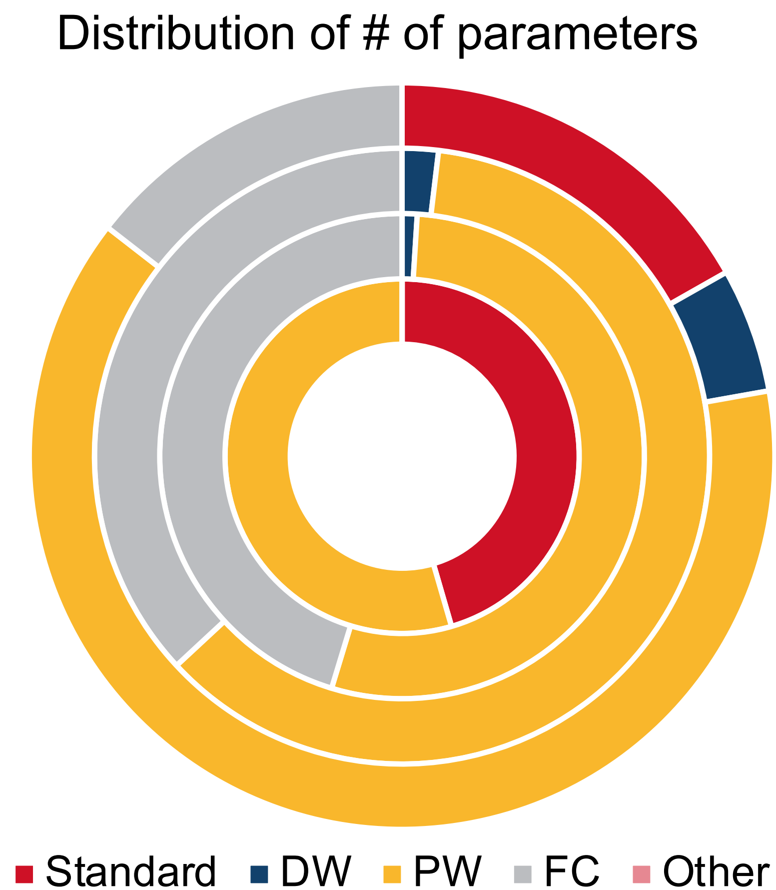



Depthwise Convolution (DW) Pointwise Convolution (PW)

# Motivation (1)

## Computation and Storage for Embedded DNNs



# Motivation (2)

Low-Precision Neural Networks

## Imagenet accuracy with binary and ternary weights and 8-bit activations

| 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 |

Faraone et al, “SYQ: Learning Symmetric Quantization For Efficient Deep Neural Networks”, CVPR’18

# Aims

- Optimise FPGA DSP architecture to better support
  - Efficient implementation of embedded DNNs
  - Wordlengths down to ternary and binary
- Talk will focus on convolutions

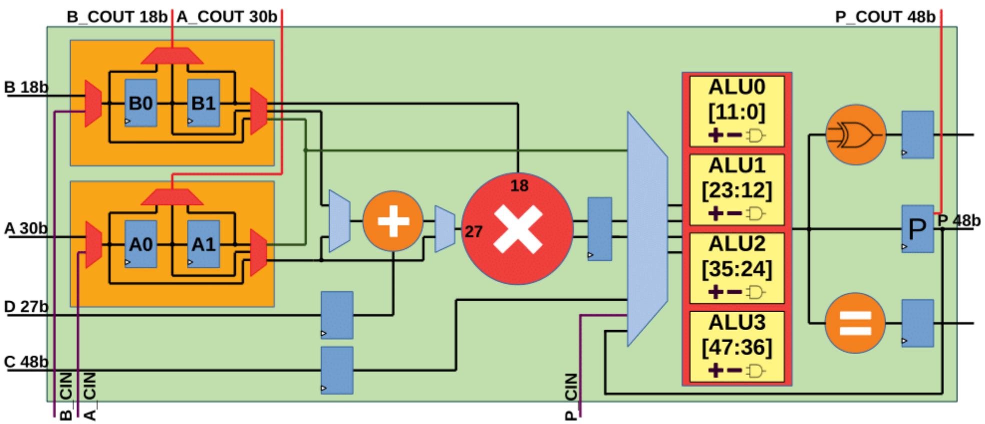
# Overview

- Introduction
- PIR-DSP Architecture
- Results
- Conclusion

# Existing DSPs

› Xilinx DSP48

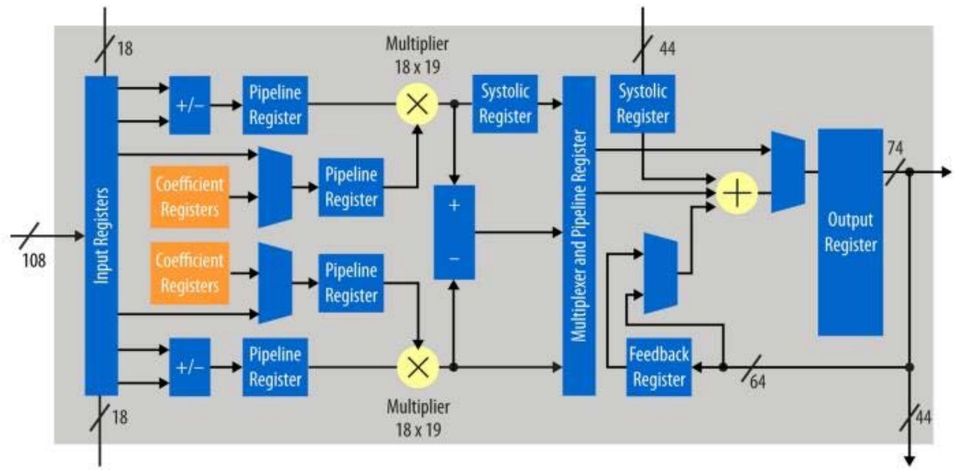
- 27×18 multiplier, 48-bit ALU (Add/Sub/Logic), 27-bit pre-adder, Wide 96bit XOR, 48-bit comparator




  - No support for low-precision computations
  - No run-time configuration
  - 1D arrangement inefficient for implementing 2D systolic arrays

› Intel (Multiprecision)

- 27×27 multiplier decomposable to two 19×18, Compile-time configurable Coefficient registers, Two 18-bit pre-adder, 54-bit adder

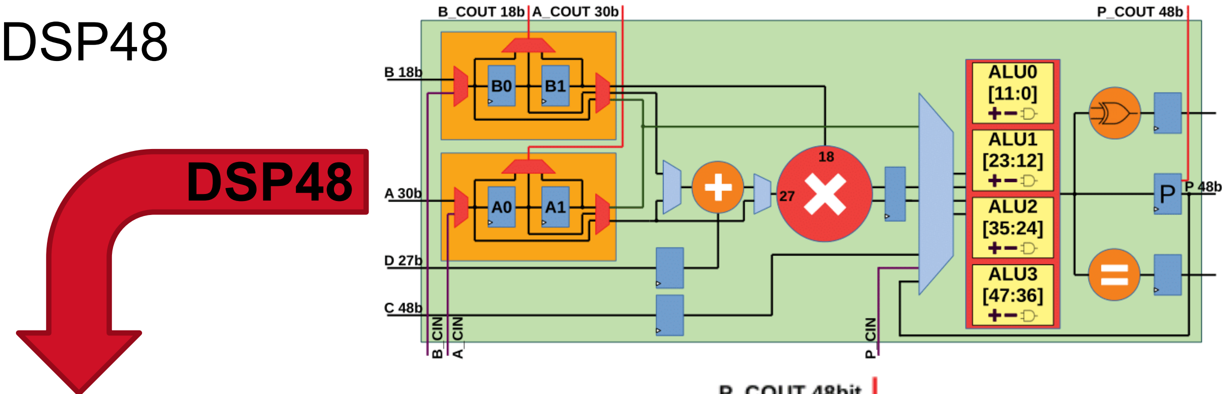


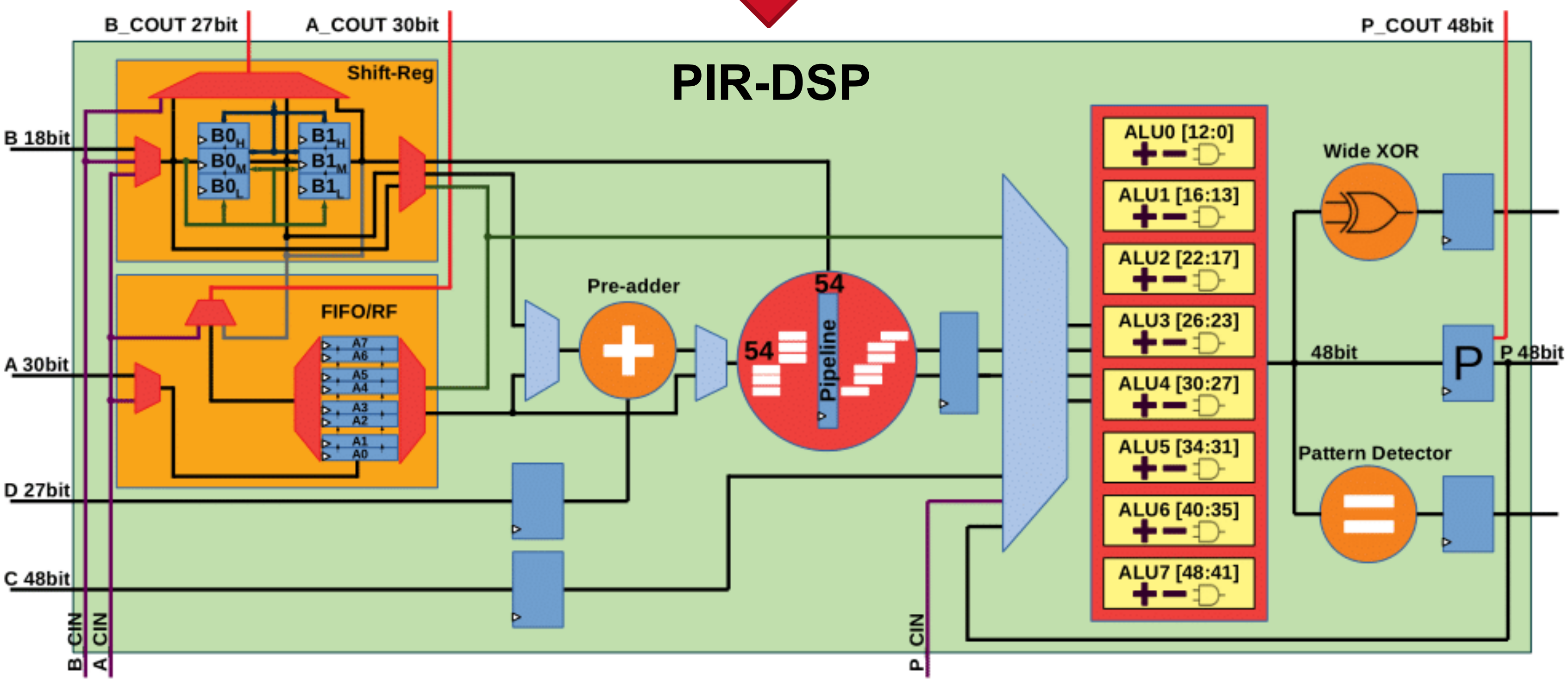
THE UNIVERSITY OF SYDNEY

# PIR-DSP

› PIR-DSP: Optimized version of DSP48

- **P**recision: Multiplier architecture
- **I**nterconnect: Shift-Reg
- **R**euse : RF/FIFO



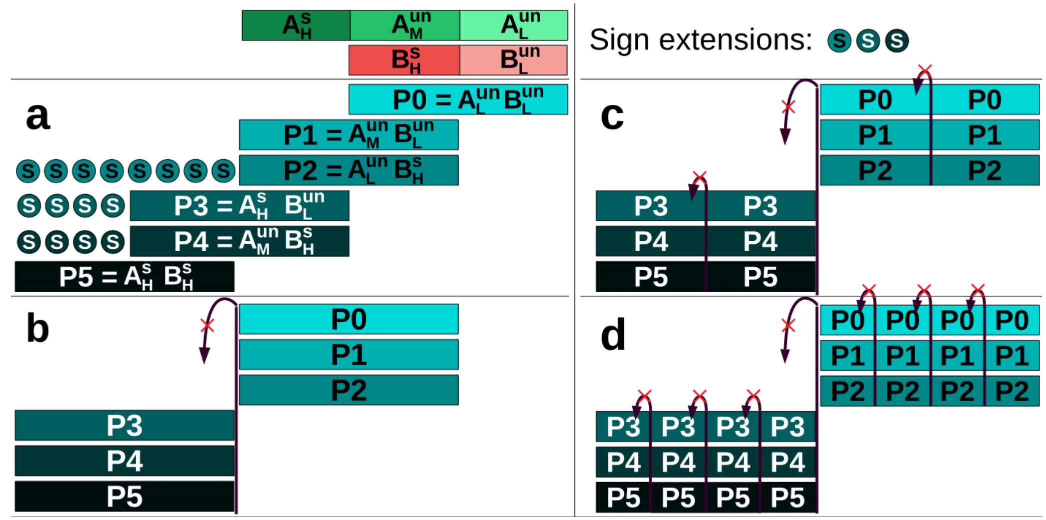



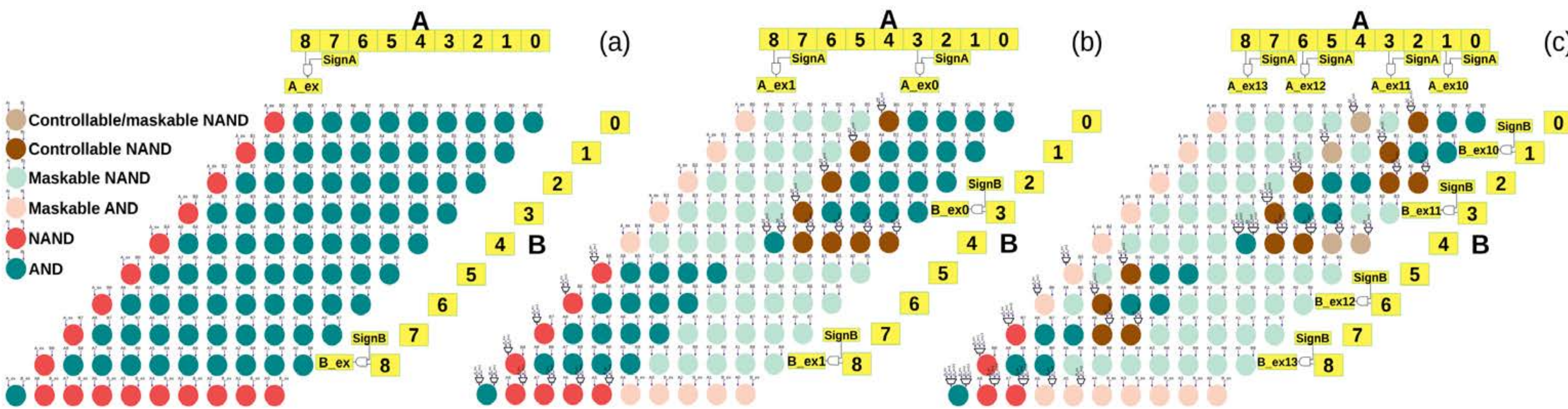
# Precision (1)

