A Comparison of SIMD in Power Architecture and ARM Technologies

FTF-NET-F0013
Chuck Corley | DMTS
Kelly Johnson | SMTS

A P R . 2 0 1 4
Abstract: A Comparison of SIMD in Power Architecture and ARM Technologies

• Session Length: 2 hours

• Freescale's e6500 core, built on Power Architecture technology, has a single instruction, multiple data execution unit called AltiVec technology. ARM architecture has advanced SIMD extension to the ARMv7-A architecture which is called Neon. AltiVec technology has 32 128-bit wide registers whereas Neon has 32 64-bit wide registers which can be used as 16 128-bit wide registers. AltiVec technology has a separate double precision floating point execution unit with its own 32 64-bit wide registers; Neon shares the 32 64-bit registers with the ARM floating point co-processor. These kinds of architectural differences will be explored along with the programming models, software tools and library support of the SIMD units in each architecture.
Agenda

• Program Example
• Introduction to NEON
• Hardware Overview
• Instruction Set Overview
• Software Support
#include <stdio.h>
#include <stdlib.h>
#endif __ARM__ #include <arm_neon.h>
#else #include <altivec.h>
#endif
#define N 4096
int main(void){
    int i;
    float a[N], b[N], sum = 0;
    for (i=0; i < N; i++) {
        a[i] = rand()%N/(float)N;
        b[i] = rand()%N/(float)N;
    }
    for (i=0; i < N; i++) {
        sum += a[i] * b[i];
    }
    printf("Dot product = %f\n",sum);
    return sum;
}
Compilation for SIMD

NEON:

ALTIVEC:
gcc -c -O3 -Wall -m32 -maltivec -mabi=altivec -mcpu=e6500 -ftree-vectorize -ftree-vectorizer-verbose=5 -ffast-math -funsafe-math-optimizations ./src/timed_simple_dot_product.c -o ./obj/timed_simple_dot_product.o
# Neon inner loop

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>842a:</td>
<td>18c2</td>
<td>adds r2, r0, r3</td>
</tr>
<tr>
<td>842c:</td>
<td>eb0c 0103</td>
<td>add.w r1, ip, r3</td>
</tr>
<tr>
<td>8430:</td>
<td>3310</td>
<td>adds r3, #16</td>
</tr>
<tr>
<td>8432:</td>
<td>ecd1 4b04</td>
<td>vldmia r1, {d20-d21}</td>
</tr>
<tr>
<td>8436:</td>
<td>ecd2 2b04</td>
<td>vldmia r2, {d18-d19}</td>
</tr>
<tr>
<td>843a:</td>
<td>f5b3 4f80</td>
<td>cmp.w r3, #16384 ; 0x4000</td>
</tr>
<tr>
<td>843e:</td>
<td>ff44 2df2</td>
<td>vmul.f32 q9, q10, q9</td>
</tr>
<tr>
<td>8442:</td>
<td>ef40 0de2</td>
<td>vadd.f32 q8, q8, q9</td>
</tr>
<tr>
<td>8446:</td>
<td>d1f0</td>
<td>bne.n 842a &lt;main+0x7a&gt;</td>
</tr>
<tr>
<td>8448:</td>
<td>ef40 0da1</td>
<td>vadd.f32 d16, d16, d17</td>
</tr>
</tbody>
</table>
**AltiVec inner loop**

10000430: 7d bd 48 ce lvx v13,r29,r9
10000434: 7c 3e 48 ce lvx v1,r30,r9
10000438: 39 29 00 10 addi r9,r9,16
1000043c: 10 0d 00 6e vmaddfp v0,v13,v1,v0
10000440: 42 00 ff f0 bdnz+ 10000430 <main+0xf0>
10000444: 10 20 02 2c vsldoi v1,v0,v0,8
Compilation without SIMD

NEON:
gcc -c -O3 -Wall -mcpu=cortex-a8 -mfpu=neon -funsafe-math-optimizations ./src/timed_simple_dot_product.c -DSCALAR -o ./obj/timed_simple_dot_product_scalar.o

(-mfloat-abi=soft prevented using VFP but made it too slow)

(adding -funsafe-math-optimizations or -ffast-math for non-IEEE 754 gives nearly Neon performance)

ALTIVEC:
gcc -c -O3 -m32 -mcpu=e6500 -mno-altivec
./src/timed_simple_dot_product.c -DSCALAR -o ./obj/timed_simple_dot_product_scalar.o
Non-Neon Scalar inner loop

