INTERNATIONAL CONFERENCE ON PARALLEL PROCESSING
ICPP / 2021 / CHICAGO / USA
acm In-Cooperation
sighpc
AUGUST 9-12, 2021

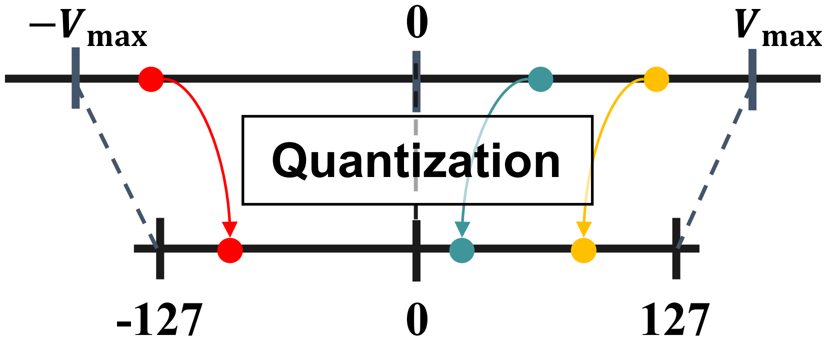
LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs

Guangli Li [1,2], Zhen Jia [3], Xiaobing Feng [1,2], Yida Wang [3]

[1] SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China
[2] University of Chinese Academy of Sciences, China
[3] Amazon Web Services, USA

Optimizing Convolution Operators

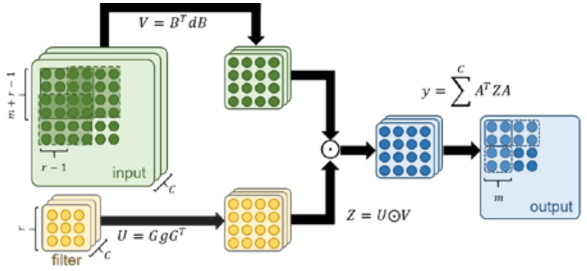
Low-Precision Computation



FP32

INT8

Fast Convolution



Winograd Algorithm

Reducing the number of multiplications tackling direct convolution.

Arithmetic Complexity Reduction

F(2x2,3x3): 2.25x
F(4x4,3x3): 4x

?

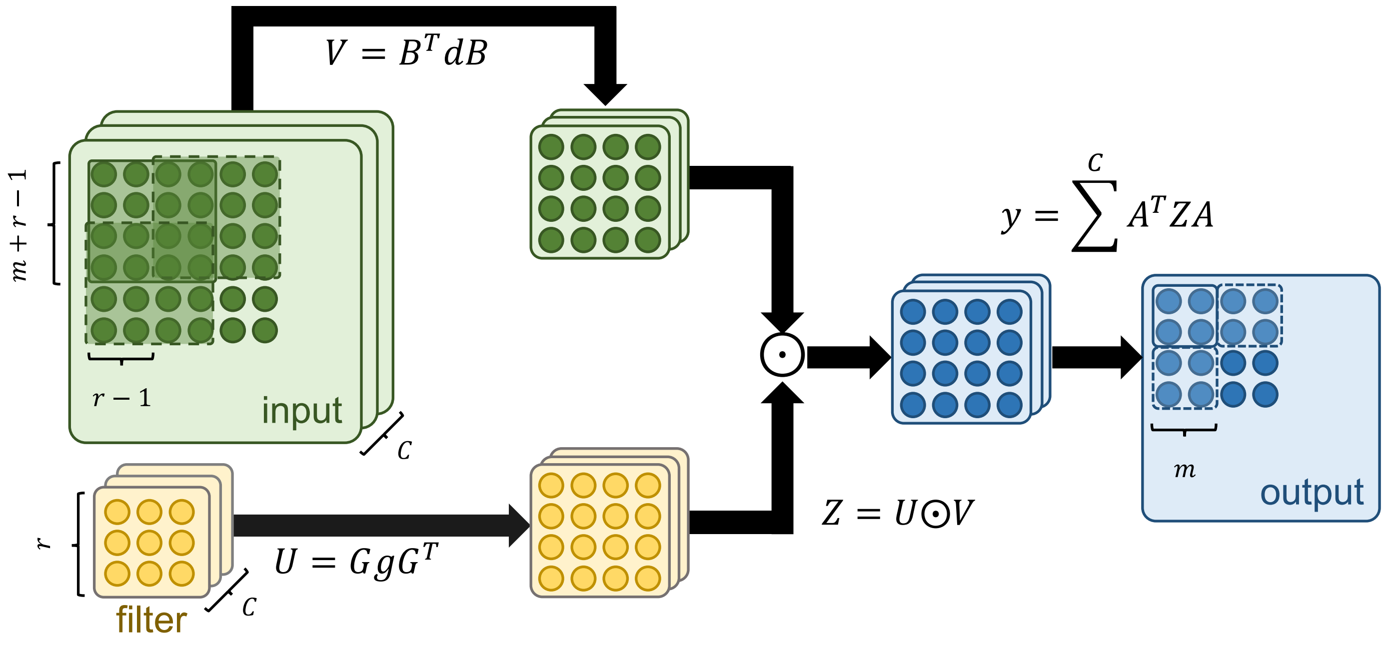
Combination of
Low-Precision Computation
and Winograd Algorithm

Winograd Convolution

- $F(m{\times}m,\ r{\times}r)$
  - Output Size: $m \times m$; Filter Size: $r \times r$;
  - Input size: $(m+r-1) \times (m+r-1)$

