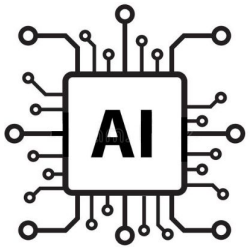
Optimizing Winograd Convolution on Manycore Processor



Student: Ken Yu (游家權)
Student ID: R10942152

Survey Papers:

- [1] **Optimizing Massively Parallel Winograd Convolution on ARM Processor**, Dongsheng Li, Dan Huang, Zhiguang Chen, Yutong Lu
  - Hardware: ARM processor

- [2] **Optimizing N-dimensional Winograd-based Convolution for Manycore CPUs**, Zhen Jia, Aleksandar Zlateski, Frédo Durand, Kai Li
  - Hardware: Intel processor

- [3] **Optimizing Winograd-Based Convolution with Tensor Cores,** Junhong Liu, Dongxu Yang and Junjie Lai
  - Hardware: NVIDIA GPU

Outline

- Background introduction
  - Winograd-based Convolution
  - ARM Manycore Processor
- Motivation
  - Work before this paper: NNPACK’s Bottleneck
- Technique used
  - Unrolling Loop
  - Block strategy
  - Reorder Data Layout
  - NUMA scheduler
- Experiment Result
- Comparision between three papers
- Conclusion

Background - 1D Winograd-based Convolution

- Winograd convolution[4] is a fast algorithm for convolution that can greatly reduce the number of multiplications by eliminating redundency.

Input Signal $d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T$

Convolution Kernel $g = [g_0 \quad g_1 \quad g_2]^T$,

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

Normal Convolution
6 multiplications

Winograd Convolution
4 multiplications

$$m_1 = (d_0 - d_2)\, g_0 \quad m_2 = (d_1 + d_2)\, \frac{g_0+g_1+g_2}{2}$$

$$m_4 = (d_1 - d_3)\, g_2 \quad m_3 = (d_2 - d_1)\, \frac{g_0-g_1+g_2}{2}$$

Matrix Form

Element-wise Multiplication

$$Y = A^T \left[(Gg) \odot (B^T d) \right]$$

Input Transformation

Output Transformation

where

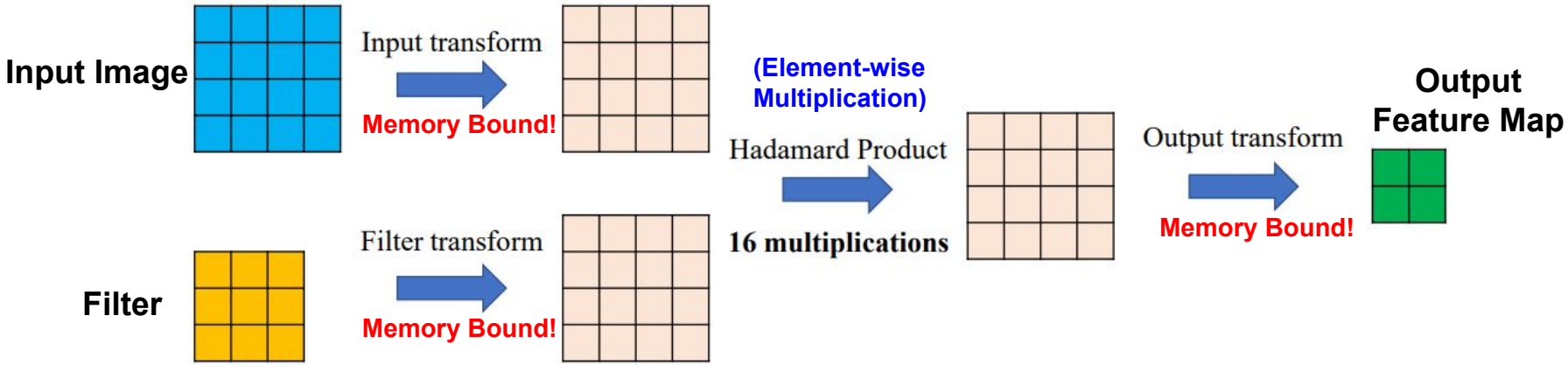
$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

