Deep Learning Accelerators

Abhishek Srivastava (as29)
Samarth Kulshreshtha (samarth5)
University of Illinois, Urbana-Champaign

Submitted as a requirement for CS 433 graduate student project

Outline

- Introduction
  - What is Deep Learning?
  - Why do we need Deep Learning Accelerators?
  - A Primer on Neural Networks
- Tensor Processing Unit (Google)
  - TPU Architecture
  - Evaluation
  - Nvidia Tesla V100
  - Cloud TPU
- Eyeriss (MIT)
  - Convolutional Neural Networks (CNNs)
  - Dataflow Taxonomy
  - Eyeriss' dataflow
  - Evaluation
- How do Eyeriss and TPU compare?
- Many more DL accelerators…
- References

Introduction

What is Deep Learning?

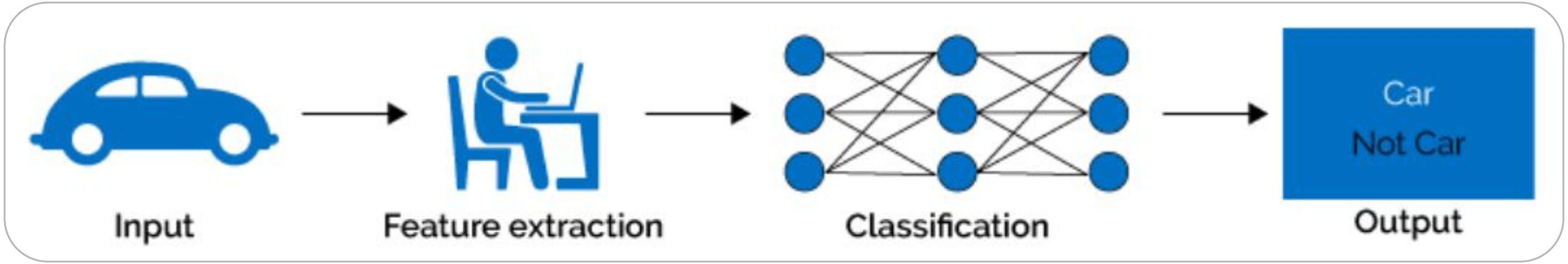


Image Source: Google Images

Why do we need DL accelerators?

- DL models essentially comprise of compute intensive operations like matrix multiplication, convolution, FFT etc.
- Input data for these models is usually of the order of GBs
- Large amount of computation over massive amounts of data
- CPUs support computations spanning all kinds of applications, hence they are bound to be slower when compared to an application specific hardware
- CPUs are sophisticated due to their need to optimize control flow (branch prediction, speculation etc.) while Deep Learning barely has any control flow
- Energy consumption can be minimized with specialization

350k tweets / minute

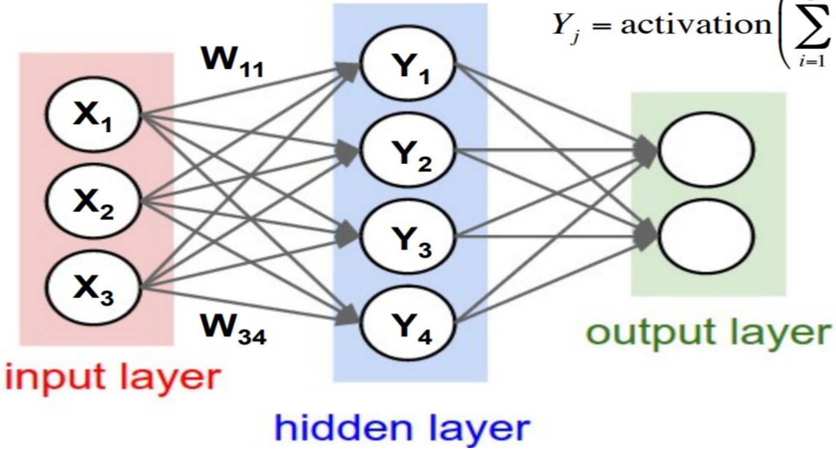
350M images / day

300 hours of video / minute

Sources: Twitter Facebook Youtube

A Primer on Neural Networks

$$Y_j = \text{activation}\left(\sum_{i=1}^{3} W_{ij} \times X_i\right)$$



$$\begin{bmatrix} X_1 & .. & X_n \end{bmatrix} \times \begin{bmatrix} W_{11} & .. & W_{1n} \\ .. & .. & .. \\ W_{n1} & .. & W_{nn} \end{bmatrix} = \begin{bmatrix} X_1 W_{11} + .. + X_n W_{n1} & .. & X_1 W_{1n} + .. + X_n W_{nn} \end{bmatrix}$$

Matrix Multiplication

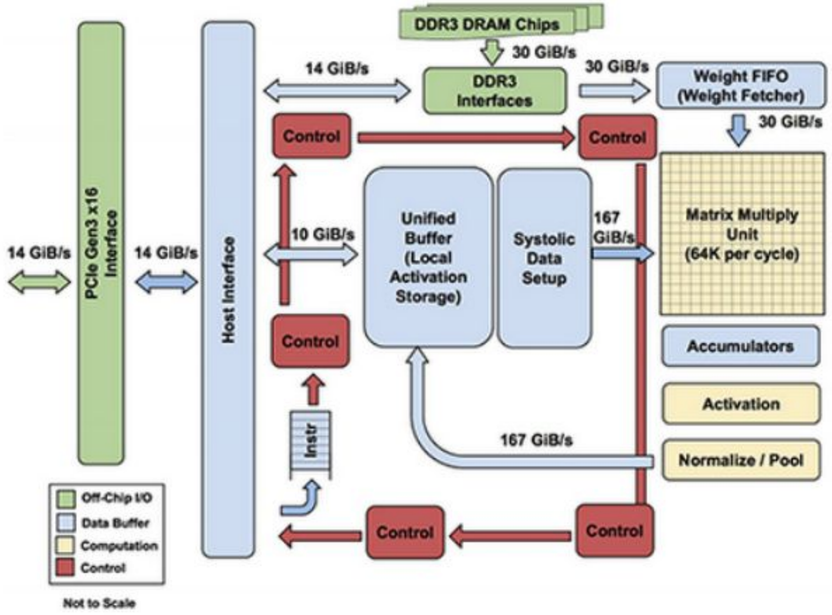
Tensor Processing Unit (Google)

Tensor Processing Unit [TPU]

- Developed by Google to accelerate neural network computations
- Production-ready co-processor connected to host via PCIe
- Powers many of Google's services like Translate, Search, Photos, Gmail etc.
- Why not GPUs?
  - GPUs don't meet the latency requirements for performing inference
  - GPUs tend to be underutilized for inference due to small batch sizes
  - GPUs are still relatively general-purpose
- Host sends instructions to TPU rather than the TPU fetching it itself
- "TPU closer in spirit to a Floating Point Unit rather than a GPU"

TPU Architecture

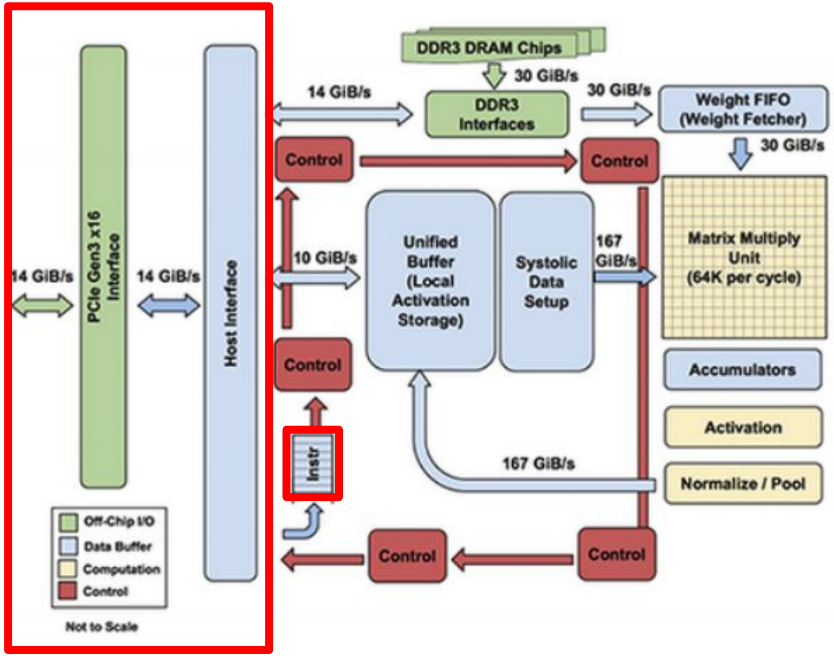
- Host sends instructions over PCIe bus into the instruction buffer
- Matrix Multiply Unit (MMU)
  - “heart” of TPU
  - 256x256 8-bit MACs
- Accumulators
  - aggregate partial sums
- Weight Memory (WM)
  - off-chip DRAM - 8 GB
- Weight FIFO (WFIFO)
  - on-chip fetcher to read from WM
- Unified Buffer (UB)
  - on-chip for intermediate values



TPU Block Diagram

TPU Architecture

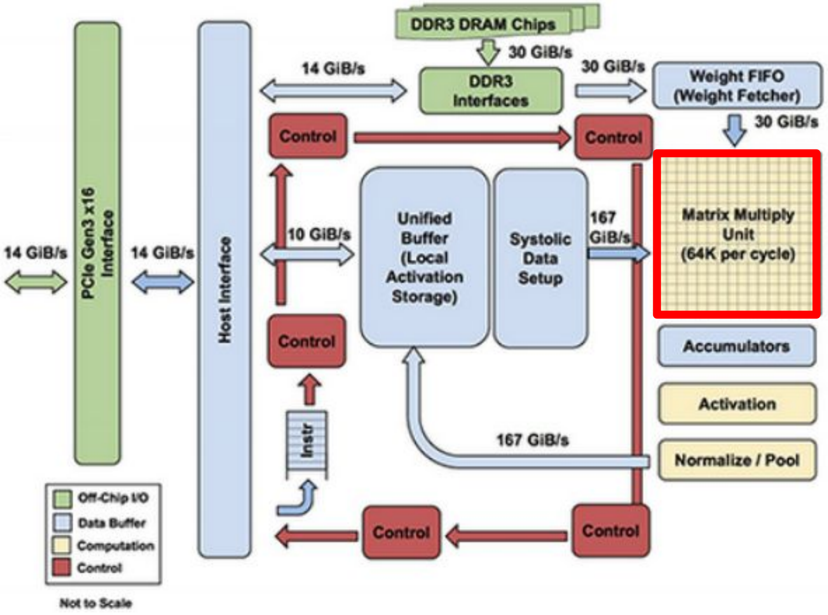
- **Host sends instructions over PCIe bus into the instruction buffer**
- Matrix Multiply Unit (MMU)
  - “heart” of TPU
  - 256x256 8-bit MACs
- Accumulators
  - aggregate partial sums
- Weight Memory (WM)
  - off-chip DRAM - 8 GB
- Weight FIFO (WFIFO)
  - on-chip fetcher to read from WM
- Unified Buffer (UB)
  - on-chip for intermediate values



