WHITE PAPER

ARM NEON support in the ARM compiler

September 2008

Introduction

This paper provides a simple introduction to the ARM NEON™ SIMD (Single Instruction Multiple Data) architecture. It discusses the compiler support for SIMD, both through automatic recognition and through the use of intrinsic functions.

The NEON instruction set is a hybrid 64/128 bit SIMD architecture extension to the ARM v7-A profile, targeted at multimedia applications. SIMD is where one instruction acts on multiple data items, usually carrying out the same operation for all data. The NEON unit is positioned within the processor to enable it to share the CPU resources for integer operation, loop control, and caching, significantly reducing the area and power cost compared to a hardware accelerator. It also uses a much simpler programming model, since the NEON unit is within the same address space as the application, so there is no need for the programmer to search for ad-hoc concurrency and integration points.

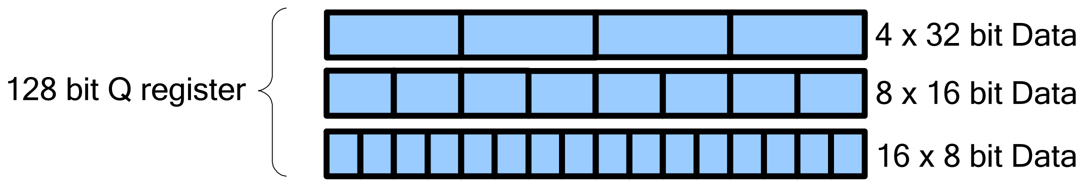
The NEON instruction set was designed to be an easy target for a compiler, including low cost promotion/demotion and structure loads capable of accessing data from their natural locations rather than forcing alignment to the vector size.

The RealView Development Suite 4.0 Professional supports NEON by using intrinsic functions and assembler, as well as by using the vectorizing compiler on standard C and C++, which will automatically generate SIMD code. The vectorizing compiler greatly reduces porting time, as well as reducing the requirement for deep architectural knowledge.

Overview of NEON Vector SIMD

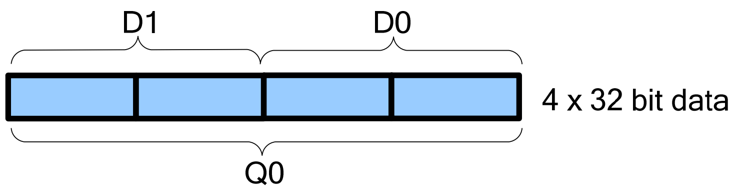
SIMD is the name of the process for operating on multiple data items in parallel using the same instruction. In the NEON extension, the data is organized into very long registers (64 or 128 bits wide). These registers can hold "vectors" of items which are 8, 16, 32 or 64 bits.

The traditional advice when optimizing or porting algorithms written in C/C++ is to use the



natural type of the machine for data handling (in the case of ARM 32 bits). The unwanted bits can then be discarded by casting and/or shifting before storing to memory. The ability of NEON to specify the data width in the instruction and hence use the whole register width for useful information means keeping the natural type for the algorithm is both possible and preferable. Keeping with the algorithms natural type reduces the cost of porting an algorithm between architectures and enables more data items to be simultaneously operated on.

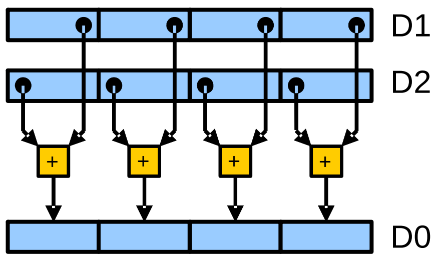
NEON appears to the programmer to have two banks of registers, 64 bit D registers and 128 bit Q registers. In reality the D and Q registers alias each other, so the 64 bit registers D0 and D1 map against the same physical bits as the register Q0.



When an operation is performed on the registers the instruction specifies the layout of the data contained in the source and, in certain cases, destination registers.

Example: Add together the 16 bit integers stored in the 64 bit vector D2 and 64 bit vector D1 storing the resultant items in the 64 bit register D0

`VADD.I16 D0, D1, D2` This instruction will cause four 16 bit adds

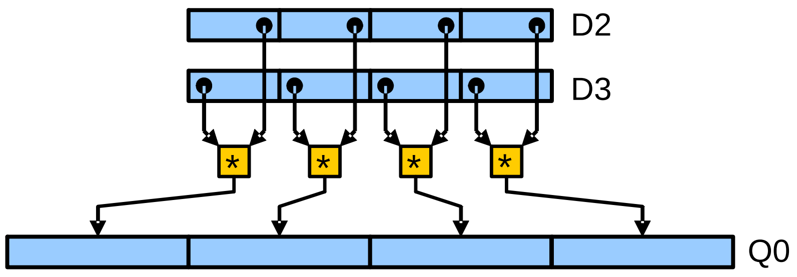


Promotion/demotion of types

Promotion/demotion of types is a very common operation in C. Casting to larger types can be used to avoid overflow or increase precision. Shifting into smaller types enables compatibility at interfaces or reduced memory usage. In contrast with some other SIMD architectures, NEON provides compound operations which combine type promotion with arithmetic operations. This enables NEON code to make better use of the register file and use fewer instructions.