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

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

Background - 2D Winograd-based Convolution

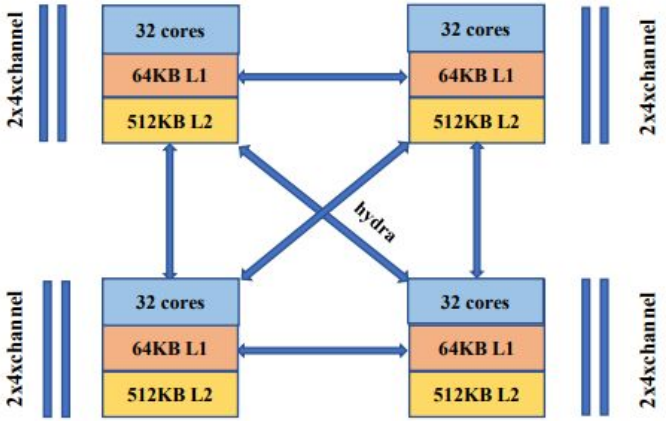
- Winograd convolution involves three steps: **input transformation**, **element-wise multiplication**, and **output transformation**.
- However, Winograd's input and output transformation requires massive memory access, which leads to the program bound by memory I/O instead of CPU computation.
- Normal convolution needs 36 multiplications, while Winogard's only needs 16.



Background - Manycore Processor

- Compare to conventional processor, manycore processor has **more cores** and every core has **lower frequency**, **simpler hardware** and **better energy efficiency**.
- Good at processing data in batch and execute in parallel; manycore processor usually adopt NUMA(Non-Uniform Memory Access) scheme.
- Take **ARM kunpeng920** processor for example, it has **128 cores** with frequency of 2.6Ghz and 4 NUMA nodes and on RISC architechture.

ARM KunPeng920 architechture



ARM KunPeng920 Spec.

Core	
number	128
name	TSv110
frequency	2.6GHZ
die count	4

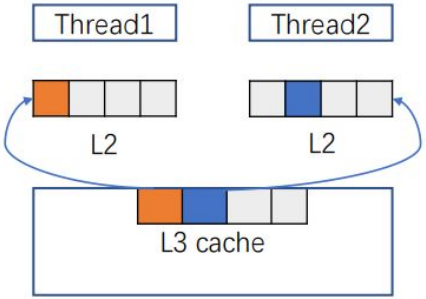
Memory	
Type	DDR4
channel	8
NUMA node	4

Cache	
L1 I	64KB x 128
L1 D	64KB x 128
L2	512KB x 128
L3	16MB x 4
L2 cache line	64B
L3 cache line	128B

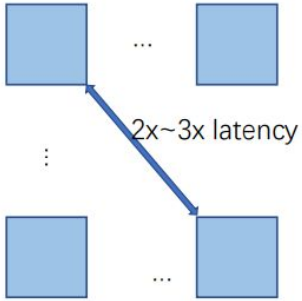
Features	
SIMD	neon

Motivation - NNPACK's Bottleneck

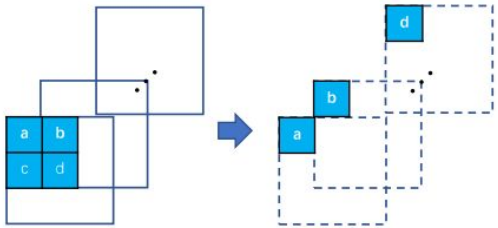
- **False sharing**: multiple threads store data on the same cache line, causing non-share data be treated like shared one and wasting time on syncronizing.
- **Remote memory access** is slow in NUMA architecture. It's 2~3 time slower than than accessing local memory.
- **Non-sequential memory access** will lead to huge number of cache misses since data are scatter in different place of memory and it's harder for cache to exploit locality.