TPU Block Diagram

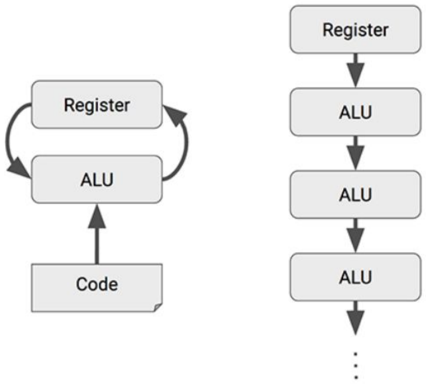
TPU Architecture

- Host sends instructions over PCIe bus into instruction buffer
- **Matrix Multiply Unit (MMU)**
  - **“heart” of TPU**
  - **256x256 8-bit MACs**
- Accumulators
  - aggregate partial sums
- Weight Memory (WM)
  - off-chip DRAM - 8 GB
- Weight FIFO (WFIFO)
  - on-chip fetcher to read from WM
- Unified Buffer (UB)
  - on-chip for intermediate values



TPU Block Diagram

MMU implemented as a systolic array

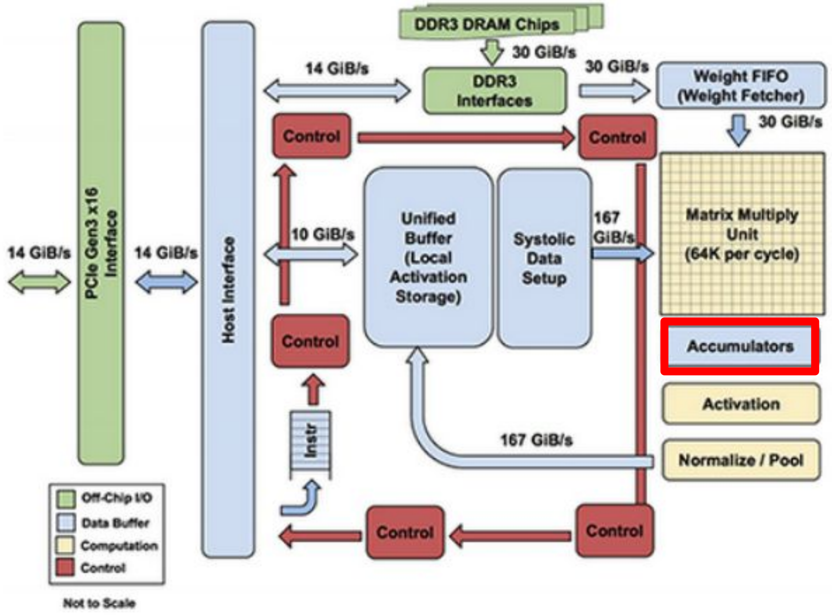


CPUs and GPUs often spend energy to access multiple registers per operation. A systolic array chains multiple ALUs together, reusing the result of reading a single register.

Multiplying an input vector by a weight matrix with a systolic array

TPU Architecture

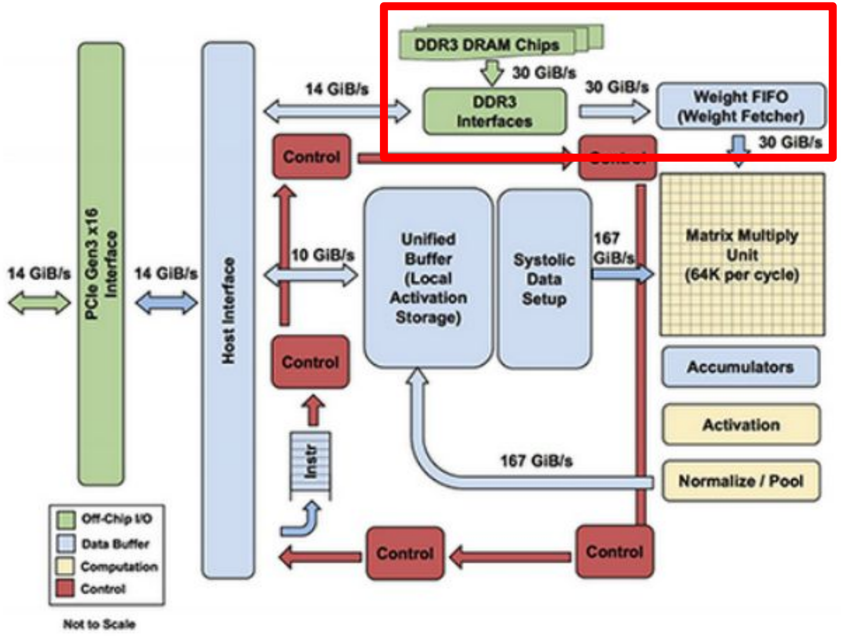
- Host sends instructions over PCIe bus into instruction buffer
- Matrix Multiply Unit (MMU)
  - “heart” of TPU
  - 256x256 8-bit MACs
- **Accumulators**
  - **aggregate partial sums**
- Weight Memory (WM)
  - off-chip DRAM - 8 GB
- Weight FIFO (WFIFO)
  - on-chip fetcher to read from WM
- Unified Buffer (UB)
  - on-chip for intermediate values



TPU Block Diagram

TPU Architecture

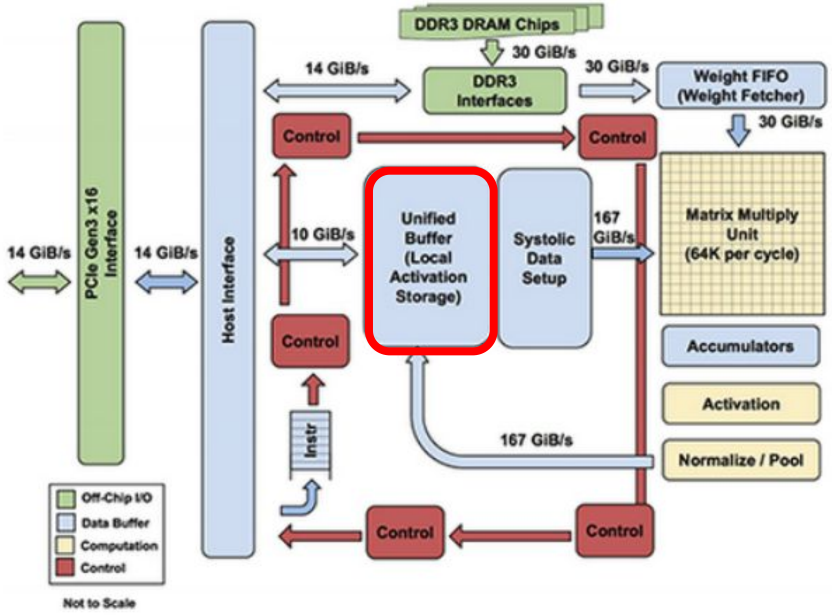
- Host sends instructions over PCIe bus into instruction buffer
- Matrix Multiply Unit (MMU)
  - “heart” of TPU
  - 256x256 8-bit MACs
- Accumulators
  - Aggregate partial sums
- **Weight Memory (WM)**
  - **Off-chip DRAM - 8 GB**
- **Weight FIFO (WFIFO)**
  - **On-chip fetcher to read from WM**
- Unified Buffer (UB)
  - On-chip for intermediate values



TPU Block Diagram

TPU Architecture

- Host sends instructions over PCIe bus into instruction buffer
- Matrix Multiply Unit (MMU)
  - “heart” of TPU
  - 256x256 8-bit MACs
- Accumulators
  - Aggregate partial sums
- Weight Memory (WM)
  - Off-chip DRAM - 8 GB
- Weight FIFO
  - On-chip fetcher to read from WM
- **Unified Buffer (UB)**
  - **On-chip for intermediate values**



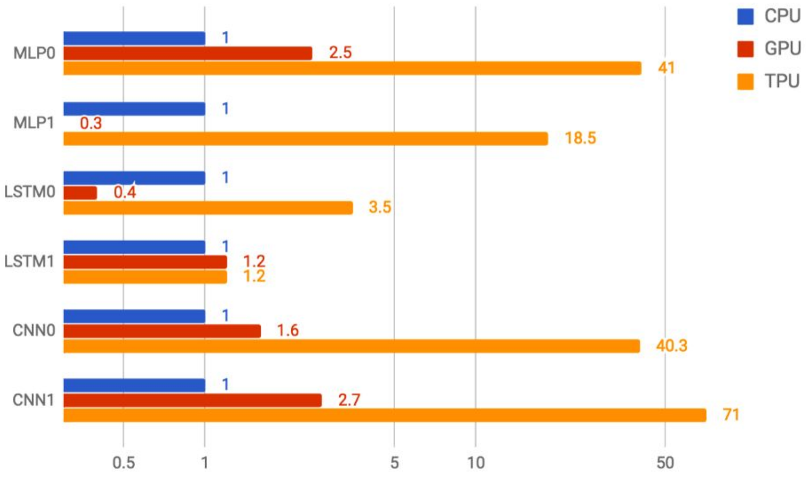
TPU Block Diagram

TPU ISA

- CISC instructions (average CPI = 10 to 20 cycles)
- 12 instructions
  - **Read_Host_Memory**: reads data from host memory into Unified Buffer
  - **Read_Weights:** reads weights from Weights Memory into Weight FIFO
  - **MatrixMultiply/Convolve:** perform matmul/convolution on data from UB and WM and store into Accumulators
    - B X 256 input and 256 X 256 weight => B X 256 output in B cycles (pipelined)
  - **Activate:** apply activation function on inputs from Accumulator and store into Unified Buffer
  - **Write_Host_Memory**: writes data from Unified Buffer into host memory
- Software stack - application code to be run on TPU written in Tensorflow and compiled into an API which can be run on TPU (or even GPU)

Evaluation

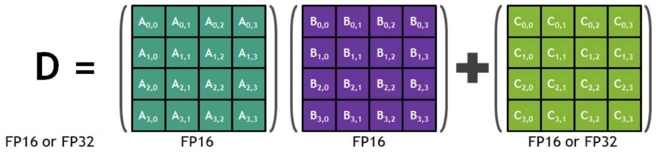
- Performance comparison based on predictions per second on common DL workloads
    - overpowers GPUs massively for CNNs
    - performs reasonably well than GPUs for MLPs
    - performs close to GPUs for LSTMs
- Good
    - programmability
    - production ready
- Bad
    - converts convolution into matmul which may not be most optimal
    - no direct support for sparsity



CPU, GPU and TPU performance on six reference workloads (in log scale)