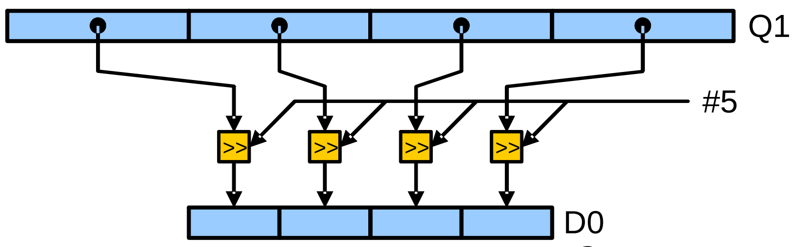
Example: Multiply together the 16 bit integers stored in the 64 bit vectors D2 and D3 storing the resultant items in the 128 bit register Q0

`VMUL.I32.S16 Q0, D2, D3;` This instruction will cause four widening multiplies



Example: Shift right by #5 the four 32 bit integers stored in 128 bit vector Q1, truncate to 16 bits and store the resultant 16 bit integers in 64 bit register D0

`VSHR.I16.I32 D0, Q1,#5` This instruction will cause four narrowing shifts



Structure load and store operations

Often items are not held in memory as simple arrays, but rather arrays of structures for logically grouped data items.

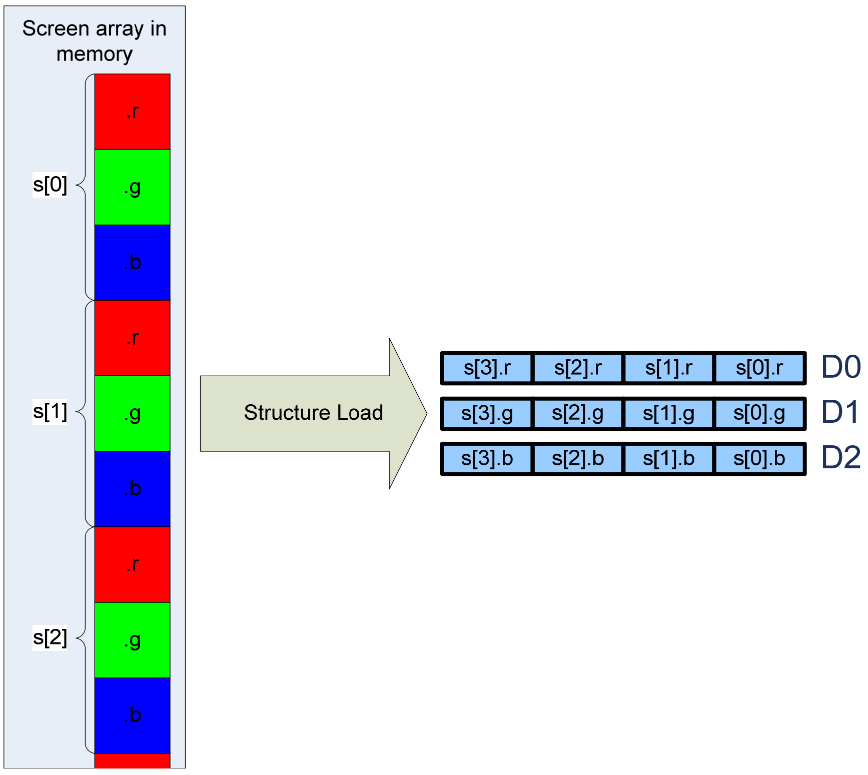
For example it is common to find a screen represented as an array of structures of pixels rather than split into three arrays of red, green and blue items. Storing all components of pixel data together enables faster operation for common operations such as colour conversion or display, however it can cause difficulties for some SIMD implementations.

```
struct rgb_pixel
{
         short r; /* Red */
         short g; /* Green */
         short b; /* Blue */
}s[X_SIZE*Y_SIZE]; /* screen */
```

The NEON unit includes special structure load instructions which can load whole structures and spilt them accordingly across multiple registers.

Example: Load 12 16 bit values from the address stored in R0, and split them over 64 bit registers D0, D1 and D2. Update R0 to point at next structure.

```
VLD3.16 {D0,D1,D2}, [R0]!
```


Screen array in memory
s[0]
.r
.g
.b
s[1]
.r
.g
.b
s[2]
.r
.g
.b
Structure Load
s[3].r s[2].r s[1].r s[0].r D0
s[3].g s[2].g s[1].g s[0].g D1
s[3].b s[2].b s[1].b s[0].b D2

Structure load and store better matches how engineers write code, so code usually does not need to be rewritten to take advantage of it.

Writing NEON code using the standard ARM compiler

The standard tools shipped with RealView Development Suite 4.0 have support for NEON directly in the assembler and embedded assembler. The compiler also provides NEON support using pseudo functions called intrinsics. Intrinsic functions compile into one or more NEON instructions which are inserted at the call site. There is at least one intrinsic for each NEON instruction, with multiple intrinsic functions where needed for signed and unsigned types.

Using intrinsics, rather than programming in assembly language directly, enables the compiler to schedule registers, as well as giving the programmer easy access to C variables and arrays.

