Integer GEMM
(under)performance

Marat Dukhan
Software Engineer on Caffe 2
GEMM in Neural Networks

- Fully-connected layers
- im2col+GEMM algorithm for convolution
- 1x1 convolutional layers
# Android CPU Landscape

## Overview of CPU microarchitectures

<table>
<thead>
<tr>
<th></th>
<th>Low-End</th>
<th>Mid-End</th>
<th>High-End</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ARMv7</strong></td>
<td>Cortex-A5</td>
<td>Cortex-A8, Cortex-A9</td>
<td>Cortex-A12, Cortex-A15, Cortex-A17 Krait</td>
</tr>
<tr>
<td><strong>Cortex-A7</strong></td>
<td>Cortex-A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>ARMv8</strong></td>
<td>Cortex-A53, Cortex-A55</td>
<td>Cortex-A8, Cortex-A9</td>
<td>Cortex-A57, Cortex-A72, Cortex-A73 Kryo Mongoose</td>
</tr>
<tr>
<td><strong>Cortex-A7</strong></td>
<td>Cortex-A7</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Cortex-A9</strong></td>
<td>Cortex-A9</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Krait</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Kryo</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Mongoose</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Android CPU Landscape
Overview of low-end microarchitecture

- **Cortex-A7**
  - 64-bit SIMD units for load/store and integer SIMD
  - NEON FP32 instructions run at 1 element/cycle (i.e. scalar execution)
  - Single-issue NEON pipeline

- **Cortex-A53**
  - 64-bit SIMD load units
  - 128-bit integer and floating-point SIMD compute and store units
  - Single-issue NEON pipeline, but with useful co-issue capabilities
    - Co-issue for NEON compute + general-purpose load
    - Co-issue for NEON 64-bit load + 64-bit move to NEON co-processor
SGEMM for mobile low-end

ARM NEON µkernel

• Load **MR** elements of **A** panel
• Load **NR** elements of **B** panel
• Use vector-scalar multiply-accumulate instruction (VMLA.F32 Qd, Qn, Qm[x]) to compute a block of **C**
• Optimal **MR** x **NR** blocks:
  • Cortex-A7: **6x6** (**6x8** is marginally worse)
  • Cortex-A53: **6x8**
SGEMM
Example of 6x8 ARM NEON µkernel

VLD1.32 {d0–d2}, [rA]!
VLD1.32 {q2–q3}, [rB]!

# 6x2 = 12 VMLA.F32 instructions
VMLA.F32 q4, q2, d0[0]
VMLA.F32 q5, q3, d0[0]
VMLA.F32 q6, q2, d0[0]
VMLA.F32 q7, q3, d0[0]
... repeat for d0[1]...d2[1]
Integer GEMM

Background

- CNNs are very tolerant to quantization noise
- Little accuracy loss with 8-bit quantization
- **Idea**: instead of a single FP32, process 4 8-bit ints
- **Theory**: 4x speedup on SIMD!
- **Implementation**: Google’s gemmlowp library
Integer GEMM
Implementation with vector-scalar multiply-accumulate

- NEON VMLAL instruction does not have a .U8 version
- Need to extend data to uint16 (VMOVL.U8) for VMLAL.U16
  - Loading uint16 data may be faster on some µarchitectures
- Two instructions cripple performance
  - VMOVL.U8 instructions, not needed in FP32 version
  - VMLAL.U16 accumulates to uint32, does only 4 MACs
U8GEMM
Example of 6x8 ARM NEON µkernel

VLD1.32 {d0}, [rA]!
VMOVL.U8 q0, d0 # extend to uint16
VLD1.32 {d1}, [rB]!
VMOVL.U8 q1, d2 # extend to uint16

VMLAL.U16 q2, d2, d0[0] # multiply–accumulate in uint32
VMLAL.U16 q3, d3, d0[0] # multiply–accumulate in uint32
... repeat for d0[1]...d1[1]
**Integer GEMM**

Implementation with vector-vector multiply-accumulate

- **Idea (gemmlowp):** use vector-vector **VMLAL.U8**
- First, **VMULL.U8 Qd, Dm, Dn** to multiply to uint16
- Then, **VPADAL.U16** to accumulate to uint32
- This µkernel assumes 8 kc values are packed sequentially
- Still problematic w.r.t performance
  - Two instructions instead of one
  - **VPADAL.U16** accumulates to uint32, outputs 4 values/cycle
  - **VPADAL.U16** is slow on low-end cores
U8GEMM
Example of 3x8 X 8x3 ARM NEON µkernel (gemmlowp)

VLD1.32 {d0-d2}, [rA]!
VLD1.32 {d4-d6}, [rB]!

