Using NEON for Parallel Data Processing

Zynq-7000
Hardware Architecture

Speaker: Leon Qin
Title: Processor Specialist
Date: Oct, 2012
ARM Architecture Evolution

Key Technology Additions by Architecture Generation

Execution Environments: Improved memory use

Improved Media and DSP

ARM9
ARM10

ARM11

Thumb®-2
TrustZone™
SIMD

VFPv2
Jazelle®

V5
V6

VFPv3

NEON™ Adv SIMD

Thumb-EE

Thumb-2 Only

V7 A&R

V7 M
Why NEON?

- General purpose SIMD processing useful for many applications
- Supports widest range multimedia codecs used for internet applications
  - Many soft codec standards: MPEG-4, H.264, On2 VP6/7/8, Real, AVS, …
  - Ideal solution for normal size ‘internet streaming’ decode of various formats
- Fewer cycles needed
  - Neon will give 60-150% performance boost on complex video codecs
  - Simple DSP algorithms can show larger performance boost (4x-8x)
  - Balance of computation and memory access is required
  - Processor can sleep sooner => overall dynamic power saving
Why NEON?

NEON is a mature advanced SIMD technology.
- SIMD exist on many 32-bit arch
  • PowerPC has AltiVec, while x86 has MMX/SSE/AVX
- Can significantly accelerate parallelable repetitive operations on large data sets.

Beneficial to many DSP or multimedia algorithms
- Clean orthogonal vector architecture, applicable to a wide range of data intensive computation
- audio, video, and image processing codecs.
- Not just for codecs – also applicable to 2D & 3D graphics etc
- Color-space conversion.
- Physics simulations.
- Error correction(such as Reed Solomon codecs, CRCs), elliptic curve cryptography, etc.
NEON vs. DSP/FPGA offload

NEON advantages
- Easy programming & debug
- Fully coherent with CPU, no cache maintenance operations
- Part of ARM arch - no hardware or software integration required
- Ecosystem support off-the-shelf, no porting required

DSP/FPGA advantages
- Runs parallel with CPU, few CPU cycles required
- More ‘realtime’ - no OS/cache variability
- Fixed function or limited codec support
- Potentially higher performance (e.g. 1080p Full HD video)
NEON Agenda

- NEON Hardware overview
- NEON Instruction set overview
- NEON Software Support
- NEON improves performance
What is NEON?

- NEON is a wide SIMD data processing architecture
  - Extension of the ARM instruction set
  - 32 registers, 64-bits wide (dual view as 16 registers, 128-bits wide)

- NEON Instructions perform “Packed SIMD” processing
  - Registers are considered as vectors of elements of the same data type
  - Data types can be: signed/unsigned 8-bit, 16-bit, 32-bit, 64-bit, single prec. float
  - Instructions perform the same operation in all lanes
### Example SIMD instruction – Vector ADD

- **Larger register size**
  - Register split into equal size any type elements
  - Operation performed on same element of each register
  - `VADD.U16 D2, D1, D0`

<table>
<thead>
<tr>
<th></th>
<th>63</th>
<th>47</th>
<th>31</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x1001</td>
<td>0x1234</td>
<td>0x7</td>
<td>0xAB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>+</td>
<td>+</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xFF0</td>
<td>0x5678</td>
<td>0xFFF8</td>
<td>0xCD</td>
<td></td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>=</td>
<td>=</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x1FF1</td>
<td>0x68AC</td>
<td>0xFFFF</td>
<td>0x178</td>
<td></td>
</tr>
</tbody>
</table>

\[ D0 = D2 + D1 \]
\[ D2 = 0x1FF1 + 0x68AC \]
\[ D0 = 0x1001 + 0x1234 \]
\[ D1 = 0xFF0 + 0x5678 \]
\[ D0 = 0x7 + 0xAB \]
\[ D1 = 0xFFF8 + 0xCD \]
\[ D0 = 0x15 \]
\[ D1 = 0x47 \]
\[ D2 = 0x31 \]
Neon Data Types

- NEON natively supports a set of common data types
  - Integer and Fixed-Point: 8-bit, 16-bit, 32-bit and 64-bit
  - 32-bit Single-precision Floating-point; 8 and 16-bit polynomial

