Arm C Language Extensions for SVE

Compiler Intrinsics for SVE
Arm C Language Extensions

Intrinsics and other features for supporting Arm features in C and C++

• ACLE extends C/C++ with Arm-specific features
  • Predefined macros: __ARM_ARCH_ISA_A64, __ARM_BIG_ENDIAN, etc.
  • Intrinsic functions: __clz(uint32_t x), __cls(uint32_t x), etc.
  • Data types: SVE, NEON and FP16 data types

• ACLE for SVE enables VLA programming with ACLE
  • Nearly one intrinsic per SVE instruction
  • Data types to represent the size-less vectors used for SVE intrinsics

• Intended for users that...
  • Want to hand-tune SVE code
  • Want to adapt or hand-optimize applications and libraries
  • Need low-level access to Arm targets
How to use ACLE

• Include the headers you need
  • arm_acle.h → for core ACLE
  • arm_fp16.h → to add scalar FP16 arithmetic
  • arm_neon.h → to add NEON intrinsics and data types
  • arm_sve.h → to add SVE intrinsics and data types

• Each of those require certain features at the compilation target
  • arm_fp16.h → Your target platform needs to support FP16 (-march=armv8-a+fp16)
  • arm_neon.h → Your target platform need to support NEON (-march=armv8-a+simd)
  • arm_sve.h → Your target platform need to support SVE (-march=armv8-a+sve)
SVE ACLE

SVE Arm C Language Extensions – aka C *intrinsics*

```c
#include <arm_sve.h>
```

- VLA Data types:
  - `svfloat64_t`, `svfloat16_t`, `svuint32_t`, ...

- Predication:
  - Merging: `_m`
  - Zeroing: `_z`
  - Don’t care: `_x`
  - Predicate type: `svbool_t`

- Use *C11 generics* for function overloading.

- Intrinsics are **not 1-1 with the ISA**.

**Examples**

```c
svfloat32_t
svadd[_n_f32]_z(svbool_t pg,
                 svfloat32_t op1,
                 float32_t op2);
```

```c
svfloat16_t
svsqrt_m(svfloat16_t inactive,
          svbool_t pg,
          svfloat16_t op)
```
Vectorizing a scalar loop with ACLE

Original Code

```c
a[:] = 2.0 * a[:]
```

```c
for (int i=0; i < N; ++i) {
    a[i] = 2.0 * a[i];
}
```

128-bit **NEON** vectorization with ACLE

```c
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
    float32x4_t va = vld1q_f32(&a[i]);
    va = vmulq_n_f32(va, 2.0);
    vst1q_f32(&a[i], va)
}

// drain loop
for (; i < N; ++i)
    a[i] = 2.0 * a[i];
```

This is NEON, not SVE!
Vectorizing a scalar loop with ACLE

\[ a[:] = 2.0 \times a[:] \]

**SVE vectorization**

```c
for (int i = 0 ; i < N; i += ????????)
{
    svbool_t Pg = svwhilelt_b32(i, N);
    svfloat32_t va = svld1(Pg, &a[i]);
    va = svmul_x(Pg, va, 2.0);
    svst1(Pg, &a[i], va);
}
```

**128-bit NEON vectorization**

```c
int i;

// vector loop
for (i=0; (i<N-3) && (N&~3); i+=4) {
    float32x4_t va = vld1q_f32(&a[i]);
    va = vmulq_n_f32(va, 2.0);
    vst1q_f32(&a[i], va)
}

// drain loop
for (; i < N; ++i)
    a[i] = 2.0 * a[i];
```
Vectorizing a scalar loop with ACLE

\[
a[i] = 2.0 \times a[i]
\]

**SVE vectorization**

```c
for (int i = 0; i < N; i += ?????????) { 

// Calculation

svfloat32_t va = svld1(Pg, &a[i]);
va = svmul_x(Pg, va, 2.0);
svst1(Pg, &a[i], va);

} 
```

**SVE vectorization with fewer branches**

```c
svcntw();
svbool_t all = svptrue_b32();
svbool_t Pg;
for (int i = 0;

    svptest_first(all,
        Pg = svwhilelt_b32(i, N));
    i += svcntw();

    

} 

svfloat32_t va = svld1(Pg, &a[i]);
va = svmul_x(Pg, va, 2.0);
svst1(Pg, &a[i], va);
```

