# Using your C compiler to exploit NEON™ Advanced SIMD

http://www.doulos.com/arm/

Marcus Harnisch <marcus.harnisch@doulos.com>

## Abstract

With the v7-A architecture, ARM has introduced a powerful SIMD implementation called NEON™. NEON is a coprocessor which comes with its own instruction set for vector operations. While NEON instructions could be hand coded in assembler language, ideally we want our compiler to generate them for us. Automatic analysis whether an iterative algorithm can be mapped to parallel vector operations is not trivial not the least because the C language is lacking constructs necessary to support this. This paper explains how the RealView compiler tools (RVCT) and other modern compilers use a blend of sophisticated analysis techniques and language extensions to fulfill their job.

## Introduction

Many algorithms in today's multi-media addicted world are ideally suited for vector operations including digital filtering (audio), pixel processing (video), matrix operations (3D, DCT). Some lesser known applications might perhaps include automation (drives, robotics).

Most vector operations carry out the same operation on all elements of their operand vector(s) in parallel (Figure 1), hence the term Single Instruction Multiple Data (SIMD) which has its roots in a computer architecture classification system called "Flynn's Taxonomy".[1]

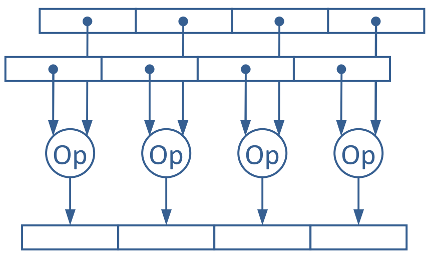


Figure 1 SIMD Operation

Available exclusively to main frame computers and digital signal processors (DSP) for a long time, SIMD didn't make it into mainstream processors until the Intel Pentium MMX hit the market in 1996.

[1] Flynn, M., Some Computer Organizations and Their Effectiveness, IEEE Trans. Comput., Vol. C-21, pp. 948, 1972.

Since then all major desktop processors started including SIMD in some form or another using brand names such as *AltiVec*, *3DNow!*, *SSE*, and most recently – *NEON*.

After software had caught up with this development, desktop applications would see a significant performance boost. But the picture looked quite different still in the embedded space. For actual signal processing (GSM, CDMA), real-time performance required dedicated processors anyhow, while the relatively low performance requirements of application software on, say, early mobile handsets did not justify the extra complexity and cost of large processor extensions.

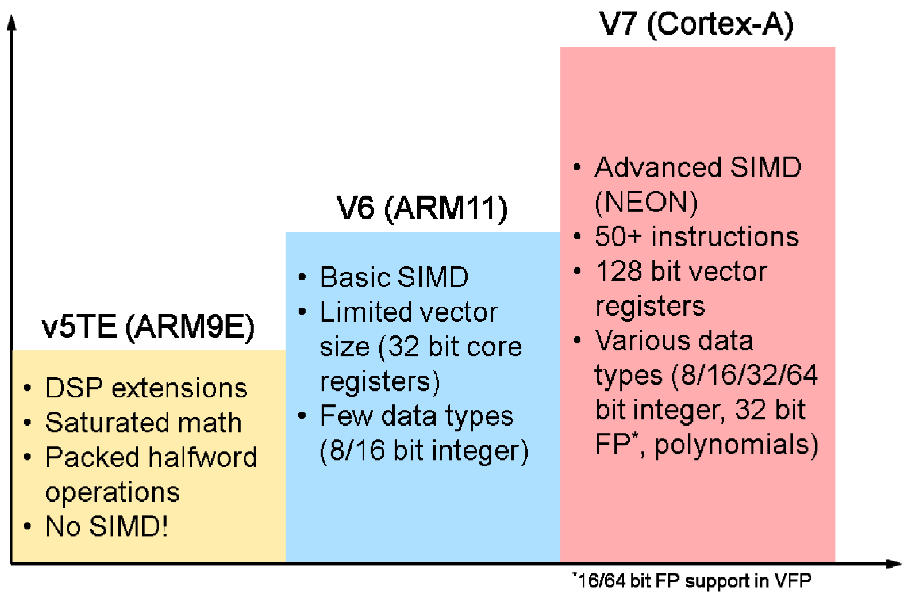
With smarter applications on mobile devices and shrinking process technologies it became worthwhile looking into this again.

