Accelerating Software Radio on ARM: Adding NEON Support to VOLK

Nathan West\textsuperscript{1,2} and Douglas Geiger\textsuperscript{1}

\textsuperscript{1}US Naval Research Laboratory
\textsuperscript{2}Oklahoma State University

Abstract—We extend GNU Radio’s VOLK library to use SIMD instructions by creating optimized signal processing routines in NEON with both compiler intrinsics functions and hand-tuned assembly where appropriate. We use source analysis and disassembly to determine when hand-tuned assembly is required for optimization. Finally, profiling results using Cortex-A8 and Cortex-A9 processors are presented that demonstrate our performance improvements.

I. INTRODUCTION

The increase in computing performance in ARM SoCs (System on Chip) along with the addition of many low power coprocessors make ARM SoCs potential size, weight, and power efficient SDR (Software Defined Radio) platforms. The current lowest hanging fruit is the NEON Media Processing Engine which has come standard with the “Application profile” ARM processors since the Cortex-A8. NEON is a SIMD (Single Instruction Multiple Data) coprocessor which makes the GNU Radio VOLK (Vector Optimized Library of Kernels) a good fit for implementation.

GNU Radio’s VOLK allows block level designs to leverage SIMD instructions without being concerned by machine portability. VOLK has a collection of kernels where each kernel is a unique operation, for example a dot product of 32-bit floats. Each kernel has at least one implementation of the operation, called proto-kernels. At a bare minimum each kernel has a generic proto-kernel written in standard C that can be compiled on any machine with a C compiler. To accelerate applications each kernel typically has several proto-kernels targeting a specific SIMD instruction set.

VOLK has proven useful in accelerating GNU Radio applications to enable more real time applications on x86 and AMD64 based processors; however, there is limited support for embedded devices. VOLK has some proto-kernels such as dot products and complex multiplies in a meta-language called ORC (Oil Runtime Compiler) which enables just-in-time compilation on machines supported by liboil; however, the ORC language has shortcomings that make many algorithms impractical and slower than proto-kernels written using SIMD intrinsics. In this paper we introduce support for and add several NEON proto-kernels for VOLK.

Section II introduces features of ARM processors and NEON that will be useful for optimizing SDR performance, and section III discusses adding NEON support to the VOLK framework as well as the methodology for measuring performance. Finally, section IV will show the performance gains the NEON provides.

II. ARM FEATURES FOR SDR

Details of the ARM architecture are outside the scope of this paper and there are many good references for such discussions \cite{2}, \cite{3}. However, it is worth pointing out features that we will take advantage of. In this paper we are dealing primarily with the ARM NEON Media Processing Engine which has been in ARM ‘Application’ profile SoCs since the Cortex-A8. The NEON coprocessor shares registers and several instructions with the VFP (Vector Floating Point) coprocessor. VFP has 32 scalar floating point registers, which map to the lower 16 NEON double-word registers. Figure II shows how the VFP scalar (s#) registers map to NEON double-word (d#) registers and quad-word (q#) registers. NEON has an additional 16 NEON double-word registers that VFP cannot access. That makes a total of 32 double-word registers in the NEON coprocessor, which map to 16 quad-word registers that can hold 128 bits each. Floating point and integer data in 8, 16, 32, or 64 bit formats can be loaded and operated on with NEON instructions.

![Fig. 1. Two VFP s# registers map in to a NEON d# register. Two d# registers map in to a single q# register. There are a total of 32 s-registers that map to the lower 16 d-registers. There are a total of 32 d-registers (d0-d31) and 16 q-registers (q0-q31). The number of items in each register depends on the item size.](image)

NEON loads and stores do not require strict alignment, although at the assembly level there are optional arguments to hint at a known alignment for potential performance gains. NEON also has flexible data load and store operations that allow interleaved loads and stores with a stride of one to four. Interleaved loads and stores are convenient for complex data which uses loads with a stride of two, putting the real parts in to one register and the imaginary parts in to a neighboring register. NEON and VFP have fused multiply addition and
subtraction, which can be used to efficiently pipeline some operations.

Finally, it is important to note that NEON is not fully IEEE 754 compliant, for example, because denormalized numbers can be rounded to zero, but this is generally not an issue in SDR.

III. ARM AND NEON IN VOLK

VOLK generates an abstraction of processor capabilities based on the available architectures. Linux on an ARM processor will be built for either softfp or hardfp ABI (as a compiler option), which VOLK uses when defining machine capabilities. Note that softfp vs hardfp is simply a calling convention for transferring floating point values to functions, and all floating point arithmetic is either done in a VFP or NEON coprocessor, if present, regardless of soft or hard floating point convention. As previously mentioned it is also possible to compile GNU Radio and VOLK with support for ORC, which would give an ORC machine. To begin this work we defined a NEON machine, which is again beyond the scope of this document but is merely an XML description of compiler flags; more detail is found in [1].

Rather than walk through every proto-kernel design and performance the next several sections will walk through example implementations that demonstrate how features of NEON are used by grouping kernels in to groups based on complexity.

A. Methodology

Although some proto-kernels are written with in-line assembly, the vast majority use compiler intrinsics. Compiler intrinsics are C functions that are closely related to compiler instructions which provides a good starting point to guarantee that SIMD instructions are generated while leaving register allocation and instruction scheduling to the compiler. The ARM compiler technical reference provides prototypes for intrinsic functions which have also been implemented by GCC and Clang. All of the code compiled for this paper uses an cross-compiling toolchain built with Open Embedded for targeting a soft-float system root. The full list of compiler flags would be unwieldy; however the following is a list of notable flags we use with GCC 4.8.2

- `-O3` (highest optimization level)
- `-mfloat-abi=hard` (hardfloat)
- `-mfpu=neon` (use NEON for floating point)
- `-mno-debug` (strip debugging symbols)
- `-funsafe-math-optimizations` (allow NEON)

We use an iterative development where the first proto-kernel is written using NEON intrinsics in a way that parallels the flow of the generic proto-kernel. Following the initial proto-kernel we occasionally rearrange the algorithm to optimize the NEON pipeline, and if further performance is required we use `objdump` for disassembly, which provides a starting point for writing hand-tuned implementations. Proto-kernel development is done using a Xilinx zc702 development board, which uses an ARM Cortex-A9 MPCORE with a maximum clock of 666MHz. In the final section we introduce a different set of boards and processors to compare results.

B. Embarrassingly Simple Kernels

By embarrassingly simple kernels we refer to kernels that have such simple implementations that there is little optimization outside of loop unrolling. Our first versions of every kernel are written using compiler intrinsics. In some cases using the intrinsics based proto-kernel is sufficient because the operation is not currently a high priority in application code or because the proto-kernel is already fast enough. We show that for this class of kernels GCC does emit NEON instructions where appropriate, but the execution time for large loops can still be decreased with compiler hints and even further with hand-tuned assembly. For analysis of our optimizations we will use the VOLK 32f_x2_add_32f kernel as an example of this class of simple operations. Several similar kernels exist, for example 32f_x2_multiply_32f, 32f_invsqrt_32f, 32f_x2_interleave_32fc are all single instructions outside of load and store operations. The potentially strange naming convention comes from VOLK where the pattern is `input description`-`kernel name`-`output description` [1]. As an example 32f_x2_add_32f has two inputs that are 32-bit floats, the operation is an add, and the output is a 32-bit float.

The existing VOLK generic proto-kernel iterates through the entire buffer one element at a time. The inner loop is shown in Listing 1. GCC recognizes that this loop can be vectorized and emits code that operates on the buffers using NEON `vadd` instructions, shown in Listing 2. This kernel also has an ORC proto-kernel with the same execution time as the generic proto-kernel, which is not surprising for a simple operation since both ORC and our compiler are emitting the same instructions.

```
for (number = 0; number < num_points; number++){
    *cPtr++ = (*aPtr++) + (*bPtr++);
}
```

Listing 1. C implementation of 32f_x2_add_32f_generic kernel’s inner loop.

```
.looop1:
  vld1.d32 [d18–d19], [r5]! @ load vecA
  add ip, ip, #1 @ number += 1
  cmp r8, ip @ number < num_points
  bhi loop1 @ repeat num_points/4
  vld1.d32 [d16–d17], [r6]! @ load vecB
  vadd.f32 q8, q9, q8 @ vector addition
  vst1.d32 [d16–d17], [r4]! @ store to memory

Listing 2. Disassembly of 32f_x2_add_32f_generic kernel’s inner loop.

Using NEON intrinsics to implement this loop results in a very subtle slow down compared to the generic kernel. This is likely caused by intrinsics removing some flexibility in GCC’s instruction scheduling. Since this operation is memory-latency limited we can use prefetching to hint the CPU to load future values in to the cache. GCC provides a built-in prefetch; the usage with this kernel is shown in Listing 3. This prefetch on our test platform results in approximately 8% less time for a 200k item long buffer; however, going to ASM allows better use of ARM’s prefetching mechanism.
```c
for(number=0; number < quarterPoints; number++){
    // Load input to NEON registers
    aVal = vld1q_f32(aPtr);
    bVal = vld1q_f32(bPtr);
    __builtin_prefetch(aPtr+4);
    __builtin_prefetch(bPtr+4);
    // vector add in NEON
    cVal = vaddq_f32(aVal, bVal);
    // Store the results back into the C container
    vsrq_f32(cPtr, cVal);
    // four floats per buffer were used
    aPtr += 4;
    bPtr += 4;
    cPtr += 4;
}
```

Listing 3. C implementation of 32f_x2_add_32f_neon kernel's inner loop with NEON intrinsics.

ARM assembler allows an offset argument to the pld instruction. Since NEON quad-word registers are 128-bits wide, the natural argument here is 128 to prefetch the next four values. The hand-optimized ASM proto-kernel, displayed in Listing 4, also loads data before loop execution and runs with 1-off the loop, which requires a post-addition after the loop is finished. The result is 17% faster execution than the GCC vectorized version. Since this kernel represents the type of operation a modern compiler can automatically vectorize we should expect no less than 10% run-time improvement in more complex kernels. We also conclude that for strictly memory-limited kernels of this complexity the preload is a primary mechanism to improve execution time in the profiler. The effect of preloading has not been investigated at the application level, which we leave as future work.

```asm
@ Optimizing for pipeline
vldl.32 {d0-d1}, [aVector:128] @ aVal
vldl.32 {d2-d3}, [bVector:128] @ bVal
subs number, number, #1

@ loop:
pld [aVector, #128] @ pre-load hint
pld [bVector, #128] @ pre-load hint
vadd.f32 cVal, bVal, aVal
vldl.32 {d0-d1}, [aVector:128] @ aVal
vldl.32 {d2-d3}, [bVector:128] @ bVal
vstl.32 {d4-d5}, [cVector:128] @ cVal
@ execute loop quarter_points times
subs number, number, #1
bne .loop1 @ first loop

@ One more time
vadd.f32 cVal, bVal, aVal
vstl.32 {d4-d5}, [cVector:128] @ cVal
```

Listing 4. NEON implementation of 32f_x2_add_32f_neonasm kernel's inner loop with ASM.

### C. Moderately Simple Kernels

Moderately simple kernels refer to operations that are not single-instruction, but occur so frequently that modern compilers are still capable of optimizing them. Using the 32fc_x2_multiply_32fc, a complex multiply, as an example we will demonstrate noticeable improvements in execution time with NEON intrinsics kernels and further improvements by using hand-tuned ASM.

The 32fc_x2_multiply_32fc generic proto-kernel, shown in Listing 5 uses a complex data type to multiply one complex input by another and store the result. The disassembly, shown in 5, is an efficient scalar routine that can easily be vectorized with intrinsics. GCC has a highly optimized complex multiply included in its runtime library that can be called by branching to __mulsc3. Listing 6 shows this runtime library call is used by GCC rather than vectorizing the loop. As a tangent to the primary discussion this is a very good example of the soft-float ABI being used since the four operands to __mulsc3 are passed via ARM’s general purpose integer registers rather than in VFP registers. The first four instructions in __mulsc3 will be to move required values in to VFP registers; however, if we were using a hardfloat ABI these parameters could be passed directly in VFP registers.