Using vector registers directly from assembler could lead to programming errors such as a 64 bit vector containing data of 8 bits wide is operated upon by a 16 bit adder. These kind of faults can be very difficult to track down as only particular corner cases will trigger an erroneous condition. In the previous addition example, the output will only differ if one of the data items overflows into another. Using intrinsics is type-safe and will not allow accidental mixing of signed/unsigned or differing width data.

Accessing vector types from C

The header file arm_neon.h is required to use the intrinsics and defines C style types for vector operations. The C types are written in the form :

`uint8x16_t`	Unsigned integers, 8 bits, vector of 16 items - 128 bit “Q” register
`int16x4_t`	Signed integers, 16 bits, vector of four items - 64 bit "D" register

As there is a basic incompatibility between scalar (ARM) and vector (NEON) types it is impossible to assign a scalar to a vector, even if they have the same bit length. Scalar values and pointers can only be used with NEON instructions that use scalars directly.

Example: Extract an unsigned 32 bit integer from lane 0 of a NEON vector

```
result =  vget_lane_u32(vec64a, 0)
```

Vector types are not operable using standard C operators except for assignment, so the appropriate VADD should be used rather than the operator “+”.

Where there are vector types which differ only in number of elements (uint32x2_t, uint32x4_t) there are specific instructions to ‘assign’ the top or bottom vector elements of a 128 bit value to a 64 bit value and vice-versa. This operation does not use any code space as long as the registers can be scheduled as aliases.

Example: Use the bottom 64 bits of a 128 bit register

```
vec64 = vget_low_u32(vec128);
```

Access to NEON instructions using C

To the programmer intrinsics look like function calls. The function calls are specified to describe the target NEON instruction as well as information about the source and destination types.

Example: To add two vectors of 8 bytes, putting the result in a vector of 8 bytes requires the instruction

```
VADD.I8 dx, dy, dz
```

This can be provoked by using either of the following intrinsic functions

```
int8x8_t vadd_s8(int8x8_t a, int8x8_t b);
uint8x8_t vadd_u8(uint8x8_t a, uint8x8_t b);
```

The use of separate intrinsics for each type means that it is difficult to accidentally perform an operation on incompatible types because the compiler will keep track of which types are held in which registers. The compiler can also reschedule program flow and use alternative faster instructions; there is no guarantee that the instructions that are generated will match the instructions implied by the intrinsic. This is especially useful when moving from one micro-architecture to another.

Programming using NEON intrinsics

The process of writing optimal NEON code directly in the assembler or by using the intrinsic function interface requires a deep understanding of the data types used as well as the NEON instructions available.

Possible SIMD operations become more obvious if you look at how an algorithm can be split into parallel operations.

Commutative operations (add, min, max) are particularly easy from a SIMD point of view.

Example: Add 8 numbers from an array

```
unsigned int acc=0;
for (i=0; i<8;i+=1)
{
   acc+=array[i]; // a + b + c + d + e + f + g + h
}
```

could be split into several adds ((a+e) + (b+f))+((c+g) + (d+h))

and recoded in C as:

Continued on next page…

```
unsigned int acc1=0;
unsigned int acc2=0;
unsigned int acc3=0;
unsigned int acc4=0;
for (i=0; i<8;i+=4)
{
   acc1+=array[i];   // (a, e)
   acc2+=array[i+1]; // (b, f)
   acc3+=array[i+2]; // (c, g)
   acc4+=array[i+3]; // (d, h)
}
acc1+=acc2;          // (a+e) + (b+f)
acc3+=acc4;          // (c+g) + (d+h)
acc1+=acc3;          // ((a+e) + (b+f))+((c+g) + (d+h))
```

It should be more apparent now that we could use a vector register holding four 32 bit values for the accumulator and temp registers then do the operation using SIMD instructions. Now extending for any multiple of four:

```
#include <arm_neon.h>
uint32_t vector_add_of_n(uint32_t* ptr, uint32_t items)
{
   uint32_t result,* i;
   uint32x2_t vec64a, vec64b;
   uint32x4_t vec128 = vdupq_n_u32(0); // clear accumulators

   for (i=ptr; i<(ptr+(items/4));i+=4)
   {
      uint32x4_t temp128 = vld1q_u32(i); // load 4x 32 bit values
      vec128=vaddq_u32(vec128, temp128); // add 128 bit vectors
   }

   vec64a = vget_low_u32(vec128);  // split 128 bit vector
   vec64b = vget_high_u32(vec128); //  into 2x 64 bit vectors

   vec64a = vadd_u32 (vec64a, vec64b); // add 64 bit vectors together

   result =  vget_lane_u32(vec64a, 0); // extract lanes and
   result += vget_lane_u32(vec64a, 1); //  add together scalars
   return result;
}
```

The `vget_high_u32` and `vget_low_u32` are not analogous to any NEON instruction, but instruct the compiler to reference the Q register used to store `vec128` as two separate 64 bit D registers.

These operations therefore do not translate into actual code, but will have an effect on which registers are used to store `vec64a` and `vec64b`.

