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

Guangli Li\textsuperscript{1,2}, Zhen Jia\textsuperscript{3}, Xiaobing Feng\textsuperscript{1,2}, Yida Wang\textsuperscript{3}

\textsuperscript{1} SKL of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, China
\textsuperscript{2} University of Chinese Academy of Sciences, China
\textsuperscript{3} Amazon Web Services, USA
Optimizing Convolution Operators

Low-Precision Computation

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: $g$

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

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

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

Multiplication
$$Z_{f32} = U_{f32} \bigodot V_{f32}$$

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

Output
$$y = \sum c A^T Z A$$

Input
$$V = B^T d B$$

Filter Trans
$$U = G g G^T$$

Input Trans
$$Z = U \bigodot V$$

Multiplication
$$Y = \sum c t Z A$$

Output
Motivating Examples

Brute-Force Approach

Input Tile: $d$
Filter: $g$

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

\[
\begin{align*}
U_{i8} &= Gg'_{i8}G^T \\
V_{i8} &= B^T d'_{i8} B \\
Z_{i32} &= U_{i8} \odot V_{i8} \\
y_{f32} &= A^T Z_{i32} A
\end{align*}
\]

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

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

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

Overflow: Out of Range!

increase values up to 4x
Motivating Examples

(a) Up-Casting Approach

\[ d' = Q(d) \]

\[ d'_{i8} \]

\[ B^T d'B\]

Multiplier (INT16)

\[ V_{i16} \]

\[ U_{i16} \]

\[ G g' G^T \]

\[ g'_{i8} \]

Cannot be calculated under int8!

(b) Down-Scaling Approach

\[ d' = Q(d) \]

\[ d'_{i8} \]

\[ B^T d'B\]

Multiplier (INT8)

\[ [\alpha V]_{i8} \]

\[ [\beta U]_{i8} \]

\[ G g' G^T \]

\[ g'_{i8} \]
Motivating Examples

(a) Up-Casting Approach

\[ d' = Q(d) \]

\[ d'_{i8} \]

Multiplication (INT16)

\[ B^T d' B \]

\[ V_{i16} \]

\[ U_{i16} \]

\[ g' = Q(g) \]

\[ g'_{i8} \]

(b) Down-Scaling Approach

\[ d' = Q(d) \]

\[ d'_{i8} \]

Multiplication (INT8)

\[ B^T d' B \]

\[ \alpha V_{i8} \]

\[ \beta U_{i8} \]

\[ g' = Q(g) \]

\[ g'_{i8} \]

Rounding Errors
## Existing Approaches

<table>
<thead>
<tr>
<th>Input Tile: $d$</th>
<th>Quantized Input Tile: $d' = Q(d)$</th>
<th>S1: Filter Trans</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter: $g$</td>
<td>Quantized Filter: $g' = Q(g)$</td>
<td>S2: Input Trans</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S3: Multiplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>S4: Output Trans</td>
</tr>
</tbody>
</table>

### Overflow

<table>
<thead>
<tr>
<th>$U_{f32} = Gg_{f32}G^T$</th>
<th>$U_{i8} = Gg'_{i8}G^T$</th>
<th>$U_{i16} = Gg'_{i8}G^T$</th>
<th>$U_{f32} = Gg'_{i8}G^T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{f32} = B^Td_{f32}B$</td>
<td>$V_{i8} = B^Td'_{i8}B$</td>
<td>$V_{i16} = B^Td'_{i8}B$</td>
<td>$V_{f32} = B^Td'_{i8}B$</td>
</tr>
<tr>
<td>$Z_{f32} = U_{f32} \circ V_{f32}$</td>
<td>$Z_{i32} = U_{i8} \circ V_{i8}$</td>
<td>$Z_{i32} = U_{i16} \circ V_{i16}$</td>
<td>$Z_{i32} = [\alpha U_{f32}]<em>{i8} \circ [\beta V</em>{f32}]_{i8}$</td>
</tr>
<tr>
<td>$y_{f32} = A^TZ_{f32}A$</td>
<td>$y_{f32} = A^TZ_{i32}A$</td>
<td>$y_{f32} = A^TZ_{i32}A$</td>
<td>$y_{f32} = A^T(\alpha^{-1}\beta^{-1}Z_{i32})A$</td>
</tr>
</tbody>
</table>