```c
for(number = 0; number < num_points; number++){
   *cPtr++ = (*aPtr++) * (*bPtr++);
}
```

Listing 5. C implementation of 32fc_x2_multiply_32fc_generic kernel's inner loop.

```asm
.mainloop:
1ld r3, [r6], #8 @ a1
mov r0, r9
1ld r1, [r5], #8 @ b1
add r7, r7, #1
1ld r12, [r6, #4] @ ar
1ld r2, [r5, #4] @ ai
str r12, [r13]
b @ GCC built-in scalar
complex mult
1ld r2, [r13, #8]
1ld r3, [r13, #12]
cmp r7, r8
str r2, [r4], #8
str r3, [r4, #4]
```

Listing 6. Disassembly of 32fc_x2_multiply_32fc_generic kernel's inner loop.

A vectorized version of 32fc_x2_multiply_32fc using NEON intrinsics is shown in Listing 7. This NEON implementation uses a series of temporary variables to store the four different products rather than two temporary variables followed by NEON’s fused multiply add (or subtract). Figure III-C shows the cycle timing for both multiy approaches under optimal conditions. According to the cycle timing tables (and using the Cortex A9’s eight-stage pipeline) the fused multiply-add variant has a one-cycle penalty. This agrees with our tests which show routines using fused multiy-add instructions are either very close to slightly slower than equivalent routines without fused multiply-adds.

```asm
for(number = 0; number < quarterPoints; ++number) {
   aVal = vldq_f32((float*)a_ptr); // a0r|alr|ar2r
   bVal = vldq_f32((float*)b_ptr); // b0r|blr|br2r
   __builtin_prefetch(a_ptr+4); // a0r|alr|ar2r|b0r|blr|br2r
   __builtin_prefetch(b_ptr+4); // a0r|alr|ar2r|b0r|blr|br2r
   // multiply the real*real and imag*imag to get real result
   // a0r|b0r|alr|blr|ar2r|br2r
   tmp_real.val[0] = vmulq_f32(aVal, bVal[0]);
   tmp_real.val[1] = vmulq_f32(aVal, bVal[1]);
   // a0i|b0i|a1i|b1i|a2i|b2i|a3i|b3i
   // multiply the real*imag and imag*real
   // a0r|b0r|alr|blr|ar2r|br2r
   tmp_imag.val[0] = vmlaq_f32(aVal, bVal[0]);
   tmp_imag.val[1] = vmlaq_f32(aVal, bVal[1]);
   // store the result
   // a0r|b0r|alr|blr|ar2r|br2r
   vstrq_f32(tmp_real.val, cVal);
   vstrq_f32(tmp_imag.val, cVal);
}
```

Listing 7. NEON implementation of 32fc_x2_multiply_32fc kernel's inner loop.
Fig. 2. Cycle timing for complex floating point multiplication in NEON using only multiplies and adds compared to multiplies with fused multiply-adds or subtractions. The src or dst blocks indicate when the source or destination operands must be ready. Boxes labeled re and wb respectively indicate the result is ready for use and written back to the register file. The D indicates a delay slot because a source is not ready yet.

Listing 7. C implementation of 32fc_x2_multiply_32fc_neon kernel's inner loop with NEON intrinsics.

Although NEON has instructions to operate on four floats at a time the Cortex-A9 requires an extra execute cycle for with quad word floating point operations. The intrinsic-based implementation of 32fc_x2_multiply_32fc_neon shown in Listing 7 is 35% faster than the generic version from Listing 5. To improve even further we disassemble the compiled NEON intrinsic to look for opportunities to hand-tune the generated code. The inner loop of this disassembled code is shown in Listing 8.

Listing 8. Disassembly of 32fc_x2_multiply_32fc_neon kernel's inner loop.

From Listing 8 it is immediately obvious that we can remove the vorr instructions that are moving data around within NEON registers. The final hand-tuned code is shown in Listing 9, which is 8% faster than the intrinsics version (42% faster than the generic version).


GCC 4.8 typically vectorizes simple operations that are known to be easily vectorized, but the complex multiply case demonstrates the need for intrinsics based kernels. The disassembly shows that even when GCC can emit vectorized code with the help of intrinsics there may still be hand-tuning to fix sub-optimal instructions.

D. Difficult Kernels

Difficult kernels represent a challenge for a compiler to properly vectorize which means a NEON intrinsics based kernel written in C provides satisfactory performance gains without going to hand-tuned ASM. If an application becomes highly dependent on one of these kernels it may be necessary to use hand-tuned assembly, but the performance gains seen with intrinsics based proto-kernels are already very large. As an example of optimizing difficult kernels we analyze the performance of VOLK’s 32f_x3_sum_of_poly_32f kernel.

The generic proto-kernel is shown in Listing 10. This is an uncommon algorithm that takes a short (four-point) dot product between a fixed four-point vector and powers of elements in an input array.

NEON intrinsics make this an easy optimization target once we recognize the pattern of multiplying powers of the input vector by a fixed vector and summing the output. Since we want an accumulation the inner loop can use multiple accumulators which get reduced to a single scalar after the loop. The NEON intrinsic’s based proto-kernel, shown in Listing 11, uses a long pipeline of instructions and keeps data dependence inside the loop to a minimum. This intrinsics based proto-kernel executes 76% faster than the generic proto-kernel. For this large of a gain with intrinsics proto-kernels it is
not necessary to use assembly proto-kernels unless this kernel becomes a bottleneck for an application, at which point there is likely opportunity for more optimization.

```c
for (; i < num_bytes >> 2; ++i) {
  fst = src0[i];
  fst = MAX(fst, *cutoff);
  sq = fst * fst;
  thrd = fst * sq;
  frth = sq * sq;
  result += (center_point_array[0] * fst +
              center_point_array[1] * sq +
              center_point_array[2] * thrd +
              center_point_array[3] * frth);
}
```

Listing 10. C implementation of 32f_x3_sum_of_poly_32f_generic kernel's inner loop.

```c
for(i=0; i < num_points/4; ++i) {
  // load x
  x_to_l = vldlq_f32( src0 );
  // Get a vector of max(src0, cutoff)
  x_to_l = vmaxq_f32(x_to_l, cutoff_vector ); // x\times1
  x_to_2 = vmulq_f32(x_to_l, x_to_l); // x\times2
  x_to_3 = vmulq_f32(x_to_2, x_to_l); // x\times3
  x_to_4 = vmulq_f32(x_to_3, x_to_l); // x\times4
  x_to_l = vmulq_f32(x_to_4, cpa_0);
  x_to_2 = vmulq_f32(x_to_4, cpa_1);
  x_to_3 = vmulq_f32(x_to_3, cpa_2);
  x_to_4 = vmulq_f32(x_to_4, cpa_3);
  accumulator1_vec = vaddq_f32(accumulator1_vec, x_to_l);
  accumulator2_vec = vaddq_f32(accumulator2_vec, x_to_2);
  accumulator3_vec = vaddq_f32(accumulator3_vec, x_to_3);
  accumulator4_vec = vaddq_f32(accumulator4_vec, x_to_4);
  src0 += 4;
}
```

Listing 11. C implementation of 32f_x3_sum_of_poly_32f_neon kernel's inner loop.

IV. RESULTS

Relative speedups compared to soft-float generic implementations for different kernels are shown in Figures 3, 4, 5, and 6. The results show run-time comparisons for hard-float (indicated with a _hf suffix) and soft-float (indicated with a _sf suffix) builds. Each graph shows run-time for input vectors with 204603 items repeated 5000 times normalized by the run-time for the soft-float generic proto-kernel. On our test system all NEON kernels show a faster run-time than generic versions, and often ORC proto-kernels are similar to our NEON versions. In general soft-float vs. hard-float has no performance difference within compiler versions, which is expected. The exceptions are 32f_sqrt_32f, 32fc_x2_multiply_32fc, and 32fc_x2_multiply_conjugate_32fc. We have not looked into reasons for this, but this highlights a side effect of VOLK in stabilizing performance gains across compiler flags.

V. CONCLUSION

A. Future Work

Many of the proto-kernels presented are limited by memory access speed and benefit from prefetching data to load vectors in the cache. The effect of prefetching data in GNU Radio applications is not well understood, and may not be necessary because data may already be in the cache. On the other hand prefetching data may result in a high data turnover in the cache resulting in a gain for stand-alone VOLK kernels, but a net loss for an application. One method to understand performance issues relating to cached data in GNU Radio would be to extend the recently released GNU Radio performance counters to include cache misses and profile applications of varying size with different sized caches.

Additionally, it’s clearly seen from profiling results that complex algorithms (such as the sum of poly) benefit from VOLK more than very simple algorithms (such as the adders, interleavers, etc...). It would be beneficial to include common algorithms in to VOLK such as OFDM frame syncs, clock recovery, and turbo coding. There are also other coprocessors such as FPGAs and GPUs available to ARM processors that might be better candidates for such algorithms. SDR applications would benefit from a unified approach to handling the
many coprocessors that are becoming available with run-time decisions of which coprocessors should be used for different tasks.

B. Summary

We added NEON proto-kernels to VOLK for improved SDR performance on ARM platforms. Using NEON intrinsics provides guarantees that inner loops are vectorized for all platforms regardless of the compiler (as long as the compiler supports intrinsics). For critical kernels that are used often or are measured to be performance bottlenecks going to assembly proto-kernels will usually provide an additional speedup, although typically small.

ACKNOWLEDGMENT

Thanks to Philip Balister for various technical support with ARM related issues and maintaining the GNU Radio (and dependant) recipes in Open Embedded.

REFERENCES

Operations with Integer Inputs

![Diagram showing run-time normalized values for different kernels with integer inputs.](image)

Fig. 6. VOLK profile results for kernels with integer input buffers. Bars are run-time normalized to generic soft-float (lower is better).
Following are code listings of proto kernels we developed in C, primarily using NEON intrinsics.

Listing 12. A NEON implementation of volk_arm_32fc_x2_square_dist_32f.

```c
#define LV_HAVE_NEON

static inline void
volk_arm_32fc_x2_square_dist_32f_neon(float *target, float *src0, int points, unsigned int num_points)
{
    const unsigned int quarter_points = num_points / 4;
    unsigned int number;

    float32x4_t a_vec[2], b_vec[2];
    float32x4_t tmp1, tmp2, dist_sq;
    float32x4_t tmp3, tmp4, dist_sq;

    a_vec = vld2q_f32(a_vec, num_points);
    b_vec = vld2q_f32(b_vec, num_points);
    tmp1 = vmlaq_f32(a_vec, a_vec);
    tmp2 = vmlaq_f32(b_vec, b_vec);
    for (number = 0; number < quarter_points; ++number)
    {
        b_vec = vld2q_f32((float*)points);
        a_vec = vld2q_f32((float*)points);
        tmp1 = vmlaq_f32(b_vec, a_vec);
        tmp2 = vmlaq_f32(a_vec, b_vec);
        for (number = quarter_points * 4; number < num_points; ++number)
        {
            dist_sq = vaddq_f32(tmp1, tmp2);
            vst1q_f32(target, dist_sq);
        }
    }
}
#endif /* LV_HAVE_NEON */
```

Listing 13. A NEON implementation of volk_arm_32fc_magnitude_32f.