<table>
<thead>
<tr>
<th>Size</th>
<th>Type</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>S8</td>
<td>.S8</td>
</tr>
<tr>
<td></td>
<td>U8</td>
<td>.U8</td>
</tr>
<tr>
<td></td>
<td>P8</td>
<td>.P8</td>
</tr>
<tr>
<td>16</td>
<td>S16</td>
<td>.S16</td>
</tr>
<tr>
<td></td>
<td>U16</td>
<td>.U16</td>
</tr>
<tr>
<td></td>
<td>P16</td>
<td>.P16</td>
</tr>
<tr>
<td>32</td>
<td>S32</td>
<td>.S32</td>
</tr>
<tr>
<td></td>
<td>U32</td>
<td>.U32</td>
</tr>
<tr>
<td></td>
<td>F32</td>
<td>.F32</td>
</tr>
<tr>
<td>64</td>
<td>S64</td>
<td>.S64</td>
</tr>
<tr>
<td></td>
<td>U64</td>
<td>.U64</td>
</tr>
</tbody>
</table>

- Data types are represented using a bit-size and format letter
  - VADD.U16 D2, D1, D0

- Not all data types available in all sizes
NEON Registers

- **NEON provides a 256-byte register file**
  - Distinct from the core registers
  - Extension to the VFPv2 register file (VFPv3)

- **Two explicitly aliased views**
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)

- **Enables register trade-off**
  - Vector length
  - Available registers

- **Also uses the summary flags in the VFP FPSCR**
  - Adds a QC integer saturation summary flag
  - No per-lane flags, so ‘carry’ handled using wider result (16bit+16bit -> 32-bit)
Vectors and Scalars

- **Registers hold one or more elements of the same data type**
  - $V_n$ can be used to reference either a 64-bit $D_n$ or 128-bit $Q_n$ register
  - A register, data type combination describes a vector of elements

- **Some instructions can reference individual scalar elements**
  - Scalar elements are referenced using the array notation $V_n[x]$
NEON Agenda

- NEON Hardware overview
- NEON Instruction set overview
- NEON Software Support
- NEON improves performance
Instruction Syntax

\[ \text{V} \langle \text{mod} \rangle \langle \text{op} \rangle \{ \langle \text{shape} \rangle \} \{ \langle \text{cond} \rangle \}\{ . \langle \text{dt} \rangle \} (\langle \text{dest} \rangle), \text{src1, src2} \]

**<mod> - Instruction Modifiers**
- Q indicates the operation uses saturating arithmetic (e.g. VQADD)
- H indicates the operation halves the result (e.g. VHADD)
- D indicates the operation doubles the result (e.g. VQDMUL)
- R indicates the operation performs rounding (e.g. VRHADD)

**<op> - Instruction Operation** (e.g. ADD, MUL, MLA, MAX, SHR, SHL, MOV)

**<shape> - Shape**
- L – The result is double the width of both operands
- W – The result and first operand are double the width of the last operand
- N – The result is half the width of both operands

**<cond> - Conditional, used with IT instruction**

**<dt> - Data type**

**<dest> - Destination, <src1> - Source operand 1, <src2> - Source operand 2**
Instruction Modifiers and Shapes

**Q Modifier**
- Saturating instructions that saturate set the “Cumulative saturation” flag (QC bit) in the Floating-point Status and Control Register (FPSCR)
- The QC flag is sticky
  - Use VMRS and VMSR instructions to read and to clear the flag

**H Modifier**
- Halves the result
- Can only be used on addition and subtraction instructions
  - VHADD, VHSUB and VRHADD

**D Modifier**
- Only available for saturating variants “long” and “high half” multiplies
- VQDMLAL, VQDMLSL, VQDMULH, VQDMULL, VQRDMULH

**R Modifier**
- Always “Round to Nearest”, as defined in the IEEE 754 standard
- Available on instructions that include a right shift
  - Including Halving and “high half” instructions
Instruction Shapes

- Long and Wide – Input elements are promoted before operation
- Narrow – Input elements are demoted before operation
Multiple 1-Element Structure Access

- **VLD1, VST1 provide standard array access**
  - An array of structures containing a single component is a basic array
  - List can contain 1, 2, 3 or 4 consecutive registers
  - Transfer multiple consecutive 8, 16, 32 or 64-bit elements

```plaintext
VLD1.16 {D7}, [R4], R3
VST1.16 {D3,D4}, [R1]
```

