ARM

White Paper

A sneak peek into SVE and VLA programming

Francesco Petrogalli

February 2020

Contents

Introduction

The *Scalable Vector Extension (SVE)* is an extension of the *ARMv8-A A64* instruction set, recently announced by ARM. Following the announcement at Hot Chips 28[1], a few articles describing what people think SVE is have appeared on the Internet. It is now our turn to give a more detailed description of some of the key features that the ISA provides.

In this whitepaper I will introduce you to some of the new architectural features included in SVE, providing code examples that will show how to use them.

SVE is *not* an extension of NEON[2], but a new set of vector instructions developed to target HPC workloads. Some of the examples will show how SVE enables vectorization of loops which for NEON would either be impossible or not beneficial to vectorize. A prior knowledge of NEON is recommended, but not required as I will explain the examples along the way.

This white paper is divided into two parts. The first one is an overview of SVE, that covers the new registers, the new instructions, the Vector Length Agnostic (VLA) programming technique, and some examples of vector code showing VLA in action. In the second part I will show how SVE can be used to vectorize `strcmp` and `strcpy` from the standard C library.

As a final note, this is not a complete list of the features that SVE provides. A full architectural specification of SVE with detailed instruction descriptions and other resources on SVE are available on developer.arm.com[3].

[1] http://www.hotchips.org
[2] https://developer.arm.com/technologies/neon
[3] https://developer.arm.com/products/software-development-tools/hpc/documentation/introducing-scalable-vector-extension-sve

1 SVE: an overview

New architectural registers

Let's start by having a look at the architectural registers introduced by SVE.

The instruction set operates on a new set of vector and predicate registers:

- 32 Z registers, z0, z1, ..., z31;
- 16 P registers, p0, p1, ..., p15;
- 1 *FFR* register.

The Z registers are data registers. The architecture specifies that their size in bits must be a *multiple of 128*, from a *minimum of 128 bits* to an implementation-defined maximum of up to 2048 bits. Data in these registers can be interpreted as 8-bit bytes, 16-bit halfwords, 32-bit words or 64-bit doublewords. For example, a 384-bit implementation of SVE can hold 48 bytes, 24 halfwords, 12 words, or 6 doublewords of data. It is also important to mention that the low 128 bits of each Z register overlap the corresponding *NEON* registers of the *Advanced SIMD* extension and therefore also the scalar floating-point registers - see figure figure 1.1.

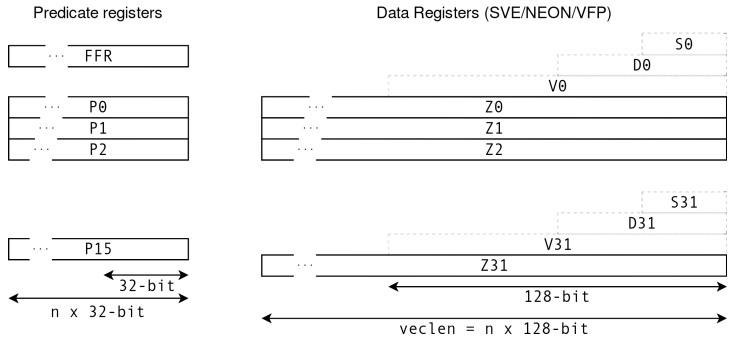

Figure 1.1: Register overlapping.

The P registers are *predicate* registers, which are unique to SVE, and hold one bit for each byte available in a Z register. For example, an implementation providing 1024-bit Z registers provides 128-bit predicate registers.

The FFR register is a *special* predicate register that differs from regular predicate registers by way of being used implicitly by some dedicated instructions, called *first faulting loads*.

Individual predicate bits encode a boolean true or false, but a predicate *lane*, which contains between one and eight predicate bits is either *active* or *inactive*, depending on the value of its least significant bit.

Similarly, in this document the terms *active* or *inactive* lane will be used qualify the lanes of data registers under the control of a predicate register.

Instruction families

SVE introduces a variety of instructions that operate on the data and predicate registers. I will present now a rough classification of the instructions that are available in SVE. I will describe some of them in detail in the subsequent sections.

The first dichotomy separates *predicated* instructions from *unpredicated* instructions, that is instructions that use a predicate register to control the lanes they operate on, versus those that do not have predication. In a predicated instruction, only the active lanes of vector operands are processed and can generate side effects - such as memory accesses and faults, or numeric exceptions.

