ARMv8-A Next-Generation Vector Architecture for HPC

Nigel Stephens
Lead ISA Architect and ARM Fellow

Hot Chips 28, Cupertino
August 22, 2016

Expanding ARMv8 vector processing

- ARMv7 Advanced SIMD (*aka* ARM NEON instructions) now 12 years old
  - Integer, fixed-point and non-IEEE single-precision float, on *well-conditioned* data
  - 16×128-bit vector registers
- AArch64 Advanced SIMD was an evolution
  - Gained full IEEE double-precision float and 64-bit integer vector ops
  - Vector register file grew from 16×128b to 32×128b
- New markets for ARMv8-A are demanding more radical changes
  - ✓ Gather load & Scatter store
  - ✓ Per-lane predication
  - ✓ Longer vectors
- But what is the preferred vector length?

Introducing the Scalable Vector Extension (SVE)

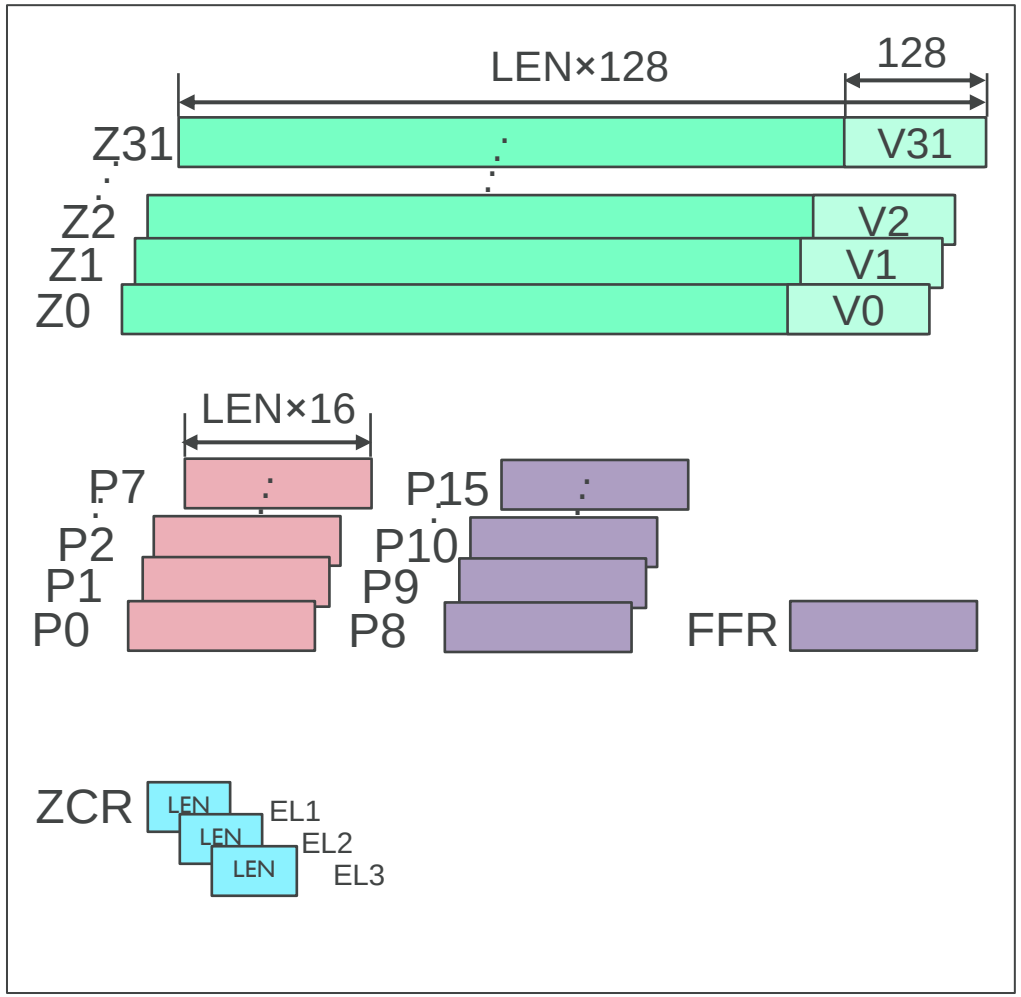
- There is no preferred vector length
  - Vector Length (VL) is hardware choice, from 128 to 2048 bits, in increments of 128
  - *Vector Length Agnostic* (VLA) programming adjusts dynamically to the available VL
  - No need to recompile, or to rewrite hand-coded SVE assembler or C intrinsics

- SVE is not an extension of Advanced SIMD
  - A separate architectural extension with a new set of A64 instruction encodings
  - Focus is HPC scientific workloads, not media/image processing

- Amdahl says you need high vector utilisation to achieve significant speedups
  - Compilers often unable to vectorize due to intra-vector data & control dependencies
  - SVE also begins to address some of the traditional barriers to auto-vectorization