![Diagram of array access](image)
Addition: Basic

- NEON supports various useful forms of basic addition

- Normal Addition - VADD, VSUB
  - Floating-point
  - Integer (8-bit to 64-bit elements)
  - 64-bit and 128-bit registers

- Long Addition - VADDL, VSUBL
  - Promotes both inputs before operation
  - Signed/unsigned (8-bit to 32-bit source elements)

- Wide Addition - VADDW, VSUBW
  - Promotes one input before operation
  - Signed/unsigned (8-bit 32-bit source elements)
Example – adding all lanes

- Input in Q0 (D0 and D1)
- u16 input values

- Now Q0 contains 4x u32 values (with 15 headroom bits)

- Reducing/folding operation needs 1 bit of headroom

- Result is u64 in D0
Summing a vector
Some NEON clever features
Data Movement: Table Lookup

- Uses byte indexes to control byte look up in a table
  - Table is a list of 1,2,3 or 4 adjacent registers

VTBL: out of range indexes generate 0 result
VTBX: out of range indexes leave destination unchanged

VTBL.8 D0, {D1, D2}, D3
Element Load Store Instructions

- All treat memory as an array of structures (AoS)
  - SIMD registers are treated as structure of arrays (SoA)
  - Enables interleaving/de-interleaving for efficient SIMD processing
  - Transfer up to 256-bits in a single instruction

- Three forms of Element Load Store instructions are provided

- Forms distinguished by type of register list provided
  - Multiple Structure Access e.g. \{D0, D1\}
  - Single Structure Access e.g. \{D0[2], D1[2]\}
  - Single Structure Load to all lanes e.g. \{D0[], D1[]\}
Multiple 2-Element Structure Access

- **VLD2, VST2** provide access to multiple 2-element structures
  - List can contain 2 or 4 registers
  - Transfer multiple consecutive 8, 16, or 32-bit 2-element structures

![Diagram showing VLD2 access to multiple 2-element structures]
Multiple 3/4-Element Structure Access

- VLD3/4, VST3/4 provide access to 3 or 4-element structures
  - Lists contain 3/4 registers; optional space for building 128-bit vectors
  - Transfer multiple consecutive 8, 16, or 32-bit 3/4-element structures
Logical

- **NEON supports bitwise logical operations**

- **VAND, VBIC, VEORR, VORN, VORR**
  - Bitwise logical operation
  - Independent of data type
  - 64-bit and 128-bit registers

- **VBIT, VBIF, VBSL**
  - Bitwise multiplex operations
  - Insert True, Insert False, Select
  - 3 versions overwrite different registers
  - 64-bit and 128-bit registers
  - Used with masks to provide selection

\[
\begin{align*}
VAND & : D0, D0, D1 \\
VORR & : Q0, Q1, Q15 \\
VEOR & : Q7, Q1, Q15 \\
VORN & : D15, D14, D1 \\
VBIC & : D0, D30, D2
\end{align*}
\]
NEON instruction summary

▪ A comprehensive set of data processing instructions
  ▪ Form a general purpose SIMD instruction set suitable for compilers

▪ NEON operations fall into the following categories
  ▪ Addition / Subtraction (Saturating, Halving, Rounding)
    ▪ MIN, MAX, NEG, MOV, ABS, ABD, …
  ▪ Multiplication (MUL, MLA, MLS, …)
  ▪ Comparison and Selection
  ▪ Logic (AND, ORR, EOR, BIC, ORN, …)
  ▪ Bitfield
  ▪ Reciprocal Estimate/Step, Reciprocal Square Root Estimate/Step
  ▪ Miscellaneous (DUP, EXT, CLZ, CLS, TBL, REV, ZIP, TRN, …)

▪ Many more…
NEON instruction reference

- Available to partners on [www.arm.com](http://www.arm.com)
Further Reading

- **Documentation**
  - ARM “ARM” v7 A&R
  - “NEON Support in the Realview compiler” white paper
  - “NEON optimizations in Android” white paper
  - Realview Compiler Guide (for intrinsics)
  - ARM Cortex-A Programmers’ Guide (from www.arm.com downloads)
  - Software blogs on www.arm.com
NEON Agenda

- NEON Hardware overview
- NEON Instruction set overview
- NEON Software Support
- NEON improves performance
How to use NEON