## SIMD in ARM processor cores

### Advanced DSP and SIMD (ARMv6)

ARM started an effort of adding advanced DSP and basic SIMD instructions with architecture v6 (Figure 2). This flavor of a SIMD instruction set is executed in the regular integer pipeline and was therefore at the same time rather simple to implement, and very effective at what it had been designed for.[2]

Since vectors in v6 SIMD are stored in core registers, the main limitation of this approach is a total vector size of only 32 bits, i.e. the maximum vector size is four (eight bit elements, e.g. pixel data).


V7 (Cortex-A)
• Advanced SIMD (NEON)
• 50+ instructions
• 128 bit vector registers
• Various data types (8/16/32/64 bit integer, 32 bit FP*, polynomials)
V6 (ARM11)
• Basic SIMD
• Limited vector size (32 bit core registers)
• Few data types (8/16 bit integer)
v5TE (ARM9E)
• DSP extensions
• Saturated math
• Packed halfword operations
• No SIMD!
*16/64 bit FP support in VFP

RM architecture

With some rather well designed two-element vector instructions v6 SIMD helps in many signal processing tasks involving complex numbers arithmetic, where real and imaginary part are represented in a 16 bit fixed point format. That way an entire complex number can be stored in a

[2] Sloss, M. et al, ARM System Developer's Guide, Elsevier, 2004, pp. 549-559

single data word. With the availability of these instructions in the just announced Cortex-M4 processor core, v6 SIMD proves to be far from obsolete.

Currently ARMv6 SIMD instructions won't be generated automatically by the RealView C compiler. These instructions are accessible via assembler language and compiler intrinsics only.

## Advanced SIMD and NEON (ARMv7-A)

With multimedia content becoming ubiquitous even in the power-sensitive market for mobile devices, the introduction of a comprehensive set of SIMD operations became essential to maintaining a leading position in the embedded processor market.

With architecture v7-A, application processors (all members of the ARM Cortex-A family) got the option to implement a SIMD instruction set which ARM trademarked "NEON". Compatible cores from other vendors will use different names (e.g. VeNum in Qualcomm's Scorpion core) although it is rather likely that NEON will become a synonym for the generic term "Advanced SIMD".

The goal of NEON is to provide a powerful, yet comparatively easy to program SIMD instruction set that covers integer data types of up to 64 bit width as well as single precision floating point (32 bit). Requiring wide data paths, NEON cannot be part of the ALU pipeline of the processor. Instead it shares its sixteen 128 bit registers with the vector floating point unit. This additional hardware complexity aligns nicely with new possibilities that modern ASIC technologies offer in terms of power consumption and gate count.

The NEON instruction set is well defined and relatively easy to understand. Developers familiar with the ARM instruction sets will be able to write NEON code without too much effort. ARM has structured the instruction syntax according to different data types, result behavior, etc. The challenge is rather in memorizing which combinations are available for each base instruction.

With Advanced SIMD being executed inside the processor core, it has several benefits over a dedicated DSP:

- Only one compiler for code generation needed
- Instruction set common for all NEON implementations
- Same instruction stream and flow control as rest of application
- No configuration necessary (other than enabling)
- Same memory interface benefits from caches
- No synchronization overhead

Executed on the same processor core, NEON performance is influenced by context switching overhead, non-deterministic memory access latency (cache/MMU access) and interrupt handling.

Advanced SIMD cannot compete with DSP off-loading in every field, but shines in programs where an application interacts a lot with the signal processing part of the code. In an AMP-like situation in which a DSP could operate largely independently of the main CPU, having a silicon side-kick could give a significant performance advantage.

## NEON Architecture

NEON is integrated into the ARM core as a coprocessor. Since NEON shares registers with the vector floating point unit (VFP), the latter will always be present in cores with Advanced SIMD extensions.

NEON stores operands and results in a set of 16 quad-word registers (q0—q15), that can be used alternatively as 32 double-word registers (d0—d31). Values can be moved between NEON registers and core registers (r0—r15), as well as between memory and NEON registers. Data types supported by NEON are vectors of 8, 16, 32, and 64 bit integer, as well as single precision (32 bit) floating point values and polynomials.

The Advanced SIMD instruction set comprises of arithmetic/logical instructions including standard operations (e.g. add, subtract, and multiply), shift and complex bit operations, as well as complex operations like square root, reciprocal estimate. All of these instructions will accept modifiers determining operand and result data types and optional rounding, saturation, doubling/halving behavior.

