The ARM Scalable Vector Extension

Nigel Stephens
Stuart Biles
Matthias Boettcher
Jacob Eapen
Mbou Eyole
Giacomo Gabrielli
Matt Horsnell
Grigorios Magklis
Alejandro Martinez
Nathanael Premillieu
Alastair Reid
Alejandro Rico
Paul Walker
ARM

The ARM Scalable Vector Extension (SVE) allows implementations to choose a vector register length between 128 and 2,048 bits. It supports a vector-length-agnostic programming model that lets code run and scale automatically across all vector lengths without recompilation. Finally, it introduces several innovative features that begin to overcome some of the traditional barriers to autovectorization.

Architecture extensions are often somewhat conservative when they are first introduced, and they are then expanded as their potential becomes better understood and transistor budgets increase. Beginning in 2002, ARM has introduced and then expanded its support for single-instruction, multiple-data (SIMD) features. ARM began with integer-only SIMD operations in the 32-bit integer register file in ARMv6-A, and then introduced 64- and 128-bit SIMD operations sharing the floating-point register file in the Advanced SIMD extensions for ARMv7-A and ARMv8-A.[1,2]

These extensions efficiently target media and image processing workloads, which typically process structured data using well-conditioned DSP algorithms. However, as our partners continue to deploy ARMv8-A into new markets, we have seen an increasing demand for more radical enhancements to the ARM SIMD architecture, including the introduction of well-known technologies such as gather-load and scatter-store, per-lane predication, and longer vectors.

But this raises the question, what should that vector length be? The conclusion from more than a decade of research into vector processing, both within ARM[3,4] and inspired by more traditional vector architectures like the CRAY-1,[5] is that there is no single preferred vector length. For this reason, the ARM Scalable Vector Extension (SVE) leaves the vector length as an implementation choice (from 128 to 2,048 bits, in 128-bit increments). Importantly, the programming model adjusts dynamically to the available vector length, with no need to recompile high-level languages or rewrite hand-coded SVE assembly or compiler intrinsics. Of course, longer vectors are only part of the solution, and achieving significant speedup also requires high vector utilization. At a high level, several key SVE features enable improved autovectorization support. The scalable vector length increases parallelism while allowing implementation choice. Rich addressing modes enable nonlinear data accesses and efficient loop unrolling. Per-lane predication allows vectorization of loops containing complex control flow. Predicate-driven loop control and management reduces vectorization overhead relative to scalar code. A rich set of horizontal operations apply to more types of reducible loop-carried dependencies. Vector partitioning and software-managed speculation enable vectorization of loops with data-dependent exits. And, finally, scalarized intravector subloops permit vectorization of loops with more complex loop-carried dependencies.

In this article, we will introduce the SVE architecture and the vector-length-agnostic (VLA) programming model, discuss the implications for compilers and some of the implementation challenges, and present initial results from our evaluation.

SVE Overview

Here, we more fully describe the architectural state and key features introduced by SVE and illustrate these with code examples, where appropriate.

Architectural State

SVE introduces a new architectural state (see Figure 1a) that includes 32 new scalable vector registers (*Z*0 to *Z*31). Their width is implementation dependent within the aforementioned range. The new registers extend the 32 128-bit wide Advanced SIMD registers (*V*0 to *V*31) to provide scalable containers for 64-, 32-, 16-, and 8-bit data elements.

Alongside the scalable vector registers are 16 scalable predicate registers (*P*0 to *P*15) and a special-purpose first-fault register (FFR). Finally, a set of control registers (*ZCR_EL*1 to *ZCR_EL*3) gives privileged execution states the ability to virtualize (by reduction) the effective vector width.

Scalable Vector Length

Within a fixed 32-bit encoding space, it is not viable to create a different instruction set every time a different vector width is demanded. SVE radically departs from this approach in being vector-length agnostic, allowing each implementation to choose a vector length that is any multiple of 128 bits between 128 and 2,048 bits (the current architectural upper limit). Not having a fixed vector length lets SVE address multiple markets with implementations targeting different performance-power-area optimization points.

This novel aspect of SVE enables software to scale gracefully to different vector lengths without the need for additional instruction encodings, recompilation, or software porting effort. SVE provides the capabilities for software to be vector-length agnostic through the use of vector partitioning while also supporting more conventional SIMD coding styles that require fixed-length, multiple-of-*N*, or power-of-two subvectors.

Predicate-Centric Approach

Predication is central to SVE's design. In this section, we describe the predicate register file, its interaction with other architectural state, and how the use of predicates enables several advanced features.

Predicate registers. The predicate register file is organized and used as follows:

- *Sixteen scalable predicate registers* (*P*0 to *P*15). Control of general memory and arithmetic operations is restricted to *P*0 to *P*7, known as the *governing predicate*; however, predicate-generating instructions (such as vector comparisons) and instructions working solely on predicates can use the full set, *P*0 to *P*15 (see Figure 1a). This balance has been validated by analyzing compiled and hand-optimized codes, and it mitigates the predicate register pressure observed on other predicate-centric architectures.[6]
- *Mixed element size control.* Each predicate comprises eight bits per 64-bit vector element, allowing control down to byte granularity. For any given element, only the least-significant bit of the corresponding governing predicate element is used as the control. This is important for vectorizing code operating on multiple data types (see Figure 1b).
- *Predicate conditions.* Predicate-generating instructions (such as vector comparisons and predicate operations) in SVE reuse the AArch64 NZCV condition code flags, which in the context of predication are interpreted differently (see Table 1).
- *Implicit order.* Predicates are interpreted in an implicit least- to most-significant element order, corresponding to an equivalent sequential execution.