› Based on two approaches:

1. Chopping

2. Recursive decomposition



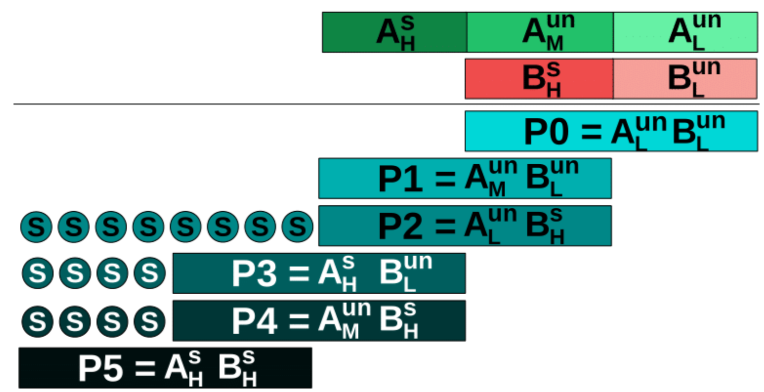
# Precision (2)

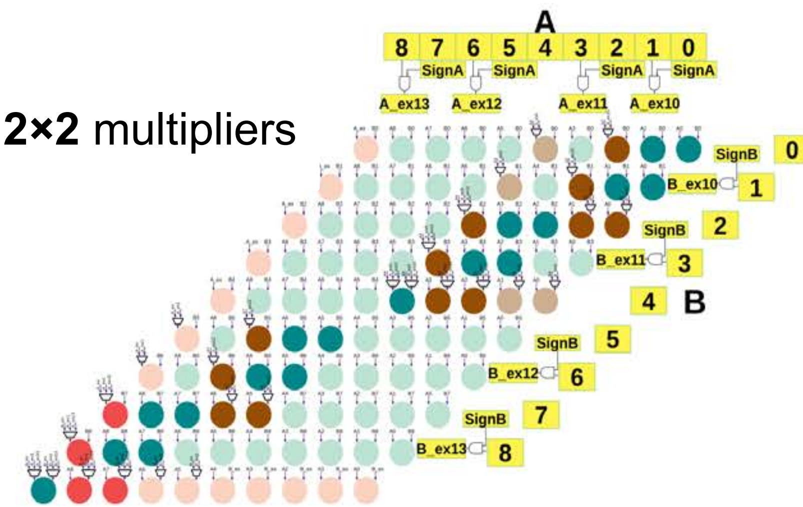
*Parameterised Decomposable MAC unit*

› Notation: M×NC*ij*D*k*

› PIR-DSP multiplier: 27×18C*32*D*2*

- **C**hopping factors 3 and 2 respectively for 27 and 18
  - (27=9+9+9)×(18=9+9)
  - Six 9×9 multiplier
- **D**ecomposing factor is 2
  - Each 9×9 multiplier decomposes to Two **4×4** or Four **2×2** multipliers




› **PIR-DSP** Modes:

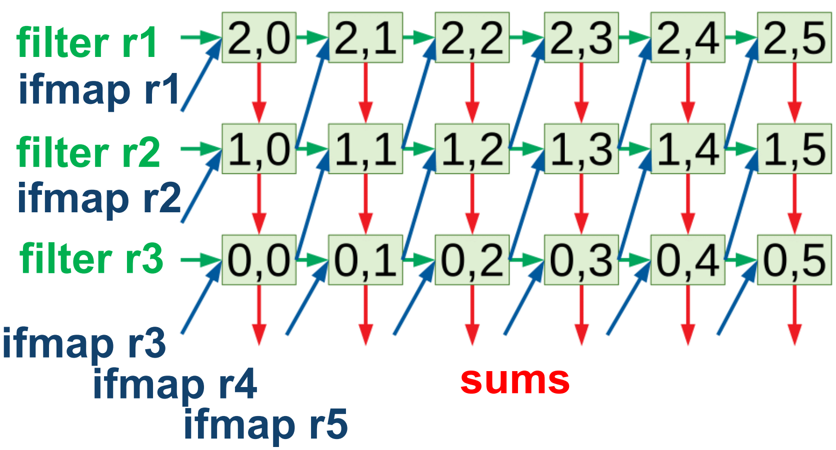
- One 27×18 → 1 MAC
- Two 9×9 + 9×9 + 9×9 → 6 MACs
- Four 4×4 + 4×4 + 4×4 → 12 MACs
- Eight 2×2 + 2×2 + 2×2 → 24 MACs

# Interconnect (1)

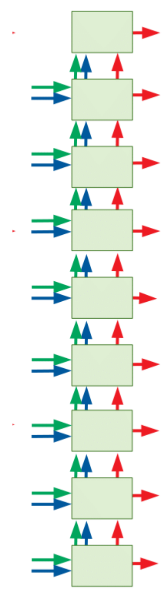
› Three types of convolutions

1- **Depth-wise**: using three PIR-DSPs

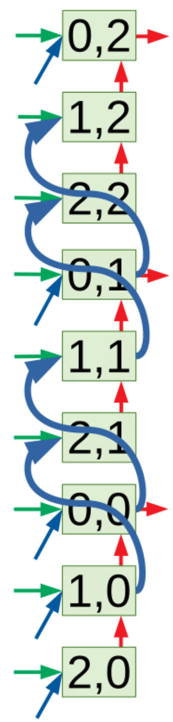
2- **Standard**: based on depth-wise convolution implementation and adding the partial results



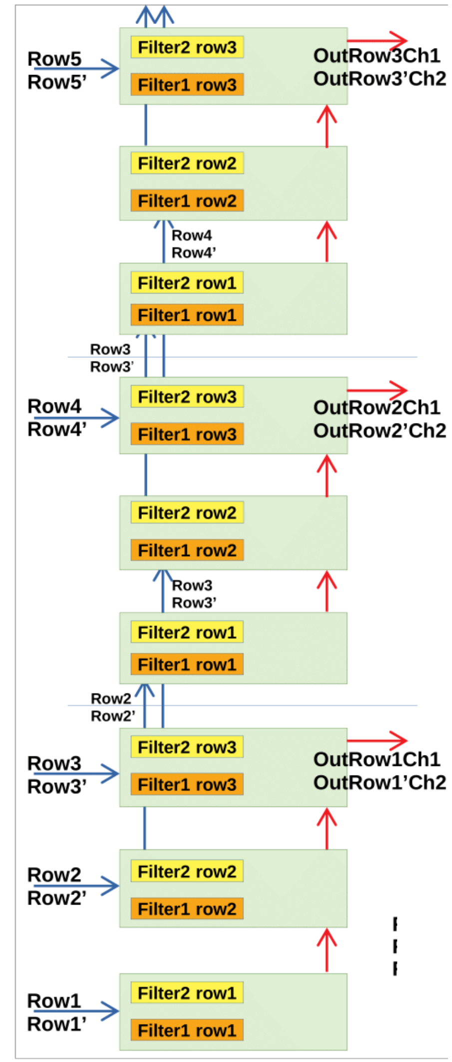

2D systolic array (Eyeriss)



conventional



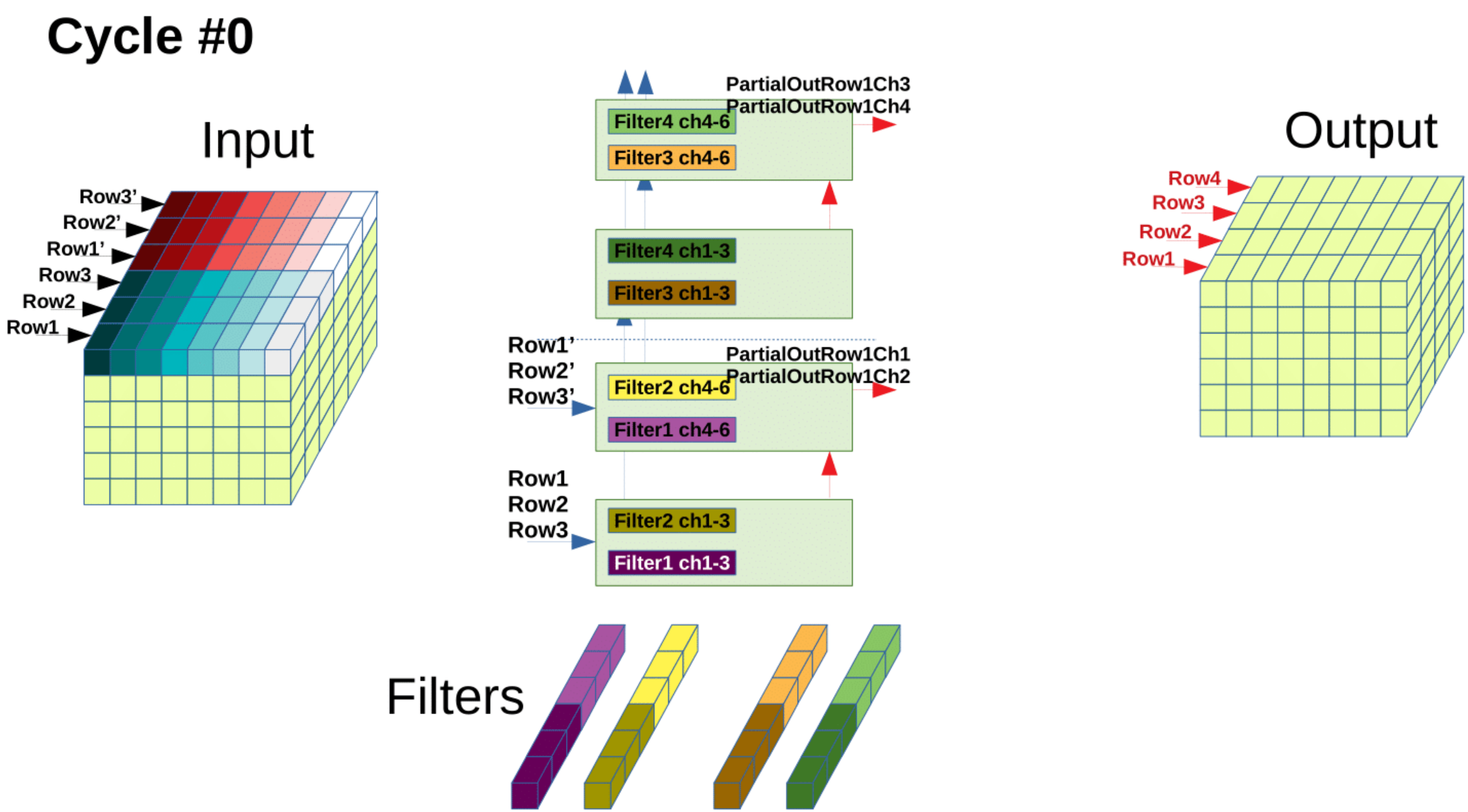

ours




depthwise convolution

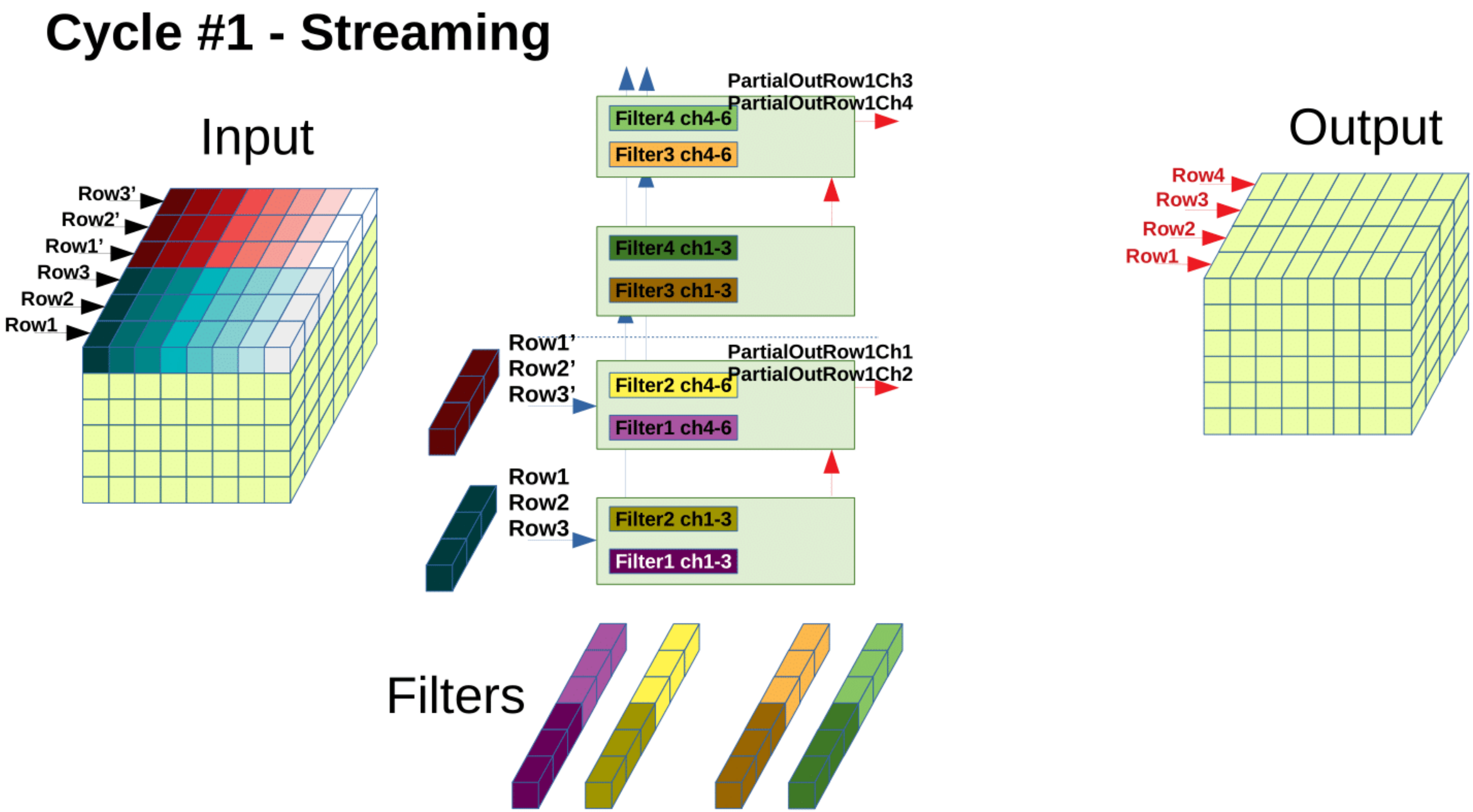
# Interconnect (2)