```c
#define LV_HAVE_NEON

static inline void
volk_arm_32fc_magnitude_32f_neon(float *magnitudeVector, const float *complexVector, unsigned int num_points)
{
    unsigned int number;
    unsigned int num_points;
    const float *complexVectorPtr = (float*)complexVector;
    float *magnitudeVectorPtr = magnitudeVector;
    for (number = 0; number < num_points; ++number)
    {
        complex_vec = vld2q_f32(complexVectorPtr);
        vec = vmlaq_f32(complex_vec, vec, 0);
        lv_32fc_t magnitude = magnitude(vec);
    }
}
#endif /* LV_HAVE_NEON */
```
for (number = 0; number < quarter_points; number++)
complex_vec = vld2q_f32(complexVectorPtr);
real_abs = vabsq_f32(complex_vec.val[0]);
imag_abs = vabsq_f32(complex_vec.val[1]);
min_vec = vminq_f32(real_abs, imag_abs);
max_vec = vmaq_f32(real_abs, imag_abs);

// effective branch to choose coefficient pair
comp0 = vcgtq_f32(min_vec, vmulq_n_f32(max_vec, threshold));
comp1 = vcleqq_f32(min_vec, vmulq_n_f32(max_vec, threshold));

// and 0s or 1s with coefficients from previous effective branch
a_vec = (float32x4_t)vaddq_s32(vandq_s32((int32x4_t)comp0, (int32x4_t)a_high),
(int32x4_t)comp1, (int32x4_t)a_low));
b_vec = (float32x4_t)vaddq_s32(vandq_s32((int32x4_t)comp0, (int32x4_t)b_high),
(int32x4_t)comp1, (int32x4_t)b_low));

// coefficients chosen, do the weighted sum
min_vec = vmulq_f32(min_vec, b_vec);
max_vec = vmaq_f32(max_vec, a_vec);
magnitude_vec = vaddq_f32(min_vec, max_vec);
vs1q_f32(magnitudeVectorPtr, magnitude_vec);
complexVectorPtr += 8;
magnitudeVectorPtr += 4;
}


#endif /* LV_HAVE_NEON */

#define LV_HAVE_NEON
#include <arm_neon.h>

/*
\brief Adds the two input vectors and store their results in the third vector
\param cVector The vector where the results will be stored
\param aVector One of the vectors to be added
\param bVector One of the vectors to be added
\param num_points The number of values in aVector and bVector to be added together and stored into cVector
*/
static inline void volk_arm_32fc_x2_add_32f_u_neon(float* cVector, const float* aVector, const float* bVector, unsigned int num_points) {
unsigned int number = 0;
const unsigned int quarterPoints = num_points / 4;
float32x4_t aVal = vlq_f32(aPtr);
bVal = vlq_f32(bPtr);
__builtin_prefetch(aPtr+4);
__builtin_prefetch(bPtr+4);

// vector add
cVal = vaddq_f32(aVal, bVal);
// Store the results back into the C container
vs1q_f32(cPtr, cVal);

aPtr += 4; // q uses quadwords, 4 floats per vadd
bPtr += 4;
cPtr += 4;
}

number = quarterPoints * 4; // should be = num_points
for (number < num_points; number++){
*pCptr++ = (*aPtr++) + (*bPtr++);
}

}
#endif /* LV_HAVE_NEON */

Listing 15. A NEON implementation of volk_arm_32f_x2_add_32f.
Listing 16. A NEON implementation of volk_arm_32fc_x2_add_32fc.

```c
#ifdef LV_HAVE_NEON
#include <arm_neon.h>
/
\brief Takes the conjugate of a complex vector.
\param cVector The vector where the results will be stored.
\param aVector Vector to be conjugated.
\param num_points The number of complex values in aVector to be conjugated and stored into cVector.
*/
static inline void
volk_arm_32fc_conjugate_32fc_a_neon(lv_32fc_t* cVector, const lv_32fc_t* aVector, unsigned int num_points)
{
    const unsigned int quarterPoints = num_points / 4;
    float32x4x2_t x;
    lv_32fc_t* c = cVector;
    const lv_32fc_t* a = aVector;
    float conj[4] = {-0.f, -0.f, -0.f, -0.f};
    //uint32x4_t conjugator;
    //conjugator = vld1q_u32( (uint32_t *)conj );
    for (number=0; number < quarterPoints; number++) {
        temp0 = ( (short) ( float * ) a) + number;
        x = vld2q_f32( (float *) a);
        // xor the imaginary lane
        x.val[1] = vnegq_f321( x.val[1]);
        vst2q_f32( (float *) c, x); // Store the results back into the C container
        a => 4;
        c => 4;
    }
}
#endif /* LV_HAVE_NEON */
```

Listing 17. A NEON implementation of volk_arm_32fc_conjugate_32fc.

```c
#ifdef LV_HAVE_NEON
#include <arm_neon.h>
static inline void
volk_arm_32fc_conjugate_32fc_a_neon(lv_32fc_t* cVector, const lv_32fc_t* aVector, unsigned int num_points)
{
    const unsigned int quarterPoints = num_points / 4;
    unsigned int i;
    int16x8_t src0_vec, src1_vec, src2_vec, src3_vec;
    int16x8_t diff12, diff34;
    int16x8_t comp0, comp1, comp2, comp3;
    int16x8_t result_vec, result2_vec;
    int16x8_t zeros;
    zeros = veorq_s16( zeros, zeros);
    for (i=0; i < quarterPoints; ++i) {
        src0_vec = vld1q_s16(src0);
        src1_vec = vld1q_s16(src1);
        src2_vec = vld1q_s16(src2);
        src3_vec = vld1q_s16(src3);
        diff12 = vsubq_s16(src0_vec, src1_vec);
        diff34 = vsubq_s16(src2_vec, src3_vec);
        comp0 = (int16x8_t)vcgeq_s16(diff12, zeros);
        comp1 = (int16x8_t)vcltuq_s16(diff12, zeros);
        comp2 = (int16x8_t)vcgeq_s16(diff34, zeros);
        comp3 = (int16x8_t)vcltuq_s16(diff34, zeros);
        result1_vec = vaddq_s16(src0_vec, comp0);
        result1_vec = vaddq_s16(src1_vec, comp1);
        result1_vec = vaddq_s16(src2_vec, comp2);
        result1_vec = vaddq_s16(src3_vec, comp3);
        result2_vec = vaddq_s16(result1_vec, result1_vec);
        result1_vec = vaddq_s16(result1_vec, result2_vec);
        result1_vec = vaddq_s16(result1_vec, result2_vec);
    }
}
#endif /* LV_HAVE_NEON */
```

Listing 18. A NEON implementation of volk_arm_16i_x4_quad_max_star_16i.

```c
# ifdef LV_HAVE_NEON
#include <arm_neon.h>
/
\brief Interleaves the I & Q vector data into the complex vector.
\param iBuffer The I buffer data to be interleaved.
\param qBuffer The Q buffer data to be interleaved.
\param complexVector The complex output vector.
\param num_points The number of complex data values to be interleaved.
*/
static inline void
volk_arm_32f_x2_interleave_32fc_neon(lv_32fc_t* complexVector, const float* iBuffer, const float* qBuffer, unsigned int num_points)
{
    unsigned int quarterPoints = num_points / 4;
    unsigned int number;
    float* complexVectorPtr = (float*)complexVector;
    float32x4x2_t complex_vec;
    for (number=0; number < quarterPoints; ++number) {
        complex_vec.val[0] = vld1q_f32(iBuffer);
        complex_vec.val[1] = vld1q_f32(qBuffer);
        vst2q_f32(complexVectorPtr, complex_vec);
    }
}
#endif /* LV_HAVE_NEON */
```
Listing 19. A NEON implementation of volk_arm_32f_x2_interleave_32fc.

```c
Listing 20. A NEON implementation of volk_arm_32fc_x2_interleave_32fc.
```

Listing 21. A NEON implementation of volk_arm_32fc_x2_multpyle_32fc.

```c
Listing 22. A NEON implementation of volk_arm_32fc_x2_multiply_32fc_neon_opttests.
```
```c
lv_32fc_t *a_ptr = (lv_32fc_t*) aVector;
v_32fc_t *b_ptr = (lv_32fc_t*) bVector;
unsigned int quarter_points = num_points / 4;
float32x4x2_t a_val, b_val, c_val;
float32x4x2_t tmp Real, tmp_imag;
unsigned int number = 0;

// TODO: I suspect the compiler is doing a poor job scheduling this. This seems
// highly optimal, but is barely better than
general
for (number = 0; number < quarter_points; ++ number) {
a_val = vld2q_f32((float*)a_ptr); // a0|a1;
|a2|a3 || a0i|a1i|a2i|a3i
b_val = vld2q_f32((float*)b_ptr); // b0|b1;
|b2|b3 || b0i|b1i|b2i|b3i
__builtin_prefetch(a_ptr+4);
__builtin_prefetch(b_ptr+4);

// do the first multiply
tmp_imag.val[1] = vmulq_f32(a_val.val[1],
b_val.val[0]);
tmp_imag.val[0] = vmulq_f32(a_val.val[0],
b_val.val[0]);

// use multiply accumulate/subtract to get result
tmp_imag.val[1] = vmlaq_f32(tmp_imag.val[1],
a_val.val[0], b_val.val[1]);
tmp_imag.val[0] = vmfaq_f32(tmp_imag.val[0],
a_val.val[1], b_val.val[1]);

// store
vs1q_f32((float*)cVector, tmp_imag); // increment pointers
a_ptr += 4; b_ptr += 4;
cVector += 4;
}

// this should be able to operate more or less
simultaneous to neon
for (i=0; i < 32; i++){
aligned_src[i] = (short) src0[((char) permuters[i/8][2*(i%8)])];
}

# endif /* LV_HAVE_NEON */
Listing 22. A NEON implementation of
volk_arm_32fc_x2_multiply_32fc.

```
```c
Listing 23. A NEON implementation of volk_arm_16i_branch_4_state_8.

```
static inline void
volk_arm_32fc_magnitude_squared_32f_u_neon(float *magnitudeVector, const lv_32fc_t* complexVector, unsigned int num_points) {
    unsigned int number = 0;
    const unsigned int quarterPoints = num_points / 4;
    const float* complexVectorPtr = (float*) complexVector;
    float* magnitudeVectorPtr = magnitudeVector;
    float32x4_t cmplx_val;
    float32x4_t result;
    for (; number < quarterPoints; number++) {
        cmplx_val = vld2q_f32(complexVectorPtr);
        complexVectorPtr += 8;
        cmplx_val.val[0] = vmulq_f32(cmplx_val.val[0], cmplx_val.val[0]); // Square the values
        cmplx_val.val[1] = vmulq_f32(cmplx_val.val[1], cmplx_val.val[1]); // Square the values
        result = vaddq_f32(cmplx_val.val[0], cmplx_val.val[1]); // Add the 12 and Q2 values
        vstlq_f32(magnitudeVectorPtr, result); // Store magnitude values
        magnitudeVectorPtr += 4;
    }
    number = quarterPoints * 4;
    for (; number < num_points; number++) {
        float val1Real = *complexVectorPtr++;
        float val1Imag = *complexVectorPtr++;
        *magnitudeVectorPtr++ = (val1Real * val1Real) + (val1Imag * val1Imag);
    }
}

Listing 25. A NEON implementation of
volk_arm_32fc_magnitude_squared_32f_u.neon.

static inline void
volk_arm_32fc_magnitude_squared_32f_a_neon(float *magnitudeVector, const lv_32fc_t* complexVector, unsigned int num_points) {
    unsigned int number = 0;
    const unsigned int quarterPoints = num_points / 4;
    const float* complexVectorPtr = (float*) complexVector;
    float* magnitudeVectorPtr = magnitudeVector;
    float32x4_t cmplx_val;
    float32x4_t result;
    for (; number < quarterPoints; number++) {
        cmplx_val = vld2q_f32(complexVectorPtr);
        complexVectorPtr += 8;
        cmplx_val.val[0] = vmulq_f32(cmplx_val.val[0], cmplx_val.val[0]); // Square the values
        cmplx_val.val[1] = vmulq_f32(cmplx_val.val[1], cmplx_val.val[1]); // Square the values
        result = vaddq_f32(cmplx_val.val[0], cmplx_val.val[1]); // Add the 12 and Q2 values
        vstlq_f32(magnitudeVectorPtr, result); // Store magnitude values
        magnitudeVectorPtr += 4;
    }
    number = quarterPoints * 4;
    for (; number < num_points; number++) {
        float val1Real = *complexVectorPtr++;
        float val1Imag = *complexVectorPtr++;
        *magnitudeVectorPtr++ = (val1Real * val1Real) + (val1Imag * val1Imag);
    }
}

Listing 26. A NEON implementation of
volk_arm_32fc_magnitude_squared_32f_a.neon.
Listing 27. A NEON implementation of volk_arm_32f_inv_sqrt_32f.

