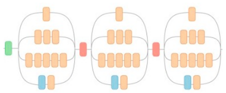
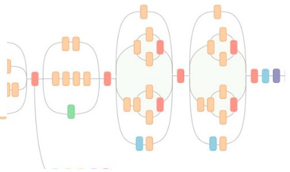
Discussing the Lottery Ticket Papers

1. Why is it hard for SGD to learn sparse representations from scratch?
2. How to make the lottery tickets fit better for modern libraries (possibly at the cost of being less sparse)?
3. Can we use graphical models to learn the sparse connection topology?
4. Can we reformulate the sparse training problem as learning low-rank representations for high-dimensional data? E.g. using nuclear norm (OLE loss)/ rate distortion (MCR loss) to induce more interpretable sparse models instead of heuristically searching for sparse connections using pruning?

Recap: Deep Learning Systems



ML Model



Automatic Differentiation

Graph-Level Optimization

Parallelization / Distributed Training

Code Optimization

Memory Optimization

ML Hardware

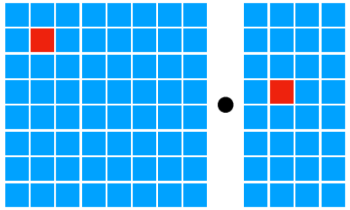
Current ML Hardware

- CPUs
  - Intel, ARM, AMD
- GPUs
  - NVIDIA, AMD
- TPUs
  - Google
- ML-specific accelerators ($20B market*)
  - Graphcore, SambaNova, Eyeriss, Cerebras, etc.

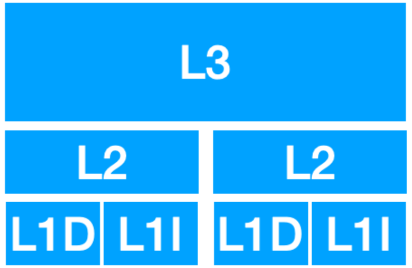
* source: https://www.idc.com/getdoc.jsp?containerId=prUS48127321

Compute Primitives | Memory Hierarchy

CPUs

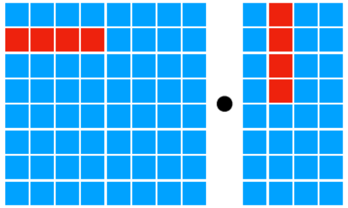


scalar



implicitly managed

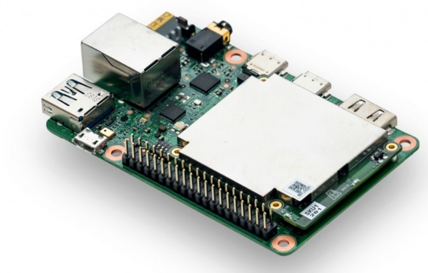
GPUs

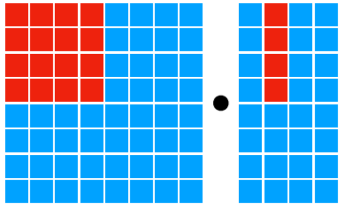


vector

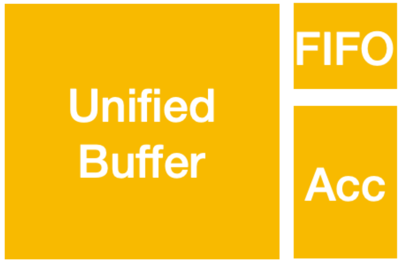
mixed

TPUs





tensor



explicitly managed

Winograd: Fast Algorithms for Convolutional Neural Networks

Andrew Lavin, Scott Gray

Overview

- **Key idea**: the Winograd filtering algorithm with minimal number of floating-point multiplications

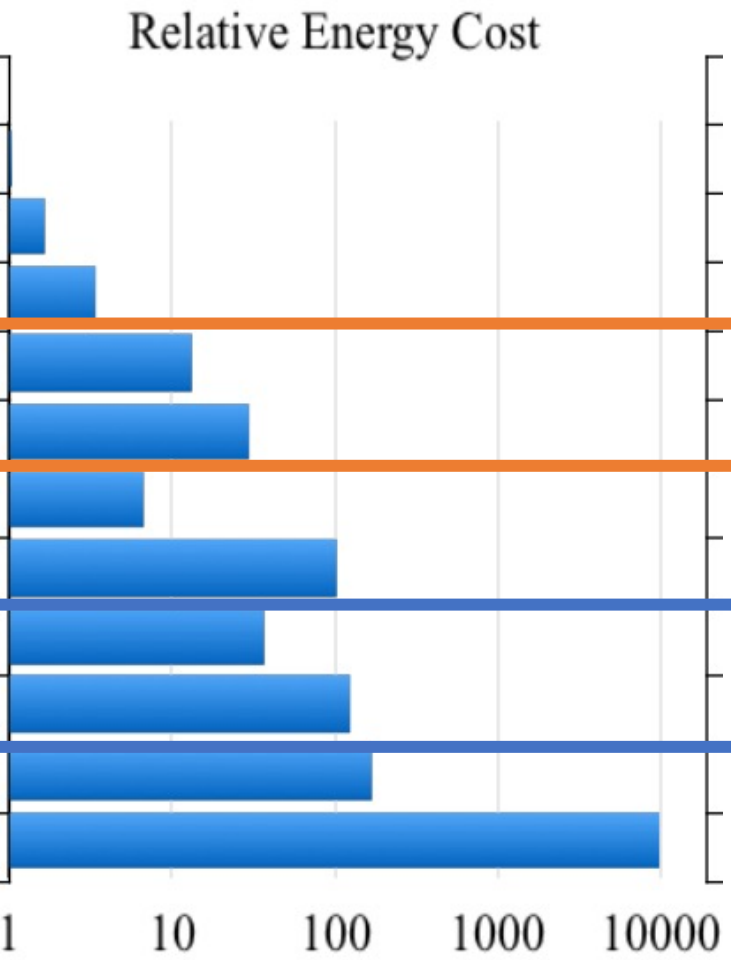
Compute a convolution with an input size M x N and a filer size R x S

- Vanilla Convolution: $O_{x,y} = \sum_{c=1}^{C} \sum_{v=1}^{R} \sum_{u=1}^{S} D_{c,x+u,y+v} * W_{c,u,v}$
  - # floating-point multiplications = C x R x S x (M-R+1) x (N-S+1)

- Winograd filtering algorithm: $O = D \circledast W = A^T [GDG^T] \circ [B^T W B] A$
  - # floating-point multiplications = C x M x N

Why are we interested in reducing # multiplications?