Cache contention



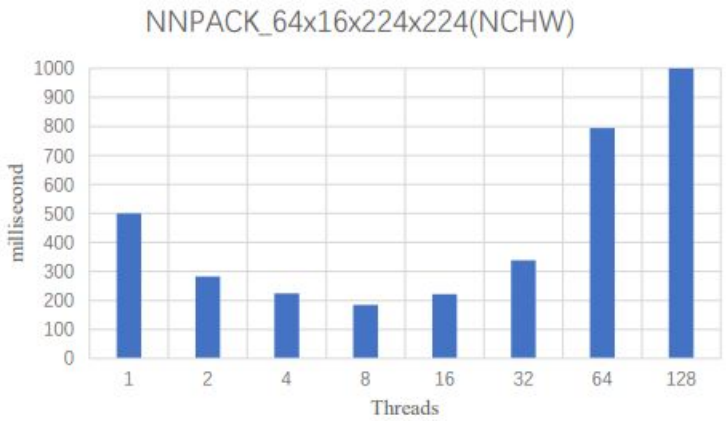
Remote access and memory congestion



Non-sequential memory access

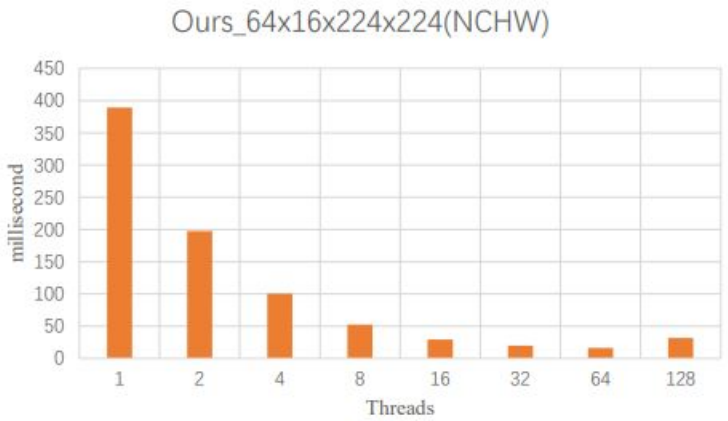
Motivation - Bottleneck Result

- NNPACK's performance stops scaling up when using more than 8 threads due to the memory accessing bottleneck.



NNPACK

VS.



Ours

Method - Overview

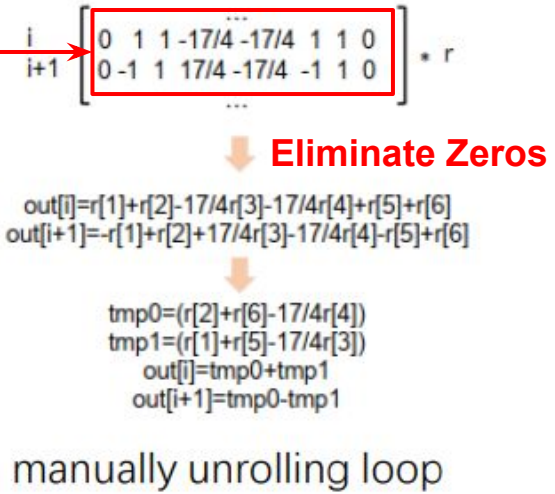
- Reduce non-sequential memory access and memory copy operations
  - Reorder data layout in transformation stage
- Achieve high hardware utilization
  - Choose data layout according to the shape of input tensor
  - Utilize GEMM routine and optimize computation for reordered layout
- Use divide-and-conquer algorithm to construct data parallelism
  - Provide a disjoint aligned buffer for each core
  - NUMA-scheduler to assign thread to corresponding NUMA nodes

Input and Filter Transformation - Loop Unrolling

- Eliminate unnecessary computation caused by redundant entries, such as zero or duplicant values in constant matrix.
- For example, the matrix shown below has lots of zeros and duplicant rows with different signs that can be eliminated by unrolling the loop

Constant Matrix when F(4x4, 3x3)

$$B^T = \begin{bmatrix} 1 & 0 & -21/4 & 0 & 21/4 & 0 & -1 & 0 \\ 0 & 1 & 1 & -17/4 & -17/4 & 1 & 1 & 0 \\ 0 & -1 & 1 & 17/4 & -17/4 & -1 & 1 & 0 \\ 0 & 1/2 & 1/4 & -5/2 & -5/4 & 2 & 1 & 0 \\ 0 & -1/2 & 1/4 & 5/2 & -5/4 & -2 & 1 & 0 \\ 0 & 2 & 4 & -5/2 & -5 & 1/2 & 1 & 0 \\ 0 & -2 & 4 & 5/2 & -5 & -1/2 & 1 & 0 \\ 0 & -1 & 0 & 21/4 & 0 & -21/4 & 0 & 1 \end{bmatrix}$$

