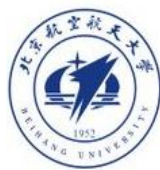
Extremely Low-bit Convolution Optimization for Quantized Neural Network on Modern Computer Architectures

Qingchang Han[1,2], Yongmin Hu[1], Fengwei Yu[2], Hailong Yang[1], Bing Liu[2], Peng Hu[1,2], Ruihao Gong[1,2], Yanfei Wang[2], Rui Wang[1], Zhongzhi Luan[1], Depei Qian[1]

School of Computer Science and Engineering
Beihang University[1], Beijing, China
SenseTime Research[2]



北京航空航天大學
BEIHANG UNIVERSITY

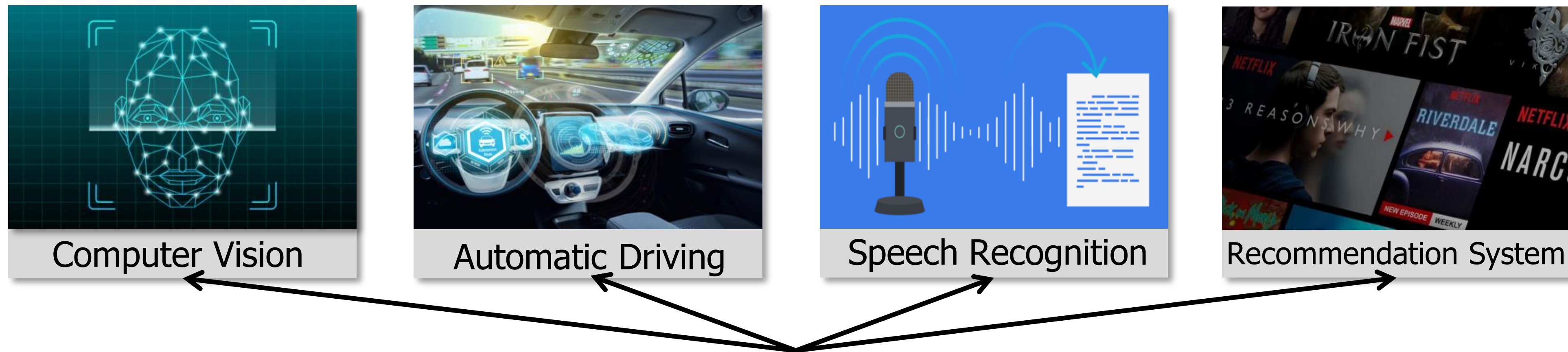
Outline

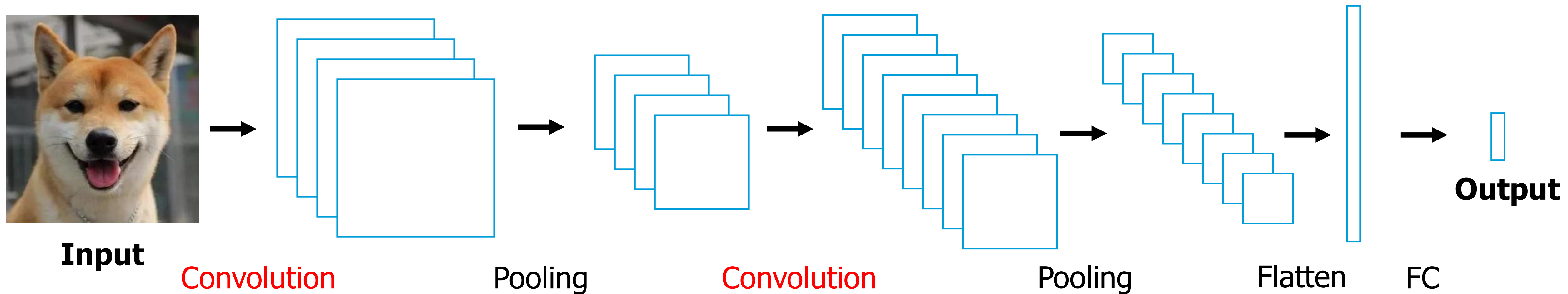
- Background & Motivation
  - CNN & Quantized Neural Network
  - Low-bit Computation on Modern Computer Architectures
- Optimization Methods
  - Low-bit Convolution on ARM CPU
  - Low-bit Convolution on NVIDIA GPU
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

Outline

- Background & Motivation
  - CNN & Quantized Neural Network
  - Low-bit Computation on Modern Computer Architectures
- Optimization Methods
  - Low-bit Convolution on ARM CPU
  - Low-bit Convolution on NVIDIA GPU
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

Convolutional Neural Network

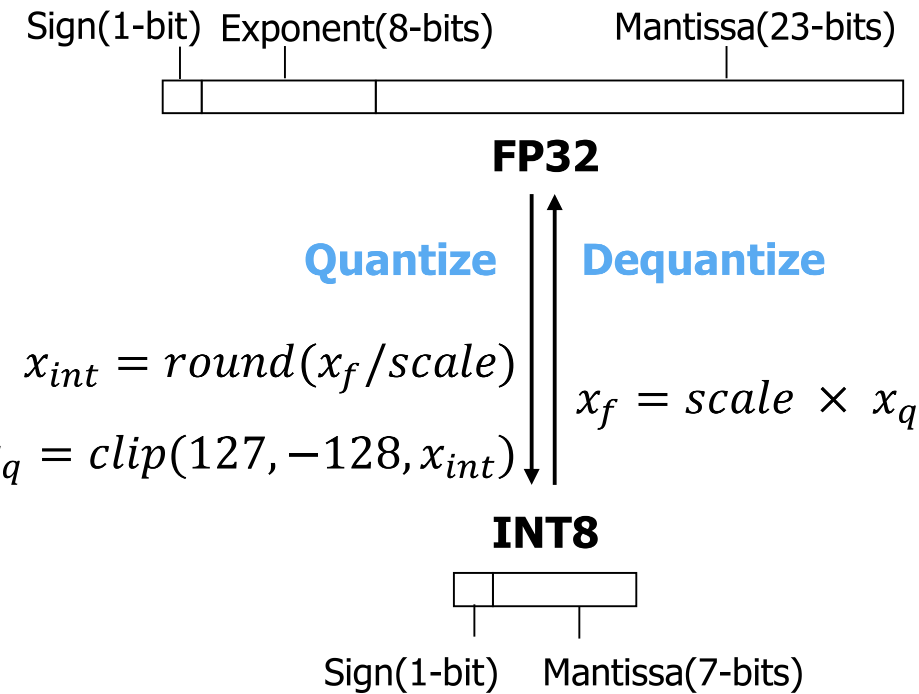


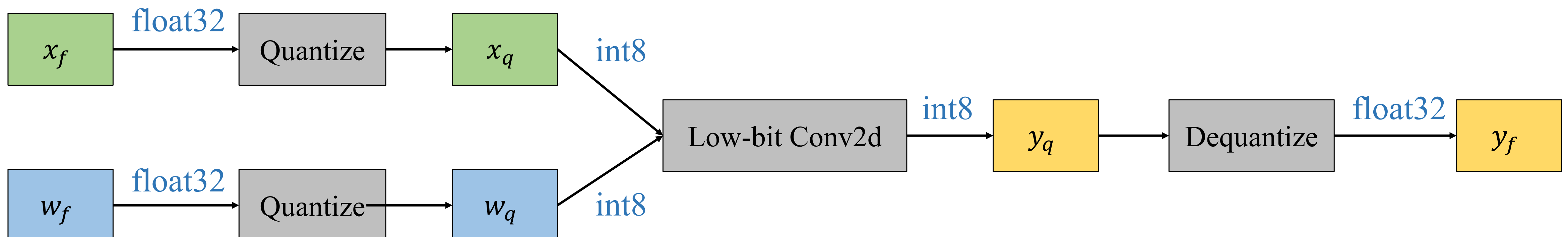


- The computation complexity and memory footprint of CNNs need to be optimized
- Convolution layers take 90% - 99% of computation and runtime [Chen et al., ISSCC'16]

Model Compression

- Model compression reduces computation complexity with acceptable accuracy
  - Network Pruning
  - **Model Quantization**
- Model Quantization
  - Mapping data to a smaller set of numerical representation
  - Improve the performance and reduce memory footprint while preserving accuracy
  - Example: int8 Conv2d quantization





Accuracy of Quantized Neural Network

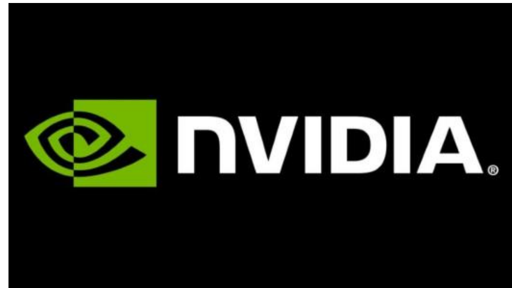
Network	Method	Top-1 Accuracy @ Precision 2	3	4	8	Top-5 Accuracy @ Precision 2	3	4	8
ResNet-18		*Full precision: 70.5*				*Full precision: 89.6*			
	LSQ (Ours)	**67.6**	**70.2**	**71.1**	**71.1**	**87.6**	**89.4**	**90.0**	**90.1**
	QIL	65.7	69.2	70.1					
	FAQ			69.8	70.0			89.1	89.3
	LQ-Nets	64.9	68.2	69.3		85.9	87.9	88.8	
	PACT	64.4	68.1	69.2		85.6	88.2	89.0	
	NICE		67.7	69.8			87.9	89.21	
	Regularization	61.7		67.3	68.1	84.4		87.9	88.2
ResNet-50		*Full precision: 76.9*				*Full precision: 93.4*			
	LSQ (Ours)	**73.7**	**75.8**	**76.7**	**76.8**	**91.5**	**92.7**	93.2	**93.4**
	PACT	72.2	75.3	76.5		90.5	92.6	93.2	
	NICE		75.1	76.5			92.3	**93.3**	
	FAQ			76.3	76.5			92.9	93.1
	LQ-Nets	71.5	74.2	75.1		90.3	91.6	92.4	
VGG-16bn		*Full precision: 73.4*				*Full precision: 91.5*			
	LSQ (Ours)	**71.4**	**73.4**	**74.0**	73.5	**90.4**	**91.5**	**92.0**	**91.6**
	FAQ			73.9	**73.7**			91.7	**91.6**
Squeeze Next-23-2x		*Full precision: 67.3*				*Full precision: 87.8*			
	LSQ (Ours)	**53.3**	**63.7**	**67.4**	**67.0**	**77.5**	**85.4**	**87.8**	**87.7**