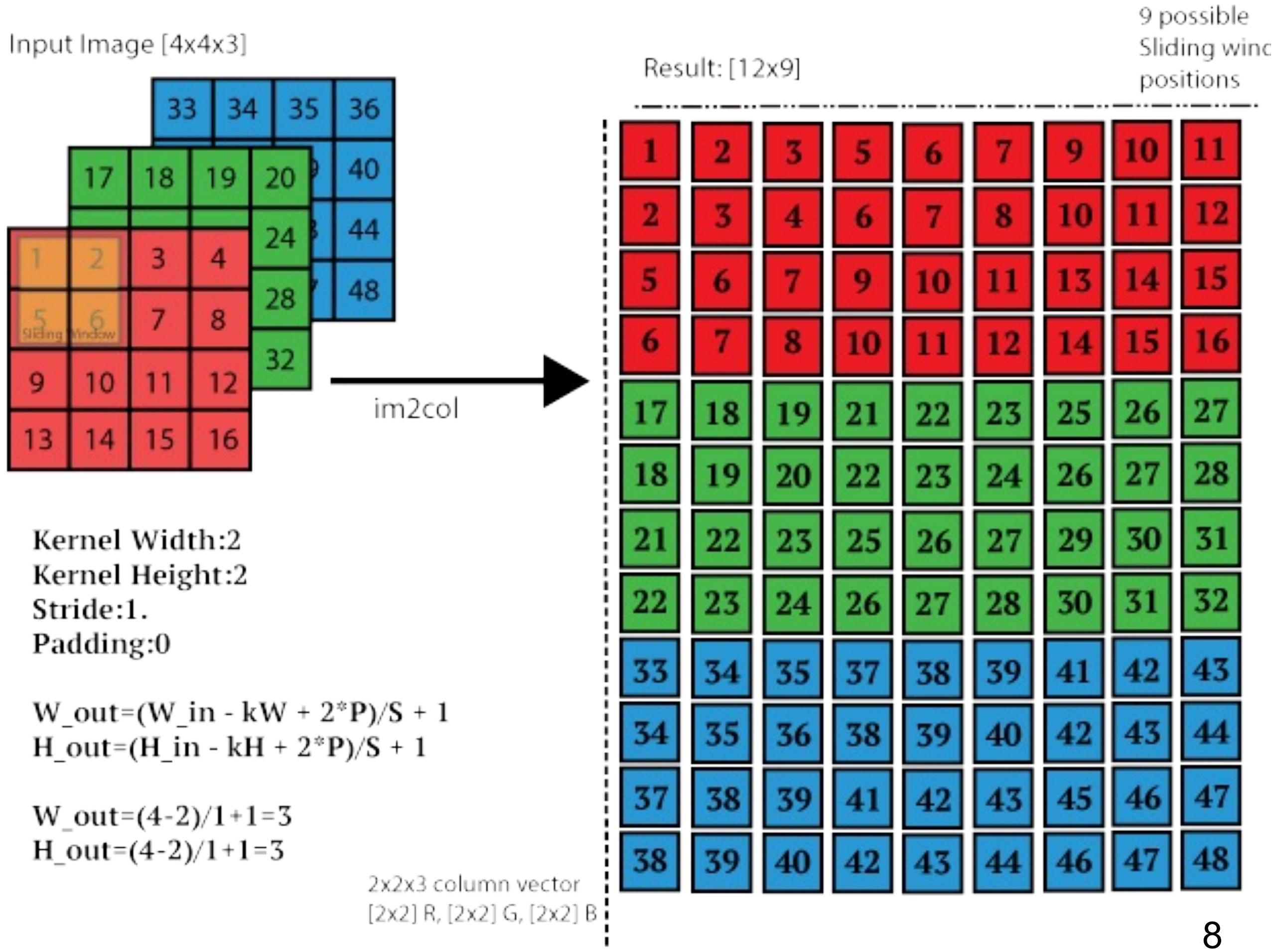
Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



Convolution to Matrix Multiplications

Motivation: transforming convolution to matrix multiplications with improved data layout/locality

- Step 1: image -> matrix

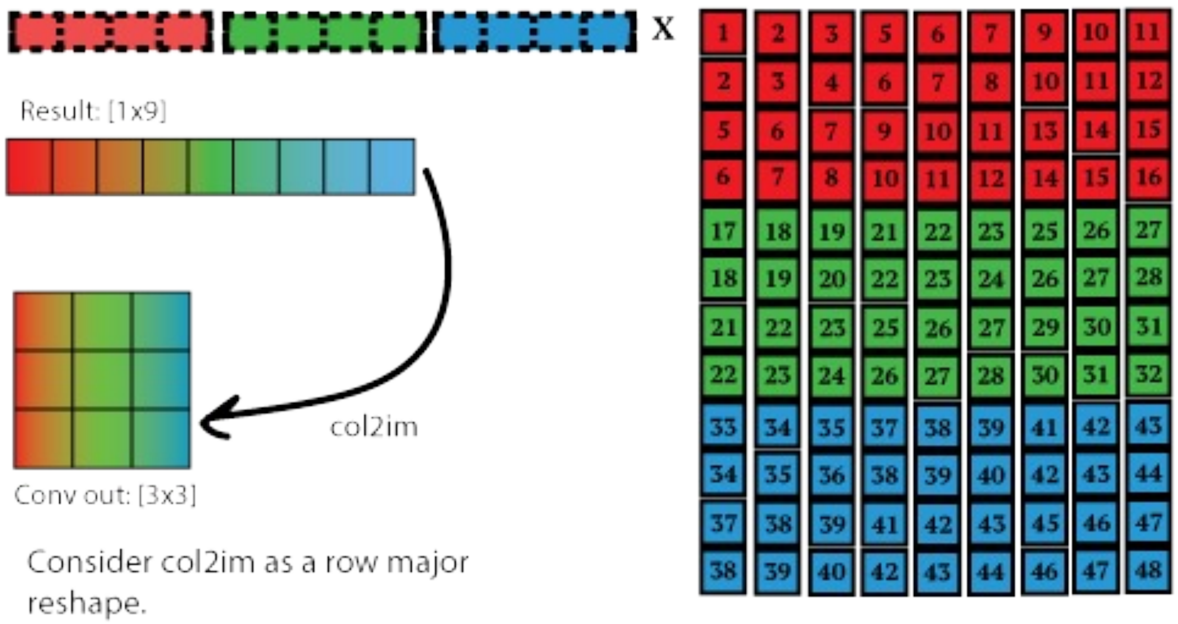


Convolution to Matrix Multiplications

Motivation: transforming convolution to matrix multiplications with improved data layout/locality

- Step 1: image -> matrix
- Step 2: conv -> matrix mul

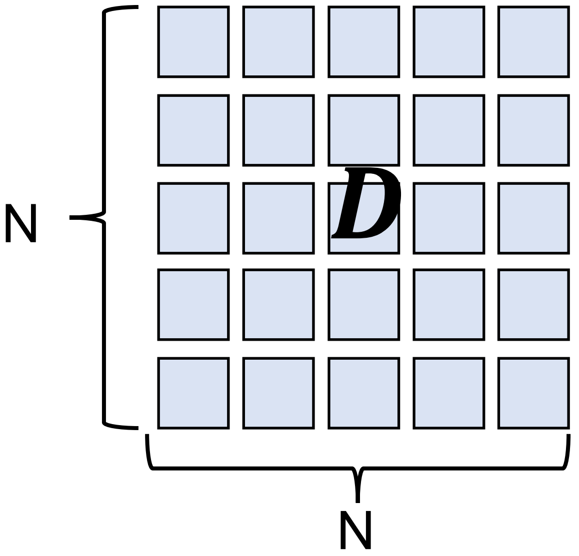
$$O_{x,y} = \sum_{c=1}^{C} \sum_{v=1}^{R} \sum_{u=1}^{S} D_{c,x+u,y+v} * W_{c,u,v}$$

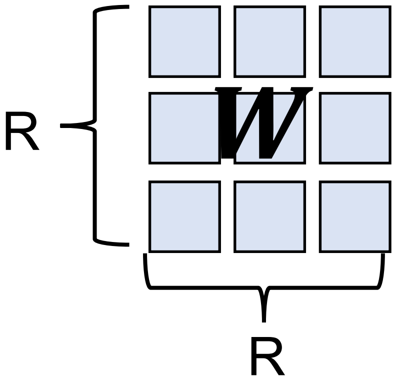


A specific matrix multiplication with data reuse opportunities.