Input Tile: d
Filter: g

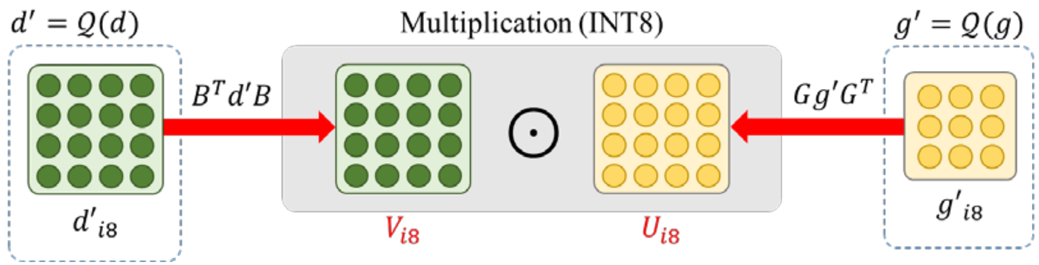
$U_{f32} = Gg_{f32}G^T$	Filter Trans
$V_{f32} = B^T d_{f32} B$	Input Trans
$Z_{f32} = U_{f32} \odot V_{f32}$	Multiplication
$Y_{f32} = A^T Z_{f32} A$	Output Trans



$$y = A^T \left[[GgG^T] \odot [B^T dB] \right] A$$

Motivating Examples

Brute-Force Approach



Input Tile: d Quantized Input Tile: $d' = Q(d)$

Filter: g Quantized Filter: $g' = Q(g)$

$$U_{i8} = G g'_{i8} G^T$$

$$V_{i8} = B^T d'_{i8} B$$

$$Z_{i32} = U_{i8} \odot V_{i8}$$

$$y_{f32} = A^T Z_{i32} A$$

Brute-Force Approach

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

$$d'_{i8} = \begin{bmatrix} 127 & 127 & 127 & 127 \\ 127 & 127 & 127 & 127 \\ 127 & 127 & 127 & 127 \\ 127 & 127 & 127 & 127 \end{bmatrix}$$

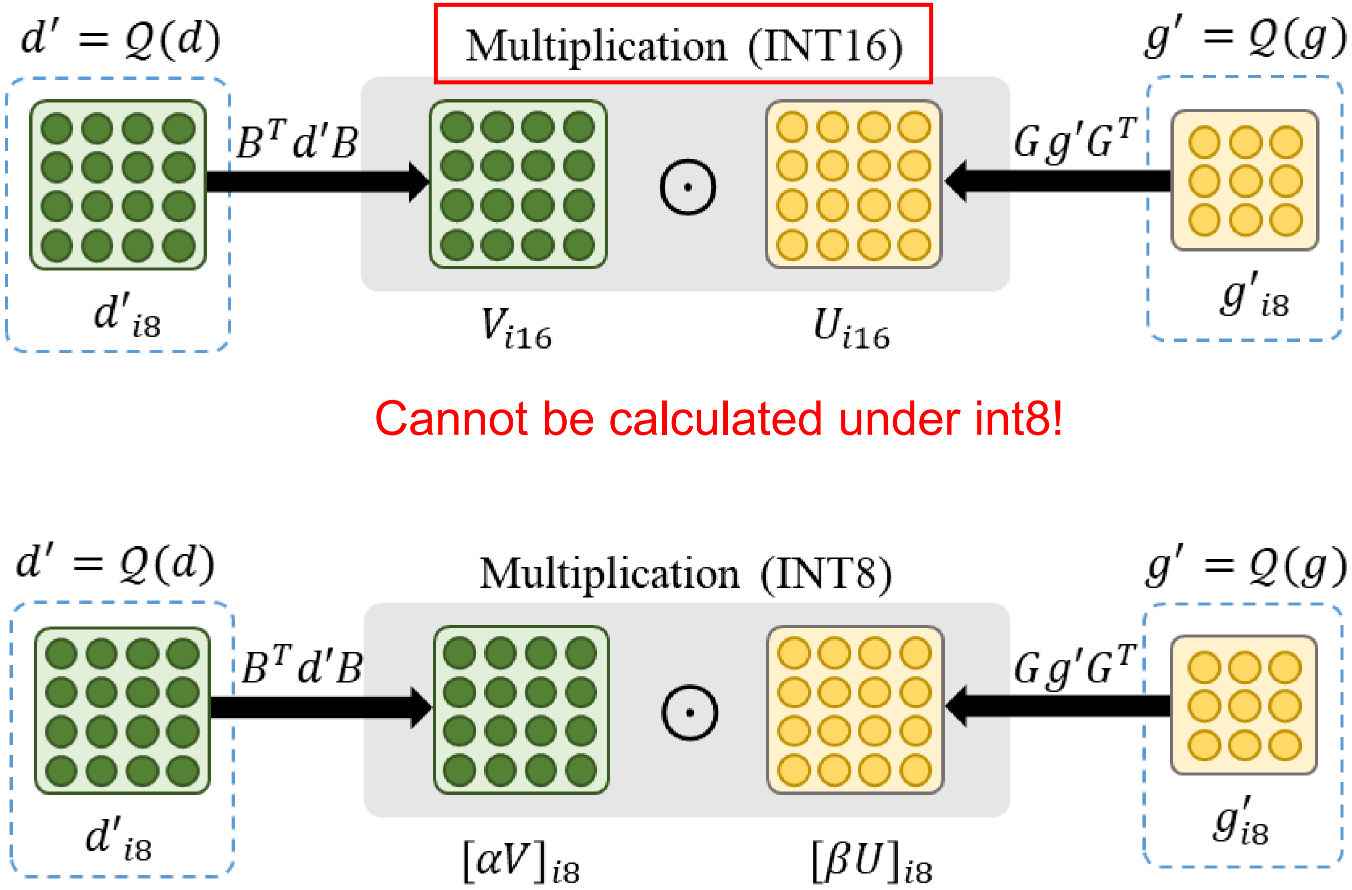
$$V_{i8} = B^T d'_{i8} B = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 508 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

increase values up to 4x

Overflow: Out of Range!