For accessing memory, there are various instructions available that support simple streaming of scalar values between consecutive memory addresses and vectors, and data structure streaming, where corresponding elements of adjoining data structures will be regarded as vectors.

```
VADD.I16  D0,D1,D2      ; Four 16-bit integer additions
VSUBL.I64 Q8,D1,D5      ; Two 32-bit integer subtraction, 64 bit result
VMUL.I16  D1,D7,D4[2]   ; Four 16 bit integer by 16-bit scalar mult.
VQADD.S16 D0,D1,D2      ; Four 16-bit integer saturating additions
VLD1.32   {D0,D1},[R1]!; load 4 32-bit elements into D0 and D1,
                        ; and update R1
VST1.32   {D0,D1},[R0]!; store 4 32-bit elements from D0 and D1,
                        ; and update R0
```

**Figure 3 NEON instruction examples**

Hand-coding NEON instructions will enable an expert to write the fastest/smallest possible implementation of a given algorithm, but this comes at a price. Even well written assembler code is harder to comprehend than well written code in a higher language. It is often not possible to tell whether a side effect of an instruction is part of the algorithm or not. Accordingly, maintenance and development costs are quite high.

In modern superscalar CPU cores, the processing pipelines are normally much too complex to be fully understood by even better-than-average developers. Comprehending the exact scheduling implications that an instruction might cause with respect to other instructions got extremely difficult in superscalar cores. The additional NEON complexity with feedback into the core pipeline doesn't exactly help. A quote from the Cortex-A8 Technical Reference Manual makes it quite clear:

*The complexity of the processor makes it impossible to guarantee precise timing information with hand calculations. The timing of an instruction is often affected by other concurrent instructions, memory system activity, and additional events outside the instruction flow. Describing all possible instruction interactions and all possible events taking place in the processor is beyond the scope of this document. Only a cycle-accurate model of the processor can produce precise timings for a particular instruction sequence.*

When assembler programming is considered, it is important to understand that the ARM Architecture Reference Manual defines Advanced SIMD on the instruction set and functional level only. Its actual hardware implementation, connection to the core logic and memory interface differs even within the ARM Cortex-A family (Figure 4).

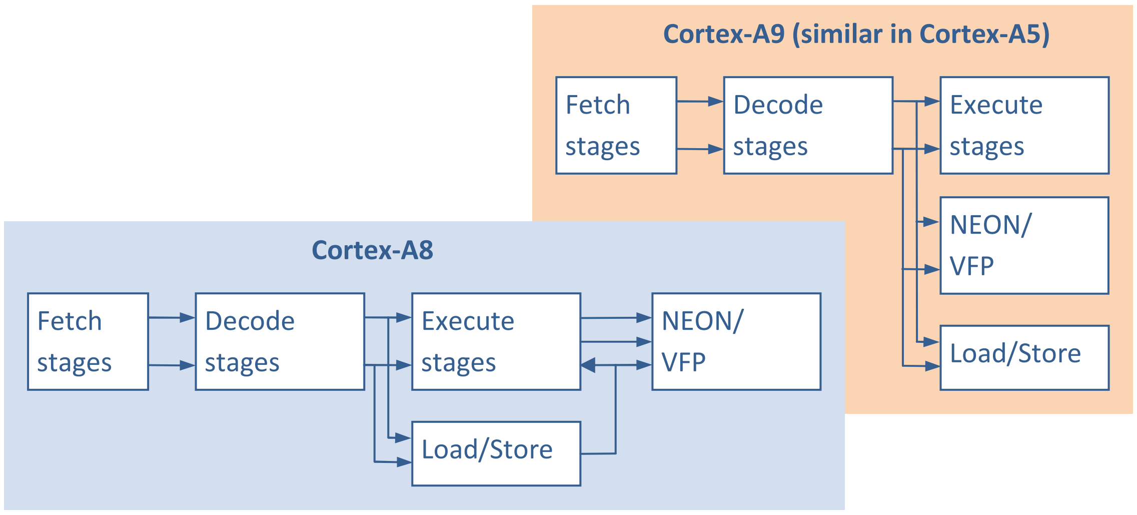


Figure 4NEON integration

Consequently, NEON code that executes nearly optimal on one ARM CPU might not deliver the same performance on others. While compatible on the instruction set level, for optimal results a separate version of hand-optimized assembler code for each CPU core will have to be maintained.