-mfpu=neon

8494: f7ff ef80 blx 8398 <_init+0x5c>
8498: 17c2 asrs r2, r0, #31
849a: 0cd2 lsrs r2, r2, #19
849c: 1883 adds r3, r0, r2
849e: 04db lsls r3, r3, #19
84a0: 0cdb lsrs r3, r3, #19
84a2: 1a9b subs r3, r3, r2
84a4: 005b lsls r3, r3, #1
84a6: ee07 3a90 vmov s15, r3
84aa: eef8 7ae7 vcv.t.f32.s32 s15, s15
84ae: ee67 7a88 vmul.f32 s15, s15, s16
84b2: ee77 7aa7 vadd.f32 s15, s15, s15
84b6: ee17 3a90 vmov r3, s15
84ba: 5133 str r3, [r6, r4]
84bc: f7ff ef6c blx 8398 <_init+0x5c>
84c0: 17c2 asrs r2, r0, #31
84c2: 0cd2 lsrs r2, r2, #19
84c4: 1883 adds r3, r0, r2
84c6: 04db lsls r3, r3, #19
84c8: 005b lsls r3, r3, #1
84ca: 1a9b subs r3, r3, r2
84cc: 005b lsls r3, r3, #1
84ce: ee07 3a90 vmov s15, r3
84d2: eef8 7ae7 vcv.t.f32.s32 s15, s15
84d6: ee67 7a88 vmul.f32 s15, s15, s16
84da: ee77 7aa7 vadd.f32 s15, s15, s15
84de: ee17 ea90 vmov lr, s15
84e2: f845 e004 str.w lr, [r5, r4]
84e6: 3404 adds r4, #4
84e8: f5b4 3f80 cmp.w r4, #65536;
84ec: d1d2 bne.n 8494 <main+0x6c>
Non-Neon Scalar inner loop

- loop -mfpu=neon -funsafe-math-optimizations
- can result in incorrect output for programs that depend on an exact implementation of IEEE

```
8504: 18c2       adds   r2, r0, r3
8506: edd2 2b00  vldr    d18, [r2]
850a: 18ca       adds   r2, r1, r3
850c: 3308       adds   r3, #8
850e: f5b3 3f80  cmp.w   r3, #65536 ; 0x10000
8512: edd2 1b00  vldr    d17, [r2]
8516: ff42 1db1  vmul.f32 d17, d18, d17
851a: ef40 0da1  vadd.f32 d16, d16, d17
851e: d1f1       bne.n   8504 <main+0xdc>
```
Non-AltiVec scalar inner loop

10000530:    7d a9 e4 2e    lfsx    f13,r9,r28
10000534:    7c 1f 4c 2e    lfsx    f0,r31,r9
10000538:    39 29 00 04    addi    r9,r9,4
1000053c:    ef ed f8 3a    fmadds  f31,f13,f0,f31
10000540:    42 00 ff f0    bdnz+  10000530 <main+0x1c0>
i.mx53 vs T4240 comparison on simple Dot Product

Execution Time
10000 loops (secs)

Scalar (or VFP) Vector

(scaled to same frequency)
Actual Results

- On i.mx53 QuickStart Board @1000 MHz without Neon (float-abi=soft):
  - 10000 iterations took exactly 11590000 cycles or 11.590 seconds (including function call) Dot product = 66071.070312

- On i.mx53 QuickStart Board @1000 MHz without Neon but with VFP:
  - 10000 iterations took exactly 5250000 cycles or 5.250 seconds (including function call) Dot product = 66071.070312

- On i.mx53 QuickStart Board @1000 MHz without Neon but -ffast-math with VFP:
  - 10000 iterations took exactly 1070000 cycles or 1.070 seconds (including function call) Dot product = 66070.898438

- On i.mx53 QuickStart Board @1000 MHz with Neon:
  - 10000 iterations took exactly 660000 cycles or 0.660 seconds (including function call) Dot product = 66070.921875

- On T4240QDS @ 1666 MHz without AltiVec:
  - 10000 iterations took exactly 740000 cycles or 0.740 seconds (including function call) Dot product = 66071.070312

- On T4240QDS @ 1666 MHz with AltiVec:
  - 10000 iterations took exactly 220000 cycles or 0.220 seconds (including function call) Dot product = 66070.921875
Introduction to NEON
What is NEON?

- **Advanced SIMD**
  - An instruction set extension that provides *Single Instruction Multiple Data (SIMD)* integer and single-precision floating-point vector operations on double word and quad word registers.

- **NEON Technology**
  - The implementation of the Advanced SIMD extension to the ARMv7-A architecture.

