embedded VISION SUMMIT 2018

Even Faster CNNs: Exploring the New Class of Winograd Algorithms

arm

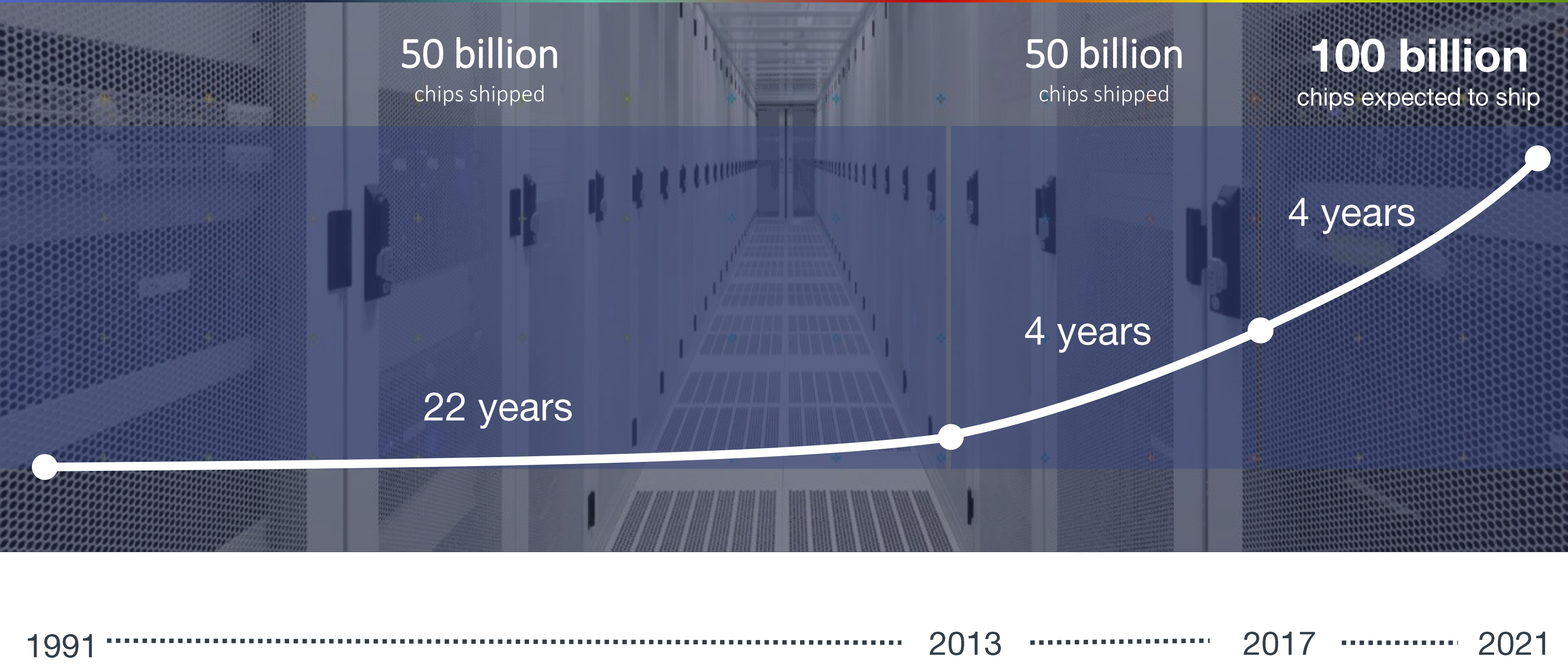
Gian Marco Iodice, Senior SW Engineer, Arm

May 22, 2018

Agenda

- Arm
- Convolutional Neural Network overview
- Efficiently implementing convolution layers
  - Memory coalescing and LWS tuning
- Winograd’s minimal algorithm to speed-up further convolution layers
  - Algorithm design
  - Performance and accuracy evaluation

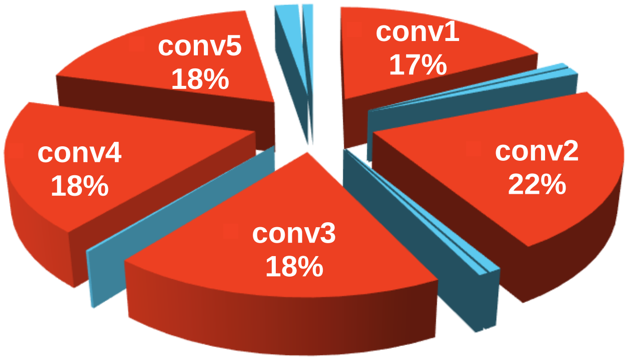
Arm: Extraordinary Growth From Sensors to Server



Convolutional Neural Network Overview

In 2016 I gave a talk about GEMM vs FFT to accelerate Deep Learning

Layer breakdown for AlexNet



FFT is only good for large kernel sizes and with stride = 1

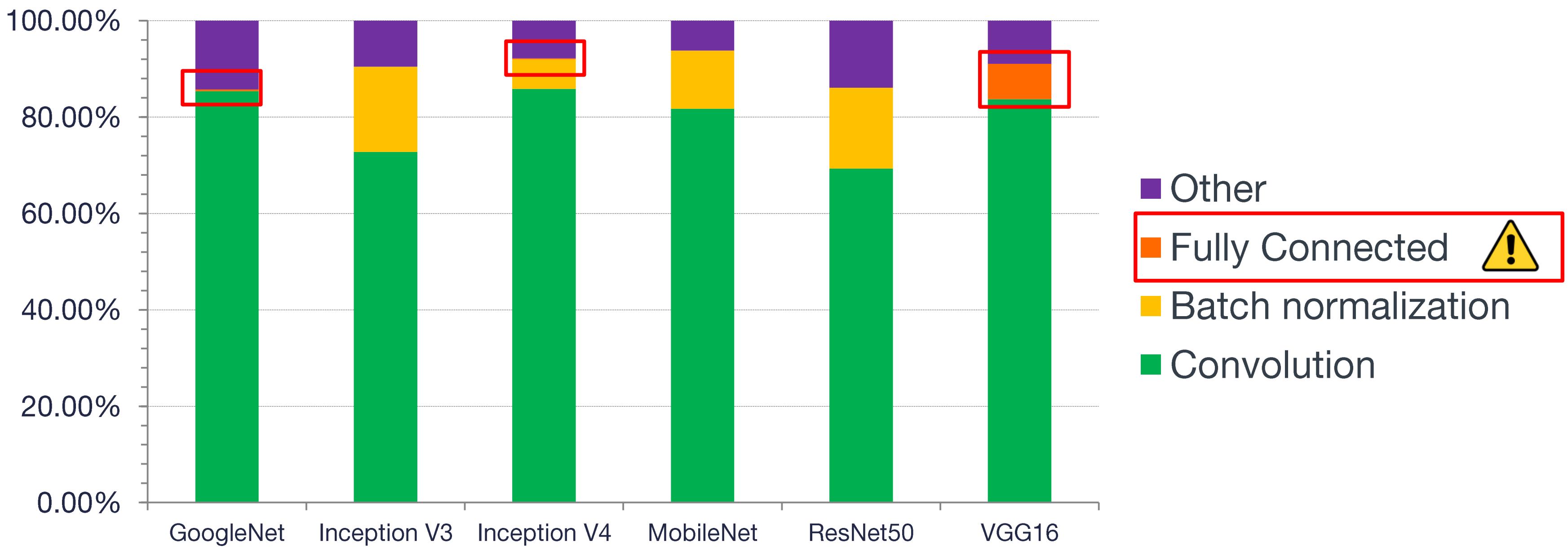
Even Smaller Convolution Kernels…

If we look at the modern CNNs, the current trend is dominated by even smaller convolution kernels (3x3 and 1x1)

1D convolutions (i.e. 1x3, 3x1, 5x1,..)

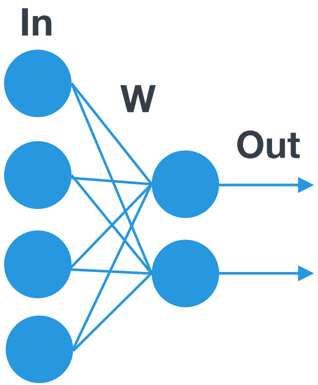
Network	% 1x1 kernels	% 3x3 kernels	% 5x5 kernels	% Others
GoogleNet	64.9	17.5	15.9	1.7
Inception V3	43.2	17.9	3.2	35.7
Inception V4	40.9	16.1	0	43
MobileNet	93.3	6.7	0	0
ResNet50	68.5	29.6	0	1.9
VGG16	0	100	0	0

CNN Layer Breakdown



Fully Connected Layer Issue (1)

Neurons in fully connected layer have connections to all output of previous layer



$$\begin{bmatrix} in_0 & in_1 & in_2 & in_3 \end{bmatrix} \times \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{11} \\ w_{20} & w_{21} \\ w_{30} & w_{31} \end{bmatrix} = \begin{bmatrix} out_0 & out_1 \end{bmatrix}$$

Fully connected as Vector-Matrix multiplication

Fully Connected Layer Issue (2)

Vector-Matrix multiplication is memory bound!