Accuracy Comparison of Low-bit QNNs on ImageNet [Esser et al., ICLR'20]

- Recent works have proved the accuracy of quantized neural network
  - 8-bit quantized model can almost reach the same accuracy as the full-precision one
  - Lower-bit quantized models (e.g., 2~4-bit) only loss the accuracy slightly compared to the full-precision ones
- However, achieving the optimal performance of QNNs across different computer architectures is challenging and less studied in literatures

The Target Architectures for Optimization

- Most widely used architectures for CNN inference
  - Edge devices – ARM CPU
  - Cloud accelerators – NVIDIA GPU



- Provide architecture support for low-bit arithmetic instructions
  - ARM CPU: **MLA/SMLAL**
  - NVIDIA GPU: **dp4a/mma**(Tensor Core)

The shipments of ARM-based chips to date

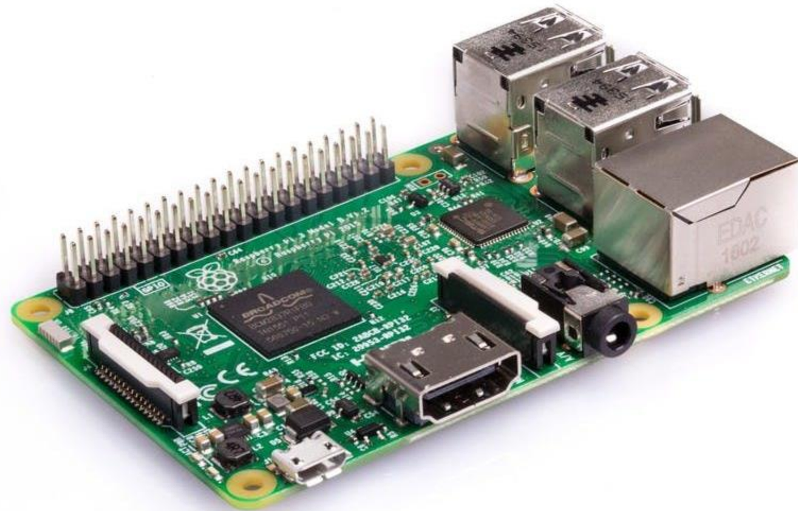
Company	Accelerator	March 2019	April 2019	May 2019
NVIDIA	GPU	97.0%	97.3%	97.4%
AMD	GPU	1.2%	1.1%	1.0%
Xilinx	FPGA	1.1%	1.0%	1.0%
Intel	FPGA	0.6%	0.6%	0.6%
Total Types	All	1,852	1,990	2,003

Source: Liftr Cloud Insights, June 2019

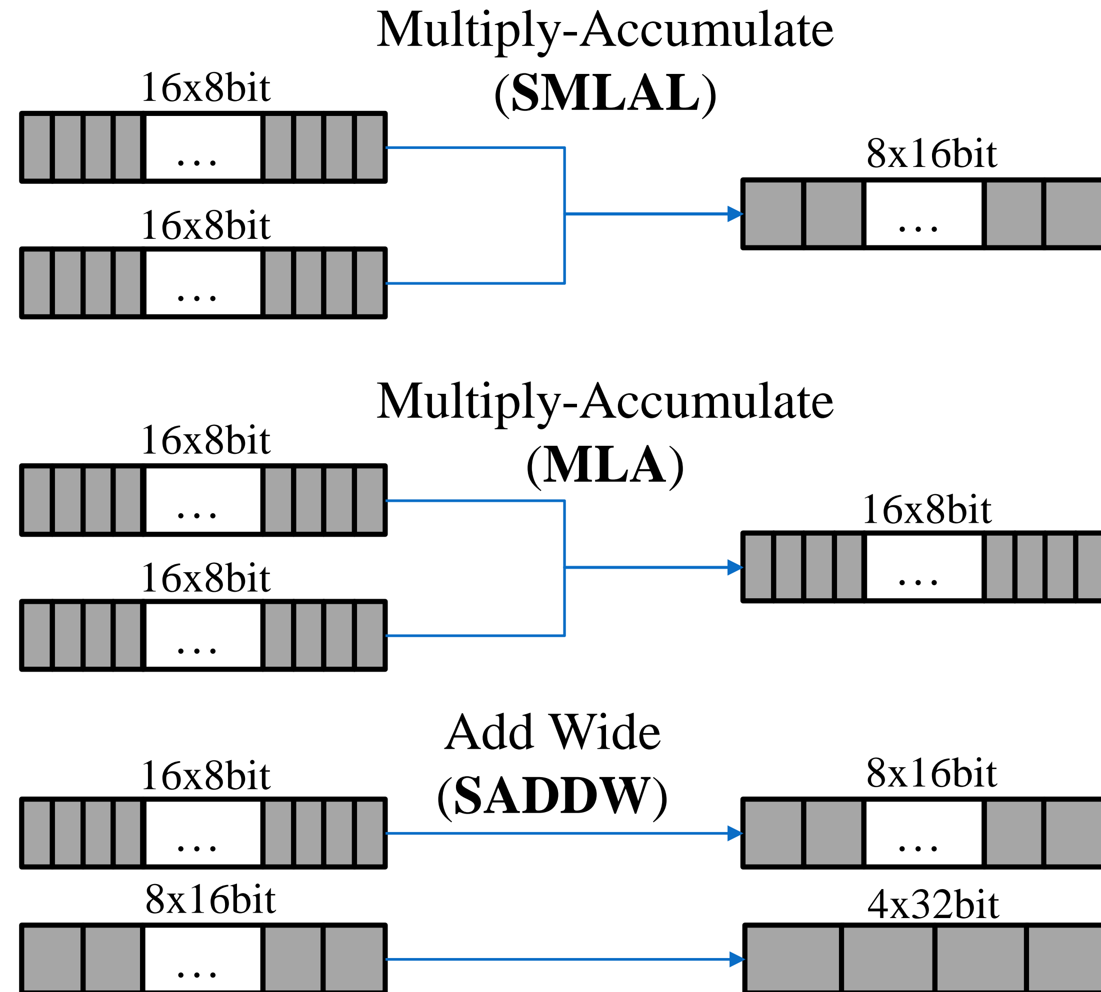
The share of types with Cloud Accelerators

Low-bit Computation Support on ARM CPU

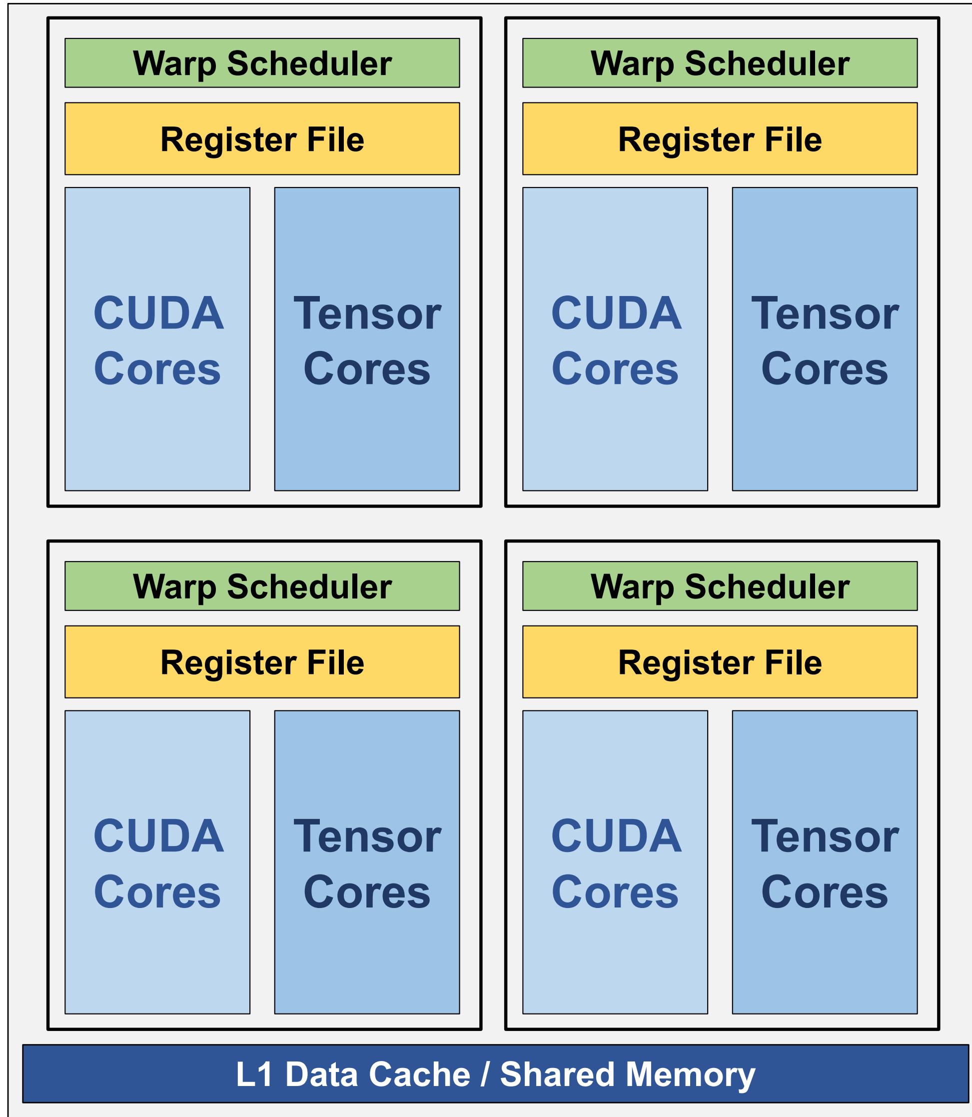
- Low-bit arithmetic instruction



ARMv8.1 architecture



Low-bit Computation Support on NVIDIA GPU



- Tensor Core
  - Natively support mixed-precision GEMM
    - INT8/INT4/INT1 for Turing Tensor Cores
  - Powerful inference performance
    - RTX 2080 Ti delivers up to 215.2 TOPS of INT8 inference performance

$$\begin{bmatrix} D_{11} & D_{12} & D_{13} & D_{14} \\ D_{21} & D_{22} & D_{23} & D_{24} \\ D_{31} & D_{32} & D_{33} & D_{34} \\ D_{41} & D_{42} & D_{43} & D_{44} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ A_{31} & A_{32} & A_{33} & A_{34} \\ A_{41} & A_{42} & A_{43} & A_{44} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ B_{31} & B_{32} & B_{33} & B_{34} \\ B_{41} & B_{42} & B_{43} & B_{44} \end{bmatrix} + \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ C_{31} & C_{32} & C_{33} & C_{34} \\ C_{41} & C_{42} & C_{43} & C_{44} \end{bmatrix}$$

