SC19

Introduction to Programming for Arm SVE

SVE

Scalable Vector Extension

*SVE is a vector extension for **AArch64** execution mode for the A64 instruction set of the Armv8 architecture.*

What makes it a Scalable Vector Extension?

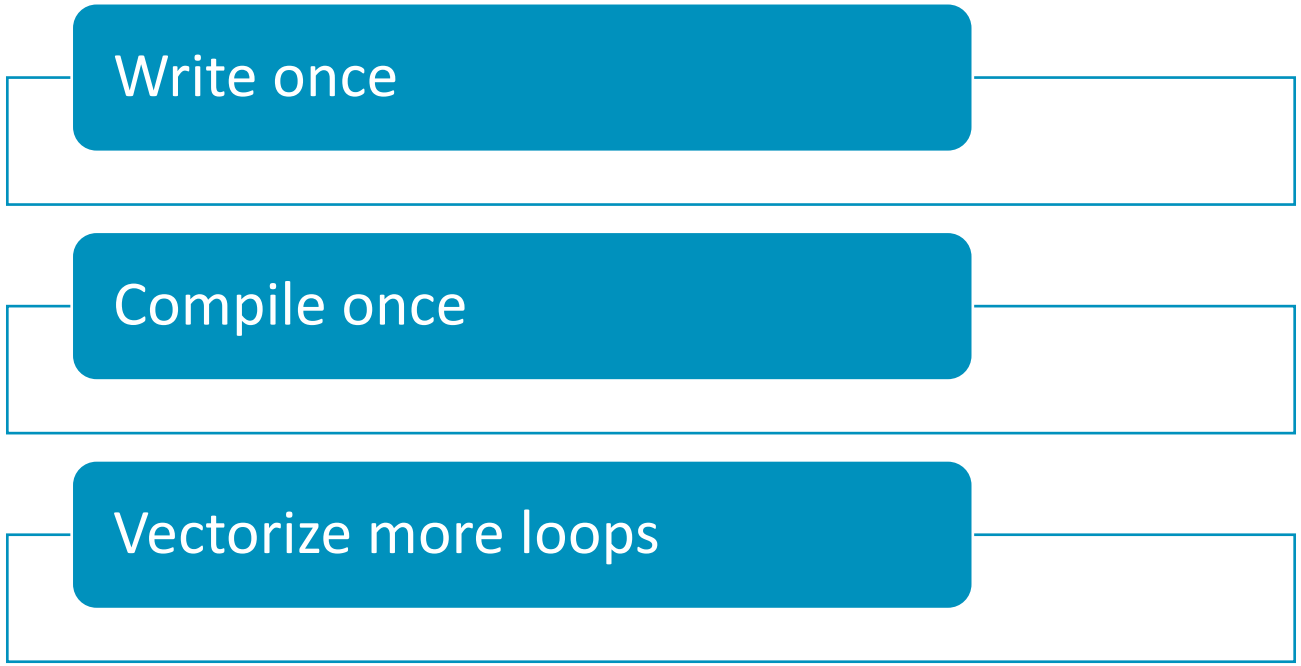
- **There is no preferred vector length**
  - The vector length (VL) is a hardware choice, 128-2048b, in increments of 128b
  - A Vector Length Agnostic (VLA) programming adjusts dynamically to the available VL

- **SVE addresses traditional barriers to auto-vectorization**
  - Software-managed speculative vectorization of uncounted loops
  - Extract more data-level parallelism (DLP) from existing C/C++/Fortran source code

- **SVE is a new approach to vectorization, not an iteration on existing ISAs (e.g. NEON)**
  - SVE is a separate, optional extension with a new set of instruction encodings
  - Initial focus is **HPC** and general-purpose server, not media/image processing