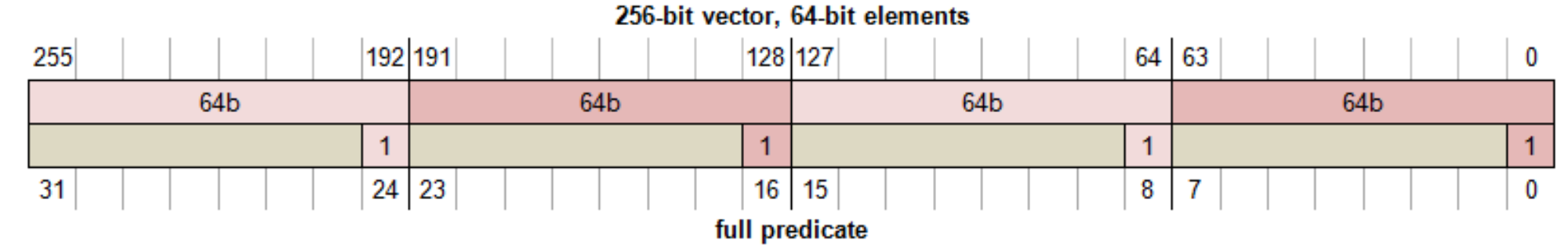
SVE architectural state

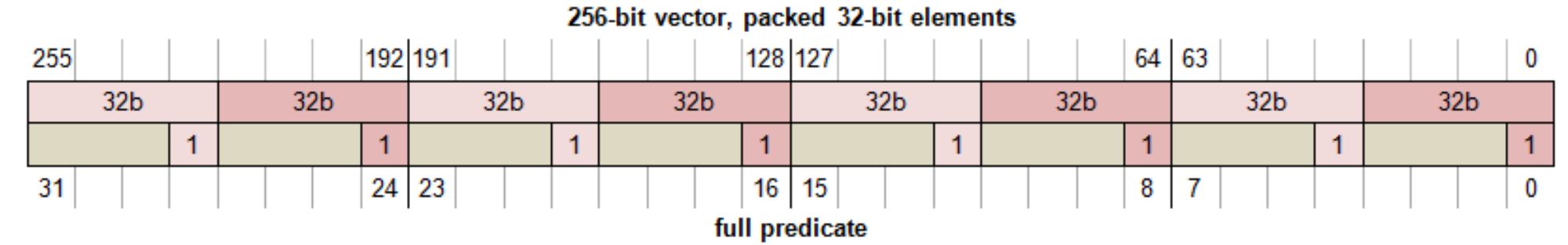
- Scalable vector registers
  - Z0-Z31 extending NEON's V0-V31
    - DP & SP floating-point
    - 64, 32, 16 & 8-bit integer
- Scalable predicate registers
  - P0-P7 lane masks for ld/st/arith
  - P8-P15 for predicate manipulation
  - FFR *first fault register*
- Scalable vector control registers
  - ZCR_ELx vector length (LEN=1..16)
  - Exception / privilege level EL1 to EL3

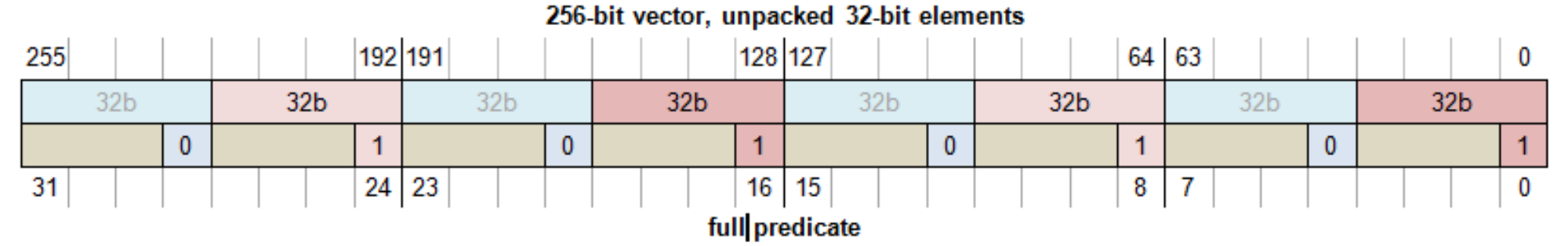


ARM

Vector & predicate element organization







ARM

Vector loop control flow

- Predicates are central to SVE
- Predicates drive loop control flow
  - Reducing loop management overhead
- Overloads NZCV condition flags:

Flag	SVE	Condition
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		Scalarized loop state, else zero

- Reuses A64 cond instructions
  - Conditional branch B.EQ → B.NONE
  - Conditional select, set, increment, etc

Condition Test	A64 Name	SVE Alias	SVE Interpretation
Z=1	**EQ**	**NONE**	No elements are active
Z=0	**NE**	**ANY**	An element is active
C=1	**CS**	**NLAST**	Last element is not active
C=0	**CC**	**LAST**	Last element is active
N=1	**MI**	**FIRST**	First element is active
N=0	**PL**	**NFRST**	First element is not active
C=1 & Z=0	**HI**	**PMORE**	An element is active but not the last: more partitions
C=0 | Z=1	**LS**	**PLAST**	Last element is active or none are: last partition
N=V	**GE**	**TCONT**	Continue scalarized loop
N!=V	**LT**	**TSTOP**	Stop scalarized loop