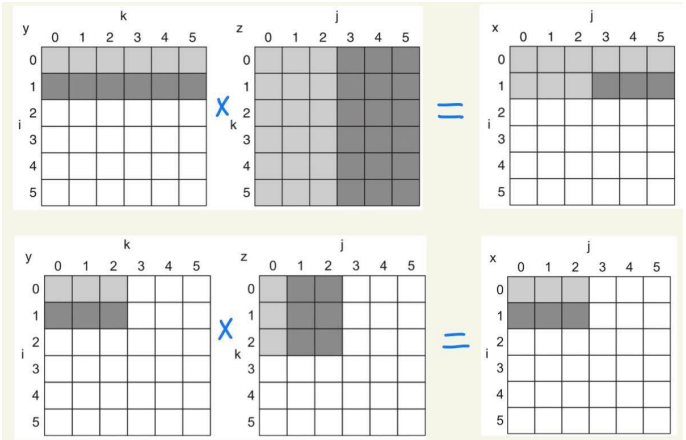


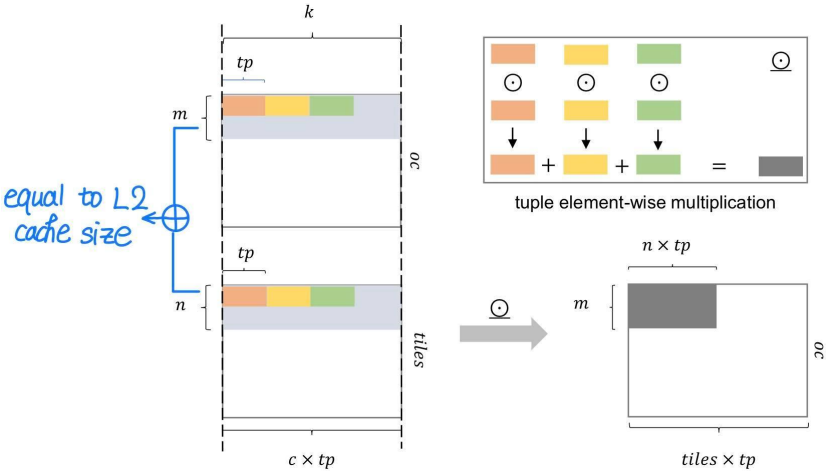
Input and Filter Transformation - Blocking Strategy

- A common technique for accelerating matrix multiplication
- When multiplying matrices, calculate a block of output rather than a single entry can make good use of the cache data.
- Tune the block size to make input and filter matrices **fit in L2 cache**; by this way, the cache miss can be greatly reduced.