Motivating Examples

(a)
Up-Casting Approach

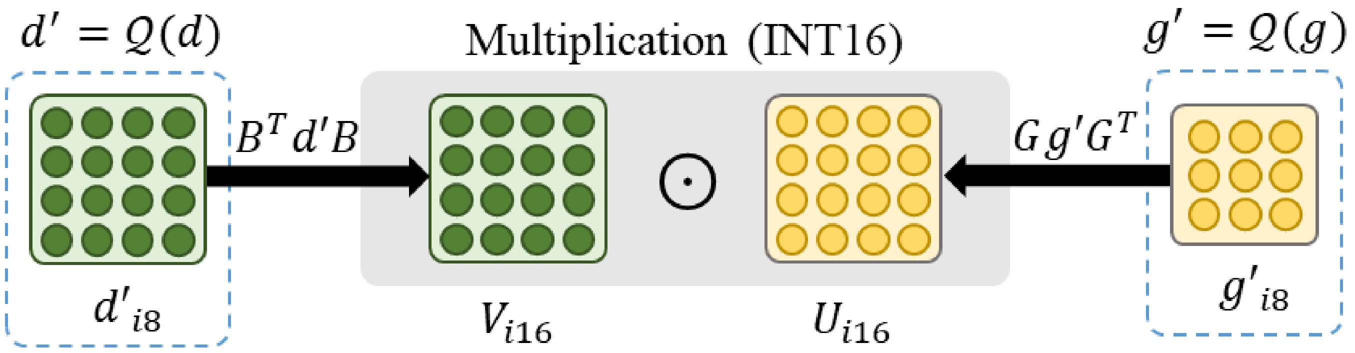


Cannot be calculated under int8!

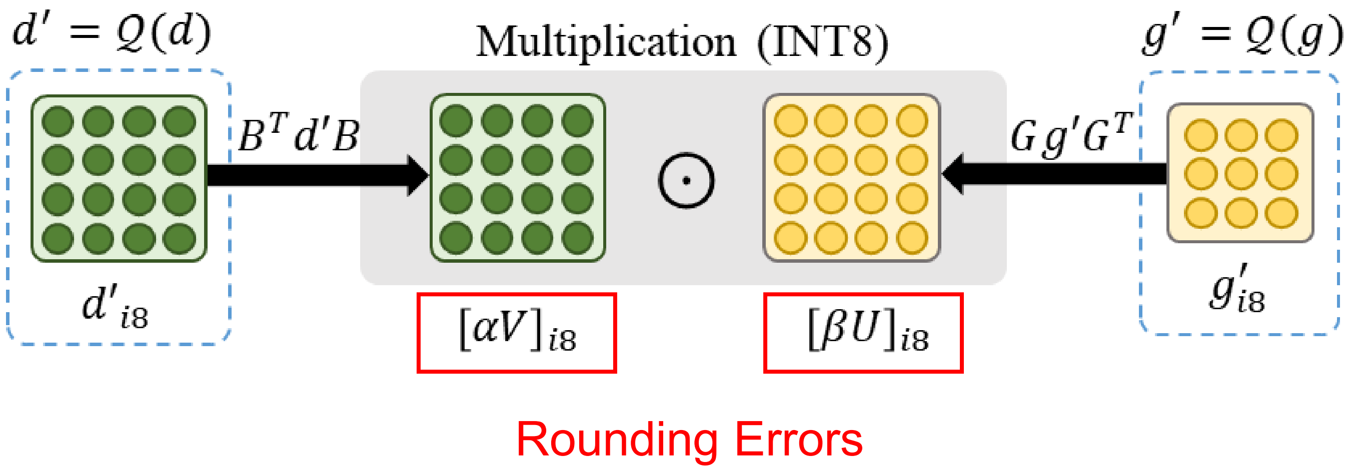
(b)
Down-Scaling Approach

Motivating Examples

(a)
Up-Casting
Approach



(b)
Down-Scaling
Approach



Existing Approaches

Input Tile: d Quantized Input Tile: $d' = Q(d)$

Filter: g Quantized Filter: $g' = Q(g)$

S1: Filter Trans

S2: Input Trans

S3: Multiplication

S4: Output Trans

	Overflow	Degradation	Distortion
$U_{f32} = G g_{f32} G^T$	$U_{i8} = G g'_{i8} G^T$	$U_{i16} = G g'_{i8} G^T$	$U_{f32} = G g'_{i8} G^T$
$V_{f32} = B^T d_{f32} B$	$V_{i8} = B^T d'_{i8} B$	$V_{i16} = B^T d'_{i8} B$	$V_{f32} = B^T d'_{i8} B$
$Z_{f32} = U_{f32} \odot V_{f32}$	$Z_{i32} = U_{i8} \odot V_{i8}$	$Z_{i32} = U_{i16} \odot V_{i16}$	$Z_{i32} = [\alpha U_{f32}]_{i8} \odot [\beta V_{f32}]_{i8}$
$y_{f32} = A^T Z_{f32} A$	$y_{f32} = A^T Z_{i32} A$	$y_{f32} = A^T Z_{i32} A$	$y_{f32} = A^T (\alpha^{-1} \beta^{-1} Z_{i32}) A$
Full-Precision Approach	Brute-Force Approach	Up-Casting Approach	Down-Scaling Approach

