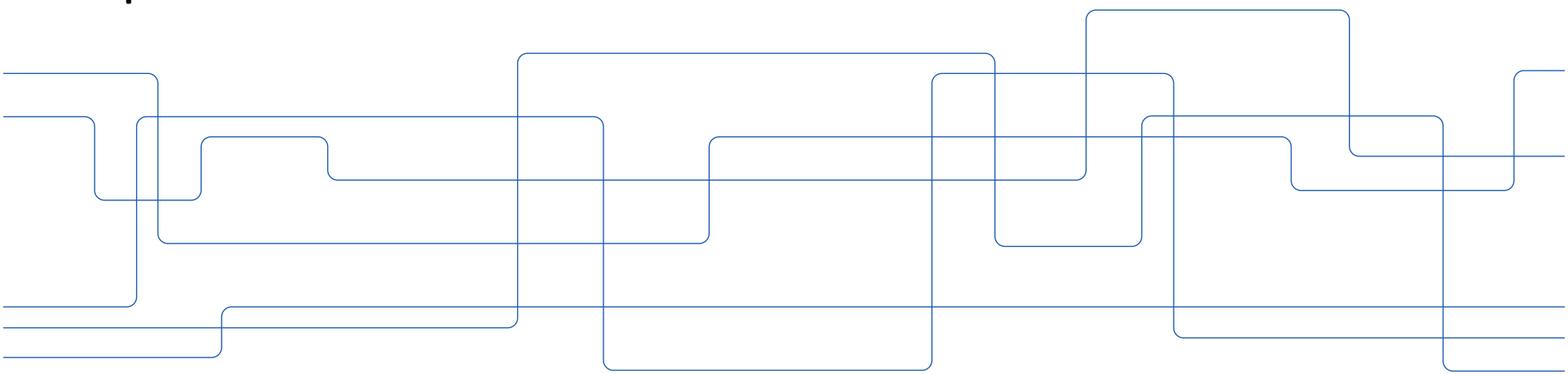
Assessing the State of Autovectorization Support based on SVE

Bine Brank (FZJ), Dirk Pleiter (KTH)

Embracing Arm for HPC at Cluster 2022

September 2022



Overview

Content

SIMD Instructions

- Definition: Instructions where a single instruction triggers operations that process in each clock cycle a data tuple
- Examples:
  - x86: SSE, AVX, AVX2, AVX512
  - POWER/PowerPC: AltiVec, VSX
  - Arm: NEON, Scalable Vector Extension (SVE)
- Special feature of Arm's SVE: Vector Length Agnostic (VLA)
  - Operand width $= 128, 256, \ldots, 2048$ bit

SIMD Programming

- **Inline assembler**
  - Advantage(s)
    - Maximum control on code generation
  - Disadvantage(s)
    - Very complex and non-portable code with support from compiler
    - Register spilling may become unavoidable
- **Intrinsics**
  - Advantage(s)
    - Strong control on code generation with support from compiler
  - Disadvantage(s)
    - Still complex and non-portable code
- **Compiler auto-vectorization**
  - Advantage(s)
    - Portable
  - Disadvantage(s)
    - Risk of compiler not detecting vectorization opportunities

Research Questions

- How well are the auto-vectorizers of today's compilers performing when using SVE?
- What is the impact of auto-vectorization on state-of-the-art hardware supporting SVE?
- Does VLA feature impact the auto-vectorizer?
- Does SVE limit the vectorization opportunities?

Content

TSVC2 Benchmark Suite

- TSVC was developed in the 1980s to test Fortran compilers using synthetic loops
- TSVC2: port to C extending suite to 151 loops
- Systematic testing for different strategies:
  - Dependence testing
  - Statement reordering
  - Loops interchange
  - Scalar expansion
- Single-precision arithmetics, i.e. $N_{\text{lane}} = 16$ for the A64FX

TSVC2 Benchmark Suite Example

- **Easy**: loop s000

```
1 for (int i = 0; i < LEN_1D; i++) {
2   a[i] = b[i] + 1;
3 }
```

