Compiler-driven SIMD offloading using Arm Streaming SVE

Mohamed Husain Noor Mohamed, **Adarsh Patil**,
Latchesar Ionkov, Eric Van Hensbergen

5th Workshop on Compiler-assisted Correctness Checking and Performance Optimization for HPC (C3PO)
13th June 2025

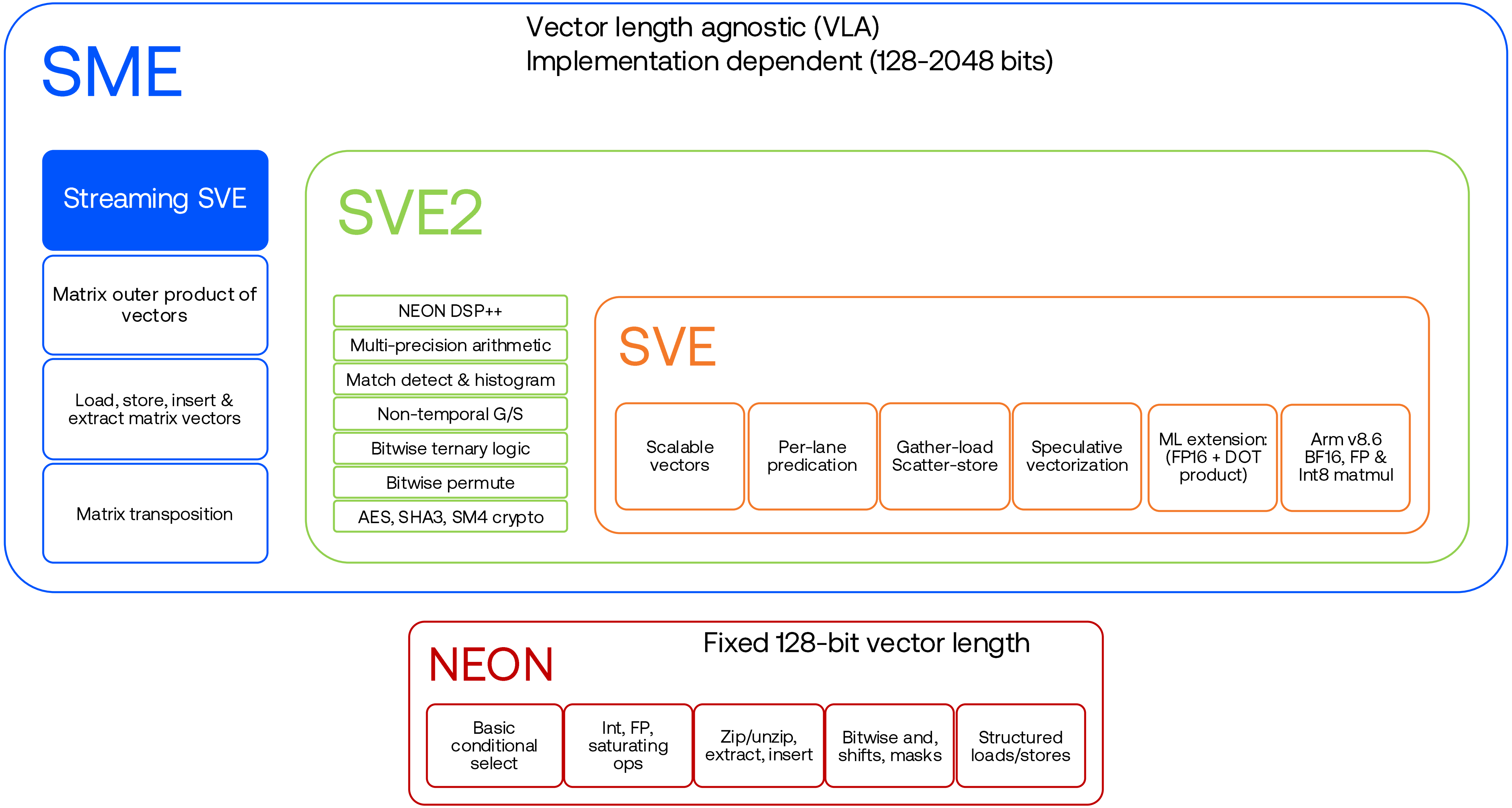
Outline

- Introduction
  - Arm SIMD instructions and Streaming SVE
- Motivation
  - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
  - Synchronizations
  - Memory access characteristics
- Compiler extension proposals
  - LLVM
  - MLIR
- Conclusions and takeaways

Outline

- Introduction
  - Arm SIMD instructions and Streaming SVE
- Motivation
  - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
  - Synchronizations
  - Memory access characteristics
- Compiler extension proposals
  - LLVM
  - MLIR
- Conclusions and takeaways

Arm vector ISA extensions

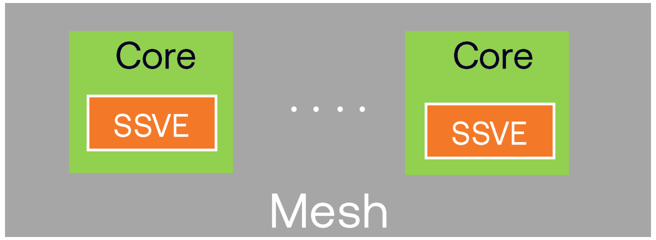

SME
Vector length agnostic (VLA)
Implementation dependent (128-2048 bits)
Streaming SVE
Matrix outer product of vectors
Load, store, insert & extract matrix vectors
Matrix transposition
SVE2
NEON DSP++
Multi-precision arithmetic
Match detect & histogram
Non-temporal G/S
Bitwise ternary logic
Bitwise permute
AES, SHA3, SM4 crypto
SVE
Scalable vectors
Per-lane predication
Gather-load Scatter-store
Speculative vectorization