## Accessing NEON from C

Rather than programming in assembler, we might seriously consider leaving it to the experts to schedule our instructions properly. These experts would be the C compiler development team.

But first and foremost, two things have to be considered when executing NEON code within a C program:

- Make sure the co-processors are actually enabled, or you will end up in the "Undefined Instruction" exception handler.[3]
- If you are planning to process single precision floating point in NEON, remember that NEON floating point operations are not strictly IEEE754 compliant. Let your compiler know that you don't want that; otherwise it can't employ NEON for the task.

Having said this we will now introduce two approaches to tap the power of NEON in a C program: Compiler intrinsics (or builtins in GNU jargon) and auto-vectorization.

### Compiler intrinsics

Intrinsics are function-like symbols, which will cause the compiler to inline a corresponding instruction sequence.[4] In some cases, intrinsics allow you to implement your algorithm efficiently

[3] Same is true, of course, for assembler programs. But these guys usually remember those little details.

*and* generic at the same time, as intrinsics will be translated to according assembler instructions depending on the target architecture. In most cases however, the compiler will generate a specific instruction (sequence) and complain if that isn't supported by the target architecture. In the RealView Compiler Tools (RVCT), intrinsics are meant to replace the "inline assembler" mechanism, which has been deprecated for ARMv7 cores (no Thumb support). Looking like regular function calls, intrinsics can also help to keep software tool chain independent by defining them as regular functions for compilers that don't support intrinsics.

To access Advanced SIMD instructions using RVCT intrinsics you will have to include a header file called *arm_neon.h*. This file is huge and defines an intrinsic for every NEON instruction including all its variants. In order to use vectors in C code, special data types have been defined (Figure 5)

```
int32x4_t   // vector of four 32bit elements
uint16x8_t  // vector of eight 16bit elements
int8x8x2_t  // array of two vectors with eight 8 bit elements
```

**Figure 5 NEON instrinsics data type examples**

These data types are opaque[5] and the only safe way to access their elements is via intrinsics. The data types are defined by the AAPCS[6] and thus identical to those used by GCC, ensuring compatibility at least on this level.

Let's have a look at an example (Figure 6). The function *fir_neon()* below implements a 64 tap FIR filter, given the parameters `s` (source data), `c` (coefficients), `y` (results), `num_samples` (number of samples). Each element in the output data array gets assigned the sum of weighted input samples.

```
void fir_neon(const int16_t *s, const int16_t *c,
              int32_t *y, const uint32_t num_samples)
{
    uint32_t i, j;
    int32_t result = 0;

    for (i = 0; i < num_samples; i++) {
        result = 0;

        for (j = 0; j<64; j++) {
            result += c[j] * s[i+j];
        }
        y[i] = result;
    }
}
```

**Figure 6 FIR example**

Looking at this nested loop, we find that there is some potential of using vector arithmetic in the inner loop. Thanks to the associativity of additions, we can shuffle things around. Rather than using just single accumulator (`result`), we could have four of them, one in each "vector lane". After exiting the inner loop, all accumulators will be added to form the result, which will be written out to memory again. An intermediate representation of this idea is shown in Figure 7.

[4] There are other intrinsics, too. See the C Compiler Reference Manual for details.
[5] You really don't want to see their definitions. If you do, you'll find them in *arm_neon.h*.
[6] ARM Architecture Procedure Call Standard, ARM IHI0042D, Appendix A.2

You will have noticed that this implementation requires `num_samples` to be divisible by four. This simplification shall be legal for the purpose of this example. You could always add an *assert()* to make sure.

```
void fir_neon(const int16_t *s, const int16_t *c,
              int32_t *y, const uint32_t num_samples)
{
    uint32_t i, j;
    int32_t o[4];

    for (i = 0; i < num_samples; i++) {
        o[0] = o[1] = o[2] = o[3] = 0;

        for (j = 0; j < 64; j+=4) {
            o[0] += c[j+0] * s[i+j+0];
            o[1] += c[j+1] * s[i+j+1];
            o[2] += c[j+2] * s[i+j+2];
            o[3] += c[j+3] * s[i+j+3];
        }
        y[i] = o[0] + o[1] + o[2] + o[3];
    }
}
```
**Figure 7 FIR: Manually unrolled inner loop**