- **Difficult**: loop s111

```
1 for (int i = 1; i < LEN_1D; i += 2) {
2     a[i] = a[i - 1] + b[i];
3 }
```

- **Impossible** (at least for today's compilers): loop s332

```
1  index = -2; value = -1.;
2  for (int i = 0; i < LEN_1D; i++) {
3      if (a[i] > t) {
4          index = i;
5          value = a[i];
6          goto L20;
7      }
8  }
9  L20:
10     chksum = value + (real_t) index;
```

Compilers

- Used compilers
  - **GNU C Compiler (GCC)**
    - Version 11.1.0
  - **Arm Compiler for Linux (ACfL)**
    - Version 22.0.1
  - **Fujitsu Compiler FCC**
    - Version 4.7 (Clang version)
- Key options enabled to
  - Enable SVE support
  - Allow to break strict compliance with IEEE 754
  - Enable fused multiply-add instructions

Evaluation Approach

- For all TSVC2 loops a manual vectorization using intrinsics was attempted
- Execution times measured on A64FX nodes of Ookami (Stony Brook)
- Measured execution times:

  Δt_{scalar}: Scalar version

  Δt_{vec}: Auto-vectorized version

  $\Delta t_{\text{intrinsic}}$: Manually vectorized version
- Speed-up

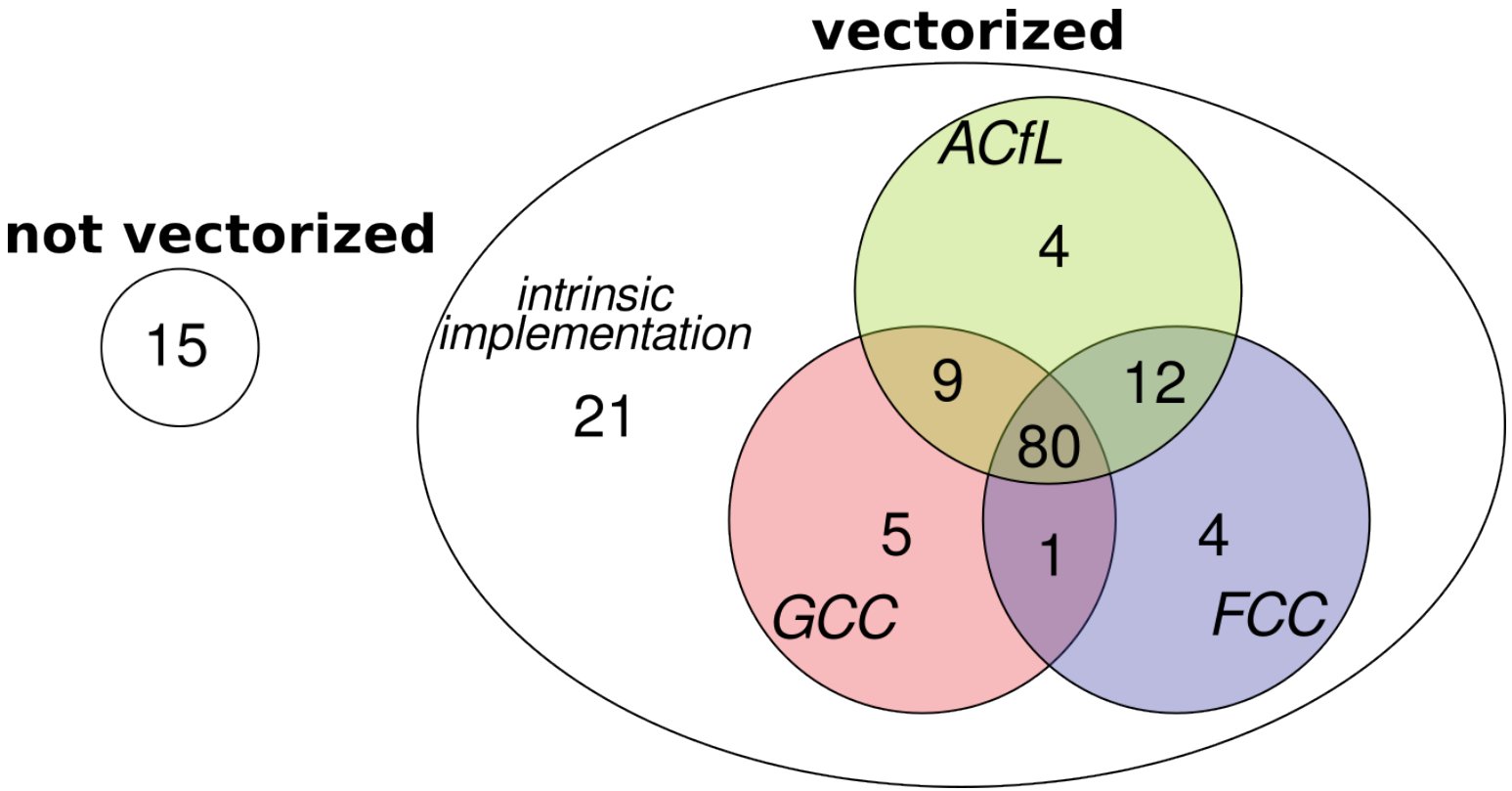
$$\eta = \frac{\Delta t_{\text{scalar}}}{\Delta t_{\text{vec}}}$$