Direct Matrix Multiplication

Blocking Strategy



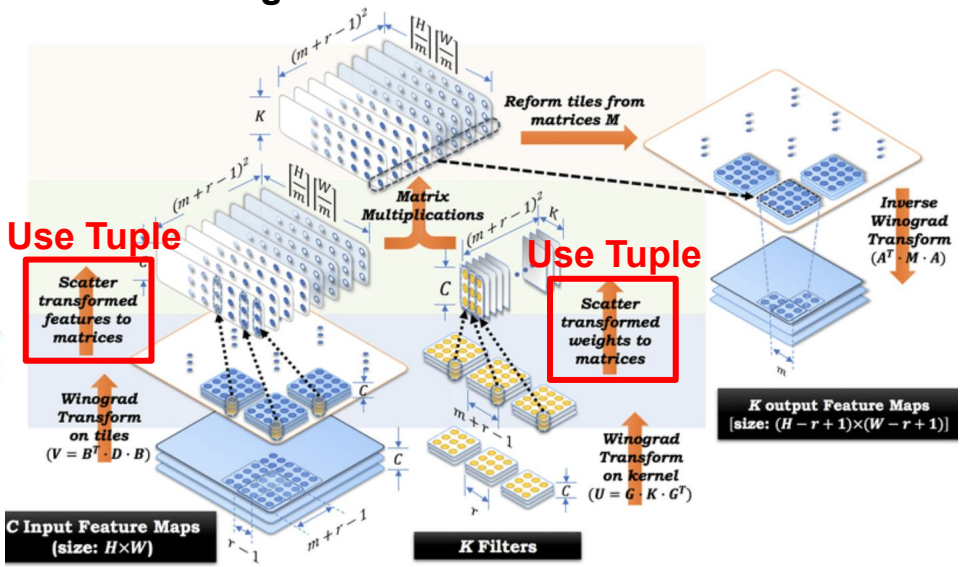


Reorder Data Layout - GEMM Winograd-based Convolution

- GEMM Winograd-based convolution replaces element-wise multiplication to matrix multiplication and collect features from different tiles to achieve higher computation efficiency.
- But GEMM can led to great cache miss because scatter data are located in non–sequential memory; TEWMM sloves it by storing scatter data in **tuple**, allowing these data being loaded in the same cache.
- GEMM performs well on small image because all title can be put in caches, while TEWMM perform better on big image since the cache miss time saved by tuple is prominent.
- Since these two algorithms can complement each other, this paper devise a **heuristic algorithm** to decide which data layout to use by calculating arithmetic intensity in advance.