© 2019 Arm Limited
ACLE for SVE Cheat Sheet

• Vector types
  • sv<datatype><datasize>_t
    • svfloat32_t
    • svint8_t
    • svuint16_t

• Predicate types
  • svbool_t

• Functions
  • svbase[disambiguator][type0][type1]…[predication]
  • base is the lower-case name of an SVE instruction
  • disambiguator distinguishes between different forms of a function
  • typeN lists the types of vectors and predicates
  • predication describes the inactive elements in the result of a predicated operation

svfloat64_t svl1d1_f64(svbool_t pg, const float64_t *base)
svbool_t svwhilelt_b8(int64_t op1, int64_t op2)
svuint32_t svmla_u32_z(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
svuint32_t svmla_u32_m(svbool_t pg, svuint32_t op1, svuint32_t op2, svuint32_t op3)
02_ACLE/01_vecadd
See README.md for details

GCC 9.3

gcc --version
 gcc (GCC) 9.3.0

gcc -fopt-info=all-vec -Ofast \
    -mcpu=native vec_add_acle.c

vec_add_acle.c:22:10: fatal error: arm_sve.h: No such file or directory
  22 | #include <arm_sve.h>
     | ^~~~~~~~~~~
compilation terminated.

GCC 11

gcc --version
 gcc (GCC) 11.0.0 20201025 (experimental)

gcc -fopt-info=all-vec -Ofast \
    -mcpu=native vec_add_acle.c

# No errors
vec_add_acle_arm.exe

whilelo p0.s, x8, x9
ld1w { z0.s }, p0/z, [x1, x8, lsl #2]
ld1w { z1.s }, p0/z, [x2, x8, lsl #2]
fadd z0.s, p0/m, z0.s, z1.s
st1w { z0.s }, p0, [x0, x8, lsl #2]
incw x8
cmp x8, #256, lsl #12
b.lo #-28 <vec_svadd_m+0x20>
ret

vec_add_arm.exe

whilelo p0.s, xzr, x9
b #24 <vec_add+0x60>
... (six nop instructions)
ld1w { z0.s }, p0/z, [x1, x8, lsl #2]
ld1w { z1.s }, p0/z, [x2, x8, lsl #2]
fadd z0.s, z1.s, z0.s
st1w { z0.s }, p0, [x0, x8, lsl #2]
incw x8
cmp x8, #256, lsl #12
b.lo #-28 <vec_svadd_m+0x20>
ret
Working with SVE Instructions
**SAXPY**

```assembly
subroutine saxpy(x,y,a,n)
real*4 x(n),y(n),a
do i = 1,n
    y(i) = a*x(i) + y(i)
enddo
```

---

**Key Operations**

- `whilelt` constructs a **predicate** (p0) to dynamically map vector operations to vector data
- `incw` increments a scalar register (x4) by the number of float elements that fit in a vector register
- No drain loop! Predication handles the remainder

---

**Scalar [-march=armv8-a]**

```assembly
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_: 
    ldrsw x3, [x3] // x3=*n
    mov x4, #0 // x4=i=0
    ldr s0, [x2] // d0=*a
    b .latch
.loop:
    ldr s1, [x0,x4,1ls1 2] // s1=x[i]
    ldr s2, [x1,x4,1ls1 2] // s2=y[i]
    fmadd s2, s1, s0, s2 // s2+=x[i]*a
    str s2, [x1,x4,1ls1 2] // y[i]=s2
    add x4, x4, #1 // i+=1
.latch:
    cmp x4, x3 // i<n
    b.lt .loop // more to do?
    ret
```

---

**SVE [-march=armv8-a+sve]**

```assembly
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy_: 
    ldrsw x3, [x3] // x3=*n
    mov x4, #0 // x4=i=0
    whilelt p0.s, x4, x3 // p0=while(i++<n)
    ld1rw z0.s, p0/z, [x2] // p0:z0=bcast(*a)
    .loop:
    ld1rw z1.s, p0/z, [x0,x4,lsl 2] // p0:z1=x[i]
    ld1w z2.s, p0/z, [x1,x4,1ls1 2] // p0:z2=y[i]
    fmadd z2.s, s1, s0, z2 // p0?z2+=x[i]*a
    st1w z2.s, p0, [x1,x4,1ls1 2] // p0?y[i]=z2
    incw x4 // i+=(VL/32)
    .latch:
    whilelt p0.s, x4, x3 // p0=while(i++<n)
    b.first .loop // more to do?
    ret
```
How do you count by vector width?