LoWino: Quantization in the Winograd Domain

$$U_{f32} = G g_{f32} G^T$$

$$V_{f32} = B^T d_{f32} B$$

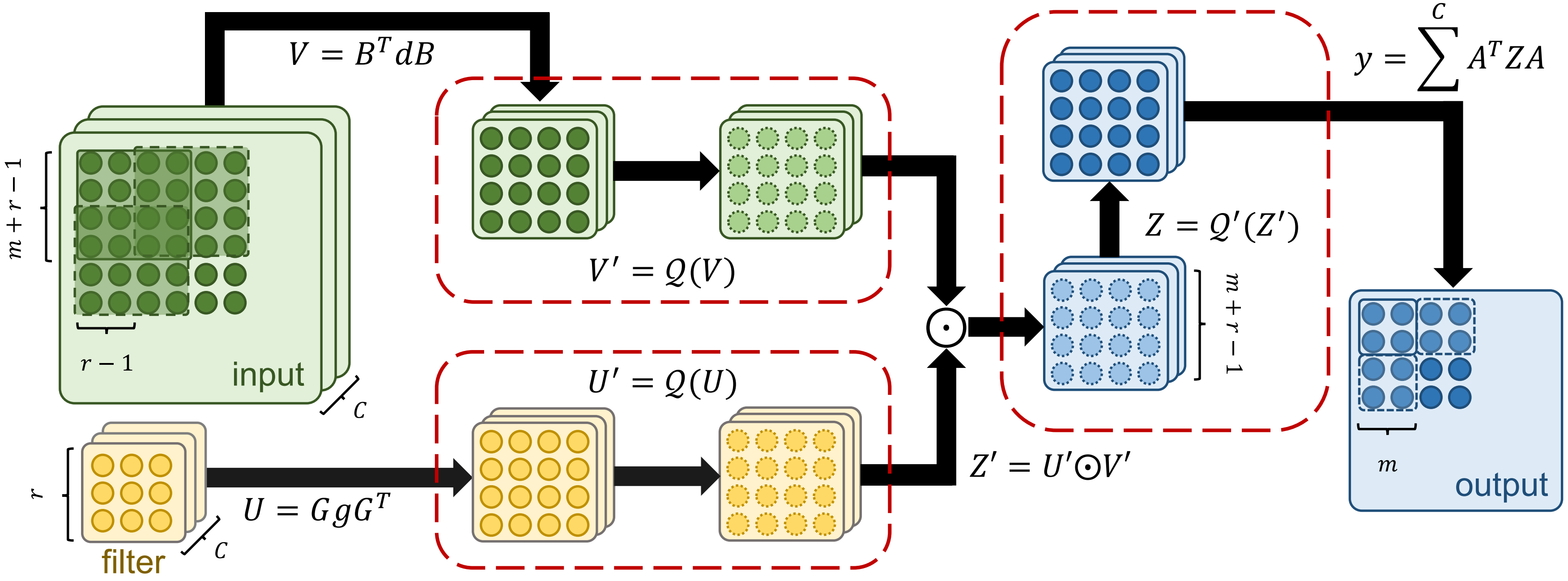
$$U'_{i8} = Q(U)$$

$$V'_{i8} = Q(V)$$

$$Z'_{i32} = U'_{i8} \odot V'_{i8}$$

$$Z_{f32} = Q'(M')$$

$$Y_{f32} = A^T Z_{f32} A$$



$$y = A^T[Q'([Q(GgG^T)] \odot [Q(B^T dB)])]A$$

Quantization Function and Calibration

$$Y = A^T[Q'([Q(GgG^T)]\odot[Q(B^TdB)])]A$$

- **Quantization Function**

$$Q(X_{\mathrm{FP32}}) = (\boxed{S_{\mathrm{INT8}}}(\alpha X_{\mathrm{FP32}}))_{\mathrm{INT8}}$$