Across these two main classes, one can find *data processing* instructions, that operate on Z registers (e.g. addition), *predicate generation* instructions, such as numeric comparisons that operate on data registers and produce predicate registers, or *predicate manipulation* instructions, that mostly cover predicate generation or logical operations on predicates.

Regarding the data manipulation instructions, most of the operations cover both floating point (FP) and integer domains, with some notable FP functionality brought by the *ordered horizontal reductions*, which provide cross-lane operations that preserve the strict C/C++ rules on non-associativity of floating-point operations.

Other interesting aspects of FP in SVE are the instructions to accelerate the calculation of vector `sin()`, `cos()` and `exp()`, and the ability to perform a four-operand fused multiply-accumulate.

A big chunk of the new instruction set is dedicated to the *vector load/store* instructions, which can perform *signed* or *unsigned extension* or *truncation* of the data, and that come with a wide range of new addressing modes that improve the efficiency of SVE code.

Vector Length Agnostic (VLA) programming

Unlike traditional SIMD architectures, which define a fixed size for their vector registers, SVE only specifies a maximum size. This freedom of choice is done to enable different ARM architectural licensees to develop their own implementation, targeting specific workloads and technologies which could benefit from a particular vector length.

A key goal of SVE is to allow the same program image to be run on any implementation of the architecture (yes, which might implement different vector lengths!), so it includes instructions which permit vector code to **adapt automatically** to the current vector length at runtime.

These features require a new programming style, called *Vector Length Agnostic (VLA) programming*.

Suppose you want to vectorize the loop inside the function `example01` of listing 1.1. With a traditional SIMD architecture, the user (or the compiler) knows how many elements the vector loop can process in one iteration. For example, the vectorized version of `example01` can be rewritten as `example_01_neon` in listing 1.2, using ARM NEON[4] (C intrinsics[5]). Here the loop operates on four elements, i.e. as many 32-bit `ints` as a NEON vector register can hold. Moreover, as in the case of other traditional unpredicated SIMD architectures, the programmer (or the compiler) needs to add an additional loop, called *loop tail*, that is responsible for processing those iterations at the end of the loop that do not fit in a full vector length.

Listing 1.1 Simple C loop processing integers.

```
1 void example01(int *restrict a, const int *b, const int *c, long N)
2 {
3   long i;
4   for (i = 0; i < N; ++i)
5     a[i] = b[i] + c[i];
6 }
```

With SVE the *fixed-width* approach is not appropriate. In listing 1.3 an assembly version of `example01` is shown, with SVE instructions. The assembly code is equivalent to the pseudo-C code presented in listing 1.5, where the `loop_body` section is repeated so long as the condition `cond` is true. The condition is tested on the predicate register `p0` that is created using the `whilelt` instruction. Let's see in detail how it works.

In the assembly code, `x3` corresponds to the value of the loop induction variable `i` and `x4` is the loop bound variable `N`. The assembly line `whilelt p0.s, x3, x4` is filling the predicate register `p0` by setting each lane as `p0.s[idx] := (x3 + idx) < x4` (`x3` and `x4` hold `i` and `N` respectively), for each of the indexes `idx` corresponding to 32-bit lanes of a vector register. For example, listing 1.4 shows the content of `p0` that `whilelt` generates in case of a 256-bit SVE implementation for `N`=7.

In building the predicate `p0` the `whilelt` also sets the condition flags. The branching instruction `b.first` following `whilelt` reads those flags and decides whether or not to branch to the `loop_body` label. In this specific case, `b.first`

[4] https://developer.arm.com/technologies/neon
[5] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ihi0073-/index.html

Listing 1.2 Vectorized version of listing 1.1, using NEON C intrinsics.

```
void example01_neon(int *restrict a, const int *b,
                    const int *c, long N)
{
  long i;
  // vector loop
  for (i = 0; i < N - 3; i += 4) {
    int32x4_t vb = vld1q_s32(b + i);
    int32x4_t vc = vld1q_s32(c + i);
    int32x4_t va = vaddq_s32(vb, vc);
    vst1q_s32(a + i, va);
  }
  // loop tail
  for (; i < N; ++i)
    a[i] = b[i] + c[i];
}
```