```c

static inline void volk_arm_32fc_inv_sqrt_32f(const float32x4_t* cVector, const float32x4_t* aVector, const float32x4_t* bVector, const float32x4_t* tVector, unsigned int num_points)
{
    const unsigned int quarter_points = num_points / 4;
    for (number = 0; number < quarter_points; ++number) {
        cVector += 4;
        bVector += 4;
        tVector += 4;
    }
}

Listing 28. A NEON implementation of volk_arm_32fc_x2_multiply_32fc.

```
Listing 30. A NEON implementation of volk_arm_32fc_x2_multiply_conjugate_32fc.

```c
// brief Selects minimum value from each entry
// between bVector and aVector and store their
// results in the cVector

static inline void volk_arm_32f_x2_min_32f_neon(
    float* cVector, const float* aVector, const float* bVector, unsigned int num_points)
{
    float32x4_t a_vec, b_vec, c_vec;
    for (number = 0; number < num_points; number++)
    {
        a_vec = vldlq_f32(a_Ptr);
        b_vec = vldlq_f32(b_Ptr);
        c_vec = vminq_f32(a_vec, b_vec);
        vstq_f32(cPtr, c_vec);
        aPtr += 4;
        bPtr += 4;
        cPtr += 4;
    }
}
```

Listing 31. A NEON implementation of volk_arm_32f_x2_min_32f.

```c
#define LV_HAVE_NEON

#define LV_HAVE_NEON

// brief Converts the input 8 bit integer data
// into 16 bit integer data

static inline void volk_arm_8i_convert_16i_neon( 
    int16_t* outputVector, const int8_t* inputVector,
    unsigned int num_points)
{
    int16_t* outputVectorPtr = outputVector;
    const int8_t* inputVectorPtr = inputVector;
    unsigned int number;

    const unsigned int eighth_points = num_points / 8;
    float scale_factor = 256;

    int8x8_t input_vec; 
    int16x8_t converted_vec;
    outputVectorPtr += 8;
    outputVectorPtr += 8;

    for (number = eighth_points * 8; number < num_points; number++)
    {
        outputVectorPtr++ = ( (int16_t)( inputVectorPtr++ ) ) * 256;
    }
}
```

Listing 32. A NEON implementation of volk_arm_8i_convert_16i.
// load the cutoff in to a vector
cutoff_vector = vdup_n_f32(*cutoff);
// ... center point array

cpa_qvector = vld1q_f32(center_point_array);

for (i = 0; i < num_points; ++i) {
    // load x (src0)
    x_to_l = vdup_n_f32(*src0);

    // Get a vector of max(src0, cutoff)
    x_to_l = vmax_f32(x_to_l, cutoff_vector);

    // x'
    x_to_2 = vmul_f32(x_to_l, x_to_1);
    x_to_3 = vmul_f32(x_to_2, x_to_1);
    x_to_4 = vmul_f32(x_to_3, x_to_1);

    // zip up doubles to interleave
    x_low = vzip_f32(x_to_1, x_to_2);
    x_high = vzip_f32(x_to_3, x_to_4);

    // float32x4_t vcombine_f32(float32x2_t low, float32x2_t high);
    // VMOV d0,d0
    x_qvector = vcombine_f32(x_low.val[0], x_high.val[0]);

    // now we finally have [x'4 | x'3 | x'2 | x'] !
    c_qvector = vmlaq_f32(c_qvector, x_qvector, cpa_qvector);
}

// there should be better vector reduction techniques
vst1q_f32(res_accumulators, c_qvector);
accumulator = res_accumulators[0] + res_accumulators[1] +
res_accumulators[2] + res_accumulators[3];

// target = accumulator + center_point_array[4] * (float)num_points;
}

Listing 33. A NEON implementation of
volk_arm_32f_x3_sum_of_poly_32f.

#ifdef LV_HAVE_NEON

static inline void
volk_arm_32f_x3_sum_of_poly_32f_neonvert(float* restrict target, float* restrict src0, float* restrict cutoff, unsigned int num_points) {

    int i;
    float zero[4] = {0.0f, 0.0f, 0.0f, 0.0f};

    float32x4_t accumulator1_vec, accumulator2_vec, accumulator3_vec, accumulator4_vec;
    accumulator1_vec = vld1q_f32(zero);
    accumulator2_vec = vld1q_f32(zero);
    accumulator3_vec = vld1q_f32(zero);
    accumulator4_vec = vld1q_f32(zero);

    float32x4_t x_to_l, x_to_2, x_to_3, x_to_4;
    float32x4_t cutoff_vector, cpa_0, cpa_1, cpa_2, cpa_3;

    // load the cutoff in to a vector
cutoff_vector = vdupq_n_f32(*cutoff);
    // ... center point array
    cpa_0 = vdupq_n_f32(center_point_array[0]);
    cpa_1 = vdupq_n_f32(center_point_array[1]);
    cpa_2 = vdupq_n_f32(center_point_array[2]);

    cpa_3 = vdupq_n_f32(center_point_array[3]);

    // nathan is not sure why this is slower *and* wrong compared to neonvertfma
    for (i = 0; i < num_points / 4; ++i) {
        // load x
        x_to_l = vld1q_f32(src0);

        // Get a vector of max(src0, cutoff)
        x_to_l = vmaxq_f32(x_to_l, cutoff_vector);

        // x'
        x_to_2 = vmulq_f32(x_to_l, x_to_1);
        x_to_3 = vmulq_f32(x_to_2, x_to_1);
        x_to_4 = vmulq_f32(x_to_3, x_to_1);

        // zip up doubles to interleave
        x_low = vzipq_f32(x_to_1, x_to_2);
        x_high = vzipq_f32(x_to_3, x_to_4);

        // float32x4_t vcombine_f32(float32x2_t low, float32x2_t high);
        // VMOV d0,d0
        x_qvector = vcombine_f32(x_low.val[0], x_high.val[0]);

        // now we finally have [x'4 | x'3 | x'2 | x'] !
        c_qvector = vmlaq_f32(c_qvector, x_qvector, cpa_qvector);

        // there should be better vector reduction techniques
        vst1q_f32(res_accumulators, c_qvector);
        accumulator = res_accumulators[0] + res_accumulators[1] +
        res_accumulators[2] + res_accumulators[3];

        // target = accumulator + center_point_array[4] * (float)num_points;
    }

    src0 += 4;

    accumulator1_vec = vaddq_f32(accumulator1_vec, accumulator2_vec);
    accumulator3_vec = vaddq_f32(accumulator3_vec, accumulator4_vec);
    accumulator5_vec = vaddq_f32(accumulator1_vec, accumulator3_vec);
    accumulator6_vec = vaddq_f32(accumulator2_vec, accumulator4_vec);

    __VOLK_ATTR_ALIGNED(32) float res_accumulators[4];
    vst1q_f32(res_accumulators, accumulator1_vec);
    accumulator = res_accumulators[0] + res_accumulators[1] +
    res_accumulators[2] + res_accumulators[3];

    float result = 0.0f;
    float fst = 0.0f;
    float sq = 0.0f;
    float thrd = 0.0f;
    float frrth = 0.0f;

    for (i = 4 * num_points / 4; i < num_points; ++i) {
        fst = src0[i];
        result = MAX(fst, *cutoff);
        sq = frrth * sq;
        thrd = frrth * sq;
        // frrth = sq * thrd;

        accumulator += (center_point_array[0] * fst +
            center_point_array[1] * sq +
            center_point_array[2] * thrd +
            center_point_array[3] * frrth);
    }

    // target = accumulator + center_point_array[4] * (float)num_points;
}

Listing 34. A NEON implementation of
volk_arm_32f_x3_sum_of_poly_32f.

#endif /* LV_HAVE_NEON*/

#ifdef LV_HAVE_GENERIC
static inline void
volk_arm_32f_x3_sum_of_poly_32f_generic(float *target, float *src0, float *center_point_array, float *cutoff, unsigned int num_points) {

    const unsigned int num_bytes = num_points*4;

    float result = 0.0;
    float sq = 0.0;
    float thrd = 0.0;
    float fth = 0.0;
    // float fth = 0.0;

    unsigned int i = 0;

    for (; i < num_bytes >> 2; ++i) {
        float rst = src0[i];
        float = MAX(fst, *cutoff);

        float = sq * fst;
        float = sq * thrd;
        float = sq * fth;
        // float fth = sq * fth;

        result += (center_point_array[0] * fst +
                   center_point_array[1] * sq +
                   center_point_array[2] * thrd +
                   center_point_array[3] * fth);

    }

    result += (float)(num_bytes >> 2) * (center_point_array[4] + (center_point_array[5]));

    *target = result;
}

Listing 35. A NEON implementation of volk_arm_32f_x3_sum_of_poly_32f.

}
#ifdef LV_HAVE_NEON

Listing 36. A NEON implementation of
volk_arm_32fc_32f_dot_prod_32fc.

static inline void
volk_arm_32fc_32f_dot_prod_32fc_a_neon (lv_32fc_t* _restrict result, const lv_32fc_t* _restrict input, const float* _restrict taps, unsigned int num_points) {
  unsigned int number; const unsigned int quarterPoints = num_points / 4;
  float real[2];
  float *realpt = &res[0], *imagpt = &res[1];
  const float* inputPtr = (float*)input;
  const float* tapsPtr = taps;
  float zero[4] = {0.0f, 0.0f, 0.0f, 0.0f};
  float* real_accum;
  float current_accum = 0.0f;
  float accVector_real[4];
  float accVector_imag[4];
  float32x4_t inputVector;
  float32x4_t tmpReal, tmpImag;
  float32x4_t real_accumulator, imag_accumulator;

  // zero out accumulators
  // take a *float, return float32x4_t
  real_accumulator = vlldq_f32( zero );
  imag_accumulator = vlldq_f32( zero );

  for(number=0; number < quarterPoints; number++){
    // load taps ( float32x2x2_t = vlldq_f32( float32_t const * ptr ) )
    // load doublewords and duplicate in to second lane
    tapsVector = vlldq_f32(tapsPtr);

    // load quadword of complex numbers in to 2 lanes. 1st lane is real, 2nd imag
    inputVector = vlldq_f32(inputPtr);

    tmp_real = vmulq_f32(tapsVector, inputVector.val[0]);
    tmp_imag = vmulq_f32(tapsVector, inputVector.val[1]);

    real_accumulator = vaddq_f32(real_accumulator, tmp_real);
    imag_accumulator = vaddq_f32(imag_accumulator, tmp_imag);

    tapsPtr += 4;
    inputPtr += 8;
  }

  // store results back to a complex (array of 2 floats)
  vst1q_f32(accVector_real, real_accumulator);
  vst1q_f32(accVector_imag, imag_accumulator);
  *realpt = accVector_real[0] + accVector_real[1] +
  *imagpt = accVector_imag[0] + accVector_imag[1] +

  // clean up the remainder
  for(number=quarterPoints + 4; number < num_points; number++){
    *realpt += (**inputPtr++) + (**tapsPtr);
    *imagpt += (**inputPtr++) + (**tapsPtr++);
  }

  *result = *(lv_32fc_t*)(res[0]);
}
#endif /*LV_HAVE_NEON*/

Listing 37. A NEON implementation of
volk_arm_32fc_32f_dot_prod_32fc.

static inline void
volk_arm_32fc_x2_dot_prod_32fc_neon(lv_32fc_t* result, const lv_32fc_t* input, const lv_32fc_t* taps, unsigned int num_points) {
  unsigned int quarter_points = num_points / 4;
  unsigned int number;
  lv_32fc_t* a_ptr = (lv_32fc_t*)taps;
  lv_32fc_t* b_ptr = (lv_32fc_t*)input;
  // for 2-lane vectors. 1st lane holds the real part.
  // 2nd lane holds the imaginary part
  float32x4_t a_val, b_val, c_val; accumulator;
  float32x4_t tmp_real, tmp_imag;
  accumulator.val[0] = vdupq_n_f32(0);
  accumulator.val[1] = vdupq_n_f32(0);

  for(number = 0; number < quarter_points; ++number) {
    a_val = vld2q_f32((float*)a_ptr);
    a0r|a1r|a2r|a3r || a0i|a1i|a2i|a3i
    b_val = vld2q_f32((float*)b_ptr);
    b0r|b1r|b2r|b3r
    b builtin_prefetch(a_ptr+8);
    __builtin_prefetch(b_ptr+8);

    // multiply the real+real and imag+imag to get real result
    // a0r*b0r|a1r*b1r|a2r*b2r|a3r*b3r
    tmp_real.val[0] = vmulq_f32(a_val.val[0],
                               b_val.val[0]);
    a0i*b0i|a1i*b1i|a2i*b2i|a3i*b3i
    tmp_real.val[1] = vmulq_f32(a_val.val[1],
                               b_val.val[1]);

    // Multiply cross terms to get the imaginary result
    // a0r*b0i|a1r*b1i|a2r*b2i|a3r*b3i
    tmp_imag.val[0] = vmulq_f32(tmp_real.val[0],
                                  b_val.val[1]);
    a0i*b0r|a1i*b1r|a2i*b2r|a3i*b3r
    tmp_imag.val[1] = vmulq_f32(tmp_real.val[1],
                                  b_val.val[0]);

    c_val.val[0] = vsubq_f32(tmp_real.val[0],
                             tmp_real.val[1]);
    c_val.val[1] = vaddq_f32(tmp_imag.val[0],
                             tmp_imag.val[1]);

    accumulator.val[0] = vaddq_f32(accumulator.val[0], c_val.val[0]);
    accumulator.val[1] = vaddq_f32(accumulator.val[1], c_val.val[1]);

    a_ptr += 4;
    b_ptr += 4;
  }
}