VLA

Vector **L**ength **A**gnostic programming model

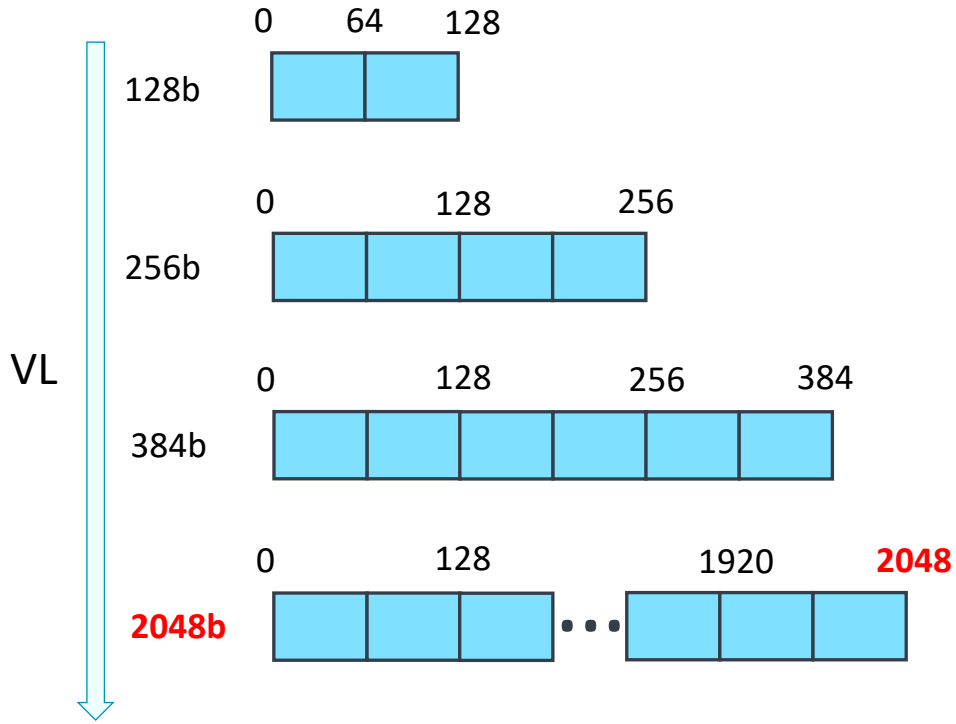


arm

SVE ISA does not mandate a single, fixed vector length

The vector length is **LEN x 128-bit** up to 2048

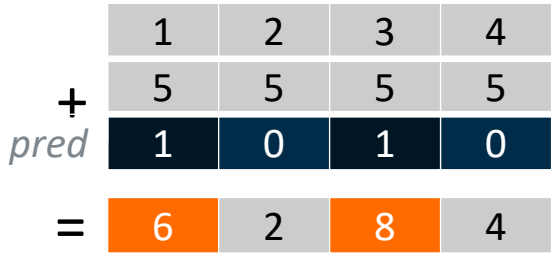
- There is no preferred vector length
- No need to recompile
- No need to rewrite hand-coded SVE assembler or C intrinsics
- The programmer's intent is expressed in the binary ➔ easier to optimize
- **Predicate Registers** indicate active vector lanes



arm

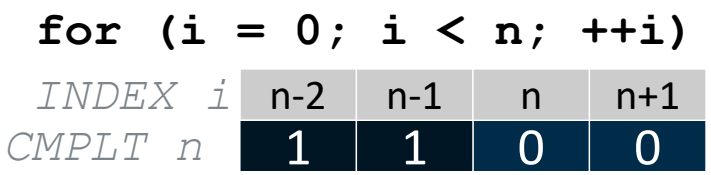
How can you program when the vector length is unknown?

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



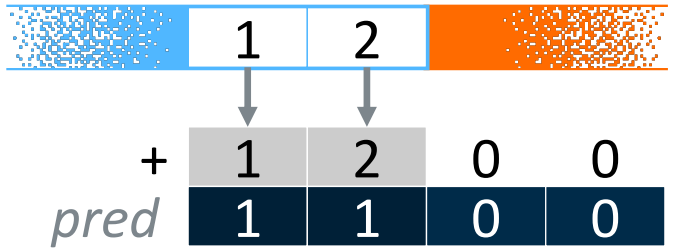
Per-lane predication

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.

arm

SVE vs Traditional ISA

How do we compute data which has ten chunks of 8-bytes?

Aarch64 (scalar)

- ❑ Ten iterations over an 8-byte register

NEON (128-bit vector engine)

- ❑ Four iterations over a 16-byte register + two iterations of a drain loop over a 8-byte register

SVE (VLA vector engine)

- ❑ Three iterations over a 32-byte **VLA register** with an adjustable **predicate**

arm

How big can an SVE vector be?