NOEN optimized Open source Libraries

- **OpenMAX DL (Development Layer):** APIs contain a comprehensive set of audio, video and imaging functions that can be used for a wide range of accelerated codec functionality such as MPEG-4, H.264, MP3, AAC and JPEG.
- Broad open source support for NEON

Vectorizing Compilers

- Exploits NEON SIMD automatically with existing C source code

NEON Intrinsics

- C function call interface to NEON operations
- Supports all data types and operations supported by NEON

Assembler Code

- For those who really want to optimize at the lowest level
OpenMAX DL v1.0 Library Summary

- **Video Domain**
  - MPEG-4 simple profile
  - H.264 baseline

- **Still Image Domain**
  - JPEG

- **Image Processing Domain**
  - Colorspace conversion
  - De-blocking / de-ringing
  - Rotation, scaling, compositing

- **Audio Domain**
  - MP3
  - AAC

- **Signal Processing Domain**
  - FIR
  - IIR
  - FFT
  - Dot Product

Spec from: [http://www.khronos.org/openmax/](http://www.khronos.org/openmax/)


NOTE: OpenMax DL provides low level data processing functions, not the complete codecs
NEON in opensource

- Google – WebM – 11,000 lines NEON assembler!
- Bluez – official Linux Bluetooth protocol stack
  - NEON sbc audio encoder
- Pixman (part of cairo 2D graphics library)
  - Compositing/alpha blending
- ffmpeg – libavcodec
  - LGPL media player used in many Linux distros
  - NEON Video: MPEG-2, MPEG-4 ASP, H.264 (AVC), VC-1, VP3, Theora
  - NEON Audio: AAC, Vorbis, WMA
- x264 – Google Summer Of Code 2009
  - GPL H.264 encoder – e.g. for video conferencing
- Android – NEON optimizations
  - Skia library, S32A_D565_Opaque 5x faster using NEON
- Eigen2 – C++ vector math / linear algebra template library
- Theorarm – libtheora NEON version (optimized by Google)
- libjpeg – optimized JPEG decode (IJG library)
- FFTW – NEON enabled FFT library
- LLVM – code generation backend used by Android Renderscript
Automatic Vectorization: ARM Compiler

- Automatic vectorization can generate code targeted for NEON from ordinary C source code
  - Less effort to produce efficient code
  - Portable - no compiler-specific source code features need to be used

- To enable automatic vectorization on armcc, use these options together:
  - \texttt{--vectorize} - enable vectorization
  - \texttt{--cpu Cortex-A9} - provide a CPU option with NEON support
  - \texttt{-O2 or -O3} - select high optimization level.
  - \texttt{-Otime} - optimize for speed over space
  - \texttt{--fpmode fast} - the precision of vectorized floating-point operations is the same as VFP RunFast mode

- \texttt{--diag-warning=optimizations} to obtain useful diagnostics from the compiler on what it could or could not optimize/vectorize

- Note:
  - When you specify \texttt{--vectorize}, automatic vectorization is enabled only if you also specify \texttt{-Otime} and an optimization level of \texttt{-O2} or \texttt{-O3}. 

Automatic Vectorization: GNU tools

- To enable automatic vectorization on GCC/g++, use these options together:

  -mcpu=cortex-a9
  -mfpu=neon
  -ftree-vectorize
  -mvectorize-with-neon-quad
  -mfloat-abi=softfp

  - Specify a suitable ARMv7-A processor
  - enable NEON support
  - support SIMD on many arch
  - O3 implies -ftree-vectorize
  - By default, GCC 4.4 only vectorize for doubleword
  - Can use “hard” for more efficient floating point parameter passing, but all code must be compiled with this option

- Understand more with -ftree-vectorizer-verbose
  - Takes an integer value specifying the level of detail to provide, where 1 enables additional printouts and higher values add even more information.
  - What vectorization the compiler is performing, or what is unable to perform because of possible dependencies
void add_int(int * __restrict pa,  
        int * __restrict pb,  
        unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < (n & ~3); i++)
        pa[i] = pb[i] + x;
}

1. Analyze each loop:  
   - Are pointer accesses safe for vectorization?  
   - What data types are being used? How do they map onto NEON vector registers?  
   - How many loop iterations are there?