```c
lv_32fc_t accum_result[4];
vs2q_f32((float*)accum_result, accumulator);
*result = accum_result[0] + accum_result[1] +
  accum_result[2] + accum_result[3];

// tail case
for (number = quarter_points * 4; number <
  num_points; ++number) {
  *result += (*a_ptr++) + (*b_ptr++);
}
}
```

Listing 38. A NEON implementation of
volk_arm_32fc_x2_dot_prod_32fc.

### LV HAVE NEON

```c
#define LV_HAVE_NEON

static inline void
volk_arm_32fc_x2_dot_prod_32fc_neon_optma(
  lv_32fc_t* result, const lv_32fc_t* input1, const
  lv_32fc_t* input2, unsigned int num_points) {
  unsigned int quarter_points = num_points / 4;
  unsigned int number;

  lv_32fc_t* a_ptr = (lv_32fc_t*) taps;
  lv_32fc_t* b_ptr = (lv_32fc_t*) input;
  // for 2-lane vectors, 1st lane holds the real
  // 2nd lane holds the imaginary part
  float32x4x2_t a_val, b_val, c_val, accumulator;
  float32x4x2_t tmp_real, tmp_imag;
  accumulator.val[0] = vdupq_n_f32(0);
  accumulator.val[1] = vdupq_n_f32(0);
  for (number = 0; number < quarter_points; ++number) {
    a_val = vld2q_f32((float*)a_ptr); // a0r|a1r
    b_val = vld2q_f32((float*)b_ptr); // b0r|b1r
    --builtin_prefetch(a_ptr+8);
    --builtin_prefetch(b_ptr+8);
    // do the first multiply
    tmp_imag.val[1] = vmlaq_f32(a_val.val[1],
      b_val.val[0], b_val.val[0]);
    tmp_imag.val[0] = vmlaq_f32(a_val.val[0],
      b_val.val[0], b_val.val[0]);
    // use multiply accumulate/subtract to get
    result = vmlaq_f32(tmp_imag.val[1],
      a_val.val[0], b_val.val[1]);
    tmp_imag.val[0] = vmlaq_f32(tmp_imag.val[0],
      a_val.val[1], b_val.val[1]);
    accumulator.val[0] = vaddq_f32(accumulator.
      val[0], tmp_imag.val[0]);
      val[1], tmp_imag.val[1]);
    // increment pointers
    a_ptr += 4;
    b_ptr += 4;
  }
  result += (*a_ptr++) + (*b_ptr++);
}
```

Listing 39. A NEON implementation of
volk_arm_32fc_x2_dot_prod_32fc.

### LV HAVE NEON

```c
#define LV_HAVE_NEON

static inline void
volk_arm_32fc_x2_dot_prod_32fc_neon_optmf(
  lv_32fc_t* result, const lv_32fc_t* input1, const
  lv_32fc_t* input2, unsigned int num_points) {
  unsigned int quarter_points = num_points / 4;
  unsigned int number;

  lv_32fc_t* a_ptr = (lv_32fc_t*) taps;
  lv_32fc_t* b_ptr = (lv_32fc_t*) input;
  // for 2-lane vectors, 1st lane holds the real
  // 2nd lane holds the imaginary part
  float32x4x2_t a_val, b_val, accumulator1, accumulator2;
  float32x4x2_t tmp_real, tmp_imag;
  accumulator1.val[0] = vdupq_n_f32(0);
  accumulator1.val[1] = vdupq_n_f32(0);
  accumulator2.val[0] = vdupq_n_f32(0);
  accumulator2.val[1] = vdupq_n_f32(0);
  for (number = 0; number < quarter_points; ++number) {
    a_val = vld2q_f32((float*)a_ptr); // a0r|a1r
    a_val = vld2q_f32((float*)a_ptr); // a0r|a1r
    a_val = vld2q_f32((float*)a_ptr); // a0r|a1r
    a_val = vld2q_f32((float*)a_ptr); // a0r|a1r
    --builtin_prefetch(a_ptr+8);
    --builtin_prefetch(b_ptr+8);
    // use 2 accumulators to remove inter-
    instruction data dependencies
    accumulator1.val[0] = vmlaq_f32(accumulator1.
      val[0], a_val.val[0], b_val.val[0]);
      val[1], a_val.val[1], b_val.val[1]);
    accumulator2.val[0] = vmlaq_f32(accumulator2.
      val[0], a_val.val[0], b_val.val[0]);
      val[1], a_val.val[1], b_val.val[1]);
    // increment pointers
    a_ptr += 4;
    b_ptr += 4;
  }
  result = vaddq_f32(accumulator1.val[0],
    accumulator2.val[0]);
  result = vaddq_f32(accumulator1.val[1],
    accumulator2.val[1]);
  result = vaddq_f32(accumulator1.val[0],
    accumulator2.val[0]);
  result = vaddq_f32(accumulator1.val[1],
    accumulator2.val[1]);
  for (number = quarter_points * 4; number <
    num_points; ++number) {
    *result += (*a_ptr++) + (*b_ptr++);
  }
}
```

Listing 40. A NEON implementation of
volk_arm_32fc_x2_dot_prod_32fc.
Listing 41. A NEON implementation of volk_arm_32fc_x2_dot_prod_32fc.

Listing 42. A NEON implementation of volk_arm_32f_sqr2f_32f_neon.

```c
#include <arm_neon.h>

static inline void volk_arm_32f_sqr2f_32f_neon(float* cVector, const float* aVector, unsigned int num_points)
{
    float* cPtr = cVector;
    const float* aPtr = aVector;
    unsigned int number = 0;
    unsigned int quarter_points = num_points / 4;
    float* ptr = aVector;
    float* recPtr = cVector;
    float* recPtr += 8;
    float* bPtr = (float*)ptr + 8;
    float* bPtr += 8;
    for (; number < quarter_points * 4; number++)
    {
        *cPtr++ += sqrtf(*aPtr++);
    }
}

#endif /* LV_HAVE_NEON */
```

Listing 41. A NEON implementation of volk_arm_32fc_x2_dot_prod_32fc.

```c
const lv_32fc_t* a_ptr = (lv_32fc_t*)嗪 tapped;
const lv_32fc_t* b_ptr = (lv_32fc_t*)input;
for (2 = lane_vectors, 1st lane holds the real part.
// 2nd lane holds the imaginary part
float32x4x4_t a_val, b_val, accumulator1, accumulator2;
float32x4x2_t reduced_accumulator;
accumulator1.val[0] = vdupq_n_f32(0);
accumulator1.val[1] = vdupq_n_f32(0);
accumulator1.val[2] = vdupq_n_f32(0);
accumulator1.val[3] = vdupq_n_f32(0);
accumulator2.val[0] = vdupq_n_f32(0);
accumulator2.val[1] = vdupq_n_f32(0);
accumulator2.val[2] = vdupq_n_f32(0);
accumulator2.val[3] = vdupq_n_f32(0);

// 8 input regs, 8 accumulators -> 16/16 neon regs are used
for (number = 0; number < quarter_points; ++number) {
    a_a_val = vld4q_f32((float*)a_ptr); // a0r | a1r
    b_b_val = vld4q_f32((float*)b_ptr); // b0r | b1r
    __builtin_prefetch(a_ptr+8);
    __builtin_prefetch(b_ptr+8);

    // use 2 accumulators to remove inter-instruction data dependencies
    accumulator1.val[0] = vmlaq_f32(accumulator1.val[0], a_val, b_val, 0);
    accumulator1.val[1] = vmlaq_f32(accumulator1.val[1], a_val, b_val, 1);
    accumulator1.val[2] = vmlaq_f32(accumulator1.val[2], a_val, b_val, 2);
    accumulator1.val[3] = vmlaq_f32(accumulator1.val[3], a_val, b_val, 3);

    accumulator2.val[0] = vmlaq_f32(accumulator2.val[0], a_val, b_val, 0);
    accumulator2.val[1] = vmlaq_f32(accumulator2.val[1], a_val, b_val, 1);
    accumulator2.val[2] = vmlaq_f32(accumulator2.val[2], a_val, b_val, 2);
    accumulator2.val[3] = vmlaq_f32(accumulator2.val[3], a_val, b_val, 3);

    // increment pointers
    a_a_ptr += 8;
    b_b_ptr += 8;
}

// reduce 8 accumulator lanes down to 2 (1 real and 1 imag)
accumulator1.val[0] = vaddq_f32(accumulator1.val[0], accumulator1.val[2]);
accumulator1.val[1] = vaddq_f32(accumulator1.val[1], accumulator1.val[3]);
accumulator2.val[0] = vaddq_f32(accumulator2.val[0], accumulator2.val[2]);
accumulator2.val[1] = vaddq_f32(accumulator2.val[1], accumulator2.val[3]);

// new reduce accumulators to scalars
lv_32fc_t accum_result[4];
vst2q_f32((float*)accum_result, reduced_accumulator);
result = accum_result[0] + accum_result[1] + accum_result[2] + accum_result[3];
```

```
// tail case
for (number = quarter_points * 8; number < num_points; ++number) {
    *result += (*a_ptr++) + (*b_ptr++);
}
```
#param aVector The initial vector
#param bVector The vector to be subtracted
#param num_points The number of values in aVector

static inline void volk_arm_32f_x2_subtract_32f_neon(
    float * cVector, const float* aVector, const float* bVector, unsigned int num_points) {
    float32x4_t a_vec, b_vec, c_vec;
    for (number = 0; number < quarter_points; number++)
    {
        a_vec = vld1q_f32(aPtr);
        b_vec = vld1q_f32(bPtr);
        c_vec = vmaxq_f32(a_vec, b_vec);
        vst1q_f32(cPtr, c_vec);
        aPtr += 4;
        bPtr += 4;
        cPtr += 4;
    }

    #endif /* LV_HAVE_NEON */
}

Listing 45. A NEON implementation of volk_arm_32f_x2_subtract_32f.
Listing 46. A NEON implementation of volk Arm_32f_s32f_multiply_32f.

```c

#define LV_HAVE_NEON

// brief Scalar float multiply
// param cVector The vector where the results will be stored
// param aVector One of the vectors to be multiplied
// param scalar the scalar value
// param num_points The number of values in aVector
// and bVector to be multiplied together and stored into cVector

static inline void volk_arm_32f_s32f_multiply_32f_a_neon(float* cVector, const float* aVector, const float scalar, unsigned int num_points) {
    unsigned int number = 0;
    const float* inputPtr = aVector;
    float* outputPtr = cVector;
    unsigned int quarterPoints = num_points / 4;

    float32x4_t aVal, cVal;
    for (number = 0; number < quarterPoints; number++) {
        aVal = vldl_f32(inputPtr); // Load into NEON regs
        cVal = vmulq_n_f32(aVal, scalar); // Do the multiply.
        vstlq_f32(outputPtr, cVal); // Store results back to output
        // print(“%2.4f * %2.4f = %2.4f\n”, *inputPtr, scalar, *outputPtr);
        inputPtr += 4;
        outputPtr += 4;
    }
    for (number = quarterPoints * 4; number < num_points; number++) {
        outputPtr += (*inputPtr++) * scalar;
    }
}

Listing 47. A NEON implementation of volk Arm_32f_s32f_multiply_32f.

```
Listing 49. A NEON implementation of volk_arm_32f_x2_dot_prod_32f.