No need for multi-versioning: one increment for all vector sizes

```
ld1w z1.s, p0/z, [x0,x4,ls1 2]  // p0:z1=x[i]
ld1w z2.s, p0/z, [x1,x4,ls1 2]  // p0:z2=y[i]
fmla z2.s, p0/m, z1.s, z0.s     // p0?z2+=x[i]*a
st1w z2.s, p0, [x1,x4,ls1 2]    // p0?y[i]=z2
incw x4                       // i+=(VL/32)
```

“Increment x4 by the number of 32-bit lanes (w) that fit in a VL.”
VLA Increment and Count

\begin{align*}
\text{inc\text{\texttt{b}}} & \quad \text{x0, mul3, mul} \ #2 \\
\text{inc\text{\texttt{d}}} & \quad \text{z0.d, pow2} \\
\text{inc\text{\texttt{p}}} & \quad \text{x0, p0.s} \\
\text{inc\text{\texttt{p}}} & \quad \text{z0.h, p0} \\
\text{cnt\text{\texttt{w}}} & \quad \text{x0} \\
\text{cnt\text{\texttt{p}}} & \quad \text{x0, p0, p1.s.} \\
\end{align*}

\begin{align*}
x0 & \begin{aligned} & += 2 \times (\text{largest mul3} \leq \text{VL.b}) \\
& \text{each lane} += \text{largest pow2} \leq \text{VL.d} \\
& \text{x0} += \# \text{active lanes} \\
& \text{each vector lane} += \# \text{active lanes of p} \\
\end{aligned} \\
x0 & = \text{VL.s} \\
x0 & = \# \text{active lanes of (p0} \&\& \text{p1.s)}
\end{align*}
Predicates: Active Lanes vs Inactive Lanes

Predicate registers track lane activity

- 16 predicate registers (P0-P15)
- 1 predicate bit per 8 vector bits (lowest predicate bit per lane is significant)
- On load, active elements update the destination
- On store, inactive lanes leave destination unchanged (p0/m) or set to 0’s (p0/z)
**Predicate Condition Flags**

**SVE is a *predicate-centric* architecture**

- Predicates support complex nested conditions and loops.
- Predicate generation also sets condition flags.
- Reduces vector loop management overhead.

**Overloading the A64 NZCV condition flags**

<table>
<thead>
<tr>
<th>Condition Test</th>
<th>A64 Name</th>
<th>SVE Alias</th>
<th>SVE Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z=1</td>
<td>EQ</td>
<td>NONE</td>
<td>No active elements are true</td>
</tr>
<tr>
<td>Z=0</td>
<td>NE</td>
<td>ANY</td>
<td>Any active element is true</td>
</tr>
<tr>
<td>C=1</td>
<td>CS</td>
<td>NLAST</td>
<td>Last active element is not true</td>
</tr>
<tr>
<td>C=0</td>
<td>CC</td>
<td>LAST</td>
<td>Last active element is true</td>
</tr>
<tr>
<td>N=1</td>
<td>MI</td>
<td>FIRST</td>
<td>First active element is true</td>
</tr>
<tr>
<td>N=0</td>
<td>PL</td>
<td>NFRST</td>
<td>First active element is not true</td>
</tr>
<tr>
<td>C=1 &amp; Z=0</td>
<td>HI</td>
<td>PMORE</td>
<td>More partitions: some active elements are true but not the last one</td>
</tr>
<tr>
<td>C=0</td>
<td>Z=1</td>
<td>LS</td>
<td>PLAST</td>
</tr>
<tr>
<td>N=V</td>
<td>GE</td>
<td>TCONT</td>
<td>Continue scalar loop</td>
</tr>
<tr>
<td>N!=V</td>
<td>LT</td>
<td>TSTOP</td>
<td>Stop scalar loop</td>
</tr>
</tbody>
</table>
Initialization when vector length is unknown

- Vectors cannot be initialized from compile-time constant, so...
  - INDEX Zd.S,#1,#4 : Zd = [1, 5, 9, 13, 17, 21, 25, 29]

- Predicates cannot be initialized 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 64b dwords in vector
  - WHILELT Pd.D, Xi, Xe : next iteration predicate Pd = [while i++ < e]