INT32 INT8/INT4 INT8/INT4 INT32

- Use of Tensor Core
  - WMMA API
  - PTX **mma** instructions(e.g. mma.m8n8k16)
  - Vendor libraries: cuBLAS/cuDNN (only fp16 now)

Existing Framework/Library Supporting Low-bit Conv2d

ARM CPU

- ncnn: 8-bit Conv2d(GEMM-based & Winograd)
- QNNPACK: 8-bit Conv2d(indirect convolution)
- TFLite: 8-bit Conv2d
- TVM: 1/2-bit Conv2d(**popcount**)/8-bit Conv2d(spatial pack)

NVIDIA GPU

- cuDNN: 8-bit Conv2d(**dp4a**)/16-bit Conv2d(**Tensor Core**)
- TensorRT: 8-bit Conv2d(**Tensor Core**)
- CUTLASS: 1/4/8-bit GEMM(**Tensor Core**)

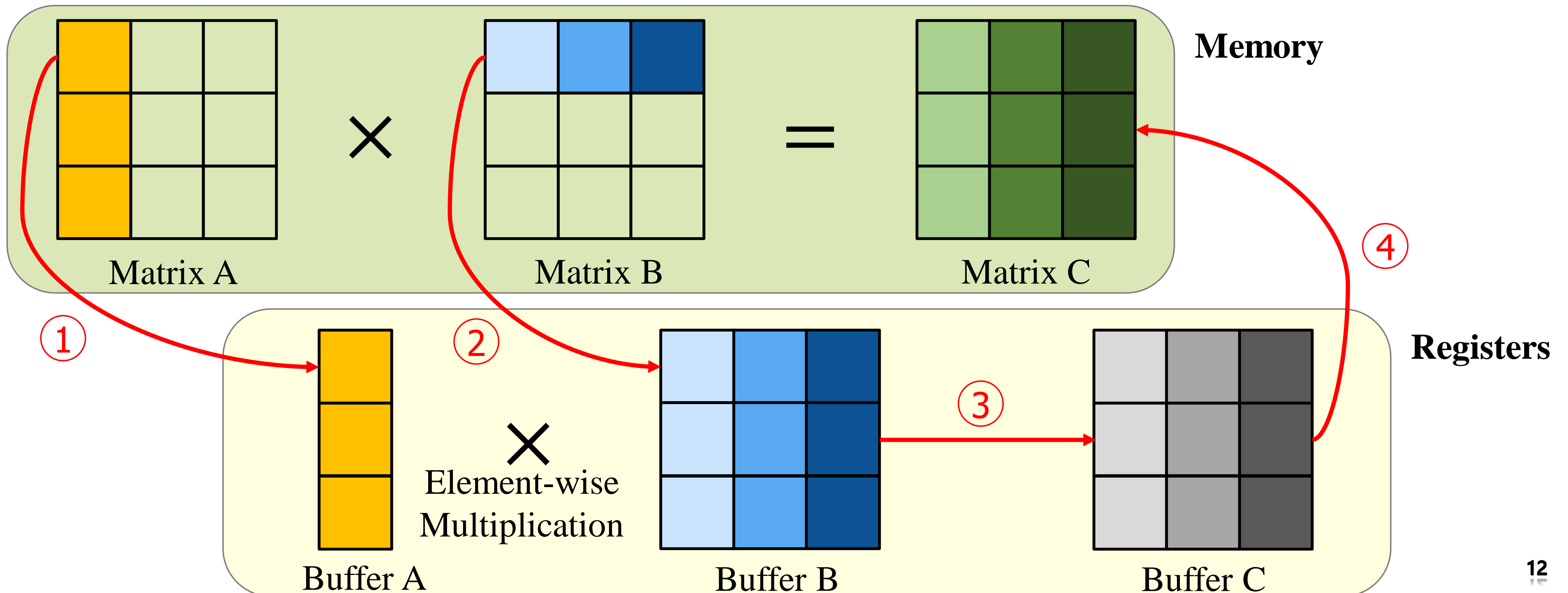
- There is no public work that can support extremely low-bit convolution covering a wide range of bit width on ARM CPU (2~8-bit) and NVIDIA GPU (4-bit/8-bit)
- The missing support for extremely low-bit convolution motivates us to provide efficient implementations on ARM CPU and NVIDIA GPU

Outline

- Background & Motivation
  - CNN & Quantized Neural Network
  - Low-bit Computation on Modern Computer Architectures
- Optimization Methods
  - Low-bit Convolution on ARM CPU
  - Low-bit Convolution on NVIDIA GPU
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

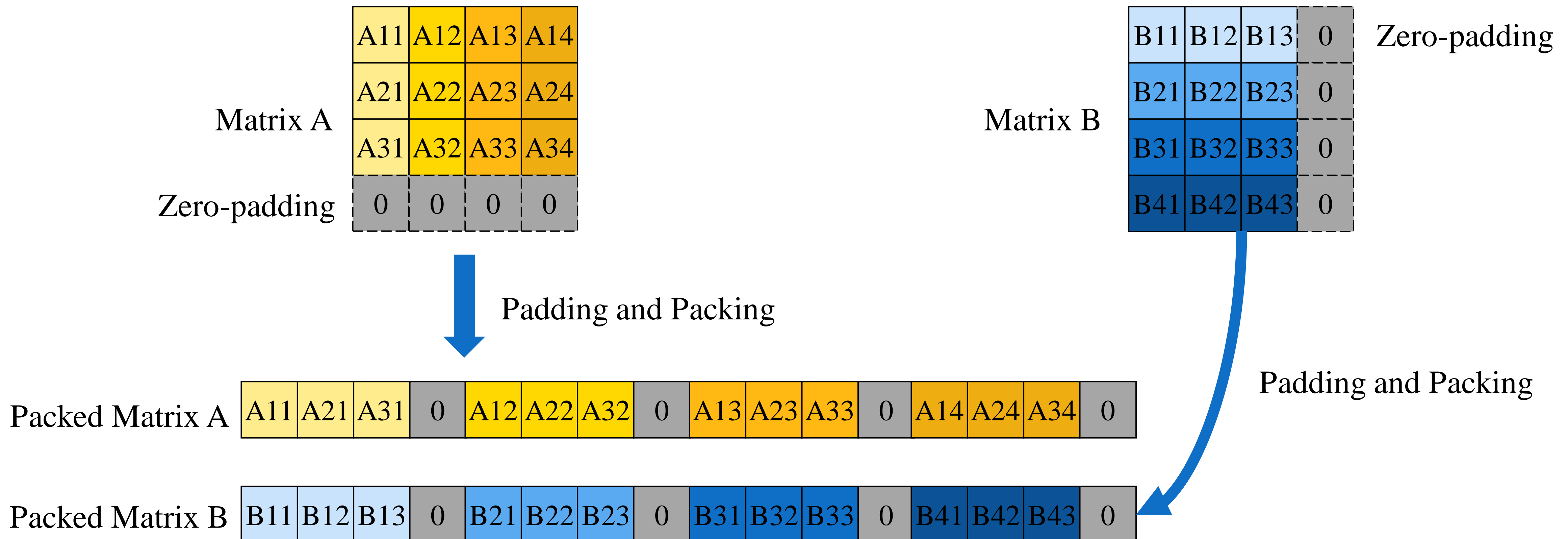
Re-designing GEMM Computation on ARM CPU

- Re-design GEMM micro-kernel
  1. Load one column of Matrix A into Buffer A
  2. Load one row of Matrix B info Buffer B, and replicate it into each row of Buffer B
  3. Perform element-wise multiplication between Buffer A and each column-vector of Buffer B, and store the results to Buffer C
  4. After all the calculations are done, copy the data of Buffer C into Matrix C



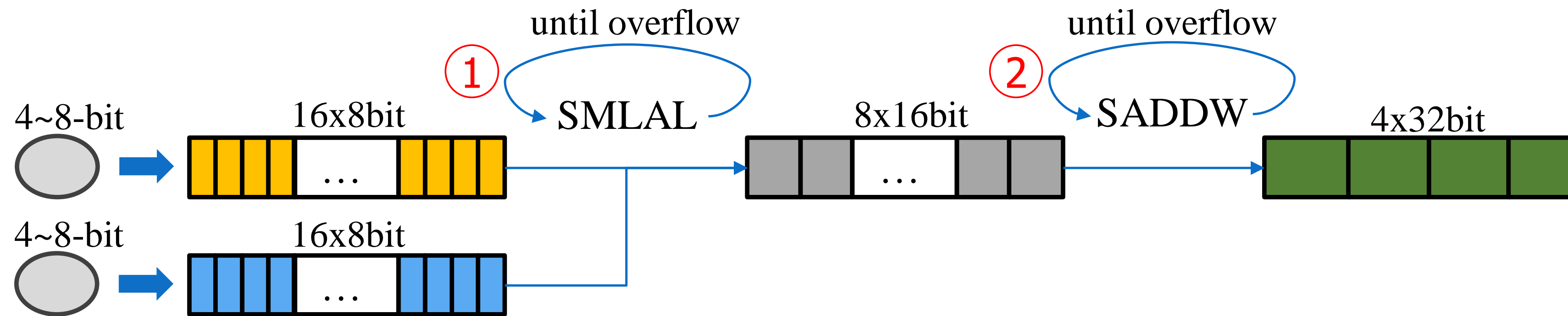
Re-designing GEMM Computation on ARM CPU

- Data padding and packing optimization
  - Perform zero-padding when the dimension of data is not a multiple of the required dimension
  - Perform data packing to enable continuous data access

