arm

New Technologies in the Arm Architecture

Linaro Connect BKK19-202

Nigel Stephens, Fellow, Arm Ltd

April 2, 2019

Introduction

- The Armv8-A architecture advances annually with relatively small-scale “ticks”.
  - Armv8.5-A is the latest version, announced in September 2018.

Introduction

- The Armv8-A architecture advances annually with relatively small-scale “ticks”.
  - Armv8.5-A is the latest version, announced in September 2018.

- But this presentation is not about the next annual architecture tick.
  - Some architectural features take much longer than the annual cadence to research and develop.

Introduction

- The Armv8-A architecture advances annually with relatively small-scale “ticks”.
  - Armv8.5-A is the latest version, announced in September 2018.
- But this presentation is not about the next annual architecture tick.
  - Some architectural features take much longer than the annual cadence to research and develop.
- Arm is announcing two major new architecture technologies for performance scaling.
  - The outcome of several years of architecture development.

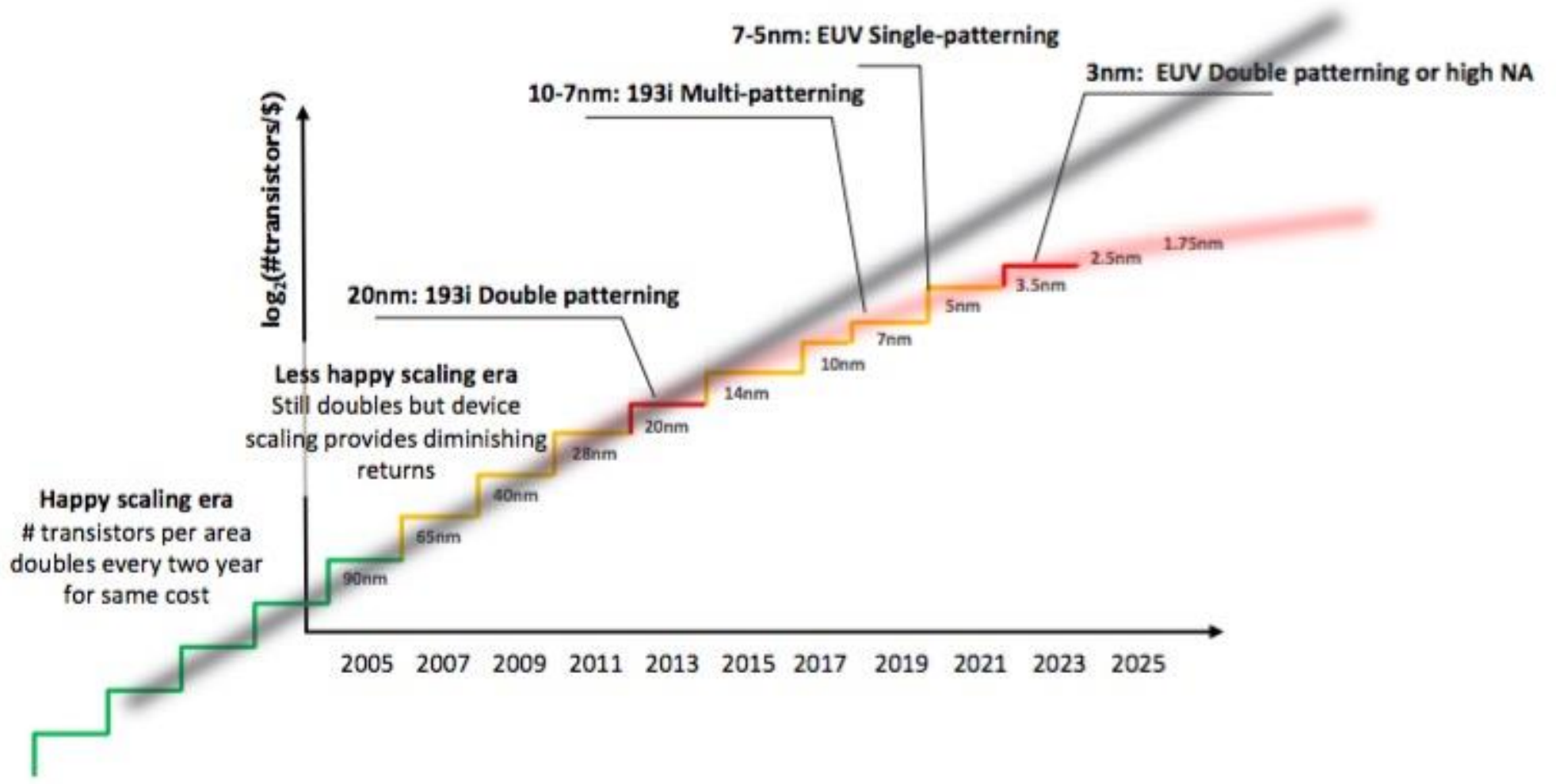
Introduction

- The Armv8-A architecture advances annually with relatively small-scale “ticks”.
  - Armv8.5-A is the latest version, announced in September 2018.
- But this presentation is not about the next annual architecture tick.
  - Some architectural features take much longer than the annual cadence to research and develop.
- Arm is announcing two major new architecture technologies for performance scaling.
  - The outcome of several years of architecture development.
- These new technologies are not yet part of any announced product roadmap.
  - But guidance to developers to prepare for future Arm architecture and CPU products.