```c
#include <arm_neon.h>
static inline void volk_arm_16i_max_star_16i_neon(
    s16 *target, short *src0, uint8_t num_points)
{
    const unsigned int eighth_points = num_points / 8;
    unsigned number;
    int6x8_t input_vec;
    int6x8_t diff, max_vec, zeros;
    uint6x8_t compl, comp2;
    zeros = veorq_s16(zeros, zeros);
    int6x8x2_t tmpvec;
    int6x8_t candidate_vec = vld1q_dup_s16(src0);
    ++src0;
    for (number = 0; number < eighth_points; ++number)
    {
        input_vec = vld1q_s16(src0);
        __builtin_prefetch(src0+16);
        compl = veceq_s16(diff, zeros);
        comp2 = vcltq_s16(diff, zeros);
        tmpvec.val[0] = vandq_s16(candidate_vec, (int6x8_t)compl);
        tmpvec.val[1] = vandq_s16(input_vec, (int6x8_t)comp2);
        candidate_vec = vaddq_s16(tmpvec.val[0], tmpvec.val[1]);
        src0 += 8;
    }
    vst1q_s16(&candidate, candidate_vec);
    for (number = 0; number < num_points % 8; number++)
    {
        candidate = ((int16_t)(candidate - src0[0]) > 0) ? candidate : src0[number];
    }
    target[0] = candidate;
}
```

Listing 50. A NEON implementation of volk_arm_16i_max_star_horizontal_16i.

```c
#include <arm_neon.h>
static inline void volk_arm_16i_max_star_horizontal_16i_neon(
    int16_t* target, int16_t* src0, unsigned int num_points)
{
    const unsigned int eighth_points = num_points / 16;
    unsigned number;
    int6x8x2_t input_vec;
    int6x8_t diff, max_vec, zeros;
    uint6x8_t compl, comp2;
    zeros = veorq_s16(zeros, zeros);
    for (number = 0; number < eighth_points; ++number)
    {
        input_vec = vld2q_s16(src0);
        // __builtin_prefetch(src0+16);
        compl = veceq_s16(diff, zeros);
        comp2 = vcltq_s16(diff, zeros);
        input_vec.val[0] = vaddq_s16(input_vec.val[0], (int16x8_t)compl);
```

Listing 51. A NEON implementation of volk_arm_16i_max_star_horizontal_16i.
Following are code listings of proto-kernels we developed in ARM assembly.

```c
@ static inline void
volk_arm_32fc_32f_dot_prod_32fc_unrollasm (
  lv_32fc_t* result, const lv_32fc_t* input,
  const float* taps, unsigned int num_points)
.global volk_arm_32fc_32f_dot_prod_32fc_unrollasm:
@ r0 – result: pointer to output array (32fc)
@ r1 – input: pointer to input array 1 (32fc)
@ r2 – taps: pointer to input array 2 (32f)
@ r3 – num_points: number of items to process

push {r4, r5, r6, r7, r8, r9}
vpush {q4–q7}
sub r13, r13, #56 @ 0x38
add r12, r13, #8
lsrs r8, r8, #3
vsrr q2, q5, q5
vadd.f32 q3, q5, q5
vadd.f32 q4, q4, q5
beq .smallvector
vld2.32 {d20–d23}, [r1!]
vld1.32 {d24–d25}, [r2!]
mov r5, #1

.mainloop:
  vld2.32 {d14–d17}, [r1!] @ q7, q8
  vld1.32 {d18–d19}, [r2!] @ q9
  vmul.f32 q0, q12, q10 @ real mult
  vmul.f32 q1, q12, q11 @ imag mult
  add r5, r5, #1
  cmp r5, r8
  vadd.f32 q4, q4, q0 @ q4 accumulates real
  vadd.f32 q5, q5, q1 @ q5 accumulates imag
  vld2.32 {d20–d23}, [r1!] @ q10–q11
  vld1.32 {d24–d25}, [r2!] @ q12
  vmul.f32 q13, q9, q7
  vmul.f32 q14, q9, q8
  vadd.f32 q2, q2, q13 @ q2 accumulates real
  vadd.f32 q3, q3, q14 @ q3 accumulates imag
  bne .mainloop
  vmul.f32 q0, q12, q10 @ real mult
  vmul.f32 q1, q12, q11 @ imag mult
  vadd.f32 q4, q4, q0 @ q4 accumulates real
  vadd.f32 q5, q5, q1 @ q5 accumulates imag

.bsmallvector:
  vadd.f32 q0, q2, q4
  add r12, r13, #24
  lsl r8, r8, #3
  vadd.f32 q1, q3, q5
  cmp r3, r8
  vadd.f32 d0, d0, d1
  vadd.f32 d1, d2, d3
  vadd.f32 s14, s0, s1
  vadd.f32 s15, s2, s3
  vstr s14, [r13]
  bis .D1
  vstr s15, [r13, #4]
  ldr r12, r8, r3
  lsr r4, r12, #2
  cmp r4, #0
  cmpne r12, r3
  lsl r5, r4, #2
  movhi r6, #0
  movls r6, #1
  bhi .smallloop
  vadd.f32 q10, q10, #0 @ 0x00000000
  mov r9, r1
  mov r7, r2
  vror q11, q10, q10

.ssmallloop:
  add r6, r6, #1
  vld2.32 {d16–d19}, [r9!]
  cmp r4, r6
  vld1.32 {d24–d25}, [r7!]
  vmla.f32 q11, q12, q8
  vmla.f32 q10, q12, q9
  bhi .smallloop
  vadd.f32 q9, q9, #0 @ 0x00000000
  cmp r12, r5
  vadd.f32 d20, d20, d21
  add r8, r8, r5
  vror q8, q9, q9
  add r1, r1, r5, lsl #3
  vadd.f32 d22, d22, d23
  add r2, r2, r5, lsl #2
  vpadd.f32 d18, d20, d20
  vpadd.f32 d16, d22, d22
  vadd.f32 d4, d18
  cmp r13, r4
  vadd.f32 s15, s13, s15
  vadd.f32 s14, s13, s14
  beq .finishreduction
  .L1:
  add r12, r8, #1
  vldr s13, [r2!]
  cmp r3, r12
  vldr s11, [r1!]
  vldr s12, [r1, #4]
  vmla.f32 s14, s13, s11
  vmla.f32 s15, s13, s12
  bls .finishreduction
  add r8, r8, #2
  vldr s13, [r2, #4]
  cmp r3, r8
  vldr s11, [r1, #8]
  vldr s12, [r1, #12]
  vmla.f32 s14, s13, s11
  vmla.f32 s15, s13, s12
  bls .finishreduction
  vldr s13, [r2, #8]
  vldr s11, [r1, #16]
  vldr s12, [r1, #20]
  vmla.f32 s14, s13, s11
  vmla.f32 s15, s13, s12
  bls .finishreduction
  .finishreduction:
  vstr s14, [r13]
  vstr s15, [r13, #4]
  .D1:
  ldr r3, [r13]
  str r3, [r0]
  ldr r3, [r13, #4]
  str r3, [r0, #4]
  add r13, r13, #56 @ 0x38
  vpop {q4–q7}
  pop {r4, r5, r6, r7, r8, r9}
  bx r14
```
Listing 53. A NEON ASM implementation of
asm/neon/volk_arm_32fc_32f_dot_prod_32fc_unrollasm.

@ static inline void
volk_arm_32fc_x2_dot_prod_32fc_neonasm(float const * cVector, const float * aVector, const float * bVector, unsigned int num_points);

.global volk_arm_32fc_x2_dot_prod_32fc_neonasm

volk_arm_32fc_x2_dot_prod_32fc_neonasm:
push {r4, r5, r6, r7, r8, lr}
vpush {q0-q17}
mov r8, r3 @ hold on to num_points (r8)
@ zero out accumulators — leave 1 reg in alu
veor q8, q15, q15
mov r7, r0 @ (r7) is cVec
veor q9, q15, q15
mov r5, r1 @ (r5) is aVec
veor q10, q15, q15
mov r6, r2 @ (r6) is bVec
veor q11, q15, q15
lsrs r3, r3, #3 @ eighth_points (r3) = num_points/8
veor q12, q15, q15
mov r12, r2 @ (r12) is bVec
veor q13, q15, q15
mov r4, r1 @ (r4) is aVec
veor q14, q15, q15
veor q15, q15, q15
beq .smallvector @ nathan optimized this file based on an old jumpdb @ but I don’t understand this jump. Seems like it should go to loop2 @ and smallvector (really vector reduction) shouldn’t need to be a label
mov r2, #0 @ 0 out r2 (now number)

.loop1:
add r2, r2, #1 @ increment number
vlld 32 {d0, d2, d4, d6}, [r12]! @ q0-q3
cmp r2, #8 @ is number < eighth_points
@ pld [rd, #64]
vlld 32 {d8, d10, d12, d14}, [r4]! @ q4-q7
@ pld [rd, #64]
vmla.32 q12, q4, q0 @ real (re*re)
vmla.32 q14, q4, q1 @ imag (re*im)
vmls.32 q15, q5, q1 @ real (im*im)
vmla.32 q13, q5, q0 @ imag (im*im)
vmla.32 q16, q5, q2 @ real (re*re)
vmls.32 q10, q3, q7 @ imag (re*im)
vmls.32 q11, q3, q6 @ imag (im*im)
bne .loop1

lsl r2, r3, #3 @ r2 = eighth_points * 8
add r6, r6, r2 @ bVec = bVec + eighth_points — whooooy gcc?!
add r5, r5, r2 @ aVec = aVec + eighth_points
@ q12-q13 were original real accumulators
@ q14-q15 were original imag accumulators
@ reduce 8 accumulators down to 2 (1 real, 1 imag)
vadd.32 q8, q10, q8 @ real + real
vadd.32 q11, q11, q9 @ imag + imag
vadd.32 q12, q12, q15 @ real + real
vadd.32 q14, q14, q13 @ imag + imag
vadd.32 q9, q9, q1 @ real + real
.vsmallvector:
lsl r4, r3, #3
cmp r8, r4
vst2.32 {d16-d19}, [sp :64] @ whaataat? no way
this is necessary!
vldr s15, [sp, #8]
vldr s17, [sp]
vldr s16, [sp, #4]

Listing 54. A NEON ASM implementation of
asm/neon/volk_arm_32fc_x2_dot_prod_32fc_neonasm.

@ static inline void
volk_arm_32fc_32f_dot_prod_32fc_neonasm(float const * lv_32fc_t*, result); const lv_32fc_t* input, const float* taps, unsigned int num_points);

.global volk_arm_32fc_32f_dot_prod_32fc_neonasm pipeline

volk_arm_32fc_32f_dot_prod_32fc_neonasmpipeline:
@ r0 — result: pointer to output array (32f)
@ r1 — input: pointer to input array 1 (32fc)
@ r2 — taps: pointer to input array 2 (32f)
@ r3 — number: number of items to process
result .req r0
input .req r1
taps .req r2
num_points .req r3
quarterPoints .req r7
number .req r8

@ Note that according to the ARM EABI (AAPCS)
Section 5.1.1: @ registers s16–s31 (d8–d15, q4–q7) must be
preserved across subroutine calls;
@ registers s0–s15 (d0–d7, q0–q3) do not need to be
preserved @ registers d16–d31 (q8–q15), if present, do not
need to be preserved.
realAccQ .req q0 @ d0–d10 @ s0–s3
compAccQ .req q1 @ d2–d3 @ s4–s7
realAccS .req s0 @ d0[0]
compAccS .req s4 @ d2[0]
tapsVal .req q2 @ d4–d5
outputVal .req q3 @ d6–d7
realMul .req q8 @ d8–d9
compMul .req q9 @ d6–d17
inRealVal .req q10 @ d18–d19
inCompVal .req q11 @ d20–d21
```