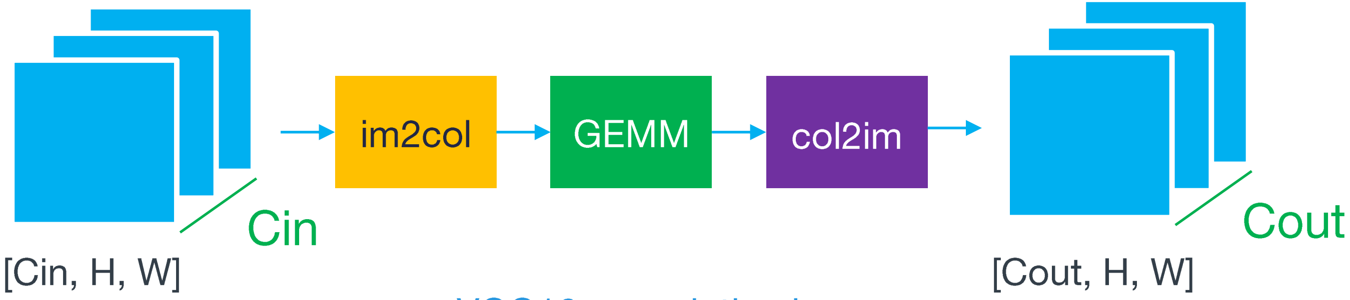
Whilst Matrix-Matrix multiplication performs $O(n^3)$ operations on $O(n^2)$ data, Vector-Matrix performs $O(n^2)$ operations on $O(n^2)$ data

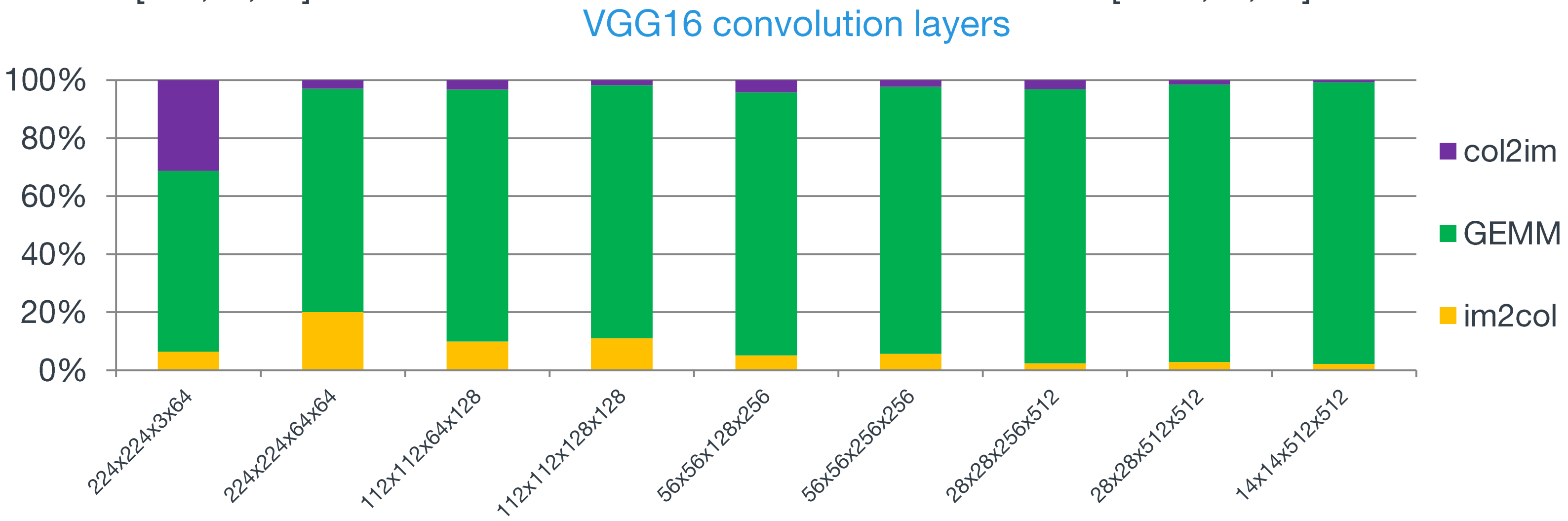
- High ratio between memory accesses and arithmetic instructions
  - difficult to hide memory latency
- Few data re-use
- Limited temporal locality

For this reason in recent networks the number of FC layers is limited

Efficiently Implementing Convolution Layer

GEMM-based Convolution

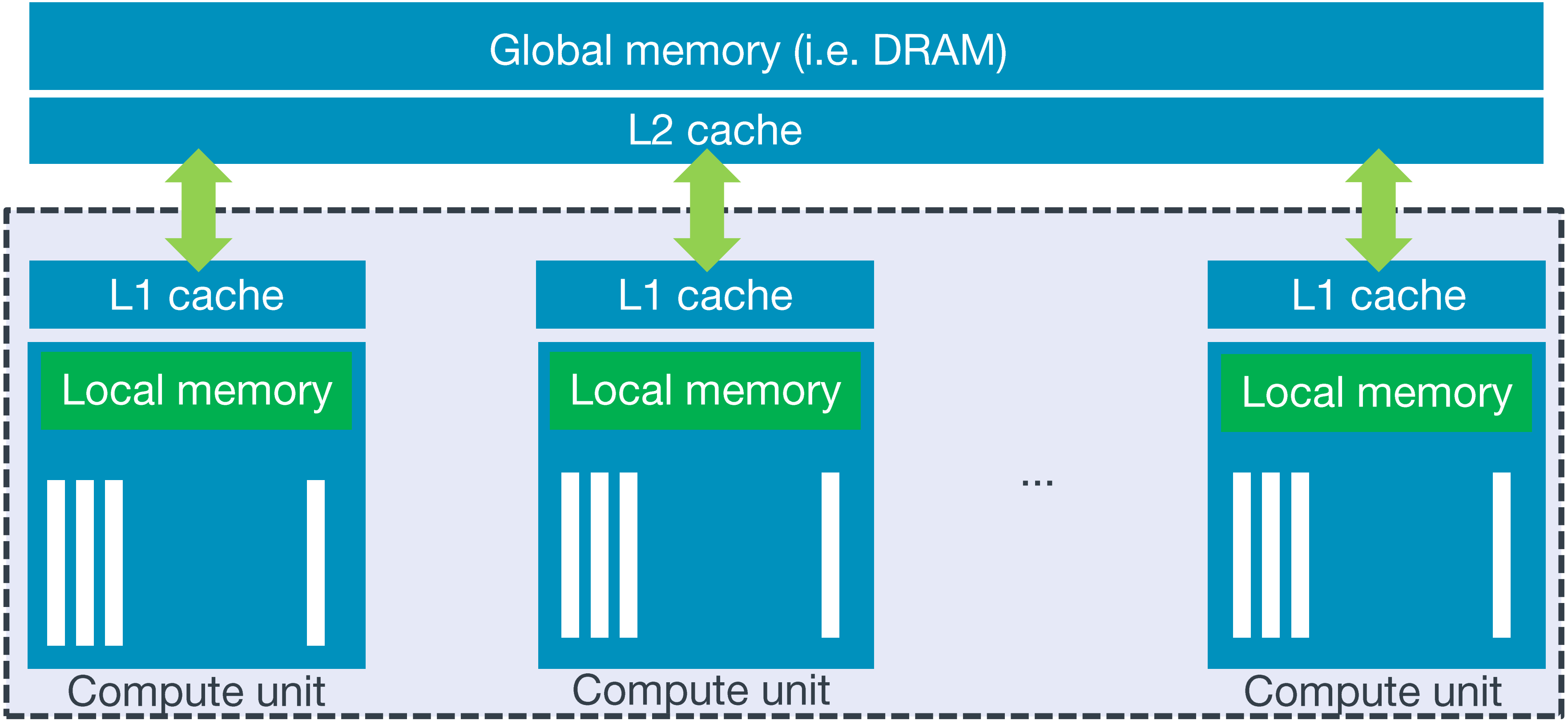




What Did We Do to Improve the Performance?

- CNN with low precision arithmetic
  - IEEE Half Floating Point (fp16) or INT8 quantized
- Improving GPU cache utilization
  - Memory coalescing and LWS tuning (GPU)

OpenCL Concepts: Platform Model

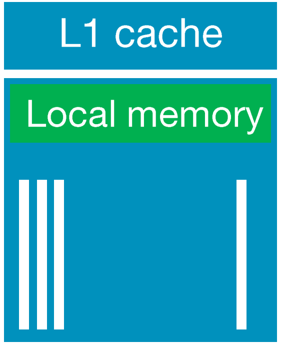


OpenCL Concepts: Compute Unit

- A compute device (i.e. GPU, CPU, accelerator...) is made up of several compute units
- Each compute unit (i.e. GPU core, CPU core) has its own L1 cache and can execute N threads in parallel (a.k.a. work-items)
- Each thread executes the same piece of code (OpenCL kernel)
- The thread id (Global-Id/Local-Id) can be used to access different memory locations

OpenCL Concepts: Work-items/work-group

- Each work-item has private registers
- Collection of threads on a same compute unit is called local-work-group (LW)
  - The LW size (LWS) is configurable
- Syncronization available only for threads within the same work-group

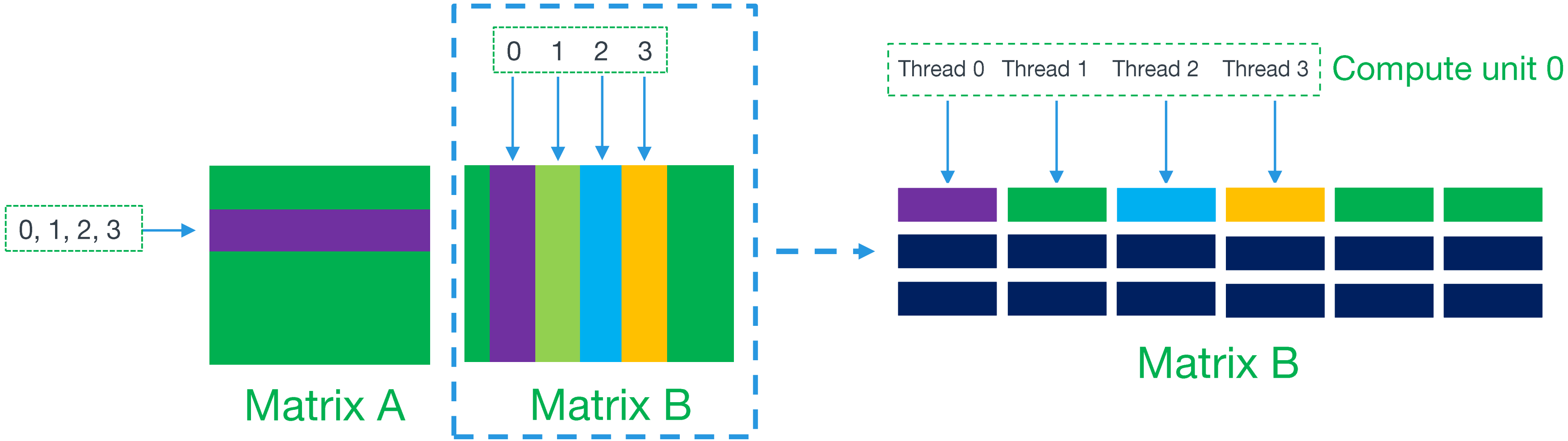


Compute unit

Improving L1 Cache Utilization: Memory Coalescing

Idea: Threads of the same work-group should access consecutive memory addresses (memory coalescing)