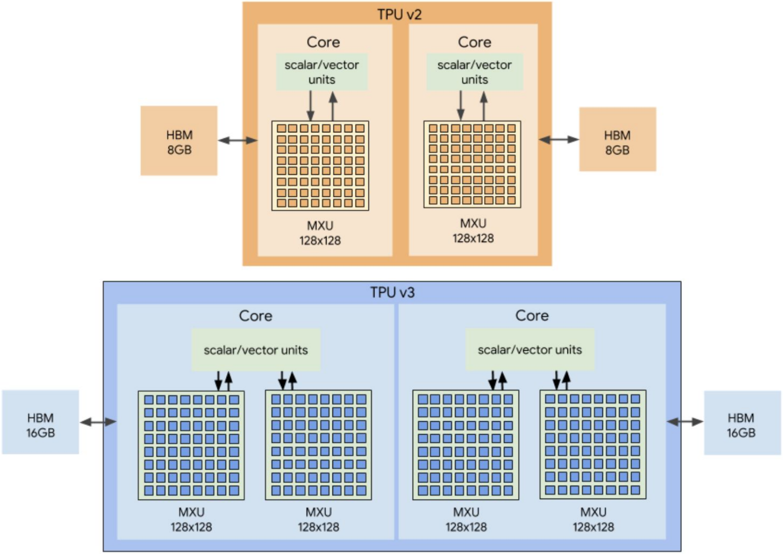
Nvidia Tesla V100

- Tensor cores
    - programmable matrix-multiply-and-accumulate units
    - 8 cores/SM => total = 640 cores
    - input - 4x4 matrices
        - A,B must be FP16
        - C,D can be FP16/FP32
- Exposed as Warp-level matmul operation in CUDA 9
- Specialized matrix load/multiply/accumulate/store operations
- Part of multi GPU system optimized using NvLink interconnect and High Bandwidth Memory



Cloud TPU

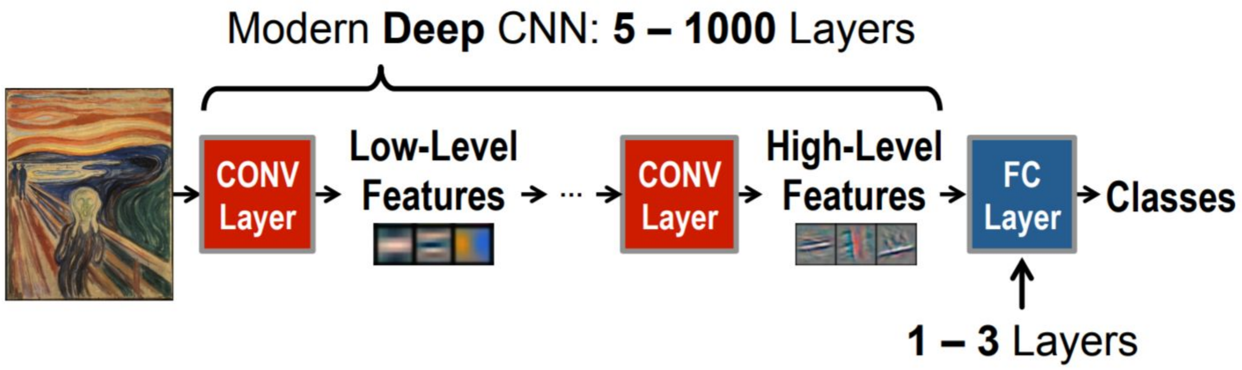
- Part of Google Cloud
- Each node comprises of 4 chips
- 2 “tensor cores“ per chip
  - each core has scalar, vector and matrix units (MXU)
  - 8/16 GB on-chip HBM per core
- 8 cores per cloud TPU node coupled with high bandwidth interconnect
- TPU Estimator APIs used to generate tensorflow computation graph, which is sent over gRPC and Just In Time compiled onto the cloud TPU node



TPU chip (v2 and v3) as part of cloud TPU node

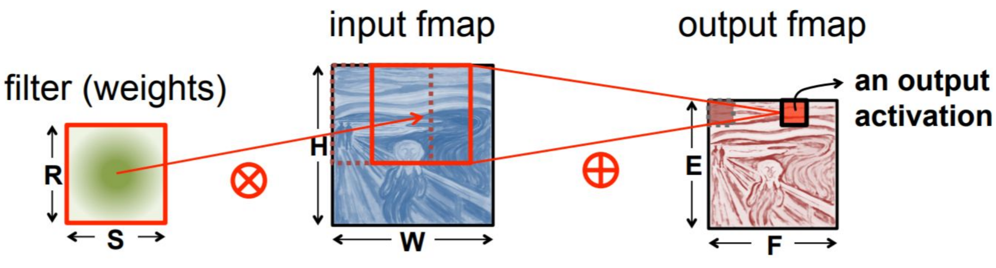
Eyeriss (MIT)

Convolutional Neural Networks

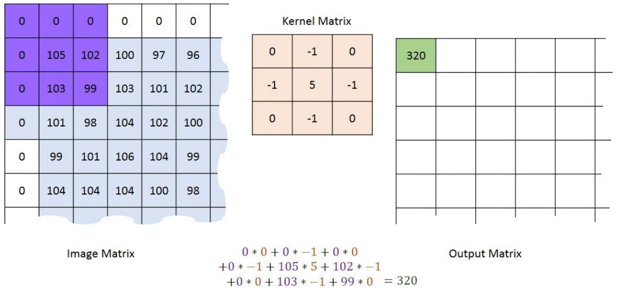


- Each convolution layer identifies certain fine grained features from the input image, aggregating over features from previous layers
- Very often there are certain optional layers in between CONV layers such as NORM/POOL layers to reduce the range/size of input values
- Convolutions account for **more than 90%** of overall **computation**, dominating **runtime** and **energy** consumption

2D Convolution operation



Sliding Window Processing



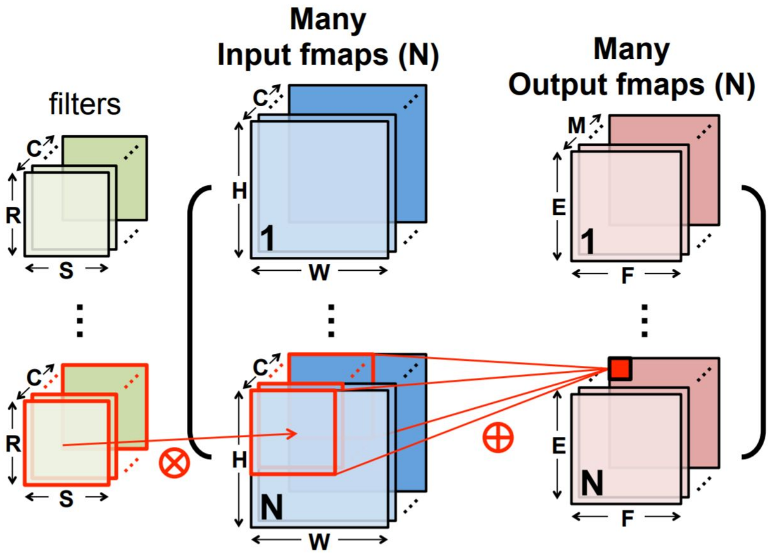
Convolution with horizontal and vertical strides = 1

- 2D convolution is a set of multiply and accumulate operations of the kernel matrix (also known as filter) and the input image feature map by sliding the filter over the image

Image Source: Understanding Convolutional Layers in Convolutional Neural Networks (CNNs)

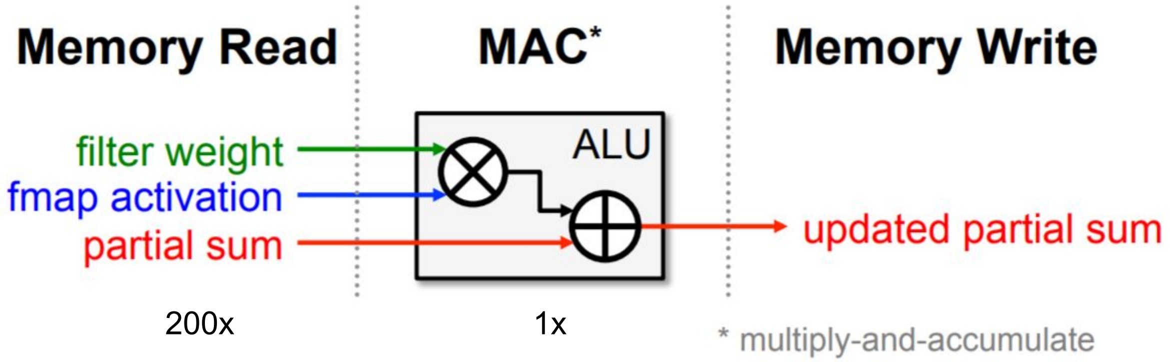
Multi-channel input with multi-channel filters

- Each filter and fmap have C channels -> the application of a filter on an input fmap across C channels results in one cell of the output fmap
- Rest of the cells of the output fmap are obtained by sliding the filter over the input fmap producing one channel of the output fmap
- Application of M such filters results in a single M channeled output fmap with as many channels as the number of filters
- Previous steps are batched over multiple input fmaps resulting in multiple output fmaps



Things to note

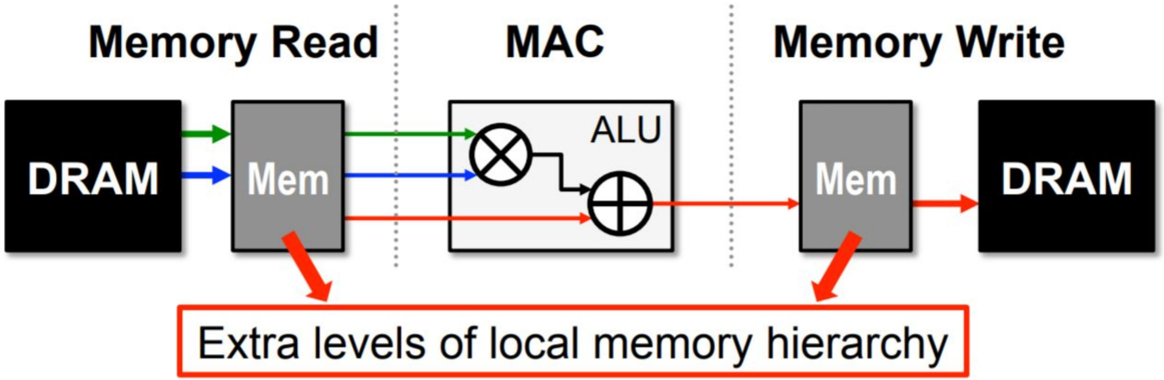
- Operations exhibit **high parallelism**
  - High throughput possible
- Memory access is the **bottleneck**
- Lot of scope for **data reuse**