### Degradation

- **Full-Precision Approach**
- **Brute-Force Approach**
- **Up-Casting Approach**
- **Down-Scaling Approach**
LoWino: Quantization in the Winograd Domain

\[ U_{f32} = G g_{f32} G^T \]
\[ V_{f32} = B^T d_{f32} B \]
\[ U'_{i8} = Q(U) \]
\[ V'_{i8} = Q(V) \]
\[ Z'_{i32} = U'_{i8} \ominus V'_{i8} \]
\[ Z_{f32} = Q'(M') \]
\[ Y_{f32} = A^T Z_{f32} A \]

\[ y = A^T [Q'(Q(G g G^T)) \ominus [Q(B^T d B)]]] A \]
Quantization Function and Calibration

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

- Quantization Function

\[ Q(X_{FP32}) = (S_{INT8}(\alpha X_{FP32}))_{INT8} \]

- De-Quantization Function

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

Conversion with Saturation

\[ S_{INT8}(x_{FP32}) = \min(\max(-128,\text{round}(x_{FP32})),127)) \]

Scaling Factor

\[ \alpha = (2^{b-1} - 1) 1^b \]

Calibration

\[ \tau = \arg \min_{\tau'} (D_{KL} (P(X_{FP32}) \mid \mid P(Q_{\tau'}(X_{FP32})))) \]
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.
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.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Data Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input images</td>
<td>( B \times \left[ \frac{C}{\varphi/\sigma} \right] \times H \times W \times \varphi \times \sigma )</td>
</tr>
<tr>
<td>Transformed inputs</td>
<td>( \left[ \frac{N}{N_{blk}} \right] \times \left[ \frac{C}{C_{blk}} \right] \times T \times N_{blk} \times C_{blk} )</td>
</tr>
<tr>
<td>Filters</td>
<td>( C \times \left[ \frac{K}{\varphi/\sigma} \right] \times r \times r \times \varphi \times \sigma )</td>
</tr>
<tr>
<td>Transformed filters</td>
<td>( \left[ \frac{C}{C_{blk}} \right] \times \left[ \frac{K}{K_{blk}} \right] \times T \times \left[ \frac{C_{blk}}{\varphi} \right] \times \left[ K_{blk} \times \varphi \right] )</td>
</tr>
<tr>
<td>Transformed outputs</td>
<td>( B \times \left[ \frac{K}{\varphi/\sigma} \right] \times N \times T \times \varphi \times \sigma )</td>
</tr>
<tr>
<td>Output images</td>
<td>( B \times \left[ \frac{K}{\varphi/\sigma} \right] \times H' \times W' \times \varphi \times \sigma )</td>
</tr>
</tbody>
</table>

**VNNI-Specific:**

- \( \sigma = 16 \)
  (the vector length of 32-bit word in a register)
- \( \varphi = 4 \)
  (the number of 8-bit elements in a 32-bit word)

**Variables:**

- \( B \) (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)
- \( N_{blk}, C_{blk}, K_{blk} \) (blocking hyper-parameters)
**Implementation and Optimization**

**Codelets Generation** (using input transformation as an example)

<table>
<thead>
<tr>
<th>Transformation Matrix</th>
<th>Transformation Meta-Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 -2 -1  2 1 1 0</td>
<td>for i = 0 to phi:</td>
</tr>
<tr>
<td></td>
<td>out[i][a] = -2 * in[i][1] - 1 * in[i][2] + 2 * in[i][3] + 1 * in[i][4];</td>
</tr>
<tr>
<td></td>
<td>out[i][b] = 2 * in[i][1] - 1 * in[i][2] - 2 * in[i][3] + 1 * in[i][4];</td>
</tr>
<tr>
<td>...</td>
<td>temp[i] = -1 * in[i][2] + 1 * in[i][4];</td>
</tr>
<tr>
<td>0  2 -1 -2 1 1 0</td>
<td>out[i][a] = -2 * in[i][1] + 2 * in[i][3] + temp[i];</td>
</tr>
<tr>
<td>...</td>
<td>out[i][b] = 2 * in[i][1] - 2 * in[i][3] + temp[i];</td>
</tr>
</tbody>
</table>

| Common Sub-ExpressionElimination |  |
|----------------------------------|  |
| 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]; |  |
| ...            |  |

| 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)

![Diagram of cache-level matrix blocking strategy](image1)

![Diagram of register-level blocking](image2)

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

```plaintext
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]);
        u_reg[c1] = u[t][c0 + c1];
        z_reg[r1][c1] = vpdpbusd(z_reg[r1][c1],
                                v_reg, u_reg[c1]);
      for c1 = 0 to col_blk:  # unroll
        non_temporal_store(output[r0 + r1][c0 + c1],
                           m_regs[r1][c1]);
```

Auto-Tuning

- Cache Blocking: $C_{blk} \times K_{blk} < 512 \times 512$
- Register Blocking: $row_{blk} \times col_{blk} + rol_{blk} < 31$