Listing 1.3 VLA SVE code for listing 1.1.

```
    // x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'i', x4 is 'N'
    mov     x3, 0 // set 'i=0'
    b       cond  // branch to 'cond'
loop_body:
    ld1w    z0.s, p0/z, [x1, x3, lsl 2] // load vector z0 from address 'b + i'
    ld1w    z1.s, p0/z, [x2, x3, lsl 2] // same, but from 'c + i' into vector z1
    add     z0.s, p0/m, z0.s, z1.s      // add the vectors
    st1w    z0.s, p0, [x0, x3, lsl 2]   // store vector z0 at 'a + i'
    incw    x3                          // increment 'i' by number of words in a vector
cond:
    whilelt p0.s, x3, x4  // build the loop predicate p0, as p0.s[idx] = (x3+idx) < x4
                          // it also sets the condition flags
    b.first    loop_body  // branch to 'loop_body' if the first bit in the predicate
                          // register 'p0' is set
    ret
```

checks whether the first (LSB) lane of `p0.b` is set to true, i.e. whether there are any further elements to process in the next iteration of the loop. Notice that the concept of lanes refers to the element size specifier used in the condition setting instruction, as per example in listing 1.4. SVE provides many conditions that can be used to check the condition flags. For example, `b.none` checks if all the predicate lanes have been set to false, `b.last` checks if the last lane (MSB) is set to true, `b.any` if any of the lanes of the predicate are set to true. Notice that concepts like *first* and *last* in the vector requires the introduction of an ordering which in case of SVE is mapped to *LSB* (or *lowest numbered element*) and *MSB* (or *highest numbered element*) respectively. The list of instructions that can be used to test the condition flags set by SVE instructions is shown in table 1.1.

Listing 1.4 Example of predicate register with 32-bit lanes view.

```
     MSB                                   LSB
P0 = [0000 0001 0001 0001 0001 0001 0001 0001]
         7    6    5    4    3    2    1    0  32-bit lanes 'idx'
```

Table 1.1: SVE branching instruction testing condition flags.

Branch instruction	SVE interpretation
`b.none`	No active elements are true.
`b.any`	An active element is true.
`b.nlast`	The last active element is not true.
`b.last`	The last active element is true.
`b.first`	The first active element is true.
`b.nfirst`	The first active element is not true.
`b.pmore`	An active element is true but not the last element.
`b.plast`	The last active element is true or none are true.
`b.tcont`	Scalarized CTERM loop termination not detected.
`b.tstop`	Scalarized CTERM loop termination detected.

The assembly example in listing 1.3 is equivalent to listing 1.5.

Listing 1.5 C pseudo code for the blocks in listing 1.3.

```
while( /* cond */ ) {
  /* loop */
}
```

The loop body is also vector length agnostic. In each iteration, the operations performed are:

- load two vectors of data from `b` and `c` respectively, with `ld1w`. Here the loads are using *register plus register addressing mode*, where the index register `x4` is being left shifted by two bits to scale the index by 4, corresponding to the size of the scalar data;
- add the values into another register with the `add` instruction;
- store the value computed into `a` with `st1w` (same register plus register addressing as `ld1w`).
- increment the index `x4` by as many words as a vector can store with `incw`.

All the instructions in the `loop_body` are predicated with `p0`, meaning that inactive 32-bit lanes are not accessed by the instruction, so that the scalar loop tail is not needed.

The `incw` instruction is an important VLA feature used in the code. This loop will execute correctly on any implementation of SVE, as `incw` will take care of increasing the loop iterator according to the current SVE vector length.

For the purpose of comparison, an assembly version of the NEON code in listing 1.2 is shown in listing 1.6, showing both the vector body and the loop tail blocks.

Listing 1.6 NEON assembly code generated from the C intrinsics in listing 1.2.

```
    # x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x8 is the loop induction variable 'i'
    mov     x8, xzr
    subs    x9, x3, 3               # x9 = N -3
    b.ls    .loop_tail_preheader    # jump to loop tail if N <= 3
.vector_body:
    ldr     q0, [x1, x8, lsl 4]     # load 4 elements from 'b+i'
    ldr     q1, [x2, x8, lsl 4]     # load 4 elements from 'c+i'
    add     v0.4s, v1.4s, v0.4s     # add the vector
    str     q0, [x0, x8, lsl 4],    # store 4 elements in 'a+i'
    add     x8, x8, 4               # increment 'i' by 4
    cmp     x8, x9                  # compare i with N - 3
    b.lo    .vector_body            # keep looping if i < N-3
.loop_tail_preheader:
    cmp     x8, x3                  # compare the loop counter with N
    b.hs    .function_exit          # if greater or equal N, terminate
.loop_tail:
    ldr     w12, [x1, x8, lsl 2]
    ldr     w13, [x2, x8, lsl 2]
    add     w12, w13, w12
    str     w12, [x0, x8, lsl 2]
    cmp     x8, x3
    b.lo    .loop_tail              # keep looping until no elements remain
.function_exit:
        ret
```