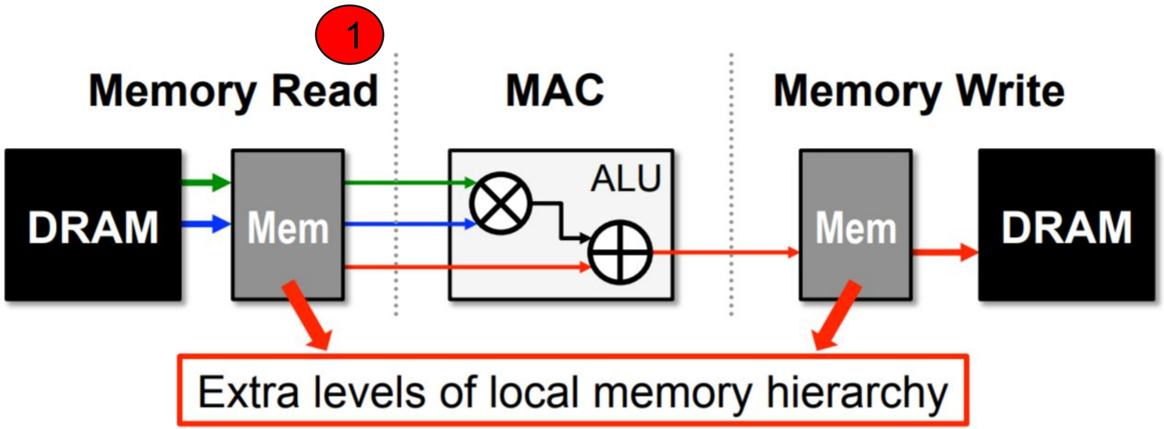
WORST CASE: all memory R/W are DRAM accesses

Example: AlexNet [NIPS 2012] -> 724M MACs = 2896M DRAM accesses required

Memory access is the bottleneck



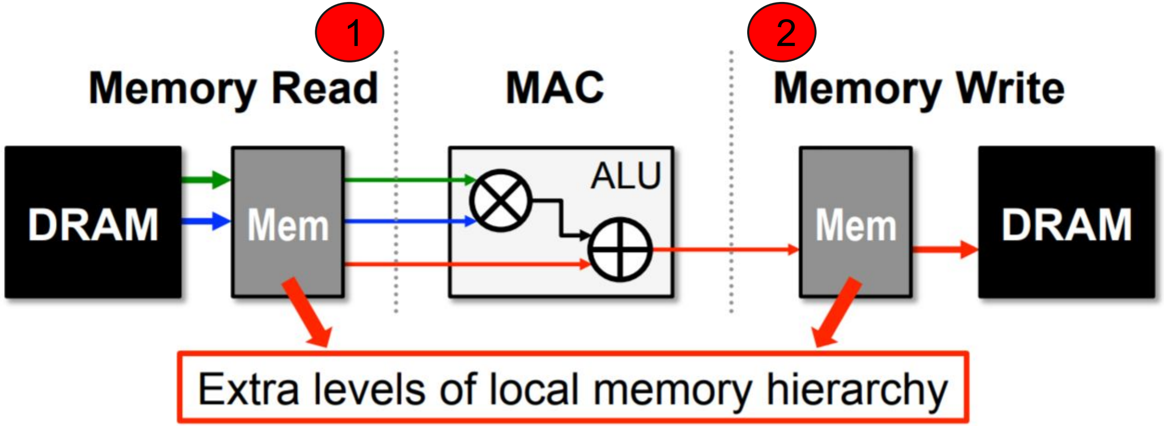
Memory access is the bottleneck



Opportunities:

1. Reuse filters/fmap reducing DRAM reads

Memory access is the bottleneck



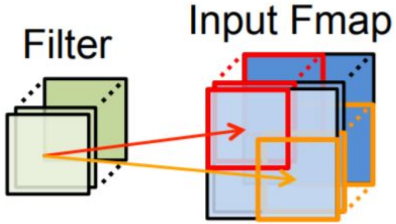
Opportunities:

1. Reuse filters/fmap reducing DRAM reads
2. Partial sum accumulation does not have to access DRAM

Types of data reuse in DNN

Convolutional Reuse

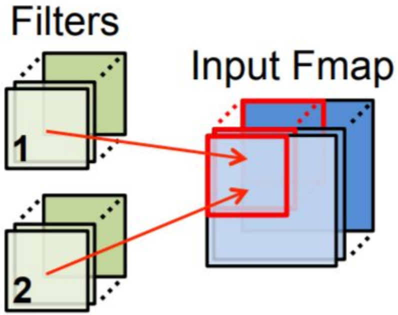
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

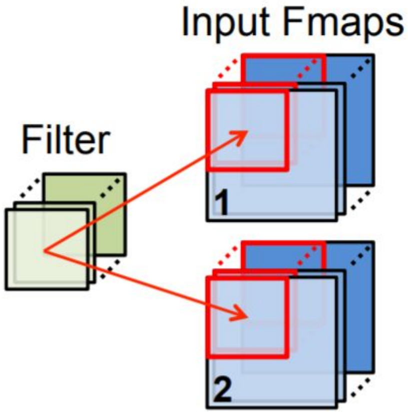
CONV and FC layers



Reuse: Activations

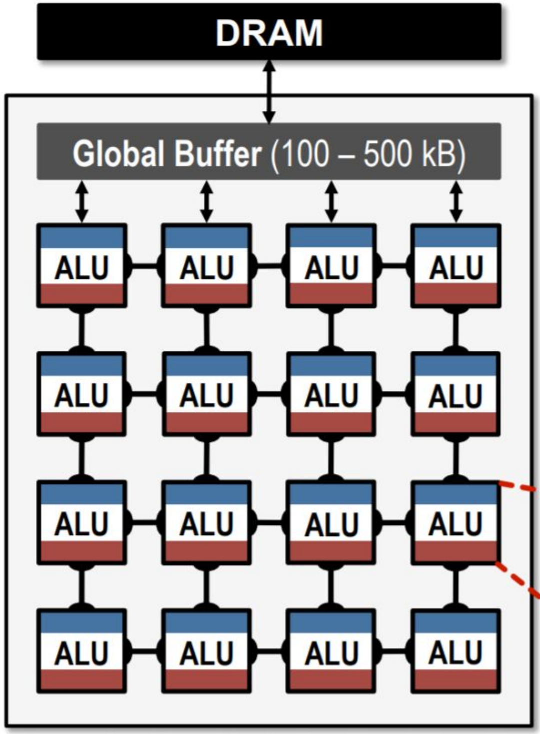
Filter Reuse

CONV and FC layers
(batch size > 1)



Reuse: Filter weights

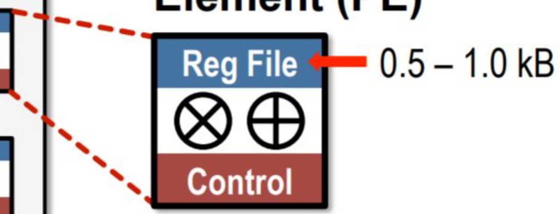
Spatial Architecture for DNN



Local Memory Hierarchy

- Global Buffer
- Direct inter-PE network
- PE-local memory (RF)

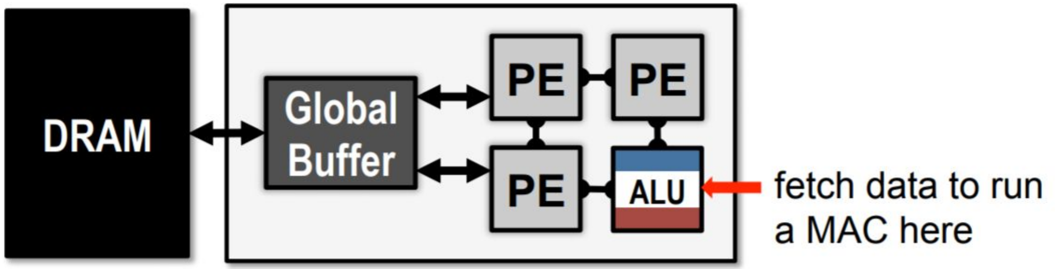
Processing Element (PE)

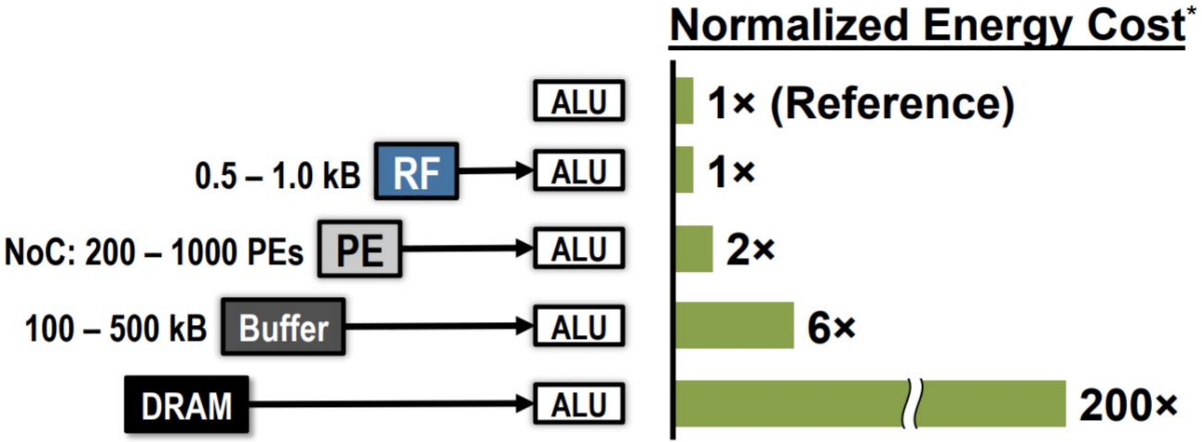


Efficient Data Reuse
Distributed local storage (RF)

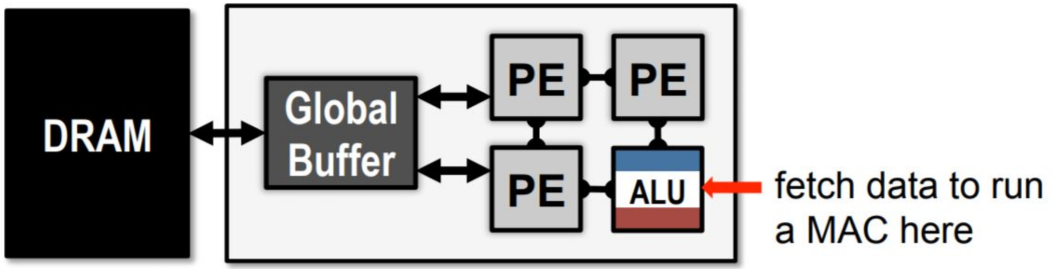
Inter PE communication
Sharing among regions of PEs