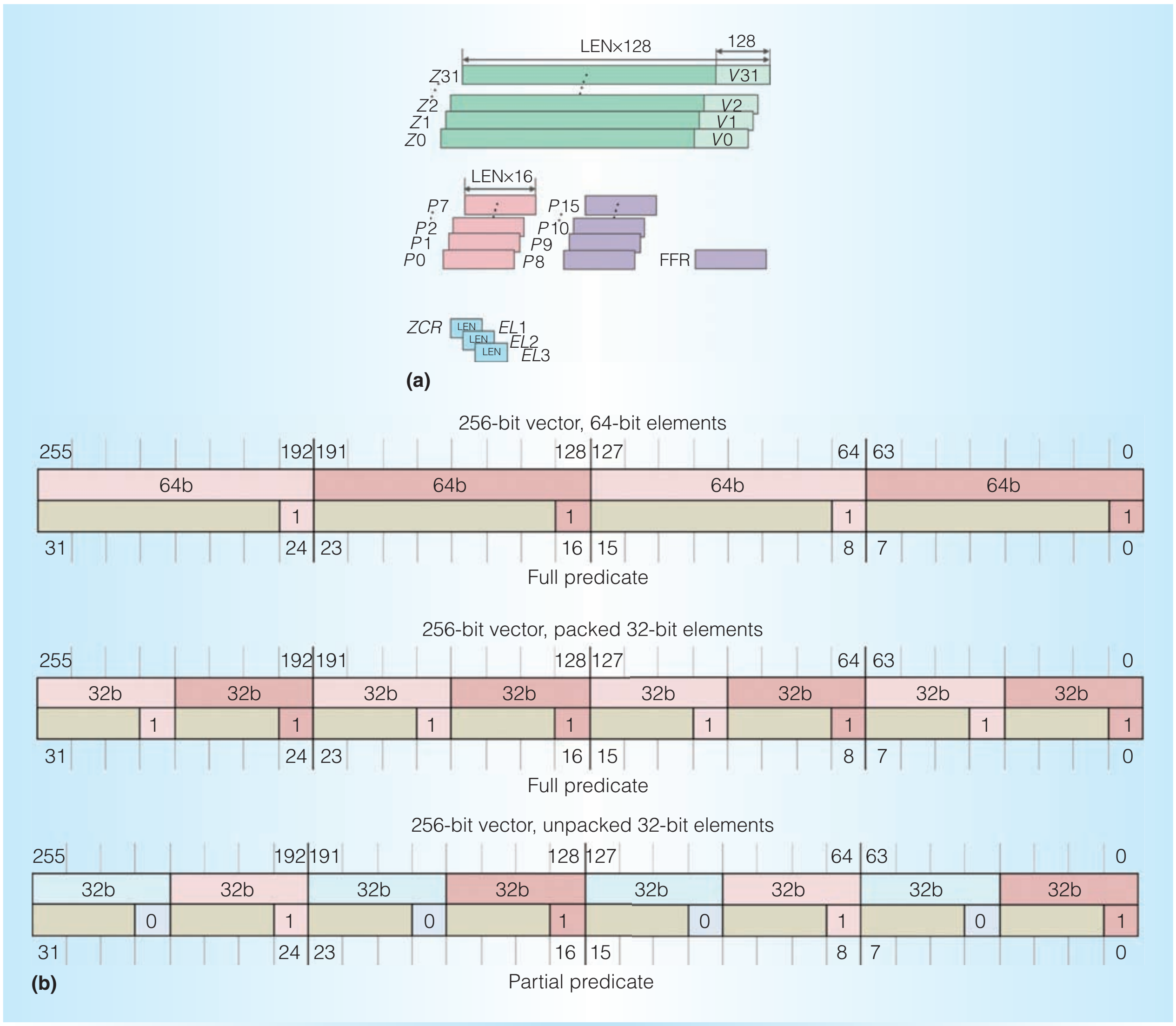


Figure 1. ARM Scalable Vector Extension (SVE) architectural state: vector registers (*Z*0 to *Z*31), predicate registers (*P*0 to *P*15), the first-fault register (FFR), and exception-level specific control registers (*ZCR_EL*1 to *ZCR_EL*3). (a) SVE registers. (b) SVE predicate organization.

Table 1. SVE condition flags overloading.

Condition flag	**SVE name**	**SVE description**
N	`First`	Set if first element is active
Z	`None`	Set if no element is active
C	`Last`	Set if last element is not active
V	`Term`	Scalarized loop state, else zero

We refer to the first and last predicate elements and associated conditions with respect to this order.

Predicate-driven loop control. Predication is used to drive vectorized loop control flow decisions in SVE. In other SIMD architectures that support predication, such as ICMI[7] and AVX-512,[8] generating the governing predicate for a loop often requires testing an induction variable. This is typically done by calculating the sequence of incrementing values in a vector register and then using that vector as input to a vector comparison.

There are two sources of overhead associated with this approach. First, a valuable vector register is wasted to store the sequence, and second, autovectorizing compilers tend to align all SIMD data in a loop to the largest element size, resulting in a potential loss of throughput when the size of the induction variable is greater than that of the data elements processed within the loop.

To overcome these limitations in the most common scenarios, SVE includes a family of `while` instructions that work with scalar count and limit registers to populate a predicate with the loop iteration controls that would have been calculated by the corresponding sequential execution of the loop. Note that if the loop counter is close to the maximum integer value, `while` will handle potential wrap-around behavior consistently with the semantic of the original sequential code. Similarly to other predicate-generating instructions, `while` also updates the condition flags.

Figure 2 demonstrates some of these concepts. It shows a unit-stride Daxpy (double-precision *Ax* plus *y*) loop in C, as ARMv8-A scalar assembly, and as ARMv8-A SVE assembly (for brevity, these examples are suboptimal). There is no overhead in instruction count for the SVE version in Figure 2c when compared to the equivalent scalar code in Figure 2b, which allows a compiler to opportunistically vectorize loops that have an unknown trip count.

The same example is illustrated in Figure 3, which steps through the SVE version of the code at both 128-bit and 256-bit vector lengths. The diagram shows the intermediate architectural state, for *P* predicate and *Z* vector registers, and the relative instructions required to process four array elements at two vector lengths. A full narrative for this example is beyond the scope of this article; see the SVE reference manual[2] and the VLA programming white paper[9] for more guidance on the instructions used.

```
void daxpy(double *x, double *y, double a, int n)
{
  for (int i = 0; i <n; i++) {
    y[i] = a*x[i] + y[i];
  }
}
```
(a)