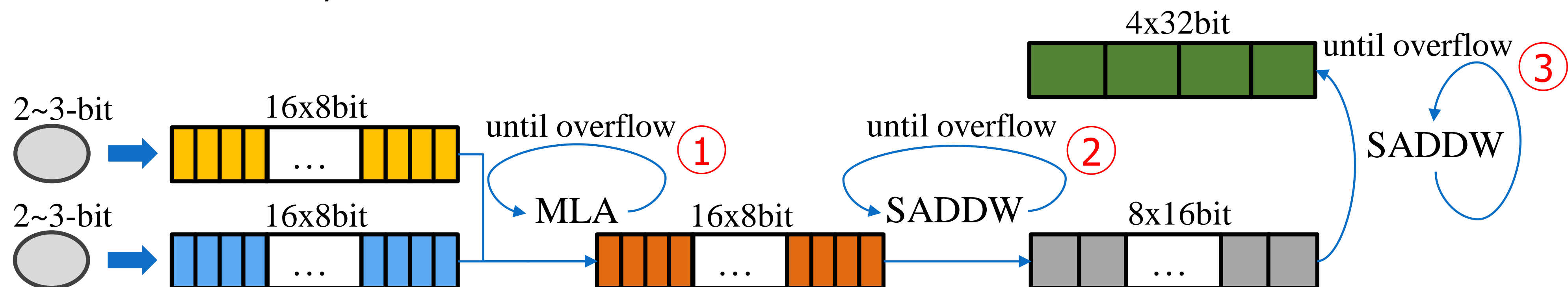


Instruction and Register Allocation Optimization on ARM CPU

- Optimized instruction schemes for GEMM
  - For 4 to 8-bit GEMM, we choose **SMLAL** and **SADDW** instructions

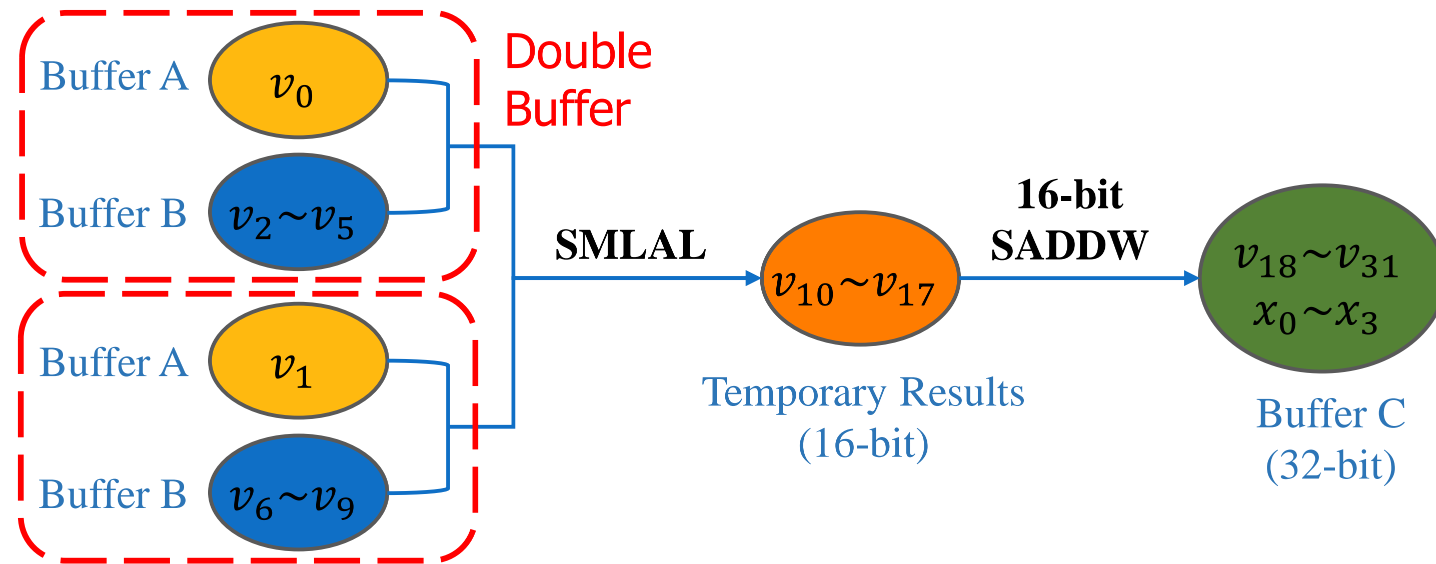


  - For 2 to 3-bit GEMM, we choose **MLA** and **SADDW** instructions

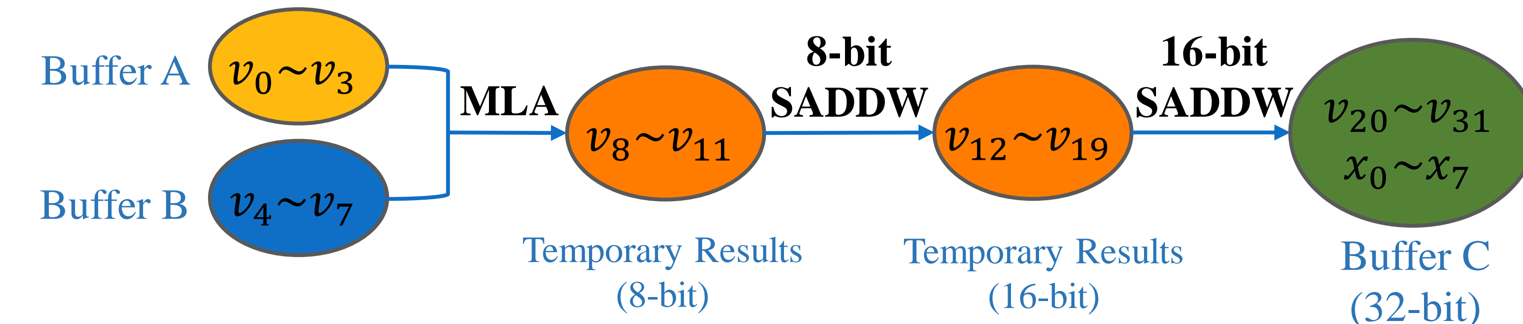


Instruction and Register Allocation Optimization on ARM CPU

- Register allocation optimization
  - For 4~8-bit input data



  - For 2~3-bit input data



Algorithm 1 The 4~8-bit GEMM kernel with register allocation optimization

Input: Padding_and_Packing { *Matrix A* and *Matrix B* }

1: **while** $k > 0$ **do**
2: ...
3: LD1 { v_0 } *addr_Matrix_A*
4: LD4R { v_2 ~v_5 } *addr_Matrix_B*
5: SMLAL(2) { v_{10} ~v_{17} } { v_1 } { v_6 ~v_9 }
6: LD1 { v_1 } *addr_Matrix_A*
7: LD4R { v_6 ~v_9 } *addr_Matrix_B*
8: SMLAL(2) { v_{10} ~v_{17} } { v_0 } { v_2 ~v_5 }
9: ...
10: MOV { v_0, v_1 } { { x_0, x_1 }, { x_2, x_3 } }
11: SADDW(2) { v_{18} ~v_{31} } { v_{10} ~v_{16} }
12: SADDW(2) { v_0, v_1 } { v_{17} }
13: MOV { { x_0, x_1 }, { x_2, x_3 } } { v_0, v_1 }
14: $k \leftarrow k - unrolling_factor$
15: **end while**
16: MOV { v_0, v_1 } { { x_0, x_1 }, { x_2, x_3 } }
17: ST1 { { v_{18} ~v_{31} }, { v_0, v_1 } } *addr_Matrix_C*

Winograd Optimization on ARM CPU

- Winograd method
  - Achieve acceleration by reducing the number of multiplications
  - Converts convolution computation to the following form:

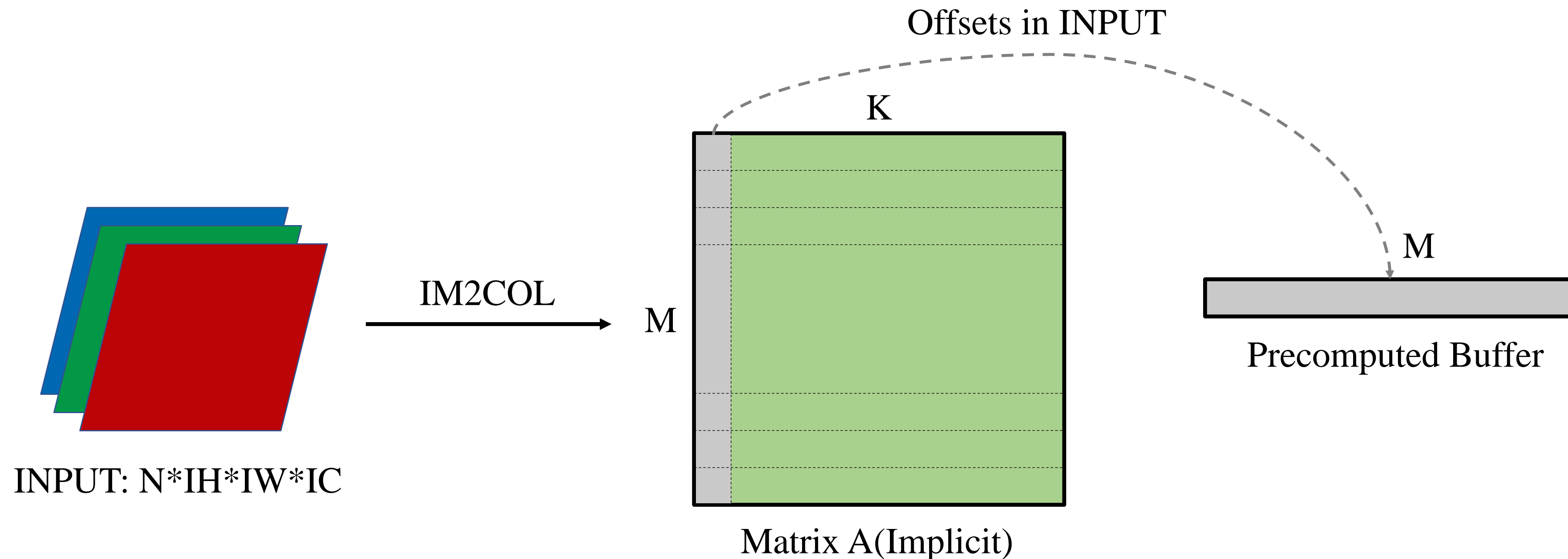
$$F(m \times m, r \times r): \quad Y = A^T[[GgG^T] \odot [B^T dB]]A$$

- Apply F(2x2, 3x3) to 4~6-bit convolution
  - Ensure the transformed data in the range of 8-bit precision
    - F(2x2, 3x3): No more than 6 bits
    - F(4x4, 3x3): Unacceptable increment of numerical range
  - 2 to 3-bit convolution?
    - The maximum theoretical speedup of F(2x2, 3x3) is 2.25×, however **MLA** instruction is 2× faster than **SMLAL** instruction
    - Offset the performance advantage of Winograd method

For more details, please refer to our paper.