Listing 55. A NEON ASM implementation of
asm/neon/volk_arm_32fc_s32f_multiply_32f_neonasm

```
Listing 57. A NEON ASM implementation of asm/neon/volk_arm_32f_x2_dot_prod_32f_neonasm.

Listing 58. A NEON ASM implementation of asm/neon/volk_arm_32f_x3_sum_of_poly_32f_a_neonasm.
Listing 59. A NEON ASM implementation of asm/neon/volk_arm_16i_max_star_horizontal_16i.

@ static inline void
volk_arm_16i_max_star_horizontal_16i_neonasm(  
float* cVector, const float* aVector, const  
float* bVector, unsigned int num_points);
.global  
volk_arm_16i_max_star_horizontal_16i_neonasm
volk_arm_16i_max_star_horizontal_16i_neonasm:
 @ r0 – cVector: pointer to output array
 @ r1 – aVector: pointer to input array 1
 @ r2 – num_points: number of items to process

volk_arm_16i_max_star_horizontal_16i_neonasm:
p1d [ r1 :128 ]
push { r4 , r5 , r6 }  
@ preserve register
lsrs r5 , r2 , #4  
@ 1/16th points =
num_points/16
vmov . i32 q12 , #0  
@ q12 = [0,0,0]
beq , smallvector
@ less than 16 elements
in vector
mov r4 , r1  
@ r4 = aVector
mov r12 , r0  
@ gcc calls this ip
mov r3 , #0  
@ number = 0
.loop1:
  vld2.16 { d16–d19 }, [ r4 ! ]  
  @ aVector, interleaved load
  pld [ r4 :128 ]
  add r3 , r3 , #1  
  @ number += 1
cmp r3 , r5  
  @ number < 1/16th points
  vsub . i16 q10 , q8 , q9  
  @ subtraction
  vcge . s16 q11 , q10 , #0  
  @ result > 0?
  vcgt . s16 q10 , q12 , #0  
  @ result < 0?
  vand . i16 q11 , q8 , q11  
  @ multiply by
  comparison
  vadd . i16 q10 , q11 , q10  
  @ add results to get max
  vst1.16 { d20–d21 }, [ r12 ! ]  
  @ store the results
  bne . loop1  
  @ at least 16 items
  left
  add r1 , r1 , r3 , sls #5
  add r0 , r0 , r3 , sls #4
  smallvector:
    ands r2 , r2 , #15
    beq . end
    mov r3 , #0
 .loop3:
    ldrh r4 , [ r1 ]
    b ic r5 , r3 , #1
    ldrh ip , [ r1 , #2 ]
    add r3 , r3 , #2
    add r1 , r1 , #4
    rsb r6 , ip , r4
    sxth r6 , r6
    cmp r6 , #0
    movgt ip , r4
    cmp r3 , r2
    strh ip , [ r0 , r5 ]
    bcc . loop3
 .end:
    pop { r4 , r5 , r6 }
    bx lr

volk_arm_32fc_x2_dot_prod_32fc_neonasm_opttests(  
float* cVector, const float* aVector, const  
float* bVector, unsigned int num_points )
.global  
volk_arm_32fc_x2_dot_prod_32fc_neonasm_opttests
volk_arm_32fc_x2_dot_prod_32fc_neonasm_opttests:
push { r4 , r5 , r6 , r7 , r8 , r9 , sl , fp , lr }

@ static inline void
volk_arm_32fc_x2_dot_prod_32fc_neonasm_opttests(  
float* cVector, const float* aVector, const  
float* bVector, unsigned int num_points )
.global  
volk_arm_32fc_x2_dot_prod_32fc_neonasm_opttests
Listing 60. A NEON ASM implementation of
asm/neon/volk_arm_32fc_w2_dot_prod_32fc_neonasm_opt.txt

@ static inline void
volk_arm_32fc_32f_dot_prod_32fc_a_neonasm (lv_32fc_t* result, const lv_32fc_t* input,
const float* taps, unsigned int num_points) {
    .global volk_arm_32fc_32f_dot_prod_32fc_a_neonasm
volk_arm_32fc_32f_dot_prod_32fc_a_neonasm:
    @ r0 – result: pointer to output array (32fc)
    @ r1 – input: pointer to input array 1 (32fc)
    @ r2 – taps: pointer to input array 2 (32f)
    @ r3 – num_points: number of items to process
    result .req r0
    input .req r1
    taps .req r2
    num_points .req r3
    quarterPoints .req r7
    number .req r8
    @ Note that according to the ARM EABI (AAPCS)
    @ registers s16–s31 (d8–d15, q4–q7) must be
    @ preserved across subroutine calls;
    @ registers s0–s15 (d0–d7, q0–q3) do not need to
    @ be preserved.
    realAccQ .req q0 @ d0–d1/s0–s3
    compAccQ .req q1 @ d2–d3/s4–s7
    realAccS .req s0 @ d0[0]
    compAccS .req s4 @ d2[0]
    tapsVal .req q2 @ d4–d5
    outVal .req q3 @ d6–d7
    realMul .req q8 @ d8–d9
    compMul .req q9 @ d16–d17
    inRealVal .req q10 @ d18–d19
    inCompVal .req q11 @ d20–d21
    stmf sp!, {r7, r8, s1} @ prologue – save register states
    veor realAccQ, realAccQ @ zero out accumulators
    veor compAccQ, compAccQ @ zero out accumulators
    movs quarterPoints, num_points, lsr #2
    beq .loop2 @ if zero into quarterPoints
    mov number, quarterPoints

    .loop1:
    @ do work here
    @pld [taps, #128] @ pre-load hint – this is
    @pld [input, #128] @ pre-load hint – this is
    implementation specific!
    @pld [taps, #128] @ pre-load hint – this is
    implementation specific!
    vid1.32 {d4–d5}, [taps]! @ tapsVal
    vid2.32 {d20–d23}, [input]! @ inRealVal, inCompVal
    vmul.f32 realMul, tapsVal, inRealVal
    vmul.f32 compMul, tapsVal, inRealVal
    vadd.f32 realAccQ, realAccQ, realMul
    vadd.f32 compAccQ, compAccQ, compMul
    subs number, number, #1
    bne .loop1 @ first loop
    @ Sum up across realAccQ and compAccQ
    vpaddd.f32 d0, d0, d1 @ realAccQ -> d0
    vpaddd.f32 d2, d2, d3 @ compAccQ -> d2
    vadd.f32 realAccS, s0, s1 @ sum the contents of
d0 together (realAccQ)
    vadd.f32 compAccS, s4, s5 @ sum the contents of
d2 together (compAccQ)
    @ critical values are now in s0 (realAccS), s4 (realAccQ)
    mov number, quarterPoints, asl #2

    .loop2:
    cmp num_points, number
    bis .done
    vid1.32 {d4[0]}, [taps]! @ s8
    vid2.32 {d5[0],d6[0]}, [input]! @ s10, s12
    vmul.f32 s5, s8, s10
    vmul.f32 s6, s8, s12
    vadd.f32 realAccS, realAccS, s5
    vadd.f32 compAccS, compAccS, s6
    add number, number, #1
    b .loop2
    .done:
    vst1.32 {d0[0]}, [result]! @ realAccS
    vst1.32 {d2[0]}, [result]! @ compAccS
    ldmdf sp!, {r7, r8, s1} @ epilogue – restore
    register states
    bx lr

Listing 61. A NEON ASM implementation of
asm/neon/volk_arm_32fc_32f_dot_prod_32fc_a_neonasm.

@ static inline void
volk_arm_32f32f_multiply_32f_a_neonasm(float* cVector, const float* aVector, const float
scalar, unsigned int num_points):
    .global volk_arm_kernel_name_here_a_neonasm
volk_arm_kernel_name_here_a_neonasm:
    @ r0 – cVector: pointer to output array
    @ r1 – aVector: pointer to input array 1
    @ r2 – scalar: pointer to input 2 (scalar or array
    depending on kernel)
    @ r3 – num_points: number of items to process
    cVector .req r0
    aVector .req r1
    scalar .req r2
    num_points .req r3
    quarterPoints .req r7
    number .req r8
    @ aliases for neon registers
    aVal .qn q0.f32 @ d0–d1
    bVal .qn q1.f32 @ d2–d3
    cVal .qn q2.f32 @ d4–d3
    scalarVal .dn d3.f32[0]
    @ AAPCS Section 5.1.1
    @ A subroutine must preserve the contents of the
    registers r4–r8, r10, r11 and SP
    stmf sp!, {r7, r8, s1} @ prologue – save register states
    @ quarterPoints = num_points / 4
    movs quarterPoints, num_points, lsr #2
    beq .loop2 @ if zero into quarterPoints
    mov number, #0 @ number, 0
    vmov scalarVal, scalar @ load scalar to neon
    register
.loop1:
    pld [ aVector , #128 ] @ pre-load hint — this is implementation specific!
    vld1 { aVal }, [ aVector:128 ]! @ load vector
    @ do operations here
    vst1 { cVal }, [ cVector:128 ]! @ store the result
    @ number += 1; if number < quarterpoints goto loop1, otherwise continue
    add number, number, #1
    cmp number, quarterPoints
    bne .loop1 @ first loop
    mov number, quarterPoints, asl #2
    @ it can make reading easier to unassign labels and reassign them here
    .unreq aVal
    aVal .dn d0.f32
    @ handle the tail case
    .loop2:
    cmp num_points, number
    b .loop2 @ epilogue — restore register states, and we are done
    .done:
    ldmfd sp!, { r7, r8, sl } @ epilogue — restore register states
    bx lr

Listing 62. A NEON ASM implementation of asm/neon/volk_arm_asm_template.

static inline void
    volk_arm_32fc_32f_dot Prod_32fc_a_neonpipeline (__float128 r0, __float128 r1)
    __attribute__((always_inline))
    { volk_arm_32fc_32f_dot Prod_32fc_a_neonpipeline_v0result, const __float128 r2, const __float32 r3, __uint32 num_points, __uint32 r7
    @ Note that according to the ARM EABI (AAPCS) Section 5.1.1:
    @ registers s16–s31 (d8–d15, q4–q7) must be preserved across subroutine calls;
    @ registers s0–s15 (d0–d7, q0–q3) do not need to be preserved
    @ registers d16–d31 (q8–q15), if present, do not need to be preserved.
    realAccQ .req q0 @ d0–d1/s0–s3
    realAccQ .req q1 @ d2/d3/s4–s7
    realAccS .req s0 @ d0[0]
    compAccS .req s4 @ d2[0]
    tapsVal .req q2 @ d4–d5
    outVal .req q3 @ d6–d7
    realNum .req q8 @ d8–d9
    compNum .req q9 @ d16–d17
    inRealVal .req q10 @ d18–d19
    inCompVal .req q11 @ d20–d21
    mov number, num points, lsr #2
    @ Optimizing for pipeline
    vld1.32 { d4–d5 }, [ taps:128 ]! @ tapsVal
    vld2.32 { d20–d23 }, [ input:128 ]! @ inRealVal,
    inCompVal
    subs number, number, #1
    .loop1:
    @ do work here
    pld [ taps, #128 ] @ pre-load hint — this is implementation specific!
    volk_arm_32fc_32f_dot Prod_32fc_a_neonpipeline_v0result, const __float128 r1
    @ Sum up across realAccQ and compAccQ
    add number, number, #1
    cmp number, quarterPoints
    bne .loop2 @ first loop
    mov number, quarterPoints, asl #2
    .loop2:
    cmp num_points, number
    b .loop2 @ epilogue — restore register states, and we are done
    .done:
Listing 63. A NEON ASM implementation of
asm/neon/volk_arm_32fc_32f_dot_prod_32fc_a_neonpipeline

@ static inline void
volk_arm_32fc_32f_dot_prod_32fc_a_neonasmvmla (  
  lv_32fc_t* result, const lv_32fc_t* input,  
  const float* taps, unsigned int num_points)