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

```
(A) VL = LEN x 128
(B) VL <= 2048
```

VL is *implementation dependent*,

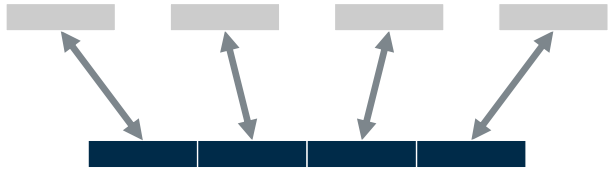
can be *reduced by the OS/Hypervisor*.

?

arm

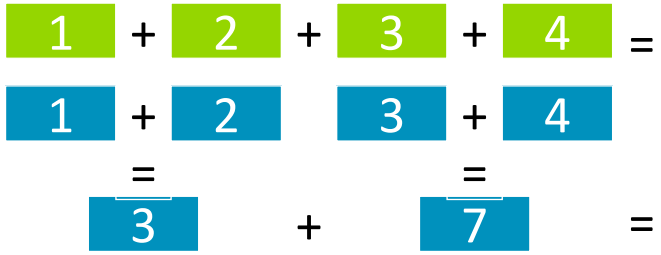
SVE supports vectorization in complex code

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.



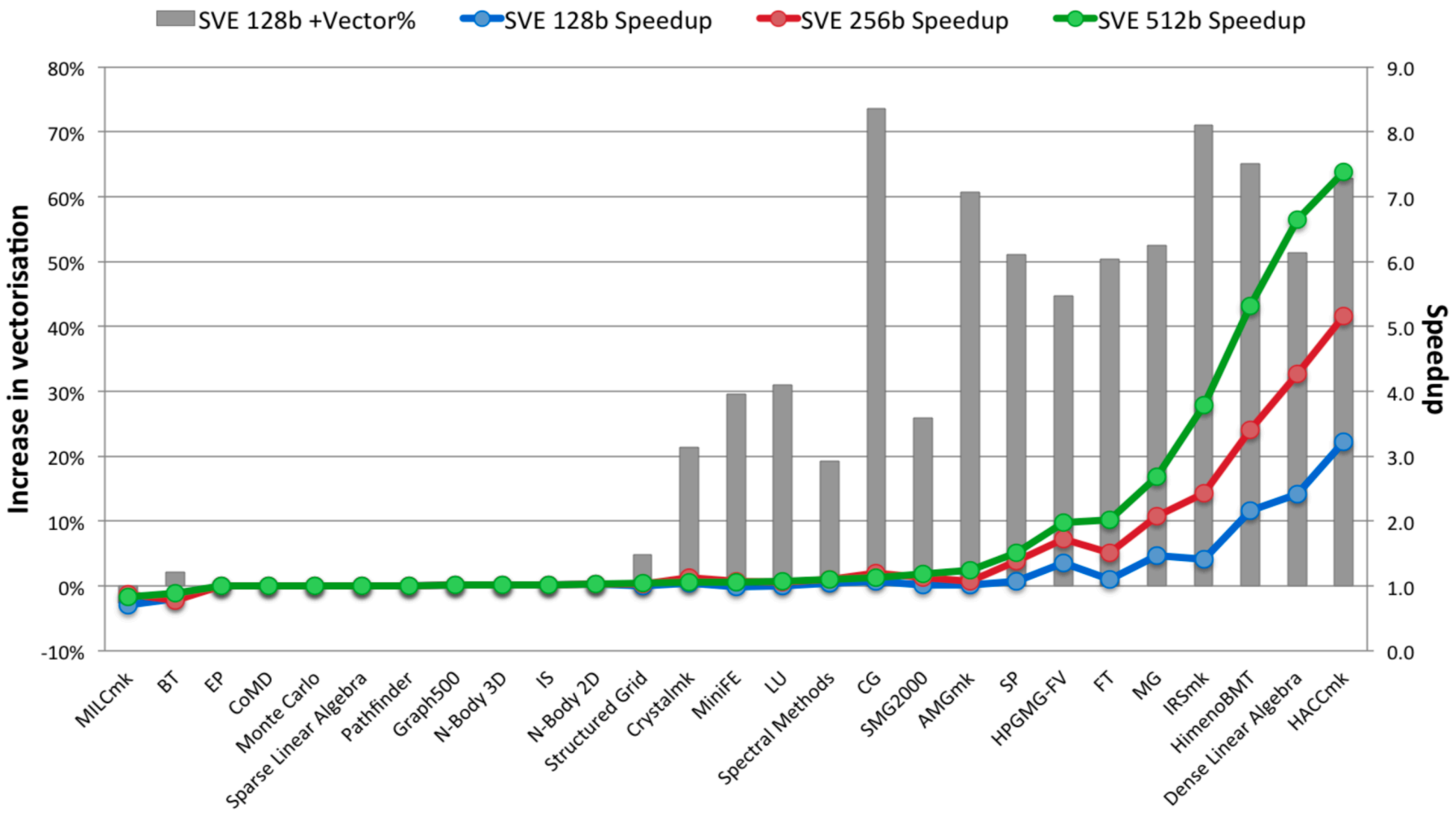
Extended floating-point horizontal reductions