Fast Winograd Convolution

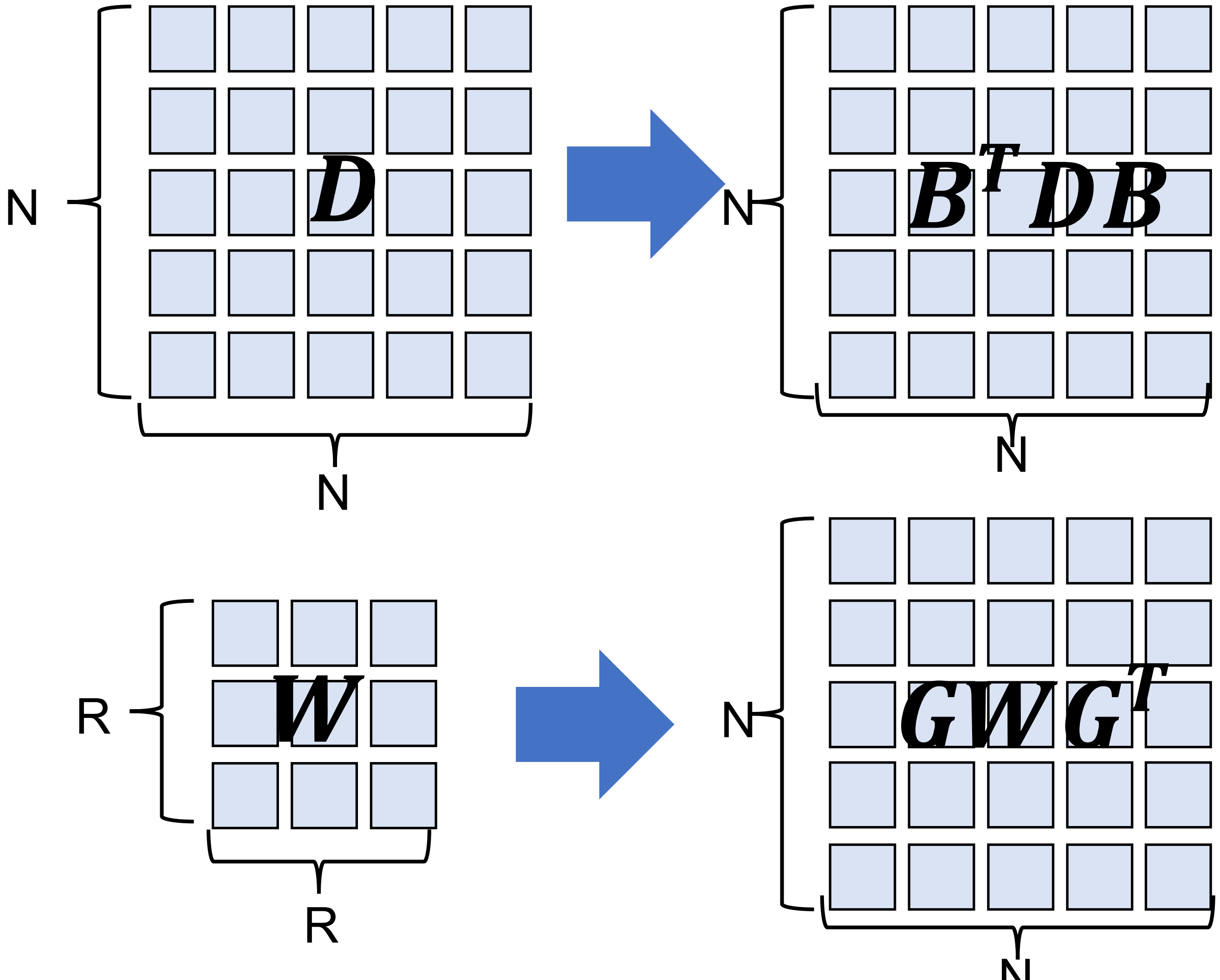
$O = D \circledast W = A^T[GWG^T] \circ [B^T D B]A$, where $A, \mathrm{B},$ and G are constant matrices





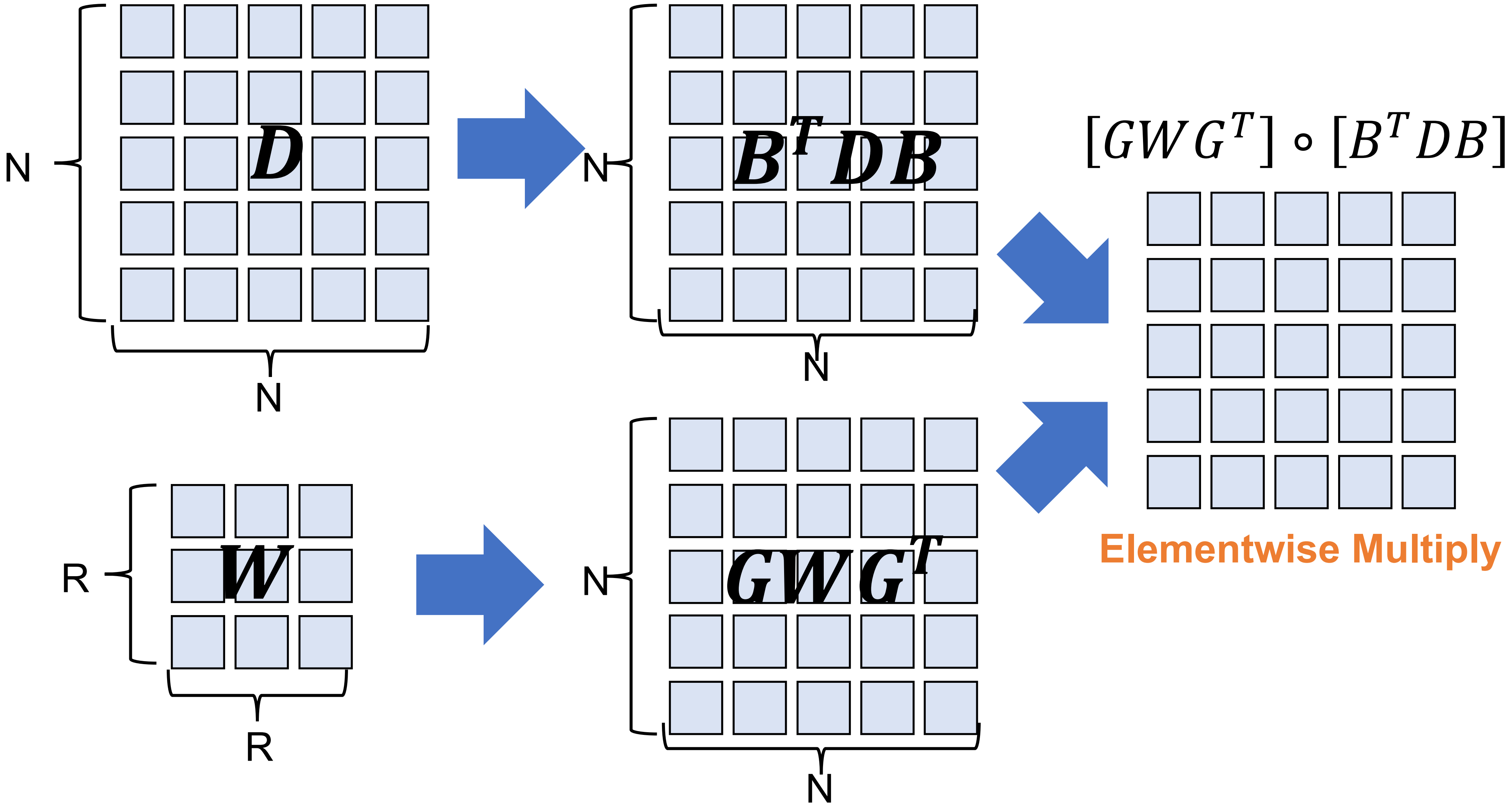
Fast Winograd Convolution

$O = D \circledast W = A^T[GWG^T] \circ [B^T DB]A$, where A, B, and G are constant matrices