Introduction

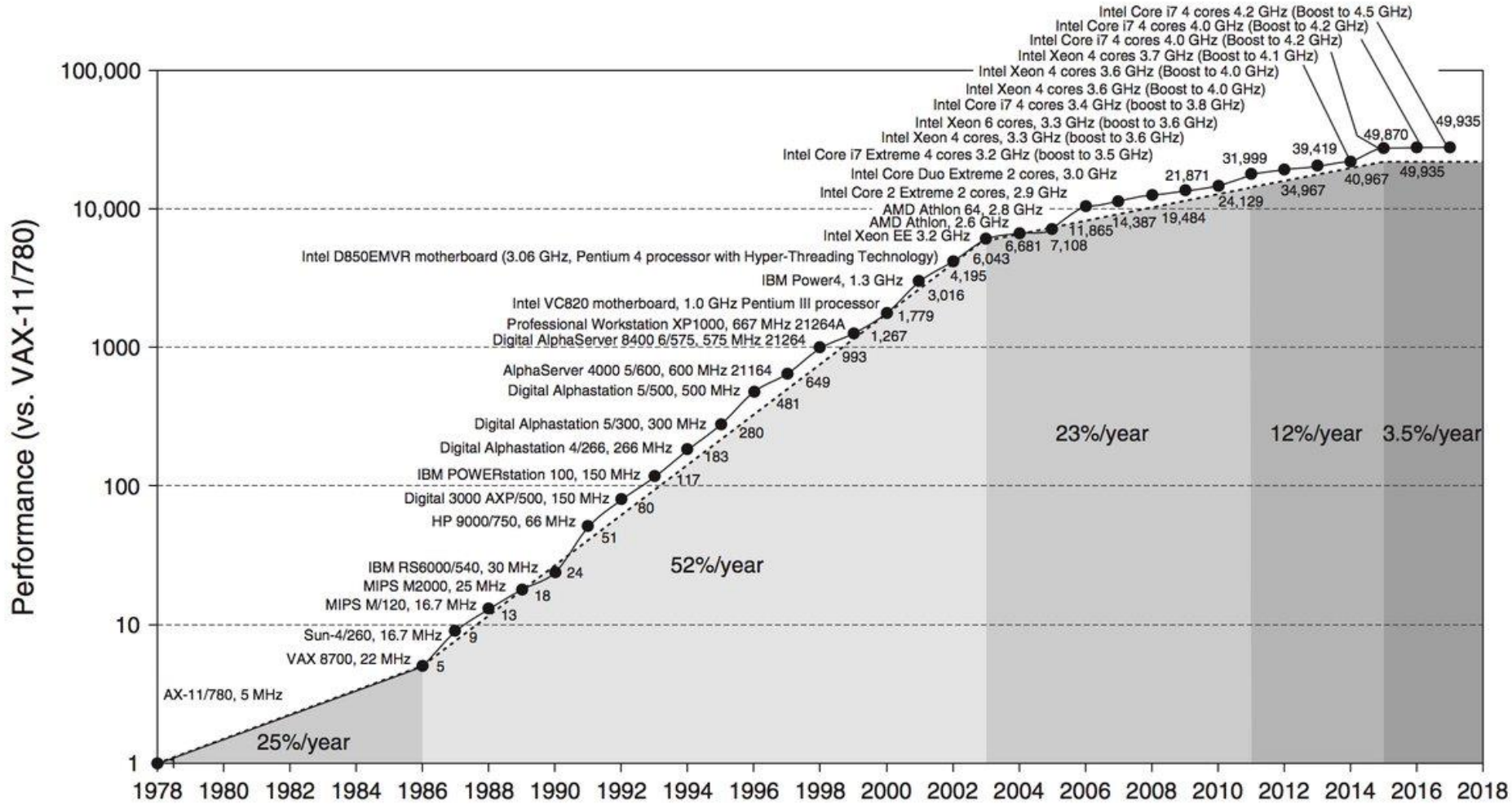
- The Armv8-A architecture advances annually with relatively small-scale “ticks”.
  - Armv8.5-A is the latest version, announced in September 2018.
- But this presentation is not about the next annual architecture tick.
  - Some architectural features take much longer than the annual cadence to research and develop.
- Arm is announcing two major new architecture technologies for performance scaling.
  - The outcome of several years of architecture development.
- These new technologies are not yet part of any announced product roadmap.
  - But guidance to developers to prepare for future Arm architecture and CPU products.
- Preparing the software ecosystem for complex new technologies can take a long time.
  - To allow this work to begin, the new instruction sets will be published by the end of next week.

Moore’s Law for transistor area scaling



arm

Moore’s Law for single-thread performance



arm

Addressing Moore's Law

- Can we use the additional transistors to unlock more CPU performance?
  - By processing more instructions or more data in parallel per cycle.
- New *microarchitecture* can extract more Instruction-Level Parallelism (ILP).
  - But there are limits to this hardware magic.
- Could new *architecture* allow us to express greater parallelism in our code?
  - Tackling the unbending nature of Amdahl's Law requires far more parallelisation.
  - But without needing to rewrite the world's software.

Scalable Vector Extension v2 (SVE2)

Scalable Data-Level Parallelism (DLP) for more applications

Built on SVE

Improved scalability

Vectorization of more workloads

Built on the SVE foundation.

- Scalable vectors with hardware choice from 128 to 2048 bits.
- Vector-length agnostic programming for “write once, run anywhere”.
- Predication and gather/scatter allows more code to be vectorized.
- Tackles some obstacles to compiler auto-vectorisation.

Scaling single-thread performance to exploit long vectors.

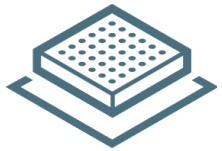
- SVE2 adds NEON™-style fixed-point DSP/multimedia plus other new features.
- Performance parity and beyond with classic NEON DSP/media SIMD.
- Tackles further obstacles to compiler auto-vectorization.

Enables vectorization of a wider range of applications than SVE.