Let us try now to translate this into NEON intrinsics. First, the header file *arm_neon.h* has to be included to define all vector data types and intrinsics. The array `o[]` will have to be turned into a real vector type. Since we want to process four samples at a time, the type `int32x4_t` (vector of four 32 bit signed integer values) would be appropriate. Note that we cannot directly access source and coefficient data in memory as in the two previous examples. We will have to define two vectors each holding  four samples. Since source data and coefficients are represented by 16 bit integers, we will declare `si` and `ci` as `int16x4_t`. For loading data into a vector, we'll have to use intrinsics. In this case a *vld1_s16()* will do, where the *s16* part indicates that the data type of vector elements would a signed 16 bit integer.

```
#include <arm_neon.h>
void fir_neon(const int16_t *s, const int16_t *c,
              int32_t *y, const uint32_t num_samples)
{
    uint32_t i, j;
    int16x4_t si, ci;
    int32x4_t o;

    for (i = 0; i < num_samples; i++) {
        o = vdupq_n_s32(0);
        for (j = 0; j < 64; j+=4) {
            si = vld1_s16(s+i+j);
            ci = vld1_s16(c+j);
            o  = vmlal_s16(o, si, ci);
        }
        y[i] = vgetq_lane_s32(o, 0)
             + vgetq_lane_s32(o, 1)
             + vgetq_lane_s32(o, 2)
             + vgetq_lane_s32(o, 3);
    }
}
```
**Figure 8 FIR: Implementation using Intrinsics**

After loading both, coefficients and samples vectors, we can now multiply/accumulate `si`, and `ci` with our accumulator vector o, using an `vmlal_s16()`. This all makes sense, but how do we add up the accumulators in the end? Being C programmers, we are going to sum all vector elements by extracting their values. This is a *big* mistake as we will see in Figure 9.

The compiler won't optimize your code when it sees intrinsics. It will slavishly translate your commands in the same order as you wrote them. Using C statements, the unsuspecting developer might believe that detailed knowledge of the NEON instruction set wasn't needed. However, NEON intrinsics are just assembler in disguise, replacing the now deprecated inline assembler in RVCT. Since intrinsics are treated as being similar to a function call, the loop-optimizer assumes potential side effects and thus won't optimize the loop. Since we are using variables instead to hold vectors, the intrinsics approach also hides the register allocation. It is rather easy this way to exceed the number of available NEON registers, causing expensive stack operations for storing and restoring

```
|L1.288|
        VMOV.I32 q0,#0
        MOV      r12,#0
        ADD      r4,r0,r5,LSL #1
|L1.300|
        ADD      r6,r4,r12,LSL #1
        VLD1.16  {d3},[r6]
        ADD      r6,r1,r12,LSL #1
        ADD      r12,r12,#4
        VLD1.16  {d2},[r6]
        CMP      r12,#0x40
        VMLAL.S16 q0,d3,d2
        BCC      |L1.300|
        VMOV.32  r12,d0[0]
        VMOV.32  r4,d0[1]
        ADD      r12,r12,r4
        VMOV.32  r4,d1[0]
        ADD      r12,r12,r4
        VMOV.32  r4,d1[1]
        ADD      r12,r12,r4
        STR      r12,[r2,r5,LSL #2]
        ADD      r5,r5,#1
        CMP      r5,r3
        BCC      |L1.288|
```

**Figure 9 Code generated from intrinsics example**

In order to put NEON intrinsics to their best use, know which instructions you want to see in the generated code. Ideally you would use NEON intrinsics in places where there is not much regular ARM code around. The lack of loop optimization and the overhead of moving data back and forth between ARM core registers and NEON registers might not be obvious at the C source code level. Instruction scheduling by the compiler is not affected by the specified processor type it seems.

To summarize: NEON intrinsics give you access to the Advanced SIMD instruction set, but are hardly easier to use than writing assembler language to begin with (in some cases worse actually) and just like assembler, they require NEON architecture and instruction set knowledge. Their main advantage is the fact that NEON intrinsics can be inlined with other C statements.

The exact impact of scheduling instructions in a certain way becomes harder to grasp with every new generation of CPU cores. Intrinsics might not save you from having to maintain separate versions of your code for different cores.