## 3- Point-wise

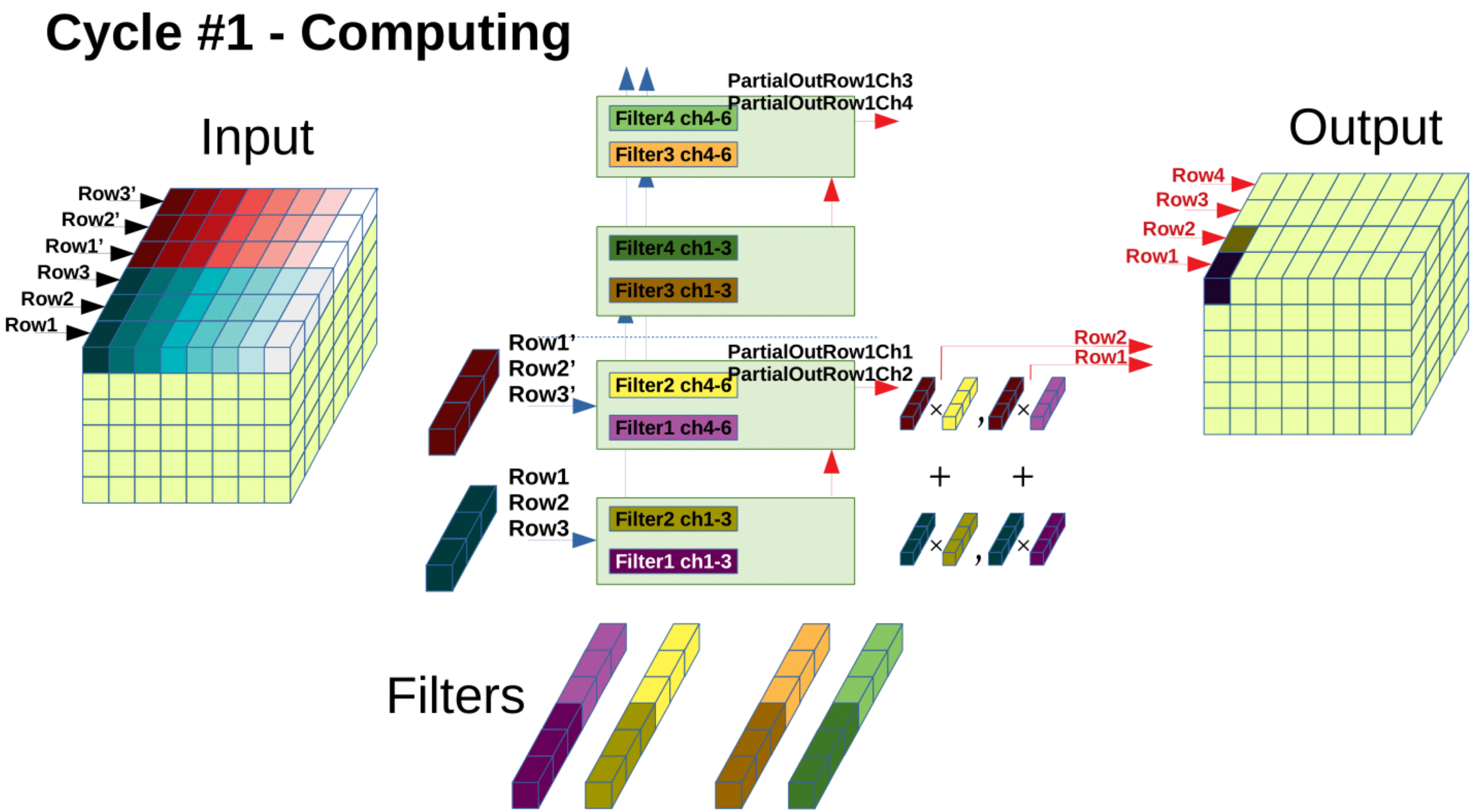


# Interconnect (2)

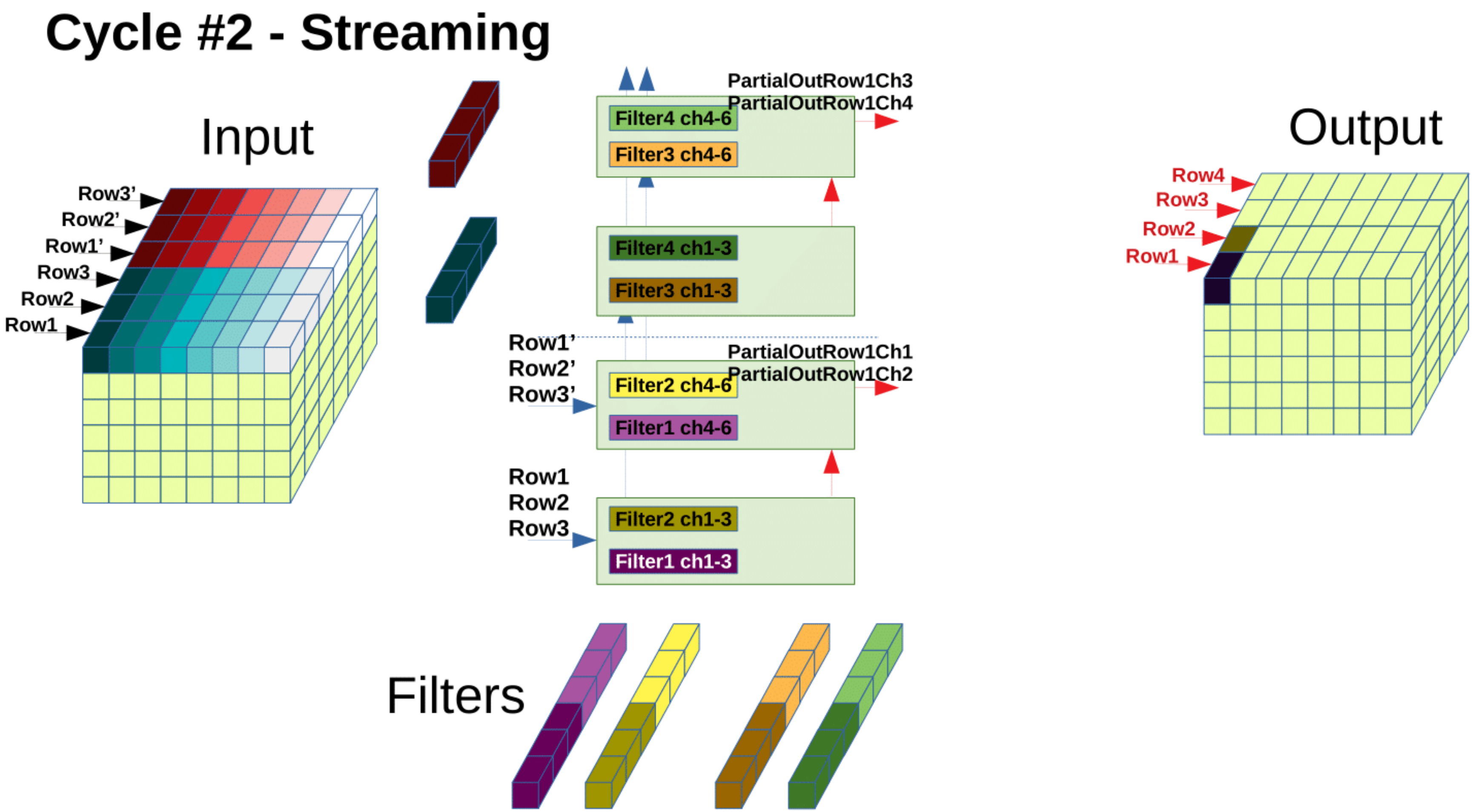
## 3- Point-wise



## 3- Point-wise

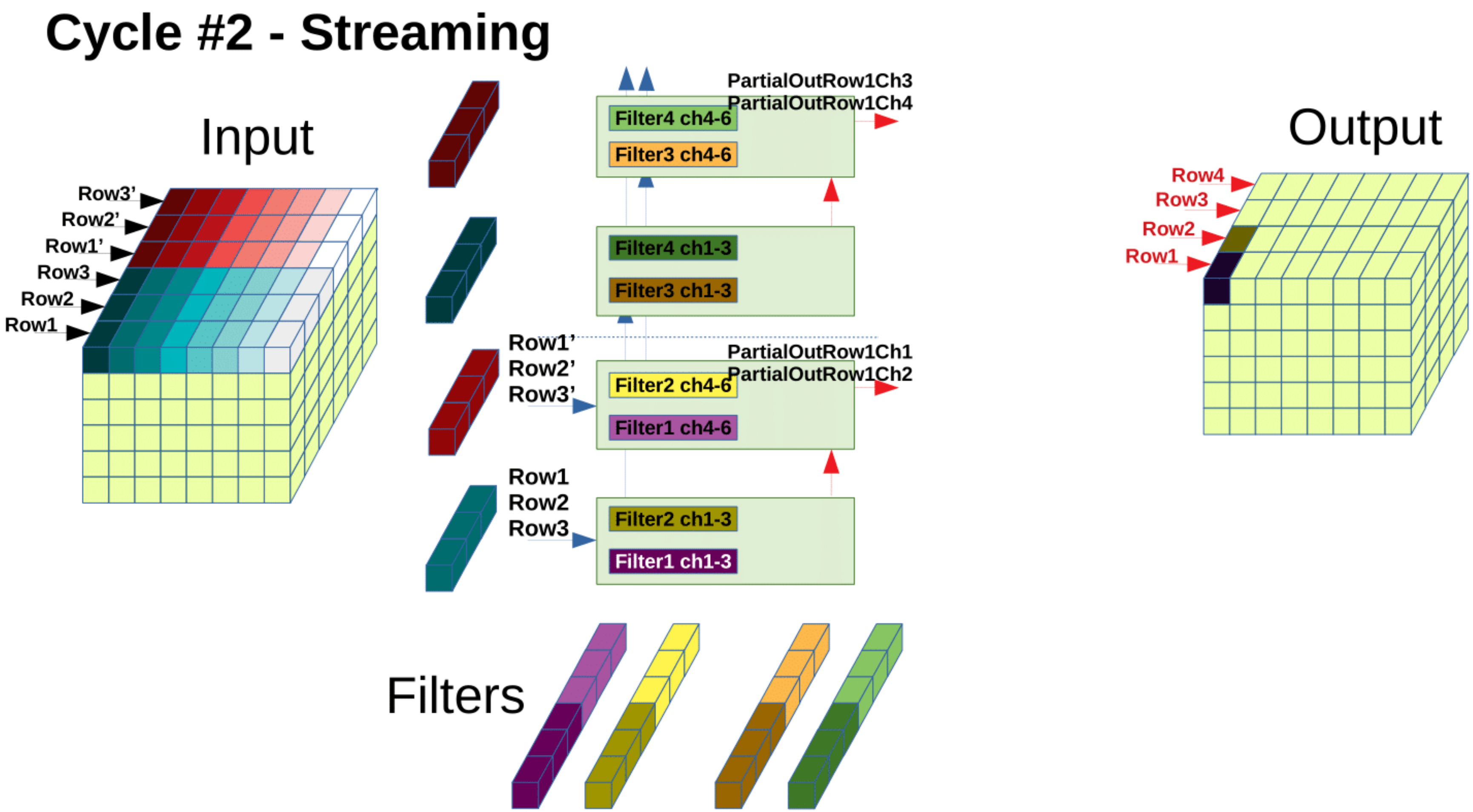

Cycle #1 - Computing
Input
Row3'
Row2'
Row1'
Row3
Row2
Row1
PartialOutRow1Ch3
PartialOutRow1Ch4
Filter4 ch4-6
Filter3 ch4-6
Filter4 ch1-3
Filter3 ch1-3
Row1'
Row2'
Row3'
PartialOutRow1Ch1
PartialOutRow1Ch2
Filter2 ch4-6
Filter1 ch4-6
Row1
Row2
Row3
Filter2 ch1-3
Filter1 ch1-3
Output
Row4
Row3
Row2
Row1
Row2
Row1
Filters

## 3- Point-wise


Cycle #2 - Streaming
Input
Row3'
Row2'
Row1'
Row3
Row2
Row1
PartialOutRow1Ch3
PartialOutRow1Ch4
Filter4 ch4-6
Filter3 ch4-6
Filter4 ch1-3
Filter3 ch1-3
Row1'
Row2'
Row3'
PartialOutRow1Ch1
PartialOutRow1Ch2
Filter2 ch4-6
Filter1 ch4-6
Row1
Row2
Row3
Filter2 ch1-3
Filter1 ch1-3
Filters
Output
Row4
Row3
Row2
Row1

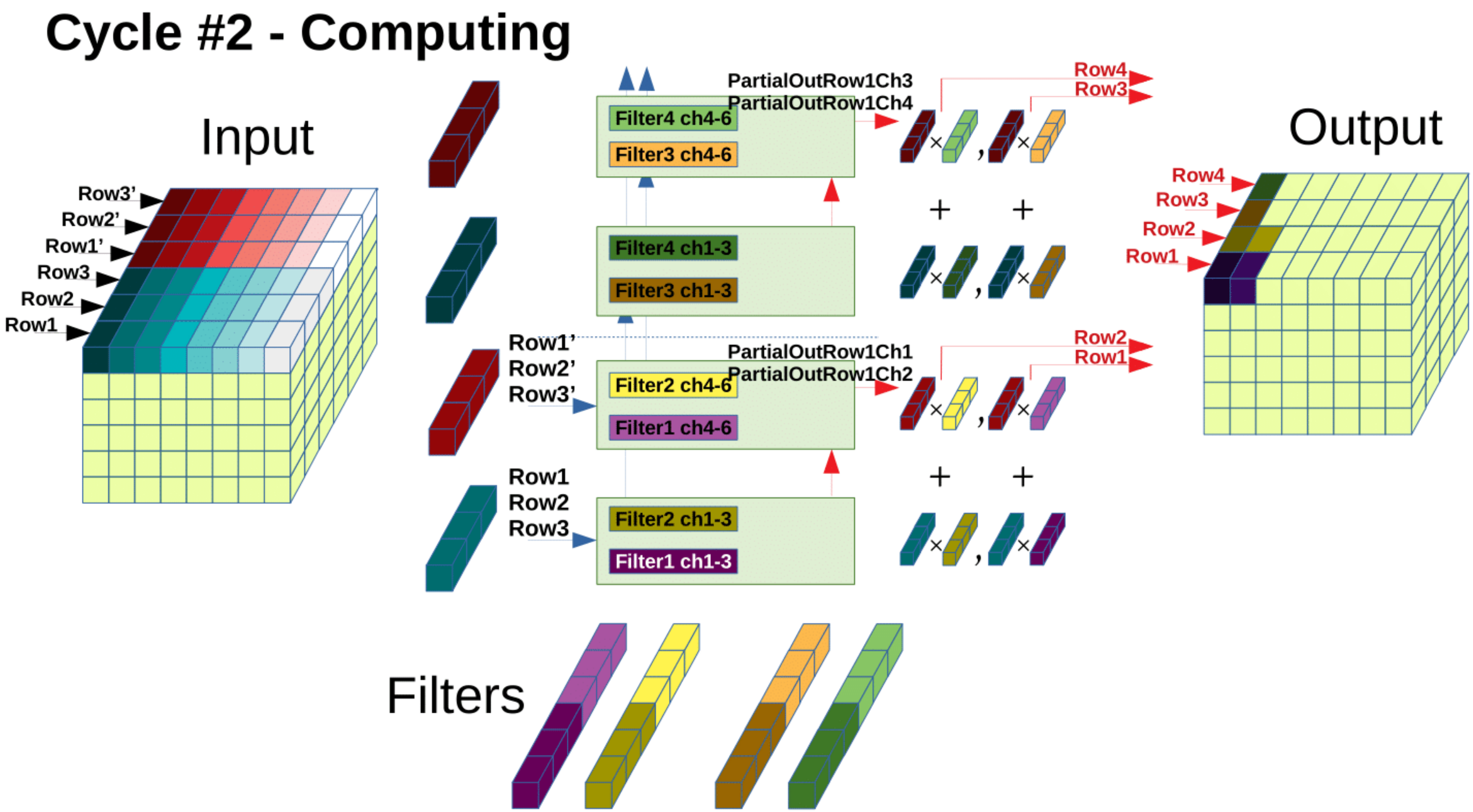
# Interconnect (2)