- Multiple use cases in Client, Edge, Server and HPC.
  - DSP, Codecs/filters, Computer vision, Photography, Game physics, AR/VR, Networking, Baseband, Database, Cryptography, Genomics, Web serving.
- Improves competitiveness of Arm-based CPU vs proprietary solutions.
- Reduces s/w development time and effort.

Transactional Memory Extension (TME)

Scalable Thread-Level Parallelism (TLP) for multi-threaded applications



Hardware Transactional Memory

Improved scalability

Simpler software design

Hardware Transactional Memory (HTM) for the Arm architecture.

- Improved competitiveness with other architectures that support HTM.
- Strong isolation between threads.
- Failure atomicity.

Scaling multi-thread performance to exploit many-core designs.

- Database.
- Network dataplane.
- Dynamic web serving.

Simplifies software design for massively multi-threaded code.

- Supports Transactional Lock Elision (TLE) for existing locking code.
- Low-level concurrent access to shared data is easier to write and debug.

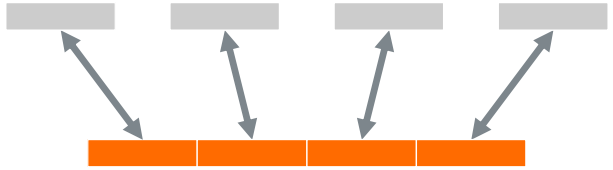
arm

arm

SVE Recap

SVE features #1

HPC vector extension announced in 2016



Gather-load and scatter-store.

- Transfer a single vector from/to a vector of addresses.
- Permits vectorization of non-linear data structures.

	1	2	3	4
+	5	5	5	5
pred	T	F	T	F
=	6	2	8	4

Per-lane predication.

- Operate on independent lanes under a predicate register.
- Permits vectorization of complex control flows & nested loops.

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

WHILELT i,n	n-2	n-1	n	n+1
B.FIRST loop	T	T	F	F

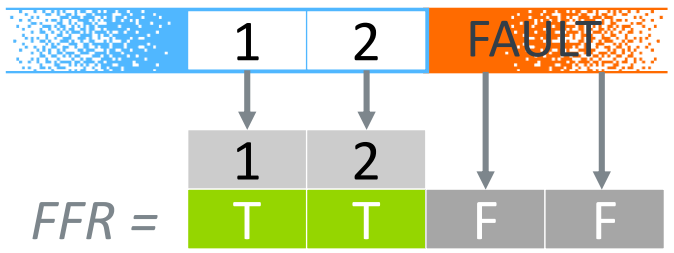
Predicate-driven loop control and management.

- Eliminate loop head/tail and other vectorization overheads.
- Permits more aggressive vectorization at -O2.

arm

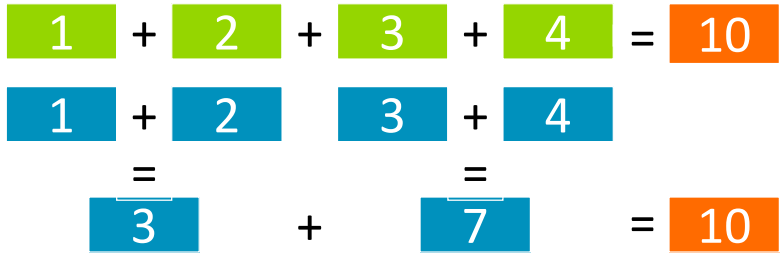
SVE features #2

HPC vector extension announced in 2016



Software-managed speculative vectorization.

- First-faulting vector load allows memory accesses to safely cross into invalid pages.
- Permits vectorization of data-dependent while/break loops.



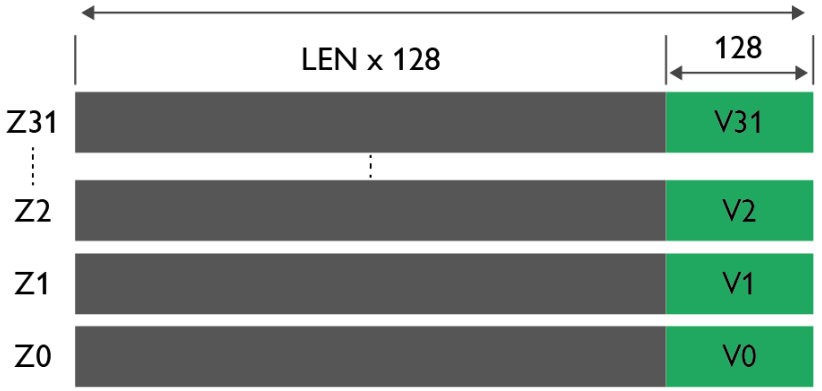
Additional vector data-processing.

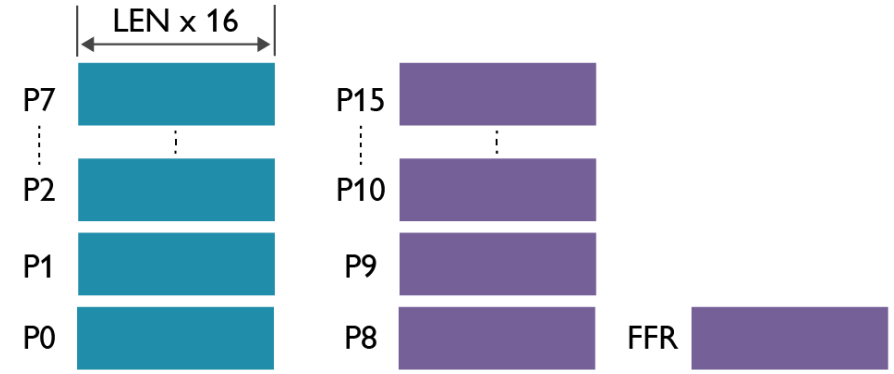
- In-order and tree-based floating-point reductions.
- Integer and bitwise logical reductions.
- Integer and FP vector divide & four operand multiply-add.
- Vector trig (sin/cos/exp), etc., etc.