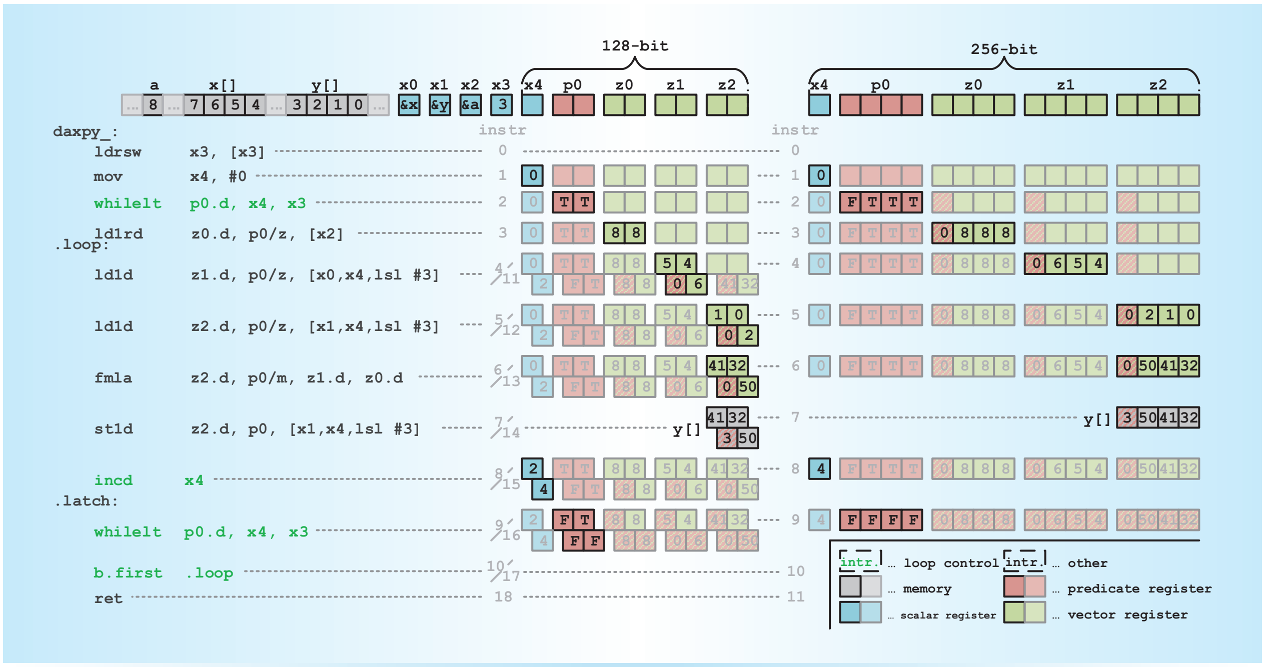
```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
daxpy_:
  ldrsw x3, [x3]                  // x3=*n
  mov x4, #0                      // x4=i=0
  ldr d0, [x2]                    // d0=*a
  b.latch
.loop:
  ldr d1, [x0, x4, lsl #3]        // d1=x[i]
  ldr d2, [x1, x4, lsl #3]        // d2=y[i]
  fmadd d2, d1, d0, d2            // d2+=x[i]*a
  str d2, [x1, x4, lsl #3]        // y[i]=d2
  add x4, x4, #1                  // i+=1
.latch:
  cmp x4, x3                      // i < n
  b.lt.loop                       // more to do?
  ret
```
(b)

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
daxpy_:
  ldrsw x3, [x3]                  // x3=*n
  mov x4, #0                      // x4=i=0
  whilelt p0.d, x4, x3            // p0=while(i++<n)
  ld1rd z0.d, p0/z, [x2]          // p0:z0=bcast(*a)
.loop:
  ld1d z1.d, p0/z, [x0, x4, lsl #3] // p0:z1=x[i]
  ld1d z2.d, p0/z, [x1, x4, lsl #3] // p0:z2=y[i]
  fmla z2.d, p0/m, z1.d, z0.d     // p0?z2+=x[i]*a
  st1d z2.d, p0, [x1, x4, lsl #3] // p0?y[i]=z2
  incd x4                         // i+=(VL/64)
.latch:
  whilelt p0.d, x4, x3            // p0=while(i++<n)
  b.first.loop                    // more to do?
  ret
```
(c)

Figure 2. Equivalent C, scalar, and SVE representations of the Daxpy kernel. (a) Daxpy C code. (b) Daxpy AArch64 scalar code. (c) Daxpy SVE code.

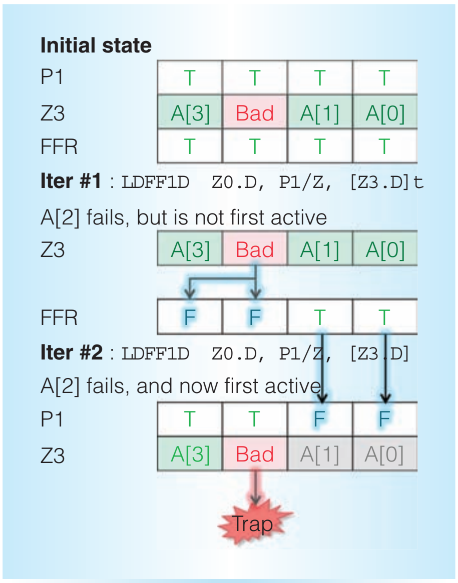
Speculative Vectorization

To vectorize loops with data-dependent termination conditions, software has to be able to perform some operations speculatively before the condition has been resolved. For some types of instructions, such as simple



