Optimizing Highly Memory Bound Data Transfer for Fast Convolutional Neuro Network

Yolanda Chen, Intel Developer Products Division

Background

- Winograd's mini filter algorithm[1] is used as fast algorithm in deep convolutional neuro networks.
- Requires scattered data access patterns which induced inefficient memory accesses and vectorization.
- Colfax published **FALCON[2]** library to optimize this algorithm and measures the performance on Intel Xeon Phi processors
- The **input and output data transformations** spent more than **15% - 50%** of execution time across the 13 layers of the VGG[3] Net configuration.

Winograd's Minimal FIR Filter (1)

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

Where

$$m_1 = (d_0 - d_2)g_0 \qquad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$
$$m_4 = (d_1 - d_3)g_2 \qquad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

Multiplications reduced: 6 -> 4 (2*3 -> 2+3-1)

Additional operations over Filter: 3 adds, 2 multiplications (Precomputed)

Additional operations over Input data: 4 adds (Precomputed)

Winograd's Minimal FIR Filter (2)

F(m, r)

$$Y = A^T (Gg) \odot (B^T d)$$

where $\odot$ indicates element-wise multiplication.

F(m*m, r*r)

$$Y = A^T[GgG^T] \odot [B^T dB]A$$

Multiplications reduced: m*m*r*r -> (m+r-1)*(m+r-1)