- Efficiency

$$\epsilon = \frac{\eta}{N_{\text{lane}}}$$

Content

Vectorization Overview



Timing and Speed-up Overview

	GCC	ACfL	FCC
ΔT_{scalar}	7.6	8.8	6.1
ΔT_{vec}	3.7	2.9	2.7
$\Delta T_{\text{intrinsic}}$	1.6	1.7	1.7

	GCC	ACfL	FCC
vectorized	94	93(+11)	97
$\eta \geq 1.15$	86	83	76
$\eta \geq 2$	81	77	73
$\eta \geq 4$	72	67	51
$\eta \geq 8$	42	46	23

Observations:

- Overall good speed-up observed
- Best auto-vectorisation speed-up achieved for FCC
- Auto-vectorisation speed-up per kernel is vastly different
- Use of intrinsics allows for significant improvement

Exploited SVE Features

- Arithmetic instructions
- Contiguous elements load/store (`ld1`/`st1`)
- Contiguous structures load/store (`ld2`/`st2`)
- Gather load/store
- Zip instructions
- Reduction instructions (e.g., `fadda`)
- Predicate registers

Example Loop s124: C

```
1  j = -1;
2  for (int i = 0; i < LEN_1D; i++) {
3      if (b[i] > (real_t)0.) {
4          j++;
5          a[j] = b[i] + d[i] * e[i];
6      } else {
7          j++;
8          a[j] = c[i] + d[i] * e[i];
9      }
10 }
```

Possible vectorized implementations:

- Compute both results for `a[j]` and use predicates to select the correct result depending on `b[i]`
- Select depending on `b[i]` either `b[i]` or `c[i]` as input for computing `a[j]`

Example Loop `s124`: GCC Assembler

```
1  .L519:
2          ld1w    z0.s, p0/z, [x1, x0, lsl 2]
3          ld1w    z1.s, p0/z, [x25, x0, lsl 2]
4          ld1w    z2.s, p0/z, [x22, x0, lsl 2]
5          ld1w    z3.s, p0/z, [x21, x0, lsl 2]
6          fcmle   p1.s, p2/z, z0.s, #0.0
7          fmla    z0.s, p2/m, z1.s, z2.s
8          movprfx z0.s, p1/m, z3.s
9          fmla    z0.s, p1/m, z1.s, z2.s
10         st1w    z0.s, p0, [x19, x0, lsl 2]
11         add     x0, x0, x23
12         whilelo p0.s, w0, w20
13         b.any   .L519
```