VMULL.U8 q4, d0, d4 # multiply to uint16
VMULL.U8 q5, d0, d5 # multiply to uint16
VMULL.U8 q6, d0, d6 # multiply to uint16

VPADAL.U16 q7, q4 # accumulate to uint32
VPADAL.U16 q8, q5 # accumulate to uint32
VPADAL.U16 q9, q6 # accumulate to uint32

# repeat for d1...d2
**Integer GEMM**

Implementation with signed vector-vector multiply-accumulate

- **Idea (gemmlowp):** \(a_1 \times b_1 + a_2 \times b_2\) fits into int16 if we restrict either \(a\)s or \(b\)s to \([-127, 127]\)
- First, **VMULL.S8 Qd, Dm, Dn** to multiply to int16
- Then, **VMLAL.S8 Qd, Dm, Dn** to multiply-accumulate in int16
- Then, **VPADAL.S16** to accumulate to uint32
- This µkernel assumes 16 \(kc\) values are packed sequentially
- Slightly improves performance
- Expensive **VPADAL** is amortized between two **VMULLs**
Example of 4x16 X 16x2 ARM NEON μkernel (gemmlowp)

VLD1.32 {d0–d2}, [rA]!
VLD1.32 {d4–d7}, [rB]!

VMULL.S8 q4, d0, d4 # multiply
VMLAL.S8 q4, d1, d5 # multiply-accumulate in int16
VPADAL.S16 q7, q4, q0 # accumulate to int32

... repeat for 4x2 tile of NEON registers
## Performance

Measured and estimated OPS/cycle

<table>
<thead>
<tr>
<th>Operation Description</th>
<th>Cortex-A7</th>
<th>Cortex-A53</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM 6x6 (FB impl): FLOPS/cycle measured</td>
<td>1.619</td>
<td>5.888</td>
</tr>
<tr>
<td>SGEMM 6x8 (FB impl): FLOPS/cycle measured</td>
<td>1.613</td>
<td>6.000</td>
</tr>
<tr>
<td>SGEMM 6x8 (FB impl): FLOPS/cycle estimated</td>
<td>1.745</td>
<td>6.56</td>
</tr>
<tr>
<td>U8GEMM 6x4 X 4x8 (FB impl): OPS/cycle est.</td>
<td>3.03</td>
<td>6.74</td>
</tr>
<tr>
<td>7x VLDR Dd, [Rn, #imm]</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>7x VMOVL.U8 Qd, Rm</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>48x VMLAL.U16 Qd, Qn, Qm[1]</td>
<td>106</td>
<td>48</td>
</tr>
<tr>
<td>U8GEMM 3x8 X 8x3 (gemmlowp): OPS/cycle est.</td>
<td>2.40</td>
<td>4.80</td>
</tr>
<tr>
<td>6x VLDR Dd, [Rn, #imm]</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>9x VMULL.U8 Qd, Dn, Dm</td>
<td>18</td>
<td>9</td>
</tr>
<tr>
<td>9x VPAAL.U16 Qd, Qn, Qm</td>
<td>32</td>
<td>18</td>
</tr>
<tr>
<td>I8GEMM 4x16 X 16x2 (gemmlowp): OPS/cycle est.</td>
<td>3.30</td>
<td>6.74</td>
</tr>
<tr>
<td>12x VLDR Dd, [Rn, #imm]</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>8x VMLAL.S8 Qd, Dn, Dm</td>
<td>17.6*</td>
<td>8</td>
</tr>
<tr>
<td>8x VMULL.S8 Qd, Dn, Dm</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>8x VPAAL.S16 Qd, Qn, Qm</td>
<td>32</td>
<td>16</td>
</tr>
</tbody>
</table>
Performance Analysis

Int8 GEMM vs SGEMM on low-end ARM cores:

- 2x speedup on Cortex-A7 (due to slow FP units)
- At most 10% speedup on Cortex-A53

Why small speedups?

- Accumulation to int32 is expensive
- No dual-issue of **VMUL + VPADAL** on low-end
Performance
Instruction set effects

Lack of instructions to multiply and accumulate neighboring lanes to 32 bits is what kills performance.

- Scalar **SMLASD** existed in ARMv6, but no NEON version
- Instruction like **DP4A** (nVidia Pascal) would be helpful

<table>
<thead>
<tr>
<th></th>
<th>Cortex-A7</th>
<th>Cortex-A53</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGEMM 6x6 (FB impl): FLOPS/cycle measured</td>
<td>1.619</td>
<td></td>
</tr>
<tr>
<td>SGEMM 6x8 (FB impl): FLOPS/cycle measured</td>
<td>1.613</td>
<td>5.888</td>
</tr>
<tr>
<td>U8GEMM 6x4 X 4x8 (NEON DP4A): OPS/cycle est.</td>
<td>12.39</td>
<td>24.77</td>
</tr>
<tr>
<td>U8GEMM 6x4 X 4x8 (NEON SMLASD): OPS/cycle est.</td>
<td>6.98</td>
<td>13.96</td>
</tr>
</tbody>
</table>
Conclusion

- 8-bit Integer GEMM promised great speedups, but in practice doesn’t deliver where we need them most - on low-end mobile phones
- This fact is due to a combination of ARM NEON ISA limitations and single-issue NEON pipelines
- 4x speedups could be realized if ARM NEON included a 4x 8-bit int dot product with 32-bit accumulation