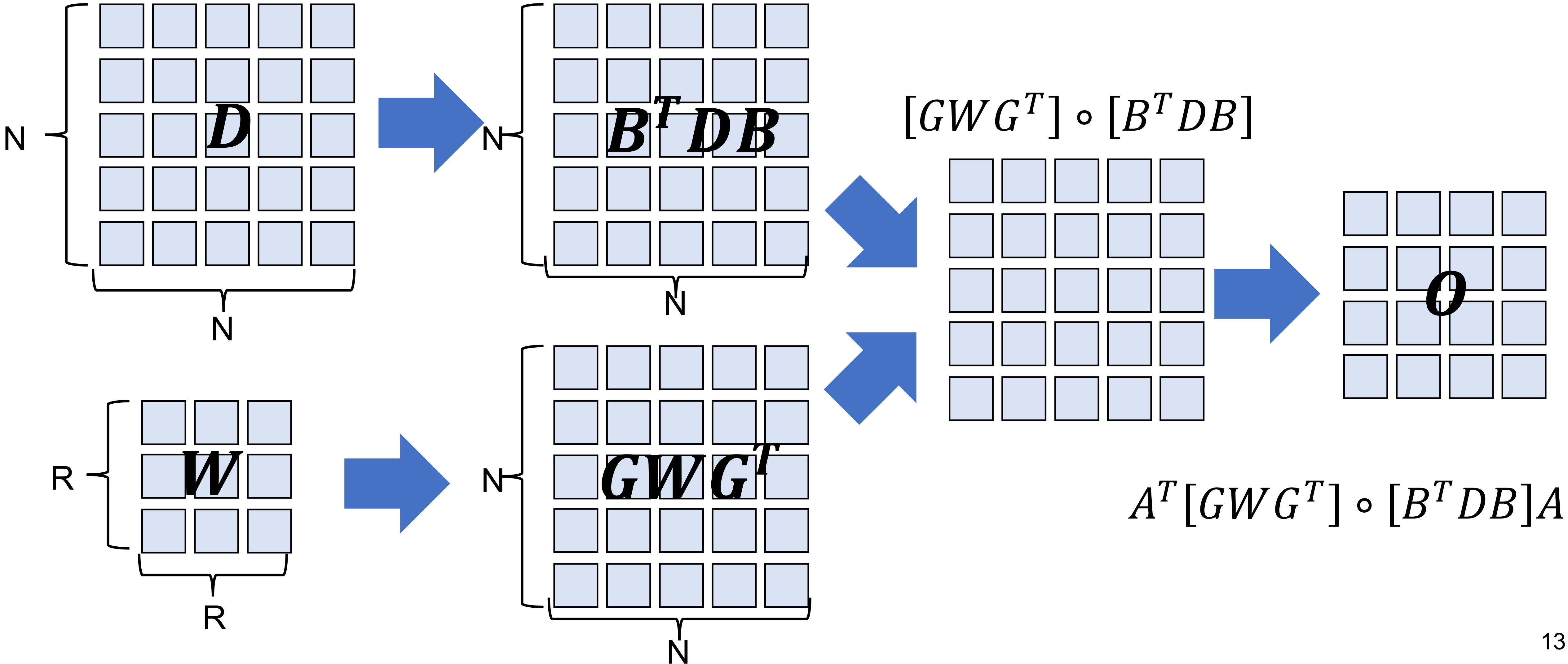
Fast Winograd Convolution

$O = D \circledast W = A^T[GWG^T] \circ [B^TDB]A$, where A, B, and G are constant matrices



Fast Winograd Convolution

$O = D \circledast W = A^T[GWG^T] \circ [B^TDB]A$, where A, B, and G are constant matrices



A Toy Example

- A 2D convolution on a 4x4 image with a filter shape of 2x2
- $O = D \circledast W = A^T[GWG^T] \circ [B^T dB]A$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, G = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Fast Algorithms for Convolutional Neural Networks. Andrew Lavin et al.