Depending on the version of the compiler, target processor and optimization options, the code generated becomes:

```
vector_add_of_n PROC
         VMOV.I8  q0,#0
         BIC      r1,r1,#3
         ADD      r1,r1,r0
         CMP      r1,r0
         BLS      |L1.36|
 |L1.20|
         VLD1.32  {d2,d3},[r0]!
         VADD.I32 q0,q0,q1
         CMP      r1,r0
         BHI      |L1.20|
 |L1.36|
         VADD.I32 d0,d0,d1
         VMOV.32  r1,d0[1]
         VMOV.32  r0,d0[0]
         ADD      r0,r0,r1
         BX       lr
         ENDP
```

Generating NEON code using the vectorizing compiler

The vectorizing compiler is part of the ARM RealView Development Suite 4.0 Professional. The vectorizing compiler uses supercomputer compiler techniques to evaluate vectorizable loops and potential SIMD applications.

The compiler optimizes more efficiently with readable C/C++. Although it can generate some SIMD code without source modification, certain coding styles can promote more optimal output. Where the vectorizer finds code with potential SIMD opportunities but does not have enough information it can generate a ‘remark’ to the user to prompt changes to the source code that can provide more useful information.

Although these modifications help the vectorizing compiler they are all standard C notation and will allow recompilation with any C99* compliant compiler.

*C99 required for parsing of keyword “`restrict`”. In other compilation modes, armcc also allows the use of the equivalent ARM-specific extension “`__restrict`”.

Compiler command line options

With the vectorizing add-on installed the compiler can be told to generate SIMD code simply by switching “vectorize” on the command line.

SIMD code is usually bigger than the equivalent ARM code due to array cleanup and other issues (see later), and the CPU should be set to a processor which has NEON hardware (at the time of writing the Cortex-A8 is the only supported processor)

To generate fast SIMD code you should use the command line:

```
armcc --cpu=Cortex-A8 -O3 -Otime --vectorize ...
```

Without an installation of the vectorizing add-on this command will respond with an error message.

Using the vectorizing compiler on the addition example

Keeping with the vector add example from page 7 we can write the code in standard C much more succinctly:

```
/* file.c */
unsigned int vector_add_of_n(unsigned int* ptr, unsigned int items)
{
    unsigned int result=0;
    unsigned int i;
    for (i=0; i<(items*4);i+=1)
    {
        result+=ptr[i];
    }
    return result;
}
```

Note: By using `(items*4)` we are telling the compiler that the size of the array is a multiple of four. Although this is not required for the vectorizer to create SIMD code, it provides the compiler with extra information about the array. In this case it knows the array can be consumed with vector arrays and does not require any extra scalar code to handle the cleanup of any ‘spare’ items.

Compile: “`armcc --cpu=Cortex-A8 -O3 –c -Otime –-vectorize file.c`”
Viewing the generated code with: “`fromelf –c file.o`”

```
vector_add_of_n PROC
        LSLS     r3,r1,#2
        MOV      r2,r0
        MOV      r0,#0
        BEQ      |L1.72|
        LSL      r3,r1,#2
        VMOV.I8  q0,#0
        LSRS     r1,r3,#2
        BEQ      |L1.48|
|L1.32|
        VLD1.32  {d2,d3},[r2]!
        VADD.I32 q0,q0,q1
        SUBS     r1,r1,#1
        BNE      |L1.32|
|L1.48|
        CMP      r3,#4
        BCC      |L1.72|
        VADD.I32 d0,d0,d1
        VPADD.I32 d0,d0,d0
        VMOV.32  r1,d0[0]
        ADD      r0,r0,r1
|L1.72|
        BX       lr
```

This disassembly generated is different to the earlier intrinsic function example; the main reason behind this is that **the hand coded example misses the important corner case** where the array is zero in length.

Although the code is longer than the handwritten example the **key parts of the routine** (the inner loop) are the **same length** and contain the same instructions, this means that the time difference for execution is trivial when the dataset becomes reasonable in size.

Things to watch out for with automatic vectorization

The vectorizing compiler works best when it can see what you are doing. Simply written code which is easy for a human to understand is much easier to vectorize than code highly tuned for a specific processor. Some typical optimization tricks that can cause problems are listed below, along with better solutions.

The C pointer aliasing

One of the biggest challenge in optimizing Standard C (ISO C90) comes from passing pointers which may (according to the standard) point at the same or overlapping datasets. In Standard C this issue is commonly worked around by assigning the data to local variables. Holding data in local variables from one loop iteration to another has to assume the availability of a certain number of registers in a particular processor design.

This makes the code much more difficult to read and requires the engineer to write this variable caching code as well as the setup and exit portions explicitly. Unless written with explicit knowledge of the compilation target pipeline and register availability, it can often generate suboptimal code (although usually better than non optimized readable code). Obviously if code is written in this way it will behave badly when simply ported to an architecture or instruction set with a different working register set, for example moving from ARM (14 working registers) to Thumb (8 working registers).

As the C standard has evolved this issue has been addressed by adding the keyword "`restrict`" to C99 and C++. Adding "`restrict`" to a pointer declaration is a promise that the data referenced through that pointer does not alias with anything else the function will modify using another pointer. This leaves the compiler to do the work in setup and exit restrictions, preload data with plenty of advance notice, and cache results.

The ARM Compiler allows the use of the keyword "`__restrict`" in all modes.

"Natural" types

Often algorithms will be written to expect certain types for legacy, memory size or peripheral reasons. It is common to cast these up to the "natural" type for the CPU as mathematical operations on "natural" sized data are usually faster, and truncation and overflow are only calculated when data gets passed or stored. Doing this for an algorithm requires knowledge of the target CPU.