## 3- Point-wise

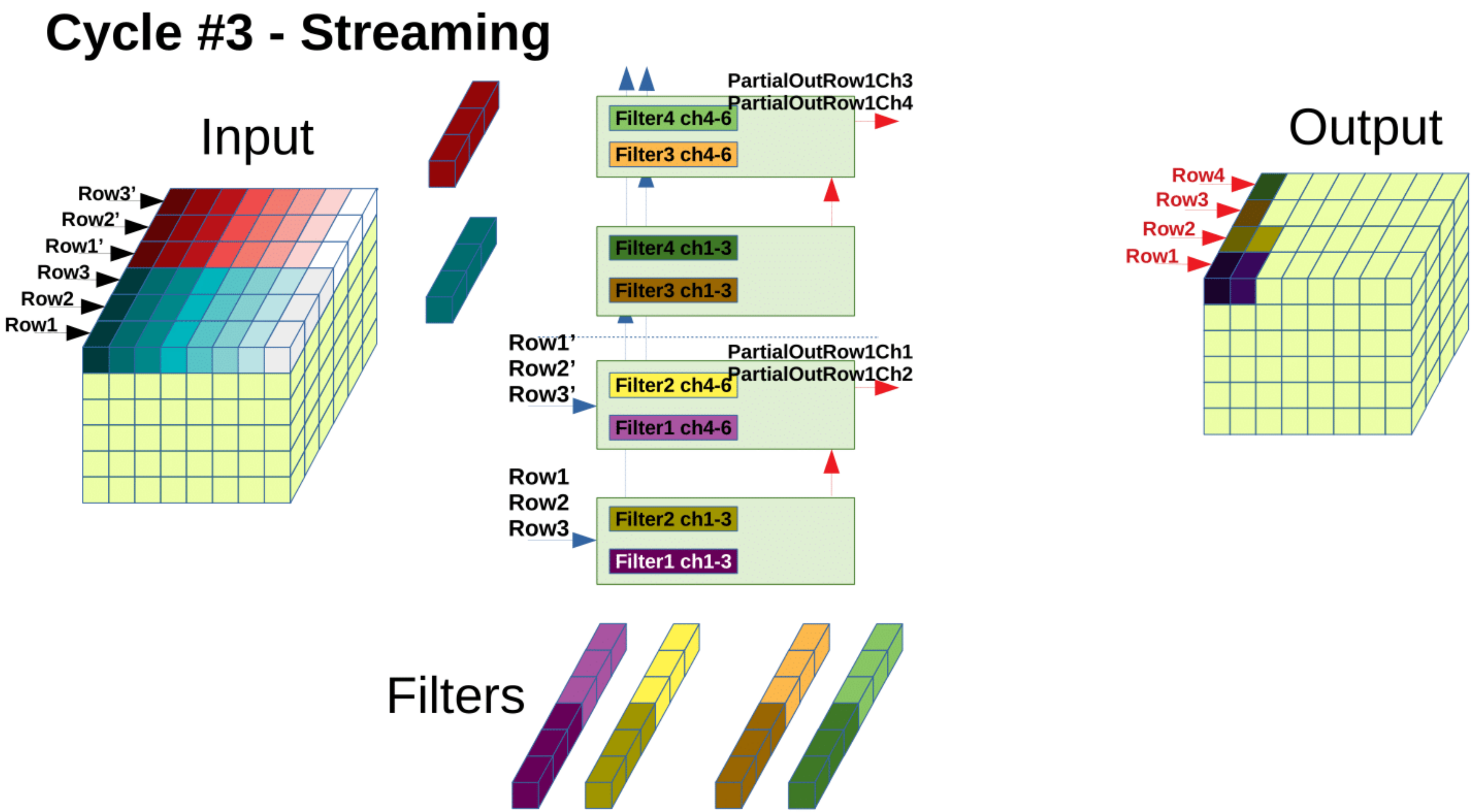


# Interconnect (2)

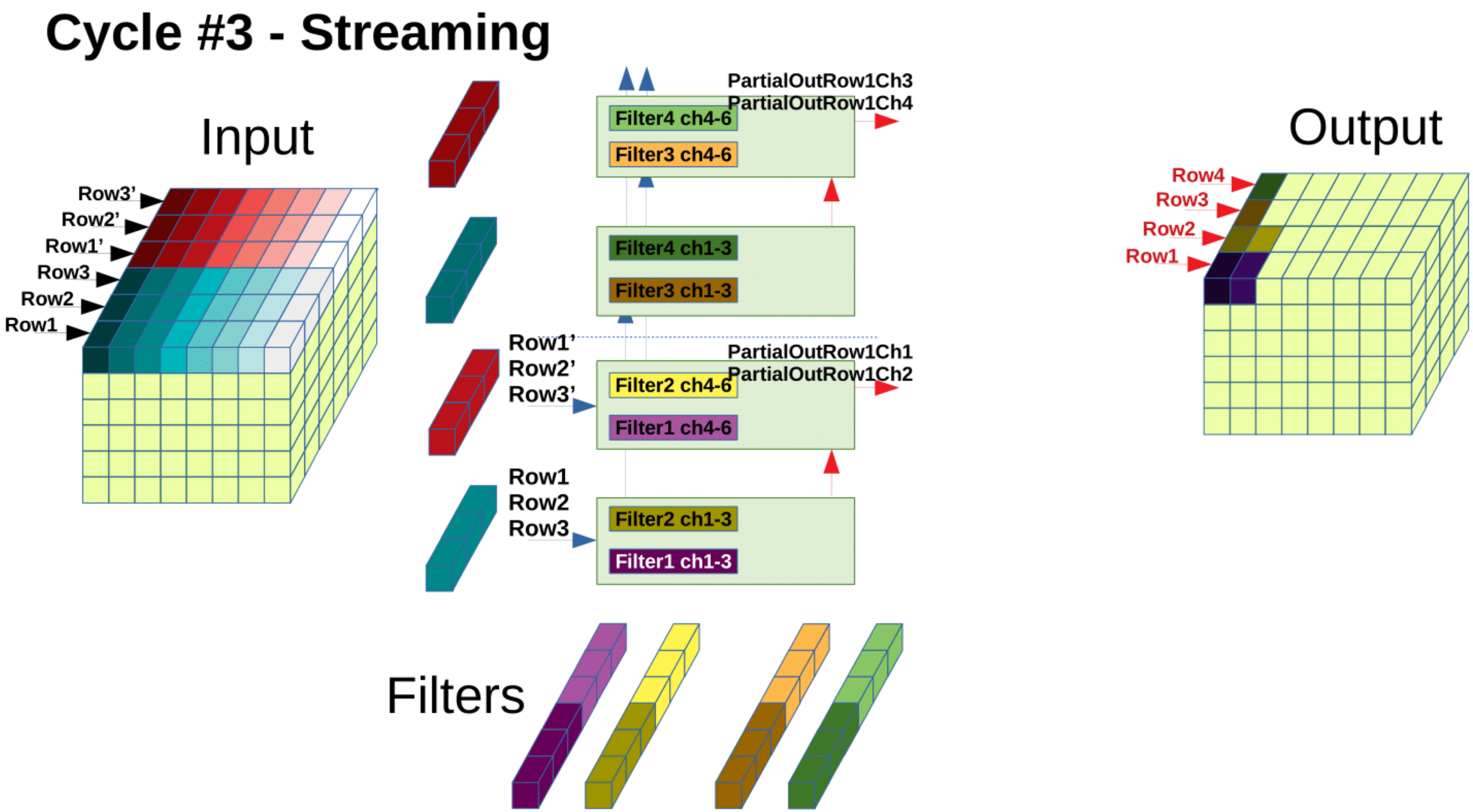
## 3- Point-wise



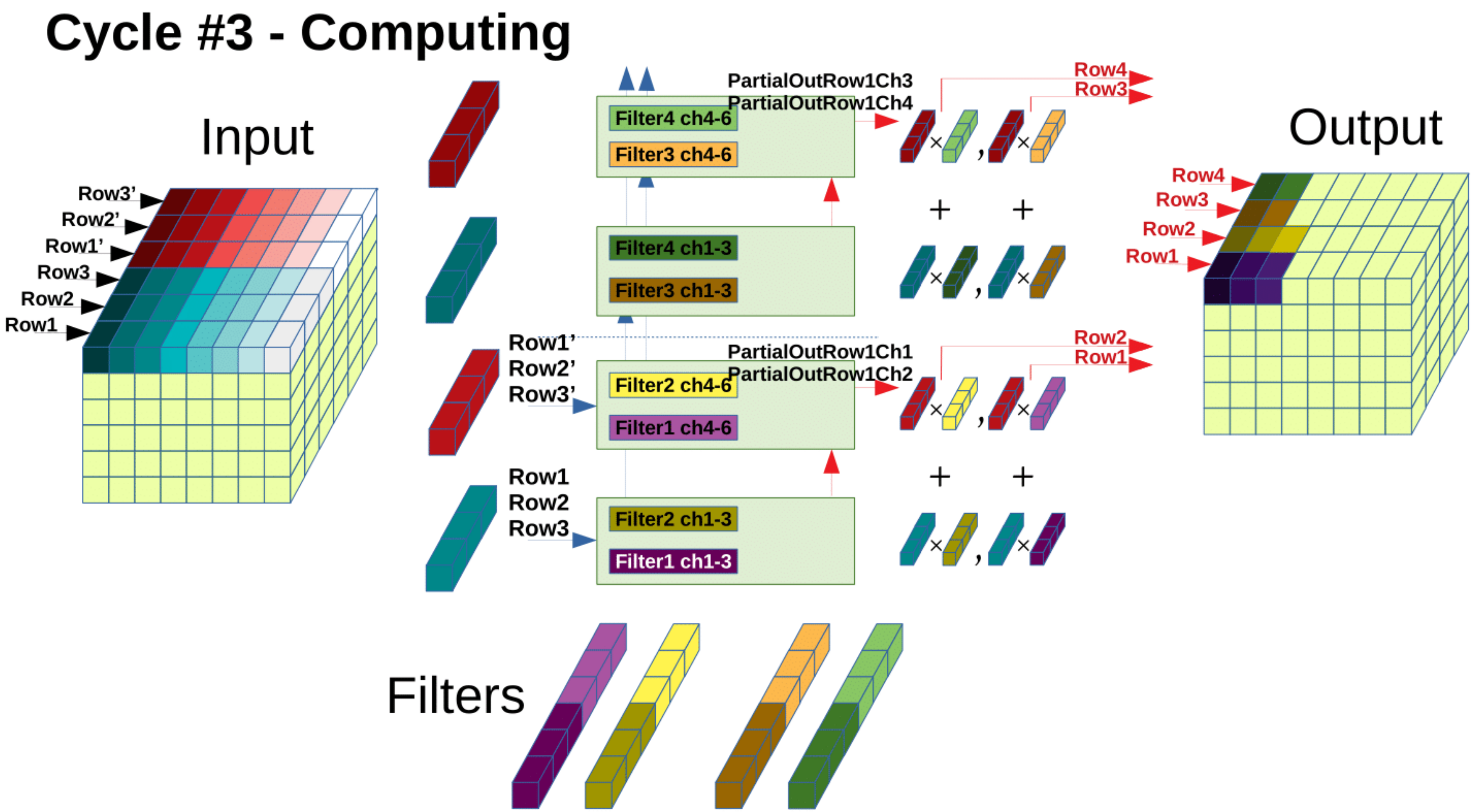
## 3- Point-wise


Cycle #3 - Streaming
Input
Row3'
Row2'
Row1'
Row3
Row2
Row1
PartialOutRow1Ch3
PartialOutRow1Ch4
Filter4 ch4-6
Filter3 ch4-6
Filter4 ch1-3
Filter3 ch1-3
Row1'
Row2'
Row3'
PartialOutRow1Ch1
PartialOutRow1Ch2
Filter2 ch4-6
Filter1 ch4-6
Row1
Row2
Row3
Filter2 ch1-3
Filter1 ch1-3
Filters
Output
Row4
Row3
Row2
Row1

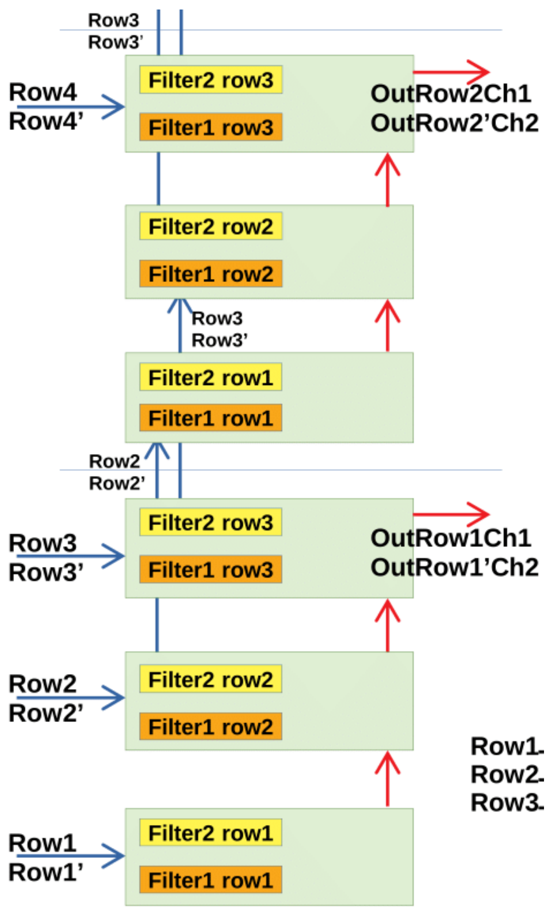
## 3- Point-wise


Cycle #3 - Streaming
Input
Row3'
Row2'
Row1'
Row3
Row2
Row1
PartialOutRow1Ch3
PartialOutRow1Ch4
Filter4 ch4-6
Filter3 ch4-6
Filter4 ch1-3
Filter3 ch1-3
Row1'
Row2'
Row3'
PartialOutRow1Ch1
PartialOutRow1Ch2
Filter2 ch4-6
Filter1 ch4-6
Row1
Row2
Row3
Filter2 ch1-3
Filter1 ch1-3
Filters
Output
Row4
Row3
Row2
Row1