Data movement is expensive

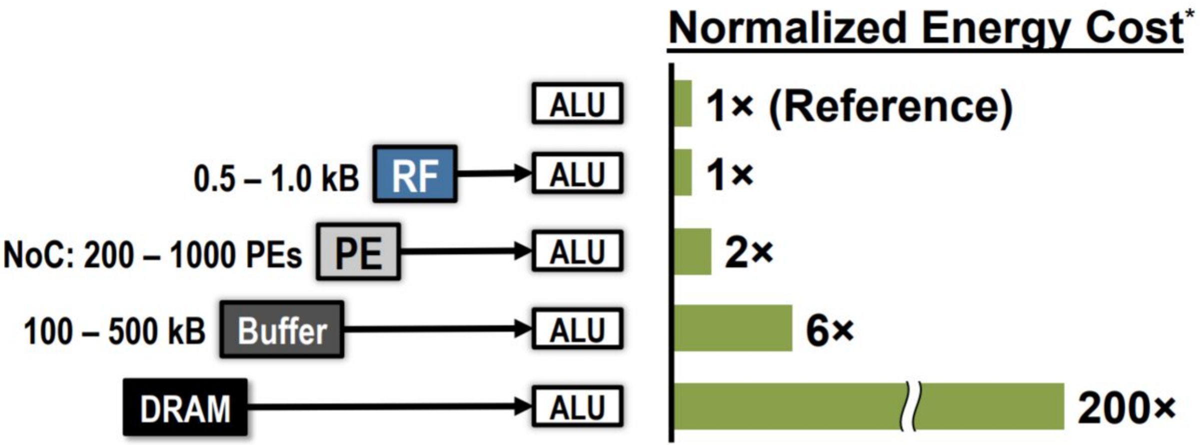




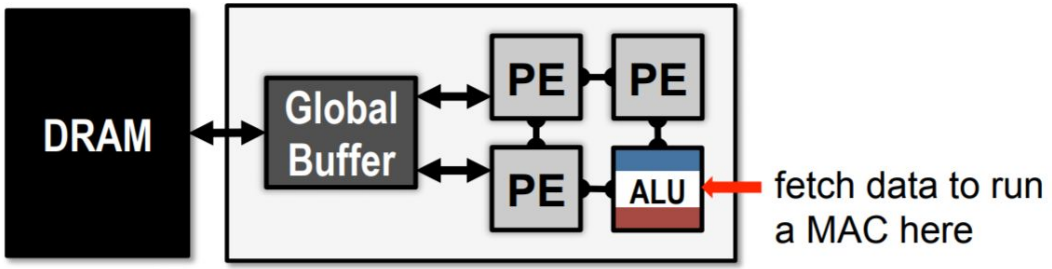
Data movement is expensive

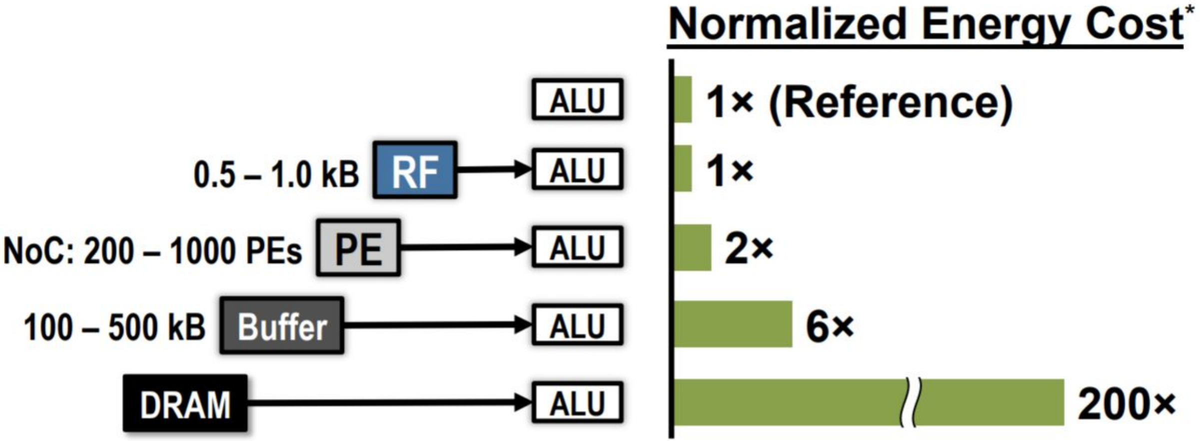


How to exploit data reuse and local accumulation with limited low-cost local storage?



Data movement is expensive





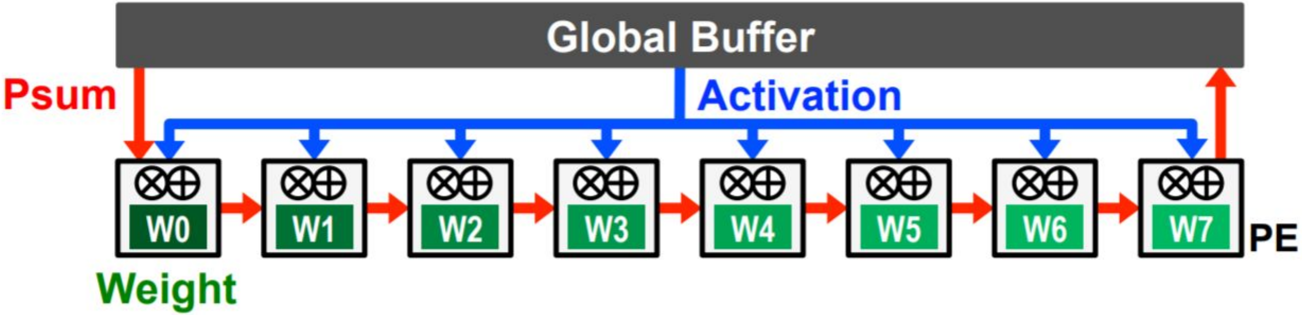
How to exploit data reuse and local accumulation with limited low-cost local storage?

Require specialized processing dataflow!

Dataflow Taxonomy

- Weight Stationary (WS) - reduce movement of filter weights
- Output Stationary (OS) - reduce movement of partial sums
- No Local Reuse (NLR) - no local storage at the PE, use a global buffer of larger size

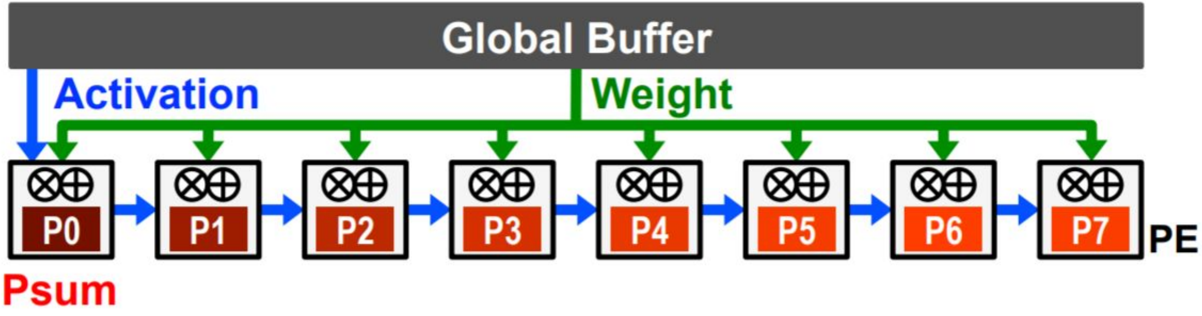
Weight Stationary



- **Minimize** weight read energy consumption
  - maximize convolutional and filter reuse of weights
- **Broadcast** activations and **accumulate** psums **spatially** across the PE array.

Examples: Chakradhar [ISCA 2010], Origami [GLSVLSI 2015]

Output Stationary

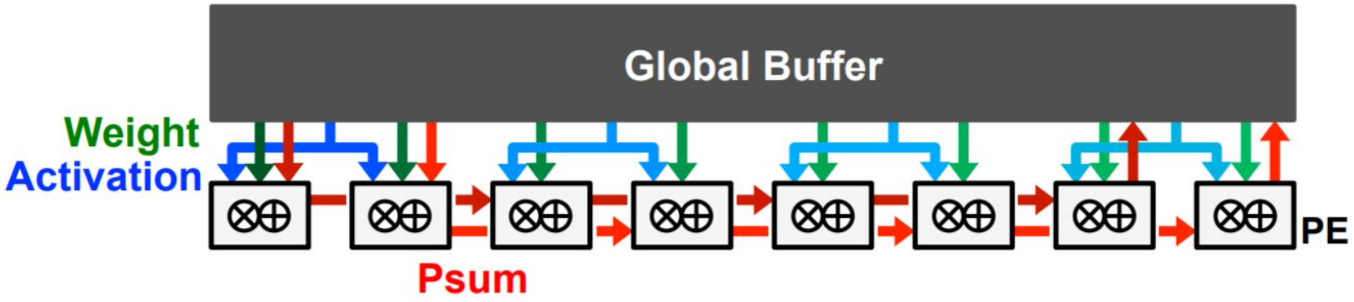


- **Minimize partial sum** R/W energy consumption
  - maximize local accumulation

- **Broadcast/Multicast filter weights** and **reuse activations spatially** across the PE array

Examples: Gupta [ICML 2015], ShiDianNao [ISCA 2015]

No Local Reuse



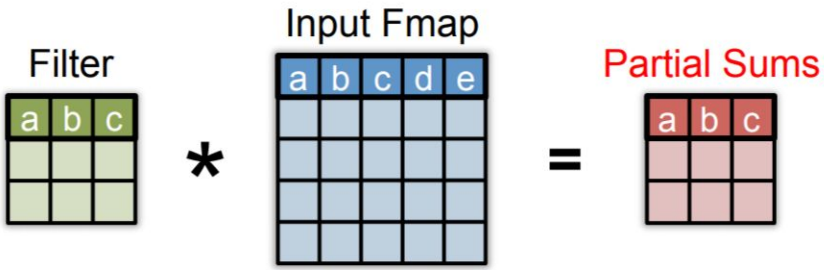
- Use a **large global buffer** as shared storage
  - Reduce **DRAM** access energy consumption
- **Multicast activations, single-cast weights, and accumulate psums spatially** across the PE array

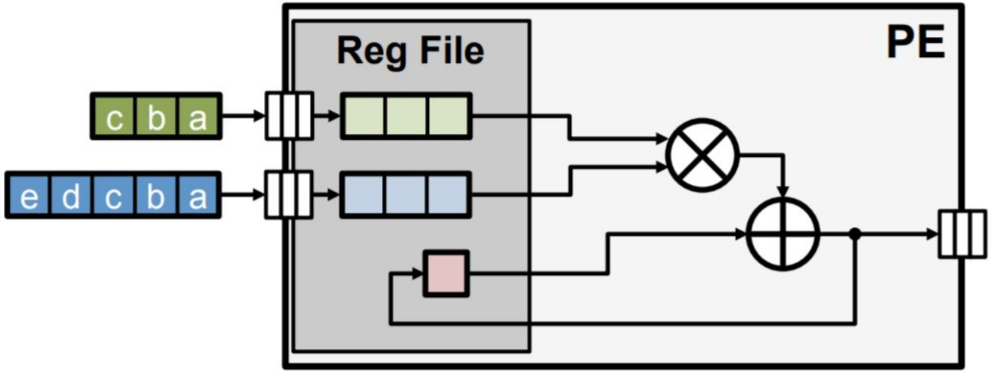
Examples: DaDianNao [MICRO 2014], Zhang [FPGA 2015]