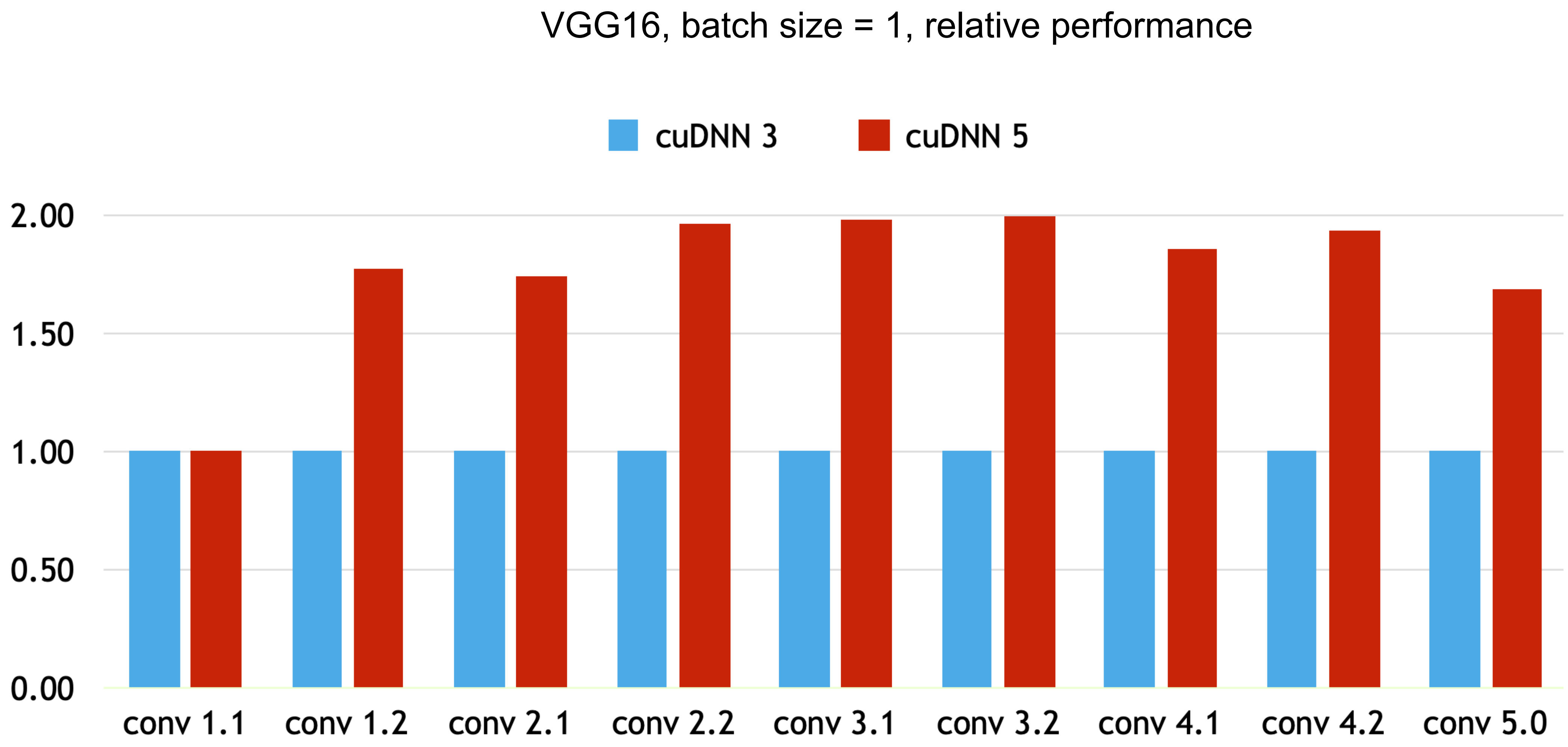
Cost of Winograd Algorithm

$$O = D \circledast W = A^T[GWG^T] \circ [B^T DB]A$$

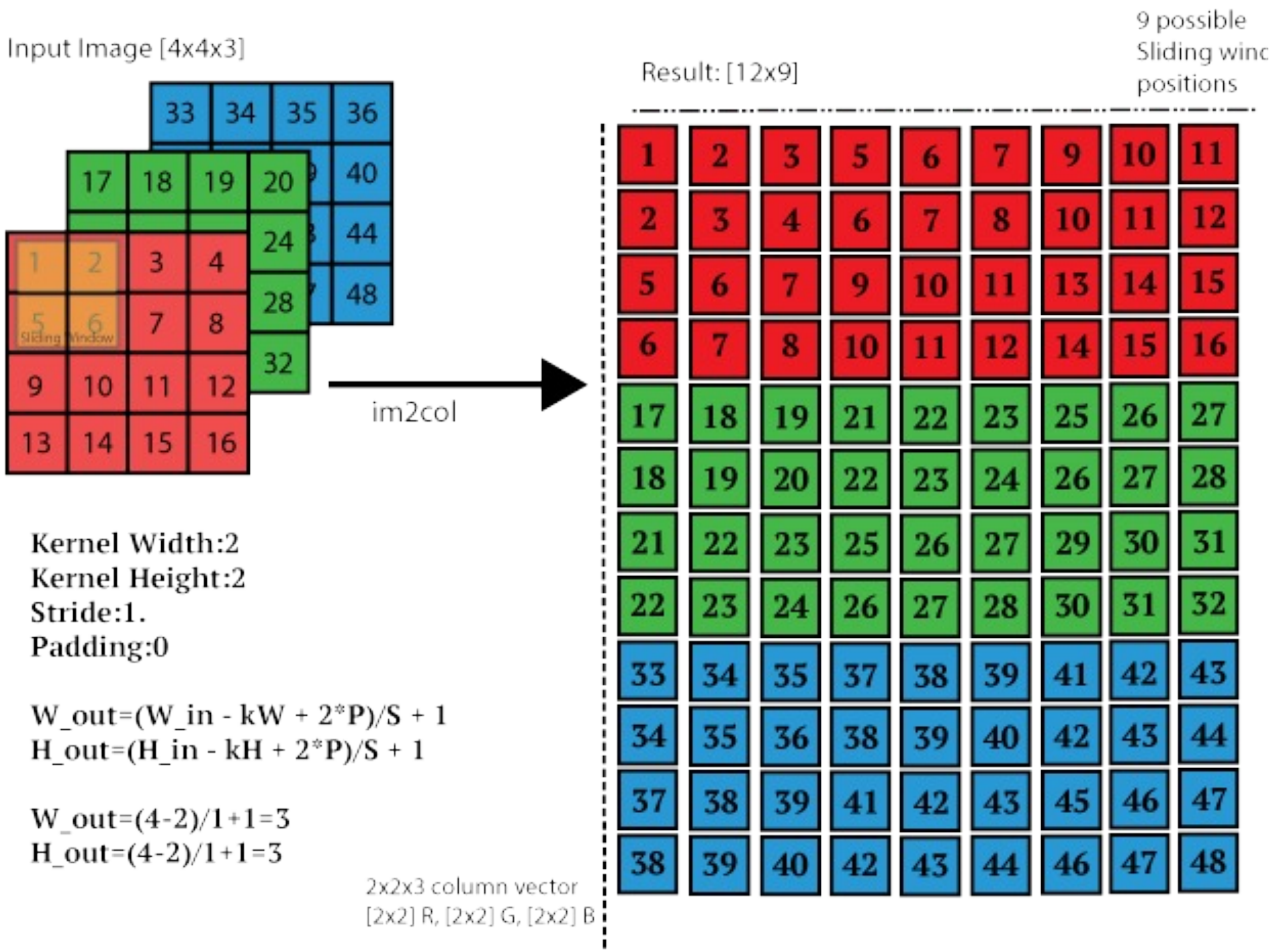
- $GWG^T, B^T\ DB$: only involve constant multiplications, much cheaper than floating point multiplications
- $[GWG^T] \circ [B^T DB]$: N x N floating point multiplications
- $A^T[GWG^T] \circ [B^T DB]A$: constant multiplications

- # floating point multiplications = $N \times N$
- Complexity reduction $\frac{N^2}{R^2 \times (N-R+1)^2}$ (up to 9 times for 3x3 convolutions)
- **What are the other costs of Winograd?**

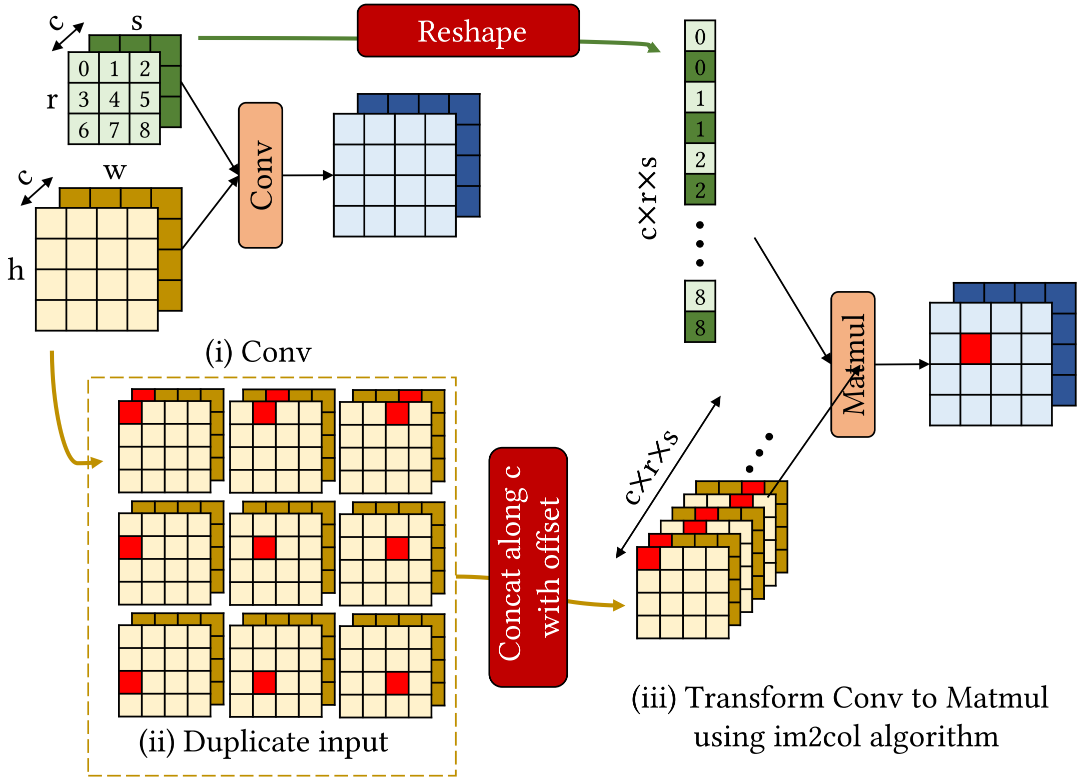
Consistently Outperforms Vanilla Convolutions