- **32 x 64-bit wide registers**
  - Registers can be used as 16 x 128-bit wide registers

- **Supported data types**
  - 8-bit, 16-bit, 32-bit, 64-bit integer (signed/unsigned)
  - Single precision floating point
  - Polynomial \{0,1\}
SIMD Review

• Multiple data elements stored in single register
• Same operation performed on each set of data
Example SIMD Instruction – Vector ADD

- Register split into equal size and type elements
- Same operation performed on each set of data
- VADD.U16 D2, D1, D0

<table>
<thead>
<tr>
<th>63</th>
<th>47</th>
<th>31</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0x200</td>
<td>0x300</td>
<td>0x400</td>
</tr>
<tr>
<td>0xA</td>
<td>0xB</td>
<td>0xC</td>
<td>0xD</td>
</tr>
</tbody>
</table>

```
D0
ADD
0x10A
```
```
D1
ADD
0x20B
```
```
D2
ADD
0x30C
```
```
ADD
0x40D
```
AltiVec 3-source, 1-destination SIMD Instructions

16 x 8-bit elements

8 x 16-bit elements

4 x 32-bit elements
Why Use NEON?

- **Performance**
  - 60-150% performance boost on complex video codecs
  - Individual simple DSP algorithms can show larger performance boost (4x-8x)

- **Cost Effective Alternative to a DSP**
  - Easier to program
  - Coding and debugging use same tools as ARM core
  - Tight coupling to the ARM processor provides a single instruction stream and a unified view of memory, presenting a single development platform target with a simpler tool flow.

- **NEON is implemented as part of ARMv7-A Architecture**
  - Third parties actively working on NEON application development

- **Well designed instruction set**
  - Suitable for hand coding or code generation by ARM’s vectorizing compiler

- **Targeted primarily for audio, video, and 3-D graphics processing**
  - More generally, works well on repeated operations on blocks of data
NEON Power Considerations

• Cortex-A15
  – Supports dynamic high-level clock gating of NEON and VFP unit in order to reduce dynamic power dissipation.
  – With NEON and VFP unit power up, the clock to the unit is enabled when Advance SIMD or VFP instruction is detected in the pipeline, and is disabled otherwise.

• If NEON and VFP unit is not required, you can reduce leakage power by turning off the power to the unit.

• OS can enable/disable NEON and VFP unit
  – Power bring up may take substantial time
  – “On demand” disabling is not easy
AltiVec Power Considerations

- The AltiVec unit will go drowsy after some number of cycles occur and no AltiVec instruction has attempted execution
  - When the AltiVec unit isn’t being used, we basically turn it off to reduce static power
  - The state of the vector registers and the VSCR are retained
  - Note that AltiVec load and store instructions are considered AltiVec instructions

- The AltiVec unit will power up when an AltiVec instruction attempts execution and the AltiVec available bit (MSR[SPV]) is set
  - The core will eventually stall waiting for the AltiVec unit to power up so the instruction can be executed and retired
Hardware Overview
Floating Point (VFP)

- Floating point
  - A floating-point coprocessor extension to the instruction set architecture.
  - For historic reasons, the Floating-point Extension is also called the VFP Extension
    - “Vector mode” was removed shortly after its introduction

- Versions of Floating-point (VFP) Extension
  - VFPv3 and VFPv4 are optional extensions to the ARM, Thumb, and ThumbEE instruction sets in the ARMv7-A and ARMv7-R profiles.
    - VFPv3/VFPv4 can be implemented with either 32 or 16 double word (64-bit) registers.
    - VFPv3U/VFPv4U is a variant of VFPv3/VFPv4 that supports the trapping of floating-point exceptions
    - VFPv3 and VFPv3U can be extended by the optional Half-precision Extension
    - VFPv4 and VFPv4U add both the Half-precision Extension and the fused multiply-add instructions to the features of VFPv3
Advanced SIMD Versions

- **Advanced SIMDv1**
  - Optional extension to the ARMv7-A and ARMv7-R profiles.

- **Advanced SIMDv1 with Half-precision Extension**
  - Advanced SIMDv1 can be extended by the Optional Half-precision Extension

- **Advanced SIMDv2**
  - Optional extension to the ARMv7-A and ARMv7-R profiles.
  - Advanced SIMDv2 adds both the Half-precision Extension and the fused multiply-add instructions to the features of Advanced SIMDv1.
Floating-point (VFP) and Advanced SIMD

- ARM architecture defines the Advanced SIMD extension as part of the coprocessors 10 and 11, that are also used for VFP extension.
  - Common features in the programmers models for these extensions means an OS that supports VFP requires little or no modifications to also support NEON.