Figure 7: The pseudo-code for matrix multiplication.
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>
<thead>
<tr>
<th>Layer</th>
<th>B</th>
<th>C</th>
<th>K</th>
<th>H&amp;W</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet_a</td>
<td>64</td>
<td>384</td>
<td>384</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>AlexNet_b</td>
<td>64</td>
<td>384</td>
<td>256</td>
<td>13</td>
<td>3</td>
</tr>
<tr>
<td>VGG16_a</td>
<td>64</td>
<td>256</td>
<td>256</td>
<td>58</td>
<td>3</td>
</tr>
<tr>
<td>VGG16_b</td>
<td>64</td>
<td>512</td>
<td>512</td>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>VGG16_c</td>
<td>64</td>
<td>512</td>
<td>512</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>ResNet-50_a</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>ResNet-50_b</td>
<td>64</td>
<td>256</td>
<td>256</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>ResNet-50_c</td>
<td>64</td>
<td>512</td>
<td>512</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>GoogLeNet_a</td>
<td>64</td>
<td>128</td>
<td>192</td>
<td>28</td>
<td>3</td>
</tr>
<tr>
<td>GoogLeNet_b</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>14</td>
<td>3</td>
</tr>
<tr>
<td>GoogLeNet_c</td>
<td>64</td>
<td>192</td>
<td>384</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>YOLOv3_a</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>64</td>
<td>3</td>
</tr>
<tr>
<td>YOLOv3_b</td>
<td>64</td>
<td>128</td>
<td>256</td>
<td>32</td>
<td>3</td>
</tr>
<tr>
<td>YOLOv3_c</td>
<td>64</td>
<td>256</td>
<td>512</td>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>FusionNet_a</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>320</td>
<td>3</td>
</tr>
<tr>
<td>FusionNet_b</td>
<td>64</td>
<td>256</td>
<td>256</td>
<td>160</td>
<td>3</td>
</tr>
<tr>
<td>FusionNet_c</td>
<td>64</td>
<td>512</td>
<td>512</td>
<td>80</td>
<td>3</td>
</tr>
<tr>
<td>U-Net_a</td>
<td>64</td>
<td>128</td>
<td>128</td>
<td>282</td>
<td>3</td>
</tr>
<tr>
<td>U-Net_b</td>
<td>64</td>
<td>256</td>
<td>256</td>
<td>138</td>
<td>3</td>
</tr>
<tr>
<td>U-Net_c</td>
<td>64</td>
<td>512</td>
<td>512</td>
<td>66</td>
<td>3</td>
</tr>
</tbody>
</table>
Experimental Results

Convolutional Layer Speedups

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

Figure 8: Normalized execution time for different convolution layers.
Experimental Results

Neural Network Accuracy

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

<table>
<thead>
<tr>
<th>Model</th>
<th>Method</th>
<th>FP32 Acc. (%)</th>
<th>INT8 Acc. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGG16</td>
<td>Non-Winograd Convolution</td>
<td>KLD [24]</td>
<td>69.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yao et al. [38]</td>
<td>69.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oneDNN [10]</td>
<td>71.59</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LoWino (Ours)</td>
<td>71.59</td>
</tr>
<tr>
<td></td>
<td>Down-Scaling Impl.</td>
<td></td>
<td>71.59</td>
</tr>
<tr>
<td></td>
<td>LoWino (Ours)</td>
<td></td>
<td>71.59</td>
</tr>
<tr>
<td>ResNet-50</td>
<td>Non-Winograd Convolution</td>
<td>KLD [24]</td>
<td>73.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Yao et al. [38]</td>
<td>75.30</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Jacob et al. [11]</td>
<td>76.40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Park et al. [26]</td>
<td>77.72</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Krishnamoorthi [14]</td>
<td>75.20</td>
</tr>
<tr>
<td></td>
<td>oneDNN [10]</td>
<td></td>
<td>76.13</td>
</tr>
<tr>
<td></td>
<td>LoWino (Ours)</td>
<td></td>
<td>76.13</td>
</tr>
<tr>
<td></td>
<td>Down-Scaling Impl.</td>
<td></td>
<td>76.13</td>
</tr>
<tr>
<td></td>
<td>LoWino (Ours)</td>
<td></td>
<td>76.13</td>
</tr>
</tbody>
</table>

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 a low-precision Winograd convolution approach, 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 an efficient implementation 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.
LoWino: Towards Efficient Low-Precision Winograd Convolutions on Modern CPUs

Thank You

liguangli@ict.ac.cn