For F(2*2, 3*3), it's 36 -> 16, **2.5x**

When m=2, r=3

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

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

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

$$g = \begin{bmatrix} g_0 & g_1 & g_2 \end{bmatrix}^T$$

$$d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 \end{bmatrix}^T$$

Formula for Convnet Layer

Image i, Filter k, Transferred Tile t, Channel c

$$Y_{t,f} \equiv \sum Y_{n,t,f,c}$$

$$= \sum_{c} A^T [\, U_{n,t,c} \odot V_{c,f} \,] A$$

$$= A^T [\, \sum_{c} U_{n,t,c} \odot V_{c,f} \,] A$$

Filter Transfer

$$U = GgG^T$$

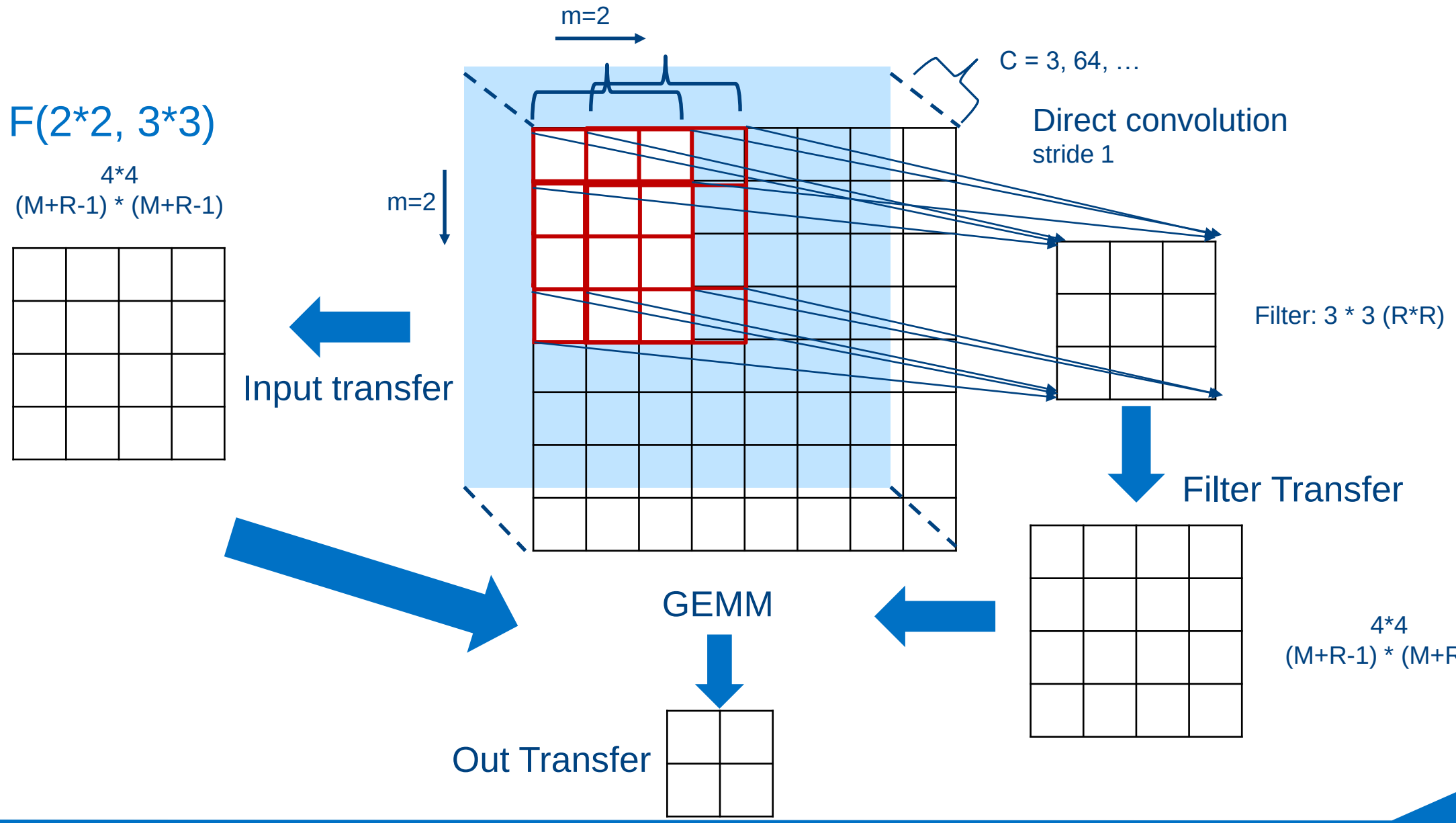
Input Data Transfer

$$V = B^T dB$$

GEMM

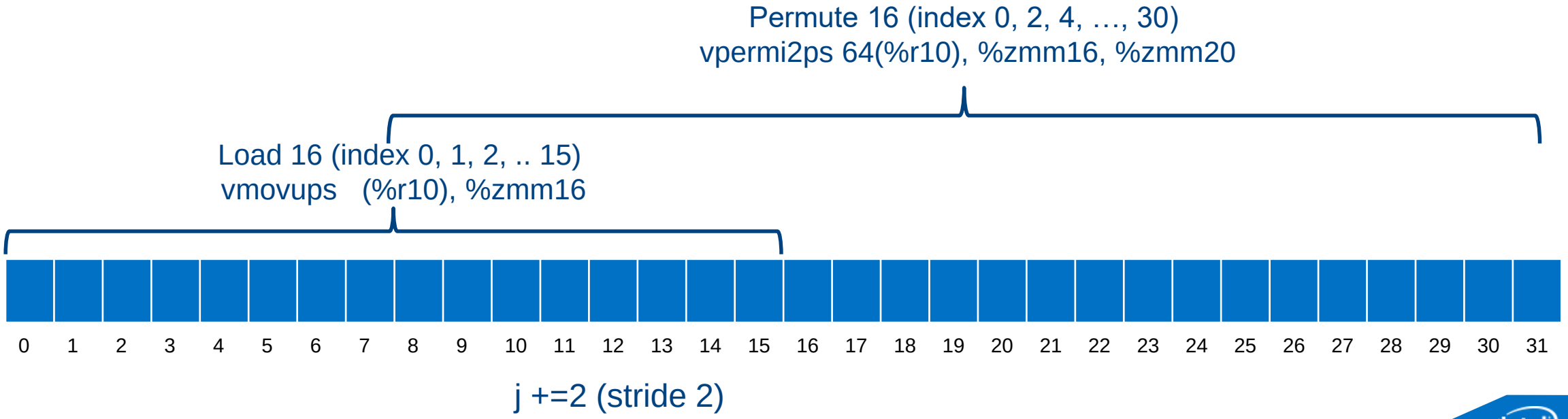
Output / Inverse Transfer

Fast convolution based on Winograd's minimal filter



AVX512 Vectorization for Input Transfer

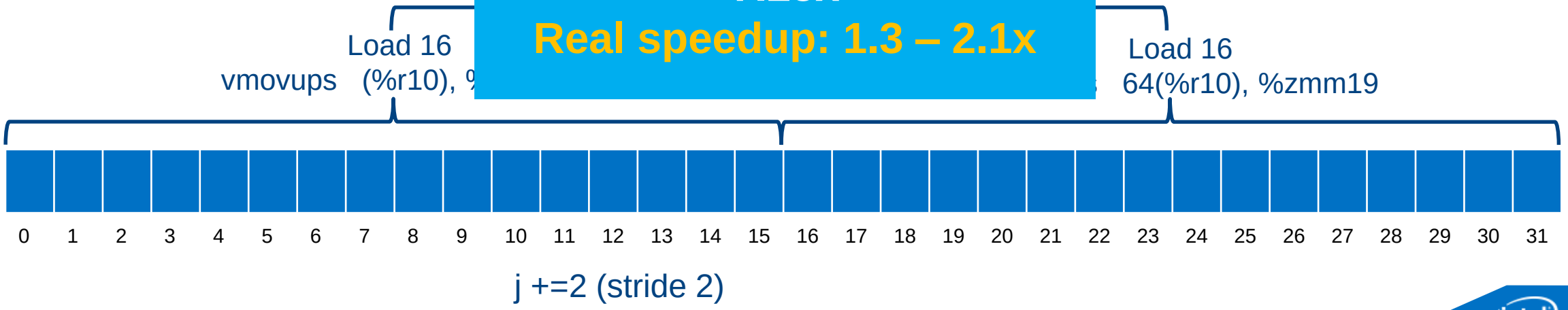
```
#pragma omp simd linear(j:2, count:1) private(temp)
for (j = 0; j < W; j += 2, count++)