- Vectors stores to stack must be dynamically allocated and indexed, so...
  - ADDVL SP, SP, #-4 : decrement stack pointer by (4*VL)
  - STR Zi, [SP,#3,MUL VL] : store vector Z1 to address (SP+3*VL)
SVE includes a rich set of horizontal operations

The operation happens across the lanes of a vector

- Addition, maximum, minimum, bitwise AND, OR, XOR ...

\[
\text{SADDV } <Dd>, <Pg>, <Zn.T>
\]
C code without intrinsics

Compile with `-march=armv8-a+sve` or `–mcpu=native` if your CPU has SVE

double ddot (double *a, double *b, int n) {
    double sum = 0.0;
    for ( int i = 0; i < n; i++ ) {
        sum += a[i] * b[i];
    }
    return sum;
}

• Accumulates in a scalar
• No horizontal reductions
C code with SVE intrinsics
Vector accumulator; horizontal reduction

```c
#include <arm_sve.h>

double ddot (double *a, double *b, int n) {
    svfloat64_t svsum = svdup_f64(0.0);
    svbool_t pg;
    svfloat64_t sva, svb, svsum;
    for (int i = 0; i < n; i += svcntd()) {
        pg = svwhilelt_b64(i, n);
        sva = svld1_f64(pg, &a[i]);
        svb = svld1_f64(pg, &b[i]);
        svsum = svmla_f64_m(pg, svsum, sva, svb);
    }
    return svaddv_f64(svptrue_b64(), svsum);
}
```

```assembly
cmp     w2, #1
b.lt    #60 <ddot+0x40>
mov     w8, wzr
cntd     x9
mov     z0.d, #0
whilelt  p0.d, w8, w2
sxtw    x10, w8
1d1d    { z1.d }, p0/z, [x0, x10, lsl #3]
1d1d    { z2.d }, p0/z, [x1, x10, lsl #3]
add     w8, w8, w9
cmp     w8, w2
fmla     z0.d, p0/m, z1.d, z2.d
b.lt     #-28 <ddot+0x14>
ptrue    p0.d
faddv   d0, p0, z0.d
ret
mov     z0.d, #0
ptrue    p0.d
faddv   d0, p0, z0.d
ret
```
Most vector instructions operate on a lane-by-lane basis

SVE includes a rich set of horizontal operations where the operation happens across the lanes of a vector

Examples of such operations are addition, maximum, minimum, bitwise AND, OR, XOR

```c
suint32_t svmaxv(svbool_t pg, svuin32_t op) op
```

![Diagram](image)
02_ACLE/02_hreduce

See README.md for details

Reduce to scalar on every iteration

```c
uint32_t max_scalar(uint32_t array[SIZE])
{
    uint32_t max = 0.0;
    uint32_t vl = svcntw();
    svbool_t p32_all = svtrue_b32();
    for (int i=0; i<SIZE; i+=vl) {
        // load array elements from memory
        svuint32_t vec = svld1(p32_all, &array[i]);
        // get max within vector
        uint32_t vecmax = svmaxv(p32_all, vec);
        if (vecmax > max) {
            max = vecmax;
        }
    }
    return max;
}
```

Reduce to vector; final reduction to scalar

```c
uint64_t max(uint64_t array[SIZE])
{
    // initialize max vector with zeros
    svuint64_t max = svdup_u64(0);
    // get number of 64-bit elements in vector
    uint64_t vl = svcntd();
    // all true mask - assuming no partial vector mask needed for simplicity
    svbool_t p64_all = svtrue_b64();
    for (int i=0; i<SIZE; i+=vl) {
        // load array elements from memory
        svuint64_t vec = svld1(p64_all, &array[i]);
        // get max between loaded vector to max
        max = svmax_m(p64_all, max, vec);
    }
    // return max across values within the vector
    return svmaxv(p64_all, max);
}
```
arm

Low-precision Dot Product with Widening
• Dot product for low-precision value with widening

\[
\text{svint32}_t\ \text{svdot}(\text{svint8}_t\ \text{src1}, \ \text{svint8}_t\ \text{src2})
\]

\[
\text{svint64}_t\ \text{svdot}(\text{svint16}_t\ \text{src1}, \ \text{svint16}_t\ \text{src2})
\]
dotprod_acle_arm.exe

dotprod_arm.exe
Vector Partition with the First-faulting Register (FFR)
Vector Partitioning: when a vector spans a protected region

With VLA, we don’t always know what data we may touch

- **Software-managed speculative vectorisation**
  - Create sub-vectors (partitions) in response to data and dynamic faults

- **First-fault load allows access to safely cross a page boundary**
  - First element is mandatory but others are a “speculative prefetch”
  - Dedicated FFR predicate register indicates successfully loaded elements

- **Allows uncounted loops with break conditions**
  - Load data using first-fault load
  - Create a *before-fault* partition from FFR
  - Test for break condition
  - Create a *before-break* partition from condition predicate
  - Process data within partition
  - Exit loop if break condition was found.
Vectorizing `strlen`

A vector load could result in a segfault if the vector spans protected memory.