- If an implementation includes both the Floating-point and Advanced SIMD Extensions:
  - It must implement the corresponding versions of the extensions:
    - if the implementation includes VFPv3 it must include Advanced SIMDv1
    - if the implementation includes VFPv3 with the Half-precision Extension it must include Advanced SIMDv1 with the half-precision extensions
    - if the implementation includes VFPv4 it must include Advanced SIMDv2.

- The two extensions use the same register bank. This means VFP must be implemented with 32 double word registers

- NEON registers viewed as either:
  - 32 x 64-bit registers (D0-D31)
  - 16 x 128-bit registers (Q0-Q15)
NEON hardware

- Cortex-A8
  - 10 Stage NEON pipeline after the ARM pipeline
  - NEON has its own load/store unit, can access L1 and L2 caches
- Cortex-A9, Cortex-A5, and Cortex-A15 (implement v7)
  - Separate execute pipeline for NEON instructions (as part of ARM pipeline)
  - NEON shares load/store unit with ARM pipeline
- NEON code is mixed in with ARM/Thumb code
  - Instructions can be cached in I-Cache
  - NEON engine includes a dedicated floating point unit
ARM Cortex A-15 Pipeline Overview

- 15 Stage Integer Pipeline
  - 4 extra cycles for multiply, load/store
  - 2-10 extra cycles for complex media instructions
Each thread: seven stage pipelines with advanced branch prediction, dual-decode, seven-issue, 64-bit simple integer & load/store execution, dual-complete

Shared: Multiply, divide, floating-point, vector execution units, L1 caches
NEON Floating Point

- NEON Floating Point Engine is not fully IEEE-754 compliant
  - Denormals are flushed to zero
  - Rounding is fixed to round-to-nearest
    - Except for conversion operations
  - Single precision only, can perform vector operations
  - No exceptions are generated

- Separate Floating Point Engine is IEEE-754 compliant
  - Usually substantially slower
  - Scalar Operations only
  - Single and Double Precision

- ARMv8 NEON is fully IEEE-754 compliant (and supports double precision floating point)
AltiVec ISA supports two floating-point modes of operation

- **Java mode**
  - Supports denorms as inputs and results (gradual underflow) for arithmetic operations
  - Provides NaN results for invalid operations
  - Supports the following floating-point exceptions
    - NaN operand exception
    - Invalid operation exception
    - Zero divide exception
    - Log of zero exception
    - Overflow exception
    - Underflow exception

- **Non-Java/non-IEEE/non-C9X mode of operation** is useful in circumstances where real-time performance is more important than strict Java and IEEE-standard compliance.
  - Denormals are flushed to zero
  - Rounding is fixed to round-to-nearest
  - Single precision only, can perform vector operations
  - No exceptions are generated

- **Separate Floating Point Engine** is IEEE-754 compliant
  - Scalar Operations only
  - Single and Double Precision
Enabling NEON in software

- The NEON/VFP unit comes up disabled on power on reset.
- Enabling NEON requires some co-processor commands.
- Assembly code required to enable NEON/VFP.
- **Step 1** – enable access to coprocessors 10 and 11 and allow NEON instructions
  
  ```
  MRC p15, 0x0, r0, c1, c0, 2 ; Read CP15 CPACR (controls accesses to coprocessors CP10,CP11)
  ORR r0, r0, #(0x0f << 20) ; enable full access for p10,11
  MCR p15, 0x0, r0, c1, c0, 2 ; Write CP15 CPACR
  ISB
  ```

- **Step 2** - Enable NEON and VFP
  
  ```
  MOV r0, #0x40000000 ; set bit 30
  VMSR FPEXC, r0 ; write r0 to FP Exception Reg
  ```

Advanced SIMD and VFP extensions are enabled and operate normally
NEON Status Registers

- **FPSID** – Floating Point System ID (read only)
  - Provides top-level information about the floating-point implementation.
  - Bit 23 should be 0 for hardware VFP and SIMD implementation

- **FPEXC** – Floating Point Exception Control Register
  - Provides a global enable for the Advanced SIMD and Floating-point (VFP) Extensions, and indicates how the state of these extensions is recorded.
  - Bit 30 set to 1 to enable NEON and VFP

- **FPSCR** – Floating Point Status and Control Register
  - Provides floating-point system status information and control.
  - This register is an Advanced SIMD and Floating-point Extension system register.