ML extension: (FP16 + DOT product)
Arm v8.6 BF16, FP & Int8 matmul
NEON
Fixed 128-bit vector length
Basic conditional select
Int, FP, saturating ops
Zip/unzip, extract, insert
Bitwise and, shifts, masks
Structured loads/stores

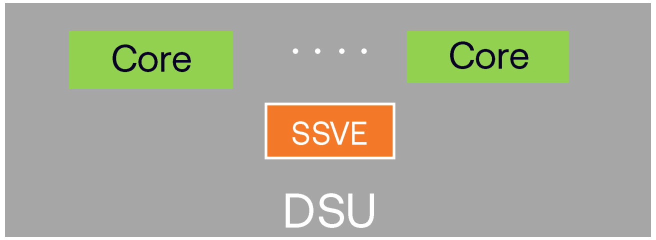
Streaming SVE deep-dive

- *Streaming SVE:* High-throughput processing of large datasets with simple loops/minimal branching
  - Can execute a subset of SVE2 instructions with vector length = matrix tile width
  - Some SVE and most NEON instructions become *Illegal*

    All other AArch64 scalar integer, FP, and control flow instructions remain *legal*
- SMSTART & SMSTOP instructions to enter/exit *Streaming SVE mode.*
  - Cheap write to PSTATE.SM execution mode bit, tracked at core front-end
  - No need to call OS/Hyp to switch, registers are cleared upon mode exit
- Streaming SVE architecture
  - SSVE unit* is disaggregated from core but architecturally acts as part of each core
  - Instructions sent to SSVE unit when streaming mode enabled; dedicated bus for control path – addr, offsets etc.
  - Data path: uses load/stores; private L1, high-bandwidth connection to shared L3
  - Multiple implementations possible from the spec

Per-core private





Multi-core shared

* SME unit used exclusively in streaming mode

Key question: What’s the best way to use Streaming SVE?