.global
volk_arm_32fc_32f_dot_prod_32fc_a_neonasmvmla
volk_arm_32fc_32f_dot_prod_32fc_a_neonasmvmmla
  @ r0 — result: pointer to output array (32fc)  
  @ r1 — input: pointer to input array 1 (32fc)  
  @ r2 — taps: pointer to input array 2 (32f)  
  @ r3 — num_points: number of items to process
result .req r0
input .req r1
taps .req r2
num_points .req r3
quarterPoints .req r7
number .req r8
@ Note that according to the ARM EABI (AAPCS)
  Section 5.1.1:  
  @ registers s16—s31 (d8—d15, q4—q7) must be  
  preserved across subroutine calls;  
  @ registers s0—s15 (d0—d7, q0—q3) do not need to  
  be preserved  
  @ registers d16—d31 (q8—q15), if present, do not  
  need to be preserved.
realAccQ .req q0 @ d0/d1/s0/s3
compAccQ .req q1 @ d2/d3/s4/s7
realAccS .req s0 @ d0[0]
compAccS .req s4 @ d2[0]
tapsVal .req q2 @ d4—d5
outVal .req q3 @ d6—d7
realMul .req q8 @ d8—d9
compMul .req q9 @ d16—d17
inRealVal .req q10 @ d18—d19
inCompVal .req q11 @ d20—d21

smfd sp!, {r7, r8, s1} @ prologue — save register
states
veor realAccQ, realAccQ @ zero out accumulators
veor compAccQ, compAccQ @ zero out accumulators
movs quarterPoints, num_points, lsr #2
beq .loop2 @ if zero into quarterPoints
mov number, quarterPoints

.loop1:
  @ do work here
  pld {taps, #128} @ pre-load hint — this is
    implementation specific!
  pld {input}, #128 @ pre-load hint — this is
    implementation specific!
  vld1.32 {d4—d5}, {taps!} @ tapsVal
  vld1.32 {d8—d21}, {input!} @ inRealVal,
inCompVal
  vmla.f32 realAccQ, tapsVal, inRealVal
  vmla.f32 compAccQ, tapsVal, inCompVal
  subs number, number, #1
  bne .loop1 @ first loop
  @ Sum up across realAccQ and compAccQ
  vpadd.f32 d0, d0, d1 @ realAccQ ↔ d0
  vpadd.f32 d2, d2, d3 @ compAccQ ↔ d2
  vadd.f32 realAccS, s0, s1 @ sum the contents of
  d0 together (realAccQ)
  vadd.f32 compAccS, s4, s5 @ sum the contents of
  d2 together (compAccQ)
  @ critical values are now in s0 (realAccS), s4 (compAccS)

Listing 64. A NEON ASM implementation of
asm/neon/volk_arm_32fc_32f_dot_prod_32fc_a_neonasmvmla.

@ static inline void
volk_arm_32fc_x2_add_32f_a_neonpipeline(float*  
  cVector, const float* aVector, const float*  
  bVector, unsigned int num_points);
.global
volk_arm_32fc_x2_add_32f_a_neonpipeline
volk_arm_32fc_x2_add_32f_a_neonpipeline:
  @ r0 — cVector: pointer to output array
  @ r1 — aVector: pointer to input array 1
  @ r2 — bVector: pointer to input array 2
  @ r3 — num_points: number of items to process
  cVector .req r0
  aVector .req r1
  bVector .req r2
  num_points .req r3
  quarterPoints .req r7
  number .req r8
  aVal .req q0 @ d0—d1
  bVal .req q1 @ d2—d3
  cVal .req q2 @ d4—d5

smfd sp!, {r7, r8, s1} @ prologue — save register
states
pld {aVector, #128} @ pre-load hint — this is
  implementation specific!
pld {bVector, #128} @ pre-load hint — this is
  implementation specific!

movs quarterPoints, num_points, lsr #2
beq .loop2 @ if zero into quarterPoints
mov number, quarterPoints

@ Optimizing for pipeline
  vld1.32 {d0—d1}, {aVector:128}! @ aVal
  vld1.32 {d2—d3}, {bVector:128}! @ bVal
  subs number, number, #1

.loop1:
  pld {aVector, #128} @ pre-load hint — this is
    implementation specific!
  pld {bVector, #128} @ pre-load hint — this is
    implementation specific!
  vadd.f32 cVal, bVal, aVal
  vld1.32 {d0—d1}, {aVector:128}! @ aVal
  vld1.32 {d2—d3}, {bVector:128}! @ bVal
  vst1.32 {d4—d5}, {cVector:128}! @ cVal
  subs number, number, #1
  bne .loop1 @ first loop
  @ One more time
  vadd.f32 cVal, bVal, aVal
Listing 65. A NEON ASM implementation of
asm/neon/volk_arm_32f_x2_add_32f_a_neonpipeline.

.virtual inline void
void arm_neon_volk addToAccumulator(float* target, float* src0, float* centerPointArray, float* cutoff, unsigned int numPoints)
{
    r0 = cVector: pointer to output array
    r1 = src0: pointer to input array 1
    r2 = centerPointArray: pointer to input array 2
    r3 = numPoints: number of items to process
    cVector .req r0
    aVector .req r1
    bVector .req r2
    numPoints .req r3
    quarterPoints .req r7
    number .req r8
    aVal .req q0 @ d0–d1
    bVal .req q1 @ d2–d3
    cVal .req q2 @ d4–d5

    @ APCS Section 5.1.1
    @ A subroutine must preserve the contents of the
    @ registers r4–r8, r10, r11 and SP
    smfld sp!, {r7, r8, s1} @ prologue – save register
    states

    movs quarterPoints, numPoints, lsr #2
    beq .loop2 @ if zero into quarterPoints

    mov number, #0 @ number = 0

    .loop1:
        pld [aVector, #128] @ pre-load hint – this is
        implementation specific!
        pld [bVector, #128] @ pre-load hint – this is
        implementation specific!

        vldl.32 {d0–d1}, [aVector:128]! @ aVal
        add number, number, #1
        vldl.32 {d2–d3}, [bVector:128]! @ bVal
        vadd.f32 s2, bVal, aVal
        cmp number, quarterPoints
        vstl.32 {d4–d5}, [cVector:128]! @ cVal
        ble .loop1 @ first loop

        mov number, quarterPoints, asl #2

    .loop2:
        cmp numPoints, number
        bls .done

        vldl.32 {d0–d1}, [aVector]!
        vldl.32 {d2–d3}, [bVector]!
        vadd.f32 s2, s1, s0

    .done:
        ldmdf sp!, {r7, r8, s1} @ epilogue – restore
        register states
        bx lr

Listing 66. A NEON ASM implementation of
asm/neon/volk_arm_32f_x2_add_32f_a_neonasm.

@ static inline void
volf arm_32f_x3_sum_of_poly_32f_a_neonasm (float* target, float* src0, float* centerPointArray, float* cutoff, unsigned int numPoints)
{
    target .req r0 @ address of vector
    src0 .req r1 @ address of vector
    centerPointArray .req r2 @ address of vector
    cutoff .req r3 @ address of scalar
    numPoints .req r4 @ scalar
    number .req r5 @ scalar used for loop
    control

    @ Note that according to the ARM EABI (APCS)
    Section 5.1.1:
    @ registers s16–s31 (d8–d15, q4–q7) must be
    preserved across subroutine calls;
    @ registers s0–s15 (d0–d7, q0–q3) do not need to
    be preserved
    @ registers d16–d31 (q8–q15), if present, do not
    need to be preserved.

    quarterPoints .req r6 @ number of items to process
    xVal .req q0 @ d0–d1 / s0–s3
    cutoffVal .req q1 @ d2–d3 / s4–s7
    accumulator .req q2 @ d4–d5 / s8–s11
    container .req q3 @ d6–d7 / s12–s15
    cpa0Val .req q8
    cpa1Val .req q9
    cpa2Val .req q10
    cpa3Val .req q11
    x2Val .req q12 @ afterthought — neded
    a register to hold x^2

    @ get numPoints from the stack
    smfld sp!, {r4–r9, s1} @ prologue – save register
    states (6 longs)
    ldr numPoints, {sp, #8(8)} @ numPoints (parameter 5)

    movs quarterPoints, numPoints, lsr #2 @
    numPoints / 4
    beq .loop2 @ if zero in to quarterPoints

    mov number, #0 @ number = 0

    vfor accumulator, accumulator @ zero out
    accumulator
    vldl.32 {d18–d19}, [centerPointArray:128]! @
    cpa_qvector

    .loop1:
    @ do work here
    pld [src0, #128] @ pre-load hint – this is
    implementation specific!
Listing 67. A NEON ASM implementation of asm/neon/volk_arm_32f_x3_sum_of_poly_32f_a_neonveraasm.

@ static inline void
volk_arm_32f_x2_dot_prod_32f_neonasm_opts(volk_arm_32f_x2 dot_prod_32f_neonasm_opts:
push {r4, r5, r6, r7, r8, r9, r10, r11}
@ sixteenth_points = num_points / 16
lsrs r8, r3, #4
sub r13, r13, #16 @ subtracting 16 from stack pointer? , wat?
@ 0 out neon accumulators
veor r0, q3, q3
veor r1, q3, q3
veor r2, q3, q3
veor r3, q3, q3
beq .smallvector @ if less than 16
points skip main loop
mov r7, r2 @ copy input ptrs
mov r6, r1 @ copy input ptrs
mov r5, #0 @ loop counter

.mainloop:
vdup.32 src_vector, src_val
@ Get a vector of max(src0, cutoff)
vmov.f32 x_to_1, src_vector, cutoff_vector
vmul.f32 x_to_2, x_to_1, x_to_2 @ x’2
vmul.f32 x_to_3, x_to_2, x_to_3 @ x’3
vmul.f32 x_to_4, x_to_3, x_to_4 @ x’4
@ zip up doubles to interleave
vmov d24, x_to_1 @ x_low
vzip.f32 d24, x_to_2 @ [x’2 | x’1 | x’2 | x’1]
@vmp d26, x_to_3 @ x_high
vzip.f32 d26, x_to_4 @ [x’4 | x’3 | x’4 | x’3]
@ x_qvector = vcombine.f32(x_low.val[0], x_high.val[0]);
@vmp d16, d24 @ x_qvector, x_low.val[0]
@vmp d17, d26 @ x_qvector, x_high.val[0]

// now we finally have [x’4 | x’3 | x’2 | x’1] !
@vmlaq.f32 c_qvector, x_qvector, cpa_qvector
@ad number, number, #1
@ b .loop2

 looph2:
@cmp num_points, number
@bli .done
@ vpl [cutoff, #128] @ pre-load hint – this is implementation specific!
@ vld1.32 {xVal}, [src0:#128]
@ vmul.32x2Val, xVal, xVal, cutoffVal @ MAX(x, cutoff)
@ TODO: look at scheduling instructions differently
@ vmlaq.32 acc_val, xVal, cpaVal
@ vmlaq.32 acc_val, xVal, cpaVal
@ vadd.32 s8, s8, s9
@ vmov d8, d17 @ x8Val @ afterthought — needed a register to hold x’2
@ The following is copied with minimal edits from Doug’s asm for tail
src_val .req r9
c_accumulator .req d4 @ s8–s11
cutoff_vector .req d1 @ s2–s3
c_pvector .req d2 @ s4–s5
src_vector .req d3 @ s6–s7
c_qvector .req q3 @ d6–d7/s12–s15
x_qvector .req q8 @ d16–d17
c_pvector .req q9 @ d18–d19
x_to_1 .req d20 @ s40–s41
x_to_2 .req d21 @ s42–s43
x_to_3 .req d22 @ s44–s45
x_to_4 .req d23 @ s46–s47
x_low .req q12 @ d24–d25
x_high .req q13 @ d26–d27

loop2:
@cmp num_points, number
@bli .done

Listing 68. A NEON ASM implementation of 

asm/neon/volk_arm_32f_x2_dot_prod_32f_neonasm_opts.

Listing 69. A NEON ASM implementation of 

asm/neon/volk_arm_32f_x2_multiply_32f_neonasm_opts.