Conversion with Saturation

$$S_{INT8}(x_{fp32}) = \min(\max(-128, round(x_{FP32})), 127))$$

Scaling Factor

$$\alpha = (2^{b-1} - 1)/\boxed{\tau}$$

- **De-Quantization Function**

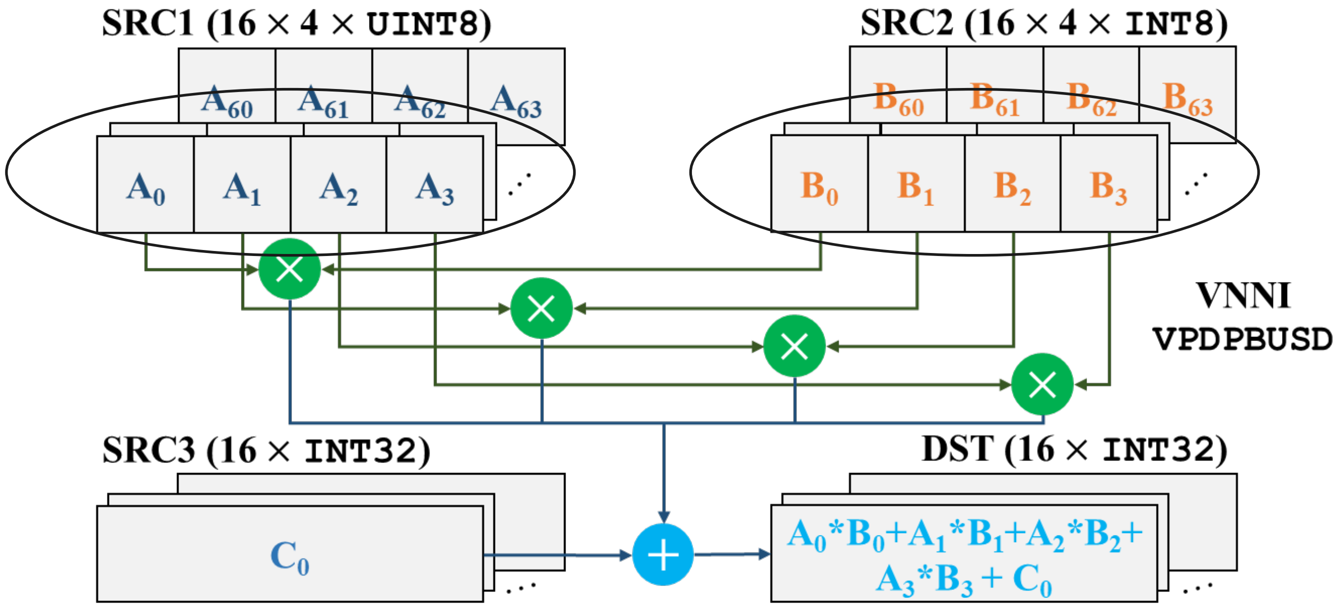
$$Q'(X_{\mathrm{INT8}}) = (\alpha^{-1} X_{\mathrm{INT8}})_{\mathrm{FP32}}$$

Calibration

$$\tau = \underset{\tau'}{arg\,min}\,(D_{KL}\,(P\,(X_{\mathrm{FP32}})\,||P\,(Q_{\tau'}\,(X_{\mathrm{FP32}}))))$$

Implementation and Optimization

To demonstrate our approach's effectiveness and efficiency, we implemented LoWino on Intel platforms by utilizing VNNI as the vehicle.

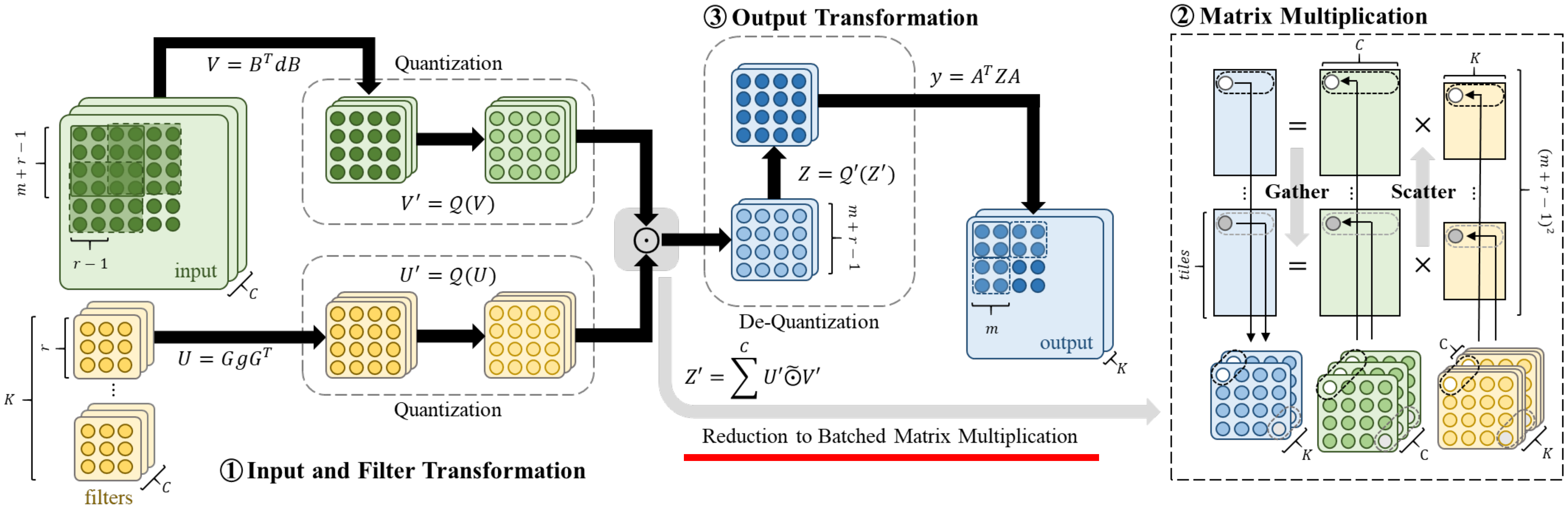


The semantic of the **vpdpbusd** instruction.

<u>Horizontal Accumulation</u>

Implementation and Optimization

1. **Input and Output Transformation**
2. **Matrix Multiplication**
3. **Output Transformation**



Implementation and Optimization

Challenges

- Extra memory operations containing non-consecutive memory access, e.g., scattering and gathering operations;
- Quantization and de-quantization overheads;
- Data type and layout requirements for low-precision computation instructions.

Our Design

- Overlapping computation and memory operations as much as possible;
- Reducing memory access latency;
- Increasing the computation efficiency through vectorization and data reuse.

Implementation and Optimization

Table 1: Customized Data Layout.