- **MVFR0/1** – Media and VFP Feature Registers (read only)
  - Describes the features provided by the Advanced SIMD and Floating-point Extensions.
    - e.g., fused multiply accumulate operations, half-precision

- For more information, see ARM v7-A ARM
AltiVec Registers

- **VSCR** – Vector Status and Control Register
  - Selects the Java-IEEE-C9X–compliant mode or a possibly faster non-Java/non-IEEE mode.
  - A sticky status bit indicating that some field in a saturating instruction saturated.
- **VRSAVE** – Vector Save/Restore Register
  - A user-level 32-bit SPR used to assist in application and operating system software in saving and restoring the architectural state across process context-switched events.
- **CR6** – PowerPC Condition Register
  - Can optionally be used, that is if an AltiVec instruction field’s record bit (Rc) is set in a vector compare instruction.
- **MSR** – Machine State Register
  - An AltiVec available field is added.
- ISA Rev 3 added IVORs for AltiVec unavailable interrupt and AltiVec assist interrupt.
- For more information, see AltiVec Power ISA Programming Environments Manual.
Instruction Set Overview
Instruction Syntax (ARMv7-A)

V{<modifier>}<operation>{<shape>}{<cond>}{.<dt>} {<dest>,} <src1>, <src2>

- All Advanced SIMD and Floating-point (VFP) instructions begin with a V
  - This distinguishes Advanced SIMD vector and floating-point instructions from ARM scalar instructions.

- <operation> - instruction operation
  - It is usually a three letter mnemonic the same as or similar to the corresponding scalar integer instruction.
  - Examples: ADD, MUL, MLA, MAX, SHR, SHL, MOV

- <cond> - Conditional, used with IT (if-then) instruction

- <.dt> - Data Type
Instruction Modifiers

V{<modifier>}<operation>{<shape>}={<cond>}{.<dt>}{<dest>,} <src1>, <src2>

Q: The operation uses saturating arithmetic
H: The operation halves the result (example usage – averaging)
D: The operation doubles the result (example usage – normalize)
R: The operation performs rounding

Examples:
VQADD, VHADD, VQDMUL, VRHADD

Modifiers are not available for every instruction – see ARM ARM
# Instruction Shapes

<table>
<thead>
<tr>
<th>Shape</th>
<th>Meaning</th>
<th>Typical Register Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td>(None)</td>
<td>The operands and result are all the same width</td>
<td>Dd,Dn,Dm Qd,Qn,Qm</td>
</tr>
<tr>
<td>L</td>
<td>Long Operation - Result is twice the width of both operands.</td>
<td>Qd,Dn,Dm</td>
</tr>
<tr>
<td>N</td>
<td>Narrow Operation - Result is half the width of both operands.</td>
<td>Dd,Qn,Qm</td>
</tr>
<tr>
<td>W</td>
<td>Wide Operation - Result and first operand are twice the width of the second operand.</td>
<td>Qd,Qn,Dm</td>
</tr>
</tbody>
</table>

Long and Wide: Input elements are promoted before operation.
Operand Shapes: Not available for every instruction – see ARM ARM.
NEON Data Types

• Data Types
  - **Unsigned integer** (U8, U16, U32, U64)
  - **Signed integer** (S8, S16, S32, S64)
  - **Integer of unspecified type** (I8, I16, I32, I64)
  - **Floating point** – single precision (F32)
  - **Polynomial** (P8, P16) - polynomial over \{0, 1\} of degree less than <size>

• `<dt>` field indicates data type contained in:
  - the second operand, if any
  - the operand, if there is no second operand
  - the result, if there are no operand registers

• The data types of the other operand and result are implied by the `<dt>` field combined with the instruction shape.

• Some instructions require only size information (duplication, load/store)
Data Type Examples

VADD.I16 D2, D1, D0
  I16 and no shaping => D1, D0 contain 4 16-bit integers
  D2 contains 4 16-bit integers after the ADD

VADDL.I32 Q0, D3, D4
  I32 and L => D3, D4 contain 4 16-bit integers that will be promoted
  to 32-bit integers prior to ADD
  I32 and L => Q0 contains 4 32-bit integers after the ADD

VADDHN.I32 D0, Q1, Q2
  I32 and N => Q1, Q2 contain 4 32-bit integers
  Modifier H => Halves the result after the ADD
  I32 and N => D0 contains 4 16-bit integers after the ADD
NEON Instruction Examples

• Addition/Subtraction
  VADD.I16  D0, D1, D2 ; adds 4 16-bit integers
  VADD.F32  Q4, Q7, Q8 ; adds 4 floats

• Absolute difference
  VABD.S16  D0, D1, D2 ; absolute difference of 4 16-bit signed integers

• Saturating Math
  VQADD.S16 D0, D1, D2 ; saturated add of 4 16-bit signed integers

• Multiplication
  VMUL.I8   Q0, Q1, Q2 ; multiplies 16 8-bit integers
  VMUL.I16  D1, D7, D4[2] ; multiplies 4 16-bit integers by second element of D4
NEON Instruction Examples