Implicit-precomp GEMM Method on GPU

- Implicit GEMM
  - Avoid global matrix transformation and reducing memory footprint
- Precomputed Buffer
  - Store the offsets of elements in precomputed buffer

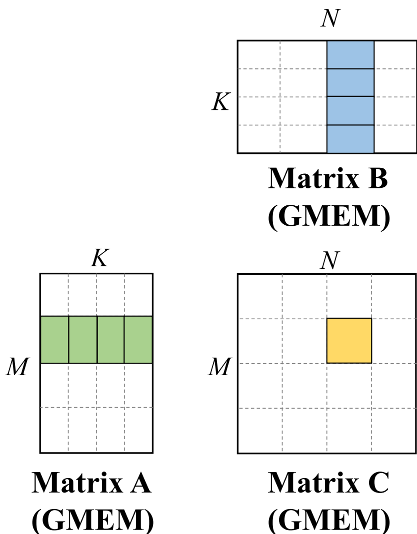


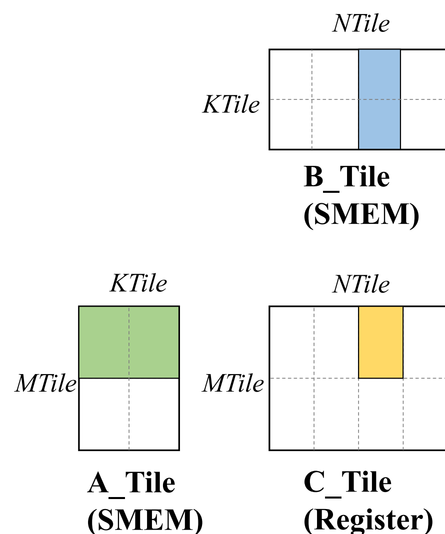
M = (N*OH*OW)   K = (KH*KW*IC)   N = OC

Data Partition along with Thread Hierarchy on GPU

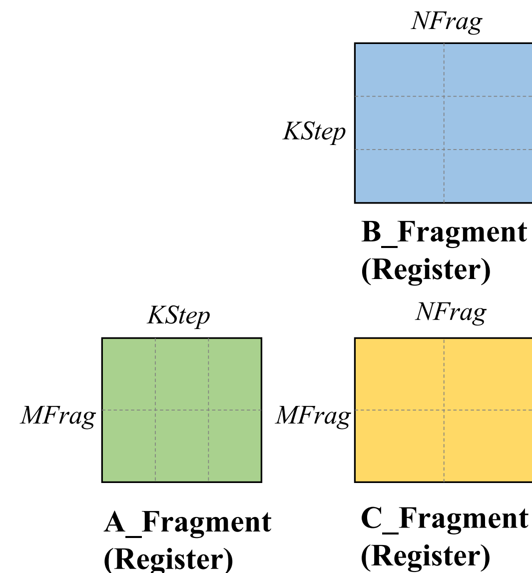
(a) Grid-Level

- Divide the matrix A, B and C into tiles by *MTile*, *NTile*, *KTile*





(a) Grid-Level



(b) Block-Level

(c) Warp-Level

Algorithm 2 Implicit-precomp GEMM-based Conv2D

Input: Shape of convolution and pointers of input, weight and output. The precomputed buffer.

Tiling Parameters: *MTile*, *NTile*, *KTile*, *KStep*, *blockRowWarpNum* and *blockColWarpNum*.

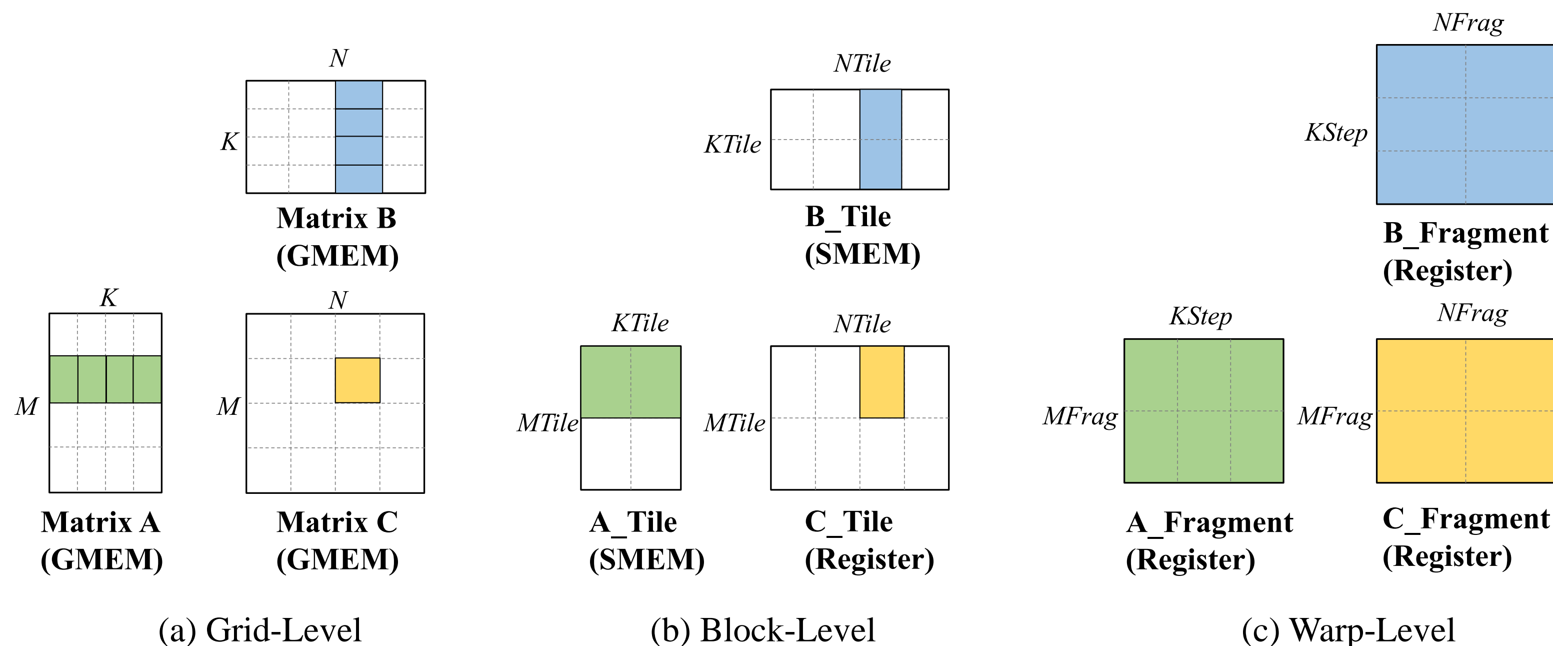
```
1:  compute KTileNum, KStepNum, MFrag, NFrag, warpRowNum
    and warpColNum;
2:  for k_outer in KTileNum do
3:      load A_Tile to shared memory by precomputed buffer;
4:      load B_Tile to shared memory;
5:      __syncthreads();
6:      for k_inner in KStepNum do
7:          load A_Fragment to register;
8:          load B_Fragment to register;
9:          for row in WarpRowNum do
10:             for col in WarpColNum do
11:                 compute C_Fragment by mma instruction;
12:             end for
13:         end for
14:     end for
15:     add bias and re-quantize on register;
16:     store C_Fragment to global memory;
17: end for
```

M = (N*OH*OW) K = (KH*KW*IC) N = OC

Data Partition along Thread Hierarchy on GPU

(b) Block-Level

- Divide C_Tile, A_Tile, B_Tile into fragments by *blockRowWarpNum*, *blockColWarpNum*
- Split the *KTile* loop by *KStep*



(a) Grid-Level (b) Block-Level (c) Warp-Level

Algorithm 2 Implicit-precomp GEMM-based Conv2D

Input: Shape of convolution and pointers of input, weight and output. The precomputed buffer.

Tiling Parameters: *MTile*, *NTile*, *KTile*, *KStep*, *blockRowWarpNum* and *blockColWarpNum*.