Variable	Data Layout
Input images	$B \times [C/\varphi/\sigma] \times H \times W \times \varphi \times \sigma$
Transformed inputs	$[N/N_{blk}] \times [C/C_{blk}] \times T \times N_{blk} \times C_{blk}$
Filters	$C \times [K/\varphi/\sigma] \times r \times r \times \varphi \times \sigma$
Transformed filters	$[C/C_{blk}] \times [K/K_{blk}] \times T \times [C_{blk}/\varphi] \times [K_{blk} \times \varphi]$
Transformed outputs	$B \times [K/\varphi/\sigma] \times N \times T \times \varphi \times \sigma$
Output images	$B \times [K/\varphi/\sigma] \times H' \times W' \times \varphi \times \sigma$

B (batch size), ***C*** (input channel), ***K*** (output channel)
H&W (input height & width), ***H'&W'*** (output height & width)
N (input tiles), ***T*** (Elements in Each single input tile)
N_blk, ***C_blk***, ***K_blk*** (bocking hyper-parameters)

VNNI-Specific:

$\boldsymbol{\sigma = 16}$

(the vector length of 32-bit word in a register)

$\boldsymbol{\varphi = 4}$

(the number of 8-bit elements in a 32-bit word)

Implementation and Optimization

Codelets Generation (using input transformation as an example)

Transformation Meta-Code

```
for i = 0 to phi:
  out[i][a] = -2 * in[i][1] - 1 * in[i][2] + 2 * in[i][3] + 1 * in[i][4];
  out[i][b] =  2 * in[i][1] - 1 * in[i][2] - 2 * in[i][3] + 1 * in[i][4];
```

Transformation Matrix

$$\begin{bmatrix} & & & \cdots & & & \\ 0 & -2 & -1 & 2 & 1 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 1 & 0 \\ & & & \cdots & & & \end{bmatrix}$$

Common Sub-Expression Elimination

```
for i = 0 to phi:
  temp[i]   = -1 * in[i][2] + 1 * in[i][4];
  out[i][a] = -2 * in[i][1] + 2 * in[i][3] + temp[i];
  out[i][b] =  2 * in[i][1] - 2 * in[i][3] + temp[i];
```

Loop Unrolling

```
  ...
  out[0][a] = -2 * in[0][1] + 2 * in[0][3] + temp[0];
  out[1][a] = -2 * in[1][1] + 2 * in[1][3] + temp[1];
  ...
```

Non-Temporal Store

Quantization and Non-Temporal Store

```
  ...
  int8_out[a][0] = saturate_cast<int8>(out[0][a] * alpha); FP32->INT8
  int8_out[a][1] = saturate_cast<int8>(out[1][a] * alpha); FP32->INT8
  ...
  non_temporal_store(next_step_input[a], int8_out[a]); // 64 * INT8
```

Implementation and Optimization

Batched Matrix Multiplication

- Two-Level Blocking (Cache and Register)

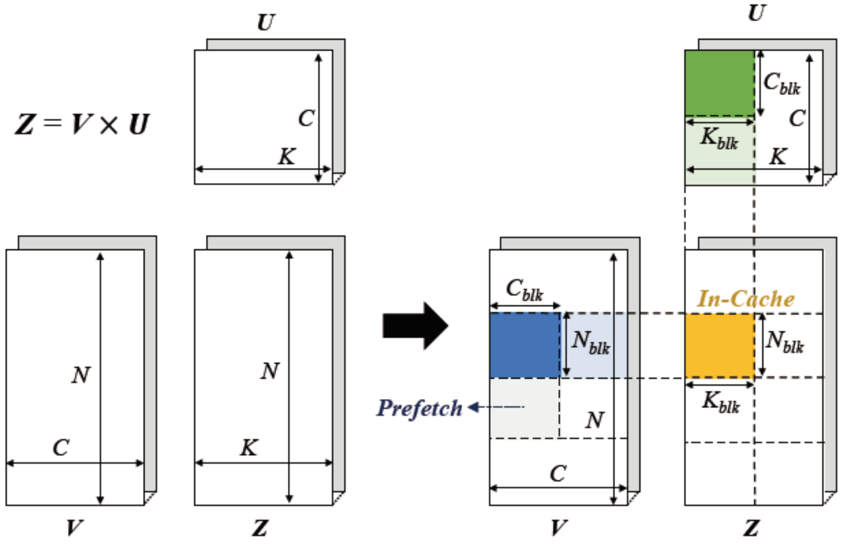


Figure 5: Cache-level matrix blocking strategy. The blocks in blue, green, and yellow represent the blocked sub-matrix of V, U and Z, respectively.

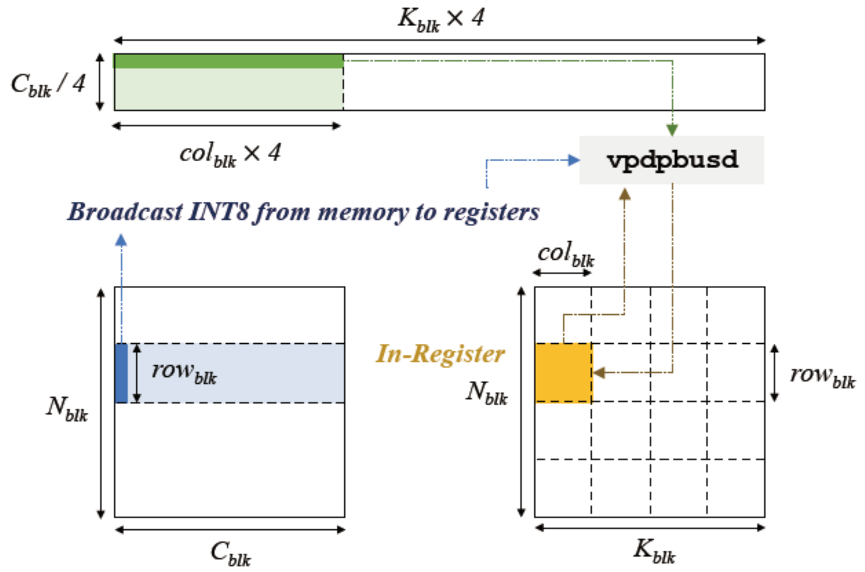


Figure 6: Register-level blocking. The blocks in blue, green, and yellow represent the blocked sub-matrix of v, u and z, respectively.