• Logical Operations
  VORR Q0, Q1, Q2 ; logical OR of 128 bits
  VAND D4, D7, D8 ; logical AND of 64 bits

• General Data Processing
  VDUP.32 Q0, R1 ; duplicates 32-bit scalar in R1 across register Q0

• Load/Store Instructions
  VLD1.32 {D0, D1}, [R1]! ; load 4 32-bit elements into D0 and D1, and update R1
  VST1.32 {D0, D1}, [R1]! ; store 4 32-bit elements from D0 and D1, and update R1

• See ARM ARMv7-AR for a listing of all NEON instructions
Instruction Syntax (AltiVec)

- AltiVec instructions are four bytes (32 bits) long and are word-aligned. AltiVec instruction set architecture (ISA) has four operands, three source vectors, and one result vector.

- Example:
AltiVec Data Types

- **Data Types**
  - **Unsigned integer** (U8, U16, U32)
  - **Signed integer** (S8, S16, S32)
  - **Integer of unspecified type** (I8, I16, I32, I64)
  - **Floating point – single precision** (32)
  - **Boolean** (8, 16, 32)
  - **Pixel** (8/8/8/8 or 1/5/5/5)
AltiVec Permute Instruction

- Provides full byte-wide data crossbar for 128-bit registers
- Selects any 16 8-bit elements from 2x16 8-bit elements
- Other AltiVec instructions are special cases of permute
  - Pack, unpack, and merge
  - Splat (element or literal replication)
  - Shift left long immediate
  - Permute also supports other higher-level functions
  - Software-managed unaligned access
  - Table look-up
Neon Permute Instruction(s) - VTBL, VTBX

- Uses byte indexes in a control vector to look up byte values in a table and generate a new vector.
- The table consists of one to four adjacent D registers.
- Other Neon instructions offer more restrictive permute functionality:
  - VEXT extracts a new vector of bytes from a pair of existing vectors
  - VREV reverses the order of 8, 16 or 32-bit elements within a vector.
  - VTRN transposes 8, 16 or 32-bit elements between a pair of vectors
Software Support
Auto-vectorization

- Compiler can perform *automatic vectorization* on your C or C++ source code.
  - access to high NEON performance without writing assembly code or using intrinsics
  - source code remains portable between different tools and target platforms
  - might have to give the compiler additional hints about where this is safe and optimal

- Supported by both armcc and gcc
Automatic Vectorization

• RVCT (Real View Compilation Tools)
  - Specify a target processor that includes NEON technology, compile for optimization level \(-O2\) or higher, and add \(-Otime\) and \(--\text{vectorize}\) to the command line, for example:
    
    ```
    armcc --cpu=Cortex-A9 -O3 -Otime --vectorize -c vectorized.c
    ```
  - You can request more verbose compiler output by adding \(--\text{remarks}\) to the command line.

• GCC
  - Add \(-\text{mfpu}=\text{neon}\) and \(-\text{ftree}-\text{vectorize}\) to the GCC command line, for example:
    
    ```
    arm-none-linux-gnueabi-gcc -mfpu=neon -ftree-vectorize -c vectorized.c
    ```
  - You can request more verbose compiler output by adding \(-\text{ftree-}\text{vectorizer-verbose}=1\) to the command line
Tuning C/C++ code for vectorizing

• Help the compiler convert your code to NEON assembly
  - Write code that is simple, straightforward and parallel

• Loops
  - Short, simple loops work best
  - Avoid breaks in loops
  - Number of loop iterations should be a power of 2
  - Make sure the number of iterations is known to the compiler
  - Functions called inside a loop should be in-lined
    ▪ Compiler can’t handle register issues with function calls

• Pointer Issues
  - Accessing arrays using simple indexing vectorizes better than using pointers
  - Indirect addressing doesn’t vectorize well
  - Use the **restrict** keyword to tell the compiler that pointers do not reference overlapping areas of memory
Example 1-4 NEON Vectorization

• Small function that the compiler can safely and optimally vectorize:

```c
void add ints(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;
}
```

• __restrict key word used to guarantee that the pointers pa and pb will not address overlapping regions of memory

• Loop will always execute a multiple of 4 times
  - bottom two bits of n masked off for the limit test.