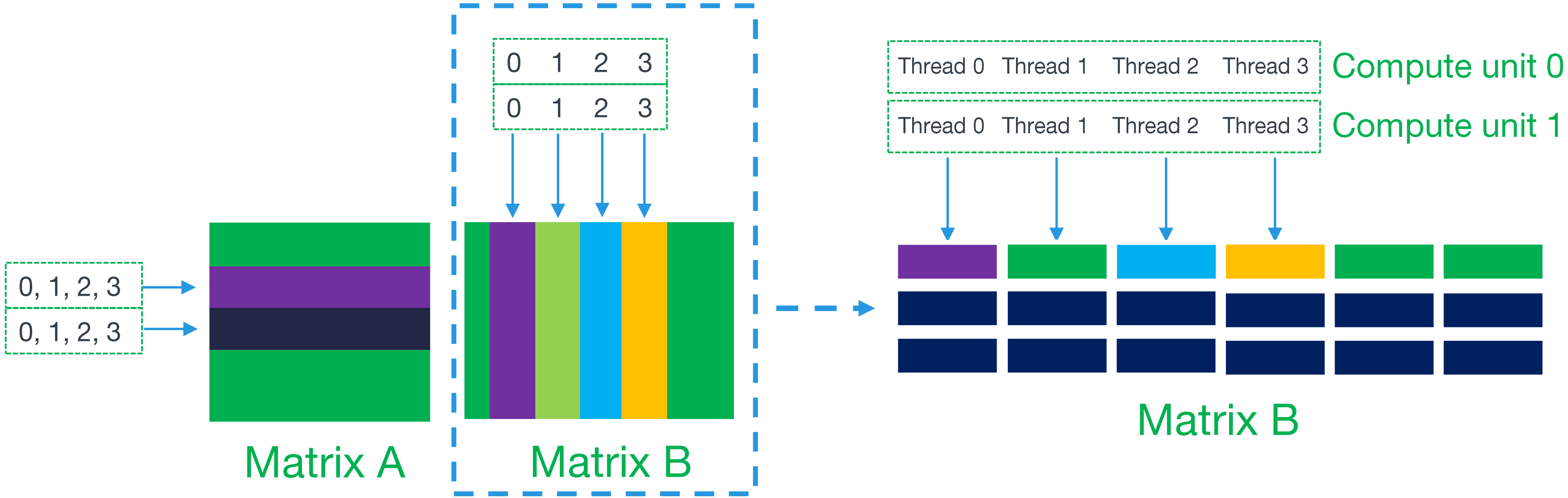
Example: Matrix multiplication



Improving L2 Cache Utilization Tuning LWS (1)

Idea: Re-use the same memory block between different compute units

Example: Matrix multiplication



Improving L2 Cache Utilization Tuning LWS (2)

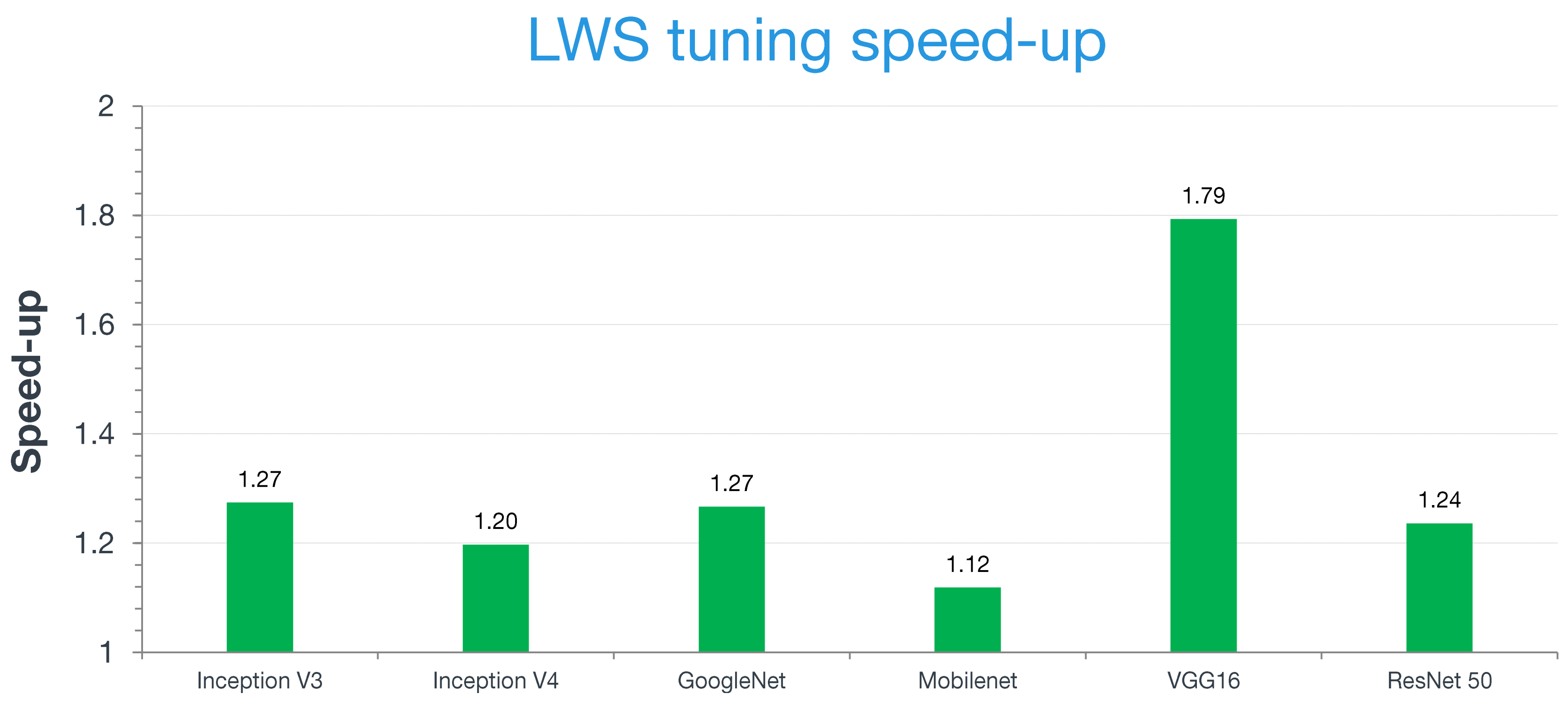
embedded VISION SUMMIT 2018

- Tweaking the number of work-items in a work-group (LWS) can have a huge performance impact

- However setting the optimal LWS can be tricky:
  - Cache size
  - Maximum number of threads per compute unit
  - Work-group dispatching
  - Input and output dimensions
  - …

- In Arm Compute Library we implemented LWS tuner to look for the optimal configuration (brute force approach)

Improving L2 Cache Utilization Tuning LWS (3)



Winograd’s Minimal Algorithm*

*Fast algorithm for Convolutional Neural Networks, Andrew Lavin, Scott Gray

Goal

Reducing the number of multiplications tackling direct convolution (not GEMM!)

Let's start from the following matrix multiplication:

$$\underbrace{\begin{bmatrix} d_{00} & d_{01} & d_{02} \\ d_{10} & d_{11} & d_{12} \end{bmatrix}}_{2x3} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}}_{3x1} = \underbrace{\begin{bmatrix} r_0 \\ r_1 \end{bmatrix}}_{2x1}$$

This matrix multiplication requires: **6 multiplications and 4 additions**

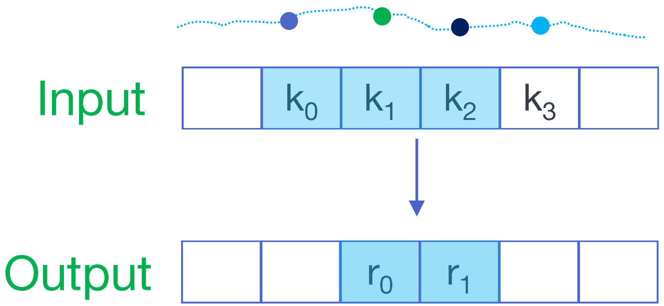
$$r_0 = (d_{00} \cdot w_0) + (d_{01} \cdot w_1) + (d_{02} \cdot w_2)$$

$$r_1 = (d_{10} \cdot w_0) + (d_{11} \cdot w_1) + (d_{12} \cdot w_2)$$

Winograd's Minimal Filtering Algorithm (1)

The same matrix multiplication can be used to compute the output of two consecutive 3-tap FIR filters (only 4 input values required)

$$\underbrace{\begin{bmatrix} k_0 & k_1 & k_2 \\ k_1 & k_2 & k_3 \end{bmatrix}}_{2\text{x}3} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}}_{3\text{x}1} = \underbrace{\begin{bmatrix} r_0 \\ r_1 \end{bmatrix}}_{2\text{x}1}$$