Implementation and Optimization

Batched Matrix Multiplication: Code Generation

```
for r0 = 0 to N_blk/col_blk:
  for c0 = 0 to K_blk/col_blk:
    for t = 0 to C_blk:                              ◁ unroll
      for r1 = 0 to row_blk:                         ◁ unroll
        v_reg = broadcast(v[r0+r1][t]);
        prefetch(next_v[r0+r1][t]);
        for c1 = 0 to col_blk:                       ◁ unroll
          u_reg[c1] = u[t][c0 + c1];
          z_reg[r1][c1] = vpdpbusd(z_reg[r1][c1],
                                   v_reg, u_reg[c1]);
    for r1 = 0 to row_blk:                           ◁ unroll
      for c1 = 0 to col_blk:                         ◁ unroll
        non_temporal_store(output[r0 + r1][c0 + c1],
                           m_regs[r1][c1]);
```

Figure 7: The pseudo-code for matrix multiplication.

Auto-Tuning

- Cache Blocking:

C_blk * K_blk < 512 * 512

- Register Blocking:

row_blk * col_blk + rol_blk < 31

Experimental Results

Specifically, we address two major research questions:
1) What is the performance of LoWino compared with the state-of-the-art implementations?
2) What is the end-to-end accuracy loss of our approach on representative convolutional neural networks?

CPU Platform:
8-Core Intel Xeon Scalable Processor (Cascade Lake)

Table 2: Benchmarked Convolutional Layers.

Layer	B	C	K	H&W	r
AlexNet_a	64	384	384	13	3
AlexNet_b	64	384	256	13	3
VGG16_a	64	256	256	58	3
VGG16_b	64	512	512	30	3
VGG16_c	64	512	512	16	3
ResNet-50_a	64	128	128	28	3
ResNet-50_b	64	256	256	14	3
ResNet-50_c	64	512	512	7	3
GoogLeNet_a	64	128	192	28	3
GoogLeNet_b	64	128	256	14	3
GoogLeNet_c	64	192	384	7	3
YOLOv3_a	1	64	128	64	3
YOLOv3_b	1	128	256	32	3
YOLOv3_c	1	256	512	16	3
FusionNet_a	1	128	128	320	3
FusionNet_b	1	256	256	160	3
FusionNet_c	1	512	512	80	3
U-Net_a	1	128	128	282	3
U-Net_b	1	256	256	138	3
U-Net_c	1	512	512	66	3

Experimental Results

Convolutional Layer Speedups

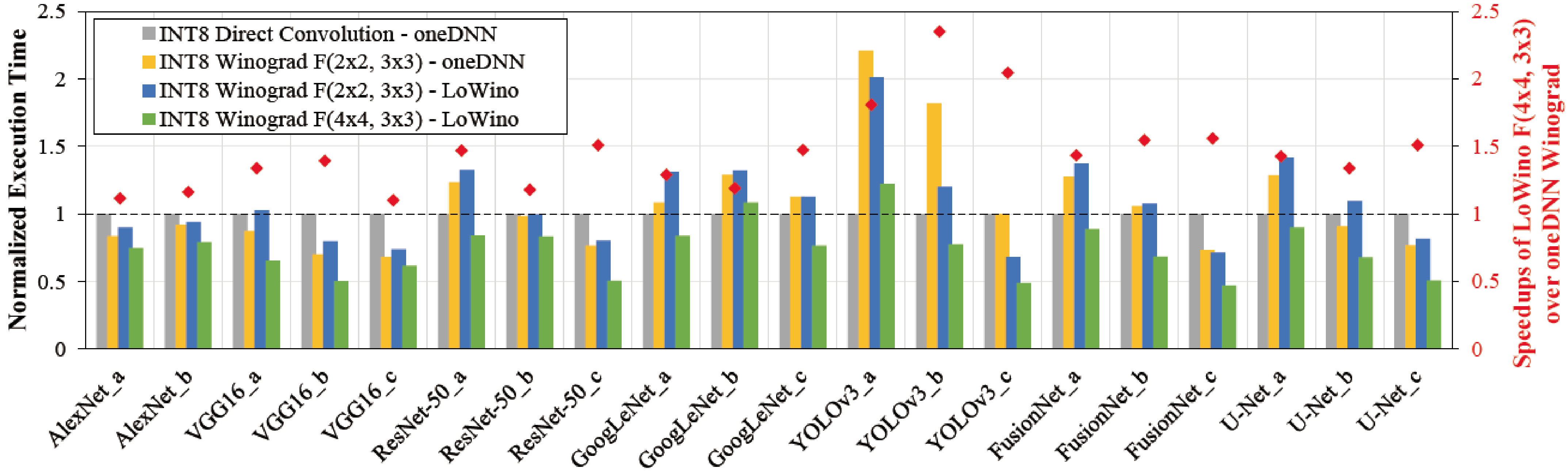


Figure 8: Normalized execution time for different convolution layers.

Our approach achieves an up to **2.04x** speedup and an average of **1.26x** speedup.

Experimental Results

Neural Network Accuracy

Table 3: The end-to-end top-1 accuracy of CNNs with our approach on the ImageNet dataset.