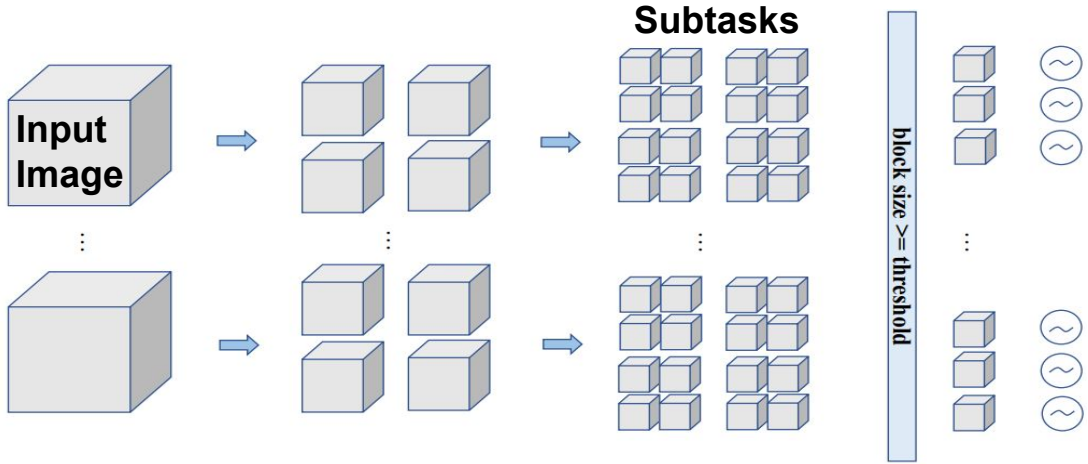
Performance for reordered and GEMM

Winograd-base Convolution Flowchart



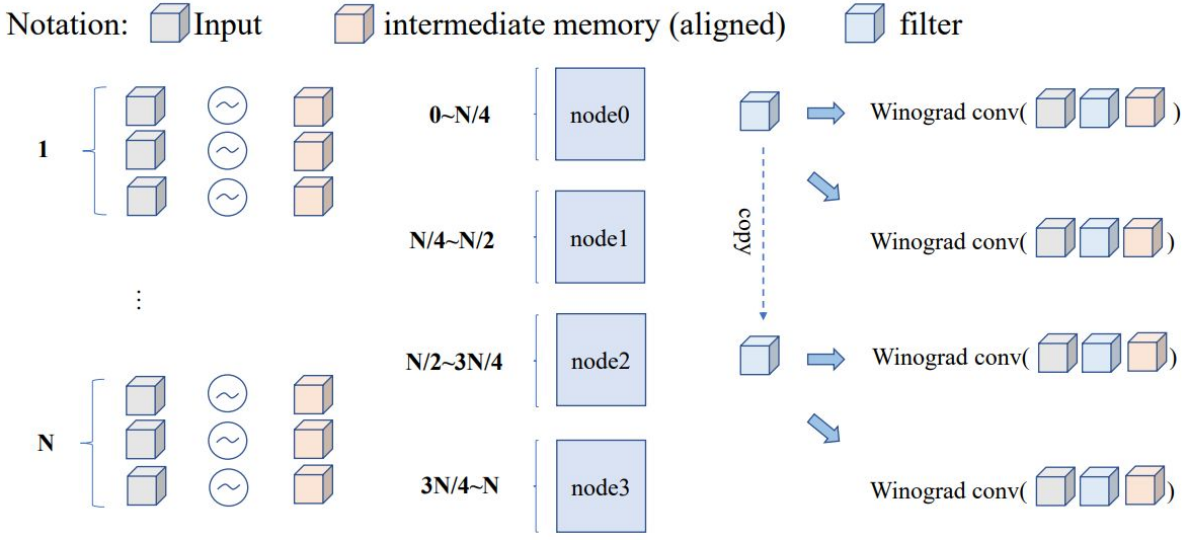
NUMA-aware Schedueling

- Divide input batches to many subtasks and send them into manycore processor evenly.
- **Nearest allocation policy**: subtasks that are derived from the same area of input image should be assigned to the same NUMA node to avoid remote memory access.
- Align buffer and data in cache line to prevent false sharing



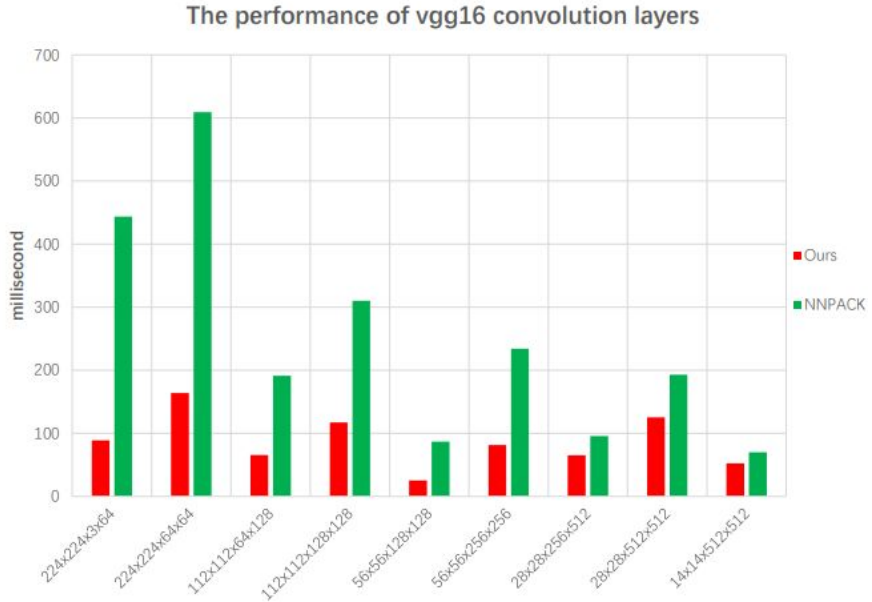
NUMA-aware Schedueling

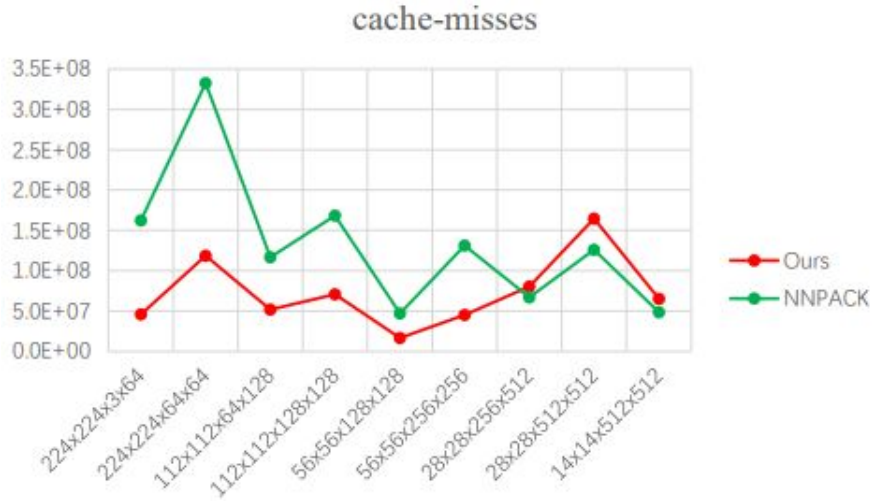
- Every subtask execute independently and store intermediate result(i.e., transformed matrix) in **buffer memory**.
- Since filter are usually very small(e.g., 3x3), it copy filters to every NUMA node to further prevent remote memory access.