Source Code

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

Scalar [ -march=armv8-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
```
Fault-tolerant Speculative Vectorization

• Some loops have dynamic exit conditions that prevent vectorization
  • E.g. the loop breaks on a particular value of the traversed array

<table>
<thead>
<tr>
<th>‘S’</th>
<th>‘C’</th>
<th>‘2’</th>
<th>‘0’</th>
<th>‘2’</th>
<th>‘0’</th>
<th>‘0’</th>
</tr>
</thead>
</table>

• The access to unallocated space **does not trap** if it is not the first element
  • Faulting elements are stored in the first-fault register (FFR)
  • Subsequent instructions are predicated using the FFR information to operate only on successful element accesses
Vectorizing `strlen`
Partitioning off protected memory

**SVE** [-march=armv8-a+sve]

```assembly
strlen:
    mov x1, x0       // e=s
    ptrue p0.b      // p0=true

.loop:
    setffr          // ffr=true
    lddff1b z0.b, p0/z, [x1]  // p0:z0=ldff(e)
    rddff 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 active=>!break
    sub x0, x1, x0 // return e-s
    ret
```

**Scalar** [-march=armv8-a]

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

_strlen:
    mov    x1, x0
    ptrue  p0.b

.loop:
    setffr
    ldff1b z0.b, p0/z, [x1]
    rdffrs p0.b, p2/z
    b.nlast 2f

    /* First fault did not fail: the whole vector is valid. Avoid depending on the contents of FFR beyond the branch. */
    incb x1, all
    cmpeq p1.b, p2/z, z0.b, 0
    b.none 0b
    decb x1, all

    /* Zero found. Select the bytes before the first and count them. */
    brkB p0.b, p2/z, p1.b
    incp x1, p0.b
    mov x0, x1
    ret

    /* First fault failed: only some of the vector is valid. Perform the comparison only on the valid bytes. */
    cmpeq p1.b, p0/z, z0.b, 0
    b.any 1b

    /* No zero found. Re-init FFR, increment, and loop. */
    setffr
    incp x1, p0.b
    b 0b

Suboptimal implementation

setffr      /* initialize FFR */
ptrue p2.b  /* all ones; loop invariant */
mov x1, 0   /* initialize length */

/* Read a vector's worth of bytes, stopping on first fault. */
0:
  ldff1b z0.b, p2/z, [x0, x1]
  rdffrs p0.b, p2/z
  b.nlast 2f

0b

/* First fault did not fail: the whole vector is valid. Avoid depending on the contents of FFR beyond the branch. */
1:
  brkB p0.b, p2/z, p1.b
  incp x1, p0.b
  mov x0, x1
  ret

/* Zero found. Select the bytes before the first and count them. */
2:
  cmpeq p1.b, p0/z, z0.b, 0
  b.any 1b

/* First fault failed: only some of the vector is valid. Perform the comparison only on the valid bytes. */

/* No zero found. Re-init FFR, increment, and loop. */
setffr
incp x1, p0.b
b 0b
Initialize predicate
Initialize FFR to all true
Initialize loop counter to 0
-BEGIN LOOP BODY -
Load str1 with first-fault
Load str2 with first-fault
Read FFR and place active elements in predicate p7
If first active element is not true handle string remainder
Increment loop counter by number of byte lanes in SVE regiser
Compare full vector of str1 chars to 0 (look for end of string)
Compare full vector of str1 to str2
-BEGIN LOOP CONDITION -
Bitwise OR of predicates to check for differences and str end
Predicate condition: no active elements are true (keep looping)
-BEGIN reminder strcmp -
Increment loop counter by number of true predicate elements
Reset FFR
Compare partial vector of str1 chars to 0
Compare partial vector of str1 to str2
Complex arithmetic with FMLA or FCMLA

Complex multiply:
\((a+ib).(c+id) = (ac-bd)+i(ad+bc)\)

FCMLA in SVE works for 4 rotations:
- 0, 90, 180 and 270

Complex multiply-add needs a pair of instructions

svcmla_z(pg, src3, src1, src2, 0);
svcmla_z(pg, src3, src1, src2, 90);
02_ACLE/04_dotprod_complex

See README.md for details

ACLE code using FMLA