- Intrinsics (as used in Geekbench)
- Libraries (e.g., Arm Compute Library)
- Compilers 🡐 Today’s talk!

Goal: Enable performant, precise, repeatable code-gen using auto-vectorizers

Outline

- Introduction
  - Arm SIMD instructions and Streaming SVE
- Motivation
  - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
  - Synchronizations
  - Memory access characteristics
- Compiler extension proposals
  - LLVM
  - MLIR
- Conclusions and takeaways

Generating SSVE instructions using auto-vectorizers

1. **Streaming mode switch** using *AArch64 SME Attribute*
   - Function level annotation `void s_callee(void) __arm_streaming;`
   - clang manages PSTATE.SM automatically

2. **Force auto-vectorization** for streaming-enabled functions using flags
   `-mllvm -enable-scalable-autovec-in-streaming-mode`

3. Generate **IR with scalable vector** type *
   - LLVM

```
define float @add_f32(<vscale x 8 x float> %a, <vscale x 4 x float> %b) {
  %r1 = call @llvm.vector.reduce.fadd.f32.nxv8f32(float - 0.0, <vscale x 8 x float> %a)
  %r2 = call @llvm.vector.reduce.fadd.f32.nxv4f32(float - 0.0, <vscale x 4 x float> %b)
  %r = fadd %r1, %r2
  ret float %r
}
```

   - MLIR

```
llvm.func @vector_splat_1d_scalable()->vector<[4] xf32> {
  %0 = llvm.mlir.constant(dense<0.000000e+00> : vector<[4]xf32>) : vector<[4]xf32>
  llvm.return %0 : vector<[4]xf32>
}
```

llvm.vscale or vector.vscale is unknown, use flags `–mcpu/-mtune` to specify

* Targeting SME from MLIR, MLIR Open Design Meeting, June 2023

Issues with auto-vectorizer generated code

- **Correctness**

  Generates SVE2 instructions but SSVE != SVE2, SSVE subset of SVE2 (leads to `EXC_BAD_INSTR`)

  Excluded instructions depends on implementation (query architectural flags)

- **Performance**

  Incompatible cost-model to decide profitability of SSVE vectorization

  Why? Different architecture

  SVE2 in-core SIMD != SSVE core-adjacent SIMD offload

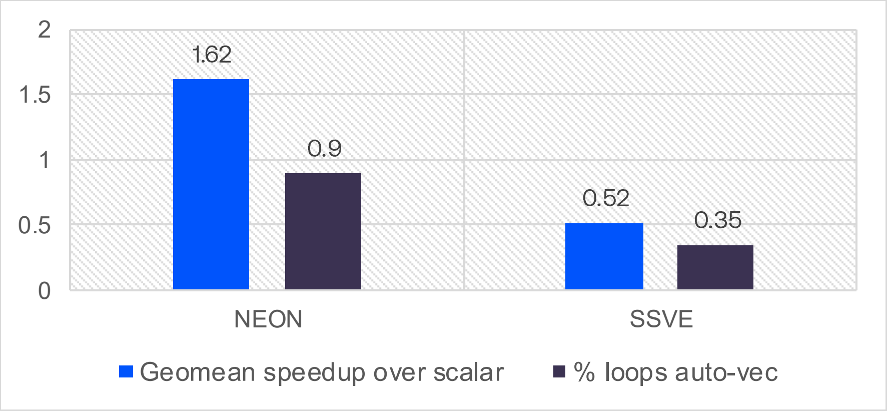
  Let’s benchmark it!