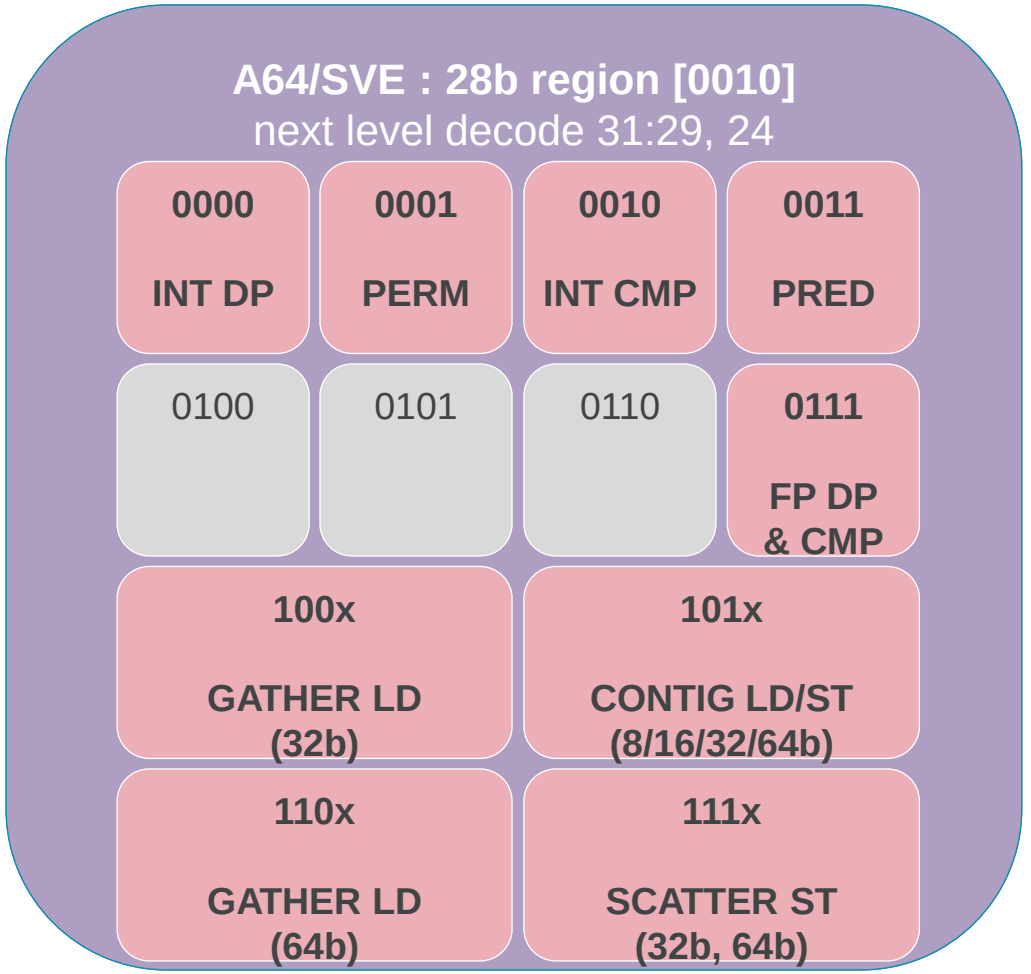
SVE instruction set

- AArch64 (A64) encoding
  - AArch32 not important for HPC
  - SVE fits in a 28-bit region
    - i.e. one sixteenth of A64

ARM

SVE instruction set

- AArch64 (A64) encoding
  - AArch32 not important for HPC
  - SVE fits in a 28-bit region
    - i.e. one sixteenth of A64
- Targets compiled programs
  - C/C++ and Fortran data types
  - Floating-point, integer & logical ops
  - Permutes, reductions, predicate ops
  - Contiguous & G/S load/store
  - × No fixed-point & DSP/media ops



ARM

Fitting predication into a restricted encoding

- *Constructive* binary instructions with predication are expensive
  - **ADD Zd.<sz>, Pg/[ZM], Zs1.<sz>, Zs2.<sz>**
  - Encoding needs **5b** + **5b** + **5b** + **3b** + **2b** + **1b** → 21 bits
  - Merging predication also needs 3 vector sources: **Zs1**, **Zs2** and old **Zd**
- Hence most SVE arithmetic is *destructive*, with merging predication
  - **ADD Zds1.<sz>, Pg/M, Zs2.<sz>** →15 bits
- If needed, constructive instructions can be formed as an *instruction pair*
  - **MOVPRFX Zd, Pg/[ZM], Zs1** → **ADD Zd, Pg/[ZM], Zs1, Zs2**

    ADD Zd, Pg/M, Zs2
  - Hardware can treat as a single 64-bit instruction (s/w rules make this easy)
    - … or as separate **MOV** and **ADD** instructions, with no hidden state