{
    //1D: BT*d
    for (t = 0; t<4; t++)
    {
        temp[t * 4 + 0] = data[t*irows + j] - data[t*irows + j + 2];        // d0 - d2
        temp[t * 4 + 1] = data[t*irows + j + 1] + data[t*irows + j + 2];      // d1 + d2
        temp[t * 4 + 2] = data[t*irows + j + 2] - data[t*irows + j + 1];      // d2 - d1
        temp[t * 4 + 3] = data[t*irows + j + 1] - data[t*irows + j + 3];      // d1 - d3
    }
```



AVX512 Vectorization for Input Transfer

```
#pragma omp simd linear(j:2, count:1) private(temp)
for (j = 0; j < W; j += 2, count++)
{
    //1D: BT*d
    for (t = 0; t<4; t++)
    {
        temp[t * 4 + 0] = data[t*irows + j] - data[t*irows + j + 2];       // d0 - d2
        temp[t * 4 + 1] = data[t*irows + j + 1] + data[t*irows + j + 2];     // d1 + d2
        temp[t * 4 + 2] = data[t*irows + j + 2] - data[t*irows + j + 1];     // d2 - d1
        temp[t * 4 + 3] = data[t*irows + j + 1] - data[t*irows + j + 3];     // d1 - d3
    }
```