```
daxpy_:
    ldrsw    x3, [x3]
    mov      x4, #0
    whilelt  p0.d, x4, x3
.loop:
    ld1rd    z0.d, p0/z, [x2]
    ld1d     z1.d, p0/z, [x0,x4,lsl #3]
    ld1d     z2.d, p0/z, [x1,x4,lsl #3]
    fmla     z2.d, p0/m, z1.d, z0.d
    st1d     z2.d, p0, [x1,x4,lsl #3]
    incd     x4
.latch:
    whilelt  p0.d, x4, x3
    b.first  .loop
    ret
```

Figure 3. Instruction-by-instruction example of Daxpy with $n = 3$ and hardware vector lengths of 128 and 256 bits. This illustrates how for the longer vector twice as many elements are processed in parallel by each SVE instruction, and therefore half as many iterations are performed.



Initial state
P1 T T T T
Z3 A[3] Bad A[1] A[0]
FFR T T T T
Iter #1 : LDFF1D Z0.D, P1/Z, [Z3.D]t
A[2] fails, but is not first active
Z3 A[3] Bad A[1] A[0]
FFR F F T T
Iter #2 : LDFF1D Z0.D, P1/Z, [Z3.D]
A[2] fails, and now first active
P1 T T F F
Z3 A[3] Bad A[1] A[0]
Trap

Figure 4. Example of speculative gather load controlled by the FFR. The first active element represents the current sequential execution of the loop that must succeed, while subsequent elements are speculative prefetches that will not cause a trap but instead return success or failure in the FFR register.

integer arithmetic, this is harmless, because there are no side effects. However, for instructions that could have side effects when operating on invalid data or addresses, it is necessary to have mechanisms to avoid those side effects.

In SVE, this is achieved with the introduction of a first-fault mechanism for vector load instructions. This mechanism suppresses memory faults if they do not result from the first active element in the vector. (We use "active element" to refer to an element within a vector for which the corresponding governing predicate bit is set *true*.) Instead, the mechanism updates a predicate value in the FFR to indicate which elements were not successfully loaded following a memory fault.

Figure 4 shows an example with a gather load that speculatively loads from addresses held in register *Z3*. In the first iteration, FFR is initialized to all *true*. The translations of *A*[0] and *A*[1] succeed, but the address *A*[2] is invalid (for example, unmapped), and it fails without taking a trap. Instead, positions corresponding to *A*[2] and *A*[3] in the FFR are set to *false*. For the second iteration, the positions corresponding to *A*[0] and *A*[1] in the instruction predicate register *P*1 will be set to *false* and FFR to *all-true* again. In this case, *A*[2] fails again—but, since it is now the first active element, it causes a trap to the OS to service the fault or terminate the program if it was an illegal access.

Figure 5 shows how speculative vectorization with the first-fault mechanism allows vectorization of the *strlen* function. The `ldff1b` instruction loads the characters in *s* and sets the FFR elements starting from the first faulty address to *false,* so only the successfully loaded elements remain set to *true* in FFR. The FFR predicate value is transferred to *P*1, which predicates the subsequent instructions that check for the end-of-string character. Following a memory fault, the next loop iteration will retry the faulty access, but now as the first active element, it will cause a trap to the operating system.

This flexible mechanism allows software-managed speculative vectorization of many loops with data-dependent terminations that could not be vectorized safely otherwise.

```
int strlen(const char *s) {
  const char *e = s;
  while (*e) e++;
  return e -s;
}
```
(a)

```
// x0 = s
strlen
  mov x1, x0                          // e=s
.loop:
  ldrb x2, [x1], #1                   // x2=*e++
  cbnz x2, .loop                      // while(*e)
.done:
  sub x0, x1, x0                      // e-s
  sub x0, x0, #1                      // return e-s-1
  ret
```
(b)

```
// x0 = s
strlen:
  mov x1, x0                          // e=s
  ptrue p0.b                          // p0=true
.loop:
  setffr                              // ffr=true
  ldff1b z0.b, p0/z, [x1]             // p0:z0=ldff(e)
  rdffr p1.b, p0/z                    // p0:p1=ffr
  cmpeq p2.b, p1/z, z0.b, #0          // p1:p2=(*e==0)
  brkbs p2.b, p1/z, p2.b              // p1:p2=until(*e==0)
  incp x1, p2.b                       // e+=popcnt(p2)
  b.last .loop                        // last=>!break
  sub x0, x1, x0                      // return e-s
  ret
```
(c)

Figure 5. Equivalent C, scalar, and SVE representations of unoptimized *strlen* function. (a) C code *strlen*. (b) AArch64 scalar *strlen*. (c) SVE *strlen*.

Dynamic Exits

We just showed an example of how SVE vectorizes a loop without an explicit iteration count. The technique that has been used is called *vector partitioning,* and it consists of operating on a partition of safe elements in response to dynamic loop conditions.

Partitions are implemented using predicate manipulating instructions and are inherited by nested conditions and loops. In this way, vector partitioning is a natural way to deal with uncounted loops with data-dependent exits (for example, *do-while* or *break*). Vectorized code must guarantee that operations with side effects that follow a loop *break* in the original sequential execution of the loop must not be architecturally performed. This is achieved by generating a before-break vector partition, operating on just that partition, and then exiting the loop if the break condition was detected within the vector.

Figure 5 shows how vector partitioning is used to vectorize the *strlen* function. We described how the `ldff1b` instruction loads values speculatively. The `rdffr` instruction reports the partition of safely loaded values. The `cmpeq` instruction uses this to compare only the safe (valid) values with zero. Furthermore, the `brkbs` instruction then generates a subpartition bounded by the loop break condition and sets the `last` (*C*) condition flag accordingly.

Horizontal Operations

Another challenge for SIMD vectorization is the presence of dependencies between loop iterations within the same vector, particularly when the vector length is not known in advance. In many cases, these dependencies can be resolved by use of a horizontal vector reduction operation. Unlike normal SIMD

```
struct {uint64val; structnode *next} *p;
uint64 res = 0;
for (p = &head; p != NULL; p = p->next)
    res ^= p->val;
```
(a)

```
for (p = &head; p != NULL; ) {
    for (i = 0; p != NULL && i < VL/64; p = p->next)
        p'[i++] = p;            // collect up to VL/64 pointers
    for (j = 0; j < i; j++)
        res ^= p'[j]->val;      // gather from pointer vector
}
```
(b)