  Platform A, Armv9.2 with SME support

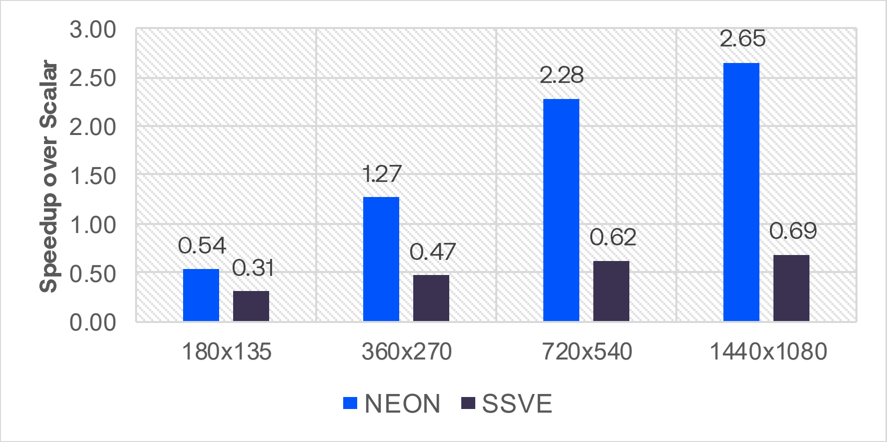
  128-bit NEON, 512-bit SSVE unit

Performance of auto-vectorized code

- TSVC_2 (1M iterations, LEN_1D 32000, LEN_2D 256)
  Correctness: Only 35% loops are auto-vec to SSVE
  Performance: slowdowns over NEON and scalar



- Mandelbrot set (varied input sizes)
  SSVE shows slowdown over NEON and scalar



- SPEC 2017
  505.mcf_r: 1.9x slowdown over NEON

Outline

- Introduction
    - Arm SIMD instructions and Streaming SVE
- Motivation
    - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
    - Synchronizations
    - Memory access characteristics
- Compiler extension proposals
    - LLVM
    - MLIR
- Conclusions and takeaways

Synchronizations

- GPR and FPR sync
  - **Issue:** Stalls occur when SSVE and core access the same stack cache lines.
  - Cause: SSVE has a private L1 and accesses primarily FP (FPR) stack objects.
  - Mitigation: Use padding between GPR- and FPR-accessed stack objects.
    Use `--aarch64-stack-hazard-size` flag
  - Effect: Padding moves FPR-only data inward, GPR-only data outward, reducing conflicts.
  - Limitation: Padding can’t isolate all objects (e.g., VLAs, shared data).
  - Impact: Residual stall penalty increases with loop count — from 17% up to 61%*

- Predicates sync
  - Predicate registers used to select active lanes in SSVE
  - SSVE unit can produce predicates independently
  - **Issue:** Core using SSVE-produced predicates requires synchronization.
    Predicate synchronization adds additional latency
  - Mitigation: use vector registers instead, place predicate usages as far as possible
  - Impact: Stall penalty can be tens of cycles *

*Data gathered by micro-benchmarking on Platform A

Memory access characteristics

- Load-store region table (LSRT) Hazards
  - LSRT manages and synchronizes translation between core and SSVE unit (4KB pages).
  - **Issue:** Hazard condition if core and SSVE access the same data, with one being a write
  - LSRT tracks hazards at aligned 1KB mem regions (4 hazard granules per addr. trans. granule)
  - Effect: New accesses stalled until prior writes in the same region are complete and coherent

  - Mitigation:
    Avoid mixing vector and scalar data in the same data structures, including the stack.
    Align stack regions to 1KB if data is accessed by both core and SSVE unit
    Generate spills and fills of both scalar/vector data accessing the same region

  - Impact:
    QPSK modulation kernel in Arm RAL shows 6x slowdown over NEON due to LSRT hazard
    Using above techniques improves performance by 4x

Memory access characteristics

- Offload overheads
  - Non-linear scaling: SSVE outperforms NEON after input crosses a “performance knee”
  - Mitigation: Use conservative vectorization factor (VF) thresholds, default to NEON

- Prefetcher on SSVE unit
  - Optimized for strided and nested access; struggles with arbitrary patterns.
  - Mitigation: Fall back to NEON if data-dependent addresses or pointers