2. Unroll the loop to the appropriate number of iterations, and perform other transformations like pointerization

void add_int(int *pa, int *pb,  
        unsigned n, int x)
{
    unsigned int i;
    for (i = ((n & ~3) >> 2); i; i--)
    {
        *(pa + 0) = *(pb + 0) + x;
        *(pa + 1) = *(pb + 1) + x;
        *(pa + 2) = *(pb + 2) + x;
        *(pa + 3) = *(pb + 3) + x;
        pa += 4; pb += 4;
    }
}

3. Map each unrolled operation onto a NEON vector lane, and generate corresponding NEON instructions
int a[256], b[256], c[256];

foo () {
    int i;

    for (i=0; i<256; i++) {
        a[i] = b[i] + c[i];
    }
}

armcc -S --cpu cortex-a8 -O3 -Otime --vectorize test.c

gcc -S -O3 -mcpu=cortex-a8 -mfpu=neon -ftree-vectorize -ftree-vectorizer-verbose=6 -Otime --vectorize test.c
The goal is to try to make the code simple, straightforward, and parallel, so that the compiler can easily convert the code to NEON assembly.

Loops can be modified for better vectorizing:
- Short, simple loops work the best (even if it means multiple loops in your code)
- Avoid breaks / loop-carried dependencies / conditions inside loops
- Try to make the number of iteration a power of 2
- Try to make sure the number of iteration is known to the compiler
- Functions called inside a loop should be inlined

Pointer issues:
- Using arrays with indexing vectorizes better than using pointer
- Indirect addressing (multiple indexing or de-reference) doesn’t vectorize
- Use __restrict key word to tell the compiler that pointers does not reference overlapping areas of memory

Use suitable data types
- For best performance, always use the smallest data type that can hold the required values
NEON Intrinsics

- Available in armcc, GCC/g++, and llvm. Syntax is the same, so source code that uses intrinsics can be compiled by any of these compilers.

Advantage:
- Provide low-level access to NEON instructions. Compiler do hard work like:
  - Register allocation.
  - Code scheduling, or re-ordering instructions.

The C compilers can reorder code to ensure the minimum number of stalls according to a specific processor.

Disadvantage:
- Possibly the compiler output is not exactly the code you want, so there is still some possibility of improvement when moving to NEON assembler code.
NEON Intrinsics - Example

- Include intrinsics header file
  
  ```
  #include <arm_neon.h>
  ```

- Use special NEON data types which correspond to D and Q registers, e.g.
  
  ```
  int8x8_t D-register containing 8x 8-bit elements
  int16x4_t D-register containing 4x 16-bit elements
  int32x4_t Q-register containing 4x 32-bit elements
  ```

- Use special intrinsics versions of NEON instructions
  
  ```
  vin1 = vld1q_s32(ptr);
  vout = vaddq_s32(vin1, vin2);
  vst1q_s32(vout, ptr);
  ```

- Strongly typed!
  
  - Use vreinterpret_s16_s32( ) to change the type
NEON Intrinsics - Example

- NEON intrinsic

```c
#include <arm_neon.h>
uint32x4_t double_elements(uint32x4_t input)
{
    return(vaddq_u32(input, input));
}
```

- Command line with GCC
  - arm-none-linux-gnueabi-gcc -mfpu=neon intrinsic.c

- Command line with RVCT
  - armcc --cpu=Cortex-A9 intrinsic.c
NEON Intrinsics : reference

For information about the intrinsic functions and vector data types, see the:

NEON Assembler Code

- Advantage:
  - Careful hand-scheduling is recommended to get the best out of any NEON assembler code you write, especially for performance-critical applications

- Disadvantage:
  - Need to be aware of some underlying hardware features, like pipelining and scheduling issues, memory access behavior and scheduling hazards.
  - Optimization is processor dependent.
NEON Assembler Code - Example

- **Gas**

```assembly
.text
.arm
.global double_elements
double_elements:
  vadd.i32 q0,q0,q0
  bx lr
.end
```

- **Command:** `arm-none-linux-gnueabihf-as -mfpu=neon asm.s`

- **RVCT**

```assembly
AREA RO, CODE, READONLY
ARM
EXPORT double_elements
double_elements
  VADD.I32 Q0, Q0, Q0
  BX LR
END
```