```
// P0 = current partition mask
    dup z0.d, #0                // res'= 0
    adr x1, head                // p = &head
loop:
    // serialized sub-loop under P0
    pfalse p1.d                 // first i
inner:
    pnext p1.d, p0, p1.d        // next i in P0
    cpy z1.d, p1/m, x1          // p'[i]=p
    ldr x1, [x1, #8]            // p=p->next
    ctermeq x1, xzr             // p==NULL?
    b.tcont inner               // !(term|last)
    brka p2.b, p0/z, p1.b       // P2[0..i] = T
                                // vectorized main loop under P2
    ld1d z2.d, p2/z, [z1.d, #0] // val'=p ->val
    eor z0.d, p2/m, z0.d, z2.d  // res'^=val'
    cbnz x1, loop               // while p!=NULL
    eorv d0, p0, z0.d           // d0=eor(res')
    umov x0, d0                 // return d0
    ret
```
(c)

Figure 6. Separation of loop-carried dependencies for partial vectorization of linked lists. (a) Linked-list loop-carried dependency. (b) Split loop: serial pointer chase and vectorizable loop. (c) Split-loop ARMv8 SVE code.

instructions, horizontal operations are a special class of instructions that perform operations horizontally across elements of the same vector register. SVE provides a very rich set of horizontal operations, including bitwise logical, integer, and floating-point reductions.

Scalarized Intravector Subloops

More complex loop-carried dependencies are a significant barrier to vectorization that SVE can also help to address. One approach to overcoming these is to use loop fission to split a loop into an explicitly serial part, allowing the rest of the loop to be profitably vectorized. However, in many cases, the cost of unpacking and packing the vectors to work on them serially negates any performance benefit from vectorization. To reduce this cost, SVE provides support for serially processing elements in place within a vector partition.

An example of this problem is traversing a linked list, because there is a loop-carried dependency between each iteration (see Figure 6a). By applying loop fission, the loop is split into a serial pointer chase followed by a vectorizable loop (see Figure 6b).

Figure 6c shows how SVE vectorizes this code. The first part is the serialized pointer chase. The `pnext` instruction allows operation on active elements one by one by setting *P*1 to the next active element and computing the `last` condition. The `cpy` instruction inserts scalar register *X*1 into the vector register *Z*1 at this position. Then, the `ctermeq` instruction detects the end of the list (`p == NULL`) or the end of the vector (by testing the `last` condition set by the `pnext` instruction). The `b.tcont` branch checks this and continues with the serial loop if more pointers are available and needed.

The partition for the loaded pointers is computed into *P*2. In this case, the vectorized loop just performs an exclusive-or operation. Finally, all the vector elements in *Z*0 are combined with a horizontal exclusive-or reduction using the `eorv` instruction. In this example, the performance gained might be insufficient to justify using vectorization of this loop, but it serves to illustrate a principle applicable to more profitable scenarios.

Compiling for SVE

We have illustrated many aspects of programming with SVE. Translating these techniques into a compiler required us to rethink our compilation strategy because of the impact of wide vectors, vector-length agnosticism, predication, and speculative vectorization with first-fault loads.

Wide Vectors and Vector-Length Agnosticism

When compiling for fixed-length vectors such as Advanced SIMD, one approach is "unroll and jam," in which a loop is first unrolled by the number of elements in a vector and then corresponding operations from each iteration are merged together into vector operations. This approach is clearly incompatible with a scalable vector length because the length is unknown at compile time. The solution is for the vectorizer to directly map scalar operations to corresponding vector operations. A second challenge is that knowledge of the constant vector length, *VL*, is often critical to vectorization. For example, when handling induction variables, a compiler might initialize a vector with the values [0, 1, …, *VL* – 1] by loading it from memory and then incrementing the vector by *VL* on each iteration. SVE addresses this with a family of instructions in which the current vector length is an implicit operand—for example, the `index` instruction initializes a vector induction variable and the `inc` instructions advance an induction variable based on the current vector length and specified element size.

Vector-length agnosticism also impacts register reads and writes to the stack, such as spilling and filling due to register pressure or passing a vector argument as part of the function-calling convention. Supporting this inside an existing compiler is challenging because compilers normally assume that all objects are at a constant offset in the stack frame, which will not be the case for variable length vector registers. Our solution is to introduce stack regions. For existing stack regions, offsets remain fixed (for example, constant byte offsets) but regions containing SVE registers are dynamically allocated, and the offsets for vector register load/store instructions encode a "stack slot number" that is automatically multiplied by the vector length in bytes.

Predication

Predicates are introduced by a conventional "if conversion" pass that replaces if-statements with instructions to calculate predicates and then uses the appropriate predicates on each operation dominated by the condition. We extend this approach to handle conditional branches out of a loop, by inserting a `brk` instruction that generates a vector partition in which those lanes following a loop exit condition become inactive.

Floating Point

Floating point can be a challenge for compiler vectorization because vectorizing a loop may change the order of floating-point operations, which could give a different result from the original scalar code. Programmers then have to choose whether they want consistent results by disabling vectorization or whether they can tolerate some variation to achieve better performance.