- 2× `fmla` instructions, one predicated

Example Loop `s124`: ACfL Assembler

```
1  .LBB168_2:
2          ld1w    { z0.s }, p0/z, [x20, x8, lsl #2]
3          ld1w    { z1.s }, p0/z, [x22, x8, lsl #2]
4          ld1w    { z3.s }, p0/z, [x23, x8, lsl #2]
5          fcmgt   p1.s, p3/z, z0.s, #0.0
6          not     p2.b, p3/z, p1.b
7          and     p2.b, p3/z, p0.b, p2.b
8          ld1w    { z2.s }, p2/z, [x21, x8, lsl #2]
9          sel     z0.s, p1, z0.s, z2.s
10         fmla    z0.s, p3/m, z3.s, z1.s
11         st1w    { z0.s }, p0, [x24, x8, lsl #2]
12         add     x8, x8, x25
13         whilelo p0.s, x8, x19
14         b.mi    .LBB168_2
```

- 1× `fmla` instructions plus 1× predicated `sel` instruction

Auto-Vectorizer Limits

Techniques used for manual vectorization using intrinsics:

- **Loop splitting**: Split loop in separate loops
- **Loop peeling**: Move initial/final loop iterations outside of the loop
- **Pre-loading**: Pre-load data into a SIMD register to avoid overwrite
  - Can, e.g., be applied for loop s211:

```
1 for (int i = 1; i < LEN_1D-1; i++) {
2     a[i] = b[i - 1] + c[i] * d[i];
3     b[i] = b[i + 1] - e[i] * d[i];
4 }
```

- Use SVE compare instructions for vectorizing searching loops

Advanced SVE: Loop s341

- Example of a loop where the compilers failed:

```
1 j = -1;
2 for (int i = 0; i < LEN_1D; i++) {
3     if (b[i] > (real_t)0.) {
4         j++;
5         a[j] = b[i];
6     }
7 }
```

- SVE instructions that can be exploited here:
  - `cmpgt`: Apply logical operator to all lanes and set predicate register
  - `compact`: Shuffle active elements of vector to the right and fill with zero

Advanced SVE: Loop s341 (cont.)

- Manually vectorized loop using intrinsics:

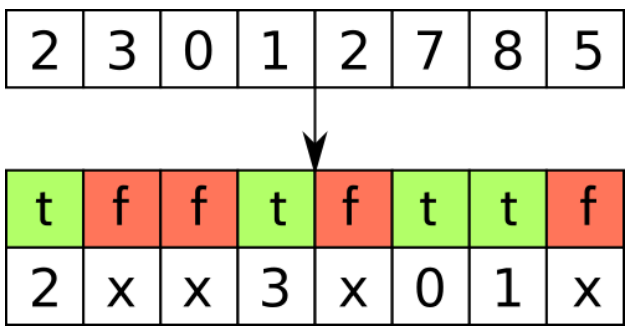
```
1  do {
2       svfloat32_t bv = svld1_f32(pg, &b[i]);
3       svbool_t cg = svcmpgt(pg, bv, zerov);
4       svfloat32_t res = svcompact_f32(cg, bv);
5       int inc_j = svcntp_b32(svptrue_b32(), cg);
6       svbool_t tg = svwhilelt_b32(0, inc_j);
7       svst1(tg, &a[j], res);
8       j += inc_j;
9       i += svcntw();
10      pg = svwhilelt_b32(i, LEN_1D);
11 } while (svptest_any(svptrue_b32(), pg));
```

ISA Limits

- Manual vectorization fails for loop s321:

```
1 j = -1;
2 for (int i = 0; i < LEN_1D; i++) {
3     if (a[i] > (real_t)0.) {
4         j++;
5         a[i] = b[j];
6     }
7 }
```

- Limitation: SVE does not include instructions that would unpack the vector under predicate control



:

Content

Summary and Conclusions

- Compilers vectorized up to 115 of 151 loops
- Loops have been found that could be vectorized manually but have not been vectorized by the compilers
- Good speed-up observed on A64FX
  - But: typically $\epsilon \ll 1$
  - Manually vectorized loops show on average better speed-up
- None of the used compilers seem no have issues auto-vectorizing for SVE with its VLA feature
  - Vectorization statistics for SVE and AVX512 are similar
- Key features of SVE are exploited, but: 3 loops could also be vectorized assuming small extensions of SVE