More on predication

Predication can also be used to vectorize loops with control flow in the loop body. The `if` statement of listing 1.7 is executed in the vector `loop-body` of the code in listing 1.8 by setting the predicate `p1` with the `cmpgt` instruction, which tests for *compare greater than*. This operation produces a predicate that selects which lanes have to be operated by the if-guarded instruction. The loads and the stores of the data in `a`, `b` and `c` arrays are performed by instructions that use the predicate `p1`, so that the only elements in memory that are modified correspond to those modified by the original C code.

Listing 1.7 a loop with conditional execution.

```
void example02(int *restrict a, const int *b, const int *c, long N,
                      const int *d)
{
  long i;
  for (i = 0; i < N; ++i)
    if (d[i] > 0)
      a[i] = b[i] + c[i];
}
```

Listing 1.8 SVE vector version of listing 1.7. Notice that the comments in the assembly code relate only to the predication specific behavior that is being shown.

```
    // x0 is 'a', x1 is 'b', x2 is 'c', x3 is 'N', x4 is 'd', x5 is 'i'
    mov     x5, 0 // set 'i = 0'
    b       cond
loop_body:
    ld1w    z4.s, p0/z, [x4, x5, lsl 2] // load a vector from 'd + i'
    cmpgt   p1.s, p0/z, z4.s, 0         // compare greater than zero
                                        // p1.s[idx] = z4.s[idx] > 0
    // from now on all the instructions depending on the 'if' statement are
    // predicated with 'p1'
    ld1w    z0.s, p1/z, [x1, x5, lsl 2]
    ld1w    z1.s, p1/z, [x2, x5, lsl 2]
    add     z0.s, p1/m, z0.s, z1.s
    st1w    z0.s, p1, [x0, x5, lsl 2]
    incw    x5
cond:
    whilelt p0.s, x5, x3
    b.ne    loop_body
    ret
```

Merging and zeroing predication

Some of the data processing instructions have two different kinds of predication, *merging* and *zeroing*. The former is indicated by the `/m` qualifier attached to the instruction's governing predicate, as in `add z0.s, p1/m, z0.s, z1.s`; the latter by the `/z` qualifier, as in `cmpgt p1.s, p0/z, z4.s, 0`.

Merging and zeroing predication differ in the way the instruction operates on inactive lanes. Zeroing sets the inactive lanes to zero, while merging leaves the inactive lanes unchanged.

The examples in listing 1.9 and listing 1.10 show how merging predication can be used to perform a conditional reduction.

Listing 1.9 a reduction.

```
int example02(int *a, int *b, long N)
{
  long i;
  int s = 0;
  for (i = 0; i < N; ++i)
    if (b[i])
      s += a[i];
  return s;
}
```

Listing 1.10 SVE vector version of listing 1.9.

```
        mov     x5, 0   # set 'i = 0'
        mov     z0.s, 0 # set the accumulator 's' to zero
        b       cond
loop_body:
        ld1w    z4.s, p0/z, [x1, x5, lsl 2] # load a  vector
                                            # at 'b + i'
        cmpne   p1.s, p0/z, z4.s, 0         # compare non zero
                                            # into predicate 'p1'
        # from now on all the instructions depending on the 'if' statement are
        # predicated with 'p1'
        ld1w    z1.s, p1/z, [x0, x5, lsl 2]
        add     z0.s, p1/m, z0.s, z1.s      # the inactive lanes
                                            # retain the partial sums
                                            #  of the previous iterations
        incw    x5
cond:
        whilelt p0.s, x5, x3
        b.first loop_body
        ptrue p0.s
        saddv d0, p0, z0.s # signed add words across the lanes of z0, and place the
                           # scalar result in d0
        mov w0, v0.s[0]
        ret
```

Gather loads

Another important feature introduced with SVE is the *gather load* / *scatter store* set of instructions, which allow it to operate on non contiguous arrays of memory, like the code shown in listing 1.11.