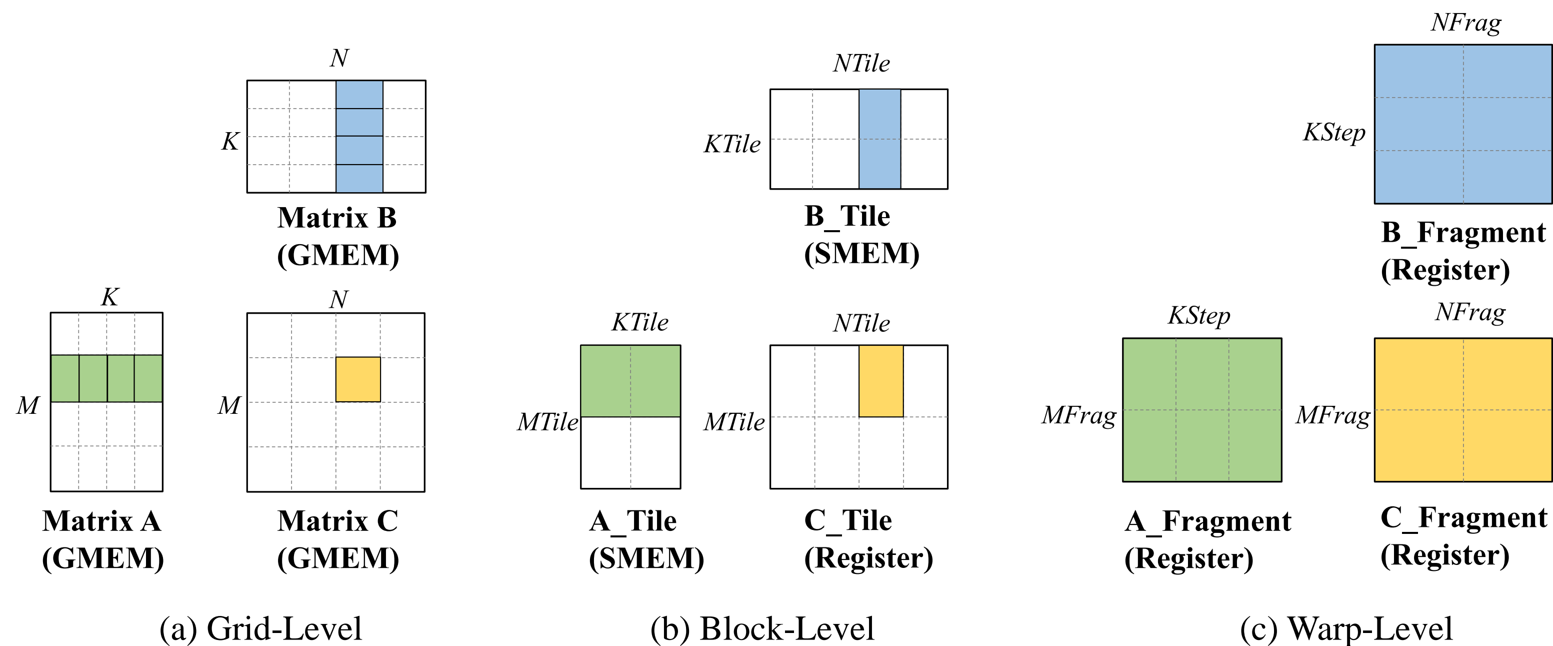
1: compute *KTileNum*, *KStepNum*, *MFrag*, *NFrag*, *warpRowNum* and *warpColNum*;
2: **for** *k_outer* in *KTileNum* **do**
3: load **A_Tile** to shared memory by precomputed buffer;
4: load **B_Tile** to shared memory;
5: __syncthreads();
6: **for** *k_inner* in *KStepNum* **do**
7: load **A_Fragment** to register;
8: load **B_Fragment** to register;
9: **for** *row* in *WarpRowNum* **do**
10: **for** *col* in *WarpColNum* **do**
11: compute **C_Fragment** by `mma` instruction;
12: **end for**
13: **end for**
14: **end for**
15: add bias and re-quantize on register;
16: store **C_Fragment** to global memory;
17: **end for**

M = (N*OH*OW) K = (KH*KW*IC) N = OC

Data Partition along Thread Hierarchy on GPU

(c) Warp-Level

- Call Tensor Core through **mma** instructions to perform the matrix multiplication



(a) Grid-Level (b) Block-Level (c) Warp-Level

M = (N*OH*OW) K = (KH*KW*IC) N = OC

Algorithm 2 Implicit-precomp GEMM-based Conv2D

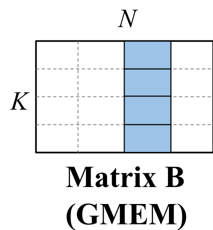
Input: Shape of convolution and pointers of input, weight and output. The precomputed buffer.

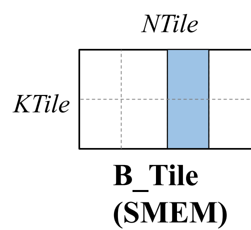
Tiling Parameters: *MTile*, *NTile*, *KTile*, *KStep*, *blockRowWarpNum* and *blockColWarpNum*.

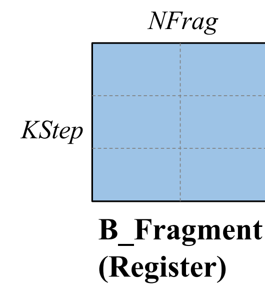
1: compute *KTileNum*, *KStepNum*, *MFrag*, *NFrag*, *warpRowNum* and *warpColNum*;
2: **for** *k_outer* in *KTileNum* **do**
3: load **A_Tile** to shared memory by precomputed buffer;
4: load **B_Tile** to shared memory;
5: __syncthreads();
6: **for** *k_inner* in *KStepNum* **do**
7: load **A_Fragment** to register;
8: load **B_Fragment** to register;
9: **for** *row* in *WarpRowNum* **do**
10: **for** *col* in *WarpColNum* **do**
11: compute **C_Fragment** by `mma` instruction;
12: **end for**
13: **end for**
14: **end for**
15: add bias and re-quantize on register;
16: store **C_Fragment** to global memory;
17: **end for**

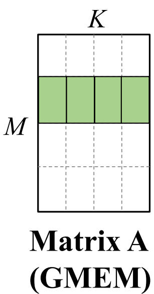
Data Partition along Thread Hierarchy on GPU

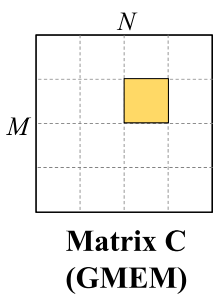
- Auto-tuning of tiling parameters
  - Use C++ function template to generate multiple kernels with different combinations of parameters
  - Choose the best one through profile runs
  - The optimal tiling parameters only need to be determined once per convolution shape with negligible overhead

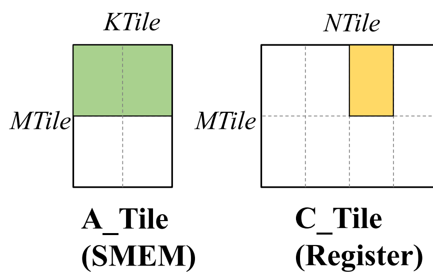


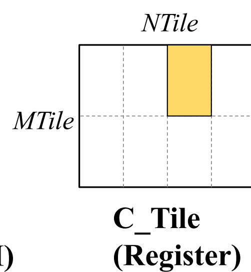


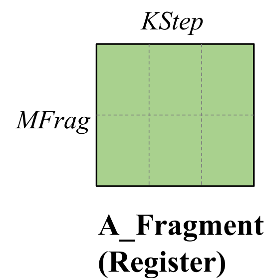


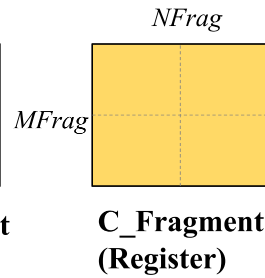












(a) Grid-Level (b) Block-Level (c) Warp-Level

Algorithm 2 Implicit-precomp GEMM-based Conv2D

Input: Shape of convolution and pointers of input, weight and output. The precomputed buffer.

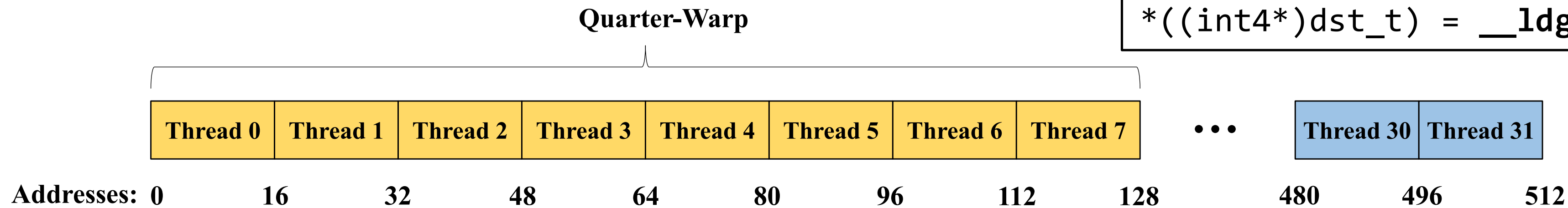
Tiling Parameters: *MTile*, *NTile*, *KTile*, *KStep*, *blockRowWarpNum* and *blockColWarpNum*.

1: compute *KTileNum*, *KStepNum*, *MFrag*, *NFrag*, *warpRowNum* and *warpColNum*;
2: **for** *k_outer* in *KTileNum* **do**
3: load **A_Tile** to shared memory by precomputed buffer;
4: load **B_Tile** to shared memory;
5: __syncthreads();
6: **for** *k_inner* in *KStepNum* **do**
7: load **A_Fragment** to register;
8: load **B_Fragment** to register;
9: **for** *row* in *WarpRowNum* **do**
10: **for** *col* in *WarpColNum* **do**
11: compute **C_Fragment** by `mma` instruction;
12: **end for**
13: **end for**
14: **end for**
15: add bias and re-quantize on register;
16: store **C_Fragment** to global memory;
17: **end for**

Multi-level Memory Access Optimization on GPU

1. Coalesced access on global memory

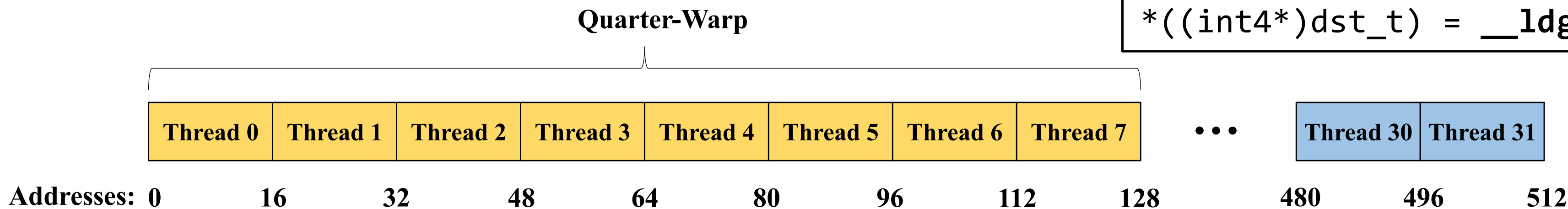
```
// Example code
// char* src_t;
// char* dst_t;
*((int4*)dst_t) = __ldg((int4*)(src_t + threadIdx.x * 16));
```