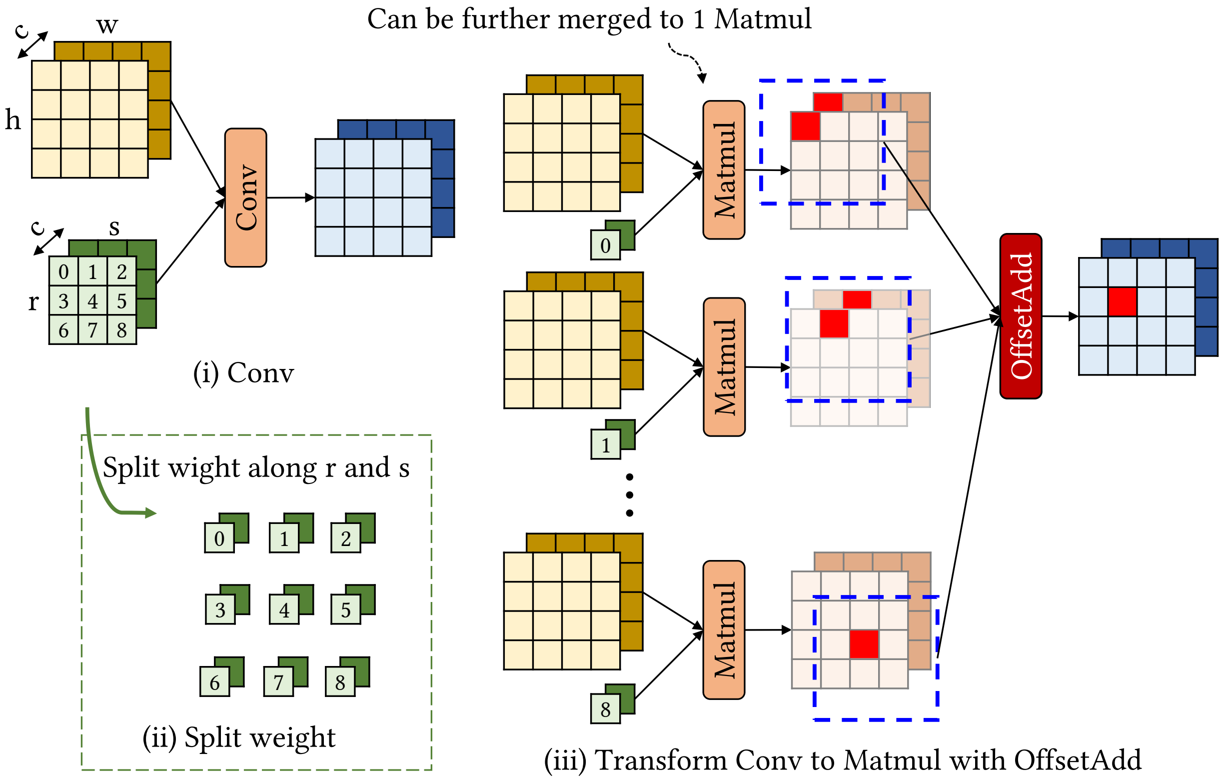
Other Ways to Transform Convolution to Matrix Multiplication?



Other Fast Implementation for Conv2Matmul



Conv2Matmul version 1 (Winograd)

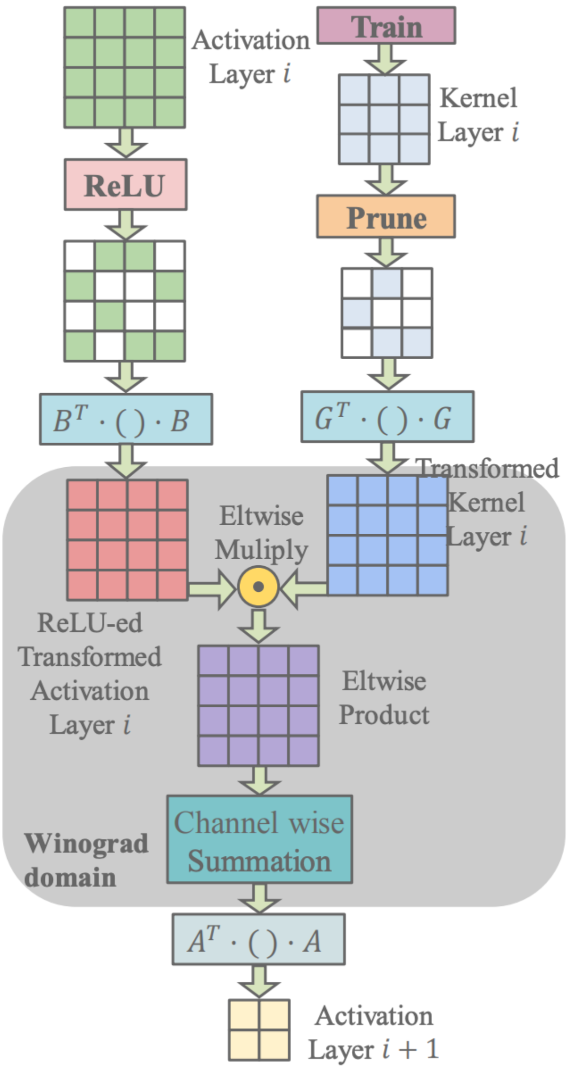


Conv2Matmul version 2 (EINNET)
40% faster than Winograd on A100 GPU

EINNET: Optimizing Tensor Programs with Derivation-Based Transformations. Wang et al.