Winograd found an algorithm to perform this with only **4 multiplications**

Winograd's minimal filtering algorithm

Winograd's Minimal Filtering Algorithm (2)

$$\underbrace{F(2,3)}_{\text{Output size}}^{\text{Filter size}} = \underbrace{\begin{bmatrix} k_0 & k_1 & k_2 \\ k_1 & k_2 & k_3 \end{bmatrix}}_{2x3} \underbrace{\begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}}_{3x1} = \underbrace{\begin{bmatrix} m_0 + m_1 + m_2 \\ m_1 - m_2 - m_3 \end{bmatrix}}_{2x1}$$

$$m_0 = (k_0 - k_2) \cdot w_0 \qquad m_1 = (k_1 + k_2) \cdot \boxed{\frac{w_0 + w_1 + w_2}{2}}$$

Precomputed if constant

$$m_3 = (k_1 - k_3) \cdot w_2 \qquad m_2 = (k_2 - k_1) \cdot \boxed{\frac{w_0 - w_1 + w_2}{2}}$$

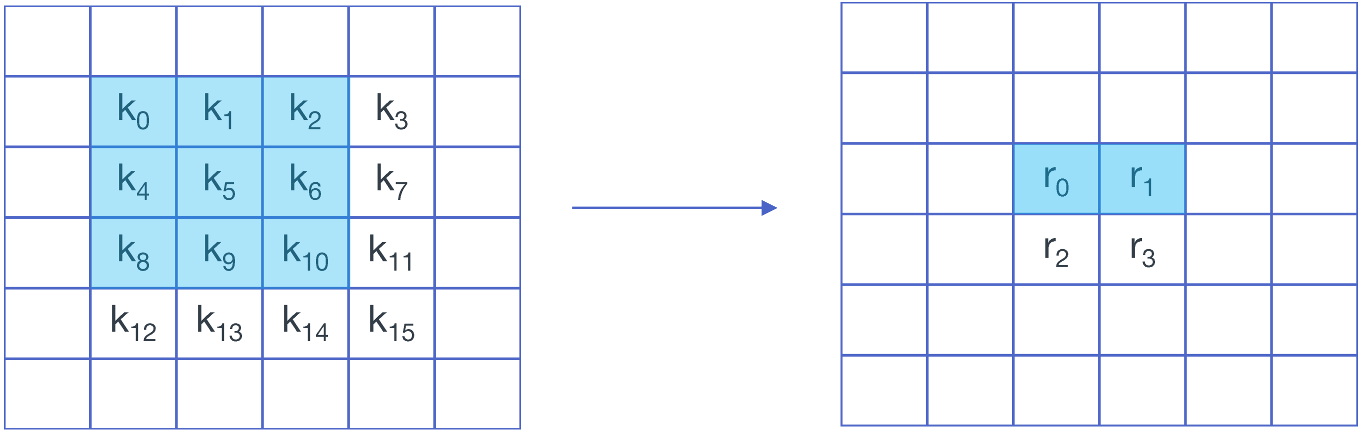
Precomputed if constant

2D Case: Nest Minimal 1D Algorithms (1)

"We can nest minimal 1D algorithms to form minimal 2D algorithms*"

A 3x3 filter needs 16 input values to compute a 2x2 output tile

(Only if **stridex = 1** and **stridey = 1**)



Input image

Output image

*Fast Algorithms for Convolutional Neural Networks, Andrew Lavin, Scott Gray

2D Case: Nest Minimal 1D Algorithms (2)

F(2x2,3x3)

$$\underbrace{\begin{pmatrix} k_0 & k_1 & k_2 & k_4 & k_5 & k_6 & k_8 & k_9 & k_{10} \\ k_1 & k_2 & k_3 & k_5 & k_6 & k_7 & k_9 & k_{10} & k_{11} \\ k_4 & k_5 & k_6 & k_8 & k_9 & k_{10} & k_{12} & k_{13} & k_{14} \\ k_5 & k_6 & k_7 & k_9 & k_{10} & k_{11} & k_{13} & k_{14} & k_{15} \end{pmatrix}}_{4x9} \underbrace{\begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{pmatrix}}_{9x1} = \underbrace{\begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{pmatrix}}_{4x1}$$

	k_0	k_1	k_2	k_3	
	k_4	k_5	k_6	k_7	
	k_8	k_9	k_{10}	k_{11}	
	k_{12}	k_{13}	k_{14}	k_{15}	

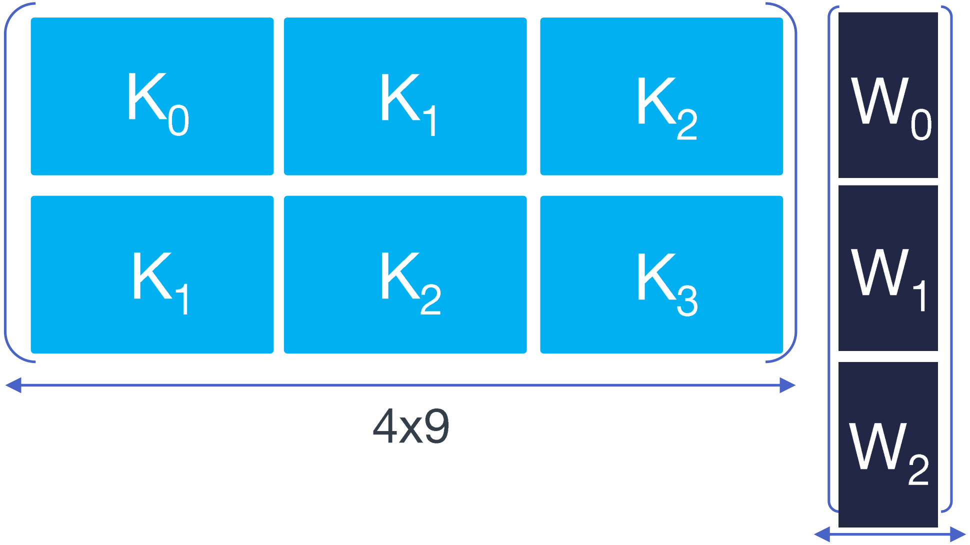
Input image

w_0	w_1	w_2
w_3	w_4	w_5
w_6	w_7	w_8

Filter

2D Case: Nest Minimal 1D Algorithms (3)

F(2x2,3x3)



$$\begin{bmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix}$$

Complexity Reduction

$$\begin{bmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{bmatrix}$$

Matrix multiply F(2,3)! **4** multiplications

$$M_0 = (K_0 - K_2) \cdot W_0$$

$$M_1 = (K_1 + K_2) \cdot \frac{W_0 + W_1 + W_2}{2}$$

$$M_3 = (K_1 - K_3) \cdot W_2$$

$$M_2 = (K_2 - K_1) \cdot \frac{W_0 - W_1 + W_2}{2}$$

16 multiplications instead of **36** of direct convolution

2.25 multiplication complexity reduction!

Algorithm Design (1)

Input transform

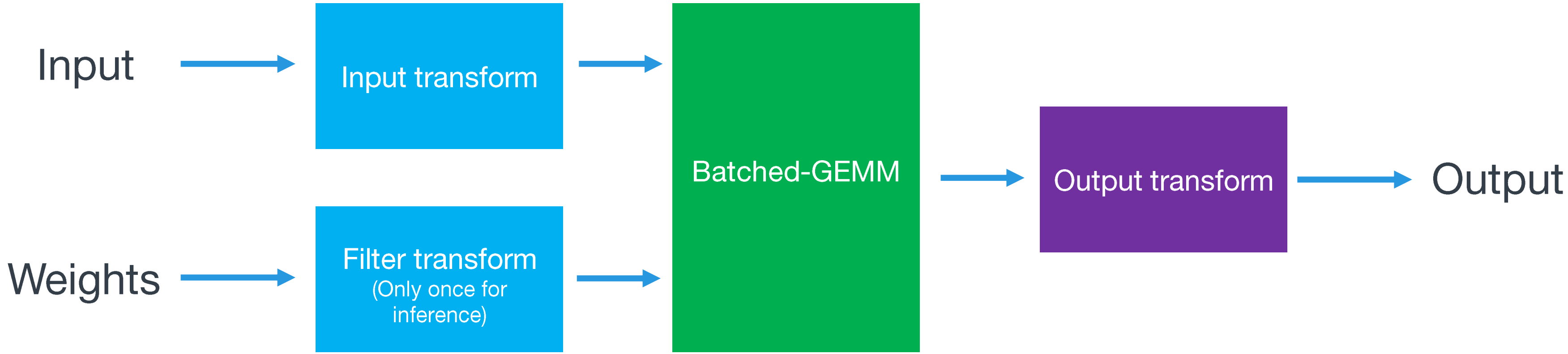
$$Y = AT[(GwG^T) \odot (B^TkB)]A$$

Filter transform

Hadamard product
(Element-wise multiplication)

- Input transform
- Filter transform (if the weights are constants, only once)
- Hadamard product (Element-wise multiplication)
- Output transform

Algorithm Design (2)



Input Transform

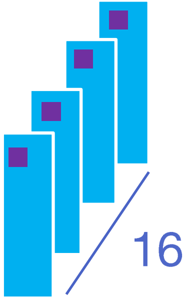
For each 4x4 input tile (50% overlapped)

1. Compute T_i
2. Store the transformed values across the 16 channels (4x4 - > 1x1x16)
3. Memory footprint increases of 4x

[Cin, H, W]

$$\left(\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}^T \cdot \blacksquare \right) \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 1 \\ -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