Model	Method		FP32 Acc. (%)	INT8 Acc. (%)
VGG16	Non-Winoragd Covolution	KLD [24]	69.40	69.20
		Yao *et al.* [38]	69.40	69.10
	F(2×2, 3×3)	oneDNN [10]	71.59	70.98
		LoWino (Ours)	71.59	71.33
	F(4×4, 3×3)	Down-Scaling Impl.	71.59	00.00
		LoWino (Ours)	71.59	69.20
ResNet-50	Non-Winoragd Convolution	KLD [24]	73.23	73.10
		Yao *et al.* [38]	75.30	74.80
		Jacob *et al.* [11]	76.40	74.90
		Park *et al.* [26]	77.72	75.67
		Krishnamoorthi [14]	75.20	75.10
	F(2×2, 3×3)	oneDNN [10]	76.13	75.91
		LoWino (Ours)	76.13	76.09
	F(4×4, 3×3)	Down-Scaling Impl.	76.13	00.00
		LoWino (Ours)	76.13	75.53

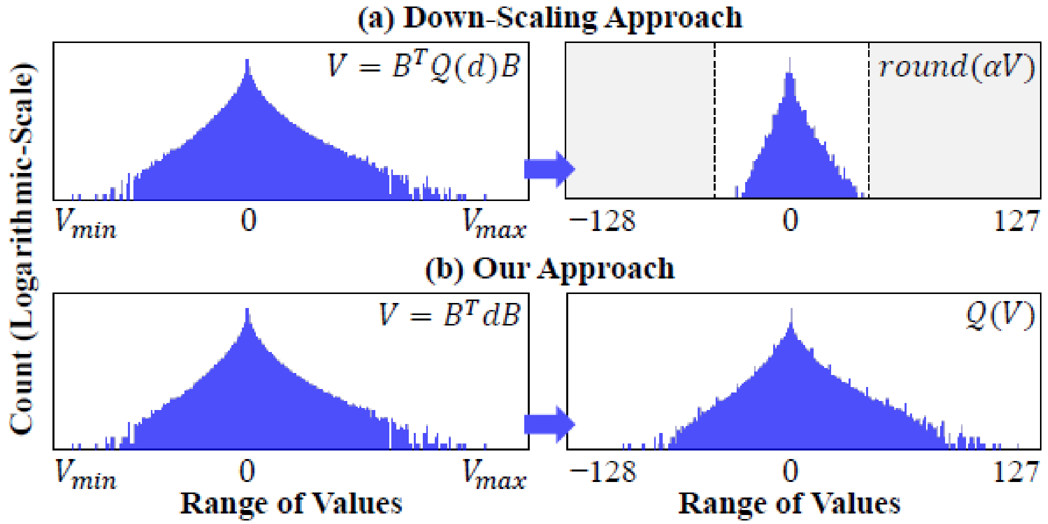
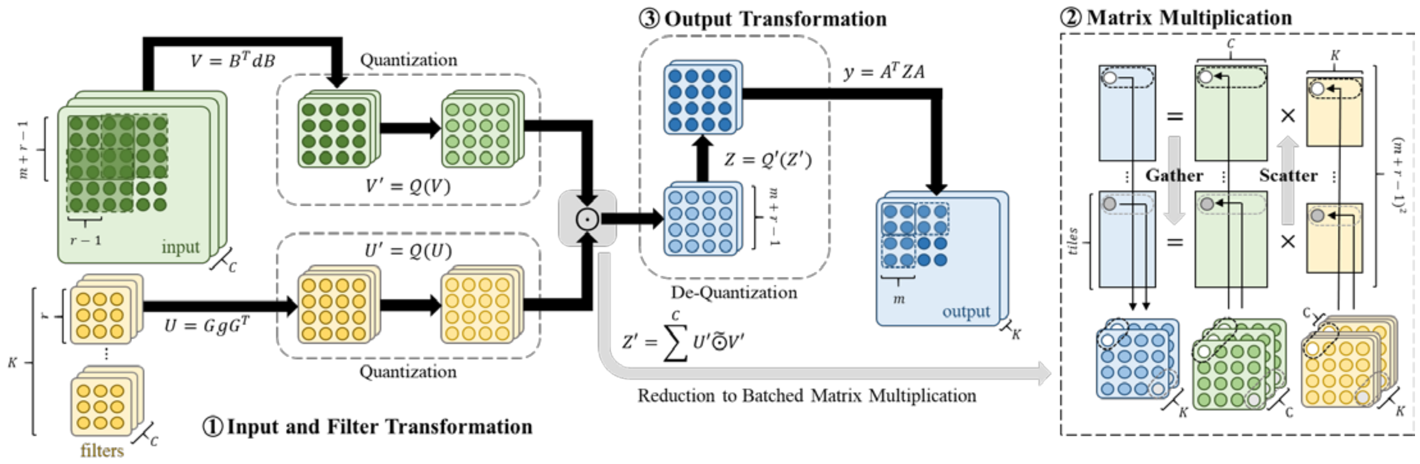


Figure 9: Comparing the down-scaling approach with ours for $F(4 \times 4, 3 \times 3)$ low-precision Winograd convolution.

Our approach can maintain the accuracy at an acceptable level for both F(2x2, 3x3) and F(4x4, 3x3).

Conclusion



- We propose **<u>a low-precision Winograd convolution approach</u>**, which introduces quantization in the transformed domain, thereby effectively exploiting the capability of low-precision computations while maintaining the accuracy at a reasonable level.
- We present **<u>an efficient implementation</u>** of low-precision Winograd convolutions on modern CPU platforms, which employs several well-designed optimization techniques to enhance the efficiency in both computation and memory access.

For more details, please refer to our paper.

Thank You

liguangli@ict.ac.cn