- **Command:** `armasm --cpu=Cortex-A9 asm.s`
Enabling NEON before using

- NEON data engine is disabled at reset
  - Needs to be enabled in software before use

- Enabling NEON requires 2 steps
  1. Enable access to coprocessors 10 and 11 and allow Neon instructions

    ```
    MRC   p15, 0x0, r0, c1, c0, 2     ; Read CP15 CPACR
    ORR   r0, r0, #(0x0f << 20)       ; Full access rights
    MCR   p15, 0x0, r0, c1, c0, 2     ; Write CP15 CPACR
    ISB
    ```

  2. Enable NEON and VFP

    ```
    MOV  r0, #0x40000000 ; set bit 30
    VMSR  FPEXC, r0      ; write r0 to Floating Point Exception Register
    ```
NEON Agenda

- NEON Hardware overview
- NEON Instruction set overview
- NEON Software Support
- NEON improves performance
NEON in Audio

▷ FFT: 256-point, 16-bit signed complex numbers
  − FFT is a key component of AAC, Voice/pattern recognition etc.
  − Hand optimized assembler in both cases

<table>
<thead>
<tr>
<th>FFT time</th>
<th>No NEON (v6 SIMD asm)</th>
<th>With NEON (v7 NEON asm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cortex-A8 500MHz</td>
<td>15.2 us</td>
<td>3.8 us</td>
</tr>
<tr>
<td>Actual silicon</td>
<td></td>
<td>(x 4.0 performance)</td>
</tr>
</tbody>
</table>

▷ Extreme example: FFT in ffmpeg: 12x faster
  − C code -> handwritten asm
  − Scalar -> vector processing
  − Single-precision floating point on Cortex-A8 (VFPlite -> NEON)
ARM RVDS vectorizing compiler

• RVDS 4.0 professional includes auto-vectorizing armcc
  – armcc --vectorize --cpu=Cortex-A8 x.c

• Up to 4x performance increase for benchmarks, with no source code changes (no source code changes are permitted for benchmarking)

  ARM vs NEON (Vectorize) on Cortex-A8

<table>
<thead>
<tr>
<th></th>
<th>Telecom</th>
<th>Consumer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>NEON</td>
<td>135%</td>
<td>169%</td>
</tr>
</tbody>
</table>

  Improved vectorization in latest RVDS 4.0

• Simple source code changes can yield significant improvements above this
  – Use C ‘__restrict’ keyword to work around C pointer aliasing issues
  – Make loops clearly multiple of $2^n$ (e.g. use $4^n$ as loop end) to aid vectorization
FFmpeg/libav performance

- git.ffmpeg.org
  - snapshot 21-Sep-09

  YouTube HQ video decode
  - 480x270, 30fps
  - Including AAC audio

- Real silicon measurements
  - OMAP3 Beagleboard
  - ARM A9TC

- NEON ~2x overall performance
AC3 decode MHz

- Using ffmpeg (LGPL opensource codec) with extensive NEON optimizations
- Dolby reference code also available optimized from Dolby and other vendors

MHz required (lower is better)

ffmpeg from git.libav.org
Checkout from 14-Nov-2011
./configure --extra-cflags="-mcpu=cortex-a9 -mfpu=neon -mfloat-abi=softfp"
Benchmarked on 500MHz Cortex-A9 (ARM Versatile Express)
Samples from:
samples.mplayerhq.hu/A-codecs
JPEG decode

- android_external_jpeg optimized for v6 and v7neon
- 16Mpixel digital camera test image - takes only 1.7s to decode on 500MHz A9+NEON (improved from 4.5s unoptimized)

![cycles/pixel - lower is better](chart.png)

Code from:
https://github.com/mansr
ARM Versatile Express 500MHz Cortex-A9 Ubuntu 11.04 Linux
djpeg -outfile /dev/null testfile.jpg

Also libjpeg-turbo via Linaro
http://libjpeg-turbo.virtualgl.org/
NEON Summary

- NEON will become standard on general purpose apps & media-centric devices.
  - NEON option across the Cortex-A roadmap
  - NEON ideal for use by open OS systems with downloaded apps

- Full enabling technology to support Neon
  - Compilers, profilers, debuggers, libraries all available now
  - Key differentiator: easy to program, popular with software engineers

- Strong ARM NEON ecosystem

- Complementary to DSP/FPGA
Zynq-7000
Hardware Architecture