Listing 1.11 loading data from an array of addresses.

```
void example03(int *restrict a, const int *b, const int *c,
               long N, const int *d)
{
  long i;
  for (i = 0; i < N; ++i)
      a[i] = b[d[i]] + c[i];
}
```

The vector code in listing 1.12 uses a special version of the `ld1w` instruction to load the data at `b[d[i]]`, `ld1w z0.s, p0/z, [x1, z1.s, sxtw 2]`. The values stored in `z1.s` are interpreted as 32-bit scaled indices, and sign extended (`sxtw`) to 64-bit before being left shifted by 2 and added to the base address `x4`. This addressing mode is called *scalar plus vector* addressing mode. This code shows only one example of it. To account for many other situations that occur in real world code, other addressing modes support a 32-bit unsigned index or a 64-bit index, with and without scaling.

Listing 1.12 SVE vectorized version of listing 1.11.

```
        mov     x5, 0
        b       cond
loop:
        ld1w    z1.s, p0/z, [x4, x5, lsl 2]
        ld1w    z0.s, p0/z, [x1, z1.s, sxtw 2] # load a vector
                                               # from  'x1 + sxtw(z1.s) << 2'
        ld1w    z1.s, p0/z, [x2, x5, lsl 2]
        add     z0.s, p0/m, z0.s, z1.s
        st1w    z0.s, p0, [x0, x5, lsl 2]
        incw    x5
cond:
        whilelt p0.s, x5, x3
        b.first    loop
    ret
```

2 Implementing `string.h` routines[6]

In the final part of this article I will show how to implement custom versions of `strcpy` and `strcmp` from the C runtime library, using SVE. Of course, both examples can be written in vector form using NEON - adding proper runtime checks for the array boundaries, and taking care of *leftovers* in the loop tail.

For each of the examples, I will point out which particular feature is novel compared to NEON (and traditional SIMD in general).

SVE includes a status register, called *First Faulting Register (FFR)*, that addresses one of the problems of runtime checks, specifically how to speculatively vectorize a loop when the bounds are unknown in advance.[7]

The FFR is an additional predicate register. Regular SVE instructions cannot access it. Instead, some special instructions are used to read or *initialize* it. There is `setffr`, which initializes all FFR elements to true, and `rdffr` - both in the predicated and unpredicated version - that copies the content of the FFR into a regular predicate register.

This register is only useful in conjunction with the *first faulting load* class of instructions. These speculatively load data from memory without raising any software visible fault other than for the element in the first lane. For example, loading a full vector from a span of memory that crosses your object boundaries might result in a runtime error in traditional SIMD architecture - what we sometimes call a *segmentation fault*. Using the SVE first faulting loads this does not happen. Instead, the load proceeds *until* the fault is detected, but instead of delivering a signal, the instruction completes without errors. What the instruction does instead is to set the FFR so that the predicate lanes which correspond to the successfully loaded lanes are active, while the subsequent predicate lanes starting from the first fault are marked as inactive, and the corresponding lanes of data are not loaded.

The segmentation fault behavior is not totally removed. If the first active element in a first faulting load would cause a segmentation fault then the segmentation fault will be raised, because this corresponds to a bug in the original scalar code.

As shown below, instructions are provided that allow the user to write loops so if the loop does not terminate, then after a detected fault the next iteration of the loop will continue from the faulting address. This technique ensures that the signal that is delivered by SVE code is identical to the signal that scalar code would deliver.

In figure 2.1, an example of first faulting load behavior is shown. This artificial example is looping through a 7 element array and loads data from it into vector registers.

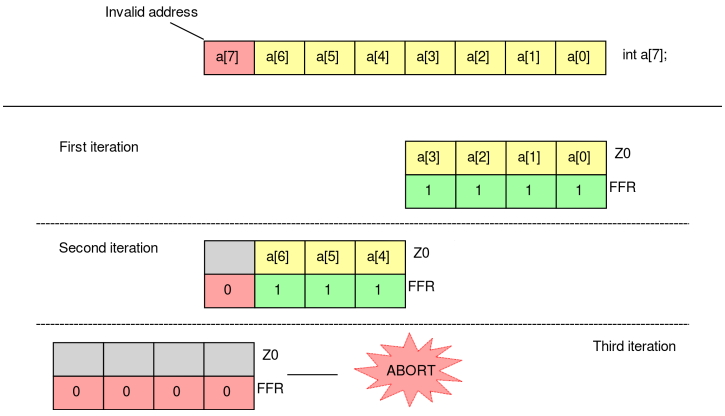

Figure 2.1: First faulting load mechanism.

Each of the iteration in the graphic shows the data and FFR output of a first faulting load, issued with the governing predicate and FFR initialised to have all lanes active.