arm

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.





arm

Scalar SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    b       .latch

.loop:
    ldr     s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr     s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd   s2, s1, s0, s2          // s2+=x[i]*a
    str     s2, [x1,x4,lsl 2]       // y[i]=s2
    add     x4, x4, #1              // i+=1
.latch:
    cmp     x4, x2                  // i < n
    b.lt    .loop                   // more to do?
    ret
```

Scalar SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    b       .latch

.loop:
    ldr     s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr     s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd   s2, s1, s0, s2          // s2+=x[i]*a
    str     s2, [x1,x4,lsl 2]       // y[i]=s2
    add     x4, x4, #1              // i+=1
.latch:
    cmp     x4, x2                  // i < n
    b.lt    .loop                   // more to do?
    ret
```


arm

Scalar SAXPY

```
// ------------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ------------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    b       .latch

.loop:
    ldr     s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr     s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd   s2, s1, s0, s2          // s2+=x[i]*a
    str     s2, [x1,x4,lsl 2]       // y[i]=s2
    add     x4, x4, #1              // i+=1
.latch:
    cmp     x4, x2                  // i < n
    b.lt    .loop                   // more to do?
    ret
```

SVE SAXPY

```
// ------------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ------------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                      // x4=i=0
    whilelt p0.s, xzr, x2               // p0=while(i++<n)
    dup     z0.s, s0                    // z0=dup(A)
.loop:
    ld1w    z1.s, p0/z, [x0,x4,lsl 2]   // p0:z1=x[i]
    ld1w    z2.s, p0/z, [x1,x4,lsl 2]   // p0:z2=y[i]
    fmla    z2.s, p0/m, z1.s, z0.s      // p0?z2+=x[i]*a
    st1w    z2.s, p0, [x1,x4,lsl 2]     // p0?y[i]=z2
    sqincw  x4                          // i+=(VL/32)

    whilelt p0.s, x4, x2                // p0=while(i++<n)
    b.first .loop                       // more to do?
    ret
```


arm

Scalar SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    b       .latch

.loop:
    ldr     s1, [x0,x4,lsl 2]       // s1=x[i]
    ldr     s2, [x1,x4,lsl 2]       // s2=y[i]
    fmadd   s2, s1, s0, s2          // s2+=x[i]*a
    str     s2, [x1,x4,lsl 2]       // y[i]=s2
    add     x4, x4, #1              // i+=1
.latch:
    cmp     x4, x2                  // i < n
    b.lt    .loop                   // more to do?
    ret
```

SVE SAXPY

```
// ----------------------------------------
// void saxpy(const float X[], float Y[],
//            float A, int N) {
//   for (int i = 0; i < N; i++)
//     Y[i] = A * X[i] + Y[i];
// }
// ----------------------------------------
// x0 = &X[0], x1 = &Y[0], s0 = A, x2 = N
saxpy:
    mov     x4, #0                  // x4=i=0
    whilelt p0.s, xzr, x2           // p0=while(i++<n)
    dup     z0.s, s0                // z0=dup(A)
.loop:
    ld1w    z1.s, p0/z, [x0,x4,lsl 2] // p0:z1=x[i]
    ld1w    z2.s, p0/z, [x1,x4,lsl 2] // p0:z2=y[i]
    fmla    z2.s, p0/m, z1.s, z0.s  // p0?z2+=x[i]*a
    st1w    z2.s, p0, [x1,x4,lsl 2] // p0?y[i]=z2
    sqincw  x4                      // i+=(VL/32)