Estimated Speedup
(Introduction cost reduction)
7.25x
Real speedup: 1.3 – 2.1x

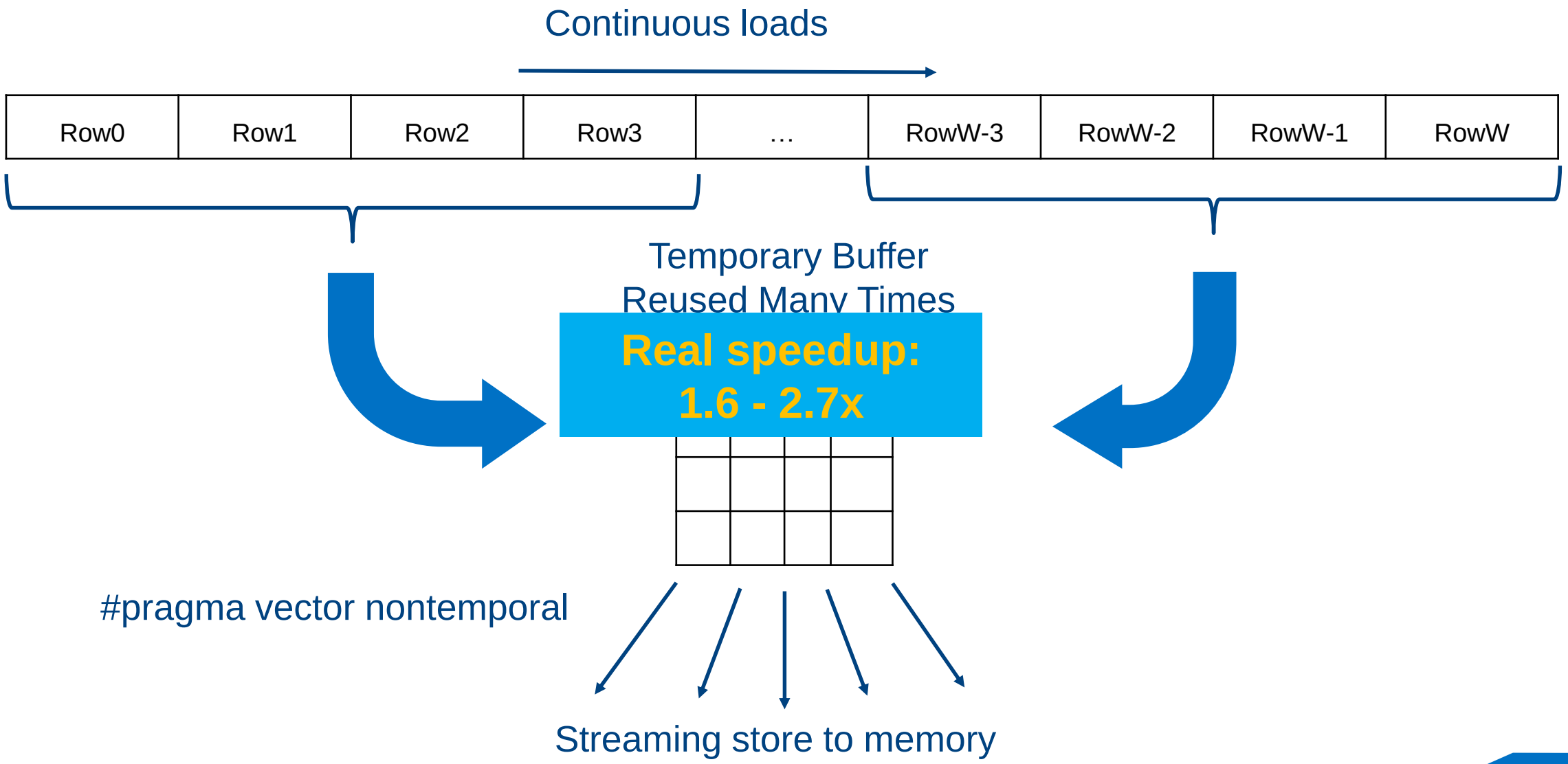


Memory latency Withholds Vectorization Speed Up

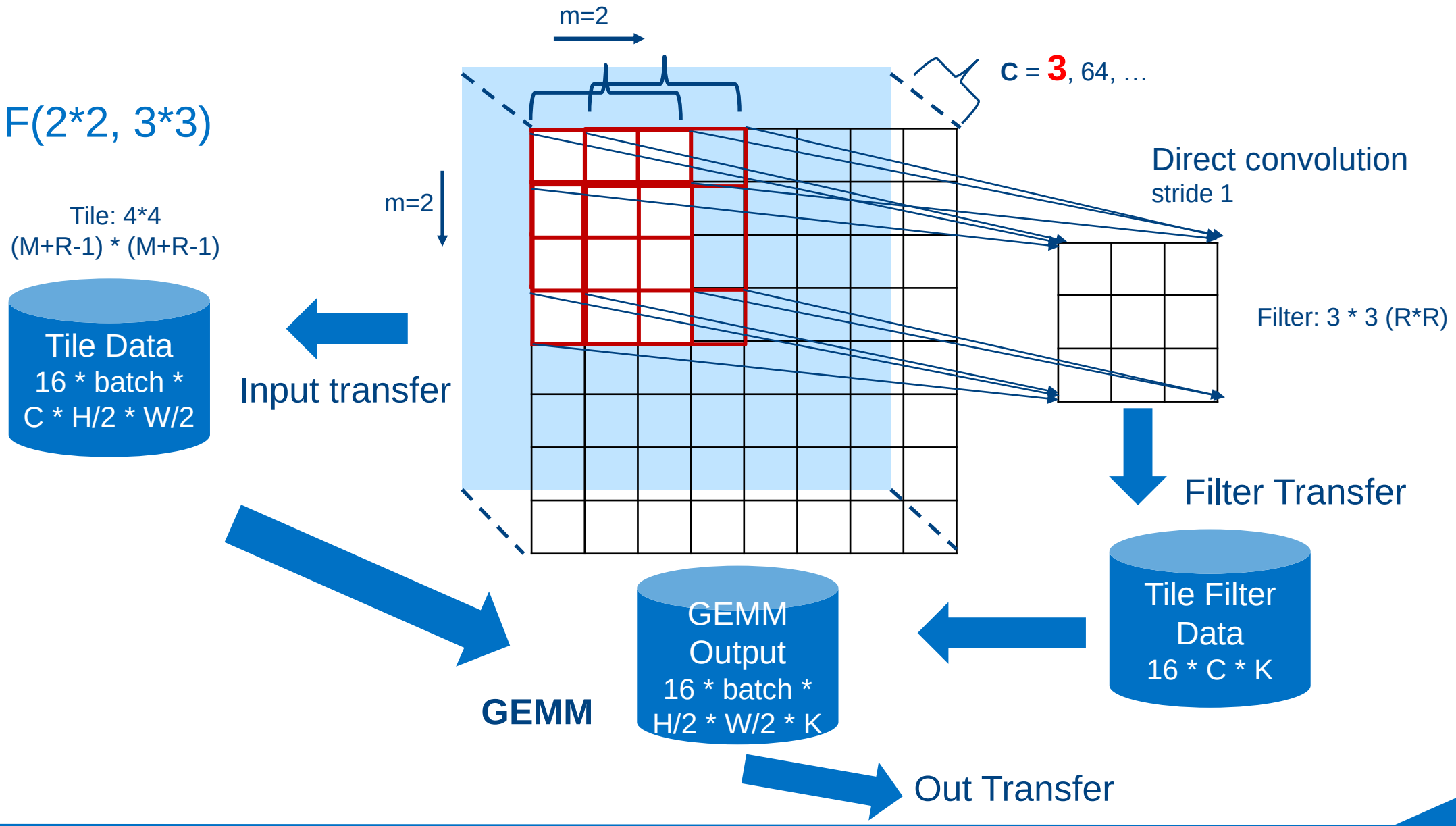
Data transfer for 1 AVX512 vector iteration

#Instructions	Scalar	AVX2 (256-bit)	AVX-512 (512-bit)
Loads (hit) to input data	4 * 61	4 * 5	4 * 1
Loads (miss) to input data	4 * 3	4 * 3	4 * 3
SP adds	16 * 32	2 * 32	32
Stores (hit) to output data	16 * 15	16	0
Stores (miss) to output data	16	16	16
Total (Reduction)	**1024 (x1)**	**128 (x8)**	**64 (x16)**