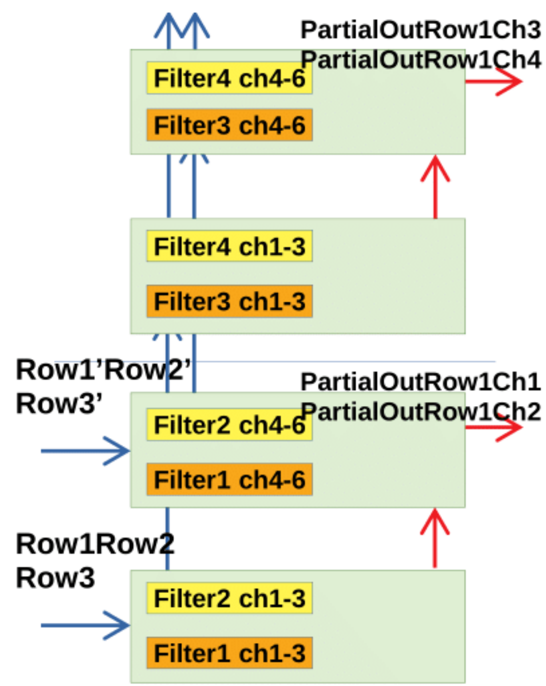
## 3- Point-wise


Cycle #3 - Computing
Input
Row3'
Row2'
Row1'
Row3
Row2
Row1
PartialOutRow1Ch3
PartialOutRow1Ch4
Filter4 ch4-6
Filter3 ch4-6
Filter4 ch1-3
Filter3 ch1-3
Row1'
Row2'
Row3'
PartialOutRow1Ch1
PartialOutRow1Ch2
Filter2 ch4-6
Filter1 ch4-6
Row1
Row2
Row3
Filter2 ch1-3
Filter1 ch1-3
Row4
Row3
Row2
Row1
Output
Row4
Row3
Row2
Row1
Filters

# Reuse




Depthwise Convolution (DW)




Pointwise Convolution (PW)

# Overview

- Introduction
- PIR-DSP Architecture
- Results
- Conclusion

# Area and Frequency

- SMIC 65-nm standard cell technology
  - Synopsis Design Compiler 2013.12

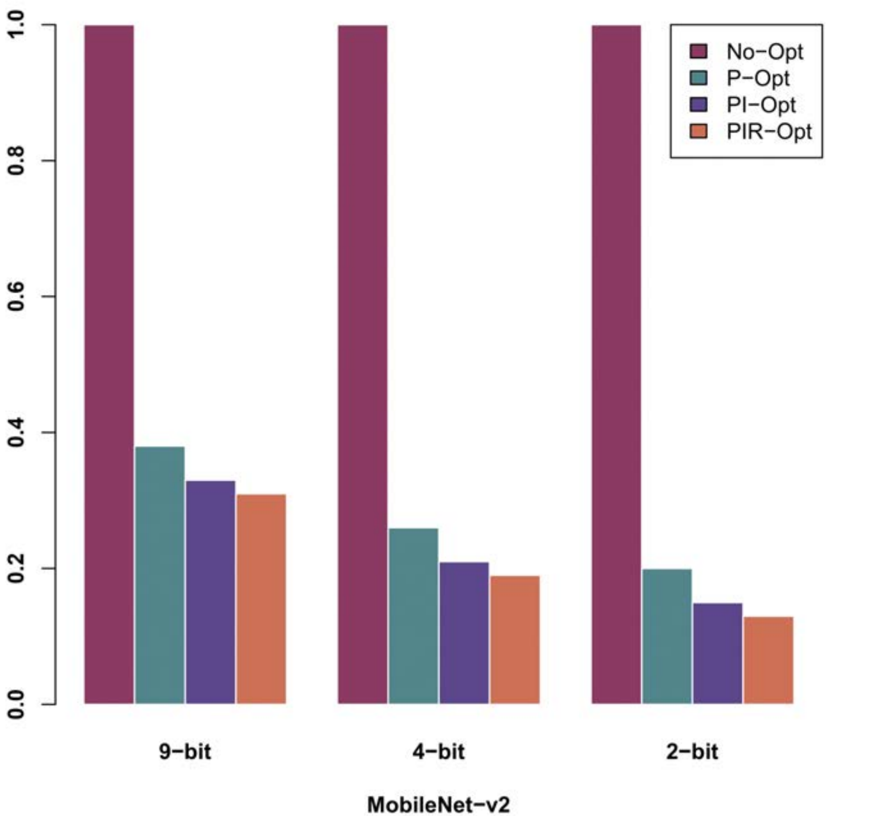
| Version | Area Ratio | Fmax |
|---|---|---|
| DSP48E2 | 1.0 | 463 |
| + M27×18C32D2 MAC-IP | 1.14 | 358 |
| + interconnect | 1.18 | 362 |
| + reuse | 1.28 | 357 |

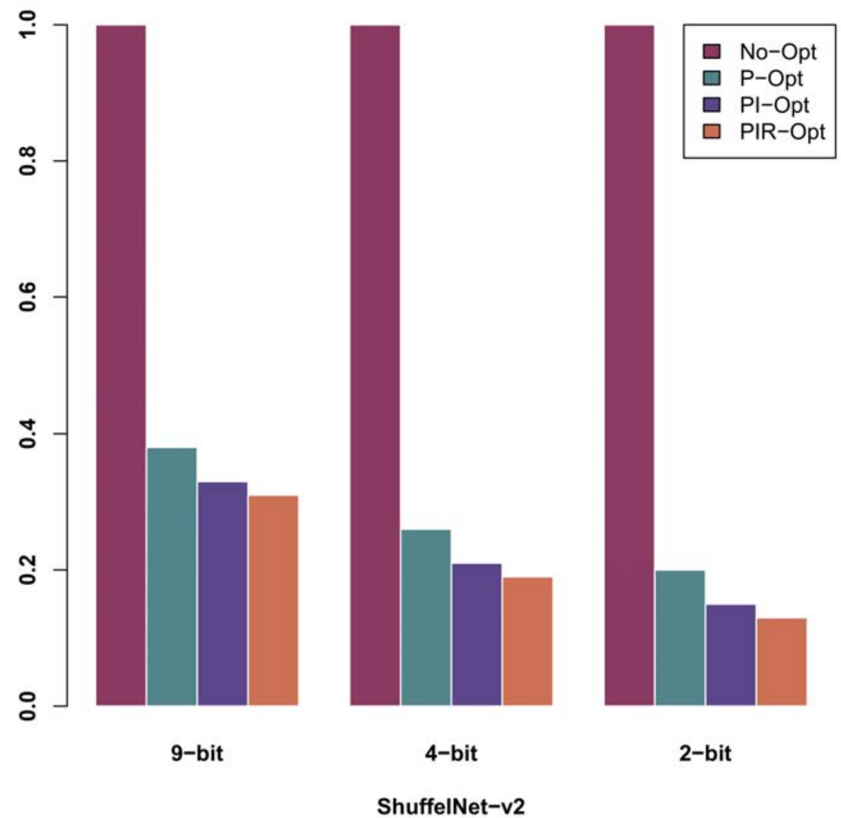
# Energy

› Other networks are similar

DATA MOVEMENT ENERGY RATIOS IN 65 NM TECHNOLOGY ($1\times = 90$FJ).

| **Energy** | FF | $SR_e$ | $RF_e$ | Chain | RF | SR | BRAM(B) | MAC |
|---|---|---|---|---|---|---|---|---|
| **Ratio** | 1 | 2 | 12.5 | 23 | 40 | 44 | 205 | 89-22 |



# Related Work

- Sits between Sharma (low-precision) and Boutros (high-precision)

| | **Bitfusion [56] ISCA'18** | **Ours** | **Boutros [44] FPL'18** | **Ours** |
|---|---|---|---|---|
| **Area** | 0.24 | 1 | 0.77 | 1 |
| **Performance Per Area** | | | | |
| 2x2 | 1 | 0.4 | | |
| 4x4 | 1 | 0.7 | 1 | 1.2 |
| 8x8 | 1 | 1.4 | 1 | 1.2 |
| 16x16 | | | 1 | 0.4 |
| 27x18 | | | 1 | 0.8 |

# Overview

- Introduction
- PIR-DSP Architecture
- Results
- Conclusion

# Summary

› Described optimizations to the DSP48 to support a range of low-precision DNNs and quantified their impact on performance

- Precision, Interconnect and Reuse
- designs are available at http://github.com/raminrasoulinezhad/PIR-DSP

› Future research

- Consider what we can do if we give up DSP48-like functionality
- Other interconnect optimisations

# AddNet: Deep Neural Networks using FPGA-Optimized Multipliers

*Julian Faraone, Martin Kumm Member, Martin Hardieck, Peter Zipf, Xueyuan Liu, David Boland, Philip H.W. Leong*

THE UNIVERSITY OF SYDNEY

# Overview

- Multipliers (and adders) play a key role in the implementation of DNNs
- This talk
  - Two speed multiplier with different critical paths for zero and non-zero recodings
  - PIR-DSP block to support a range of precisions
  - **AddNet** which uses k-levels of shifted values as multipliers
  - A fully pipelined DNN implementation with ternary coefficients

# Introduction

- Reconfigurable constant coefficient multipliers (RCCMs) implement y = cx for a fixed set of coefficients c using only adds and shifts e.g.

$$(x << 2) + (x << 1) = 6x$$

- Present FPGA logic element optimised RCCMs which implement large sets c with few resources
- When applied to neural networks (AddNet), achieve up to 50% resource savings over traditional 8-bit quantized networks

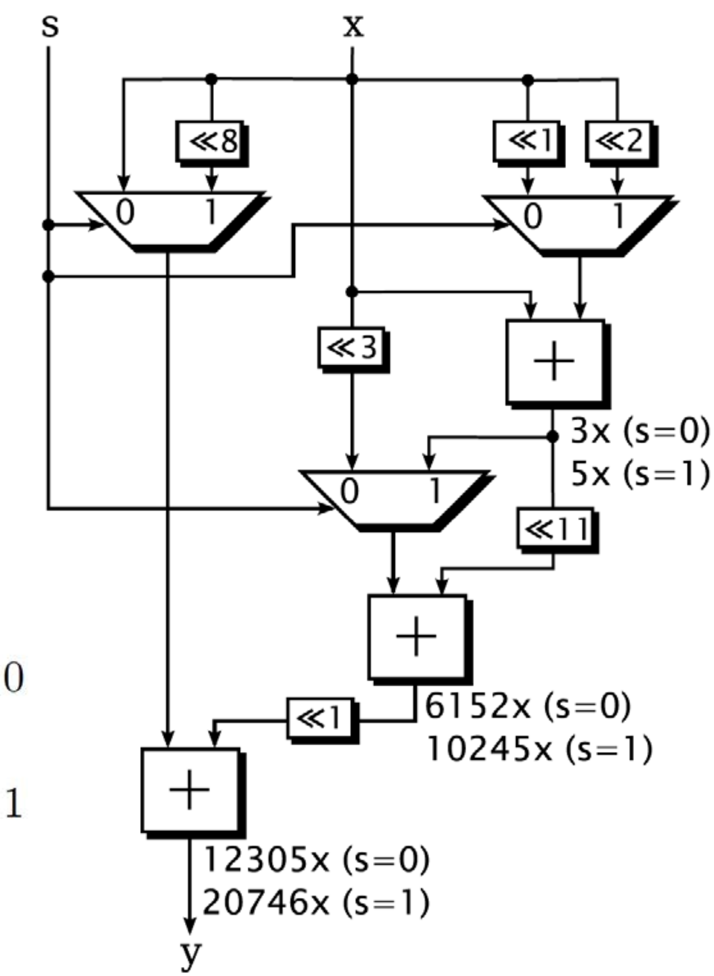
# Reconfigurable constant coefficient multipliers (RCCM)

- Another example has the coefficient set C={12305, 20746}
- Top adder computes

$$\begin{cases} x + (x << 1) = 3x & \text{if} \quad s = 0 \\ x + (x << 2) = 5x & \text{if} \quad s = 1 \end{cases}$$

- Bottom adder computes

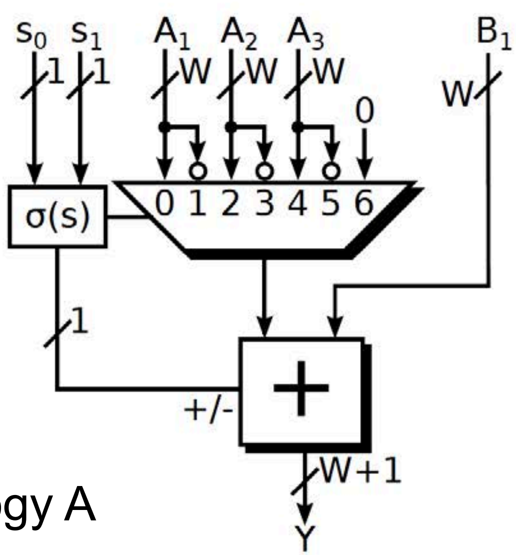
$$y = \begin{cases} x + ((x << 3) + ((x + x << 1) \\ << 11) << 1) = 12305x & \text{if} \quad s = 0 \\ (x << 8) + ((x + x << 2) + \\ ((x + x << 2) << 11) << 1) = 20746x & \text{if} \quad s = 1 \end{cases}$$



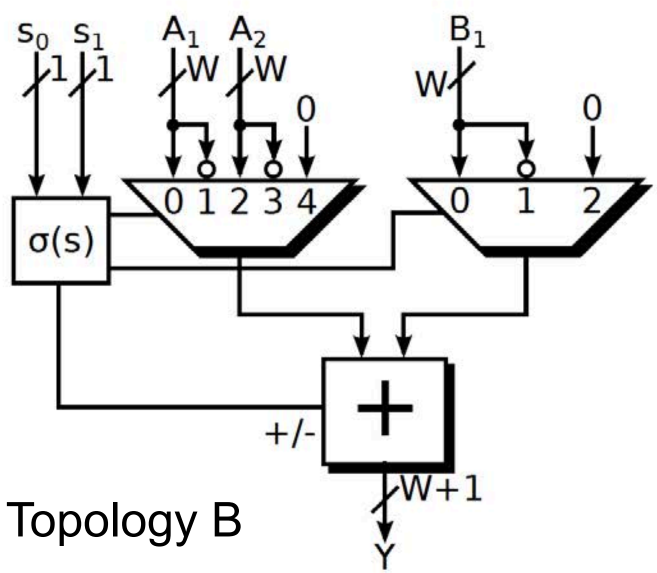
# Base Topologies

- Topology A has potentially a larger coefficient set $\pm A_p \pm B_1$ $\sigma(s)$ chooses operation
- Topology B allows symmetric coefficients around 0 as $A_p - B_1 = -(A_p - B_1)$
- Maximum possible set size is $2^{w_s}$

| RCCM | $w_s$ | #unique coefficient sets |
|---|---|---|
| 2-Add | 4 | 1145 |
| 3-Add | 6 | 44198 |
| 4-Add | 8 | 4040952 |



