Road to SVE enablement in LLDB

Omair Javaid
omair.javaid@linaro.org
omjavaid @ Freenode and OFTC IRC
Introduction - Road to SVE enablement in LLDB

- What is Scalable Vector Extension?
- Key architectural features of SVE
- VFP to SVE: Arm vector floating-point architecture evolution
- SVE in Linux user mode and ptrace interface
- Challenge of SVE for LLDB
- What’s next for SVE and LLDB?
What is Scalable Vector Extension?

- SVE is an optional extension to the Armv8-A architecture (Armv8.2-A onwards).
- SVE is an add-on on top of AArch64 Advanced SIMD and floating-point unit.
- Finds its application in processing of large data sets in HPC works loads.
- SVE hardware aimed to complement auto vectorization capability of modern compilers.

Picture Source: wikipedia (Property of Fujitsu)
Key architectural features of SVE

- **Vector Length Agnostic (VLA)**
  - 32 vector registers (Zn) of scalable length from 128 to 2048 bits in multiples of 128 bits.
  - 0 - 127 bits of Zn maps on to AArch64 Vn register

- **Gather-load and scatter-store**
  - Loads a Zn register from non-contiguous memory locations
  - Store result from Zn register to non-contiguous memory locations
Key architectural features of SVE

Predication

- **P0 .. P7** - Per-lane predicate driven load/store/arithmetic
- **P8 .. P15** - Predicate-driven loop management
- **FFR** - Vector partitioning for software-managed speculation allows vector accesses to cross into invalid pages
VFP to SVE: Arm architecture evolution

- **VFP** (32 - 32 bit registers single precision)
- **VFP v2 or VFP v3-D16** (16 - 64 bit registers double precision, backward compatible with VFP with 32 - 32 bit registers single precision)
- **VFP v3-D32 and Neon** (Integer, fixed-point and floating point operations)
  - Qn 16 - 128 bit vectors
  - Dn 32 - 64 bit double precision or
  - Sn 32 - 32 bit single precision

![Diagram showing the evolution from VFP to SVE](image-url)
VFP to SVE: Arm architecture evolution (cont..)

AArch32 - Armv7 Advanced SIMD
- 16 128-bit vector registers
- Integer, fixed-point and non-IEEE single-precision float

AArch64 - Advanced SIMD
- 32 128-bit vector registers
- Full IEEE double-precision float and 64-bit integer vector operations
VFP to SVE: Arm architecture evolution (cont..)

- Scalable data registers Z0-Z31
  - IEEE 754 compliant floating point
    - Double, single and half precision elements
  - Packed 64, 32, 16 & 8-bit integer

- Scalable predicate registers (P0-P7) for load/store/arithmetic

- Scalable predicate registers (P8-P15) for loop management

- FFR first fault register for software speculation
SVE in Linux user mode

- Linux has per-thread SVE register context containing Zn, Pn, FFR, FPCR, FPSR and VL.
- User space can query SVE support using aux vector (HWCAP_SVE)
- Debuggers can interact with SVE registers via ptrace interface using regset flag NT_ARM_SVE
- SVE specific prctl() calls are added to allow programs to manage the SVE vector length configurations

```c
/* SVE/FP/SIMD state (NT_ARM_SVE) */

struct user_sve_header {
    __u32 size; /* total meaningful regset content in bytes */
    __u32 max_size; /* maximum possible size for this thread */
    __u16 vl; /* current vector length */
    __u16 max_vl; /* maximum possible vector length */
    __u16 flags;
    __u16 __reserved;
};
```
SVE in Linux user mode (cont...)