NEON SIMD has 8, 16, 32 and 64 bit type support, using the smallest valid type means that more data can be held in each SIMD register and executed in parallel, so when operating on 16 bit data, 8 items can be operated on with the same instruction.

Array grouping

For CPU designs which have few memory pointers (such as x86), it is common to group several arrays into one. This can enable several different offsets from the same pointer to allow access to different parts of the data. Grouping arrays in this way can confuse the compiler into thinking that the offsets may cause overlapping datasets. Avoid this unless you can guarantee that there are no writes into the array. Splitting composite arrays into component restrict pointers should remove this risk.

Inside knowledge

For an anonymous array to be turned into SIMD code, the compiler has to assume the size may be anywhere between 0 and 4GB. Without additional information the compiler has to generate setup code which tests if the array is too small to consume a whole vector register as well as cleanup code which consumes the last few items in the array using the scalar pipeline.

In some cases these sizes are known at compile time and should be specified directly rather than passed as arguments. In other cases it is common for the engineer to know more about the array layouts than the system, for instance, screen sizes tend to be expressed in powers of 2, so an engineer may know that a loop condition will always be a multiple of 2, 4, 8, 16, 32 etc. and be able to write their code for this effect. Passing an array size in raw form does not communicate this to the compiler, but passing the bounds as “size/16” and using “size*16” as the loop iterations enables the compiler to see this and remove cleanup code.

Write code with NEON instructions in mind

Write loops to imply SIMD

Writing loops to use all parts of the structure together is good practice as it exploits the cache better. Using the vectorizing compiler it becomes more important as each part of the structure needs to be accessed in the same loop:

```
for (...) { outbuffer[i].r = ....; }
for (...) { outbuffer[i].g = ....; }
for (...) { outbuffer[i].b = ....; }
```

Splitting the work here into three separate loops may have been done for a machine with very few working registers. A simple rewrite to fuse the loop will give better results on all cached processors as well as allowing the compiler to vectorize the loop:

```
for (...) {
  outbuffer[i].r = ....;
  outbuffer[i].g = ....;
  outbuffer[i].b = ....;
}
```

Write structures to imply SIMD

NEON structure loads require that the structure contains equal sized elements. The compiler will only try to vectorize loads if they use all data items from the structure.

In some cases structures are padded to maintain alignment, in the example below some padding has been added to ensure that each pixel is in an aligned 32 bit word. This might be done to facilitate the use of special instructions such as ARMv6 SIMD extensions.

```
struct aligned_pixel
{
   char r;
   char g;
   char b;
   char not_used; /* Padding used to keep r aligned on 32 bits */
}screen[10];
```

The use of padding will cause the structure to not match against available NEON scatter/gather loads and as such will not vectorize. There is no benefit in NEON of aligning every “r” component as NEON can load unaligned structures in many cases without penalty. Removing the padding in this case is preferable as the normal arrays will be 75% the size of aligned arrays.

Another possible difficulty is where items in the structure are of different lengths. The NEON structure loads require that all items in the structure are the same length. Therefore the compiler will not attempt to use vector loads for the following code.

```
struct pixel
{
   char r;
   short g; /* green contains more information */
   char b;
}screen[10];
```

If “g” is to be held with higher precision consider widening the other elements to allow vectorizable loads.

Tell the compiler where to unroll inner loops

```
#pragma unroll (n)
```

Can be used before a “for” statement to tell the compiler to unroll loops a certain number of times. Typically this is of use to totally unroll small internal loops which enables the compiler to vectorize an outer loop in more complex algorithms.

A real life example – FIR filter

The FIR filter is a reasonably common and compact example of a vectorizable loop. In this final part of the paper we will look at the process of generating NEON code using both automatic compilation and using intrinsic functions.

```
/*fir.c*/
void fir(short * y,
   const short *x, const short *h,
   int n_out, int n_coefs)
{
   int n;
   for (n = 0; n < n_out; n++)
   {
      int k, sum = 0;
      for(k = 0; k < n_coefs; k++)
      {
         sum += h[k] * x[n - n_coefs + 1 + k];
      }
      y[n] = ((sum>>15) + 1) >> 1;
   }
}
```

Using intrinsics for SIMD

If we look at the code, the central loop looks vectorizable; it is a summation of the product of two arrays, each of which have a stride of one. By comparison the outer loop doesn't look vectorizable; we do not know how many n_coefs we have.