• This extra information makes it safe for the compiler to vectorize this function into NEON load and store operations.
C Intrinsic

- An intrinsic function appears as a function call in C or C++, but is replaced during compilation by a sequence of low-level instructions.
  - Highly specific to core and architecture (unlike C/C++)
  - Using intrinsics means the developer does not have to consider register allocation and interlock issues, because the compiler handles these.
- **NEON intrinsics defined in header file** `arm_neon.h`
  - Also defines a sect of vector data types of different sizes, e.g.,
    - `int16x4_t` - vector of four 16-bit short int values
    - `float32x4_t` - vector of four 32-bit float values
- GCC and RVCT support the same NEON intrinsic syntax, making C or C++ code portable between the toolchains.
Example 1-3 NEON Intrinsics

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

- To use NEON intrinsics in GCC, you must specify `–mfpu=neon` on the compiler command line:
  ```
  arm-none-linux-gnueabi-gcc –mfpu=neon intrinsic.c
  ```

- RVCT accepts NEON intrinsics if you specify, on the compiler command line, a target processor that supports the NEON instructions:
  ```
  armcc --cpu=Cortex-A9 intrinsic.c
  ```
AltiVec Intrinsics

- AltiVec also has intrinsics which compile to the appropriate assembly language instruction
- AltiVec intrinsics defined in header file `altivec.h`
- GCC and other compilers support the same AltiVec intrinsic syntax
- For additional information, see AltiVec Technology Programming Interface Manual
- Example AltiVec intrinsics:

```
vec_adds
Vector Add Saturated

d = vec_adds(a, b)

n ← number of elements
do i=0 to n-1
d_i ← Saturate(a_i + b_i)
end
```
Software Support - Neon Libraries
OpenMAX (Open Media Acceleration)

www.khronos.org/openmax
OpenMAX Development Library (DL)

- OpenMAX is a royalty-free cross platform API standard created and distributed by the Khronos Group.
- ARM has created an ARMv7 NEON optimized implementation of the OpenMAX Development Layer (DL).
  - Interface between a codec and underlying hardware
  - Download from [http://www.arm.com](http://www.arm.com)
2.2.2.1.1 DotProd_S16

Prototype
OMX_S32 omxSP_DotProd_S16 (const OMX_S16 *pSrc1, const OMX_S16 *pSrc2, OMX_INT len);

Description
Calculates the dot product of the two input vectors. This function does not perform scaling. The internal accumulator width must be at least 32 bits. If any of the partially accumulated values exceeds the range of a signed 32-bit integer then the result is undefined.

Input Arguments
• pSrc1 – pointer to the first input vector; must be aligned on an 8-byte boundary.
• pSrc2 – pointer to the second input vector; must be aligned on an 8-byte boundary.
• len – length of the vectors in pSrc1 and pSrc2

Returns
• The dot product result
OpenMAX example

- OpenMAX is a royalty-free cross platform API standard created and distributed by the Khronos Group. ARM has created an ARMv7 NEON optimized implementation of the OpenMAX Development Layer (DL). You can download this from http://www.arm.com.
- **Example 1.5** calculates the dot product of the values in two vectors of signed 16-bit integers by calling the OpenMAX function omxSP_DotProd_S16(). This function is implemented using NEON vector operations when using the ARMv7 optimized OpenMAX DL library.

**Example 1.5. OpenMAX example**

```c
#include <omxSP.h>
OMX_S16 source1[] = {42, 23, 983, 7456, 124, 11111, 4554, 10002};
OMX_S16 source2[] = {242, 423, 9832, 746, 1124, 1411, 2254, 1298};
OMX_S32 source_dotproduct(void) {
    OMX_INT len = sizeof(source1)/sizeof(OMX_S16);
    return omxSP_DotProd_S16(source1, source2, len);
}
```
OpenMAX DL v1.0.3 API

- OpenMAX DL is split into five application domains:
  - AC – Audio Codecs (MP3 decoder and AAC decoder components)
  - ID – Image Codecs (JPEG components)
  - IP – Image Processing (Generic image processing functions)
  - SP – Signal Processing (Generic audio processing functions)
  - VC – Video Codecs (MPEG-4 part10 and MPEG-4 part2 components)
- Video encoding functions are not included
Software Support - AltiVec Libraries
Mentor Embedded Performance Library (MEPL) for Freescale’s Altivec Technology

- Optimized for QorIQ AMP Series Altivec Technology

- Over 800 math and signal processing functions
  - Element-wise Vector Functions
  - Reduction Functions
  - Real/Complex Conversion Functions
  - FIR and IIR Filter Functions
  - 1D and 2D FFT, Convolution & Correlation Functions
  - General Signal and Image Processing Functions
  - Linear Algebra Functions

News Release
Freescale Selects Mentor Graphics to Develop Software Libraries for Enhanced Altivec Engine

Highly advanced software intended to optimize performance of Freescale’s QorIQ AMP processors