Combining WinoGrad with Model Compression

- Issue: transformations for inputs and weights convert sparse activations and weights to dense ones



Sparse-Winograd Convolution

Standard Winograd

- $S = A^T \times [G \times Prune(g) \times G^T] \circ [B^T \times ReLU(d) \times B] \times A$
- The sparse matrices Prune(g) and ReLU(d) become dense again when transformed to Winograd domain

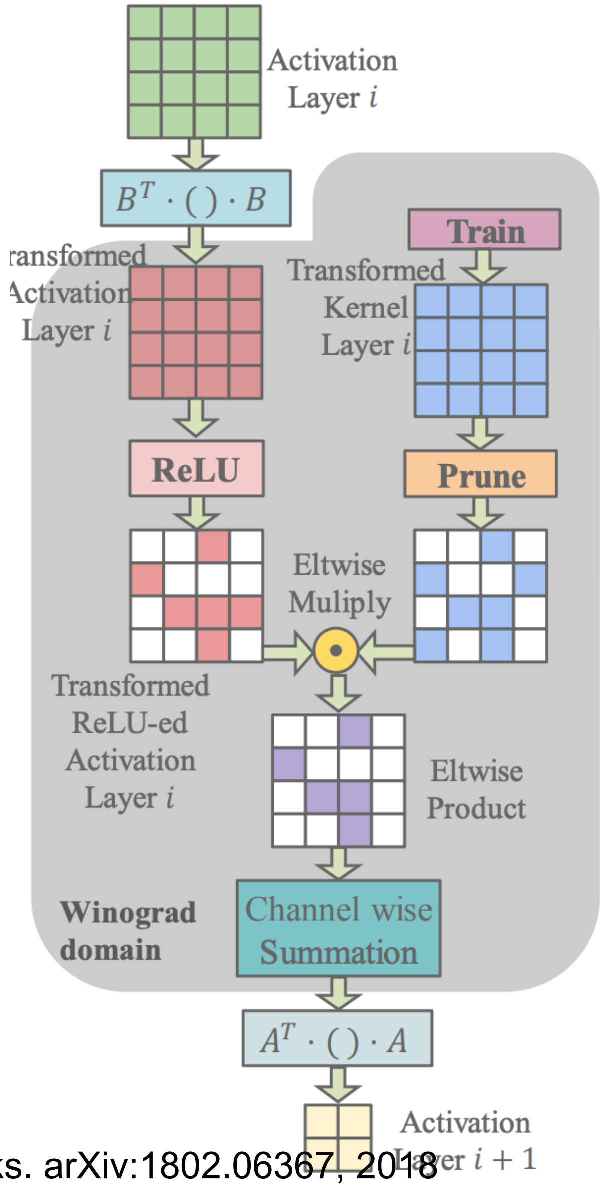
Sparse-Winograd-ReLU CNN:

- $S = A^T [Prune(GgG^T)] \circ [ReLU(B^T dB)] A$
- Move the pruning and ReLU operations into Winograd to make the results sparse

Liu, X; Pool, J; Han, S; Dally, W. Efficient Sparse-Winograd Convolutional Neural Networks. arXiv:1802.06367, 2018

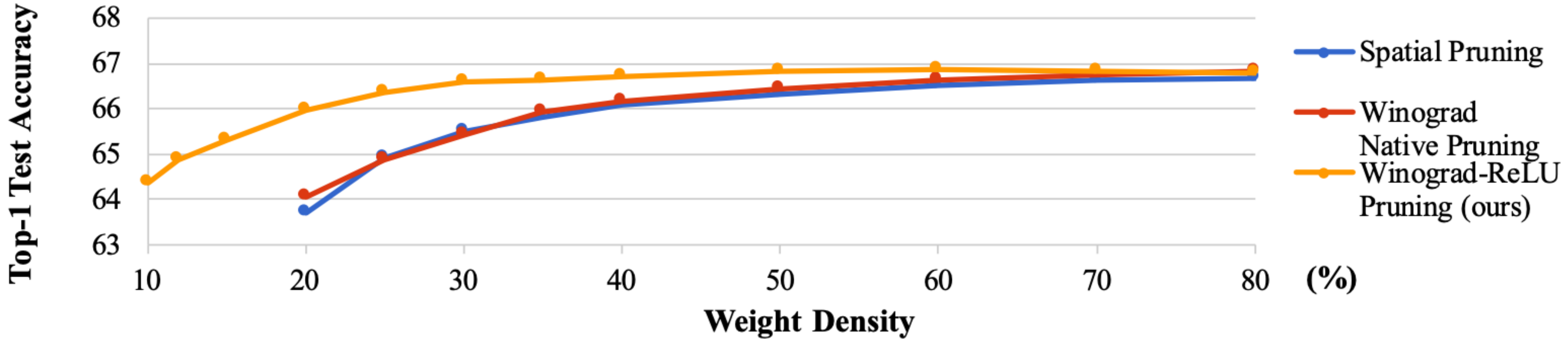
Sparse-Winograd Convolution

- Key Idea: move the pruning and ReLU operations into Winograd to make the results sparse



Liu, X; Pool, J; Han, S; Dally, W. Efficient Sparse-Winograd Convolutional Neural Networks. arXiv:1802.06367, 2018

Sparse Winograd Algorithm Reduces Multiplications by 10x while Maintaining/Improving Accuracy

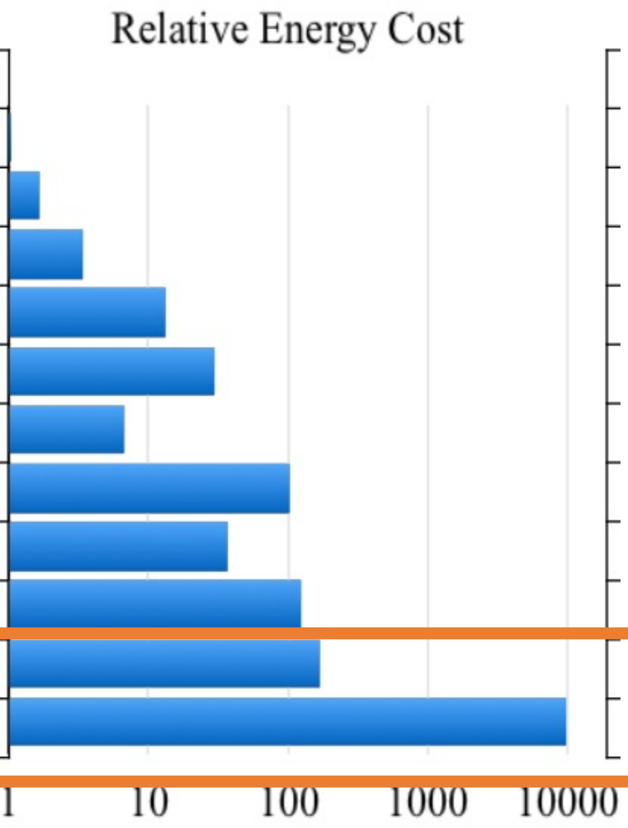


Top-1 validation accuracy vs density for ResNet-18 on ImageNet

Liu, X; Pool, J; Han, S; Dally, W. Efficient Sparse-Winograd Convolutional Neural Networks. arXiv:1802.06367, 2018

Other Techniques to Optimize ML on Hardware

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640



Memory access is a bottleneck

Loop Tiling: Leverage Local Memory for Data Reuse

```
dram float A[n][n], B[n][n], C[n][n];
for (int i = 0; i < n; ++i) {
  for (int j = 0; j < n; ++j) {
    register float c = 0;
    for (int k = 0; k < n; ++k) {
      register float a = A[i][k];
      register float b = B[j][k];
      c += a * b;
    }
    C[i][j] = c;
  }
}
```