T_i

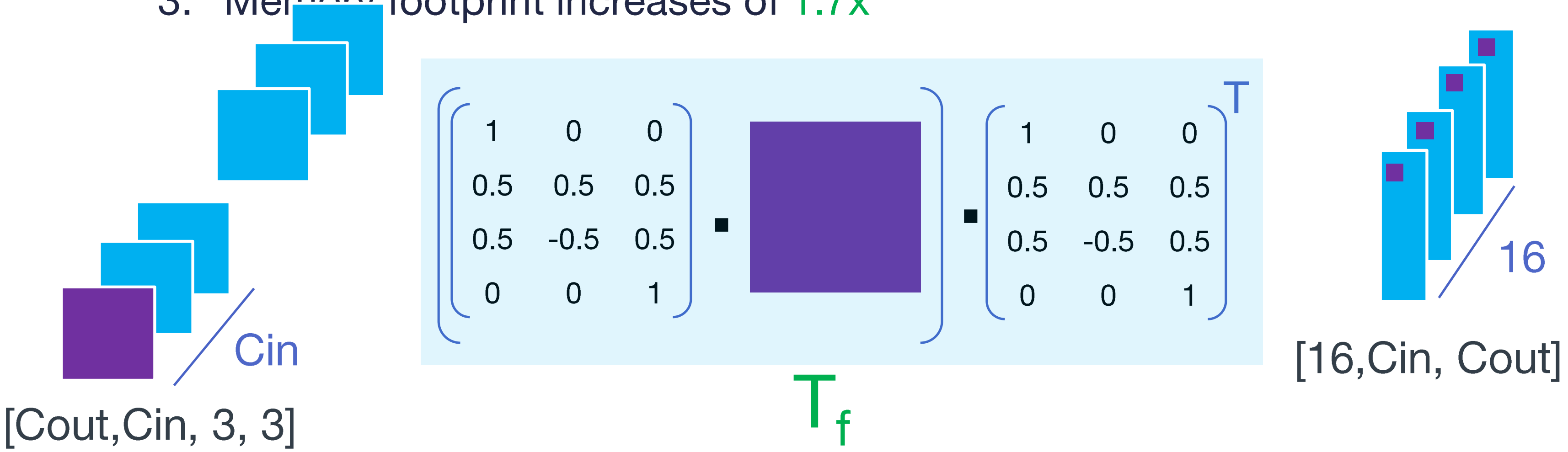


[16, (W*H/4), Cin]

Filter Transform

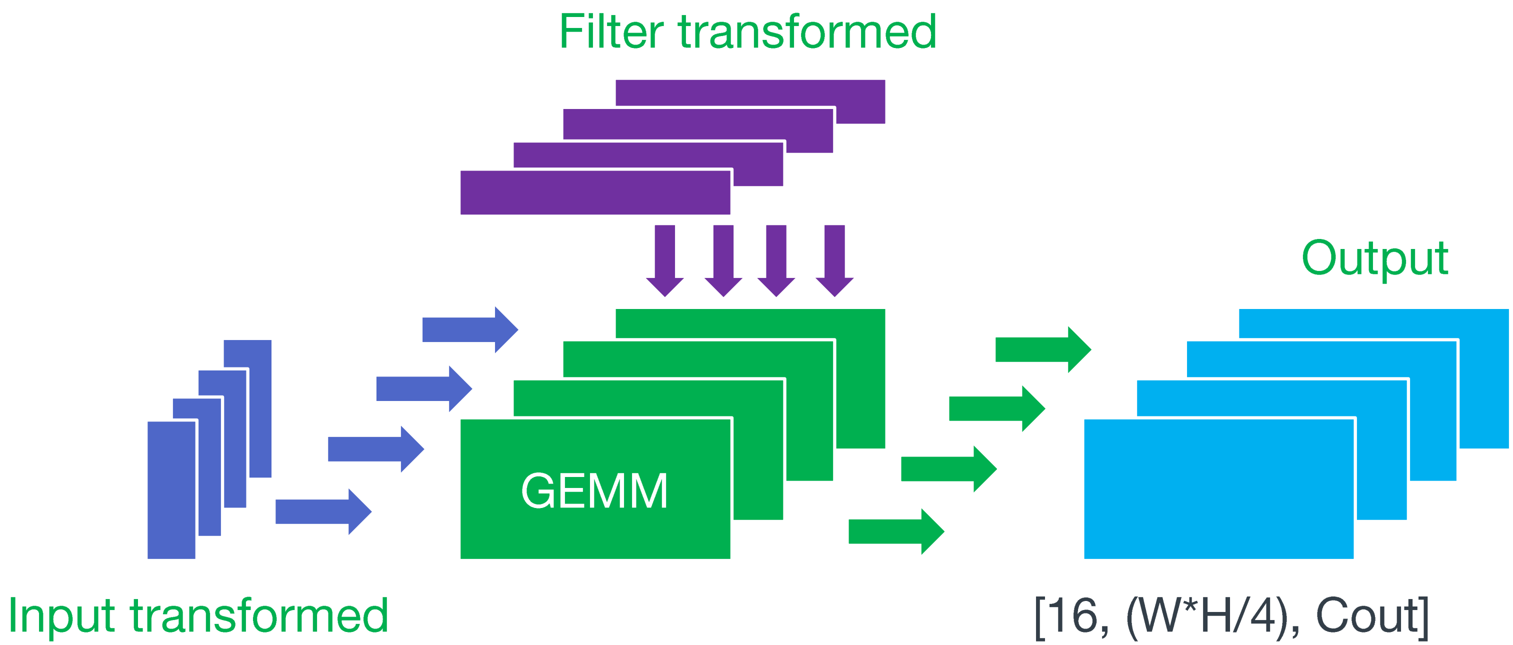
Similar to the input transform. Extract each 3x3 filter plane

1. Compute T_f
2. Store the transformed values across the 16 channels (4x4 - > 1x1x16)
3. Memory footprint increases of 1.7x



Element-wise Multiplication as Batched GEMM

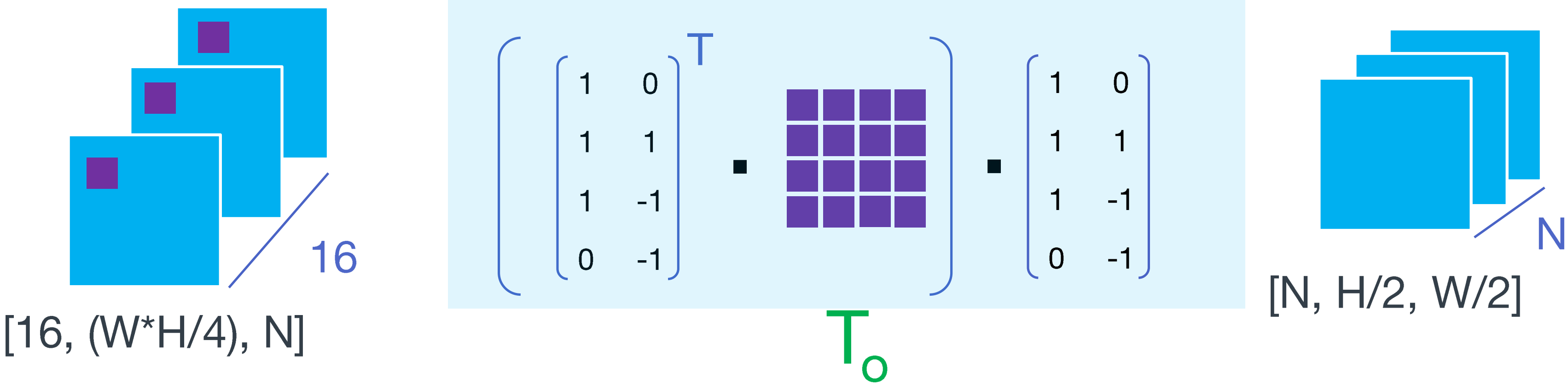
Element-wise multiplication can be reduced to batched GEMM (16 GEMMs)



Output Transform

The output transform:

1. Combine results across channels to form a 4x4 tile
2. Compute To
3. Store the 2x2 output tile in the space domain



Memory Footprint: GEMM-based vs Winograd-based

- Assuming 3x3 kernels, unit pads and unit strides
  - W: Width image, H: Height image, Cin: Input channels and Cout: Output channels

	GEMM-based	Winograd-based	
	Im2col	**Input Transform**	**Input transform / im2col**
Size (elements)	W * H * 3 * 3 * Cin	16 * (W/2) * (H/2) * Cin	0.44
	n.a.	**Filter transform**	
Size (elements)		16 * Cin * Cout	
	Col2im (only if NCHW)	**Output transform**	**Output transform / col2im**
Size (elements)	W * H * Cout	16 * (W/2) * (H/2) * Cout	4

Optimizing Input/Output Transform

Input, (filter) and output transform must be carefully optimized

- Expanding out the matrix multiplications can help a lot

$$\begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} = \begin{bmatrix} a_0 * b_0 + a_1 * b_1 \\ a_2 * b_0 + a_3 * b_1 \end{bmatrix}$$