- A NT_ARM_SVE note will be added to each core dump for each thread of the dumped process.
- Detailed documentation can be found in Linux kernel sources at:
  Documentation/arm64/sve.rst

```c
/* SVE/FP/SIMD state (NT_ARM_SVE) */

struct user_sve_header {
    __u32 size; /* total meaningful regset content in bytes */
    __u32 max_size; /* maximum possible size for this thread */
    __u16 vl; /* current vector length */
    __u16 max_vl; /* maximum possible vector length */
    __u16 flags;
    __u16 __reserved;
};
```
**PTRACE interface**

**AArch64 ptrace organization**

- **Non SVE AArch64**
  - ptrace register access fails using NT_ARM_SVE
  - FP/SIMD registers accessible using NT_FPREGSET

- **SVE AArch64 - SVE context not live**
  - user_sve_header.flags = SVE_PT_REGS_FPSIMD

![NT_FPREGSET](image)

```c
struct user_fpsimd_state {
    __uint128_t vregs[32];
    __u32 fpsr;
    __u32 fpcr;
    __u32 __reserved[2];
};
```

![SVE mode FPSIMD Payload](image)

- SVE mode
- FPSIMD Payload
- user_sve_header
- v0, v1
- V30, V31
- FPSR, *FPCR
PTRACE interface (cont...)

AArch64 ptrace organization

- SVE AArch64 - SVE context is live
  - `user_sve_header.flags = SVE_PT_REGS_SVE`
Challenge of SVE for LLDB

- Variable length registers - Dynamic register descriptions
- LLDB has static register descriptions for arm and arm64
- SVE register length is unknown till application run time.
- SVE register lengths can change at run-time
Challenge of SVE for LLDB (cont...)

- Variable sized payload
- V registers are pseudo registers mapping on to Z registers when SVE payload is available
- Application can switch between Neon and SVE modes at run time.
- SVE payload vs FPSIMD payload in SVE mode
Challenge of SVE for LLDB (cont…)

- Register description sync with remote (lldb-server)
- VG register in stop-reply vs DWARF expression evaluation

```c
struct RegisterInfo {
    const char *name;  // Name of this register, can’t be NULL
    const char *alt_name;  // Alternate name of this register, can be NULL
    uint32_t byte_size;  // Size in bytes of the register
    uint32_t byte_offset;  // The byte offset in the register context data where
                          // this register’s value is found.
    // This is optional, and can be 0 if a particular RegisterContext does not
    // need to address its registers by byte offset.
    lldb::Encoding encoding;  // Encoding of the register bits
    lldb::Format format;  // Default display format
    uint32_t kinds[lldb::kNumRegisterKinds];  // Holds all of the various register
                          // numbers for all register kinds
    // List of registers (terminated with
    // LLDB_INVALID_REGNUM). If this value is not null,
    // all registers in this list will be read first, at
    // which point the value for this register will be
    // valid. For example, the value list for ah would be
    // eax (x86) or rax (x64).
    uint32_t *value_regs;
    uint32_t *invalidate_regs;  // List of registers (terminated with
                               // LLDB_INVALID_REGNUM). If this value is not
                               // null, all registers in this list will be
                               // invalidated when the value of this register
                               // changes. For example, the invalidate list for
                               // eax would be rax, ax, ah, and al.
    const uint8_t *dynamic_size_dwarf_expr_bytes;  // A DWARF expression that when
                                                     // evaluated gives
    // the byte size of this register.
    size_t dynamic_size_dwarf_len;  // The length of the DWARF expression in bytes
                                     // in the dynamic_size_dwarf_expr_bytes
                                     // member.
```
Challenge of SVE for LLDB (cont...)

- qRegisterInfoN packet vs Target XML
- Per-thread register lengths
- Target description is exchanged per process
- Register infos are fixed per process
- LLDB has per-thread register context
What’s next for SVE and LLDB?

● LLDB Arm maintainership
  ○ Buildbots and bug fixes
● LLDB SVE testing
● Pointer Authentication - Tagged pointers
● Scalable Vector Extension 2 (SVE2)
● Transactional Memory Extension (TME)
References

- [1] SVE resources on arm.com
- [2] SVE Linux kernel documentation
- [3] Linaro blog post SVE debugging with GDB on QEMU by Alex Bennée
- [4] LLDB source tree
- [5] LLDB GDB remote protocol documentation
- [6] Arm architecture wikipedia
Questions & Comments

Omair Javaid
omair.javaid@linaro.org
omjavaid @ Freenode and OFTC IRC
Thank you

Join Linaro to accelerate deployment of your Arm-based solutions through collaboration

contactus@linaro.org