AUSTIN, Texas, Sep 06, 2011 (BUSINESS WIRE) --

Freescale Semiconductor (NYSE:FSL) has selected Mentor Graphics Corporation (NASDAQ: MENT) to develop a software library of advanced mathematical and signal processing functions optimized for the latest version of Freescale’s Altivec processing engine. The Mentor(R) Embedded Performance Library for Freescale’s Altivec technology is designed to help Freescale’s recently announced QorIQ Advanced Multiprocessing (AMP) processors achieve maximum performance from the newest Altivec engine while simultaneously supporting software developer productivity.
Overview of AltiVec Library Contents

• Basic Linear Algebra Subprograms (BLAS)
  − Standard building blocks for performing basic vector and matrix operations
    ▪ Level 1: scalar, vector, & vector-vector operations
    ▪ Level 2: matrix-vector operations
    ▪ Level 3: matrix-matrix operations

• C Linear Algebra PACKage (CLAPACK) standard functions
  − 1300 linear algebra functions – most benefit from BLAS AltiVec optimizations

• Mentor Embedded Performance Library (MEPL)
  − Includes general signal processing, FIR and IIR, FFT, convolution and correlation, image processing, etc.

• All in binary form as part of MEPL
Using AltiVec Performance Library

• In your c source code:
  
  ```
  #include <mepl.h> (possibly #include <mepl/cblas.h>?)
  ```
  Then provide data and call desired function in accordance with API described herein or in references.

• In your gcc compiler command line:
  
  ```
  -I/<install directory>/MEPL/bin/mepl-1.0/e6500-32/include
  ```
  Or for 64b: 
  ```
  -I/<install directory>/MEPL/bin/mepl-1.0/e6500-64/include
  ```

• In your gcc linker command line:
  
  ```
  -L/<install directory>/MEPL/bin/mepl-1.0/e6500-32/lib -lcblas -latlas -lmepl
  ```
  Or for 64b: 
  ```
  -L/<install directory>/MEPL/bin/mepl-1.0/e6500-64/lib -lcblas -latlas -lmepl
  ```
Conclusion
Conclusion

• Advanced SIMD extension to the ARMv7-A architecture (Neon) supports SIMD execution
  – 32 64b registers (shared with Vector Floating Point)
  – 8b, 16b, 32b integers, 32b float, boolean, 8b polynomials data types

• AltiVec functional unit in the e600 or e6500 Power architecture cores supports SIMD execution
  – 32 128b registers
  – 8b, 16b, 32b integers, 32b float, boolean, 8b packed pixel data types

• Capability is comparable

• Performance may vary from implementation to implementation

• Compiler support is comparable

• Library support seems to be application specific – consumer for ARM; scientific/engineering for AltiVec.
Introducing The QorIQ LS2 Family

Breakthrough, software-defined approach to advance the world’s new virtualized networks

New, high-performance architecture built with ease-of-use in mind
Groundbreaking, flexible architecture that abstracts hardware complexity and enables customers to focus their resources on innovation at the application level

Optimized for software-defined networking applications
Balanced integration of CPU performance with network I/O and C-programmable datapath acceleration that is right-sized (power/performance/cost) to deliver advanced SoC technology for the SDN era

Extending the industry’s broadest portfolio of 64-bit multicore SoCs
Built on the ARM® Cortex®-A57 architecture with integrated L2 switch enabling interconnect and peripherals to provide a complete system-on-chip solution
QorIQ LS2 Family

Key Features

SDN/NFV Switching

Data Center

Wireless Access

Unprecedented performance and ease of use for smarter, more capable networks

High performance cores with leading interconnect and memory bandwidth
- 8x ARM Cortex-A57 cores, 2.0GHz, 4MB L2 cache, w Neon SIMD
- 1MB L3 platform cache w/ECC
- 2x 64b DDR4 up to 2.4GT/s

A high performance datapath designed with software developers in mind
- New datapath hardware and abstracted acceleration that is called via standard Linux objects
- 40 Gbps Packet processing performance with 20Gbps acceleration (crypto, Pattern Match/RegEx, Data Compression)
- Management complex provides all init/setup/teardown tasks

Leading network I/O integration
- 8x1/10GbE + 8x1G, MACSec on up to 4x 1/10GbE
- Integrated L2 switching capability for cost savings
- 4 PCIe Gen3 controllers, 1 with SR-IOV support
- 2 x SATA 3.0, 2 x USB 3.0 with PHY
See the LS2 Family First in the Tech Lab!

4 new demos built on QorIQ LS2 processors:

- Performance Analysis Made Easy
- Leave the Packet Processing To Us
- Combining Ease of Use with Performance
- Tools for Every Step of Your Design