Impacts of VL-agnostic (VLA) approach

- Vectors cannot be initialised from compile-time constant in memory, so…
  - `INDEX Zd.S,#1,#4` : `Zd = [ 1, 5, 9, 13, 17, 21, 25, 29 ]`
- Predicates cannot be initialised from memory, so…
  - `PTRUE Pd.S,MUL3` : `Pd = [ T, T, T , T, T, T , F, F ]`
- Vector loop increment and trip count are unknown at compile-time, so…
  - `INCD Xi` : increment scalar `Xi` by # of 64-bit dwords in vector
  - `WHILELT Pd.D,Xi,Xe` : next iteration predicate `Pd = [ while i++ < e ]`
- Vector register spill & fill must adjust to vector length, so…
  - `ADDVL SP,SP,#-4` : decrement stack pointer by (4*VL)
  - `STR Z1,[SP,#3,MUL VL]` : store vector Z1 to address (SP + 3*VL)

DAXPY (scalar)

```
// ----------------------------------------
//      subroutine daxpy(x,y,a,n)
//      real*8 x(n),y(n),a
//      do i = 1,n
//         y(i) = a*x(i) + y(i)
//      enddo
// ----------------------------------------
// 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
```

DAXPY (SVE)

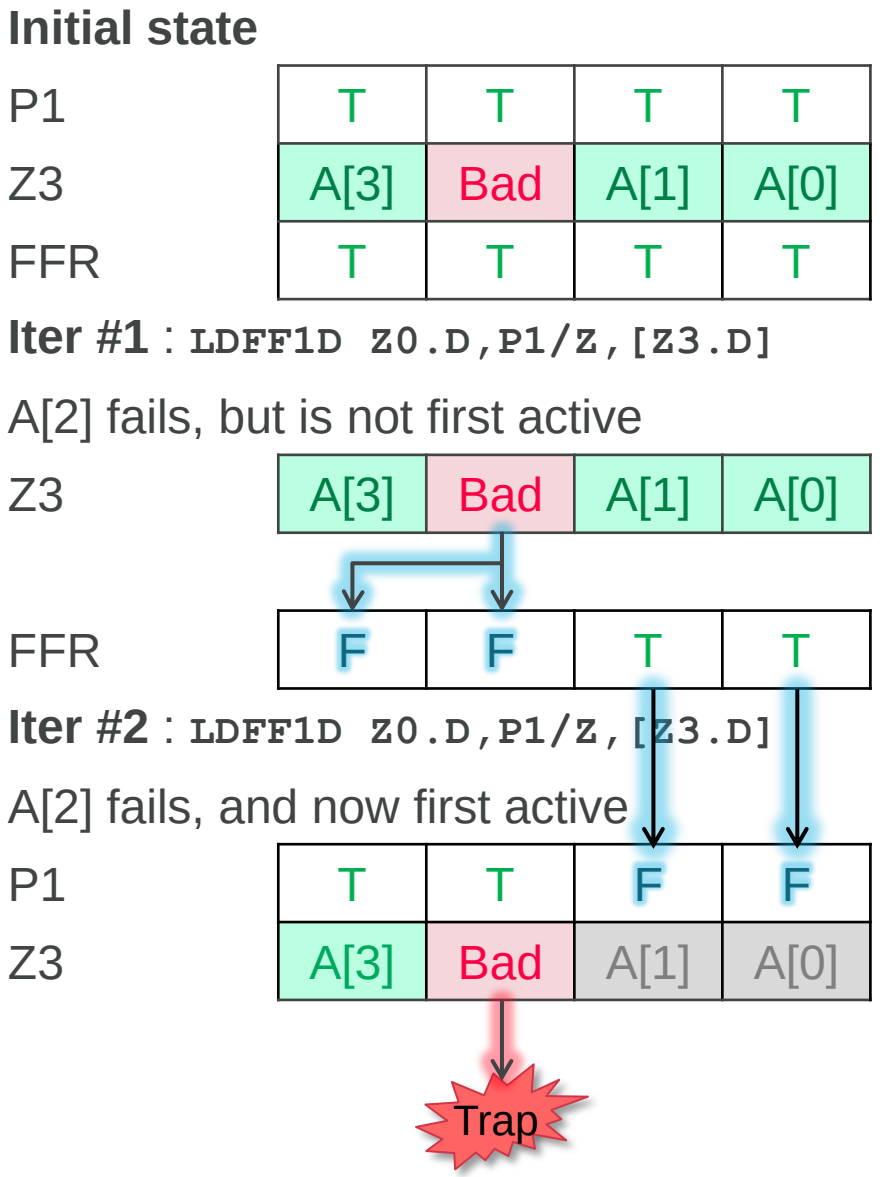
```
// ----------------------------------------
//      subroutine daxpy(x,y,a,n)
//      real*8 x(n),y(n),a
//      do i = 1,n
//         y(i) = a*x(i) + y(i)
//      enddo
// ----------------------------------------
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
daxpy_:
    ldrsw   x3, [x3]              // 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
```