First iteration The data is fully loaded, and the FFR is set to all true.

[6] As of September 2018 I became aware of a performance problem with the SVE code presented in this chapter, that uses the first faulting load mechanism. The original code is left here for completeness, and provides a functional introduction to the FFR register and related instructions. In a dedicated section in the appendix I describe how to avoid the performance degradation from suboptimal use of the FFR register.

[7] It doesn't solve other runtime check problems (like overlapping arrays, for example).

Second iteration The full vector crosses the boundary of the valid memory. Elements beyond the boundary are discarded, the failing lane is set to false in the FFR, and the failure propagates to the subsequent lanes. The FFR can be combined with the main predicate of the loop to avoid processing lanes which correspond to scalar loop iterations subsequent to the first fault.

Third iteration The final iteration restarts from the last failure (the invalid address), but this time, since the invalid address now corresponds to the first element, a segmentation fault is raised.

Copying strings

The first use of the FFR I will present is in the string copy routine, `strcpy`. The C code is presented in listing 2.1: the `while` statement is performed until a string terminator character is found when dereferencing `src`.

In this section we need to present another innovative aspect introduced with SVE, the concept of loops operating on a *dynamic* vector partition (as opposed to a whole vector). A vector partition is what you get from instructions that produce a predicate (e.g. a `while` instruction) in response to dynamic values such as the loop counter, loop bound and vector length. You can also generate a partition from a vector compare instruction if you can't use a simple monotonic counter. You narrow the current partition in response to other dynamic conditions such as memory faults using instructions that read the FFR and then data-dependent loop termination conditions using other partitioning instructions.

The need for dynamic partition is well illustrated by code shown in listing 2.1. One way of vectorizing this loop is to provide a set of instructions that are able is to determine which lanes represent iterations prior to a break condition and which follow it, like SVE does with the `brka/b` family of instructions.

Listing 2.1 Custom `strcpy`-like routine, C code. In this custom version, the routine does not return a copy of the original value of the `dst` argument.

```
1 void strcpy(char *restrict dst, const char *src)
2 {
3   while (1) {
4     *dst = *src;
5     if (*src == '\0') break;
6     src++; dst++;
7   }
8 }
```

The SVE assembly in listing 2.2 is split in three parts: a *header* , a *loop body*, and the *function exit*. The header prepares the data needed before entering the loop body. There is an *implicit* loop counter `x2`, set to zero, and a predicate register `p2` whose elements are all set to true with `ptrue`.

The *loop body* performs the following sequence of actions:

1. Initializes the FFR to all true with `setffr`. This operation is needed because elements in FFR are only changed by `ld1ff` as the result of a fault.
2. Attempts to speculatively load a full vector of data from `src`, plus the current offset in the implicit loop counter `x2`, using the first-faulting 'ldff1b' instruction.
3. Copy the content of the FFR into `p0`. At this point `p0` holds the information about which elements in the `src` vector contain valid data.
4. Now that we have loaded a chunk of the `src` vector, we need to copy it, up to the terminating zero byte. To find this point, we can use the `cmpeq` instruction, which tells us in which lanes of `z0` there is a zero byte. A key point here is that the comparison condition flags are set according to the governing predicate of the `cmpeq` instruction, i.e. ignoring the value of lanes which were marked as faulty in FFR.

5. We use `brka` - *break after* to clear out those lanes in the predicate that we don't want to copy, effectively partitioning the vector in two. This instruction operates on a predicate register, setting lanes up to and including the first true comparison to be active, and then all subsequent lanes to be inactive. The predicate it builds, `p0`, accounts for conditions 1) the data is valid (thanks to the first-faulting mechanism), and 2) the data is only part of `src` (through the `cmpeq` instruction).
6. Now we can store the data loaded from `src` into `dst` using `st1b` with the predicate obtained in the previous step (the counter `x2` is used as an offset from the base address of `dst`).
7. Increment the implicit counter `x2`, using `incp`which increments its general purpose register operand by the count of active lanes in the predicate operand.
8. The condition flags set by `cmpeq` are read by `b.none` to decide whether *none* of the `src` bytes are zero, and keep looping or else exit the routine with `ret`.