We know that the accumulation operation is commutative, so to vectorize it we could spilt it thus:

```
   for(k = 0; k < n_coefs/4; k++)
   {
      sum0 += h[k*4] * x[n - n_coefs + 1 + k*4];
      sum1 += h[k*4+1] * x[n - n_coefs + 1 + k*4 + 1];
      sum2 += h[k*4+2] * x[n - n_coefs + 1 + k*4 + 2];
      sum3 += h[k*4+3] * x[n - n_coefs + 1 + k*4 + 3];
   }
   sum = sum0 + sum1 + sum2 + sum3;
```

One side effect of this would be to quarter the loop overhead. This works for the majority of the array, but we do not know that the number of array items is a multiple of four, so extra code must be added to deal with this situation:

```
   if(n_coefs % 4)
   {
      for(k = n_coefs - (n_coefs % 4); k < n_coefs; k++){
         sum += h[k] * x[n - n_coefs + 1 + k];
      }
   }
```

We can now rewrite the main loop to use a vector operation instead of four scalars:

```
   sum = 0;
   result_vec = vdupq_n_s32(0); /* Clear the sum vector */

   for(k = 0; k < n_coefs / 4; k++){
      h_vec = vld1_s16(&h[k*4]);
      x_vec = vld1_s16(&x[n - n_coefs + 1 + k*4]);
      result_vec = vmlal_s16(result_vec, h_vec, x_vec);
   }
   sum += vgetq_lane_s32(result_vec, 0) +
      vgetq_lane_s32(result_vec, 1) +
      vgetq_lane_s32(result_vec, 2) +
      vgetq_lane_s32(result_vec, 3);
```

The final code is ARM compiler specific and looks like this:

```
#include <arm_neon.h>
void fir(short * y,
   const short *x, const short *h,
   int n_out, int n_coefs)
{
   int n, k;
   int sum;
   int16x4_t h_vec;
   int16x4_t x_vec;
   int32x4_t result_vec;

   for (n = 0; n < n_out; n++)
   {
/* Clear the scalar and vector sums */
      sum = 0;
      result_vec = vdupq_n_s32(0);

      for(k = 0; k < n_coefs / 4; k++)
      {
/* Four vector multiply-accumulate operations in parallel */
         h_vec = vld1_s16(&h[k*4]);
         x_vec = vld1_s16(&x[n - n_coefs + 1 + k*4]);
         result_vec = vmlal_s16(result_vec, h_vec, x_vec);
      }
/* Reduction operation - add each vector lane result to the sum */
      sum += vgetq_lane_s32(result_vec, 0);
      sum += vgetq_lane_s32(result_vec, 1);
      sum += vgetq_lane_s32(result_vec, 2);
      sum += vgetq_lane_s32(result_vec, 3);
/* consume the last few data using scalar operations */
      if(n_coefs % 4)
      {
         for(k = n_coefs - (n_coefs % 4); k < n_coefs; k++)
            sum += h[k] * x[n - n_coefs + 1 + k];
      }
/* Store the adjusted result */
      y[n] = ((sum>>15) + 1) >> 1;
   }
```

}

Using the vectorizing compiler

Compile the unmodified FIR code from page 13 with

```
armcc -O3 -Otime --vectorize --cpu=Cortex-A8 –c fir.c
```

No source code changes are required, the code stays **portable**, **readable** and you know that you have not changed the behaviour by accident.

Side by side output comparison

NEON Intrinsics

```
||fir|| PROC
        PUSH      {r4-r9,lr}
        MOV       r6,#0
        LDR       r5,[sp,#0x1c]
        ASR       r4,r5,#31
        ADD       r12,r5,r4,LSR #30
        B         |L1.200|
|L1.24|
        MOV       lr,#0
        VMOV.I8   q0,#0
        MOV       r4,lr
        SUB       r7,r6,r5
        B         |L1.76|
|L1.44|
        ADD       r8,r2,r4,LSL #3
  1     VLD1.16   {d2},[r8]
        ADD       r8,r7,r4,LSL #2
        ADD       r4,r4,#1
        ADD       r8,r1,r8,LSL #1
        ADD       r8,r8,#2
        VLD1.16   {d3},[r8]
        VMLAL.S16 q0,d2,d3
|L1.76|
        CMP       r4,r12,ASR #2
        BLT       |L1.44|
        TST       r5,#3
        VMOV.32   r4,d0[0]
  2     ADD       r4,r4,lr
        VMOV.32   lr,d0[1]
        ADD       r4,r4,lr
        VMOV.32   lr,d1[0]
        ADD       r4,r4,lr
        VMOV.32   lr,d1[1]
        ADD       lr,lr,r4
        BEQ       |L1.176|
        BIC       r4,r12,#3
        SUB       r4,r5,r4
        SUB       r4,r5,r4
|L1.136|
        CMP       r4,r5
        BGE       |L1.176|
  3
```