    whilelt p0.s, x4, x2            // p0=while(i++<n)
    b.first .loop                   // more to do?
    ret
```


arm

SAXPY using ACLE vector intrinsics

Arm C Language Extensions for NEON and SVE

NEON ACLE intrinsics

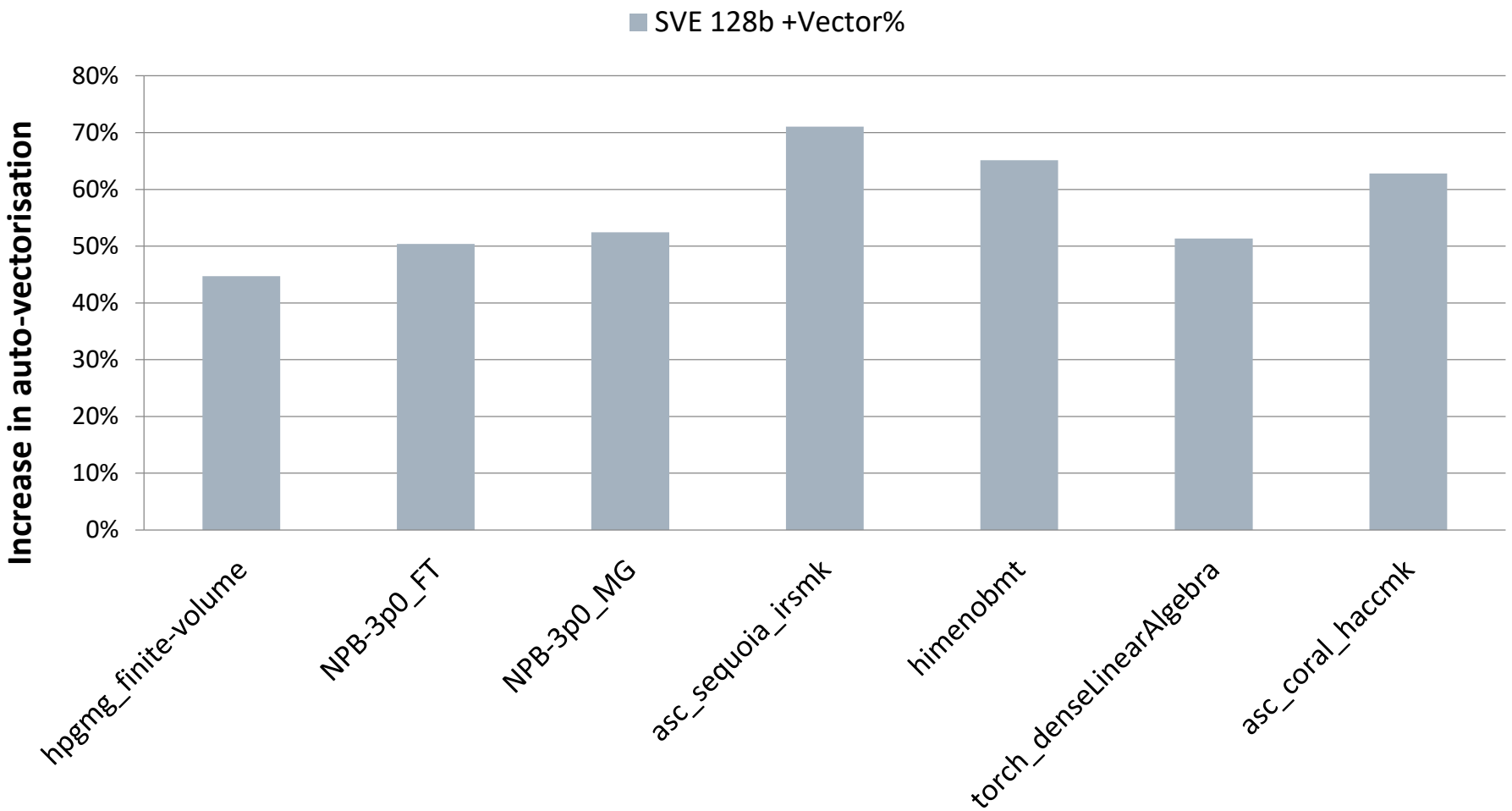
```
void saxpy(const float X[], float Y[],
           float A, int N) {
  for (; N % 4 != 0; N--)
    *Y++ += *X++ * A; // scalar loop head
  for (i = 0; i < N; i += 4) {
    float32x4_t vx = vld1q_f32(X);
    float32x4_t vy = vld1q_f32(Y);
    vy = vmlaq_f32(vy, vx, A);
    vst1q_f32(Y, vy);
    X += 4; Y += 4;
  }
}
```

SVE ACLE intrinsics