Eyeriss' data flow: Row Stationary

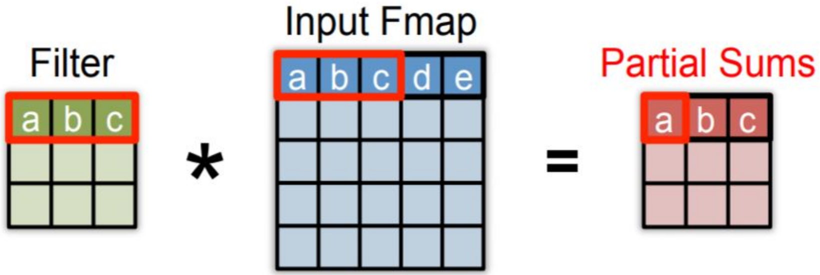
- Previous approaches only optimize for certain types of data reuse -> this may lead to performance degradation when input dimensions vary
- Eyeriss maximizes reuse and accumulation at RF
- Eyeriss optimizes for overall energy efficiency instead of only a specific input type (input fmap, filters, psums)
- Eyeriss tries to break high dimensional convolution into 1D convolutional primitives which operate on one row of filter weights, one row of input feature map generating one row of partial sums => "Row Stationary"

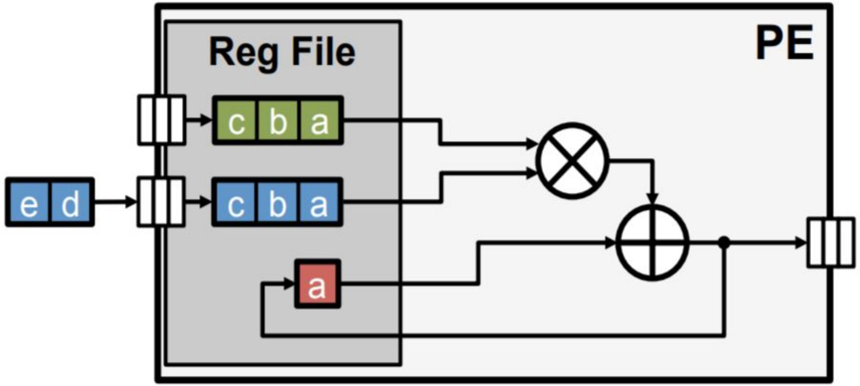
1D Row Convolution in PE



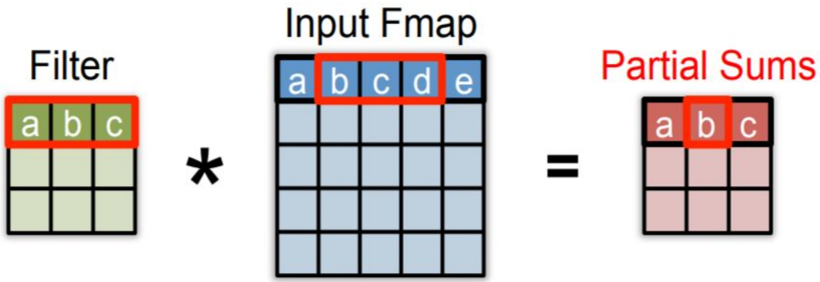


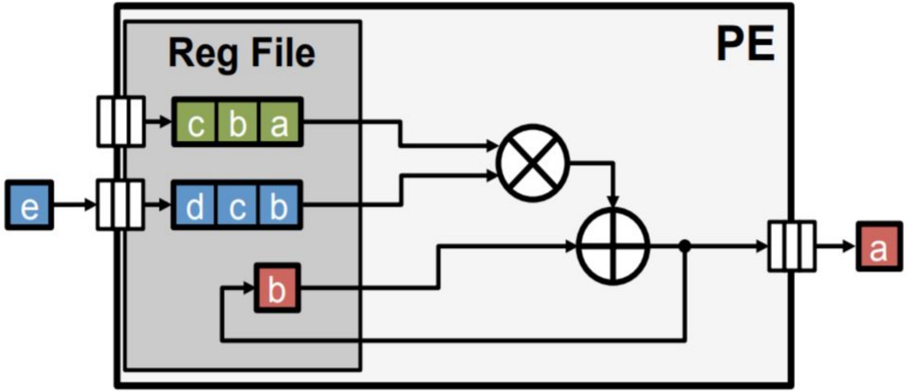
1D Row Convolution in PE



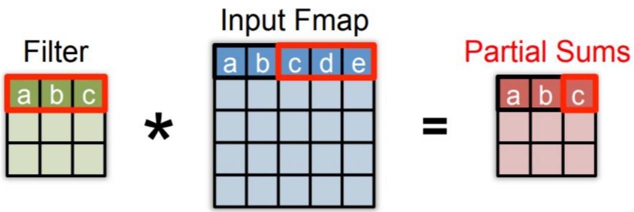


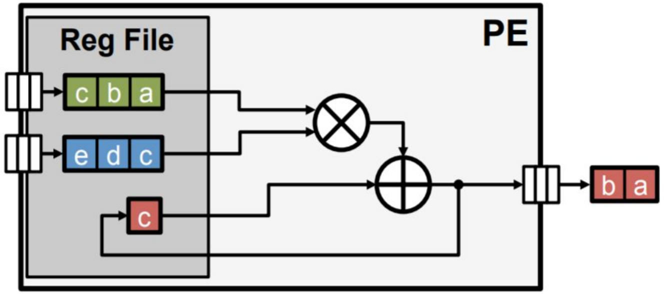
1D Row Convolution in PE





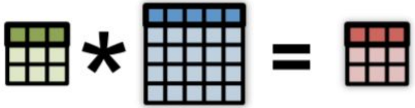
1D Row Convolution in PE



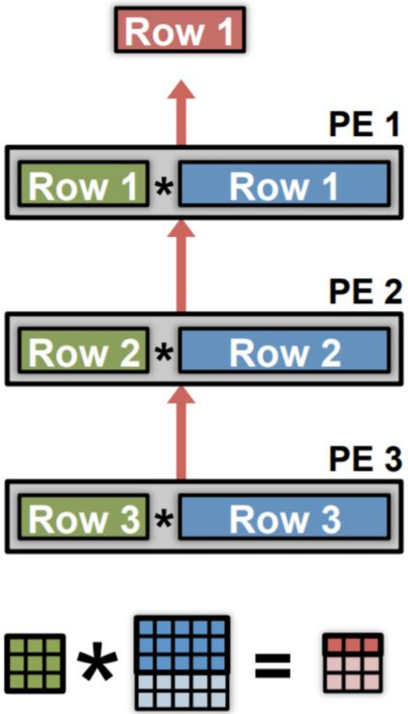


- Maximize row **convolutional reuse** in RF
  - Keep a **filter** row and **fmap** sliding window in RF
- Maximize row **psum** accumulation in RF