Vector partitioning and speculation

- Vector partitioning uses predication to allow speculative vectorisation
  - Operate on a *partition* of “safe” elements in response to dynamic conditions
  - Partitions are inherited by nested conditions and loops
- Uncounted loops with data-dependent exits (do...while, break, etc)
  - Operations with side-effects following a break must not be architecturally *performed*
  - Operate on before-break vector partition, then exit loop if break was detected
- Speculative vector loads
  - Vector loads required to detect the break condition might fault on lanes following it
  - Operate on before-fault vector partition, then iterate until break is detected
- Vector-Length Agnosticism is a special case of vector partitioning
  - Partition defined by dynamic vector length

Speculative first-fault loads using FFR

- Speculative gather from addresses in Z3
  - **`SETFFR`**: initialises FFR to all TRUE
  - A[2] is a Bad (e.g. unmapped) address
- **Iteration #1**:
  - A[0]: first active succeeds
  - A[1]: speculative succeeds
  - A[2]: speculative FAILS
    - no trap, FFR[2..VL] are cleared
  - **`RDFFR`** : returns FFR partition
- **Iteration #2**:
  - Clear P1[0..1], reinitialise FFR to all TRUE
  - A[2]: now first active FAILS
    - traps to OS to service fault, or
    - terminate program if illegal access
  - Latter is real bug: please stop before A[2]!



strlen (scalar)

```
// -----------------------------------------
//      int strlen(const char *s) {
//          const char *e = s;
//          while (*e) e++;
//          return e - s;
//      }
// -----------------------------------------
// x0 = s

// Unoptimized A64 scalar strlen
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
```

strlen (SVE)

```
// -----------------------------------------
//      int strlen(const char *s) {
//          const char *e = s;
//          while (*e) e++;
//          return e - s;
//      }
// -----------------------------------------
// x0 = s

// Unoptimized SVE strlen
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
```

Loop carried dependencies: linked list

- Linked list is a loop-carried dependency between each iteration

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

- Split loop into serial pointer chase followed by vectorizable loop

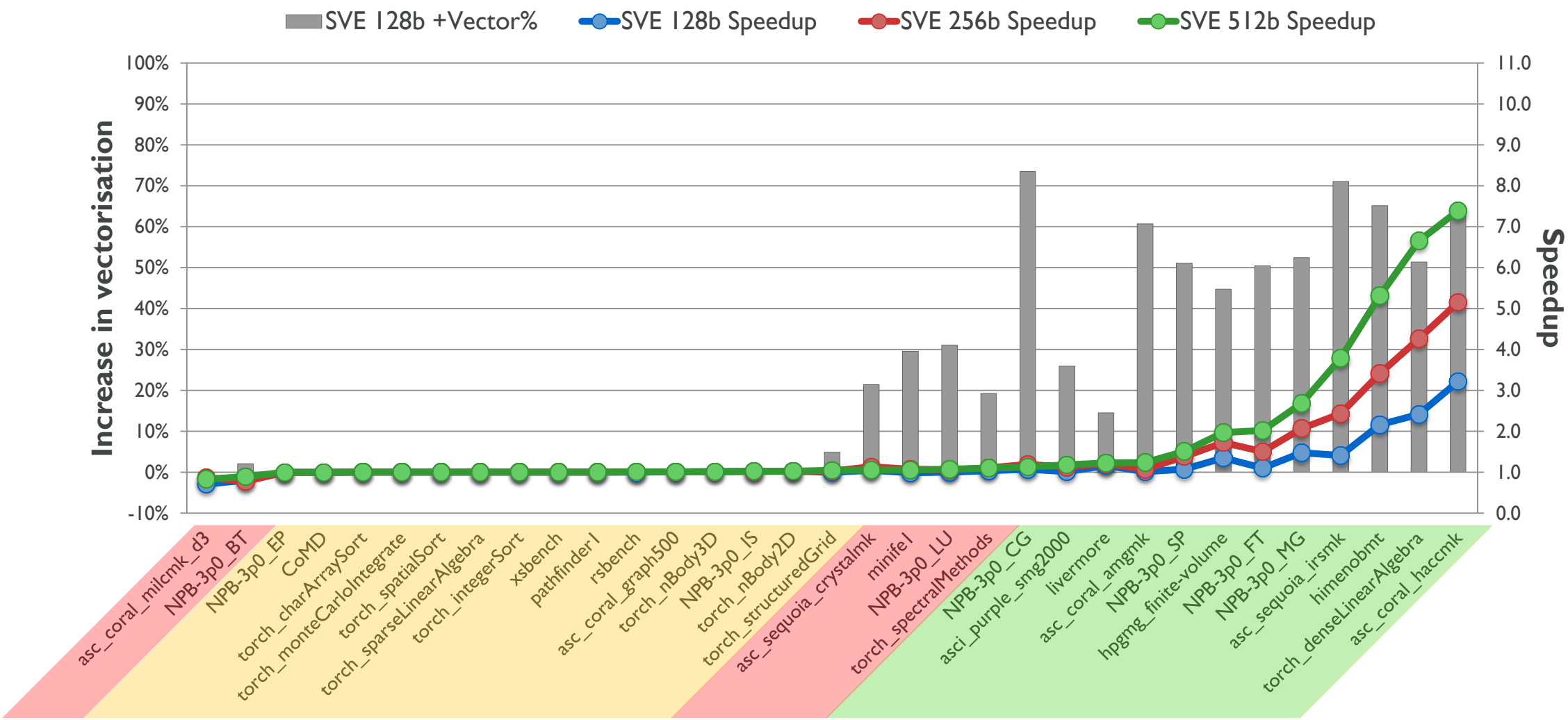
```
for (p = &head; p != NULL; ) {
   for (i = 0; p != NULL && i < VL/64; p = p->next)
    p'[i++] = p;            // collect up to VL/64 pointers
   for (j = 0; j < i; j++)
    res ^= p'[j]->val;      // gather from pointer vector
}
```