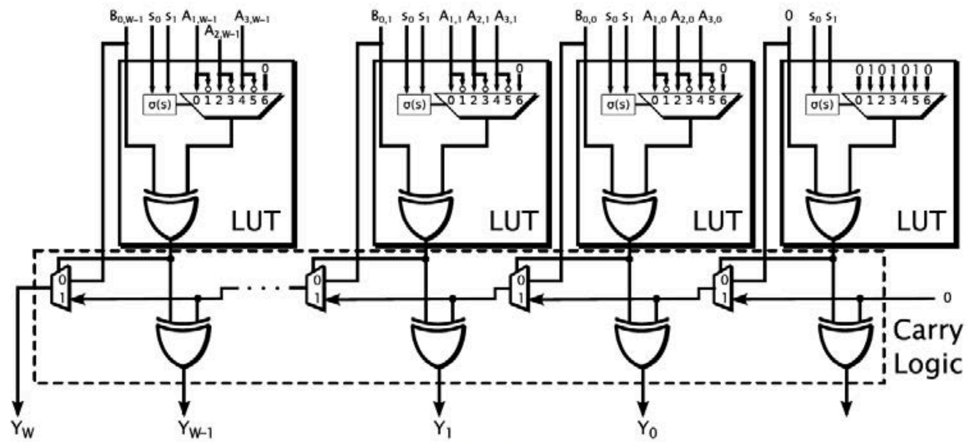

Topology A



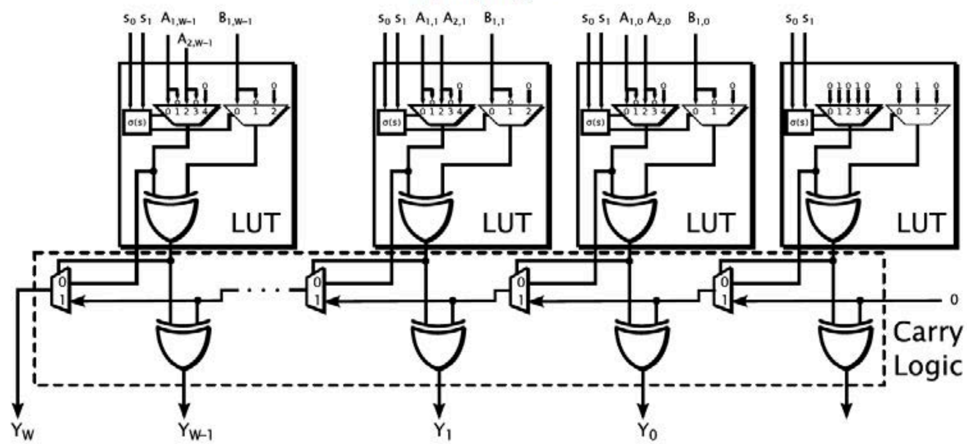

Topology B

# Bit Level Slice Mapping

Works for any 6-LUT FPGA such as Xilinx Ultrascale or Intel Stratix X

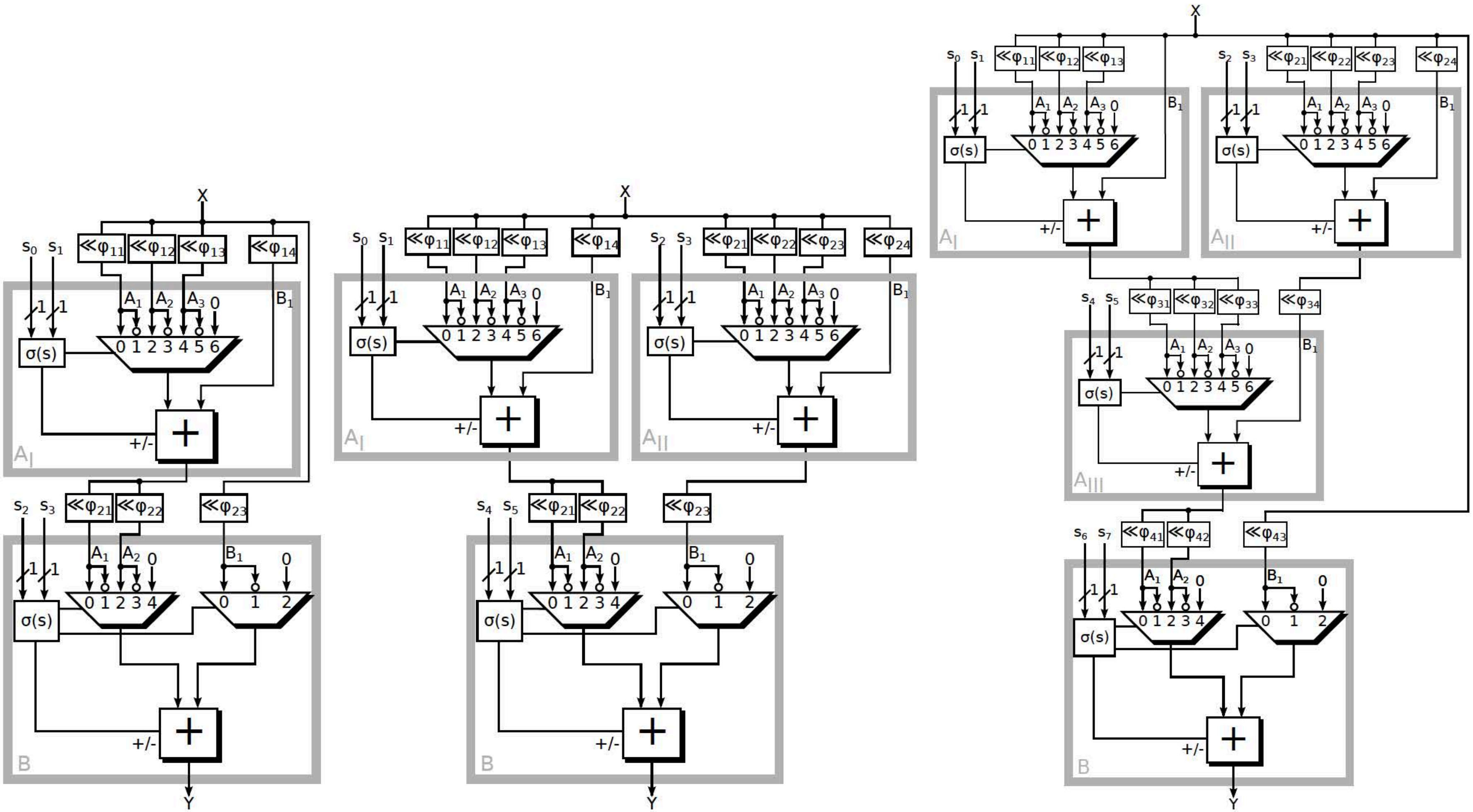



(a) Topology A




(b) Topology B

# RCCM Circuits




(a) 2-Add RCCM (b) 3-Add RCCM (c) 4-Add RCCM

# Distribution Matching

› Weights in a DNN follow a distribution (Gaussian-like)

› RCCM coefficients can be chosen to match

- Find best 5 in terms of Kullback-Leibler divergence, then choose set with largest number of coefficients

$$D_{\mathrm{KL}}(P\|R) = \sum_{i=0}^{N-1} P(i) \log \frac{P(i)}{R(i)}$$

| RCCM | type | $s_1\ s_0$ 00 | 01 | 10 | 11 | shifts $\varphi_{i1}$ | $\varphi_{i2}$ | $\varphi_{i3}$ | $\varphi_{i4}$ |
|---|---|---|---|---|---|---|---|---|---|
| 2-Add | $A_{\mathrm{I}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | $B1$ | 0 | 1 | 3 | 2 |
| | B | -$A1$+$B1$ | -$A2$+$B1$ | $A1$-$B1$ | $A2$-$B1$ | 0 | 3 | 2 | – |
| 3-Add | $A_{\mathrm{I}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | -$A2$+$B1$ | 0 | 2 | 3 | 3 |
| | $A_{\mathrm{II}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | -$A1$+$B1$ | 0 | 1 | 3 | 0 |
| | B | -$A1$+$B1$ | -$A2$+$B1$ | $A1$-$B1$ | $A2$-$B1$ | 0 | 3 | 0 | – |
| 4-Add | $A_{\mathrm{I}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | -$A3$+$B1$ | 0 | 1 | 3 | 0 |
| | $A_{\mathrm{II}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | -$A2$+$B1$ | 0 | 1 | 3 | 1 |
| | $A_{\mathrm{III}}$ | $A1$+$B1$ | $A2$+$B1$ | $A3$+$B1$ | -$A1$+$B1$ | 0 | 1 | 3 | 3 |
| | B | -$A1$+$B1$ | -$A2$+$B1$ | $A1$-$B1$ | $A2$-$B1$ | 0 | 3 | 1 | – |

| arch. | #coeff | Coefficient set (±) |
|---|---|---|
| 2-Add | 15 | 0 1 2 8 28 36 44 92 |
| 3-Add | 59 | 0 1 2 3 4 5 6 7 9 10 12 13 14 16 23 29 30 32 63 69 70 72 87 93 94 96 119 125 126 128 |
| 4-Add | 207 | 0 1 2 4 5 7 8 9 11 13 14 15 16 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 36 37 38 39 40 46 48 54 58 64 69 70 71 74 75 76 78 80 81 82 84 85 87 94 96 102 114 118 126 134 142 150 166 174 182 190 194 198 206 214 222 230 238 246 258 262 270 278 286 302 310 318 326 334 382 398 446 450 526 566 574 582 614 622 654 662 670 686 694 710 766 782 830 1214 |

# DNN Training using AddNet

- Trained using straight-through estimator (STE)

**Algorithm 1** Training a CNN using AddNet representations

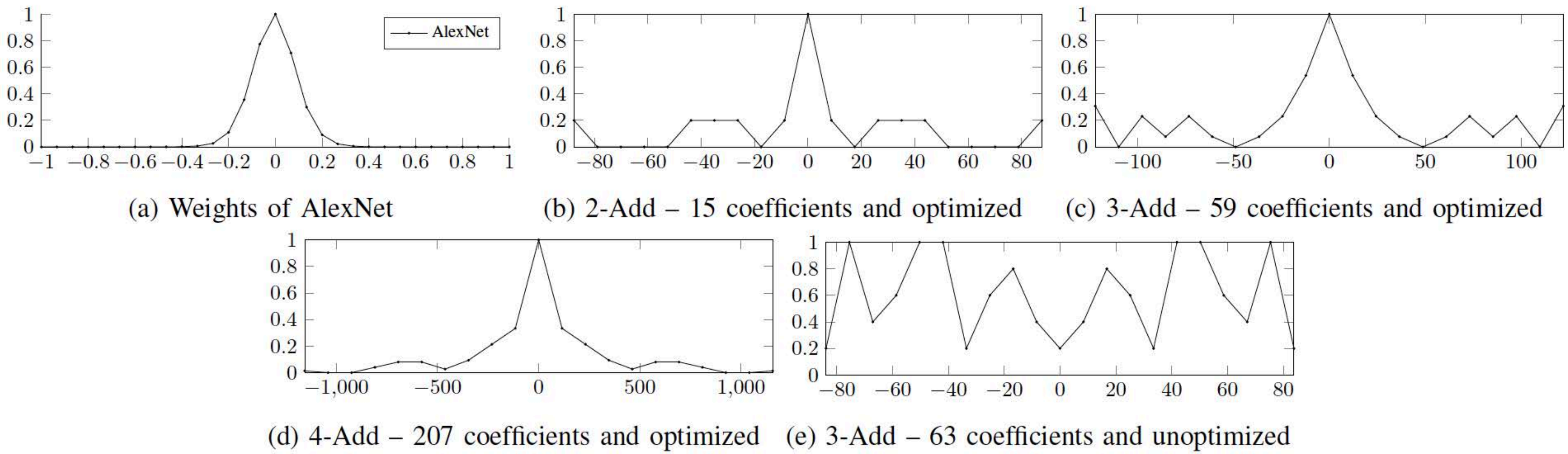
**Initialize:** Pre-train model
Set adder size
c = DistributionMatching($\sigma(s)$) using (4)
**Inputs:** Minibatch of inputs & targets $(I, Y)$, Loss function $L(Y, \hat{Y})$, current weights $\boldsymbol{W_t}$ and learning rate, $\gamma_t$
**Outputs:** Updated $\boldsymbol{W_{t+1}}$, $\boldsymbol{\lambda_{t+1}}$ and $\gamma_{t+1}$

***Forward propagation:***
**for** l=1 to L **do**
  $\boldsymbol{Q}_l$ = Quantize($\boldsymbol{W}_l$) using (5) and (7)
**end for**
$\hat{Y}$ = **ForwardPropagation** $(I, Y, \boldsymbol{Q}_l)$ using (7)

***Backward Propagation:***
$\frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}$ = **WeightBackward**($\boldsymbol{Q}_l, \frac{\partial \hat{L}}{\partial \hat{Y}}$)
$\frac{\partial \hat{L}}{\partial \boldsymbol{\lambda}_l}$ = **ScalarBackward**($\frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}, \boldsymbol{\lambda}_l, \frac{\partial \hat{L}}{\partial \hat{Y}}$)
$\boldsymbol{W_{t+1}}$ = **UpdateWeights**($\boldsymbol{W_t}, \frac{\partial \hat{L}}{\partial \boldsymbol{Q}_l}, \gamma$)
$\boldsymbol{\lambda_{t+1}}$ = **UpdateScalars**($\boldsymbol{\lambda_t}, \frac{\partial \hat{L}}{\partial \boldsymbol{\lambda}_l}, \gamma$)
$\gamma_{t+1}$ = **UpdateLearningRate**($\gamma_t, t$)

# Distribution-Optimised Coefficient Sets (for Gaussian)

› AlexNet top1/top5: 53.8%/76.9% (unoptimised) vs 55.8%/79.8% (optimised)




(a) Weights of AlexNet

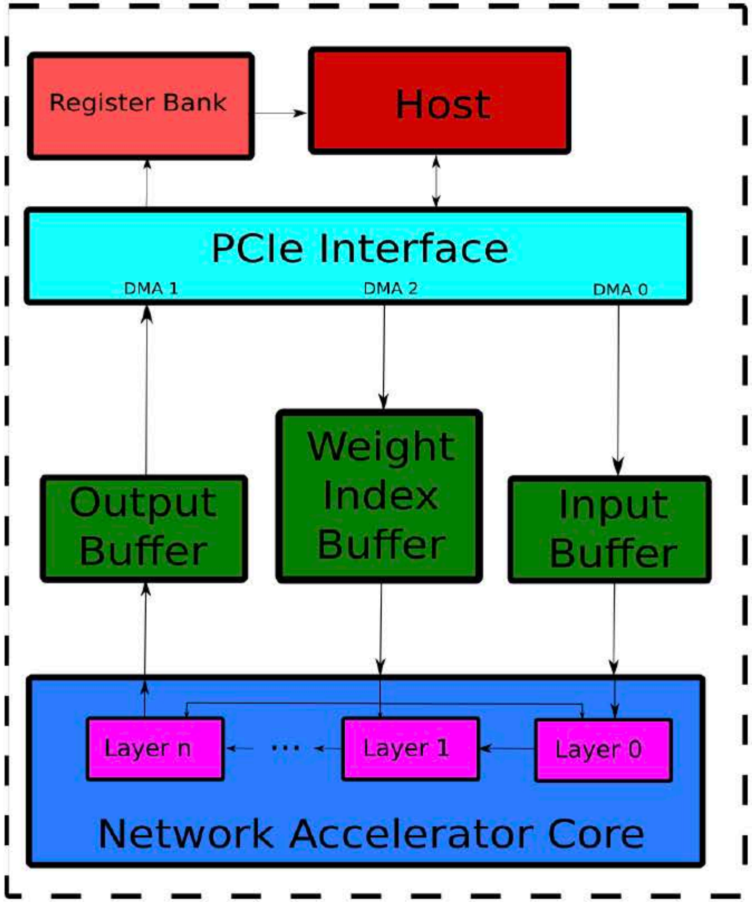
(b) 2-Add – 15 coefficients and optimized