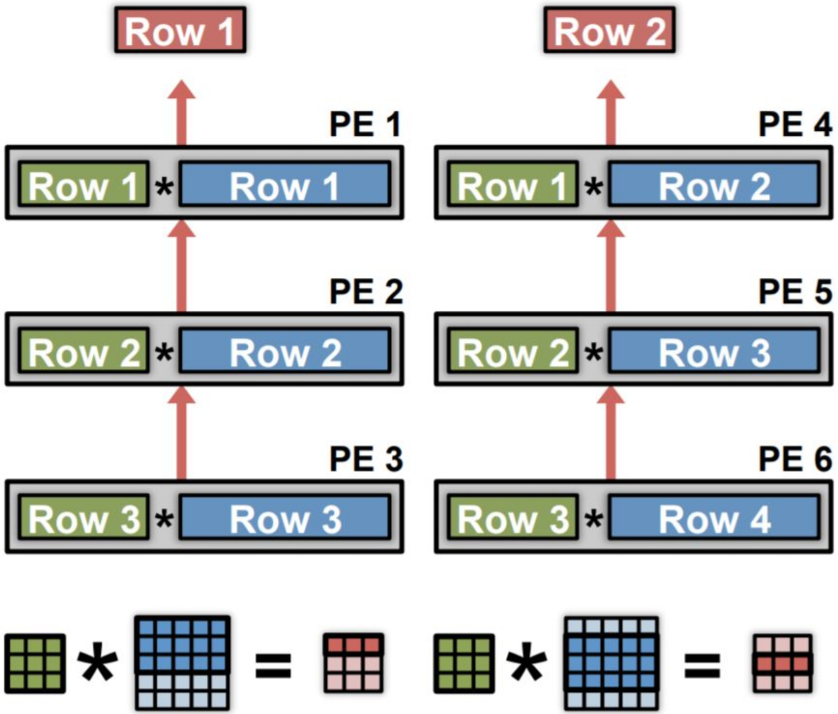
2D convolution in a PE array



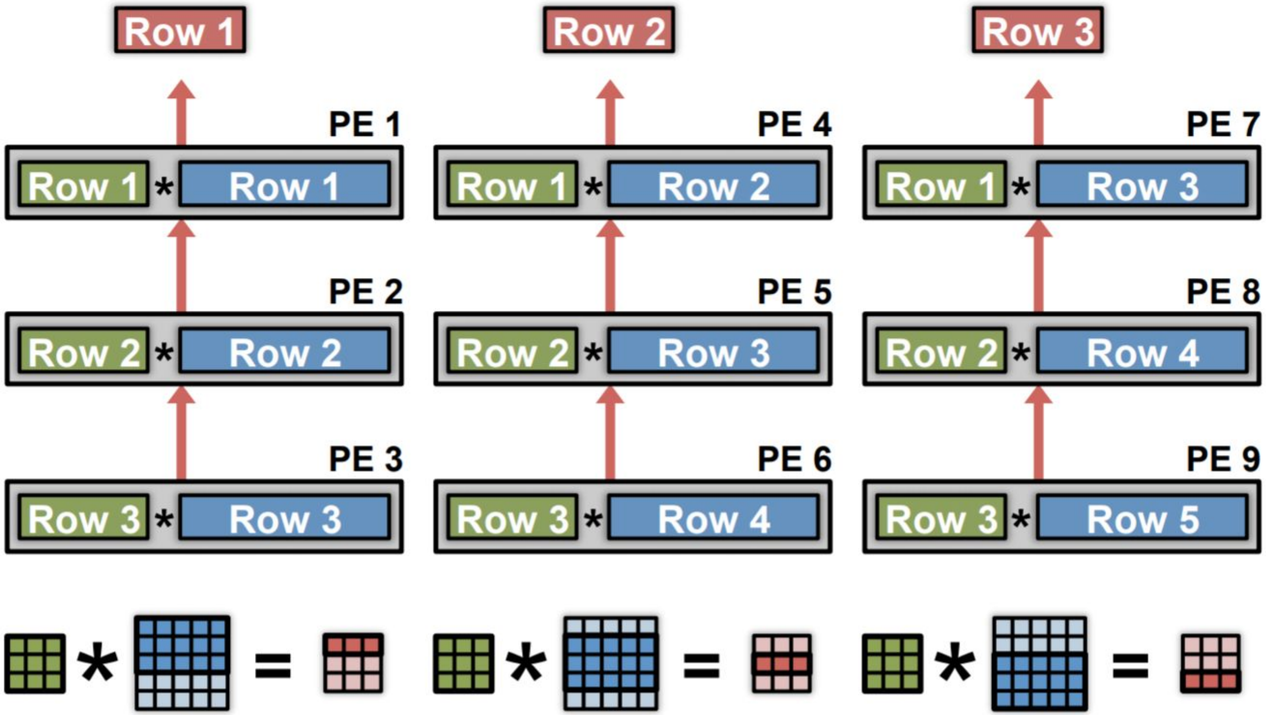
2D convolution in a PE array



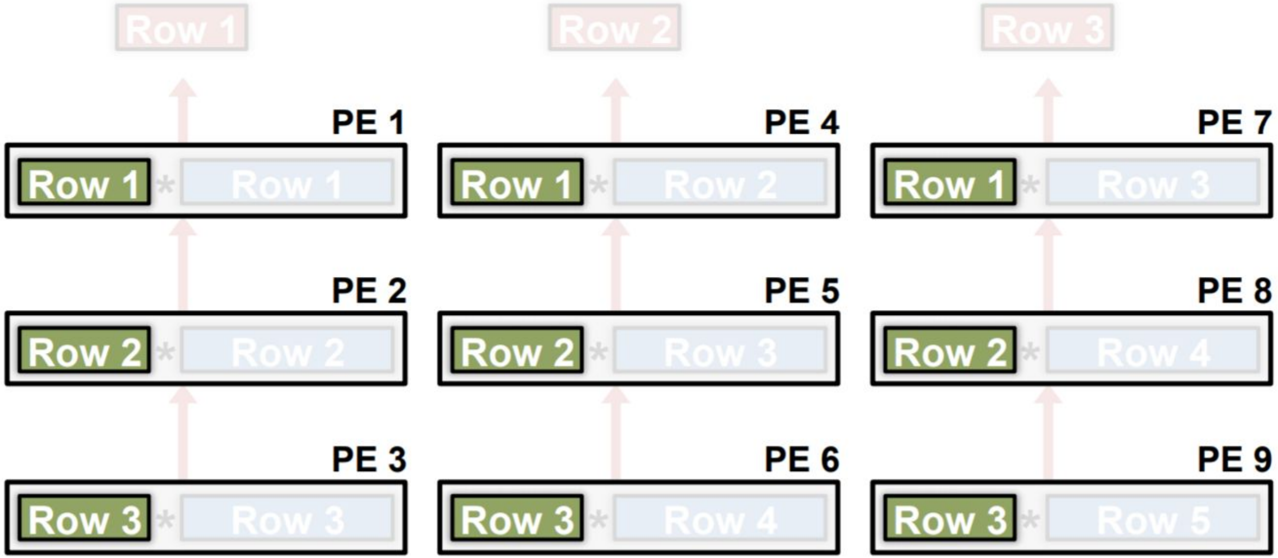
2D convolution in a PE array



2D convolution in a PE array

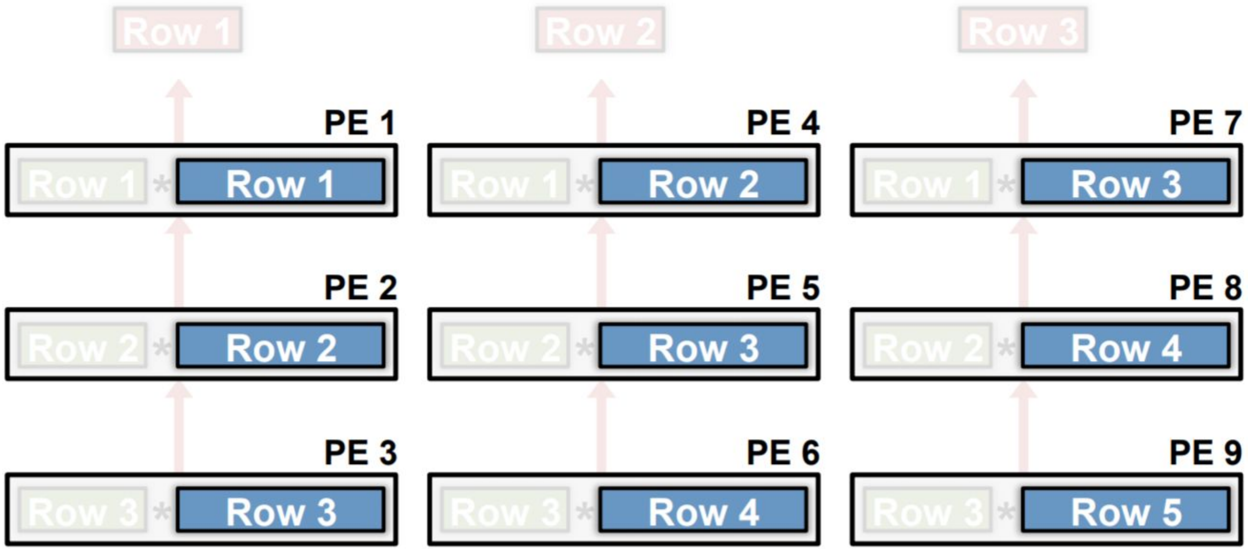


Convolutional Reuse Maximized



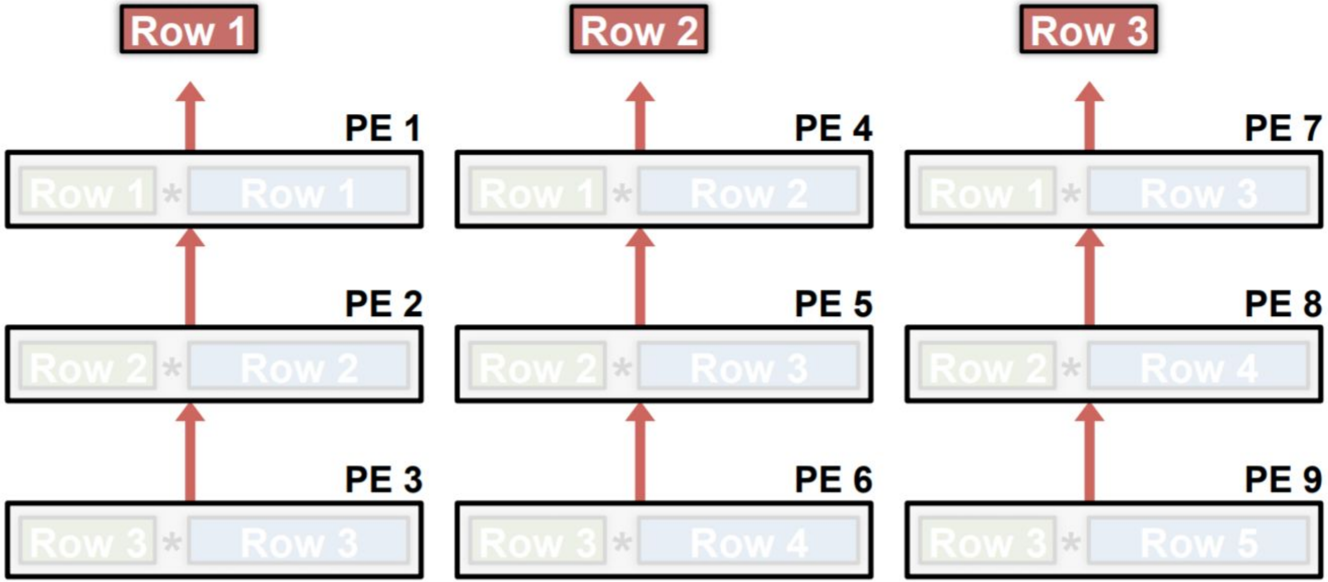
Filter rows are reused across PEs horizontally

Convolutional Reuse Maximized



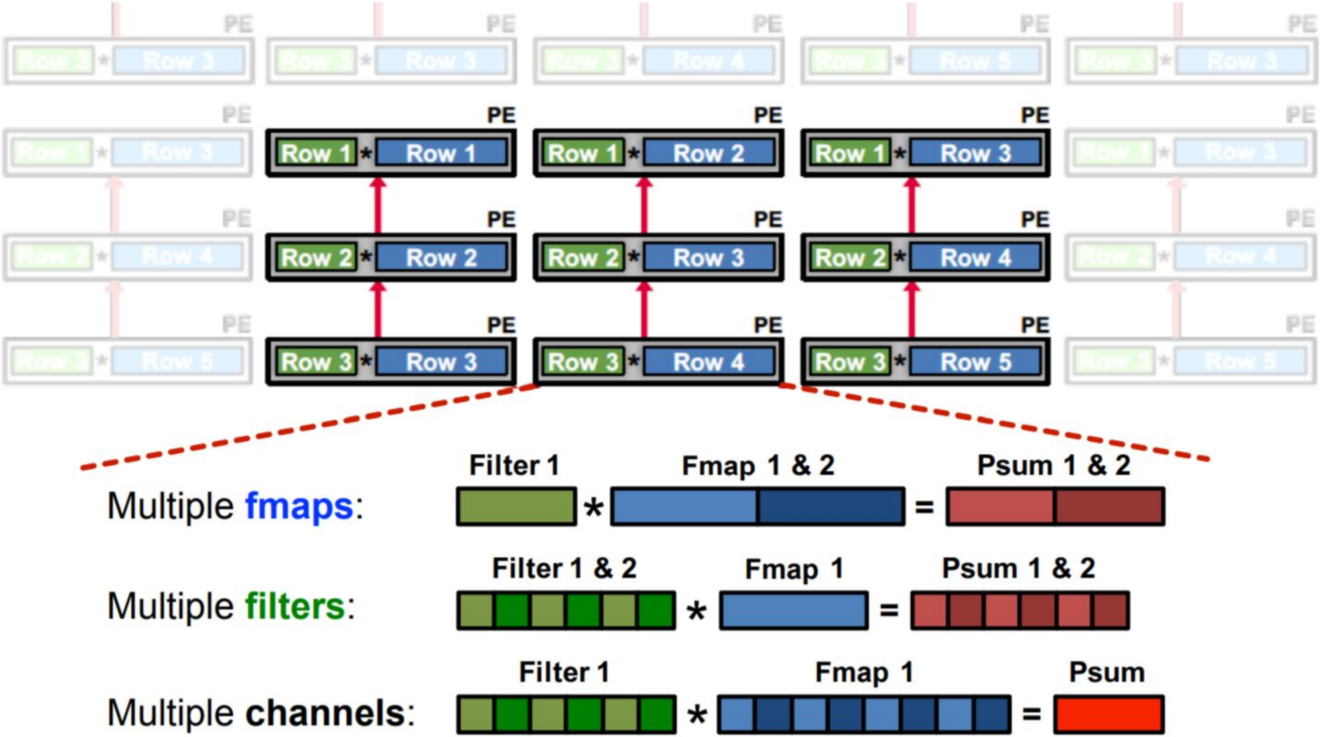
Fmap rows are reused across PEs diagonally