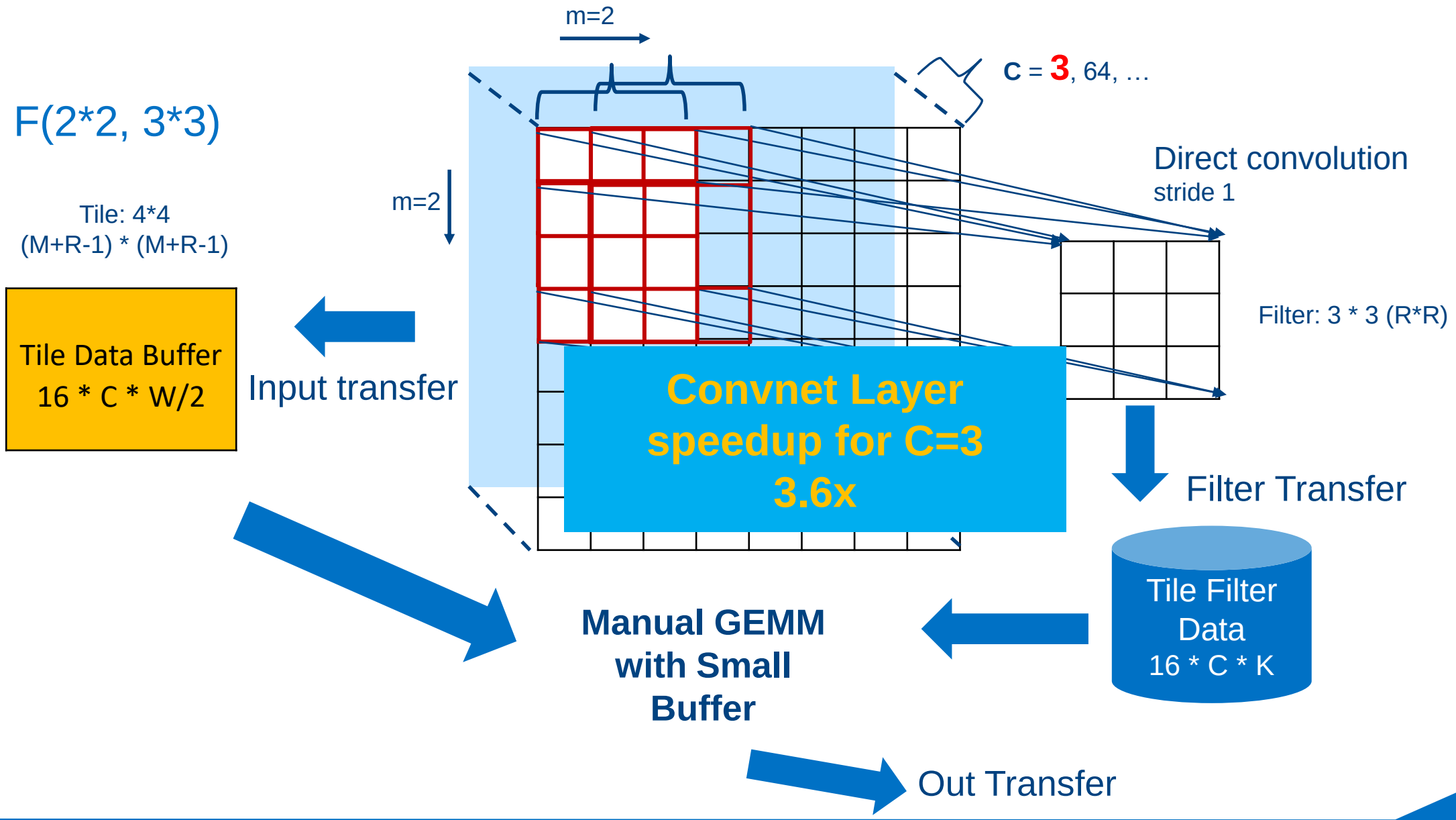
Continuous Loads and Streaming Stores



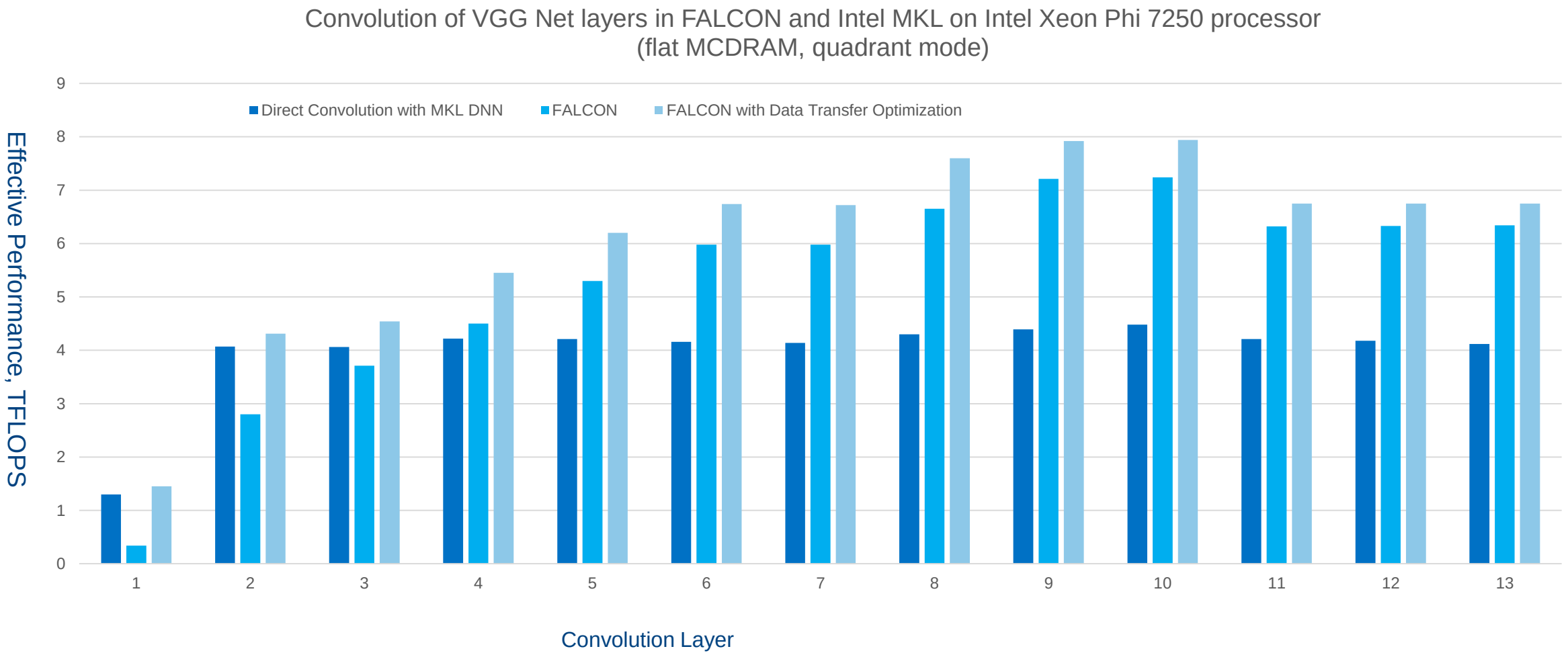
Inefficient GEMM for Small Channel Size



Inefficient GEMM for Small Channel Size



Experiment Result



Summary and Future Work

- Intel compiler vectorizes loops with non-unit strides and adjacent indices efficiently.
- Memory latency restrains potential vectorization speedup for long vectors
- Streaming loads and stores when data won’t be reused.
- To maximize data reuse may require fusion or refine data transfer flow to reduce intermediate storage.

Thanks to

Rakesh Krishnaiyer from Intel Compiler Team

References

1. Fast convolution: https://arxiv.org/abs/1509.09308

2. FALCON library: https://colfaxresearch.com/falcon-library/

3.VGG Net: https://arxiv.org/abs/1409.1556

Q&A

Backup

Large batch sizes adversely affect convergence of the network, so the minimum batch size that can be computed efficiently places an upper limit on cluster size. State of the art convnet architectures for image recognition use deep networks of 3 × 3 convolutional layers, because they achieve better accuracy with fewer weights than shallow networks with larger filters.

Therefore there is a strong need for fast convnet algorithms for small batch sizes and small filters.[1]

Legal Disclaimer & Optimization Notice

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804