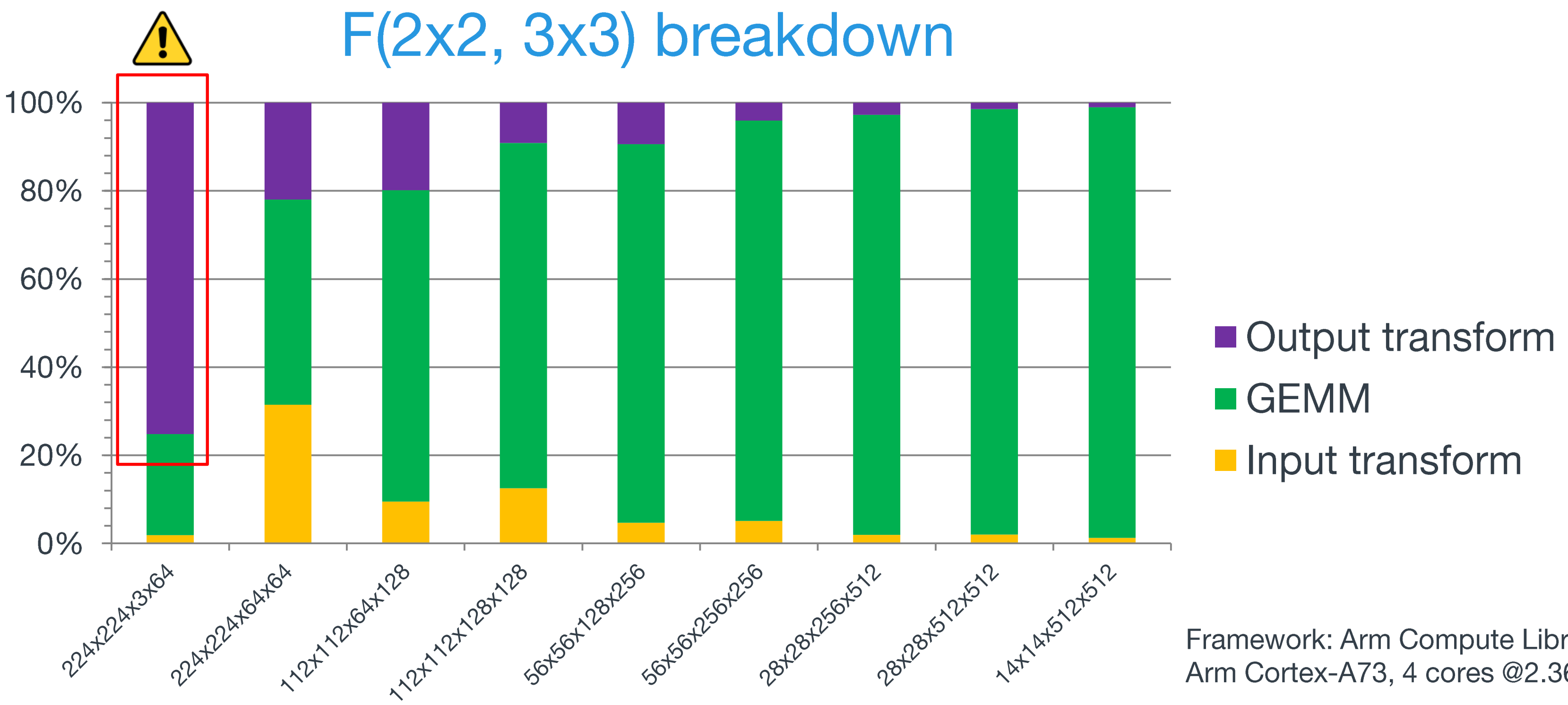
  - No for loops needed
  - Re-use of partial results in different places
    - The compiler is not always able to decompose the operations to preserve fp32 arithmetic ordering (i.e. factorization)

Optimizing batched-GEMM

The same general optimizations for GEMM are still valid for batched-GEMM

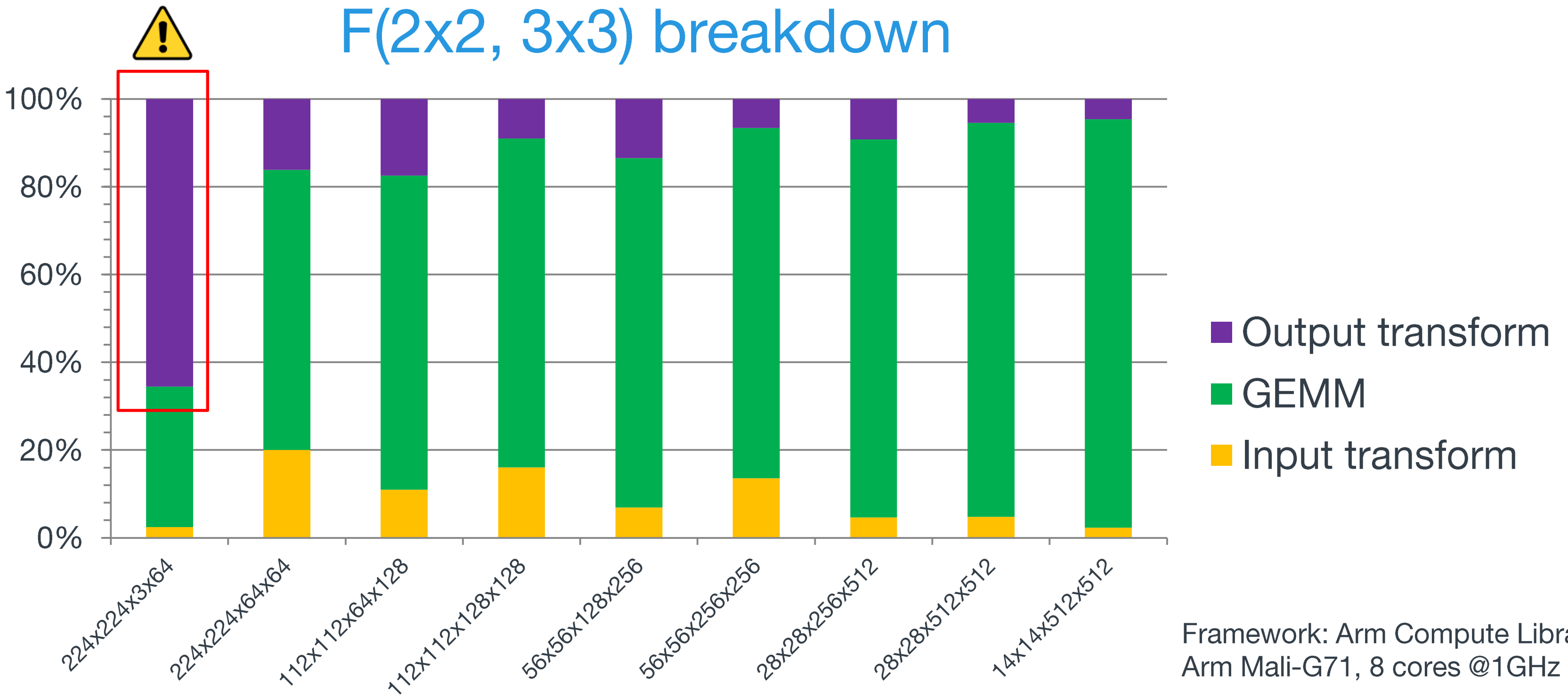
- Moreover on the GPU we can use a single OpenCL kernel to run the 16 GEMMs in parallel
  - Driver overhead reduction
  - Memory transfer reduction
  - Improvement of GPU utilization (experimented 14% improvement)

VGG16 Convolution Layers Breakdown (CPU)



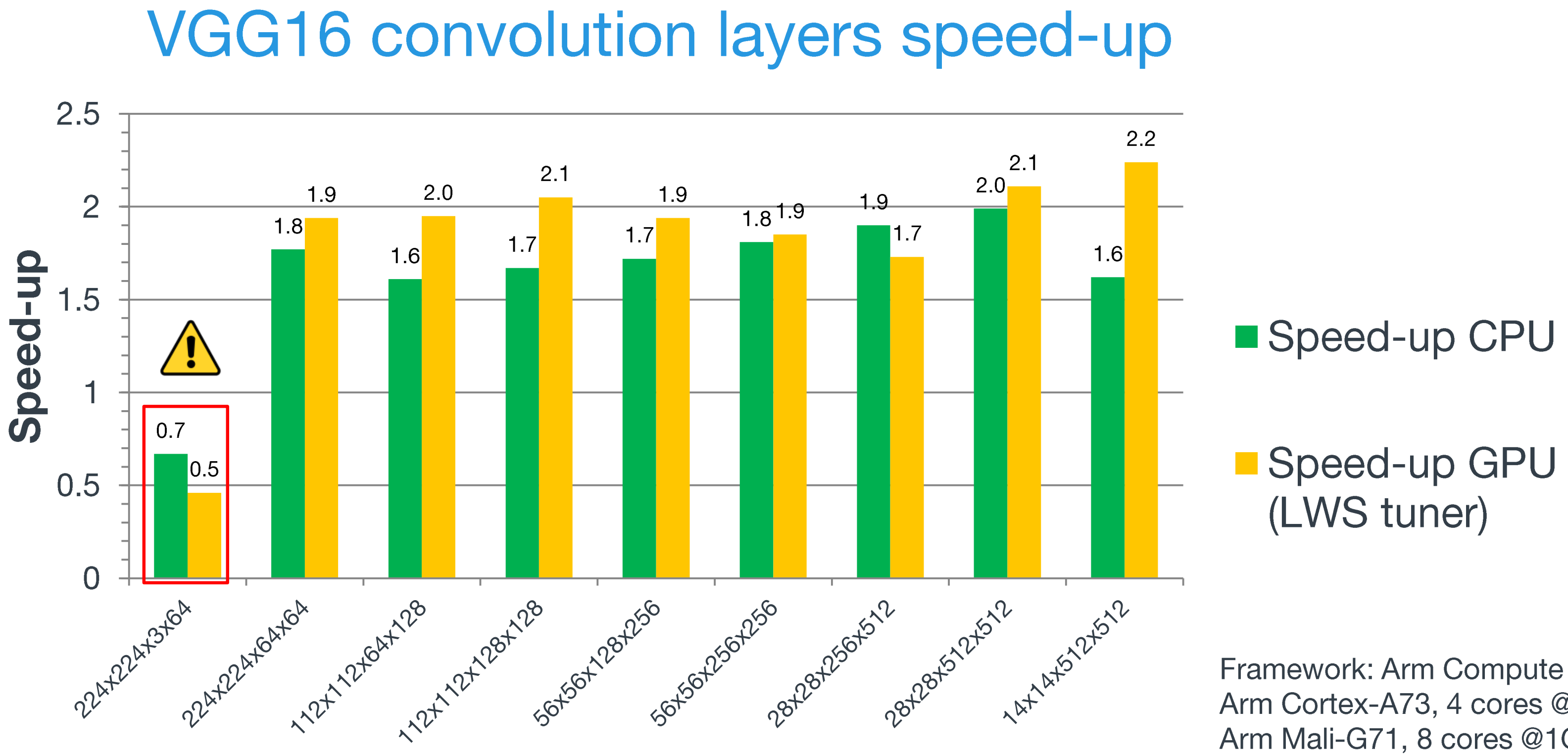
Framework: Arm Compute Library
Arm Cortex-A73, 4 cores @2.36GHz

VGG16 Convolution Layers Breakdown (GPU)