Multi-level Memory Access Optimization on GPU

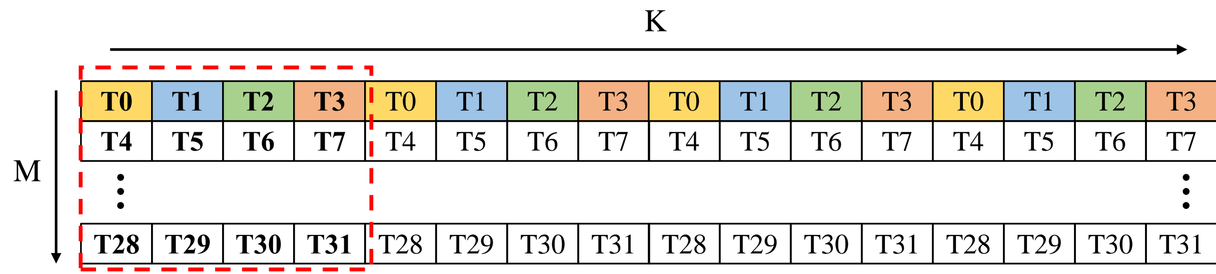
1. Coalesced access on global memory

```
// Example code
// char* src_t;
// char* dst_t;
*((int4*)dst_t) = __ldg((int4*)(src_t + threadIdx.x * 16));
```

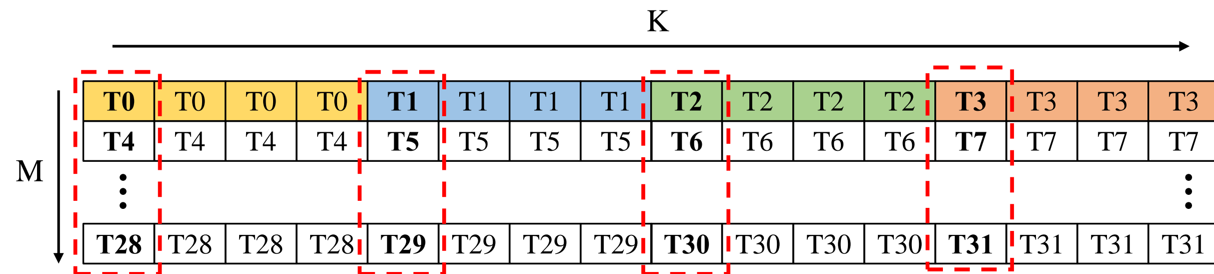


2. Reordering memory access on shared memory

- Reduce the number of LDS instructions to 1/4 of the original



(a) Before Reordering

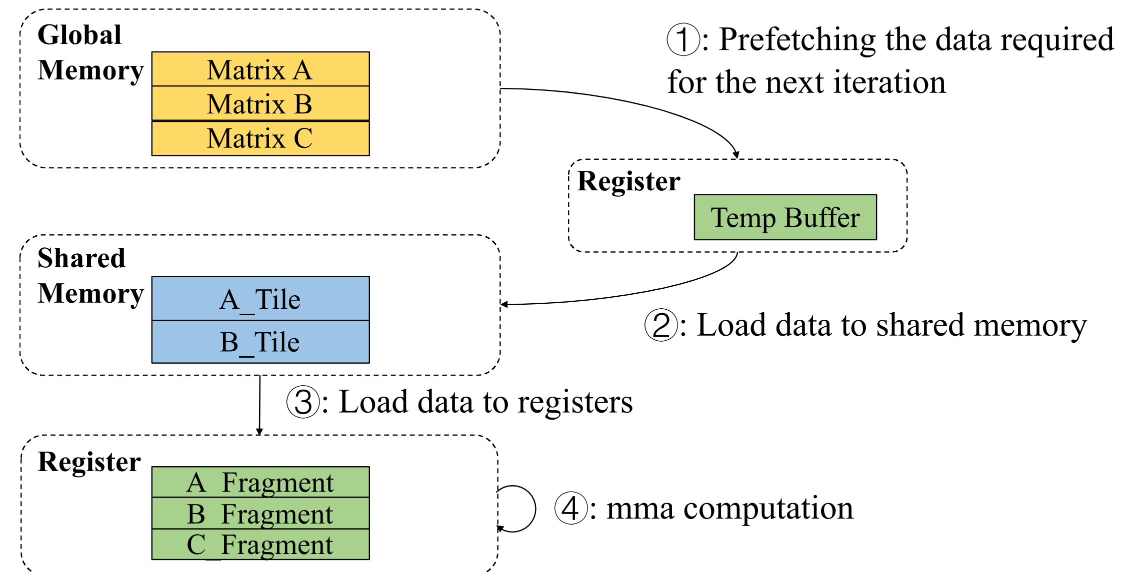


(b) After Reordering

Multi-level Memory Access Optimization on GPU

3. Overlapped computation and memory access using registers

- A temporary buffer on registers to prefetch the data required for the next iteration
- The processes ① and ④ can be performed simultaneously



4. In-place calculation of bias and re-quantization

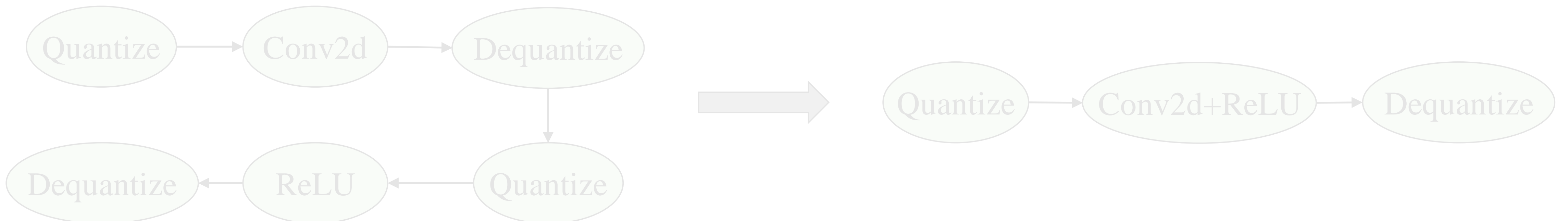
- After finishing the **mma** calculation, directly apply bias and re-quantization on the registers

Quantization Fusion on GPU

1. Fusion of convolution and dequantization
   - Directly transform the results from int32 to float32 in convolution kernel
   - Skip storing the intermediate results with int8 data type

For more details, please refer to our paper.

2. Fusion of convolution and ReLU
   - Change the truncated range of re-quantization in convolution kernel
   - Eliminate the overhead of unnecessary computation and memory access



Outline

- Background & Motivation
  - CNN & Quantized Neural Network
  - Low-bit Computation on Modern Computer Architectures
- Optimization Methods
  - Low-bit Convolution on ARM CPU
  - Low-bit Convolution on NVIDIA GPU
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

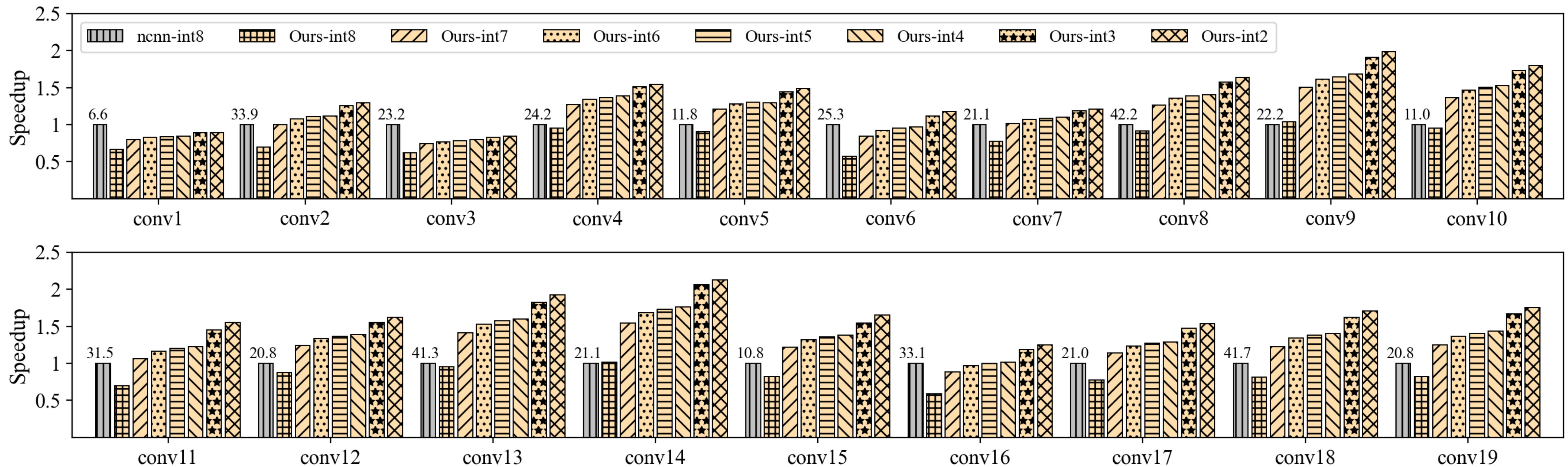
Experiment Setup

- Hardware and software
- Models
  - **ResNet-50(all non-redundant layers)**
  - DenseNet-121
- Batch size
  - ARM: 1
  - GPU: 1 & 16
- Methods for comparison
  - ARM:
    - ncnn 8-bit Conv2d(baseline)
    - TVM 2-bit Conv2d
  - GPU:
    - cuDNN 8-bit Conv2d with dp4a instruction(baseline)
    - TensorRT 8-bit Conv2d with Tensor Core

Table 1: Hardware and software configurations.