```
void saxpy(const float X[], float Y[],
           float A, int N) {
for (i = 0; i < N; i += svcntw()) {
    svbool_t part = svwhilelt_b32(i, N);
    svfloat32_t vx = svld1(part, &X[i]);
    svfloat32_t vy = svld1(part, &Y[i]);
    vy = svmla(part, vy, vx, A);
    svst1(part, &Y[i], vy);
  }
}
```

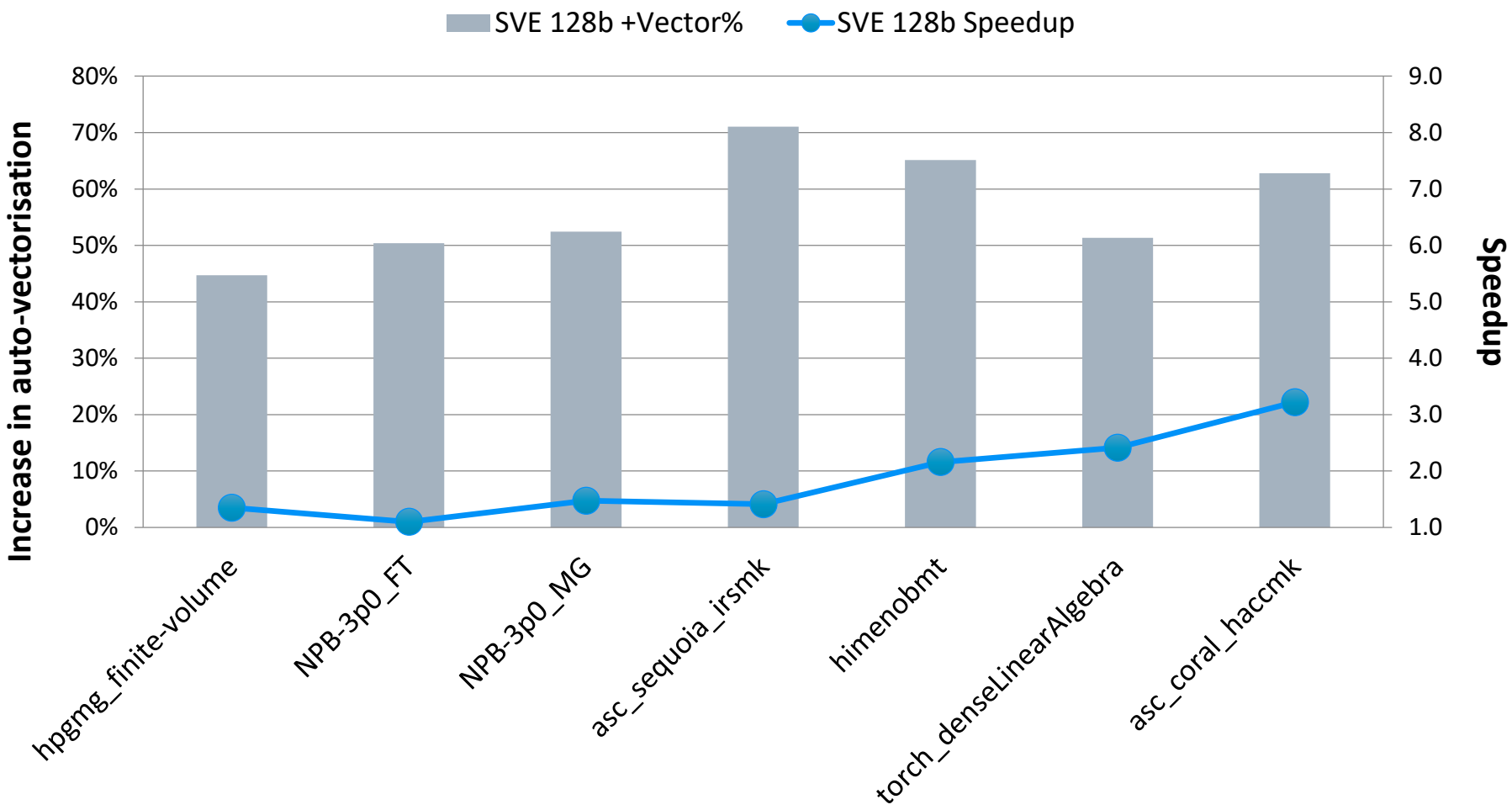

arm

HPC performance SVE vs. NEON



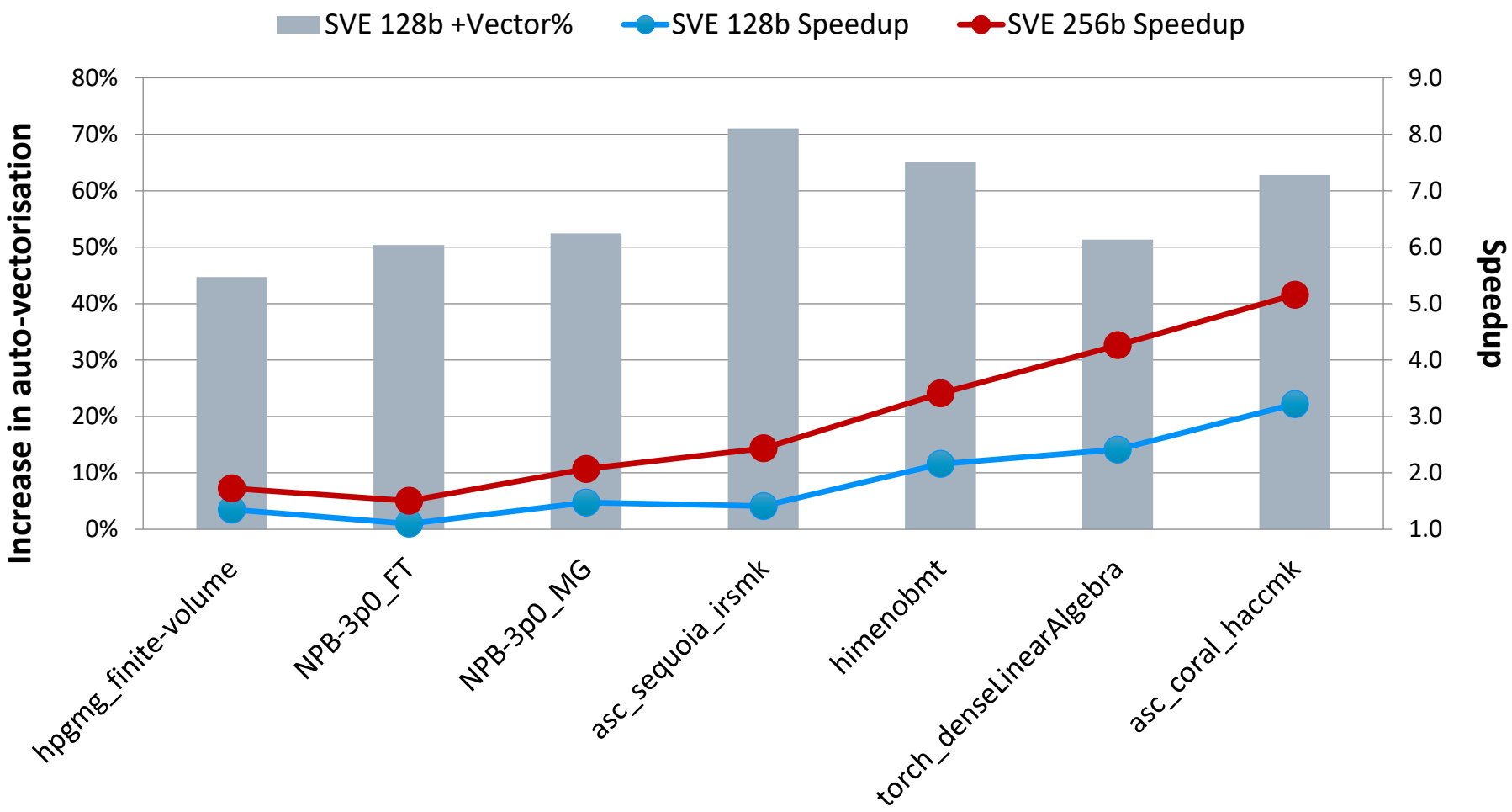
arm

HPC performance SVE vs. NEON



arm

HPC performance SVE vs. NEON



arm

HPC performance SVE vs. NEON

arm

arm

Introducing SVE2

SVE2 enhancements

- NEON-style “DSP” instructions
  - Trad NEON fixed-p, widen, narrow & pairwise ops
  - Fixed-point complex dot product, etc. (LTE)
  - Interleaved add w/ carry (wide multiply, BigNums)
  - Multi-register table lookup (LTE, CV, shuffle)
  - Enhanced vector extract (FIR, FFT)
- Cross-lane match detect / count
