Arm’s Scalable Vector Extension (SVE)
An ISA feature which Si partners can implement at length – 128 to 2048 bits

How SVE works

The hardware sets the vector length

In software, vectors have no length

The *exact same* binary code runs on hardware with different vector lengths

SVE improves auto-vectorization

Gather-load and scatter-store

Per-lane predication

Predicate-driven loop control and management

Vector partitioning and software-managed speculation

Extended floating-point horizontal reductions
VLA

Vector Length Agnostic
programming model

- Write once
- Compile once
- Vectorize more loops
SVE vs Traditional ISA
How do we compute data which has ten chunks of 4-bytes?

**Aarch64 (scalar)**
- Ten iterations over a 4-byte register

**NEON (128-bit vector engine)**
- Two iterations over a 16-byte register + two iterations of a drain loop over a 4-byte register

**SVE (128-bit VLA vector engine)**
- Three iterations over a 16-byte VLA register with an adjustable predicate
How big can an SVE vector be?

Any multiple of 128 bits up to 2048 bits, and it can be dynamically reduced.

\[(A) \quad VL = LEN \times 128\]
\[(B) \quad VL \leq 2048\]

\(VL\) is implementation dependent, can be reduced by the OS/Hypervisor.
How can you program when the vector length is unknown?

SVE provides features to enable VLA programming from the assembly level and up

Per-lane predication
Operations work on individual lanes under control of a predicate register.

Predicate-driven loop control and management
Eliminate scalar loop heads and tails by processing partial vectors.

Vector partitioning & software-managed speculation
First Faulting Load instructions allow memory accesses to cross into invalid pages.
SVE Registers

- **Scalable vector registers**
  - Z0-Z31 extending NEON’s 128-bit v0-v31.
  - Packed DP, SP & HP floating-point elements.
  - Packed 64, 32, 16 & 8-bit integer elements.

- **Scalable predicate registers**
  - P0-P7 governing predicates for load/store/arithmetic.
  - P8-P15 additional predicates for loop management.
  - FFR first fault register for software speculation.
SVE vector & predicate register organization

256-bit vector, 64-bit elements

255 | 192 | 191 | 128 | 127 | 64 | 63 | 0
---|---|---|---|---|---|---|---
64b | 64b | 64b | 64b | 64b | 64b | 64b | 64b
31 | 24 | 23 | 16 | 15 | 8 | 7 | 0

full predicate

256-bit vector, packed 32-bit elements

255 | 192 | 191 | 128 | 127 | 64 | 63 | 0
---|---|---|---|---|---|---|---
32b | 32b | 32b | 32b | 32b | 32b | 32b | 32b
31 | 24 | 23 | 16 | 15 | 8 | 7 | 0

full predicate

256-bit vector, unpacked 32-bit elements

255 | 192 | 191 | 128 | 127 | 64 | 63 | 0
---|---|---|---|---|---|---|---
32b | 32b | 32b | 32b | 32b | 32b | 32b | 32b
31 | 24 | 23 | 16 | 15 | 8 | 7 | 0

partial predicate
VLA Programming Approaches

Don’t panic!

- **Compilers:**
  - Auto-vectorization: GCC, Arm Compiler for HPC, Cray, Fujitsu
  - Compiler directives, e.g. OpenMP
    - `#pragma omp parallel for simd`
    - `#pragma vector always`

- **Libraries:**
  - Arm Performance Library (ArmPL)
  - Cray LibSci
  - Fujitsu SSL II

- **Intrinsics (ACLE):**
  - [Arm Scalable Vector Extensions and Application to Machine Learning](https://developer.arm.com/ip-products/machine-learning/)

- **Assembly:**
  - Full ISA Specification: [The Scalable Vector Extension for Armv8-A](https://developer.arm.com/ip-products/machine-learning/)

© 2019 Arm Limited
SVE supports vectorization in complex code
Right from the start, SVE was engineered to handle codes that usually won’t vectorize.

Gather-load and scatter-store
Loads a single register from several non-contiguous memory locations.

Extended floating-point horizontal reductions
In-order and tree-based reductions trade-off performance and repeatability.
Portability

Is it really possible to run a vectorized application anywhere?

**Write once**: can my code *compile* for machines with different VL?

- Code that is auto-vectorized by the compiler
- Hand-written assembly
- Hand-written C intrinsics

**Compile once**: Can I take my executable and *run it* on machines with different VL?

- Self contained programs with no external dependencies
- But what about programs that depend on external libraries? ... (spoiler: 🙌)
Auto-vectorize external calls: \texttt{libm} example.

```c
float sinf(float);
```

**NEON**

- Neon has \texttt{128-bit} and \texttt{64-bit} register split.

- The library has to provide at least 2 symbols, because it doesn’t know where the auto-vec code comes from:
  
  - \texttt{ZGVnN2v_sinf}
  - \texttt{ZGVnN4v_sinf}

**SVE**

- Does \texttt{libm} need to provide a symbol for each VL?
  
  - \texttt{ZGVsM4v_sinf}
  - \texttt{ZGVsM6v_sinf}
  - \texttt{ZGVsM8v_sinf}
  - \texttt{ZGVsM10v_sinf}
  - \texttt{...}

- One symbol!
  
  \texttt{ZGVsMxv_sinf}
Open source support

- **Arm actively posting SVE open source patches upstream**
  - Beginning with first public announcement of SVE at HotChips 2016

- **Available upstream**
  - **GNU Binutils-2.28**: released Feb 2017, includes SVE assembler & disassembler
  - GCC 8: Full assembly, disassembly and basic auto-vectorization
  - LLVM 7: Full assembly, disassembly
  - QEMU 3: User space SVE emulation
  - GDB 8.2: HPC use cases fully included

- **Under upstream review**
  - **LLVM**: Since Nov 2016, as presented at LLVM conference
  - **Linux kernel**: Since Mar 2017, LWN article on SVE support
SVE: More Powerful Vectorization on V1

SVE vectorizes more codes and makes better use of the vector units

**Simulation Projections**

Lower is better
Quick Recap

- SVE enables Vector Length Agnostic (VLA) programming
- VLA enables portability, scalability, and optimization
- Predicates control which operations affect which vector lanes
  - Predicates are not bitmasks
  - You can think of them as dynamically resizing the vector registers
- The actual vector length is set by the CPU architect
  - Any multiple of 128 bits up to 2048 bits
  - May be dynamically reduced by the OS or hypervisor
- SVE was designed for HPC and can vectorize complex structures
- Many open source and commercial tools currently support SVE

```
for (i = 0; i < n; ++i)

WHILELT n

INDEX i

n-2 n-1 n n+1

1 1 0 0

+ pred

1 2 0 0

GNU! GNU! GNU!
```
Hands On: HACC
05_Apps/01_HACC

See README.md for details

- Computationally intensive part of an N-body cosmology code.
- Application performance is dominated by a long chain of floating point instructions.
- Performance scales well with vector length.
- FOM: Wall clock time spent in the application loop reported in seconds.

```
./hacc_arm_neon.exe 1000
Maximum OpenMP Threads: 48
Iterations: 1000
Gravity Short-Range-Force Kernel (5th Order): 12823.6 -444.108 -645.349: 5.19743 s
```

```
./hacc_arm_sve.exe 1000
Maximum OpenMP Threads: 48
Iterations: 1000
Gravity Short-Range-Force Kernel (5th Order): 12823.6 -444.108 -645.349: 1.55594 s
```