The challenge of SVE in QEMU

Alex Bennée
Senior Virtualization Engineer
alex.bennee@linaro.org

Agenda

- What is ARM’s SVE?
- Why implement SVE in QEMU?
- How does QEMU’s TCG work?
- Challenges for QEMU’s emulation
- Current Status

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Scalable Vector Extension (SVE)

- Single Instruction, Multiple Data (SIMD)
  - Allows same operation on multiple, non-related, data elements
  - Builds on Neon and AdvancedSIMD
- Larger vector lengths
  - Bigger =~ better (for vectorizable code)
  - Maximum vector length is IMPDEF
- Allows vector length agnostic code
  - Predicate registers
  - Loop constructs
  - Same binary works across all SVE implementations
- Code needs to be vectorizable
  - Ideally auto-vectorized by compiler

ENGINEERS
AND DEVICES
WORKING
TOGETHER

SVE Example (C code)

```
void strcpy(char *restrict dst, const char *src)
{
  while (1) {
    *dst = *src;
    if (*src == '\0') break;
    src++; dst++;
  }
}
```

Source:
https://developer.arm.com/-/media/developer/developers/hpc/white-papers/a-sneak-peek-into-sve-and-vla-programming.pdf

ENGINEERS
AND DEVICES
WORKING
TOGETHER

SVE Assembly

```
sve_strcpy:                          # header
  mov x2, 0
  ptrue p2.b
loop:                                # loop body
  setffr                             # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                   # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b              # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                # function exit
```

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Set up Predicates

```
sve_strcpy:                          # header
  mov x2, 0
  ptrue p2.b
loop:                                # loop body
  setffr                             # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                   # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b              # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                # function exit
```

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Load source string

```
sve_strcpy:                          # header
  mov x2, 0
  ptrue p2.b
loop:                                # loop body
  setffr                             # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                   # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b              # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                # function exit
```

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Check for Termination Condition

```
sve_strcpy:                          # header
  mov x2, 0
  ptrue p2.b
loop:                                # loop body
  setffr                             # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                   # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b              # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                # function exit
```

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Store to destination

```
sve_strcpy:                          # header
  mov x2, 0
  ptrue p2.b
loop:                                # loop body
  setffr                             # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                   # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b              # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                # function exit
```

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Increment string index

```
sve_strcpy:                             # header
  mov x2, 0
  ptrue p2.b
loop:                                   # loop body
  setffr                                # set first fault register
  ldff1b z0.b, p2/z, [x1, x2]
  rdffr p0.b, p2/z                      # read ffr into p0
  cmpeq p1.b, p0/z, z0.b, 0
  brka p0.b, p0/z, p1.b                 # break after
  st1b z0.b, p0, [x0, x2]
  incp x2, p0.b
  b.none loop
  ret                                   # function exit
```

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Why implement SVE in QEMU?

- Early access to SVE
  - Devs can start on SVE enablement before silicon available
- Open source solution
  - FLOSS developers generally prefer fully open stack
  - Academics and research institutions may not have access to proprietary models
- Faster than models
  - linux-user faster than full system emulation
  - QEMU has MTTCG for system emulation
- Good for QEMU
  - Keep pushing/improving the core TCG

QEMU's Instruction Emulation

- Tiny Code Generator (TCG)
  - Just in Time (JIT) recompilation engine
  - Instruction decomposed into intermediate representation (IR)
  - Basic optimisations (dead code, constants etc)
- Generate host instructions
  - Most integer
  - Logic ops
  - Flow control
- Call "helper" functions
  - Complex Instructions
    - System instructions (system registers, TLB manipulation)
    - Exception generation
  - All floating point
    - Scalar float operations
    - Vector float operations

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

QEMU Challenges

- Prerequisite Features
- SoftFloat
- Code generation efficiency
- Diagnostics
- Testing

Prerequisites

SVE is an optional extension but requires:

- Base requirement ARMv8.2-A
  - Especially ARMv8.2-FP16 half-precision floating point
  - ARMv8.1 additional AdvancedSIMD instructions
    - Rounding Double Multiply Add/Subtract
- ARMv8.3-A AArch64 complex numbers

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

SoftFloat

- Helpers call SoftFloat for all floating point ops
  - Maintaining ISA FP semantics on foreign ISA is hard
  - IEEE754 compliance is not enough
- Current QEMU code based on SoftFloat 2a
  - Stuck at 2a due to licence incompatibility
  - Heavy QEMU modifications w.r.t upstream
  - No support for FP16 and later IEEE754 features
  - No further upstream development on 2 series
- Options
  - Migrate to SoftFloat 3c
    - Has FP16 and other newer features
    - SoftFloat3 is a re-write
  - Migrate to another library
    - GNU MPFR?
  - Backport/reimplement on existing 2a
- Current Plan
  - Backport/reimplement on existing 2a

Code Generation

- NEON/AdvancedSIMD mostly helpers
  - All floating point uses SoftFloat
  - Integer can be in generated code
  - Processed in per-element chunks
- TCG currently limited to 32/64 bit operations
  - Expanding with all vector sizes messy
  - Need a scalable length agnostic approach

Diagnostics

- Guest disassembler (for ARM)
  - Libvixl based, updated regularly
- Host dissembler
  - Uses old fork of binutils (pre-GPLv3 or later)
  - Looking at Capstone Engine

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Testing

- RISU - Random Instructions in Userspace
  - Was very helpful during ARMv8 work
  - Now have record/replay support
- QEMU linux-user updates
  - Support SVE signal frames

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Currently In Progress

- Back-porting/re-implementing FP16 for SoftFloat 2a
- Variable-length vectors in TCG operations
- Reviewing v3 of Linux Kernel support
- ARMv8.1 Advanced SIMD instructions
- Capstone Disassembler

Linaro
connect
San Francisco 2017

ENGINEERS
AND DEVICES
WORKING
TOGETHER

Current TODO

- RISU Support (for remaining test-driven development)
- ARMv8.2 FP16 (Half Precision Floating Point)
- ARMv8.3 complex number instructions
- SVE Predicate Instructions
- SVE Memory Faults
- SVE Load/Store instructions
- SVE Calculation instructions
- Linux-user support for SVE stack frame
- System emulation: SVE System Registers

Linaro connect
San Francisco 2017

Any Questions?

Linaro connect
San Francisco 2017

Thank You

#SFO17

SFO17 keynotes and videos on: connect.linaro.org

For further information: www.linaro.org