Listing 2.2 SVE assembly implementing `strcpy` in listing 2.1.

```
1  sve_strcpy:             # header
2          mov     x2, 0
3          ptrue   p2.b
4  loop:                   #loop body
5          setffr
6          ldff1b  z0.b, p2/z, [x1, x2]
7          rdffr   p0.b, p2/z
8          cmpeq   p1.b, p0/z, z0.b, 0
9          brka    p0.b, p0/z, p1.b
10         st1b    z0.b, p0, [x0, x2]
11         incp    x2, p0.b
12         b.none  loop
13         ret             # function exit
```

The correctness of this particular algorithm relies in the combined use of the first-faulting mechanism, the `brka` and `incp` instructions. The increment of the loop counter with the length of the partition, i.e. `brka` + `incp`, guarantees that the computation is performed without leaving holes between subsequent iterations.

The code in listing 2.2 is quite simple. With NEON, the programmer (or the compiler) would have had to:

1. add some tricks to make sure that the loads cannot cross the allocated memory for `src`;
2. process whatever tail elements are left in an additional `loop_tail` block.

Comparing strings

The last example I am going to present is an SVE implementation of the `strcmp` routine defined in the C runtime library. This time the C code shown in listing 2.3 is compatible with the definition in the standard[8].

The SVE assembly code shown in listing 2.4 uses the FFR, and has a structure similar to one of the previous example `strcpy`: a *loop header* followed by a *loop body* and a *post-loop* block that terminates the procedure.

As before, the loop header sets an implicit loop counter `x2` and a predicate `p0` with all bits set to active.

We need to speculatively load data from `lhs` and `rhs` to perform the comparisons in the while condition, so we need to make sure that both associated loads are reading data from valid addresses. In the loop body, this is achieved by chaining the two `ldff1b` instructions: the FFR accumulates the faults from each `ld1ffb` instruction so that `rdffr`

[8] http://en.cppreference.com/w/c/string/byte/strcmp

Listing 2.3 An implementation of `strcmp`.

```
int strcmp(const char *lhs, const char *rhs)
{
  while (*lhs == *rhs && *lhs != 0)
    lhs++, rhs++;
  return (int) ((unsigned int) *lhs - (unsigned int) *rhs);
}
```

returns a (partition) predicate indicating which lanes were successfully loaded from both `lhs` and `rhs`. The implicit loop counter `x2` is incremented by the number of active lanes in FFR following the speculative load.

From now on, `p1` becomes the main predicate for the loop. Two compare instructions - `cmpeq` (equal) and `cmpne` (not equal) - build the predicates `p2` and `p3` needed to test `*lhs == *rhs` and `*lhs != 0`, respectively.

The final predicate `p2` is built with a `nands` instruction, that checks `not(p2 and p3)`, and sets the condition flags. If none of the lanes of `p2` are true, the branch instruction `b.none` jumps back to the start of the loop.

The `terminate` block finalizes the comparison:

1. The `brkb` (break *before*) splits the comparison result predicate `p2` by setting lanes in `p2` up to but not including the first true comparison (the first mismatch or the end of both strings) to be active, and then all subsequent lanes to be inactive.
2. The `lasta` instructions extract the lane of the input vectors `z0` and `z1` where a mismatch or a terminating `'\0'` was found, and place them in `w0` and `w1` respectively.
3. A final `sub` computes the difference of the two unsigned values.

Listing 2.4 SVE implementation of `strcmp` in listing 2.3.

```
        mov x2, 0       // loop header
        ptrue   p0.b
loop:                   // loop body
        setffr
        ldff1b  z0.b, p0/z, [x0, x2]
        ldff1b  z1.b, p0/z, [x1, x2]
        rdffr   p1.b, p0/z
        incp    x2, p1.b
        cmpeq   p2.b, p1/z, z0.b, z1.b
        cmpne   p3.b, p1/z, z0.b, 0
        nands   p2.b, p1/z, p2.b, p3.b
        b.none  loop
terminate:              // post-loop block
        brkb    p2.b, p1/z, p2.b
        lasta   w0, p2, z0.b
        lasta   w1, p2, z1.b
        sub     w0, w0, w1
        ret
```

A NEON version of this routine has to take care of the same additional runtime checks and loop tail blocks as the `strcpy` in the example before, plus a mechanism to extract the correct element from the result vector - which is performed with `lasta` in SVE.

3 Conclusions

In this whitepaper we have explained some of the key features that are introduced by the new Scalable Vector Extension for AArch64, and shown some examples of how loops can be speculatively vectorized using the Vector Length Agnostic approach.

Although the instruction set will only be publicly released at the beginning of 2017, a set of patches in the GNU binutils[9] repository already provide assembler functionality for building SVE-enabled programs.