- Converting gather/scatter
  - Regular strides can be converted to load/store using ZIP/UNZIP instructions
  - Mitigation: Fall back to NEON for random strides

Outline

- Introduction
  - Arm SIMD instructions and Streaming SVE
- Motivation
  - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
  - Synchronizations
  - Memory access characteristics
- Compiler extension proposals
  - LLVM
  - MLIR
- Conclusions and takeaways

Cursory thoughts on compiler extensions

- LLVM: Loop Vectorizer pass
  - Cost modelling for basic auto-vectorization

    Vectorization Factor (VF) for profitability analysis expressed as `<vscale X N X type>`

    Code generated compatible for any `vscale` between `vscale_range(min[,max])`

    Issue: Model assumes linear scaling of performance with increasing vector lengths – incorrect!

    Fixed instruction costs in compiler table

    Issue: SVE2 instruction cost changes in streaming vs non-streaming mode

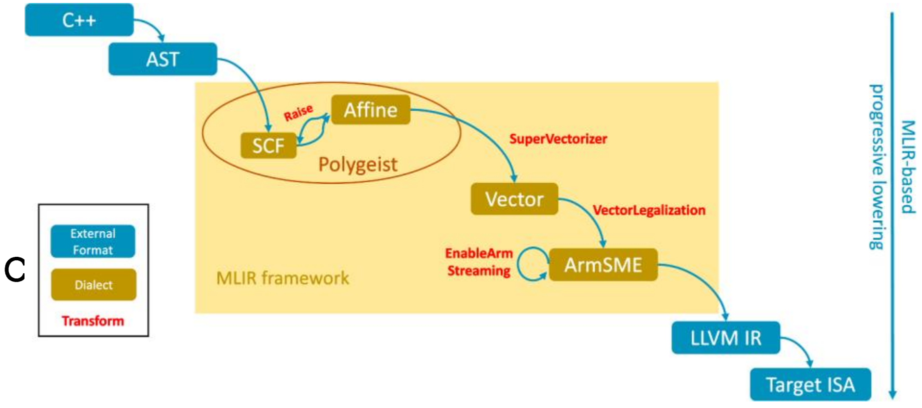
  - Vectorization Plans (VPlan) for advanced vectorization strategies

    Cost computed by recusing into VPBasicBlocks and into VPRecipes

    Issue: Does not model streaming mode overheads (Instantiate VPlan with static cost for SSVE blocks)

- MLIR: Polygeist – C/C++ compilation flow
  - *Vector* dialect supports representing Scalable Vectors
  - Extend *ArmSME* dialect with SSVE operations - exists
  - *EnableArmStreaming* transform to emit smstart/smstop – exists
  - **Issue:** *SuperVectorizer* transform does not generate Scalable Vec

    requires a new cost model!



Outline

- Introduction
  - Arm SIMD instructions and Streaming SVE
- Motivation
  - Compiler auto-vectorizers limitations
- Architectural considerations for code-gen & cost-model
  - Synchronizations
  - Memory access characteristics
- Compiler extension proposals
  - LLVM
  - MLIR
- Conclusions and takeaways

Takeaways

- New **“offload” SIMD compute** paradigm – Arm Streaming SVE
  - Tightly-coupled, core-adjacent accelerator
  - Ultra-wider vector widths
  - Supports (almost) same ISA as the core
- **Compilers well-suited to enable** SIMD offload
  - Automatic/transparent offload when profitable
  - Reduce programming complexity
  - Generalize and increase applicability
- **Call to action**: Updating compilers to support SIMD offload
  - Auto-vectorizer cost-model – updates to vectorization models, candidates, heuristics
  - Rethink instruction costs – mode and implementation dependent
  - Non-linear performance scaling – offload overheads

Merci

Danke

Gracias

Grazie

谢谢

ありがとう

Asante

Thank You

감사합니다

धन्यवाद

Kiitos

شكرًا

ধন্যবাদ

תודה

ధన్యవాదములు

Köszönöm