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

	Low-End	Mid-End	High-End
ARMv7	Cortex-A5 Cortex-A7	Cortex-A8 Cortex-A9	Cortex-A12 Cortex-A15 Cortex-A17 Krait
ARMv8	Cortex-A53 Cortex-A55		Cortex-A57 Cortex-A72 Cortex-A73 Kryo Mongoose

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: **6**x**6** (**6**x**8** is marginally worse)
  - Cortex-A53: **6**x**8**

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 * **b**1 + **a**2 * **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 **VMULL**s

I8GEMM

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

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

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

	Cortex-A7	Cortex-A53
SGEMM 6x6 (FB impl): FLOPS/cycle measured	1.619	
SGEMM 6x8 (FB impl): FLOPS/cycle measured	1.613	5.888
U8GEMM 6x4 X 4x8 (NEON DP4A): OPS/cycle est.	12.39	24.77
U8GEMM 6x4 X 4x8 (NEON SMLASD): OPS/cycle est.	6.98	13.96

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