  - In-memory histograms (CV, HPC, sorting)
  - In-register histograms (CV, G/S pointer de-alias)
  - Multi-character search (parsers, packet inspection)
- Non-temporal Gather / Scatter
  - Explicit cache segregation (CV, HPC, sorting)
- Bitwise operations
  - PMULL32→64, EORBT, EORTB (CRC, ECC, etc.)
  - BCAX, BSL, EOR3, XAR (ternary logic + rotate)
- Bit shuffle
  - *BDEP*, *BEXT*, *BGRP* (LTE, compression, genomics)
- Cryptography

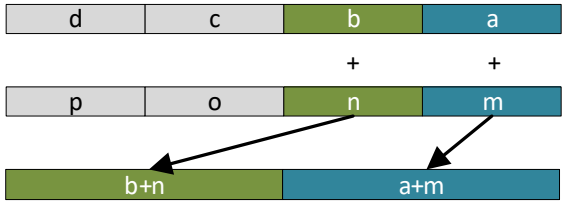
  Optional
  - *AES*, *SM4*, *SHA3*, *PMULL*64→128
- Miscellaneous vectorisation
  - WHILEGE/GT/HI/HS (down-counting loops)
  - WHILEWR/RW (contiguous pointer de-alias)
  - FLOGB (other vector trig)
- ID register changes only for SVE Linux kernel

arm

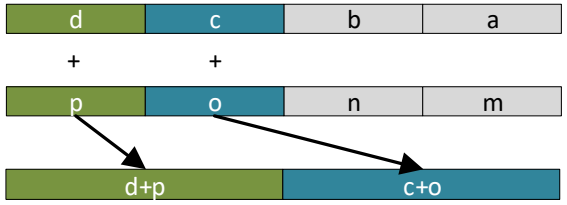
SVE2 widening & narrowing vs. NEON

NEON

UADDL Vd.2D, Vn.2S, Vm.2S



UADDL2 Vd.2D, Vn.4S, Vm.4S