(c) 3-Add – 59 coefficients and optimized

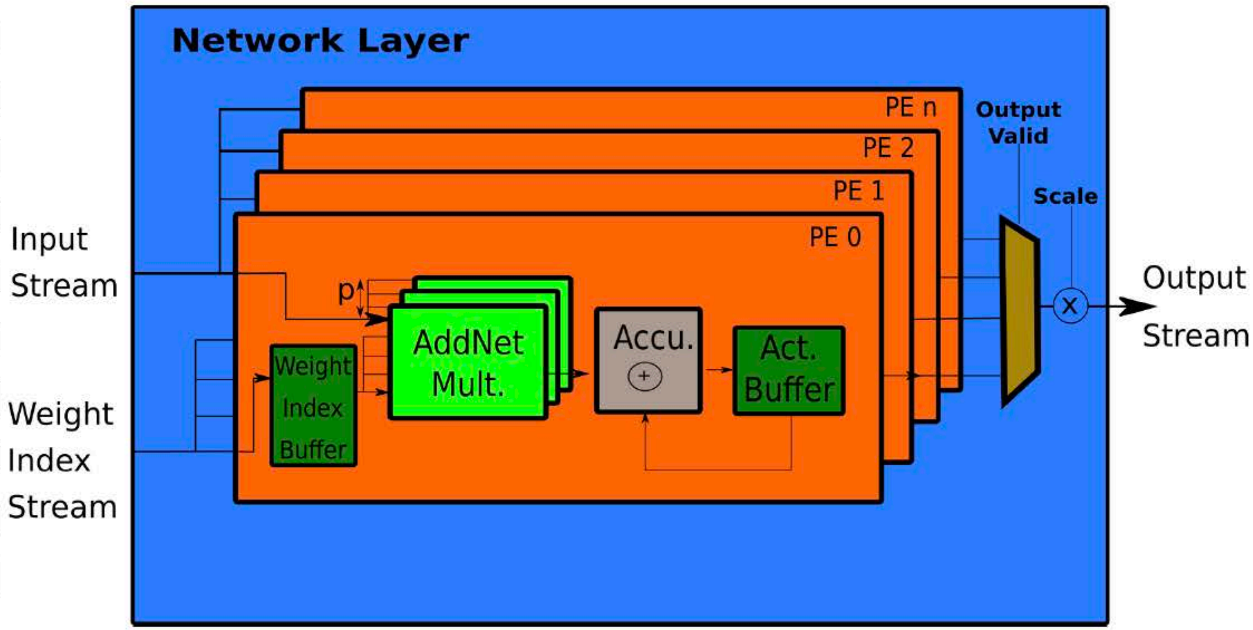
(d) 4-Add – 207 coefficients and optimized

(e) 3-Add – 63 coefficients and unoptimized

# Implementation

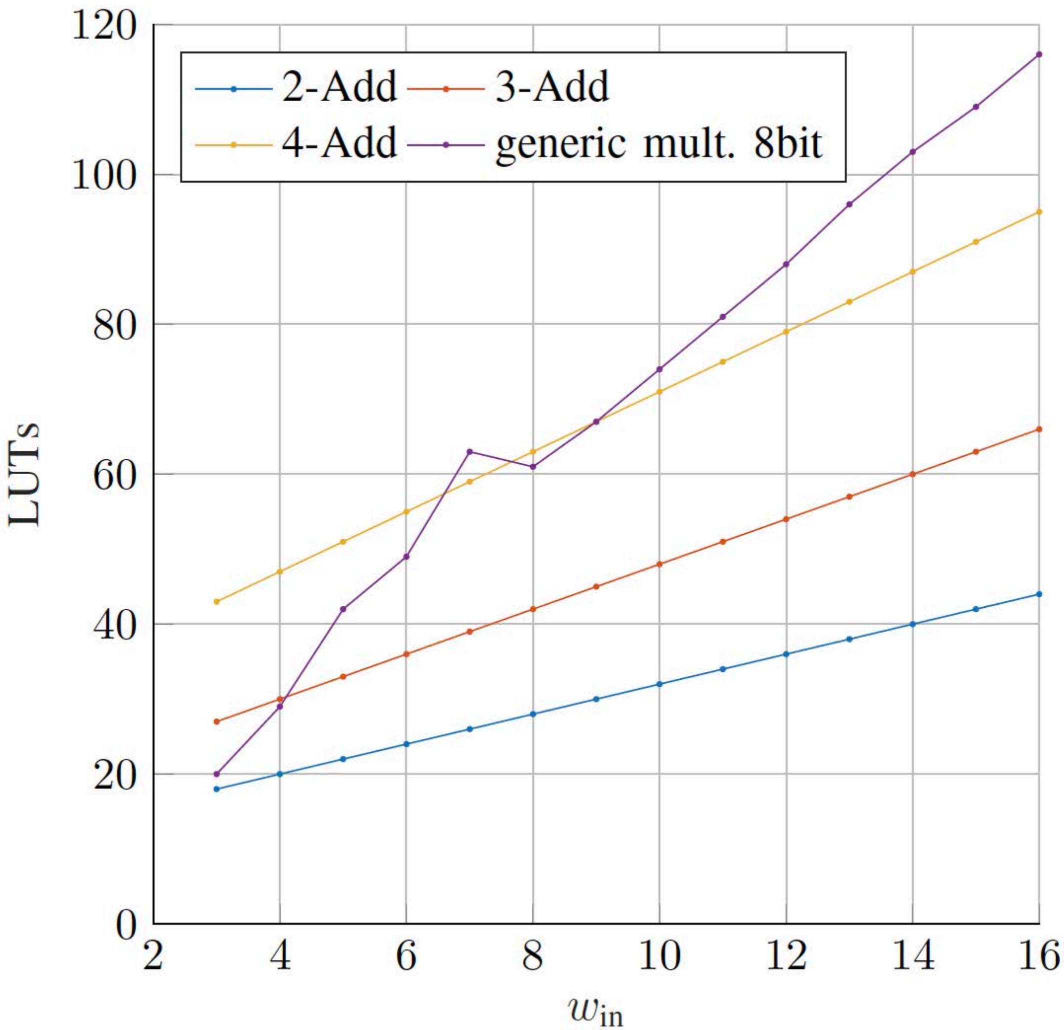



(a) AlphaData Library System Design




(b) Network Layer Accelerator Core including AddNet multipliers

# Resource Utilization




| Type | 2-Add | 3-Add | 4-Add |
|---|---|---|---|
| Original | 447.43 MHz | 483.09 MHz | 342.82 MHz |
| Pipelined | 770.42 MHz | 578.03 MHz | 623.83 MHz |

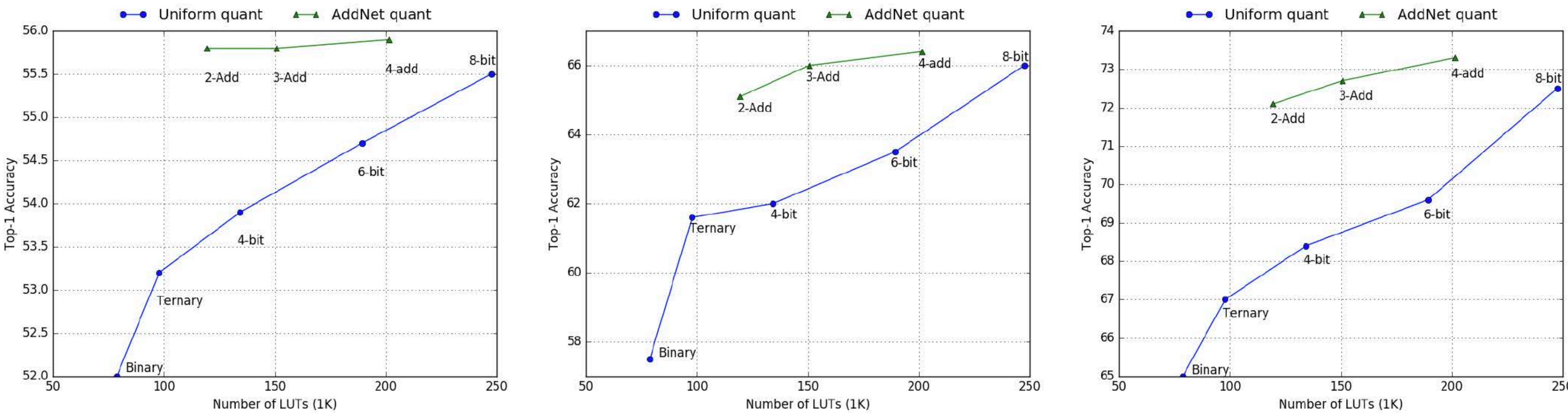
THE UNIVERSITY OF SYDNEY

# Power Consumption

| Xilinx KU115 | Architecture | 2-Add | 3-Add | 4-Add | 8-bit disab. [41]. | 8-bit enab. [41] |
|---|---|---|---|---|---|---|
| BRAM (2160) | Conv Layer | 1154 | 1154 | 1154 | 1154 | 170 |
| | AlexNet | 1365 | 1557 | 1557 | 1557 | 1229 |
| DSP (5520) | Conv Layer | 48 | 48 | 48 | 48 | 96 |
| | AlexNet | 48 | 48 | 48 | 48 | 3760 |
| LUTs (663K) | Conv Layer | 187.0 | 205.6 | 255.8 | 383.0 | 36.2 |
| | AlexNet | 331.7 | 372.8 | 430.7 | 467.1 | 128.8 |
| Estim. Power | Conv Layer | 7.6W | 7.6W | 7.8W | 7.5W | 7.2W |
| | AlexNet | 39W | 44W | 48W | 52W | 29W |

- 8-bit disab. = DSP disabled, 8-bit enab. = DSP enabled

# Accuracy-Area Tradeoff Compared with Uniform Multipliers




(a) AlexNet (b) ResNet-18 (c) ResNet-50

# Summary

- Described AddNet which uses constant coefficient multipliers to save area
- Showed it is advantageous over uniform multipliers
- Uses trainability of DNNs to match computer architecture

# Unrolling Ternary Networks

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

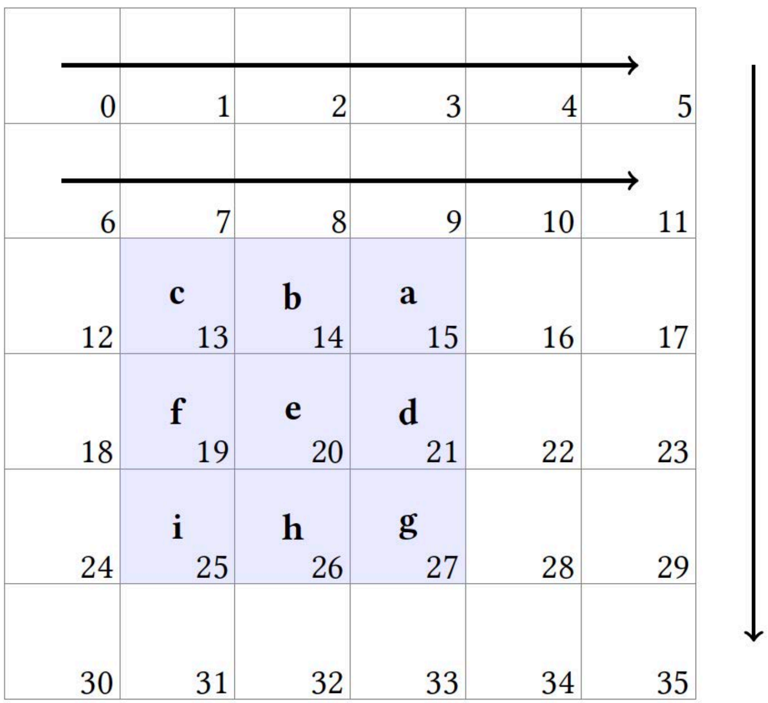
# Overview

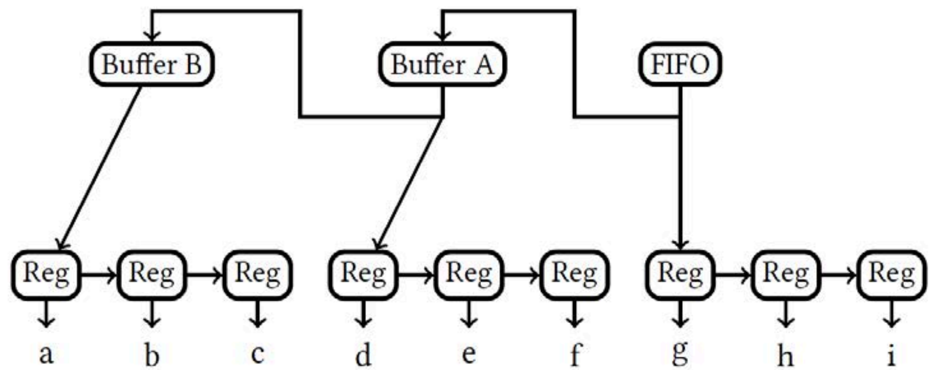
- Multipliers (and adders) play a key role in the implementation of DNNs
- This talk
  - Two speed multiplier with different critical paths for zero and non-zero recodings
  - PIR-DSP block to support a range of precisions
  - AddNet which uses k-levels of shifted values as multipliers
  - **A fully pipelined DNN** implementation with ternary coefficients

# Introduction

- Not possible 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

Implement Pipelined 3x3 Convolution




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

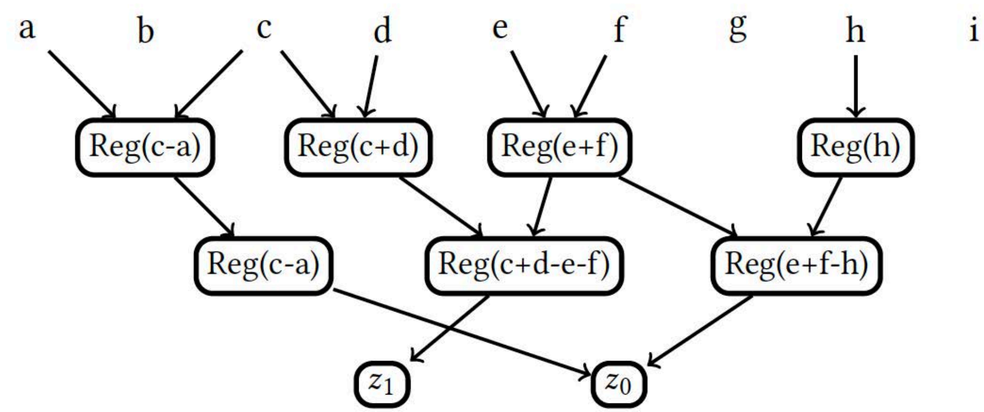
- Weights are ternary
  - So multiplication with $\pm 1$ is either addition or subtraction
  - Multiplication with 0 makes matrix sparse