```c
for (int i=0; i<SIZE; i+=v1) {
    svfloat32x2_t va = svld2(p32_all, (float32_t*)&a[i]);
    svfloat32x2_t vb = svld2(p32_all, (float32_t*)&b[i]);
    svfloat32x2_t vc = svld2(p32_all, (float32_t*)&c[i]);
    vc.v0 = svmla_m(p32_all, vc.v0, va.v0, vb.v0); //c.re += a.re * b.re
    vc.v1 = svmla_m(p32_all, vc.v1, va.v1, vb.v1); //c.im += a.im * b.im
    svst2(p32_all, (float32_t*)&c[i], vc);
}
```

ACLE code using FCMLA

```c
for (int i=0; i<SIZE; i+=v1/2) {
    svfloat32_t va = svld1(p32_all, (float32_t*)&a[i]);
    svfloat32_t vb = svld1(p32_all, (float32_t*)&b[i]);
    svfloat32_t vc = svld1(p32_all, (float32_t*)&c[i]);
    vc.v0 = svcmla_m(p32_all, vc, va, vb, 0); //c += a * b
    vc.v1 = svcmla_m(p32_all, vc, va, vb, 90);  //c += a * 90'
    svst1(p32_all, (float32_t*)&c[i], vc);
}
```
Gather/Scatter Operations

- Enable vectorization of codes with non-adjacent accesses on adjacent lanes
- Examples:
  - Outer loop vectorization
  - Strided accesses (larger than +1)
  - Random accesses
- Performance implementation dependent
  - Worst case one separate access per element
- LD1D $\langle Zt \rangle.D, Ps/Z [\langle Xn \rangle, \langle Zm \rangle.D]$
Array of Structures vs. Structure of Arrays

typedef struct {
    uin64_t num_projects;
    float caffeine;
    bool vim_nemacs;
} Programmer_t;

Programmer_t programmers[N];

typedef struct {
    uin64_t num_projects[N];
    float caffeine[N];
    bool vim_nemacs[N];
} Programmer_t;

Programmer_t programmers;
• Use SVE vector tuples to access structure
• Performance depends on u-arch and memory system

typedef struct {
    uint32_t x;
    uint32_t y;
} Particle_t;

Particle_t particles[N];

Structure loads:
LD2W {<Zt>.H, <Zt+1>.H}, P0/Z, [X0]
02_ACLE/05_gather
See README.md for details

Original code

```c
typedef struct {
    int32_t x;
    int32_t y;
} Particle_t;

void move(Particle_t p[SIZE], int32_t x, int32_t y) {
    for (int i=0; i<SIZE; i++) {
        p[i].x += x;
        p[i].y += y;
    }
}
```

Use Vector Tuples