## Using auto-vectorization

The first paragraph of the previous section said it all: leave it to the experts. The idea of using compilers is to throw arbitrary high-level code at the compiler and it will make best use of all available processor resources. This is *the* fundamental task of a compiler already, so why shouldn't this also apply to SIMD or NEON? With a special license, RVCT supports auto-vectorization. Alternatively GCC supports this, too.

The compiler will analyze your code and find out whether certain loops could be mapped to a vector algorithm. In that sense, vectorization is closely related to loop optimization. RVCT enables loop transformation and vectorization only if highest optimization levels are chosen and the optimization goal is execution time.

Going back to our original FIR filter example (Figure 6), we now enable the RVCT auto-vectorizer with these command line options:

```
armcc --cpu=Cortex-A9 -O3 -Otime --vectorize --remarks -c fir_neon.c
```

The `--remarks` option causes the compiler to print diagnostic information about what it was able to deduce from our source code description. With our unmodified source code the report said:

```
"fir_neon.c", line 66: #2171-D: Optimization: Outer loop unrolled inside
inner loop (i)
"fir_neon.c", line 71: #1679-D: Optimization: Loop vectorized (j)
```

The last statement indicates that the compiler was able to vectorize the loop automatically. But not only that, the compiler seemed to have transformed the loop structure. Let's see for ourselves:

```
|L1.80|
    VLD1.16   {d4},[r1]!
    SUBS      r7,r7,#1
    VLD1.16   {d21},[r1]!
    VLD1.16   {d18},[r12]!
    VMLAL.S16 q0,d4,d18
    VLD1.16   {d5},[r5]!
    VMLAL.S16 q1,d4,d5
    VLD1.16   {d19},[r6]!
    VMLAL.S16 q3,d4,d19
    VLD1.16   {d20},[r4]!
    VMLAL.S16 q8,d4,d20
    VLD1.16   {d22},[r12]!
    VMLAL.S16 q0,d21,d22
    VLD1.16   {d23},[r5]!
    VMLAL.S16 q1,d21,d23
    VLD1.16   {d24},[r6]!
    VMLAL.S16 q3,d21,d24
    VLD1.16   {d25},[r4]!
    VMLAL.S16 q8,d21,d25
    BNE       |L1.80|
```

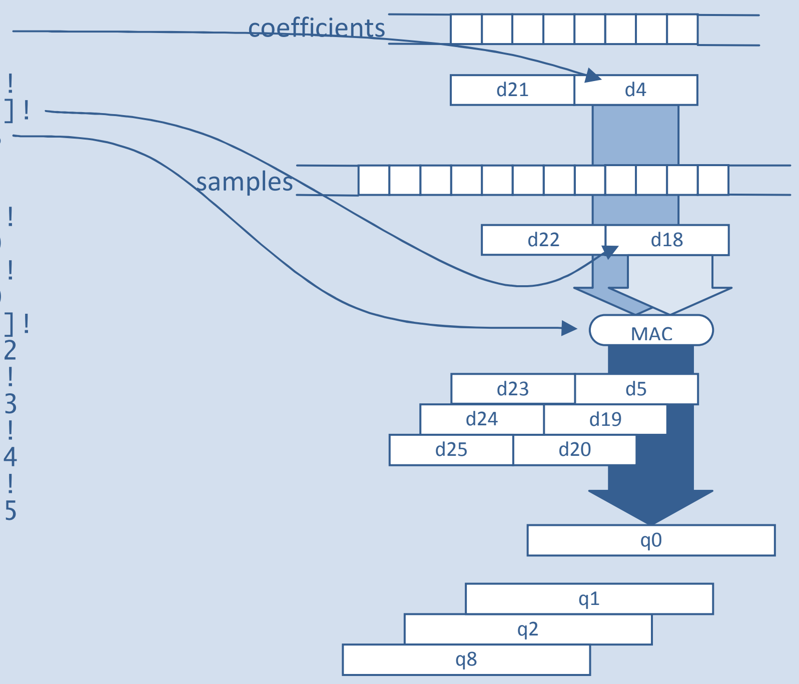


**Figure 10 FIR auto-vectorized: Inner loop**

The first thing we notice in the main loop is that there is a rather long contiguous sequence of NEON instructions. Only the loop counter decrement[7] and the conditional branch are executed in the core pipeline. We can also see what was meant by that remark about the "outer loop unrolled inside inner loop". The inner loop loads eight coefficients into d18 and d21, to be multiplied with eight samples at four different starting points (r12, r5, r6, r4). So not only did the loop get vectorized, we are also processing four iterations of the outer loop in parallel, thereby reusing coefficients data that had been fetched from memory already.