In-order and tree-based reductions trade-off performance and repeatability.

arm

HPC benchmarks of SVE vs. NEON

IEEE MICRO, Vol. 37 Issue 2, March - April 2017 – DOI: 10.1109/MM.2017.35



arm

Open source support

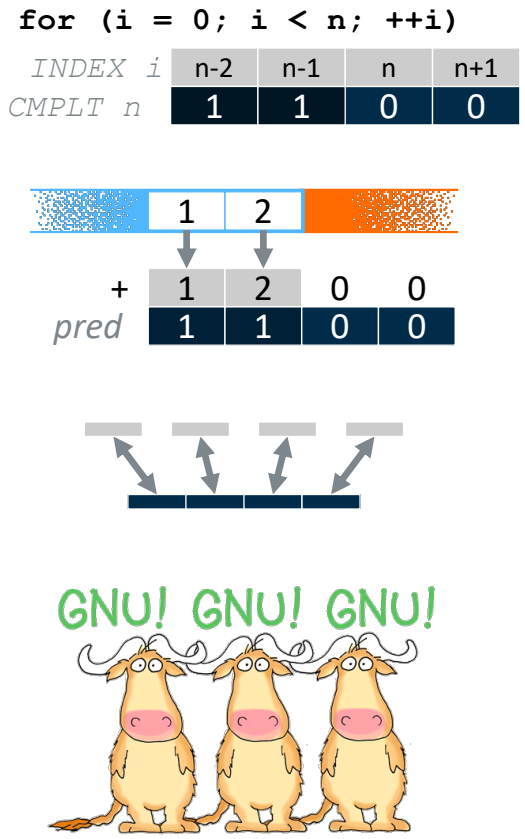
- **Arm actively posting SVE open source patches upstream**
  - Beginning with first public announcement of SVE at HotChips 2016

- **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

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



arm

arm

SVE Compilers

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`
- Intrinsics:
  - Arm C Language Extensions for SVE
  - Arm Scalable Vector Extensions and Application to Machine Learning
- Assembly:
  - Full ISA Specification: The Scalable Vector Extension for Armv8-A
  - Lots of worked examples: A Sneak Peek Into SVE and VLA Programming

SVE Compiler Support

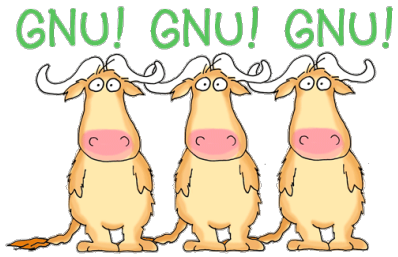
FUJITSU

CRAY

Compiler	Assembly / Disassembly	Inline Assembly	ACLE	Auto-vectorization	Math Libraries
Arm Compiler for HPC	SVE + SVE2	SVE + SVE2	SVE + SVE2	SVE+ SVE2	SVE
LLVM/Clang	SVE + SVE2	SVE + SVE2	SVE + SVE2 in LLVM 10	SVE + SVE2 in LLVM 11	
GNU	SVE + SVE2	SVE + SVE2	SVE + SVE2 in GNU 10	SVE now SVE2 in GNU10	