Vector-length agnosticism introduces even more variation because a different vector length could cause a different ordering, and therefore a different result. SVE mitigates this by providing a strictly ordered `fadda` reduction, which lets a compiler vectorize loops in which the original sequential order of floating-point additions must be maintained for correctness.

Speculative Vectorization

We implemented all of changes discussed as extensions of the LLVM compiler's existing vectorization pass,[10] but it was not feasible to support speculative vectorization within that structure. We implemented speculative vectorization in a separate pass that focused on expanding loop coverage rather than generating the highest-quality code.

The new vectorizer works in largely the same way as LLVM's current vectorizer, but it has more advanced predicate handling to support loops with multiple exits. It splits the loop body into multiple regions, each under the control of a different predicate. Broadly speaking, these regions represent operations that are always safe to execute, instructions that are required to calculate a conditional exit predicate, and instructions that occur after the conditional exit. For the latter two regions, we use first-fault loads and partitioning operations.

Design Challenges

Encoding space is a scarce resource for a fixed-width instruction set such as A64, so an early design constraint for SVE was to limit its overall encoding footprint in order to retain space for future expansion of the instruction set. To achieve that, ARM adopted various solutions in different areas of the ISA to keep the encoding space used by SVE below 28 bits (see Figure 7).

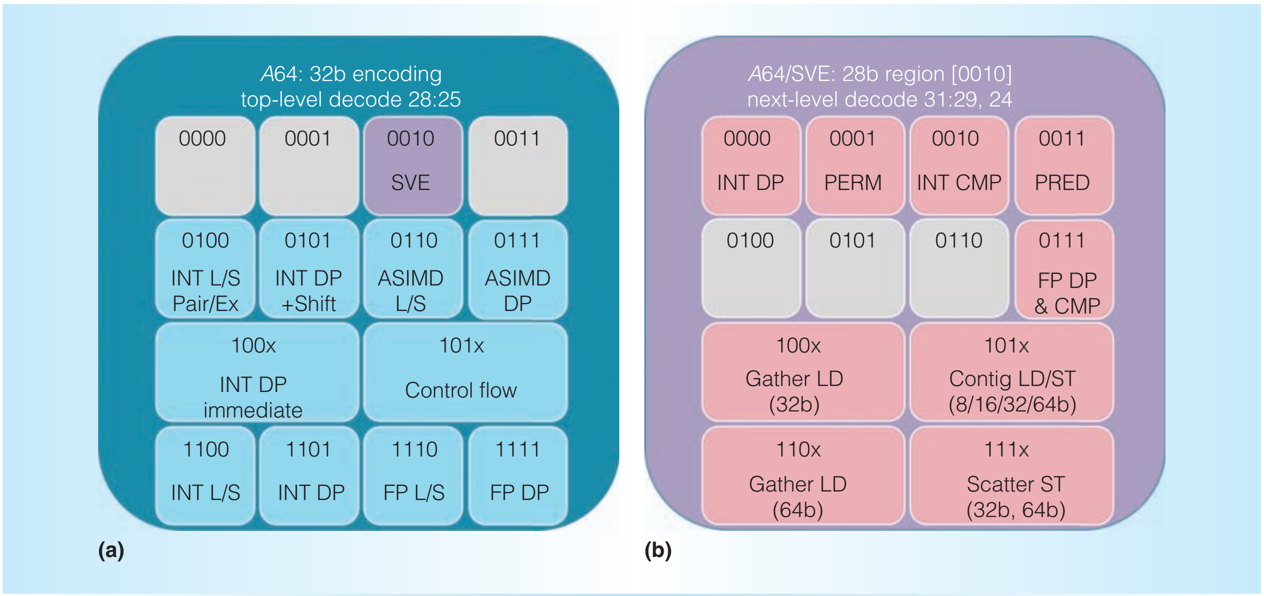


Figure 7. SVE encoding footprint. (a) *A*64 top-level encoding structure, with SVE occupying a single 28-bit region. (b) SVE encoding structure. Some room for future expansion is left in this region.

Constructive versus Destructive Forms

Although compilers prefer to have constructive forms of instructions (that is, those with a destination operand distinct from the source operands), the encoding space required to provide both predication and constructivity for the entire set of data-processing operations would easily exceed the maximum encoding budget. (For example, three vector registers and a governing predicate register would require 19 bits alone, without accounting for other control fields.) The tradeoff that SVE makes is to provide only destructive predicated forms of most data processing and provide constructive unpredicated forms of only the most essential opcodes, based on compiler studies.

Move Prefix

To fulfill the occasional need for fully constructive and predicated instructions, SVE introduces a `movprfx` instruction, which is easy for hardware to decode and combine with the immediately following instruction to create a single constructive operation. However, it can also be implemented as a discrete *vector move* instruction, and the result of executing the pair of instructions with or without combining them is identical. SVE supports both predicated and unpredicated forms of `movprfx`.

Restricted Access to Predicate Registers

As we mentioned earlier, to further reduce the encoding space, we restrict predicated load, store, and data-processing instructions to using a governing predicate register $P0$ to $P7$, whereas predicate-generating instructions can typically access all 16 predicate registers.

Hardware Implementation Cost

Aside from encoding space, another important consideration for SVE was the additional hardware cost required to support the new functionality above and beyond that of Advanced SIMD, particularly for smaller CPUs. A key decision was to overlay the SVE vector register file on the existing Advanced SIMD and floating-point register file (see Figure 1a), thus minimizing the area overhead for shorter vector lengths.

Existing Advanced SIMD and floating-point instructions are required to zero the higher-numbered bits of any vector register that they write, avoiding partial updates, which are notoriously hard to handle in high-performance microarchitectures. In addition, the vast majority of SVE operations can be mapped efficiently onto an existing Advanced SIMD datapath and functional units, with only the necessary modifications to support

Table 2. Model configuration parameters.

Model parameter	Configuration
L1 instruction cache	64-Kbyte, 4-way set associative, 64-byte line
L1 data cache	64-Kbyte, 4-way set associative, 64-byte line, 12-entry MSHR
L2 cache	256-Kbyte, 8-way set associative, 64-byte line
Decode width	4 instructions/cycle
Retire width	4 instructions/cycle
Reorder buffer	128 entries
Integer execution	2 × 24 entries scheduler (symmetric ALUs)
Vector/FP execution	2 × 24 entries scheduler (symmetric FUs)
Load/store execution	2 × 24 entries scheduler (2 loads/1 store)

predication and a larger vector width, if required.

Although a wider datapath for vector processing is a requirement for exploiting the data-level parallelism exhibited by several workloads in the HPC domain, this must be coupled with a corresponding improvement in memory access capabilities and bandwidth. SVE includes contiguous vector load-and-store instructions that support a rich set of addressing modes and data types. Load-and-replicate instructions can broadcast a single element from memory across a vector, exploiting existing load datapaths to remove the need for explicit permute operations in many cases.

SVE also includes gather-load and scatter-store instructions, which are an enabling feature that permit vectorization of loops that access noncontiguous data and would otherwise be unprofitable to vectorize. These instructions can benefit from advanced vector load/store units that perform multiple accesses in parallel,[3] but it is also acceptable to use a more conservative approach that cracks them into individual micro-operations, as long as this is not noticeably slower than performing the equivalent sequence of scalar loads or stores.

SVE Performance