When summing up the individual accumulator vectors, efficient "horizontal addition" is used (VPADD) and the results are transferred back to ARM core registers (Figure 11).

```
VADD.I32 d0,d0,d1
VPADD.I32 d0,d0,d0
VMOV.32  r1,d0[0]
VADD.I32 d0,d2,d3
VPADD.I32 d0,d0,d0
VMOV.32  r12,d0[0]
VADD.I32 d0,d6,d7
VPADD.I32 d0,d0,d0
[…]
VMOV.32  r4,d0[0]
VADD.I32 d0,d16,d17
[…]
VPADD.I32 d0,d0,d0
[…]
VMOV.32  r5,d0[0]
```

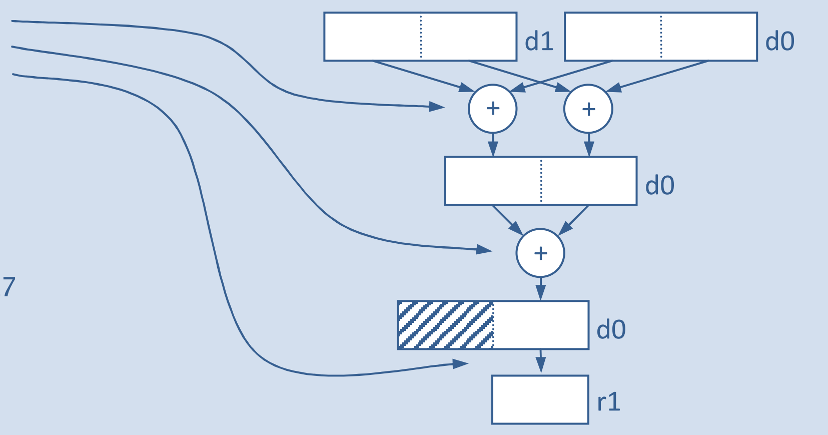


**Figure 11 FIR auto-vectorized: End of inner loop**

NEON wizards might object that this could be done even more efficiently. There is perhaps additional optimization potential using NEON internal register transfers rather than repeated access to the same memory locations. Maybe this is actually true, but considering the development and maintenance costs it may not be worth investigating this except in most critical places. Even proving that a hand optimized assembler solution performs better than another implementation *in a realistic environment* could turn out to be rather difficult.

In an ideal world, auto-vectorization would work with arbitrary code, but of course it doesn't. Just flipping compiler switches will not result in code that executes significantly faster. To get good results developers should follow some basic rules for writing code that makes it easy for the compiler to map operation to available SIMD vectors.

## Know your algorithm

If *you* can see clearly how your source code maps to vector operations, chances are the *compiler* can, too.

[7] Did I write decrement? Thanks to loop transformation the variable gets *decremented*, while in the source code the variable is *incremented*.

## Flow control in loops

With few exceptions, conditionals (if, switch-case) in loops don't vectorize well. The ARM instruction (as opposed to Thumb) encoding of Advanced SIMD does not allow conditional execution. In Thumb-2 conditional execution is available indirectly via the IT instruction. In some cases you can substitute arithmetic operations such as multiplication by a Boolean value.

Don't terminate a loop prematurely (break, continue). The compiler will have to assume that this could happen in any iteration, effectively making it impossible to process a vector of *n* elements per iteration.

## Communicating facts

Let the compiler know as much about your program as you do.

Contributing to the legendary advantage of hand-optimized code is the phenomenon that developers writing assembler code are used to making all sorts of (valid) assumptions about the context which certain code is being used in. Trying the same in C could prove difficult and compilers might not even take advantage of this extra information. Modern compilers like RVCT *do* support extended keywords that allow a developer to specify certain facts (`__promise`, `__expect`, `__restrict`) that escape the expressiveness of the C language. These keywords and occasionally even standard constructs will help the compiler to better understand the algorithm. A loop header "`for (i=0; i<(n & ~3); i++)`", for instance, implies that the number of iterations will always be divisible by four so that in our example the compiler might not have to create extra code for processing samples  that do not fill an entire vector.