NEON vectorizing compiler

```
||fir|| PROC
        PUSH      {r4-r11,lr}
        CMP       r3,#0
  4     MOV       r9,#0
        LDR       r7,[sp,#0x24]
        BLE       |L1.208|
        MOV       lr,#1
        ASR       r12,r7,#31
        ADD       r11,r7,r12,LSR #30
|L1.32|
        CMP       r7,#0
        MOV       r5,#0
        BLE       |L1.180|
        VMOV.I8   q0,#0
        SUB       r8,r9,r7
        MOV       r4,r2
        ADD       r12,r1,r8,LSL #1
        ADD       r6,r12,#2
        ASRS      r12,r11,#2
        BEQ       |L1.92|
|L1.72|
        VLD1.16   {d3},[r4]!
  1     SUBS      r12,r12,#1
        VLD1.16   {d2},[r6]!
        VMLAL.S16 q0,d3,d2
        BNE       |L1.72|
|L1.92|
        AND       r12,r7,#3
        CMP       r12,#0
        BLE       |L1.152|
        SUB       r12,r7,r12
        CMP       r12,r7
        BGE       |L1.152|
|L1.116|
        ADD       r4,r2,r12,LSL #1
        LDRH      r6,[r4,#0]
        ADD       r4,r8,r12
  3     ADD       r12,r12,#1
        ADD       r4,r1,r4,LSL #1
        CMP       r12,r7
        LDRH      r4,[r4,#2]
        ADD      r9,r7,r4
        ADD      r8,r2,r4,LSL #1
        ADD      r4,r4,#1
 3      ADD      r9,r1,r9,LSL #1



        MOV      r4,#1
        ADD      r7,r0,r6,LSL #1
        ADD      r4,r4,lr,ASR #15
        ADD      r6,r6,#1
        ASR      r4,r4,#1
        STRH     r4,[r7,#0]
|L1.200|
        CMP      r6,r3
        BLT      |L1.24|
        POP      {r4-r9,pc}
        ENDP
```

```
        SMLABB   r5,r6,r4,r5
        BLT      |L1.116|
|L1.152|
        ADD      r12,r7,#3
        CMP      r12,#7
        BCC      |L1.180|
 2      VADD.I32 d0,d0,d1
        VPADD.I32 d0,d0,d0
        VMOV.32  r12,d0[0]
        ADD      r5,r5,r12
|L1.180|
        ADD      r4,r0,r9,LSL #1
        ADD      r9,r9,#1
        ADD      r12,lr,r5,ASR #15
        CMP      r9,r3
        ASR      r12,r12,#1
        STRH     r12,[r4,#0]
        BLT      |L1.32|
|L1.208|
        POP      {r4-r11,pc}
        ENDP
```

Code shown in [1] is the main loop. Here we see that the complex calculation for memory address has been better optimized by the vectorizing compiler. Our simple re-write using intrinsics is not easily optimized.

The effect of this is that the vectorizing compiler loop is **five** instructions long, where the hand coded loop is **ten**. As the instructions in this loop will be run the most, you can expect that the vectorizing compiler version will run significantly faster.

Code shown in [3] is the cleanup code for the last results in the array. This code is effectively the same, except that the code has been written by the compiler in the automatic case, and written by a human in the intrinsics case. The algorithmic changes will require a peer review.

Code shown in [2] is the reduction code. Here you can see a typical example of user error. The compiler knows which NEON instructions are available and picks the best ones for the task. In the intrinsics case we have chosen the `vgetq_lane_s32` instruction to get each lane and performed the addition in the scalar core. Although correct algorithmically, we would have been better off using vector adds and retrieving a single scalar value.

Code shown in [4] is code missing completely from the intrinsics version. The code written to use intrinsic functions has totally missed the corner case where the array is empty.

Adding inside knowledge

If we know that n_coefs is always divisible by four we can drop the cleanup code from the intrinsics version, but how can we tell the compiler in the vectorizing case?

When we specify the loop count, we should show the compiler by modifying the values used

```
for (n = 0; n < n_out; n++)
```

becomes

```
for (n = 0; n < ((n_out/4)*4); n++)
```

or with some modification in the caller

```
for (n = 0; n < n_out_by_4*4; n++)
```

We recommend modification in the caller because it becomes obvious at the function interface that this source change has been made, and this will aid correct reuse.

With this modification the function code now looks like this:

```
void fir(short * y,
        const short *x, const short *h,
        int n_out, int n_coefs_by_4)
{
        int n;
        for (n = 0; n < n_out; n++)
        {
                int k, sum = 0;
                for(k = 0; k < n_coefs_by_4*4; k++)
                {
                        sum += h[k] * x[n - n_coefs_by_4*4 + 1 + k];
                }
                y[n] = ((sum>>15) + 1) >> 1;
        }
}
```

Note that the change is completely C portable and readable

Side by side output comparison where n_coefs is divisible by 4

NEON Intrinsics

```
||fir|| PROC
        PUSH     {r4-r9}
        CMP      r3,#0
        MOV      r6,#0
        LDR      r7,[sp,#0x18]
        BLE      |L1.144|
        MOV      r9,#1
        ASR      r12,r7,#31
        ADD      r5,r7,r12,LSR #30
|L1.32|
        VMOV.I8  q0,#0
        MOV      r12,#0
        SUB      r4,r6,r7
        B        |L1.80|
|L1.48|
        ADD      r8,r2,r12,LSL #3
  1     VLD1.16  {d2},[r8]
        ADD      r8,r4,r12,LSL #2
        ADD      r12,r12,#1
        ADD      r8,r1,r8,LSL #1
        ADD      r8,r8,#2
        VLD1.16  {d3},[r8]
        VMLAL.S16 q0,d2,d3
|L1.80|
        CMP      r12,r5,ASR #2
        BLT      |L1.48|
        VMOV.32  r12,d0[0]
  2     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
        ADD      r4,r0,r6,LSL #1
        ADD      r6,r6,#1
        ADD      r12,r9,r12,ASR #15
        CMP      r6,r3
        ASR      r12,r12,#1
        STRH     r12,[r4,#0]
        BLT      |L1.32|
|L1.144|
        POP      {r4-r9}
        BX       lr
        ENDP
```