SVE scalar subloops

- **PNEXT:** iterates over partition
  - Operate in-place on active elements
- Also used in this example:
  - **CPY:** insert pointer into active lane
  - **CTERM:** detect end of list or end of vector, set N&V flags accordingly
  - **BRKA:** generate final predicate for assembled vector of pointers
  - **EORV:** horizontal exclusive-or reduction
- In reality an enabling mechanism
  - Only worthwhile if sufficient vectorizable code exists in the rest of the loop

```
  // 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
```

HPC benchmarks SVE vs NEON



ARM

Next Steps

- SVE designed for partners wishing to enter HPC market with ARMv8-A
  - Lead partners are implementing SVE, see recent announcements at ISC16
- Beginning engagement with open-source community
  - Upstreaming of patches and discussions to start within weeks
    - LLVM, GCC, Binutils, GDB
    - Linux kernel & KVM
- General specification availability in late 2016 / early 2017
  - SVE Architecture Overview
  - SVE AArch64 ABI changes
  - SVE C/C++ intrinsics

ARM

FUJITSU

shaping tomorrow with you

Introduction of Fujitsu's HPC Processor for the Post-K Computer

August 22nd, 2016

Toshio Yoshida
Advanced System Research & Development Unit
FUJITSU LIMITED

Key Message

FUJITSU

- Fujitsu, as a “lead partner,” has been collaborating closely with ARM and contributed to the development of the HPC extensions (called SVE) for ARMv8-A, a cutting-edge ISA optimized for a wide range of HPC applications

- Fujitsu is developing a new HPC processor conforming to ARMv8-A with SVE for the Post-K computer, based on our own microarchitecture, as used in our ongoing SPARC64 and mainframe processor development

Post-K Processor Overview

FUJITSU

- Enhances and inherits the superior features of the K computer, PRIMEHPC FX10 and FX100
  - High performance for a wide range of real applications
    - Many-core processor with 512-bit wide SIMD
    - Fujitsu HPC compiler for the ARM ISA, optimized for our microarchitecture
  - High scalability
    - Scalable, integrated Tofu interconnect
    - Assistant cores for daemons, IOs & MPI asynchronous communications
  - Optimized performance-per-watt

	Post-K	PRIMEHPC FX100	K computer/ PRIMEHPC FX10
ISA	ARMv8-A + SVE	SPARC-V9 + HPC-ACE2	SPARC-V9 + HPC-ACE
SIMD	512-bit	256-bit	128-bit
Four-operand FMA	✓Enhanced	✓	✓
Gather/ Scatter	✓Enhanced	✓	
Predicated Operation	✓Enhanced	✓	✓
Math. Acceleration	✓Enhanced	✓Enhanced	✓
HW Barrier*/ Sector Cache*	✓Enhanced	✓Enhanced	✓

* Fujitsu extensions, utilizing the ARM ISA's "implementation-defined" space

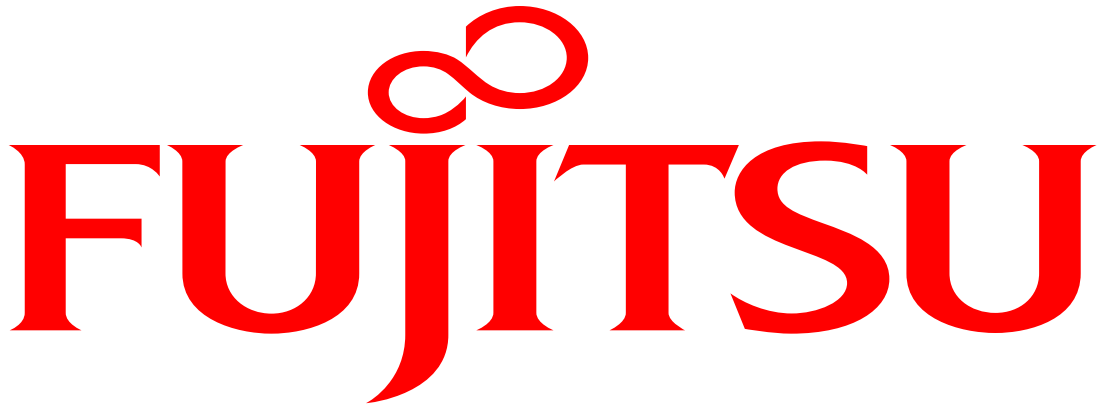
Post-K Development at Fujitsu

FUJITSU

- Fujitsu has 40 years' experience in supercomputers, where we have consistently adopted the most appropriate ISA for each project, such as the HPC extensions for SPARC-V9 called HPC-ACE