arm

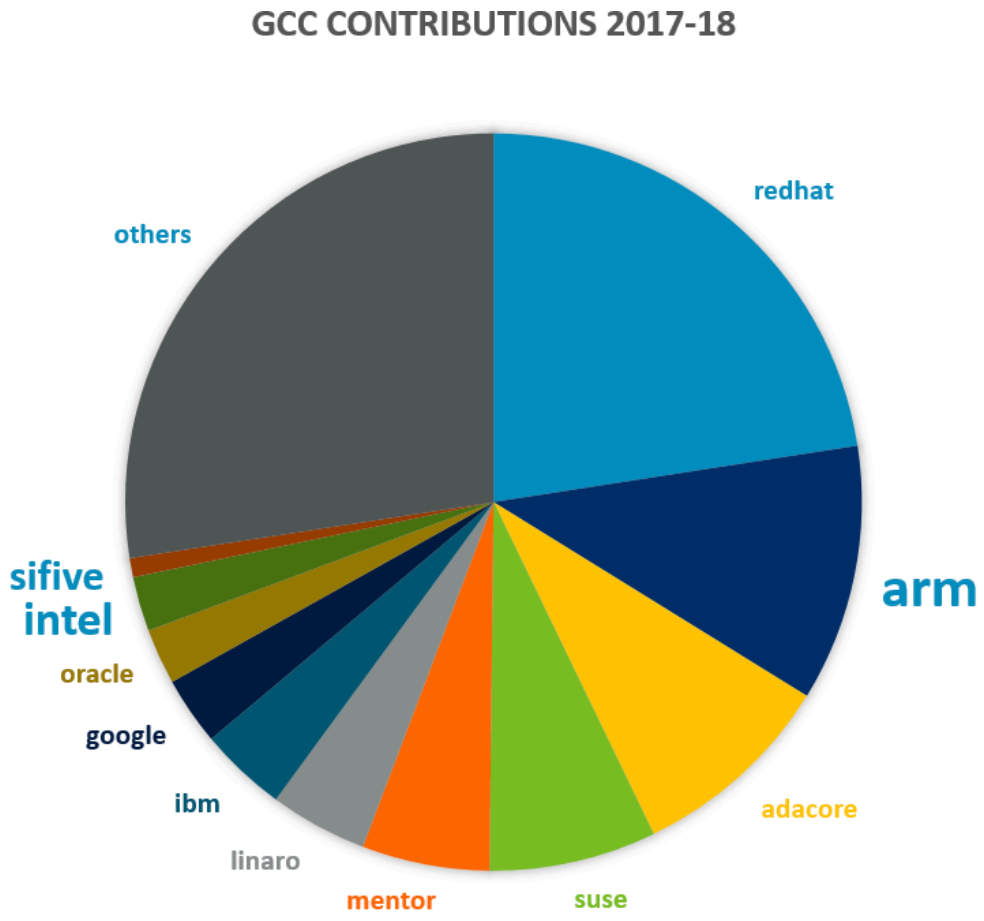
GNU compilers are best-in-class in the Arm ecosystem



With Arm being significant contributor to upstream GNU projects

On Arm, GNU compilers are first class compilers alongside commercial compilers.

- Unlike other architectures
- Arm is 2nd largest contributor to GCC in 2017-18
- Focus on enablement and performance
- Key for Arm to succeed in Cloud/Data center segment



GNU toolchain ships with Arm Allinea Studio

- Best effort support (bug fixes and performance improvements in upcoming GNU releases)
- GCC 8.2 part of Allinea Studio 19.0

arm

arm COMPILER

Arm's commercially-supported C/C++/Fortran compiler

Compilers tuned for Scientific Computing and HPC

Latest features and performance optimizations

Commercially supported by Arm

Tuned for Scientific Computing, HPC and Enterprise workloads

- Processor-specific optimizations for various server-class platforms
- Optimal shared-memory parallelism via Arm's optimized OpenMP runtime

Linux user-space compiler with latest features

- C++ 14 and Fortran 2003 language support with OpenMP 4.5*
- Support for Armv8-A and SVE architecture extension
- Based on LLVM and Flang, leading open-source compiler projects

Commercially supported by Arm

- Available for a wide range of Arm-based platforms running leading Linux distributions – RedHat, SUSE and Ubuntu

Arm Compiler for HPC: Front-end

Clang and Flang

C/C++

- Clang front-end
  - C11 including GNU11 extensions and C++14
  - Arm's 10-year roadmap for Clang is routinely reviewed and updated to respond to customers
- C11 with GNU11 extensions and C++14
- **Auto-vectorization for SVE and NEON**
- OpenMP 4.5