Experiment Result

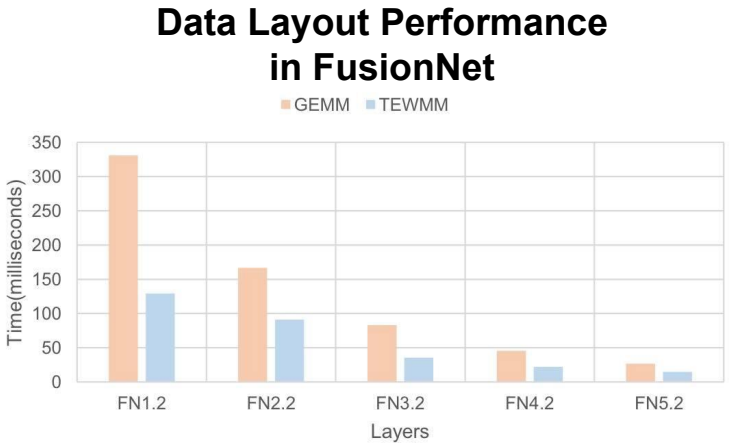
- This paper[1] achieves 1.5~3 speedup compare to NNPACK, especially at large image size layer because of its improved data layout.
- The cache misses rate is significant lower due to the improvements.





Experiment Result

- TEWMM reduces the transformation overhead by 60%.
- Although it reduces execute time in most network, when batch size is small the speedup is smaller (e.g FN has batch size of one) because there isn't enough input data to exploit data-level parallelism



16 Cores Speedup

	Ours	NWM	Im2col	FFT
VN1.2	13.6	7.0	6.9	9.7
VN2.2	14.6	7.3	6.5	10.7
VN3.2	14.3	8.7	6.3	12.3
VN4.2	13.5	10.2	6.5	12.9
VN5.2	9.9	9.6	11.9	12.1
FN1.2	6.3	7.0	7.4	9.9
FN2.2	6.5	8.9	10.7	11.9
FN3.2	6.0	11.1	9.9	13.4
FN4.2	7.5	12.6	10.8	14.1
FN5.2	7.9	13.3	12.2	14.0
RN2	13.9	6.4	6.4	9.6
RN3	14.2	9.4	6.5	12.2
RN4	12.2	10.5	6.8	12.7
RN5	9.6	10.8	12.8	13.3

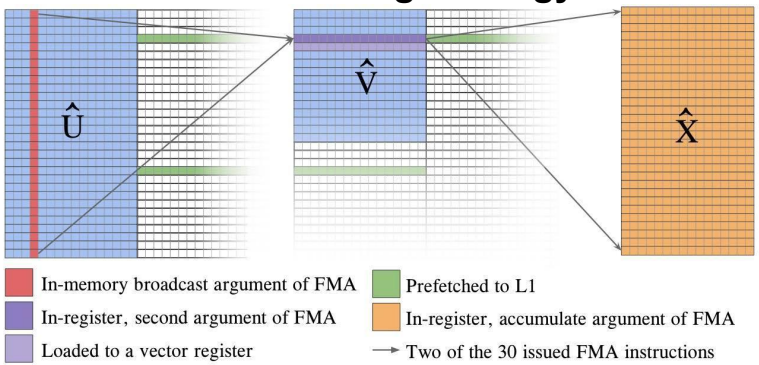
16
8
1

- VN = Vgg Net
- FN = Fusion Net
- RN = Residual Net

Comparison: Optimizing N-dimensional Winograd-based Convolution for manycore CPUs [2]

- Similar techniques used in [2]
  - Utilize unrolling loop and blocking to boost the multiplication speed
  - Data layout vectorization to lower TLB and cache miss rate
  - Use static scheduling to distribute workload evenly
- New techniques
  - Omit kernel transformation when network is inferencing
  - Use a **JIT compiler** to optimize memory management by **pre-fetching**
  - Synchronize threads by busywaiting **barriers**

