Optimizing Highly Memory Bound Data Transfer for Fast Convolutional Neural Network

Yolanda Chen, Intel Developer Products Division
Background

• Winograd’s mini filter algorithm\(^1\) is used as fast algorithm in deep convolutional neural 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

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

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**
Formula for Convnet Layer

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

\[ Y_{t,f} \equiv \sum Y_{n,t,f,c} \]
\[ = \sum A^T [ \sum U_{n,t,c} \odot V_{c,f} ] A \]
\[ = A^T [ \sum U_{n,t,c} \odot V_{c,f} ] A \]

\[ U = GgG^T \]
\[ V = B^T dB \]

Filter Transfer

Input Data Transfer
Fast convolution based on Winograd’s minimal filter

F(2*2, 3*3)

Filter: 3 * 3 (R*R)

Direct convolution stride 1

Input transfer

Filter Transfer

GEMM

Out Transfer

C = 3, 64, ...

m=2

4*4
(M+R-1) * (M+R-1)
AVX512 Vectorization for Input Transfer

```c
#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*iorws + j] - data[t*iorws + j + 2]; // d0 - d2
        temp[t * 4 + 1] = data[t*iorws + j + 1] + data[t*iorws + j + 2]; // d1 + d2
        temp[t * 4 + 2] = data[t*iorws + j + 2] - data[t*iorws + j + 1]; // d2 - d1
        temp[t * 4 + 3] = data[t*iorws + j + 1] - data[t*iorws + j + 3]; // d1 - d3
    }
}
```

```
Load 16 (index 0, 1, 2, .. 15)
vmovups (%r10), %zmm16
```

```
Permute 16 (index 0, 2, 4, ..., 30)
vpermi2ps 64(%r10), %zmm16, %zmm20
```

```
j +=2 (stride 2)
```
AVX512 Vectorization for Input Transfer

```c
#include <x86intrin.h>

#pragma omp simd linear(j:2, count:1) private(temp)
for (j = 0; j < W; j += 2, count++)
{
    // ID: 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

<table>
<thead>
<tr>
<th>#Instructions</th>
<th>Scalar</th>
<th>AVX2 (256-bit)</th>
<th>AVX-512 (512-bit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads (hit) to input data</td>
<td>4 * 61</td>
<td>4 * 5</td>
<td>4 * 1</td>
</tr>
<tr>
<td>Loads (miss) to input data</td>
<td>4 * 3</td>
<td>4 * 3</td>
<td>4 * 3</td>
</tr>
<tr>
<td>SP adds</td>
<td>16 * 32</td>
<td>2 * 32</td>
<td>32</td>
</tr>
<tr>
<td>Stores (hit) to output data</td>
<td>16 * 15</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Stores (miss) to output data</td>
<td>16</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>Total (Reduction)</td>
<td>1024 (x1)</td>
<td>128 (x8)</td>
<td>64 (x16)</td>
</tr>
</tbody>
</table>
Continuous Loads and Streaming Stores

Continuous loads

Row0 | Row1 | Row2 | Row3 | ... | RowW-3 | RowW-2 | RowW-1 | RowW

Temporary Buffer
Reused Many Times

Real speedup: 1.6 - 2.7x

#pragma vector nontemporal

Streaming store to memory
Inefficient GEMM for Small Channel Size

F(2*2, 3*3)

Tile: 4*4  
(M+R-1) * (M+R-1)

Input transfer

Direct convolution 
stride 1

Filter: 3 * 3 (R*R)

Filter Transfer

Tile Filter 
Data 
16 * C * K

Tile Data 
16 * batch * 
C * H/2 * W/2

GEMM

Output 
16 * batch * 
H/2 * W/2 * K

Out Transfer
Inefficient GEMM for Small Channel Size

F(2*2, 3*3)

Tile: 4*4
(M+R-1) * (M+R-1)

Tile Data Buffer
16 * C * W/2

Input transfer

Direct convolution
stride 1

Filter: 3 * 3 (R*R)

Filter Transfer

Manual GEMM
with Small Buffer

Convnet Layer
speedup for C=3
3.6x

16 * C * K

Tile Filter
Data

Out Transfer
Convolution of VGG Net layers in FALCON and Intel MKL on Intel Xeon Phi 7250 processor (flat MCDRAM, quadrant mode)
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

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

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

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.

Copyright © 2015, Intel Corporation. All rights reserved. Intel, Pentium, Xeon, Xeon Phi, Core, VTune, Cilk, and the Intel logo are trademarks of Intel Corporation in the U.S. and other countries.

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