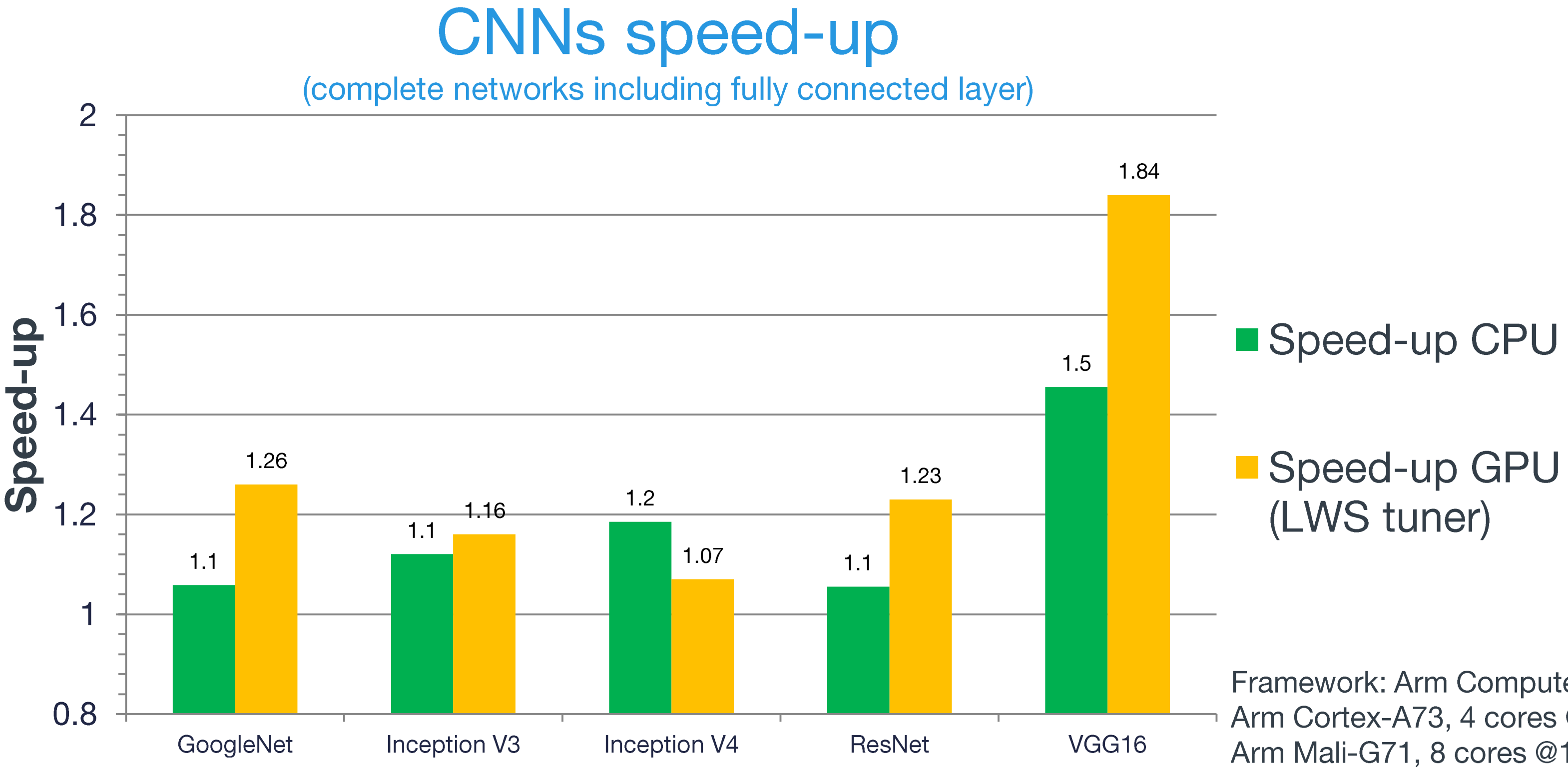
Framework: Arm Compute Library
Arm Mali-G71, 8 cores @1GHz

GEMM-based vs Winograd-based Convolution (1)



Framework: Arm Compute Library
Arm Cortex-A73, 4 cores @2.36GHz
Arm Mali-G71, 8 cores @1GHz

GEMM-based vs Winograd-based Convolution (2)



Framework: Arm Compute Library
Arm Cortex-A73, 4 cores @2.36GHz
Arm Mali-G71, 8 cores @1GHz

Extending Winograd-based Convolution: F(4x4,3x3)

The algorithm can be extended to even bigger output tile

i.e. F(4x4, 3x3)

4x multiplication complexity reduction!

Input transform matrix

$$\begin{pmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \end{pmatrix}$$

Filter transform matrix

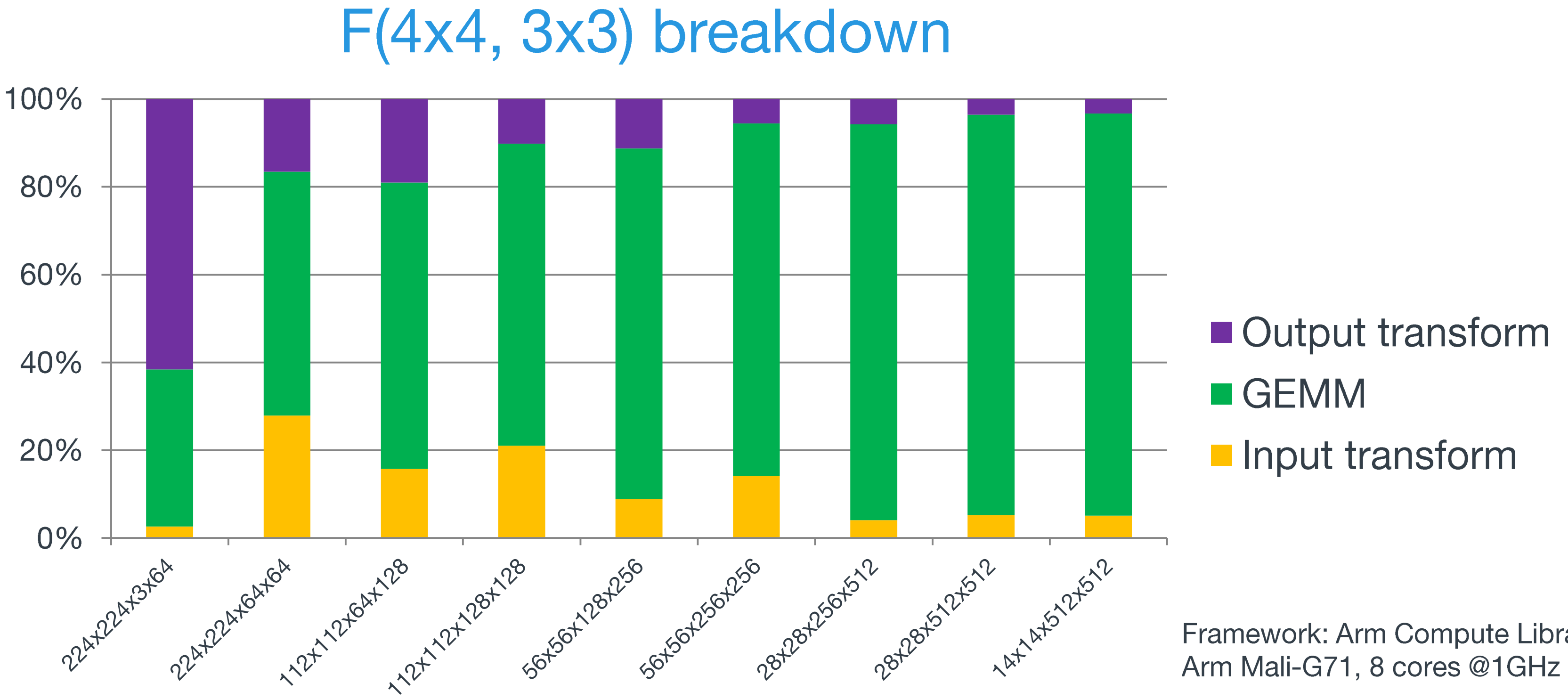
$$\begin{pmatrix} 1/4 & 0 & 0 \\ -1/6 & -1/6 & -1/6 \\ -1/6 & 1/6 & -1/6 \\ 1/24 & 1/12 & 1/6 \\ 1/24 & -1/12 & 1/6 \\ 0 & 0 & 1 \end{pmatrix}$$

Output transform matrix

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{pmatrix}$$

VGG16 Convolution Layers Breakdown (GPU)