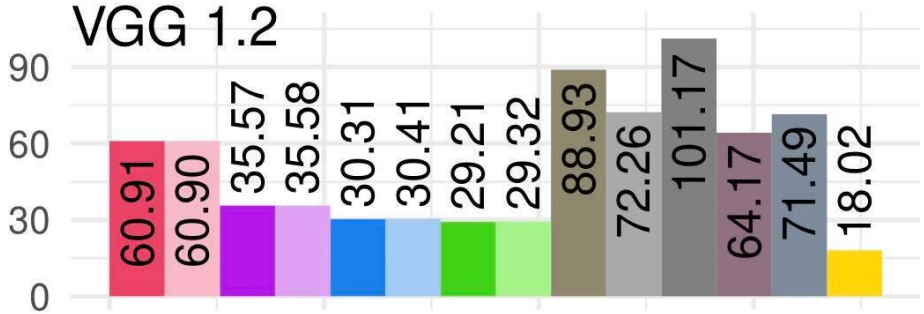
Pre-fetching Strategy



Comparison: [2] Experiment Result

- When running the first layer of VGG, [1] needs 90 ms, while [2] only needs 60 ms. I think it's because Intel core is more sophisticated than AMR's.
- The inference optimization is not saving any time.

Experiment Result of [2]



Comparison: [3] Optimizing Winograd-Based Convolution with Tensor Cores

- Hardware: Nvidia Ampere A100 80GB
  - FP16 Tensor core is able to calculate 4x4 matrix multiplication directly
  - 432 Tensor cores
- Due to GPU's huge global memory, this work doesn't have to optimize memory and cache storage; instead, it focus on computational accuracy and bigger Winograd tile.
- Experiment result : In VGGNet first layer, [3] spend 1.3ms while [1] spend 90ms.

Experiment Result of [3]

Network	Layer No.	$Cin \times H \times W$	Cout	Ours		cuDNN Winograd		cuDNN GEMM convolution		Speedup	
				usec	TFLOPs	usec	TFLOPs	usec	TFLOPs	cuDNN Winograd	cuDNN GEMM
VGG	1	$128 \times 56 \times 56$	256	1319.88	179.38	1554.08	152.35	1562.79	151.50	1.17x	1.18x
	2	$256 \times 56 \times 56$	256	1812.67	261.23	2246.04	210.82	2667.40	177.52	1.24x	1.47x
	3	$256 \times 28 \times 28$	512	745.53	317.57	913.05	259.31	1287.63	183.87	1.22x	1.73x
	4	$512 \times 28 \times 28$	512	1085.47	436.24	1383.48	342.27	2619.22	180.79	1.27x	2.41x
	5	$512 \times 14 \times 14$	512	388.28	304.88	489.68	241.75	615.12	192.45	1.26x	1.58x
FusionNet	6	$256 \times 160 \times 160$	256	121.12	249.33	1902.62	15.87	214.03	141.10	15.71x	1.77x
	7	$512 \times 80 \times 80$	512	108.86	277.41	1146.47	26.34	227.63	132.67	10.53x	2.09x
	8	$1024 \times 40 \times 40$	1024	179.81	167.95	984.79	30.67	258.11	117.00	5.48x	1.44x

Conclusion

- Winograd convolution is a fast algorithm that greatly reduce number of multiplications.
- Hardware architecture can decide what kind of optimization techniques should be used.
- Cache miss, remote memory access are the culprit of bad performance in NUMA archietchrue, we should find method to cope with these issues.
- Pre-feteching, blocking, loop unrolling are good strategies for matrix multiplicati on parallelism.
- Tasks and data should be distributed evenly among all cores to maximize data-level parallelism.

Reference

[1] Optimizing Massively Parallel Winograd Convolution on ARM Processor, Dongsheng Li, Dan Huang, Zhiguang Chen, Yutong Lu, doi/abs/10.1145/3472456.3472496

[2] Optimizing N-dimensional Winograd-based Convolution for Manycore CPUs, Zhen Jia, Aleksandar Zlateski, Frédo Durand, Kai Li

[3] Optimizing Winograd-Based Convolution with Tensor Cores, Junhong Liu, Dongxu Yang and Junjie Lai

[4] Fast Algorithms for Convolutional Neural Networks, Andrew Lavin, Scott Gray, arxiv.org/abs/1509.09308