Another example: Assuming that according to your program logic, one of the integer-type function parameters can only possibly assume two discrete values. Perhaps splitting the function into two versions, one for each value of this parameter might give better results. Maybe a C++ template could do the hard work for you.

## Memory access

Think about memory access patterns. Inefficient memory access is a major reason for poor performance. A lot of the source code out there, especially legacy code, was not written with multi-gigahertz embedded processors sporting several cache levels in mind. Memory access latency on these devices has a much larger impact to performance than the difference of a single perhaps unnecessary extra instruction in an inner loop. As far as mobile devices are concerned, battery life depends on efficient cache utilization more than on anything else.

Loading streams of consecutive data items is cache friendly *and* can potentially be vectorized. Gathering data elements from (as far as the compiler knows) all over the place does not vectorize nor does it benefit as much from having caches. High performance cores like ARM Cortex-A9 are trying to detect regular access patterns and generate speculative memory accesses to minimize cache misses whenever regular access crosses a cache line.

For accessing data as array, use simple expressions as index. This way, a compiler will have it easier to identify potential vector lanes. A complex index can sometimes be broken down into setting a start pointer from which items will be accessed as array elements.

## Paradox

Ironically, some of the guidelines for getting good auto-vectorization results contradict those guidelines for generating efficient (on ARM cores anyway) non-vectorized code:

| Auto-vectorization | Regular code |
|---|---|
| Use arrays for accessing adjacent objects. Avoid pointer access, which often can't be translated into vector operations | Avoid arrays and use pointer arithmetic. The compiler will use more efficient addressing modes (e.g. post-increment). |
| Make your local variables that represent vector elements as small as possible, so that the compiler packs up more into a single register. | Always use 32 bit types for local variables. They'll occupy a register anyway and you are spared the zero/sign-extension overhead. |

## Optimization Trap

With all the effort we put into writing new C code that vectorizes well, we should revisit accrued wisdom that while once true, may be obsolete today and turn out to be detrimental to performance on a modern CPU. Example: The generic C implementation of a color-space conversion routine in a popular video codec uses a look-up-table (LUT) to avoid an integer multiply-add operation. While this operation used to be expensive and the LUT access might have been quicker on those small cores of the time, the situation has changed completely: LUTs are often associated with poor caching behavior and might provoke slow and power-hungry external memory access. Irregular by nature, LUT access makes it impossible to vectorize this function. In contrast, MAC operations are quite efficient these days and execute in the pipeline.

## Verify results

While a vectorizing compiler might save you from having to *come up with* clever instruction sequences, it still helps to *understand* the generated code in order to judge whether the compiler did a good job. A good indicator is the RVCT output generated by the compiler option `--remarks`. The GCC equivalent would be `-ftree-vectorizer-verbose=`*x*, where the argument specifies how much information the compiler would generate.

## Conclusion

Vectorizing compilers do exist and they do a great job if the description of an algorithm can be broken down by the compiler. Sometimes the compiler needs extra help. Restructuring C code or using extended keywords allows the compiler to extract information from our code. NEON/SIMD knowledge is beneficial to writing SIMD-friendly C code. Thanks to a regular structure and more efficient memory access, code that is SIMD-friendly often performs better even on those cores that don't implement NEON.

The advantages from having generic implementations of algorithms shouldn't be underestimated in product development.

Auto-vectorization is not perfect and, just as regular code optimization, will probably never be. It will remain a tradeoff between performance and implementation cost. If you think about it, the same is true for *any* type of code. However, the number of critical software routines where currently hand-optimized assembler is essential to ensure high performance, will decrease.

## Further Reading

Introducing NEON™ Development Article
(http://infocenter.arm.com/help/topic/com.arm.doc.dht0002a/index.html)

NEON™ Support in Compilation Tools Development Article
(http://infocenter.arm.com/help/topic/com.arm.doc.dht0004a/index.html)

RealView® Compilation Tools Compiler User Guide – "Using the NEON Vectorizing Compiler"
(http://infocenter.arm.com/help/topic/com.arm.doc.dui0205i/BABEGGJG.html)

RealView® Compilation Tools Compiler Reference Guide – "Using NEON Support"
(http://infocenter.arm.com/help/topic/com.arm.doc.dui0348b/Badcdfad.html)