Convolutional Reuse Maximized

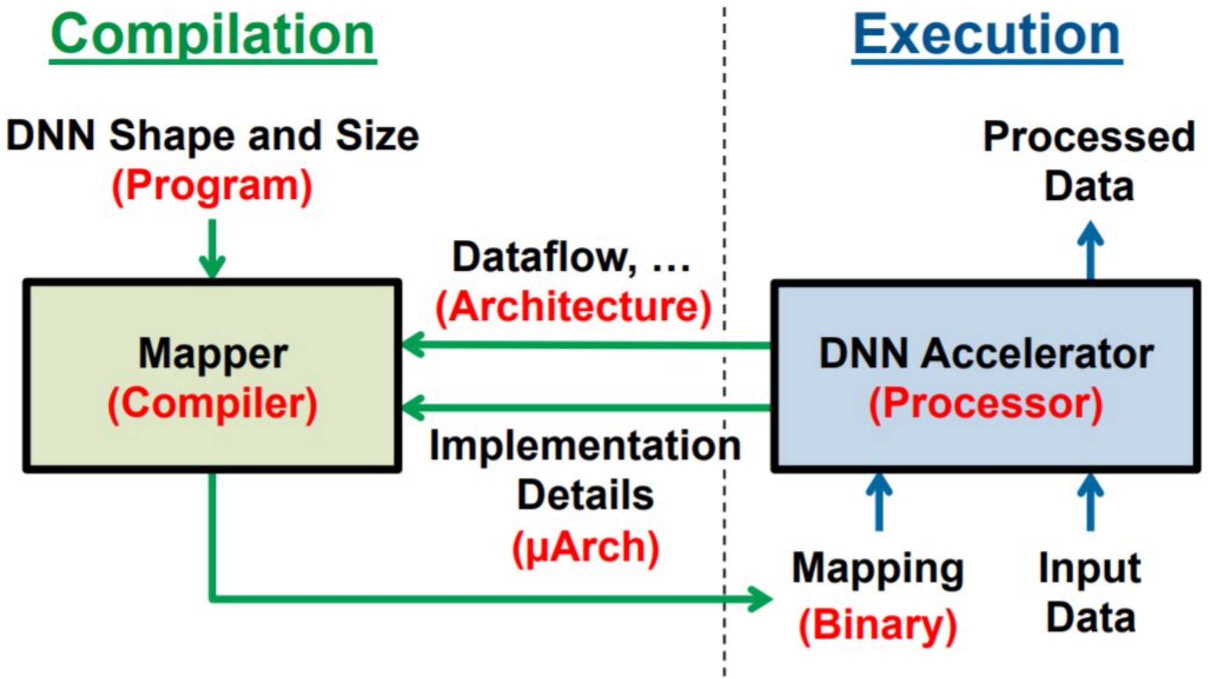


Partial sums accumulated across PEs vertically

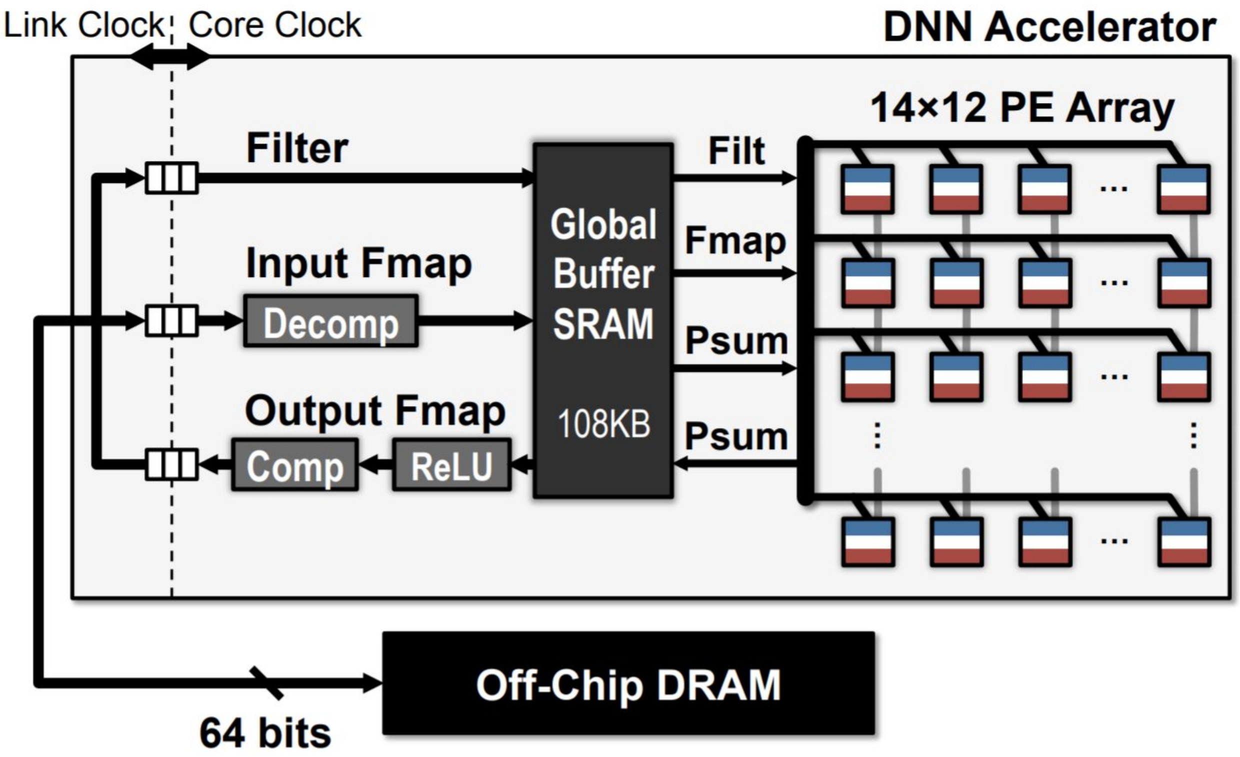
DNN Processing - The Full Picture



Mapping DNN to the PEs

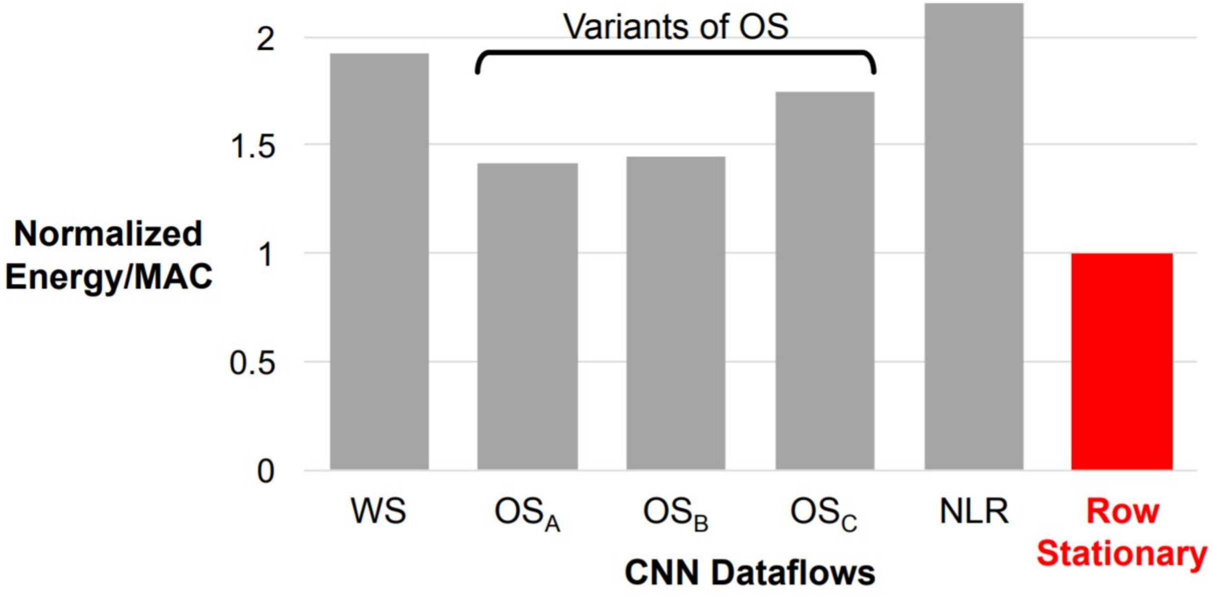


Eyeriss Deep CNN Accelerator



Evaluation

- Same total area
- AlexNet CONV layers
- 256 PEs
- Batch size = 16



How do Eyeriss and TPU compare ?

- Programmability?
  - TPU is far more programmable than Eyeriss
- Usability?
  - TPU is relatively more general purpose while Eyeriss is highly optimized for CNNs
- Memory hierarchy?
  - Eyeriss' memory hierarchy also includes Inter PE communication while TPU's does not explicitly
- Applications?
  - TPUs are being pushed towards training workloads while Eyeriss is optimized for inference
- Energy?
- Chip size and cost?

Many more DL accelerators...

- State-of-the-art neural networks (AlexNet, ResNet, LeNet etc)
  - large in size
  - high power consumption due to memory access
  - difficult to deploy on embedded devices
- End-to-end deployment solution (Song et.al.)
  - use "deep compression" to make network fit into SRAM
  - deploy it on EIE (Energy efficient Inference Engine) which accelerates resulting sparse vector matrix multiplication on the compressed network
- Accelerators for other DL models
  - Generative Adversarial Networks - GANAX (Amir et.al.)
  - RNNs, LSTMs - FPGA based accelerators, ESE (Song et.al.)
- Mobile phone SoCs
  - Google Pixel 2 - Visual Core, IPhone X - Neural Engine, Samsung Exynos - NPU

References

- An in-depth look at Google's first Tensor Processing Unit (TPU)
- In-Datacenter Performance Analysis of a Tensor Processing Unit
- Images and some content pertaining to the Eyeriss architecture has been lifted as is from the Eyeriss tutorial.
- Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks
- http://eyeriss.mit.edu
- EIE: Efficient Inference Engine on Compressed Deep Neural Network
- GANAX: A Unified MIMD-SIMD Acceleration for Generative Adversarial Networks
- ESE: Efficient Speech Recognition Engine with Sparse LSTMs on FPGA