We expect the SVE architecture to be implemented by multiple ARM partners on a variety of microarchitectures. Thus, for our evaluation of SVE, we used several representative microarchitecture models. For the results presented here, we chose a single model of a typical, medium-sized, out-of-order microprocessor that does not correspond to any real design, but which we believe gives a fair estimate of the performance to expect from SVE. Table 2 shows this model's main parameters.

Instruction execution and register file access latencies in the model are set to correspond to RTL synthesis experiments. For operations that cross lanes (that is, vector permutes and reductions), the model takes a penalty proportional to *VL*. The cache in the model is a true dual-ported cache with the maximum access size being the full cache line of 512 bits. Accesses crossing cache lines take an associated penalty.

Our evaluation uses an experimental compiler that can autovectorize code for SVE. We chose a set of high-performance computing applications from various well-known benchmark suites.[11–18] At present, our compiler supports only C and C++, so the choice of benchmarks is restricted to these languages. We use the original code from the publicly available versions of the benchmarks with minor modifications in a few cases to help guide the autovectorizer (for example, adding `restrict` qualifiers or OpenMP `simd` pragmas).

Figure 8 shows the results of our evaluation. We compared Advanced SIMD with three configurations of SVE with 128-, 256-, and 512-bit vector lengths, respectively. All four simulations used the same binary executable program and processor configuration, but the vector length was varied.

One immediate observation from the results is that SVE achieves higher vector utilization than Advanced SIMD because the features we

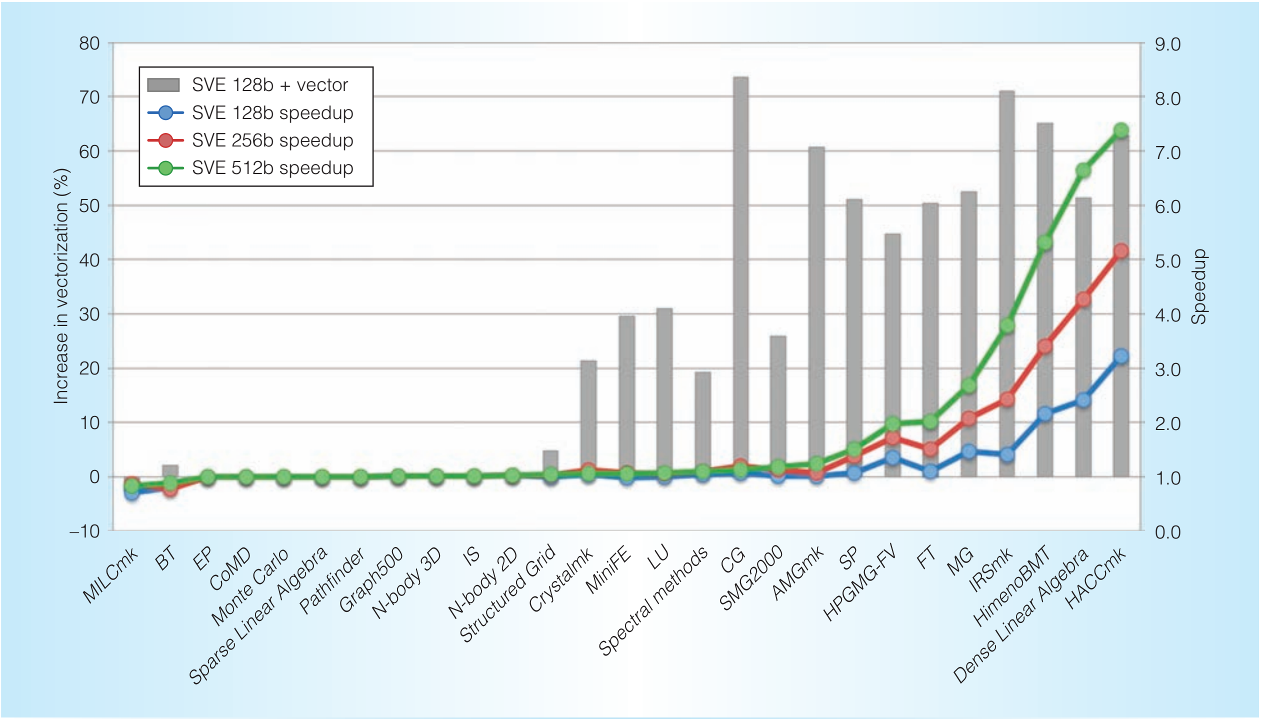


Figure 8. Performance of SVE at three vector lengths relative to Advanced SIMD. The bars show the increase in the percentage of dynamically executed vector instructions for SVE at the same 128-bit vector length, and the lines show the measured speedup at the three vector lengths.

introduced to the architecture allow the compiler to vectorize code with complex control flow and noncontiguous memory accesses. For this reason, SVE can achieve speedups of up to 3 times even when the vectors are the same length as Advanced SIMD. For example, in the case of *HACCmk,* the main loop has two conditional assignments that inhibit vectorization for Advanced SIMD, but the code is trivially vectorized for SVE.

The figure also demonstrates the benefits of vector-length agnostic code. We can clearly see how performance scales, simply by running the same executable on implementations of SVE with larger vectors. This is one of the most important benefits of SVE.

There are three clearly identifiable categories of benchmarks. On the right of Figure 8, we see a group of benchmarks that show much higher vectorization with SVE and performance that scales well with the vector length (up to seven times). Some of these benchmarks do not scale as well as others, and this is mainly because of the use of gather-scatter operations. Although these instructions enable vectorization, our assumed implementation conservatively cracks the operations, and so does not scale with vector length. In other cases, such as in *HimenoBMT,* the reason for poor scaling is bad instruction scheduling by the compiler.

On the left of Figure 8, we can see a group of benchmarks for which there is minimal, or in some cases zero, vector utilization for both Advanced SIMD and SVE. Our investigations show that this is because of how the code is structured or the compiler's limitations rather than a shortcoming of the architecture. For example, we know that by restructuring the code in *CoMD* we can achieve significant improvement in vectorization and execution time. Also, we note that the toolchain used for these experiments did not have vectorized versions of some basic math library functions such as `pow()` and `log()`, which inhibited vectorization of some loops (such as in *EP*). Finally, there are cases in which the algorithm itself is not vectorizable, such as in Graph500, where the program mostly traverses graph structures following pointers. We do not expect SVE to help here unless the algorithm is refactored with vectorization in mind.

The third group of benchmarks includes cases where the compiler has vectorized significantly more code for SVE than for Advanced SIMD, but we do not see much performance uplift. All these cases are due to code-generation issues with the compiler that we are currently addressing. In SMG2000, for example, a combination of bad instruction selection compounded by extensive use of gather loads results in very small benefit for SVE. It is worth noting here that the Advanced SIMD compiler cannot vectorize the code at all.

MILCmk is another interesting case, in which a series of poor compiler decisions contributes to performance loss for SVE compared to Advanced SIMD. In this case, the compiler decided to vectorize the outermost loop in a loop nest, generating unnecessary overheads (the Advanced SIMD compiler vectorizes the inner loop), and did not recognize some trivially vectorizable loops as such.

We expect that over time, with improvements to compilers and libraries, many of these issues will be resolved.

SVE opens a new chapter for the ARM architecture in terms of the scale and opportunity for increasing levels of vector processing on ARM-based processors. It is early days for SVE tools and software, and it will take time for SVE compilers and the rest of the SVE software ecosystem to mature. HPC is the current focus and catalyst for this compiler work, and it creates development momentum in areas such as Linux distributions and optimized libraries for SVE, as well as in tools and software from ARM and third parties.

We are already engaging with key members of the ARM partnership, and are now broadening that engagement across the open-source community and wider ARM ecosystem to support development of SVE and the HPC market, enabling a path to efficient exascale computing. MICRO

Acknowledgments

SVE has been a multiteam, multisite, and multiyear effort across ARM's research and product groups. We thank everyone involved in the development of SVE and those who helped review this article.