Fortran

- Flang front-end
  - Extended to support gfortran flags
- Fortran 2003 with some 2008
- **Auto-vectorization for SVE and NEON**
- OpenMP 3.1
- Transition to flang "F18" in progress
  - Extensible front-end written in C++17
  - Complete Fortran 2008 support
  - OpenMP 4.5 support

arm

Arm Compiler for HPC: Back-end

LLVM9

- Arm pulls all relevant cost models and optimizations into the downstream codebase
  - Arm's si-partners are committed to upstreaming cost models for future cores to LLVM
- Auto-vectorization via LLVM **vectorizers:**
  - Use cost models to drive decisions about what code blocks can and/or should be vectorized
  - Two different vectorizers from LLVM: Loop Vectorizer and SLP Vectorizer
- Loop Vectorizer support for SVE and NEON:

  - Loops with unknown trip count
  - Runtime checks of pointers
  - Reductions
  - Inductions
  - "If" conversion
  - Pointer induction variables
  - Reverse iterators
  - Scatter / gather
  - Vectorization of mixed types
  - Global structures alias analysis

arm

Compile and link your application on Arm

- Modify the Makefile/installation scripts to ensure compilation for aarch64 happens
- Compile the code with the **Arm Compiler for HPC**
- Link the code with the **Arm Performance Libraries**

- Examples:
- `$> armclang -c –I/path/armpl/include example.c -o example.o`
- `$> armclang example.o -armpl -o example.exe -lm`

Arm Compiler for HPC	GNU Compiler
armclang	gcc
armclang++	g++
armflang	gfortran

arm

Targeting SVE with both Arm compiler and GNU (8+)

- Compilation targets a specific architecture based on an architecture revision
  - -mcpu=native -march=armv8.1-a+lse+sve
    - Learn more: https://community.arm.com/.../compiler-flags-across-architectures-march-mtune-and-mcpu
- -march=armv8-a
  - Target V8-a
  - Will generate NEON instructions
  - No SVE
- -march=armv8-a+sve
  - Will add SVE instruction generations
- Check the assembly (-S)
  - ***armclang++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp***
  - ***g++ -S -o code.s -Ofast -g -march=armv8-a+sve code.cpp***

Optimization Remarks for Improving Vectorization

Let the compiler tell you how to improve vectorization

To enable optimization remarks, pass the following `–Rpass` options to `armclang`:

Flag	Description
-Rpass=<regexp>	What was optimized.
-Rpass-analysis=<regexp>	What was analyzed.
-Rpass-missed=<regexp>	What failed to optimize.

For each flag, replace <regexp> with an expression for the type of remarks you wish to view.
Recommended <regexp> queries are:

- `–Rpass=\(loop–vectorize\|inline\)\`
- `–Rpass–missed=\(loop–vectorize\|inline\)`
- `–Rpass–analysis=\(loop–vectorize\|inline\)`

where loop-vectorize will filter remarks regarding vectorized loops, and inline for remarks regarding inlining.

Optimization remarks example

https://developer.arm.com/products/software-development-tools/hpc/arm-fortran-compiler/optimization-remarks

```
armclang -O3 -Rpass=.* -Rpass-analysis=.* example.c
```

```
example.c:8:18: remark: hoisting zext
[-Rpass=licm]
        for (int i=0;i<K; i++)
        ^
example.c:8:4: remark: vectorized loop (vectorization width: 4, interleaved count: 2)
[-Rpass=loop-vectorize]
        for (int i=0;i<K; i++)
        ^ example.c:7:1: remark: 28 instructions in function
[-Rpass-analysis=asm-printer]
     void foo(int K) {     ^
```

```
armflang -O3 -Rpass=loop-vectorize example.F90 -gline-tables-only
```

```
example.F90:21: vectorized loop (vectorization width: 2, interleaved count: 1)
[-Rpass=loop-vectorize]
        END DO
```