- Fujitsu chose to adopt ARMv8-A with SVE in order to best position the Post-K computer to contribute to a wider user base and utilize the assets. This decision was also a natural result of collaboration with ARM on the development of the HPC extensions

- The Post-K computer will be a massively parallel supercomputer system based on Fujitsu's ARM ISA-equipped HPC processor, leveraging Fujitsu's trusted HPC technologies to protect and enhance users' application assets



shaping tomorrow with you

Questions

ARMv8-A SVE
Backup Slides

28

SVE Exception Model

- Low impact on OS & Hypervisor
  - No change to address fault handling
- Manage SVE register context
  - SVE enable per privilege level
  - New “SVE disabled” exception code for lazy context switch
- New system registers
  - ZIDR_EL1: scalable vector ID register is max VL at current Exception level
  - ZCR_ELx: constrains VL at ELx and below (x=1-3)
  - New performance monitor events
- Floating-point control & status
  - Uses FPCR & FPSR as for scalar
- Simple memory model
  - Unaligned accesses
  - Relaxed memory order model
  - Speculative accesses to Device memory not performed, terminate first-fault loads
- Debug unaffected
  - Breakpoints & single-step unchanged
  - Speculative data watchpoints not triggered, terminate first-fault loads

SVE Assembler Syntax Examples

Assembly Language	Meaning
`add z1.d, p2/m, z1.d, z3.d`	64-bit integer destructive vector addition Z1 += Z3 Merging predication under P2
`fadd z1.d, p2/m, z1.d, z3.d`	64-bit DP floating-point destructive vector add Z1 += Z3 Merging predication under P2
`add z1.h, z2.h, z3.h`	16-bit integer constructive vector addition Z1 = Z2 + Z3 Unpredicated
`ands p4.b, p1/z, p2.b, p3.b`	Bitwise AND predicates P4 = P2 & P3, setting cond flags Zeroing predication and setting flags under P1 (3 sources)
`cmpne p1.s, p0/z, z1.s, #3`	32-bit integer vector compare P1 = (Z1 != 3) Zeroing predication and setting flags under P0
`ld1sw z1.d, p3/z, [x0, z2.d]`	Gather load one vector from signed 32-bit words at scalar X0 + 64-bit offsets in Z2 to 64-bit vector Z1 Zeroing predication under P3

DAXPY, unrolled x 4

```
// ----------------------------------------
//      subroutine daxpy(x,y,a,n)
//      real*8 x(n),y(n),a
//      do i = 1,n
//         y(i) = a*x(i) + y(i)
//      enddo
//      end
// ----------------------------------------

         // x0 = &x[0]
         // x1 = &y[0]
         // x2 = &a
         // x3 = &n

        .globl daxpy_
daxpy_:
        ptrue   p0.d
        ld1rd   z1.d, p0/z, [x2,#0]             // z1 = dup(a)

        // When unrolling x 4 it's probably better to increment the two
        // array pointers with one variable predicate and a fixup loop
        // for the final 0-3 iterations.

        // calculate end of "x" array
        ldrsw   x5, [x3,#0]                     // x5 = (int64)n
        add     x5, x0, x5, lsl #3              // xend = &x[n]

        // If there are any active elements in iteration #4 then iterations
        // #1-3 must be all active, so point "x" at iteration #4 (offset 3).

        incb    x0, all, mul #3                 // x = (char *)x + (VL/B * 3)
        b       .Lcheck

.Luloop:
        ld1d    z2.d, p0/z, [x1,#0,mul vl]      // z2 = y[0*vl]
        ld1d    z3.d, p0/z, [x1,#1,mul vl]      // z3 = y[1*vl]
        ld1d    z4.d, p0/z, [x1,#2,mul vl]      // z4 = y[2*vl]
        ld1d    z5.d, p1/z, [x1,#3,mul vl]      // z5 = y[3*vl]

        ld1d    z6.d, p0/z, [x0,#-3,mul vl]     // z6 = x[-3*vl]
        ld1d    z7.d, p0/z, [x0,#-2,mul vl]     // z7 = x[-2*vl]
        ld1d    z8.d, p0/z, [x0,#-1,mul vl]     // z8 = x[-1*vl]
        ld1d    z9.d, p1/z, [x0,#0,mul vl]      // z9 = x[ 0*vl]

        fmla    z2.d, p0/m, z6.d, z1.d          // z2 = y[0] + x[-3] * a
        fmla    z3.d, p0/m, z7.d, z1.d          // z3 = y[1] + x[-2] * a
        fmla    z4.d, p0/m, z8.d, z1.d          // z4 = y[2] + x[-1] * a
        fmla    z5.d, p1/m, z9.d, z1.d          // z5 = y[3] + x[ 0] * a

        st1d    z2.d, p0, [x1,#0,mul vl]        // y[0*vl] = z2
        st1d    z3.d, p0, [x1,#1,mul vl]        // y[1*vl] = z3
        st1d    z4.d, p0, [x1,#2,mul vl]        // y[2*vl] = z4
        st1d    z5.d, p1, [x1,#3,mul vl]        // y[3*vl] = z5

        incb    x0, all, mul #4                 // x += (VL/B * 4)
        incb    x1, all, mul #4                 // y += (VL/B * 4)
.Lcheck: whilelo p1.b, x0, x5                   // x < xend
        b.first .Luloop                         // more to do in iter #4?

        // Rolled fixup loop for final 0-3 vectors.
        // Could switch to incrementing the index but probably
        // not worth the pain.

        decb    x0, all, mul #3                 // adjust x back
        whilelo p1.d, x0, x5                    // x < xend
.Lrloop:
        ld1d    z2.d, p1/z, [x1,#0,mul vl]      // z2 = y[0*vl]
        ld1d    z6.d, p1/z, [x0,#0,mul vl]      // z6 = x[0*vl]
        fmla    z2.d, p1/m, z6.d, z1.d          // z2 = y[0] + x[0] * a
        st1d    z2.d, p1, [x1,#0,mul vl]        // y[0] = z2
        incb    x0, all                         // x += VL/B
        incb    x1, all                         // y += VL/B
        whilelo p1.b, x0, x5                    // x < xend
        b.first .Lrloop                         // more to do?

        ret
```