References

1. *ARM Architecture Reference Manual* (ARMv6 edition), ARM, 2005.
2. "A-Profile Architecture Specifications," *ARM Developer*, http://developer.arm.com/products/architecture/a-profile/docs.
3. M. Woh et al., "From SODA to Scotch: The Evolution of a Wireless Baseband Processor," *Proc. 41st Ann. IEEE/ACM Int'l Symp. Microarchitecture*, 2008, pp. 152–163.
4. M. Boettcher et al., "Advanced SIMD: Extending the Reach of Contemporary SIMD Architectures," *Proc. Conf. Design, Automation & Test in Europe*, 2014, pp. 24:1–24:4.
5. R.M. Russell, "The CRAY-1 Computer System," *Comm. ACM*, vol. 21, Jan. 1978, pp. 63–72.
6. S.S. Baghsorkhi, N. Vasudevan, and Y. Wu, "Flexvec: Autovectorization for Irregular Loops," *Proc. 37th ACM SIGPLAN Conf. Programming Language Design and Implementation*, 2016, pp. 697–710.
7. L. Seiler et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," *IEEE Micro*, vol. 29, no. 1, 2009, pp. 10–21.
8. A. Sodani, "Knights Landing (KNL): 2nd Generation Intel Xeon Phi Processor," *Proc. Hot Chips 27 Symp.*, 2015, pp. 1–24.
9. *A Sneak Peek into SVE and VLA Programming*, white paper, ARM, 2016.
10. C. Lattner and V. Adve, "The LLVM Compiler Framework and Infrastructure Tutorial," *Proc. Int'l Workshop on Languages and Compilers for Parallel Computing*, 2004, pp. 15–16.
11. "NAS Parallel Benchmarks 3.0 OpenMP C version," Nov. 2014; http://github.com/benchmark-subsetting/NPB3.0-omp-C.
12. "TORCH (Testbed for Optimization Research)," Sept. 2016; http://crd.lbl.gov/departments/computer-science/PAR/research/previous-projects/torch-testbed.
13. "The ASCI Purple Benchmark Codes," Aug. 2003; http://asc.llnl.gov/computing_resources/purple/archive/benchmarks.

14. "ASC Sequoia Benchmark Codes," June 2013; http://asc.llnl.gov/sequoia/benchmarks.
15. "Himeno Benchmark"; http://accc.riken.jp/en/supercom/himenobmt.
16. "HPGMG: High-Performance Geometric Multigrid," Jan. 2017; http://bitbucket.org/hpgmg/hpgmg.
17. "Mantevo Project," 2017; http://mantevo.org.
18. "CORAL Benchmark Codes," Feb. 2014; http://asc.llnl.gov/CORAL-benchmarks.

Nigel Stephens is a Fellow and lead ISA architect at ARM, where he led the development of the ARMv8-A Scalable Vector Extension (SVE). Stephens received a BSc in computer science from University College, London. Contact him at nigel.stephens@arm.com.

Stuart Biles is a Fellow and director of research architecture at ARM. His research interests include security, processor performance, heterogeneous computing, and academic enablement. Biles received a BEng in electronic engineering from the University of Southampton. He is a member of IEEE. Contact him at stuart.biles@arm.com.

Matthias Boettcher is a senior research engineer at ARM. His research interests include secure architectures and systems and functional safety in high-performance systems. Boettcher received an MSc in information and communication technologies from the University of Applied Sciences Zittau/Goerlitz and an MSc in microelectronic system design from the University of Southampton. Contact him at matthias.boettcher@arm.com.

Jacob Eapen is a senior CPU modeling engineer with the Architecture and Technology group at ARM. His research includes development and benchmarking of new CPU architecture concepts for ARM's next-generation processors. Eapen received a BTech in instrumentation from Cochin University of Science and Technology, India. Contact him at jacob.eapen@arm.com.

Mbou Eyole is a staff research engineer in ARM's research division. His research interests include energy-efficient architectures. Eyole received a PhD in sentient computing from the University of Cambridge. He is a Chartered Engineer. Contact him at mbou.eyole@arm.com.

Giacomo Gabrielli is a staff research engineer at ARM. His research interests include vector/SIMD instruction sets, microarchitectures, digital signal processing, and performance modeling. Gabrielli received a PhD in computer engineering from the University of Pisa. He is a professional member of ACM. Contact him at giacomo.gabrielli@arm.com.

Matt Horsnell is a principal research engineer at ARM, where he leads the Architecture Research group. His research interests include scalable parallel systems, synchronization, and vector architectures. Horsnell received a PhD in computer science from the University of Manchester. Contact him at matt.horsnell@arm.com.

Grigorios Magklis is a principal processor architect at ARM. His research interests include scalable parallel systems, vector architectures, and instruction set design. Magklis received a PhD in computer science from the University of Rochester. Contact him at grigorios.magklis@arm.com.

Alejandro Martinez is a staff engineer in the Architecture and Technology Department at ARM, where he works on developing future ARM architectures. Martinez received a PhD in computer science from the University of Castilla-La Mancha, Spain. Contact him at alejandro.martinezvicente@arm.com.

Nathanael Premillieu is a senior engineer with the CPU modeling team at ARM. His research interests include computer architecture and microarchitecture for performance. Premillieu received a PhD in computer science from the University of Rennes 1, France. Contact him at nathanael.premillieu@arm.com.

Alastair Reid is a principal research engineer at ARM. His research interests include security of IoT systems and creating, checking, and using formal specifications of computer architecture. Reid received an MSc in formal methods from the University of Glasgow. Contact him at alastair.reid@arm.com.

Alejandro Rico is a senior research engineer at ARM. His research interests include high-performance computing, vector processing, multicore scalability, and heterogeneous architectures. Rico received a PhD in computer architecture from the Universitat Politecnica de Catalunya. He is a member of ACM and IEEE. Contact him at alejandro.rico@arm.com.

Paul Walker is a principal engineer at ARM. His research interests include advanced compiler design, particularly analysis, debugging, and optimization within HPC environments. Walker received an MEng in microelectronic systems engineering from the University of Manchester Institute of Science and Technology. Contact him at paul.walker@arm.com.