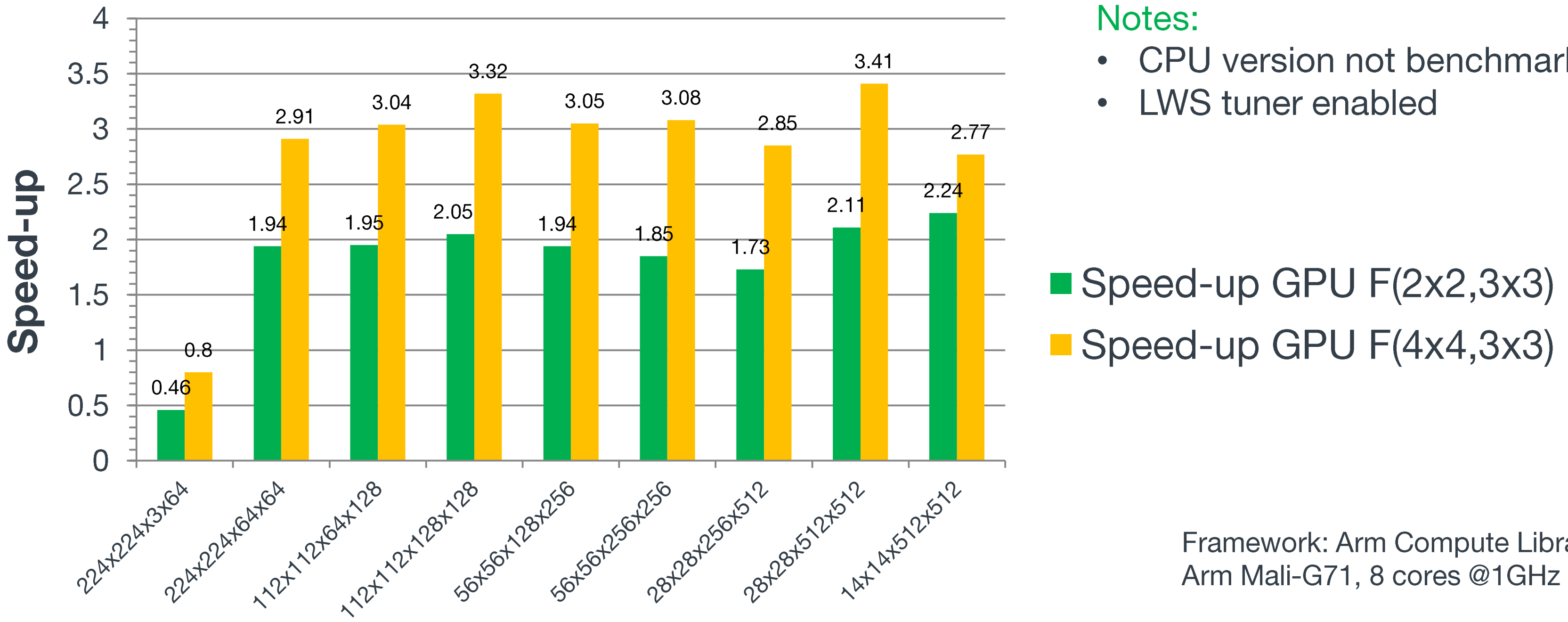
embedded VISION SUMMIT 2018



Framework: Arm Compute Library
Arm Mali-G71, 8 cores @1GHz

GEMM-based vs Winograd-based Convolution (1)

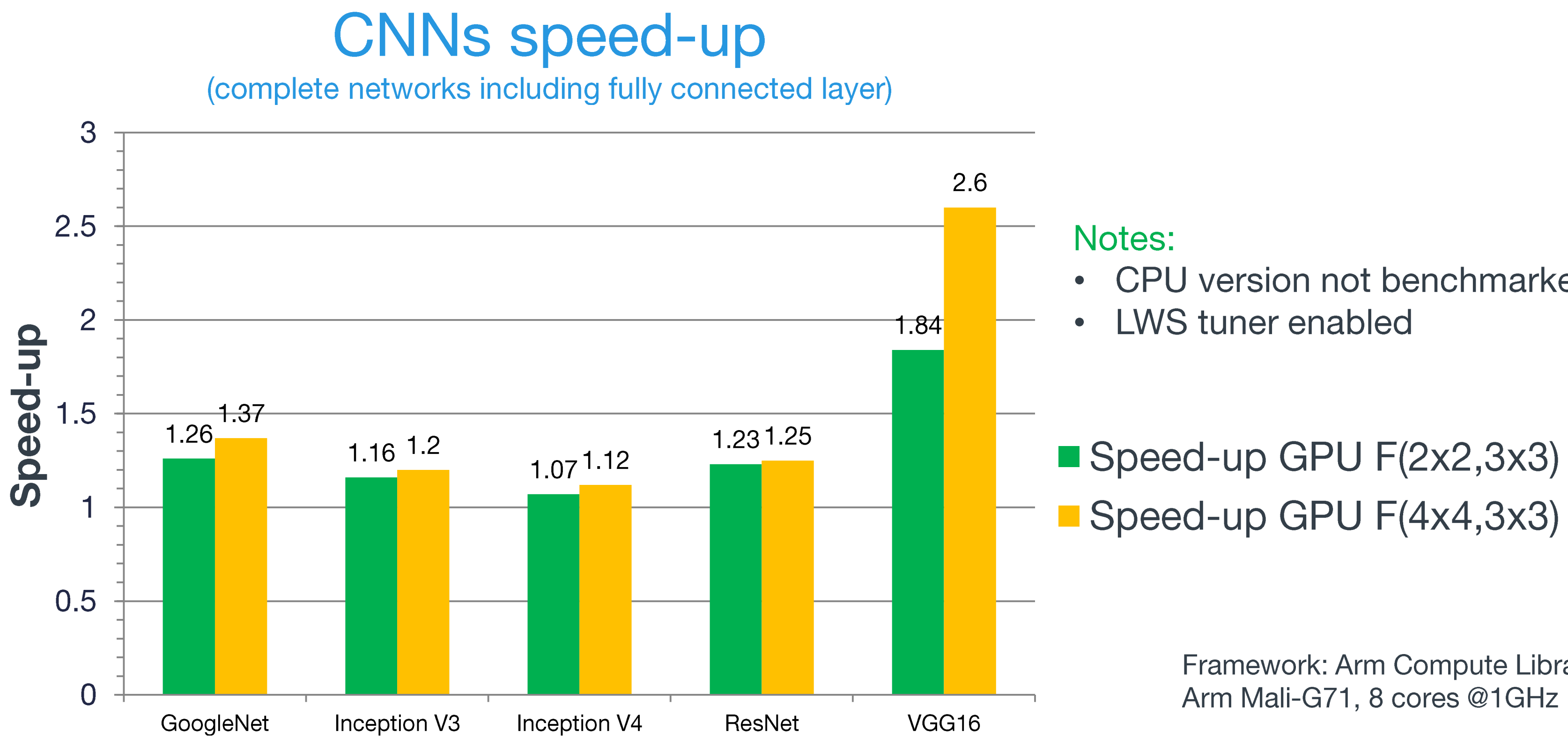
VGG16 convolution layers speed-up



Notes:
- CPU version not benchmarked
- LWS tuner enabled

Framework: Arm Compute Library
Arm Mali-G71, 8 cores @1GHz

GEMM-based vs Winograd-based Convolution (2)



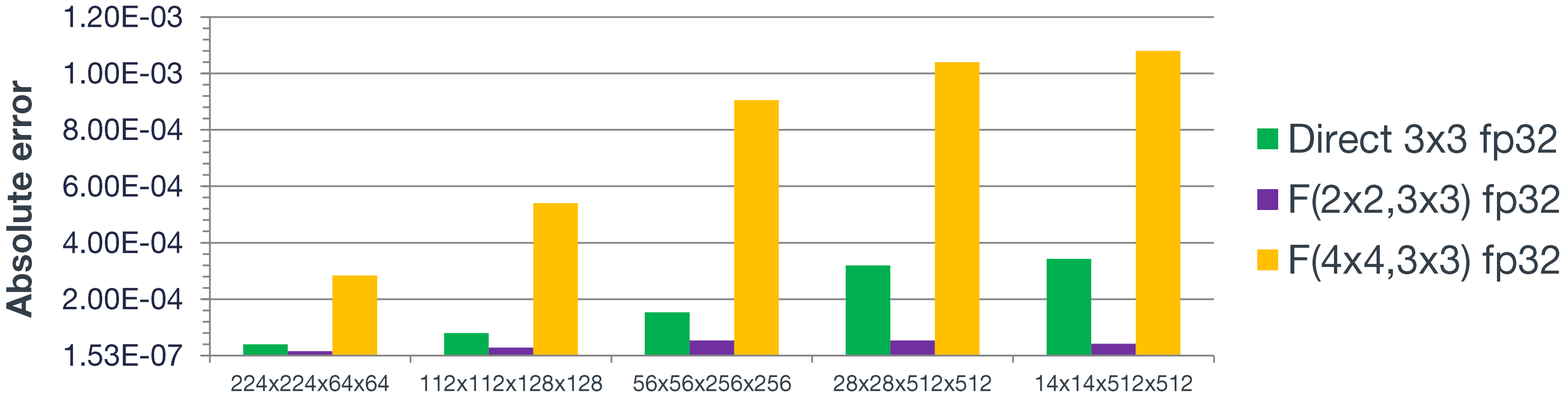
Notes:
- CPU version not benchmarked
- LWS tuner enabled

Framework: Arm Compute Library
Arm Mali-G71, 8 cores @1GHz

Accuracy: Absolute Error

The accuracy has been evaluated on VGG16 convolution layers*

- Reference implementation: fp64 (double precision) direct convolution
- Tensors uniform filled with random values [-1, 1]



F(2x2, 3x3) is more accurate than direct convolution fp32!

*Fast algorithm for Convolutional Neural Networks, Andrew Lavin, Scott Gray

Accuracy: ILSVRC2012

ILSVRC2012: ImageNet Large Scale Visual Recognition Challenge

Re-evaluated the accuracy on ILSVRC2012 validation data

- 50000 images centre cropped (256x256)
- Tested GoogleNet, VGG16 and ResNet50

The accuracy does not change with F(2x2, 3x3) and F(4x4,3x3)!

Current Investigations

- Understanding the impact of larger output and kernel sizes on network accuracy
  - i.e. F(6x6, 3x3), F(4x4, 5x5)…
- Winograd with limited numerical precision
  - Experimenting mixed-precision fp32/fp16 (fp32 accumulators, fp16 data?)
  - Int8 quantized seems requiring int32 accumulators…

Conclusion

- Winograd represents an incredible opportunity to improve further the performance of our CNNs
    - Over **2.5x** on VGG16 with F(4x4, 3x3) without deteriorating the accuracy

- GEMM is still the heart of convolutional layer also with Winograd fast algorithms
    - The optimization effort is limited to input and output transforms

- Winograd is a suitable algorithm for both CPU and GPU architectures

References

- "Fast algorithm for Convolutional Neural Networks", 2015
  - Andrew Lavin, Scott Gray
- Winograd-based convolution available in Arm Compute Library
  - https://github.com/ARM-software/ComputeLibrary
- ArmNN
  - https://github.com/ARM-software/armnn

Thank you!