DAXPY inner loop, unrolled x 4

```
.Luloop:
        ld1d    z2.d, p0/z, [x1, #0, mul vl]      // z2 = y[0*vl]
        ld1d    z3.d, p0/z, [x1, #1, mul vl]      // z3 = y[1*vl]
        ld1d    z4.d, p0/z, [x1, #2, mul vl]      // z4 = y[2*vl]
        ld1d    z5.d, p1/z, [x1, #3, mul vl]      // z5 = y[3*vl]
        ld1d    z6.d, p0/z, [x0, #-3, mul vl]     // z6 = x[-3*vl]
        ld1d    z7.d, p0/z, [x0, #-2, mul vl]     // z7 = x[-2*vl]
        ld1d    z8.d, p0/z, [x0, #-1, mul vl]     // z8 = x[-1*vl]
        ld1d    z9.d, p1/z, [x0, #0, mul vl]      // z9 = x[ 0*vl]
        fmla    z2.d, p0/m, z6.d, z1.d            // z2 = y[0] + x[-3] * a
        fmla    z3.d, p0/m, z7.d, z1.d            // z3 = y[1] + x[-2] * a
        fmla    z4.d, p0/m, z8.d, z1.d            // z4 = y[2] + x[-1] * a
        fmla    z5.d, p1/m, z9.d, z1.d            // z5 = y[3] + x[ 0] * a
        st1d    z2.d, p0, [x1, #0, mul vl]        // y[0*vl] = z2
        st1d    z3.d, p0, [x1, #1, mul vl]        // y[1*vl] = z3
        st1d    z4.d, p0, [x1, #2, mul vl]        // y[2*vl] = z4
        st1d    z5.d, p1, [x1, #3, mul vl]        // y[3*vl] = z5

        incb    x0, all, mul #4                   // x += (VL/B * 4)
        incb    x1, all, mul #4                   // y += (VL/B * 4)
        whilelo p1.b, x0, x5                      // x < xend
        b.first .Luloop                           // more to do in iter #4?
```

strcmp

```
// int strcmp (const char *str1, const char *str2)
//     char c1, c2;
//     do {
//        c1 = *str1++;
//        c2 = *str2++;
//     } while (c1 != '\0' && c1 == c2);
//     return c1 - c2;
strcmp:
    setffr                          // preset FFR
    mov    x2, #0                   // i = 0
.loop:
    ptrue   p7.b
    ldff1b  z0.b, p7/z, [x0, x2]    // vc1 = str1[i]
    ldff1b  z2.b, p7/z, [x1, x2]    // vc2 = str2[i]
    rdffrs  p4.b, p7/z          // read FFR, set flgs
    b.nlast .fault                  // !last=>fault
    // fast path: no fault
    incb    x2                      // i += VL/8
.test:
    cmpeq   p0.b, p7/z, z0.b, #0    // A=(vc1==\0)
    cmpne   p2.b, p7/z, z0.b, z2.b // B=(vc1!=vc2)
    orrs    p0.b, p7/z, p0.b, p2.b  // A|B, set flgs
    b.none  .loop                   // no break

    // return last difference (c1 - c2)
    sub     z0.b, z0.b, z2.b        // vc1 -= vc2
    brkb    p0.b, p7/z, p0.b        // until(A|B)
    lasta   b2, p0, z0.b            // b2=vc1[last]
    smov    w0, b2                  // return(int)b2
    ret

.fault:
    // slow path: fault detected
    incp    x2, p4.b                // i += cnt(FFR)
    mov     p7.b, p4.b              // use FFR
    setffr                          // reset FFR
    b       .test
```