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

# Common Subexpression Elimination

- 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 $z_1 = c + d - e - f$

# Common Subexpression Elimination (RPAG)

› RPAG Algorithm

- Greedy algorithm for the related Multiple Constant Multiplication problem
- Looks at all the outputs of a matrix-vector multiplication and calculates the minimal tree depth, d, required to get the results
- Tries to determine the minimum number of terms needed at depth d − 1 to compute the terms at depth d and iterates until d=1 (whole tree generated)

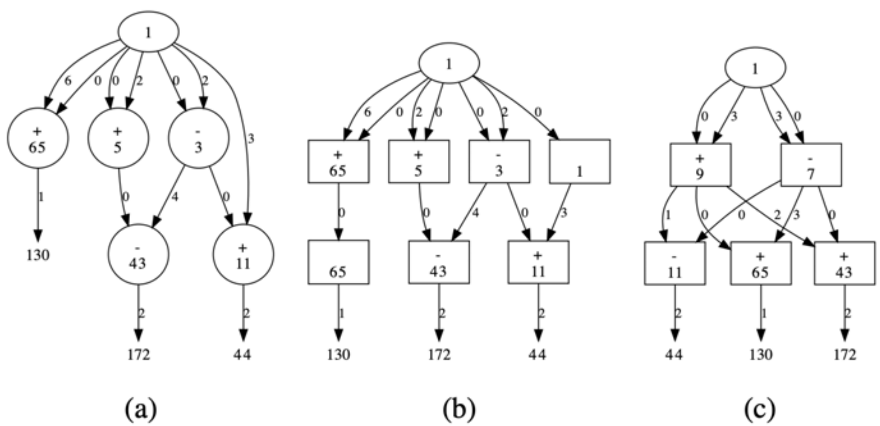



Fig. 1. Graph realizations of coefficient set $\{44, 130, 172\}$: (a) Adder graph obtained by $H_{cub}$ AD min, (b) PAG using ASAP pipelining, (c) optimal PAG

## Top-down CSE (TD-CSE)

- Builds multiple adder trees from the inputs to the outputs by creating an adder each iteration
- Count frequency of all size 2 subexpressions, replace most frequent ($x_6$=$x_2$+$x_3$)

$$\mathbf{y} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_2 + x_3 \\ x_0 + x_2 + x_3 + x_4 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_0 + x_2 + x_3 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \end{pmatrix}.$$

$$\mathbf{y} = \begin{pmatrix} x_6 \\ x_0 + x_4 + x_6 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_0 + x_6 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \end{pmatrix}.$$

# Bottom-up CSE (BU-CSE)

- Starts at the outputs and works back to the inputs
- More computation than TD-CSE but can find larger common subexpressions
- Largest common subexpression is then selected to be removed e.g. $x_6 = x_0 + x_2 + x_3$ appears twice and is added to the bottom row

$$
\mathbf{y} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} = \begin{pmatrix} x_2 + x_3 \\ x_0 + x_2 + x_3 + x_4 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_0 + x_2 + x_3 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \end{pmatrix}. \qquad \mathbf{y} = \begin{pmatrix} x_2 + x_3 \\ x_4 + x_6 \\ x_1 + x_4 + x_5 \\ x_1 + x_5 \\ x_6 \\ x_0 + x_3 \\ x_1 + x_4 + x_5 \\ x_0 + x_2 + x_3 \end{pmatrix}.
$$

(1) Compute the number of common terms for each pair of vectors and store this as the *pattern matrix*

(2) Find the largest value in the pattern matrix and the vectors it corresponds to

(3) Remove that subexpression from all matching vectors following the process described for the example in Equation 8

(4) Update the *pattern matrix*

(5) Go to step 2 until the largest value in the *pattern matrix* is 1

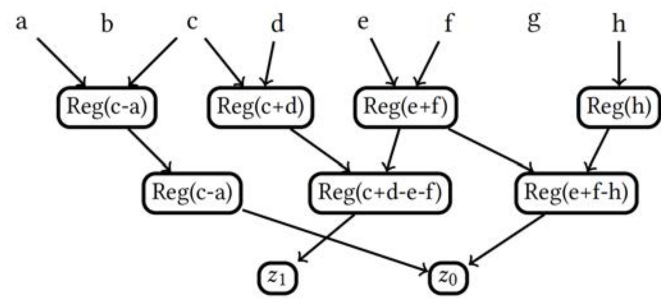
# Comparison of CSE Techniques for all Layers

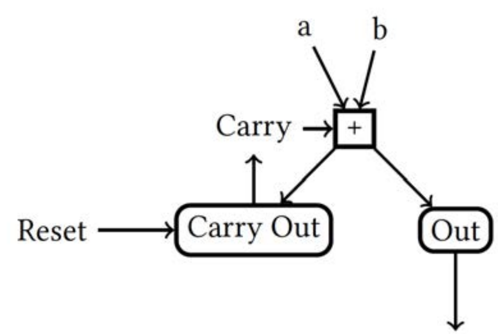
| Layer | Method | Adds | Regs | Adds+Regs | Time(s) | Mem(GB) | CLB/148K | FF/2.4M | LUTS/1.2M | P&R(hrs) |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | None | 731 | 137 | 868 | - | - | 1400 | 8723 | 8272 | 0.5 |
| | RPAG | 451 | 31 | **482** | 64 | 0.008 | 894 | 5764 | 6260 | 0.48 |
| | TD-CSE | 295 | 304 | 599 | 0.4 | 0.029 | - | - | - | - |
| | BU-CSE | 295 | 321 | 616 | 0.5 | 0.03 | 820 | 4499 | 5230 | 0.45 |
| 2 | None | 8432 | 249 | 8681 | - | - | 15231 | 119848 | 116345 | 1.08 |
| | TD-CSE | 3782 | 1517 | 5299 | 24 | 0.1 | - | - | - | - |
| | BU-CSE | 3686 | 858 | **4544** | 64 | 0.17 | 10258 | 71908 | 66131 | 0.93 |
| 3 | None | 17481 | 491 | 17972 | - | - | 15171 | 102657 | 77743 | 1.9 |
| | TD-CSE | 8466 | 2299 | 10765 | 89 | 0.18 | - | - | - | - |
| | BU-CSE | 8492 | 1878 | **10370** | 545 | 1.1 | 8772 | 61965 | 36611 | 1.13 |
| 4 | None | 36155 | 586 | 36741 | - | - | 30536 | 206940 | 164458 | 4.25 |
| | TD-CSE | 17143 | 4214 | 21357 | 873 | 0.63 | - | - | - | - |
| | BU-CSE | 17309 | 3056 | **20365** | 2937 | 6.6 | 16909 | 118476 | 73581 | 2.68 |
| 5 | None | 71050 | 1198 | 72248 | - | - | 18414 | 165794 | 85743 | 3.86 |
| | TD-CSE | 32829 | 6830 | 39659 | 3088 | 1.2 | - | - | - | - |
| | BU-CSE | 33026 | 6109 | **39135** | 25634 | 44 | 7579 | 89820 | 39805 | 1.72 |
| 6 | None | 144813 | 1270 | 146083 | - | - | 35117 | 335134 | 180402 | 11.15 |
| | TD-CSE | 62653 | 13852 | 76505 | 26720 | 4.8 | - | - | - | - |
| | BU-CSE | 63832 | 10103 | **73935** | 147390 | 191.0 | 13764 | 160634 | 74696 | 3.08 |

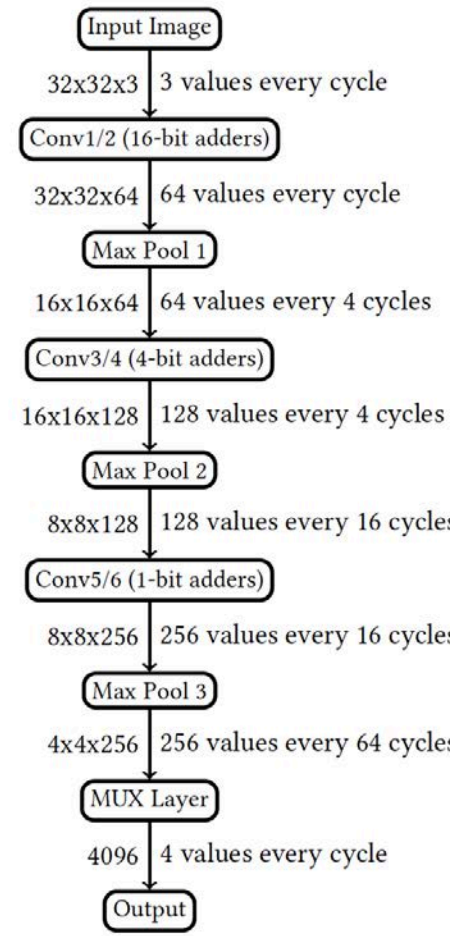
- RPAG too computationally expensive for layers 2-6

# 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 $p \geq 1$ 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



# 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

| 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

| 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 |

# 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 $\epsilon$ to control sparsity

| TNN Type | $\epsilon$ | 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 | $\epsilon$ | 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% |

THE UNIVERSITY OF SYDNEY

# Improvement in using CSE

| Layer | % decrease in Adds+Regs | % decrease in CLBs | %decrease in FFs | % decrease in LUTs |
|---|---|---|---|---|
| 1 | -29.0 | -41.4 | -48.4 | -36.8 |
| 2 | -47.7 | -32.6 | -40.0 | -43.2 |
| 3 | -42.3 | -42.1 | -39.6 | -52.9 |
| 4 | -44.6 | -44.6 | -42.3 | -55.3 |
| 5 | -45.8 | -58.8 | -45.8 | -53.6 |
| 6 | -49.4 | -60.8 | -52.1 | -58.6 |

# 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% ) |

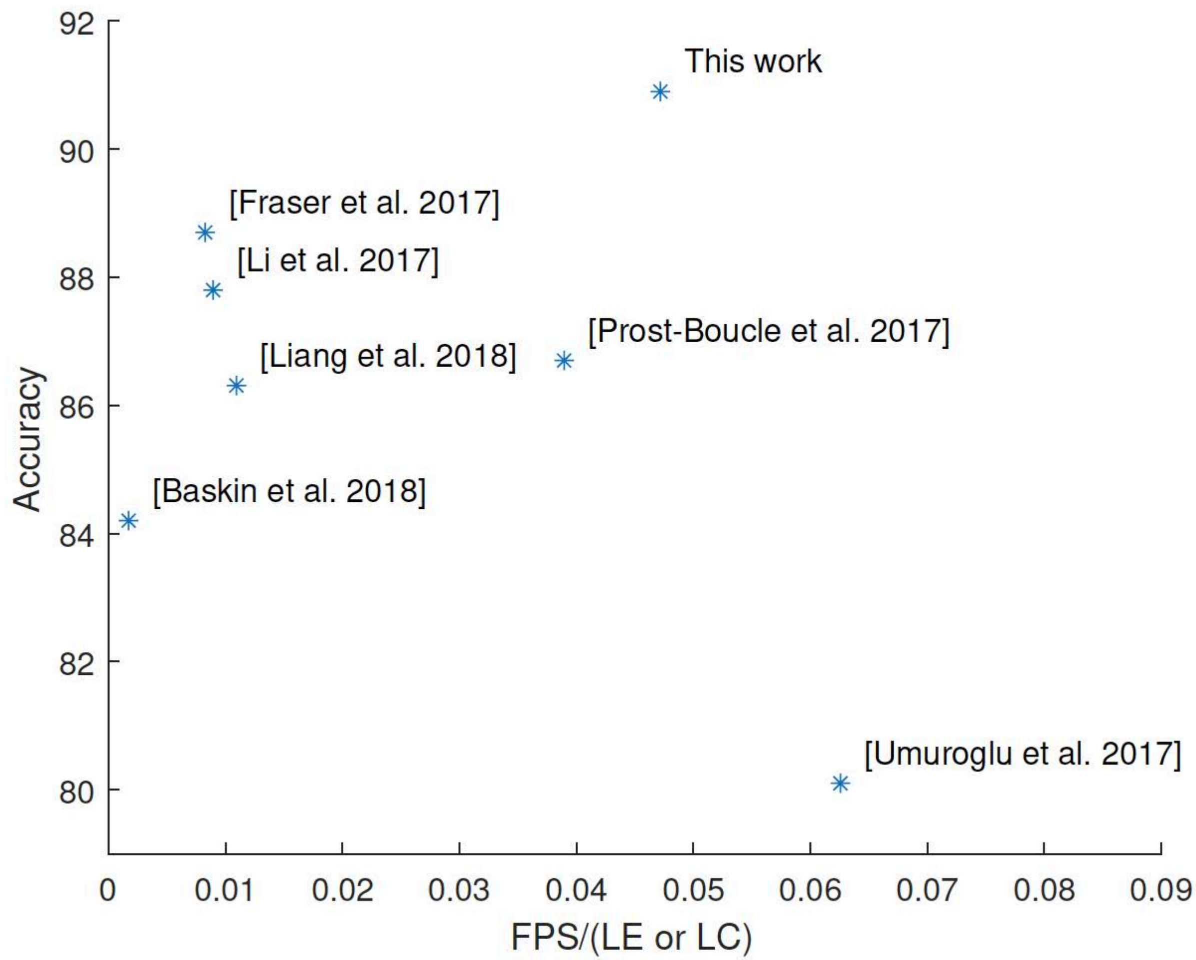
# Accuracy

## Comparison with ASIC and FPGA implementations

| Reference | Hardware ($mm^2$,nm,LE[5]/LC[5] $\times 10^6$) | 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



THE UNIVERSITY OF SYDNEY

# 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

THE UNIVERSITY OF SYDNEY

# Summary of the Four Techniques

- Multipliers form the basis for the computational part of ML
- Presented a number of different techniques to trade off flexibility, throughput and area
- First three are applicable to ASICs or FPGA hard blocks but unrolled ternary is FPGA-specific

| | Flexibility | Throughput | Area |
|---|---|---|---|
| Two speed | Normal | Normal | Normal |
| PIR | High | High | High |
| AddNet | Reduced | High | Low |
| Unrolled ternary | Ternary | High | Low |

# References

[1] D. J. M. Moss, D. Boland, and P. H. W. Leong. A two-speed, radix-4, serial–parallel multiplier. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(4):769–777, April 2019. (doi:10.1109/TVLSI.2018.2883645)

[2] SeyedRamin Rasoulinezhad, Hao Zhou, Lingli Wang, and Philip H.W. Leong. PIR-DSP: An FPGA DSP block architecture for multi-precision deep neural networks. In *Proc. IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–8, 2019. (doi:10.1109/FCCM.2019.00015)

[3] Julian Faraone, Martin Kumm, Martin Hardieck, Peter Zipf, Xueyuan Liu, David Boland, and Philip H.W. Leong. AddNet: Deep neural networks using FPGA-optimized multipliers. *IEEE Transactions on VLSI Systems*, page to appear (accepted 11 Aug 2019), 2019. (doi:10.1109/TVLSI.2019.2939429)

[4] 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.

https://phwl.github.io/talks

THE UNIVERSITY OF SYDNEY