```c
void move(Particle_t p[SIZE], int32_t x, int32_t y) {
    uint32_t v1 = svcntw();
    svbool_t p32_all = svtrue_b32();
    for (int i=0; i<SIZE; i+=v1) {
        svint32x2_t vp = svld2(p32_all, (int32_t*)&p[i]);
        vp.v0 = svadd_m(p32_all, vp.v0, x);
        vp.v1 = svadd_m(p32_all, vp.v1, y);
        svst2(p32_all, (int32_t*)&p[i], vp);
    }
}
```
SVE Load Replicate and GEMM

Neat theory; YMMV
GEMM

\[ c_{ij} += \sum_{k=0}^{K-1} a_{ik} b_{kj} \]

Diagram:
- Matrix A with dimensions \( M \times K \)
- Matrix B with dimensions \( K \times N \)
- Matrix C with dimensions \( K \times M \)
- Formula: \( c_{ij} += \sum_{k=0}^{K-1} a_{ik} b_{kj} \)
GEMM – step 1 – vectorize along the columns

\[ c_{ij} += \sum_{k=0}^{K-1} a_{ik} b_{kj} \]
GEMM – step 2 – unroll along rows

\[ c_{ij} += \sum_{k=0}^{K-1} a_{ik} b_{kj} \]
DGEMM kernel (NEON, 128-bit, 24 accumulators + 4 + 3)

Each k iteration:
- 4 vector loads from A
- 3 vector loads from B
- 24 indexed FMLA in C

\[
\begin{align*}
\text{fmla } & C_{20}.2d, B_0.2d, A_{23}.d[0] \\
\text{fmla } & C_{21}.2d, B_1.2d, A_{23}.d[0] \\
\text{fmla } & C_{22}.2d, B_2.2d, A_{23}.d[0] \\
\text{fmla } & C_{30}.2d, B_0.2d, A_{23}.d[1] \\
\text{fmla } & C_{31}.2d, B_1.2d, A_{23}.d[1] \\
\text{fmla } & C_{32}.2d, B_2.2d, A_{23}.d[1]
\end{align*}
\]
SVE load Replicate Quadword instructions: \texttt{LD1RQ [BHWD]}

double *A;

\begin{align*}
\text{Mem.} & \quad 128 \text{ bit} & \quad 128 \text{ bit} \\
\text{LD1D \{ Z0.D \}, P0/Z, [X0]} & \\
& Z0 \\
\text{LD1RQD \{ Z1.D \}, P0/Z, [X0]} & \\
& Z1
\end{align*}
DGEMM kernel (SVE, \textit{LEN} x 128-bit, 24 accumulators + 4 + 3)

Each \( k \) iteration:
- 4 vector \textit{load replicate} from \( A \)
- 3 vector loads from \( B \)
- 24 \textit{indexed FMLA} in \( C \)

\begin{align*}
\text{fmla } & C_{20}.d, B_0.d, A_{23}.d[0] \\
\text{fmla } & C_{21}.d, B_1.d, A_{23}.d[0] \\
\text{fmla } & C_{22}.d, B_2.d, A_{23}.d[0] \\
\text{fmla } & C_{30}.d, B_0.d, A_{23}.d[1] \\
\text{fmla } & C_{31}.d, B_1.d, A_{23}.d[1] \\
\text{fmla } & C_{32}.d, B_2.d, A_{23}.d[1]
\end{align*}
# DGEMM: SVE vs NEON

<table>
<thead>
<tr>
<th>Each k iteration</th>
<th>Bytes in one k iteration</th>
<th>Total C area computed</th>
<th>SVE/NEON A and B data reads for same C area</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SVE</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 4 vector <em>load replicate</em> from A</td>
<td>4 x 128b + 3 x LEN x 128b</td>
<td>24 x 128b x LEN</td>
<td>SVE/NEON = (\frac{4 + 3 \times \text{LEN}}{7})</td>
</tr>
<tr>
<td>• 3 vector loads from B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>NEON</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• 4 vector loads from A</td>
<td>7 x 128b</td>
<td>24 x 128b</td>
<td></td>
</tr>
<tr>
<td>• 3 vector loads from B</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DGEMM: SVE more memory efficient than LEN times NEON

![Bar chart showing SVE and NEON vector bits efficiency comparison. The x-axis represents SVE vector bits (128, 256, 384, 512, 640, 768, 896, 1024, 2048), and the y-axis represents the ratio of SVE to NEON efficiency. The chart indicates that SVE is more efficient across different vector bit lengths.](chart.png)
03_SVE: Micro-examples for individual SVE instructions

Based on https://developer.arm.com/documentation/dai0548/latest

Using the Micro-examples

- PDF document describes each example
- Directory name indicates section in the PDF
- Includes both SVE and SVE2 examples
- May need ArmIE to run some examples, e.g. SVE2 examples require ArmIE to run on A64FX
SVE Resources
http://developer.arm.com/hpc

• Porting and Optimizing Guides
  • For SVE: https://developer.arm.com/docs/101726/0110
  • For Arm in general: https://developer.arm.com/docs/101725/0110

• The SVE Specification and Arm Instruction Reference
  • Arm Architecture Reference Manual Supplement, SVE for ARMv8-A
  • https://developer.arm.com/docs/ddi0596/i/a64-sve-instructions-alphabetic-order

• ACLE References and Examples
  • ACLE for SVE: https://developer.arm.com/docs/100987/latest
  • Worked examples: A Sneak Peek Into SVE and VLA Programming
  • Optimized machine learning: Arm SVE and Applications to Machine Learning