38 Instructions, 10 in internal loop –
Missing corner case

NEON vectorizing compiler

```
||fir|| PROC
        PUSH     {r4-r11}
        CMP      r3,#0
        MOV      r6,#0
        LDR      r9,[sp,#0x20]
        BLE      |L1.148|
        MOV      r11,#1
        RSB      r10,r9,#0
        LSL      r8,r9,#2
|L1.32|
        MOV      r7,#0
        CMP      r8,#0
        BLE      |L1.120|
        VMOV.I8  q0,#0
        ADD      r12,r6,r10,LSL #2
        MOV      r4,r2
        ADD      r12,r1,r12,LSL #1
        ADD      r5,r12,#2
        MOVS     r12,r9
        BEQ      |L1.92|
|L1.72|
        VLD1.16  {d3},[r4]!
  1     SUBS     r12,r12,#1
        VLD1.16  {d2},[r5]!
        VMLAL.S16 q0,d3,d2
        BNE      |L1.72|
|L1.92|
        ADD      r12,r8,#3
        CMP      r12,#7
        BCC      |L1.120|
        VADD.I32 d0,d0,d1
  2     VPADD.I32 d0,d0,d0
        VMOV.32  r12,d0[0]
        ADD      r7,r7,r12
|L1.120|
        ADD      r4,r0,r6,LSL #1
        ADD      r6,r6,#1
        ADD      r12,r11,r7,ASR #15
        CMP      r6,r3
        ASR      r12,r12,#1
        STRH     r12,[r4,#0]
        BLT      |L1.32|
|L1.148|
        POP      {r4-r11}
        BX       lr
        ENDP
```

39 Instructions, 5 in internal loop –
Corner case covered

The compiler is generating code to catch corner cases. If we know a certain variable is a constant we could propagate it using #define rather than passing it by value:

```
#define N_COEFS 16
void fir(short * y,
        const short *x, const short *h,
        int n_out /*, int n_coefs */)
{
    int n;
    for (n = 0; n < n_out; n++){
        int k, sum = 0;
        for(k = 0; k < N_COEFS; k++) {
            sum += h[k] * x[n - N_COEFS + 1 + k];
        }
        y[n] = ((sum>>15) + 1) >> 1;
    }
}
```

NEON Intrinsics

```
||fir|| PROC
        PUSH     {r4-r6}
        CMP      r3,#0
        MOV      r4,#0
        BLE      |L1.124|
        MOV      r5,#1
|L1.20|
        VMOV.I8  q0,#0
        MOV      r12,#0
|L1.28|
  1     ADD      r6,r2,r12,LSL #3
        VLD1.16  {d3},[r6]
        ADD      r6,r4,r12,LSL #2
        ADD      r12,r12,#1
        ADD      r6,r1,r6,LSL #1
        CMP      r12,#3
        SUB      r6,r6,#0x16
        VLD1.16  {d2},[r6]
        VMLAL.S16 q0,d3,d2
        BLT      |L1.28|
        VMOV.32  r12,d0[0]
  2     VMOV.32  r6,d0[1]
        ADD      r12,r12,r6
        VMOV.32  r6,d1[0]
        ADD      r12,r12,r6
        VMOV.32  r6,d1[1]
        ADD      r12,r12,r6
        ADD      r6,r0,r4,LSL #1
        ADD      r4,r4,#1
        ADD      r12,r5,r12,ASR #15
        CMP      r4,r3
        ASR      r12,r12,#1
        STRH     r12,[r6,#0]
        BLT      |L1.20|
|L1.124|
        POP      {r4-r6}
        BX       lr
```

NEON vectorizing compiler

```
||fir|| PROC
        PUSH     {r4-r8}
        CMP      r3,#0
        MOV      r6,#0
        BLE      |L1.108|
        MOV      r8,#1
|L1.20|
        MOV      r7,#0
        VMOV.I8  q0,#0
        ADD      r4,r1,r6,LSL #1
        MOV      r12,r2
        SUB      r4,r4,#0x1e
        MOV      r5,#4
|L1.44|
        VLD1.16  {d3},[r12]!
  1     SUBS     r5,r5,#1
        VLD1.16  {d2},[r4]!
        VMLAL.S16 q0,d3,d2
        BNE      |L1.44|
        ADD      r4,r0,r6,LSL #1
        ADD      r6,r6,#1
        CMP      r6,r3
        VADD.I32 d0,d0,d1
  2     VPADD.I32 d0,d0,d0
        VMOV.32  r12,d0[0]
        ADD      r12,r12,r7
        ADD      r12,r8,r12,ASR #15
        ASR      r12,r12,#1
        STRH     r12,[r4,#0]
        BLT      |L1.20|
|L1.108|
        POP      {r4-r8}
        BX       lr
        ENDP
```

Conclusion

Using the ARM vectorizing compiler enables fast porting of algorithms from one architecture to another, or the sharing of the same code across many processors, as C / C++ code can be used unchanged (or with minimum but standards compliant changes).

The use of the vectorizing compiler enables engineers without SIMD experience to generate fast vector code without having to spend a lot of time having to learn a new instruction set or understand how it interacts with the rest of the system.

Using the NEON extensions reduces system cost compared with a CPU with additional hardware accelerator in terms of power and area as well as ongoing royalty costs

Using the vectorizing compiler reduces development cost in terms of design and debugging time as well as the removing the cost of maintaining multiple tools from different vendors.