arm

GCC Optimization and Vectorization Reports

- https://gcc.gnu.org/onlinedocs/gcc/Developer-Options.html

`-fopt-info`	
`-fopt-info-optimized`	What was optimized
`-fopt-info-missed`	What was not optimized
`-fopt-info-all`	Everything
`-fopt-info-all=file.out`	Dump to file

```
[phirid01@co-c6-16-1 ~]$ gfortran -mcpu=thunderx2t99 -o test test.f90 -O3  -fopt-info-missed
test.f90:8:0: note: misalign = 0 bytes of ref c[_11]
test.f90:8:0: note: can't use a fully-masked loop because the target doesn't have the appropriate masked load or st
ore.
test.f90:8:0: note: cost model: the vector iteration cost = 4 divided by the scalar iteration cost = 1 is greater o
r equal to the vectorization factor = 2.
test.f90:8:0: note: not vectorized: vectorization not profitable.
test.f90:8:0: note: not vectorized: vector version will never be profitable.
test.f90:8:0: note: not vectorized: unsupported data-type
test.f90:8:0: note: can't determine vectorization factor.
test.f90:7:0: note: misalign = 0 bytes of ref b[_8]
```

Arm Compiler for HPC: Vectorization Control

OpenMP and clang directives are supported by the Arm Compiler for HPC

C/C++	Fortran	Description
`#pragma ivdep`	`!DIR$ IVDEP`	Ignore potential memory dependencies and vectorize the loop.
`#pragma vector always`	`!DIR$ VECTOR ALWAYS`	Forces the compiler to vectorize a loop irrespective of any potential performance implications.
`#pragma novector`	`!DIR$ NO VECTOR`	Disables vectorization of the loop.

Clang compiler directives for C/C++	Description
`#pragma clang loop vectorize(assume_safety)`	Assume there are no aliasing issues in a loop.
`#pragma clang loop unroll_count(_value_)`	Force a scalar loop to unroll by a given factor.
`#pragma clang loop interleave_count(_value_)`	Force a vectorized loop to interleave by a factor

arm

Arm Compiler for HPC: Vectorization

Vectorization in QMCPACK

QMCPACK computeDistances()

```
int first=0;
float x0=pos[0], y0=pos[1], z0=pos[2];

#pragma omp simd aligned(temp_r,px,py,pz,dx,dy,dz)
for(int iat=first; iat<last; ++iat)
{
  dx[iat]=px[iat]-x0;
  dy[iat]=py[iat]-y0;
  dz[iat]=pz[iat]-z0;
  temp_r[iat]=sqrt(dx[iat]*dx[iat] +
                   dy[iat]*dy[iat] +
                   dz[iat]*dz[iat]);
}
```

How to vectorize

- The fsqrt instruction doesn't handle glibc `sqrt()` return values.
- armclang vectorizes this with any one of:
  - `-fno-math-errno`
  - `-ffast-math`
  - `-Ofast`
- Note: Intel's ICC has internal implementations of sqrt(), one of which (probably) handles return values, nans, etc, and one of which doesn't. They always inline one of them (they never call glibc sqrt).

arm

Arm Compiler for HPC: Vectorization

Vectorization in HACC

HACC Step16_int()

```
for ( k = 0; k < count1; k++ ) {
  dxc = xx1[k] - xxi;  dyc = yy1[k] - yyi;
  dzc = zz1[k] - zzi;
  r2 = dxc * dxc + dyc * dyc + dzc * dzc;
  m = ( r2 < fsrrmax2 ) ? mass1[k] : 0.0f;
  s0 = r2 + mp_rsm2;
  s1 = s0 * s0 * s0;
  s2 = 1.0f / sqrtf( s1 ) -
       ( ma0 + r2*(ma1 +
                   r2*(ma2 +
                       r2*(ma3 +
                           r2*(ma4 + r2*ma5)))));
  f = ( r2 > 0.0f ) ? m * s2 : 0.0f;
  *ax = *ax + f * dxc;  *ay = *ay + f * dyc;
  *az = *az + f * dzc;
}
```

How to vectorize

```
reproducer.c:17:30: remark: List vectorization was
possible but not beneficial with cost 0 >= 0
[-Rpass-missed=slp-vectorizer]
        r2 = dxc * dxc + dyc * dyc + dzc * dzc;
```

Add clang vectorization directive:

```
#pragma clang loop vectorize(enable)
for ( k = 0; k < count1; k++ )
```

```
reproducer.c:11:5: remark: vectorized loop
(vectorization width: 4, interleaved count: 1)
[-Rpass=sve-loop-vectorize]
```


arm