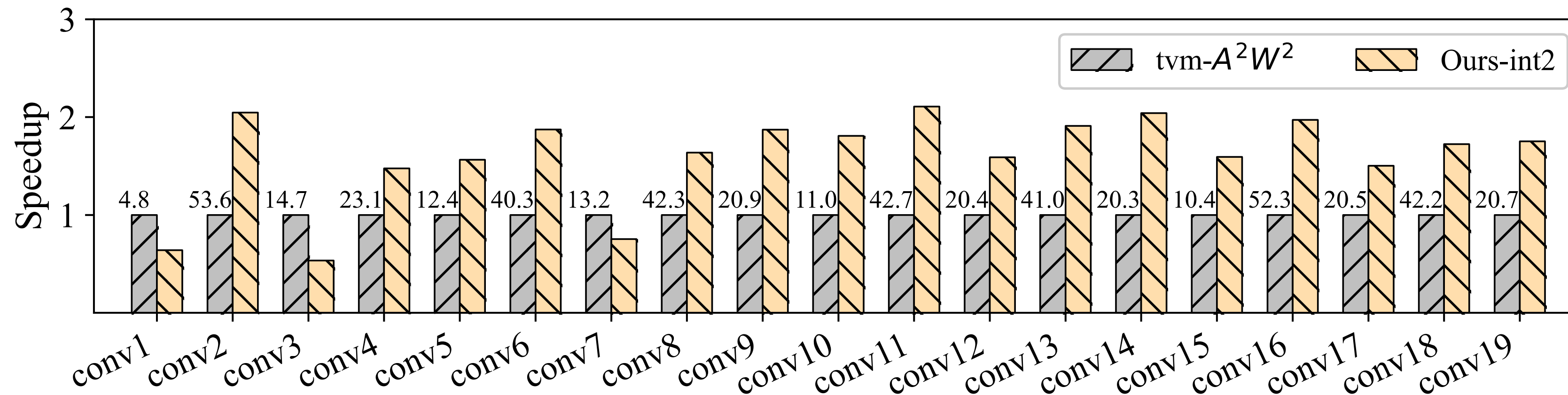
Platform	ARM CPU	NVIDIA GPU
Device	Raspberry Pi 3B	RTX 2080Ti
Architecture	ARM Cortex-A53	NVIDIA Turing TU102
Software	Ubuntu 16.04 LTS for Raspberry Pi, gcc 5.4.0, ncnn with commit 6f2ef19	Ubuntu 16.04 LTS, gcc 5.4.0, CUDA 10.2, cuDNN 7.6.5, TensorRT 7

Performance Comparison On ARM CPU



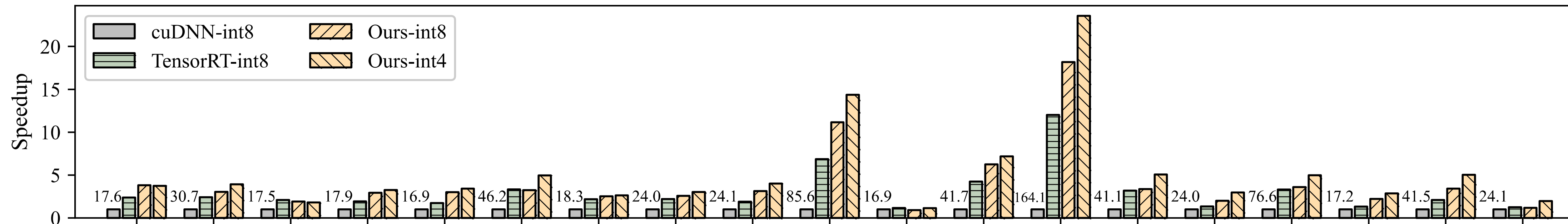
- The performance of our optimized 2~7-bit convolution kernels exceeds ncnn in most layers for ResNet-50, with average speedup of 1.60×, 1.54×, 1.38×, 1.38×, 1.34× and 1.27×, respectively

Performance Comparison with TVM On ARM CPU

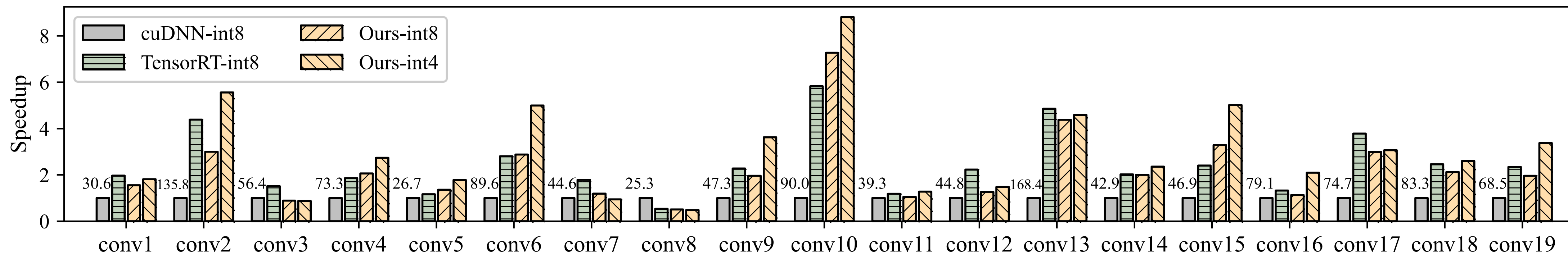


- Our 2-bit implementation outperforms TVM in most cases (16 out of 19 cases), with the highest speedup of 2.11× and the average speedup of 1.78×

Performance Comparison On NVIDIA GPU



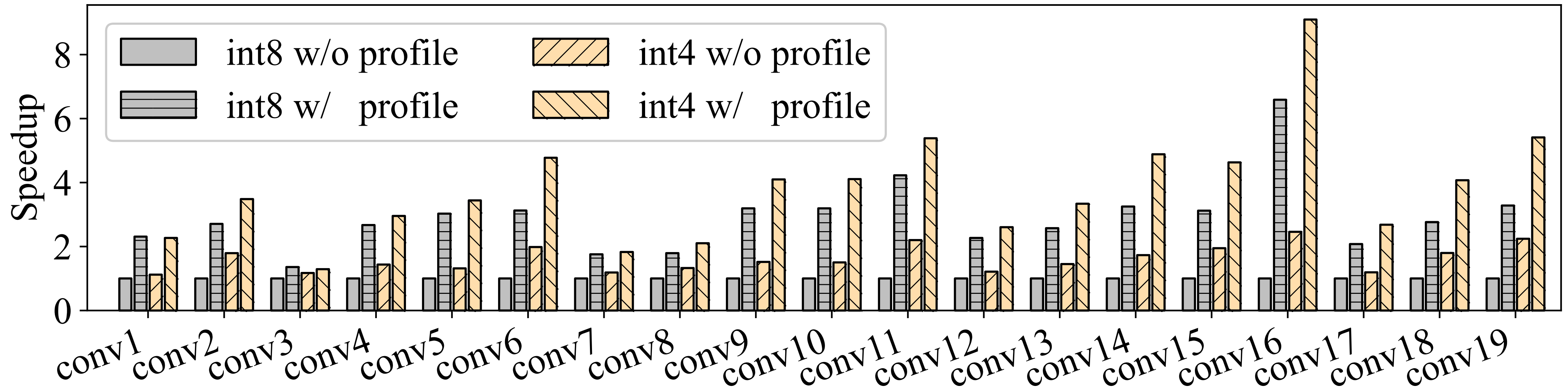
(a) batch size = 1



(b) batch size = 16

- With the batch size of 1, our 4-bit and 8-bit convolution kernels outperform TensorRT in most cases, with the average speedup of 1.78× and 1.44×, respectively
- With the batch size of 16, our 4-bit kernels also outperform TensorRT in 12 layers by an average speedup of 1.46×

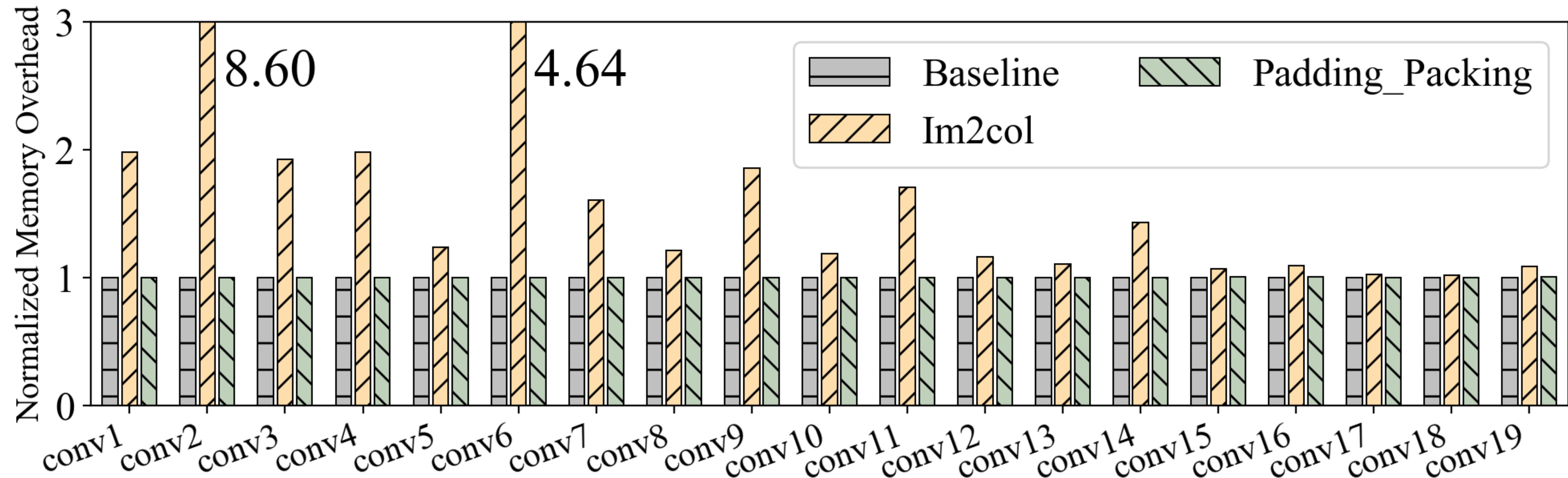
Performance Improvement with Profile Runs on GPU



- The average speedup of 4-bit and 8-bit convolution kernels with the profile runs enabled is 2.29× and 2.91×, respectively

Space Overhead

- GPU: Negligible overhead consumed by precomputed buffer
- ARM: The space overhead of im2col, data padding and packing operations
  - The baseline is space occupation of activation and weight for each layer
  - The overhead of im2col for some layers(e.g., *conv2* and *conv6*) is relatively high
  - The space overhead of im2col is determined by convolution kernel size, stride, and input size



Outline

- Background & Motivation
  - CNN & Quantized Neural Network
  - Low-bit Computation on Modern Computer Architectures
- Optimization Methods
  - Low-bit Convolution on ARM CPU
  - Low-bit Convolution on NVIDIA GPU
- Evaluation
  - Experiment Setup
  - Performance Analysis
- Conclusion

Conclusion

- Explore extremely low-bit convolution optimizations
  - ARM CPU
    - Re-design GEMM computation
    - Instruction and register allocation optimization
    - Winograd optimization
  - NVIDIA GPU
    - Data partition along with thread hierarchy
    - Multi-level memory access optimization
    - Quantization fusion
- Significant speedup compared to existing framework/library
  - ARM CPU: 1.60 x (2-bit) / 1.38 x (4-bit)
  - NVIDIA GPU: 5.26 x (4-bit) / 4.31 x (8-bit)

Thanks! Q&A