Moreover, both `gcc`[10] and `clang`[11] will provide full SVE support soon, which will allow programmers to write VLA code in SVE assembly language, or using the SVE *ACLE* C intrinsics, or by enabling the new auto-vectorization capabilities which ARM is contributing to these compilers to support SVE.

Happy SVE hacking!

[9] http://sourceware.org/binutils/
[10] https://gcc.gnu.org/
[11] http://clang.llvm.org/

Appendix

Optimal use of the first faulting mechanism

As I have shown in the section on the string routines, the first faulting mechanism ensures that data can be loaded across page boundaries. In the case of the `strcpy` routine in example listing 2.2, the FFR is used to force the SIMD processing instructions to operate only on non-faulting lanes, and to recover computation from the last position of valid memory.

However, as soon as we are dealing with relatively large strings, the probability of processing partial vectors due to attempted accesses to invalid memory locations becomes lower than the probability of processing full vectors. This means that it is better to separate the code into two blocks, one that processes full vectors, the `full-vector` block, with no dependencies on the value of the FFR, and one that takes care of *fixing up* the partial vector loop iteration in case a fault happens, the `fixup` block.

By removing the dependency on the governing predicate in the `full-vector` block and the FFR in the `load` block, an out-of-order processor can speculate on subsequent iterations of the `full-vector` block in advance, and rewind the computation of the speculated iterations in the rare case of a faulting event.

For the implementation of `strcpy` in listing 2.2, an equivalent (but better) version is shown in figure listing 3.2. Notice that the use of the `setffr` instruction is further optimized by making sure that once the FFR is set at function entry, it is re-set only when the fixup block is executed.

The implementation of `strcmp` in listing 2.4 can be optimized with the same technique. The code of the new optimized version is shown in listing 3.2.

Listing 3.1 Optimal use of the FFR in `strcpy`.

```
1  sve_strcpy_optimized:
2          mov     x2, 0
3          ptrue   p2.b
4          setffr
5  loop:
6          ldff1b  z0.b, p2/z, [x1, x2] # load block
7          rdffrs  p0.b, p2/z           #
8          b.nlast fixup                #
9
10         cmpeq   p1.b, p2/z, z0.b, 0  # full-vector block
11         brka    p3.b, p2/z, p1.b     #
12         st1b    z0.b, p3, [x0, x2]   #
13         incb    x2                   #
14         b.none  loop                 #
15
16         ret
17 fixup:
18         cmpeq   p1.b, p0/z, z0.b, 0  # fixup block
19         brka    p3.b, p0/z, p1.b     #
20         st1b    z0.b, p3, [x0, x2]   #
21         incp    x2, p0.b             #
22         setffr                       #
23         b.none   loop                #
24
25         ret
```

Listing 3.2 Optimal use of the FFR in `strcmp`.

```
sve_strcmp_optimized:                   // @sve_strcmp_optimized
        mov x2, 0
        ptrue   p0.b
        setffr
loop:
        ldff1b  z0.b, p0/z, [x0, x2]    # load block
        ldff1b  z1.b, p0/z, [x1, x2]    #
        rdffrs  p1.b, p0/z              #
        b.nlast fixup                   #

        cmpeq   p2.b, p0/z, z0.b, z1.b # full-vector block
        cmpne   p3.b, p0/z, z0.b, 0     #
        nands   p2.b, p0/z, p2.b, p3.b #
        incb    x2, all                 #
        b.none  loop                    #
terminate:
        brkb    p2.b, p1/z, p2.b
        lasta   w0, p2, z0.b
        lasta   w1, p2, z1.b
        sub     w0, w0, w1
        ret
fixup:
        cmpeq   p2.b, p1/z, z0.b, z1.b # fixup block
        cmpne   p3.b, p1/z, z0.b, 0     #
        nands   p2.b, p1/z, p2.b, p3.b #
        b.any   terminate               #
        incp    x2, p1.b                #
        setffr                          #
        b       loop                    #
```

Changelog

February 2020 Fix register to variable mappings in listing 1.3. Defect reported by Nicholas Dingle.

September 2018 List of changes:

- Added appendix, with a section on optimal use of the first faulting mechanism.
- Address C11 compliancy for the `strcmp` function in listing 2.3. Feedback provided by Richard Henderson.

November 2016 Initial release.

Trademarks

The trademarks featured in this document are registered and/or unregistered trademarks of ARM Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved. All other marks featured may be trademarks of their respective owners. For more information, visit ARM website[12].

[12] arm.com/about/trademarks