A's DRAM accesses: n^3
B's DRAM accesses: n^3

Loop Tiling: Leverage Local Memory for Data Reuse

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];
for (int i = 0; i < n/v1; ++i) {
  for (int j = 0; j < n/v2; ++j) {
    register float c[v1][v2] = 0;
    for (int k = 0; k < n / v3; ++k) {
      register float a[v1][v3] = A[i][k];
      register float b[v2][v3] = B[j][k];
      c += dot(a, b.T);
    }
    C[i][j] = c;
  }
}
```

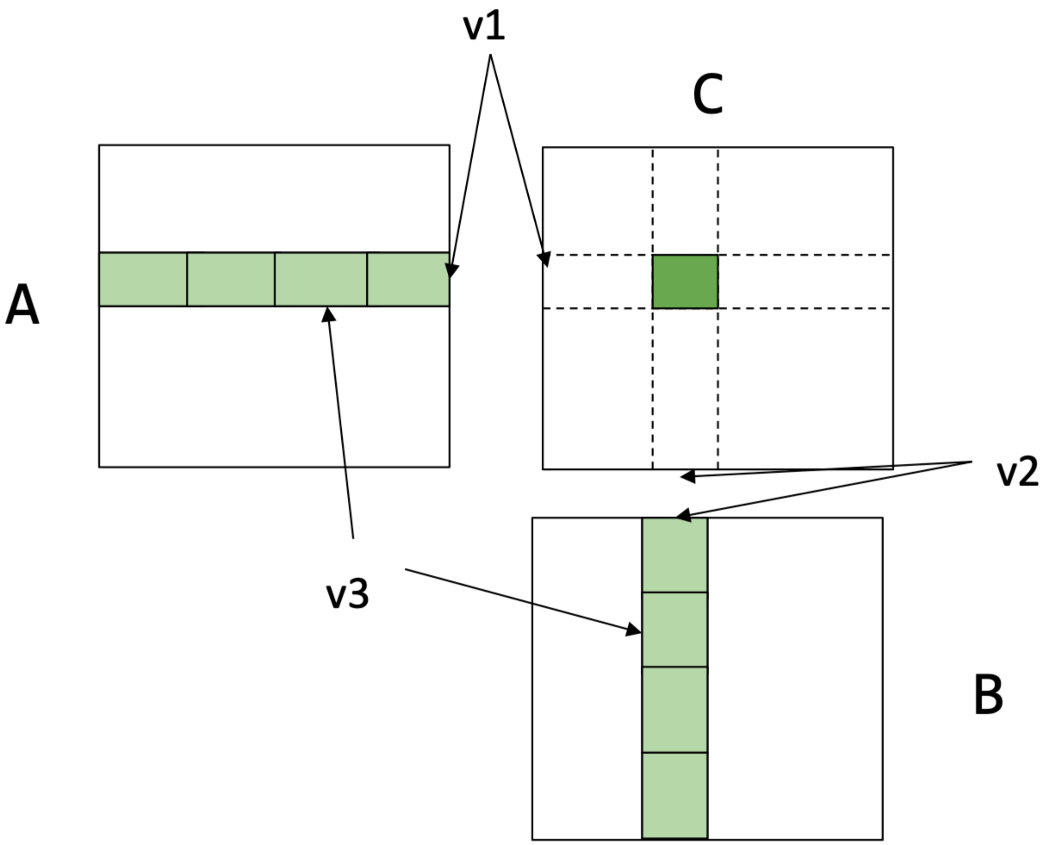


Figure from Tianqi Chen

Tiled Matrix Multiplication

```
dram float A[n/v1][n/v3][v1][v3];
dram float B[n/v2][n/v3][v2][v3];
dram float C[n/v1][n/v2][v1][v2];
for (int i = 0; i < n/v1; ++i) {
  for (int j = 0; j < n/v2; ++j) {
    register float c[v1][v2] = 0;
    for (int k = 0; k < n / v3; ++k) {
      register float a[v1][v3] = A[i][k];
      register float b[v2][v3] = B[j][k];
      c += dot(a, b.T);
    }
    C[i][j] = c;
  }
}
```

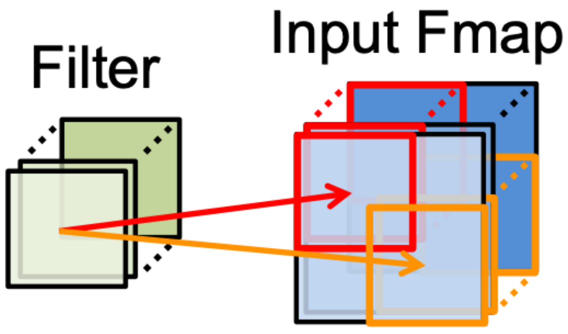
A's DRAM accesses: n^3 / v2
B's DRAM accesses: n^3 / v1
A's register cost: v1 * v3
B's register cost: v2 * v3
C's register cost: v1 * v2

Figure from Tianqi Chen

Common Data Reuse in DNN Computations

Convolutional Reuse

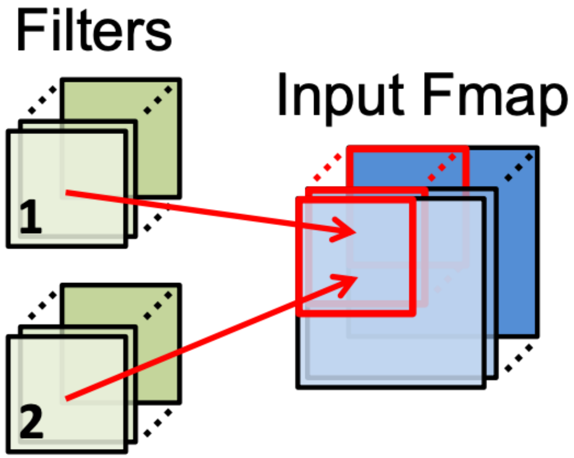
CONV layers only
(sliding window)



Reuse: Activations
Filter weights

Fmap Reuse

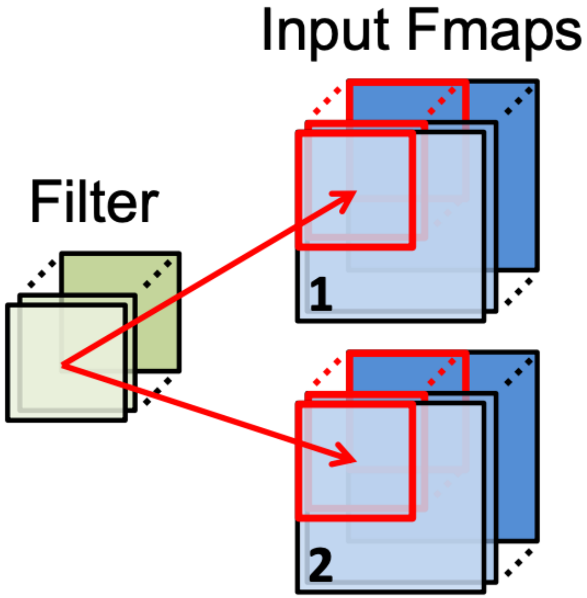
CONV and FC layers



Reuse: Activations

Filter Reuse

CONV and FC layers
(batch size > 1)



Reuse: Filter weights

Discussions

- Q1: Does the Winograd algorithm work for other linear algebra operators in deep learning, such as depth-wise convolution, transposed convolution?

- Q2: What are the hardware optimizations we should consider to make deep learning faster?

Operation:	Energy (pJ)
8b Add	0.03
16b Add	0.05
32b Add	0.1
16b FP Add	0.4
32b FP Add	0.9
8b Mult	0.2
32b Mult	3.1
16b FP Mult	1.1
32b FP Mult	3.7
32b SRAM Read (8KB)	5
32b DRAM Read	640