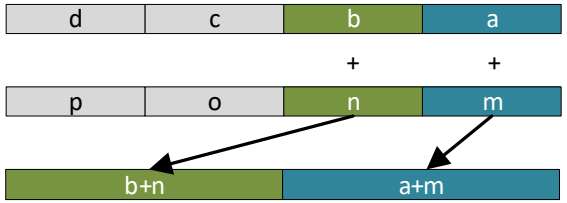
- NEON uses high/low half of vector
  - Too costly for vectors >>128b

arm

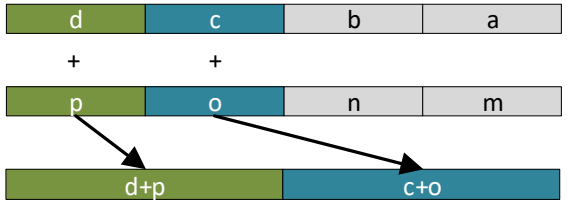
SVE2 widening & narrowing vs. NEON

NEON

UADDL Vd.2D, Vn.2S, Vm.2S

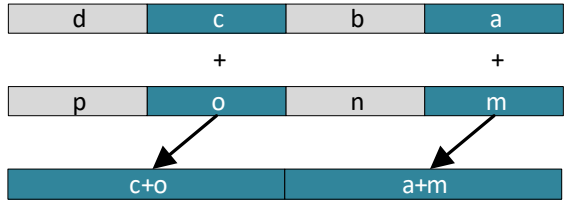


UADDL2 Vd.2D, Vn.4S, Vm.4S

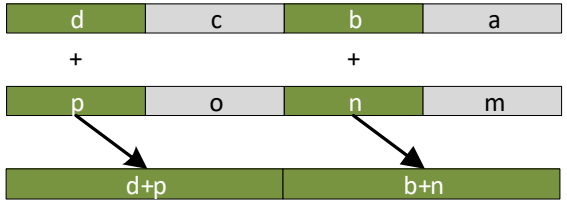


SVE2

UADDLB Zd.D, Zn.S, Zm.S



UADDLT Zd.D, Zn.S, Zm.S

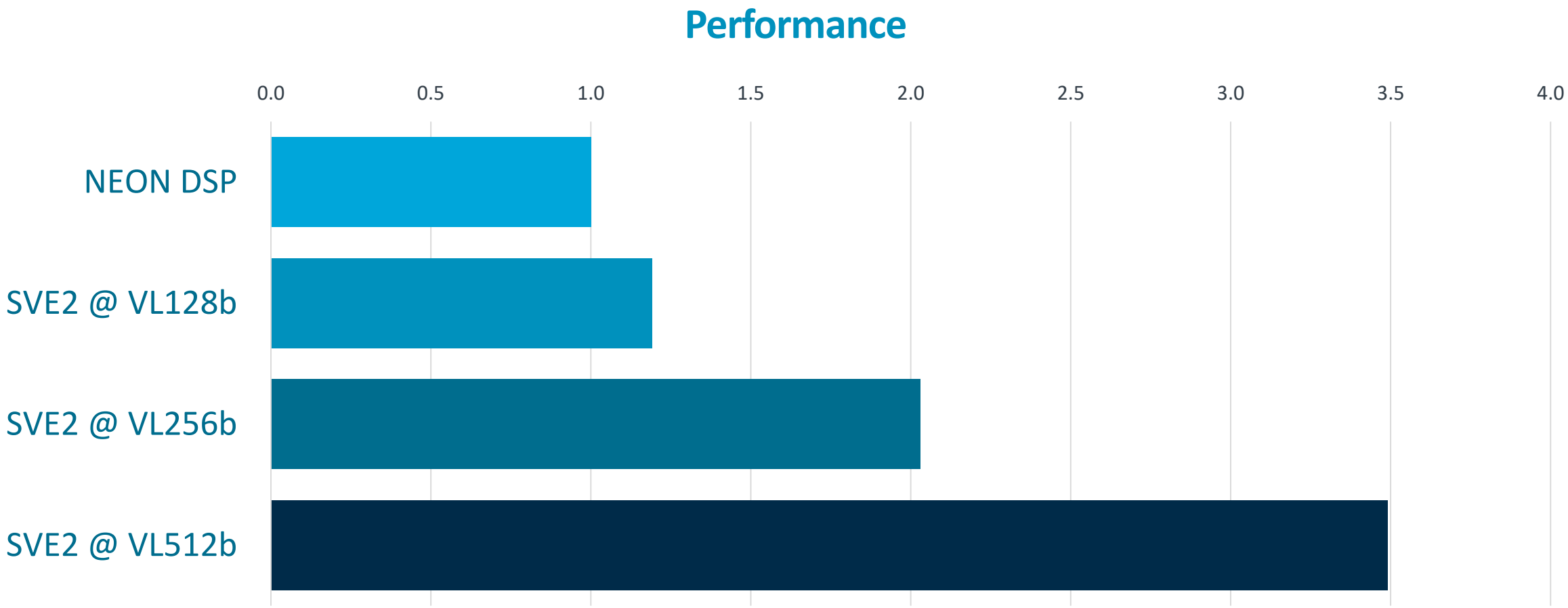


- NEON uses high/low half of vector
  - Too costly for vectors >>128b
- SVE2 uses even/odd elements
  - Bottom & top variants of instructions
  - Widens and narrows “in lane”
- Widening deinterleaves the inputs
  - But narrowing reinterleaves again
  - And reduction is order-insensitive

arm

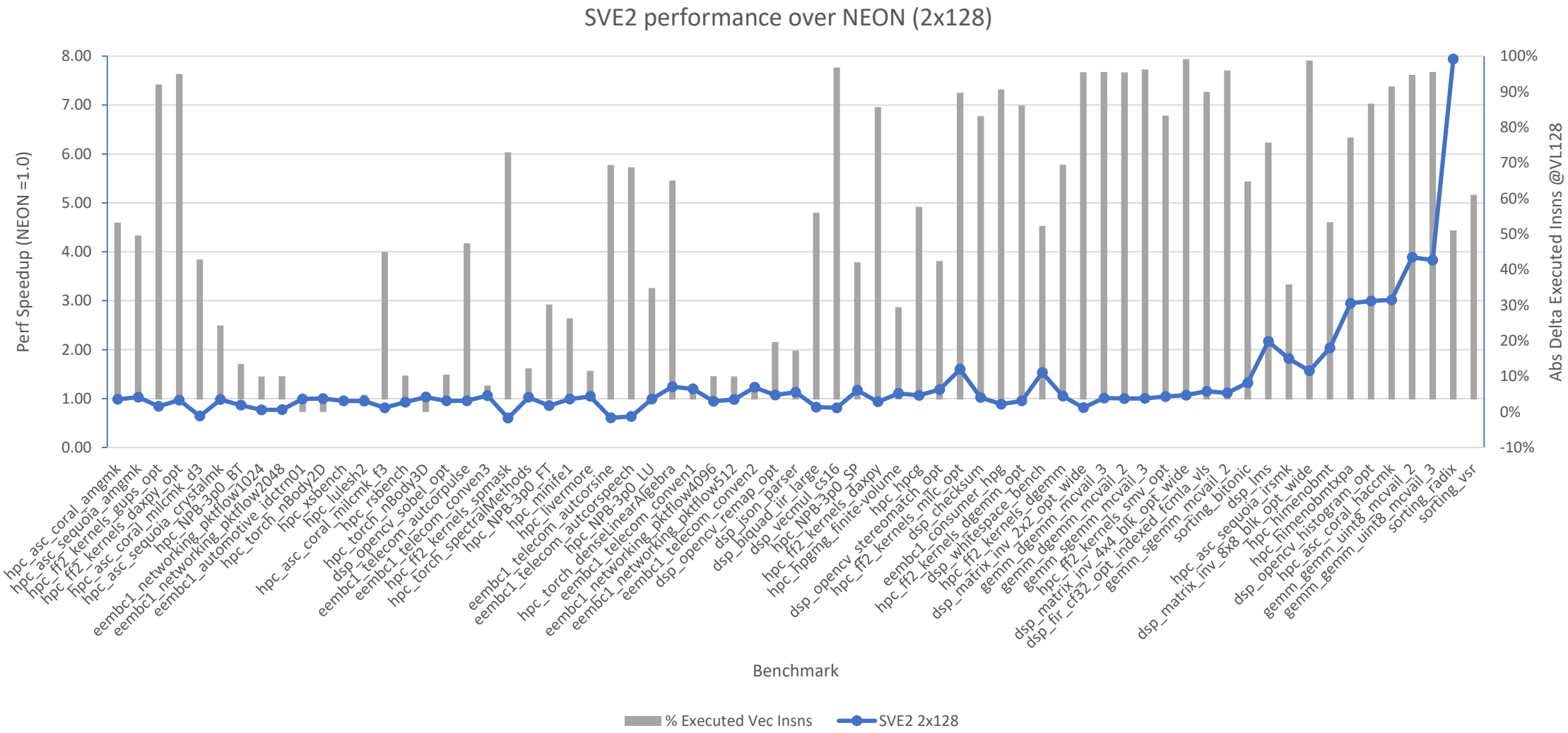
DSP/Multimedia performance SVE2 vs. NEON

Parity and beyond with traditional NEON DSP/Media workloads



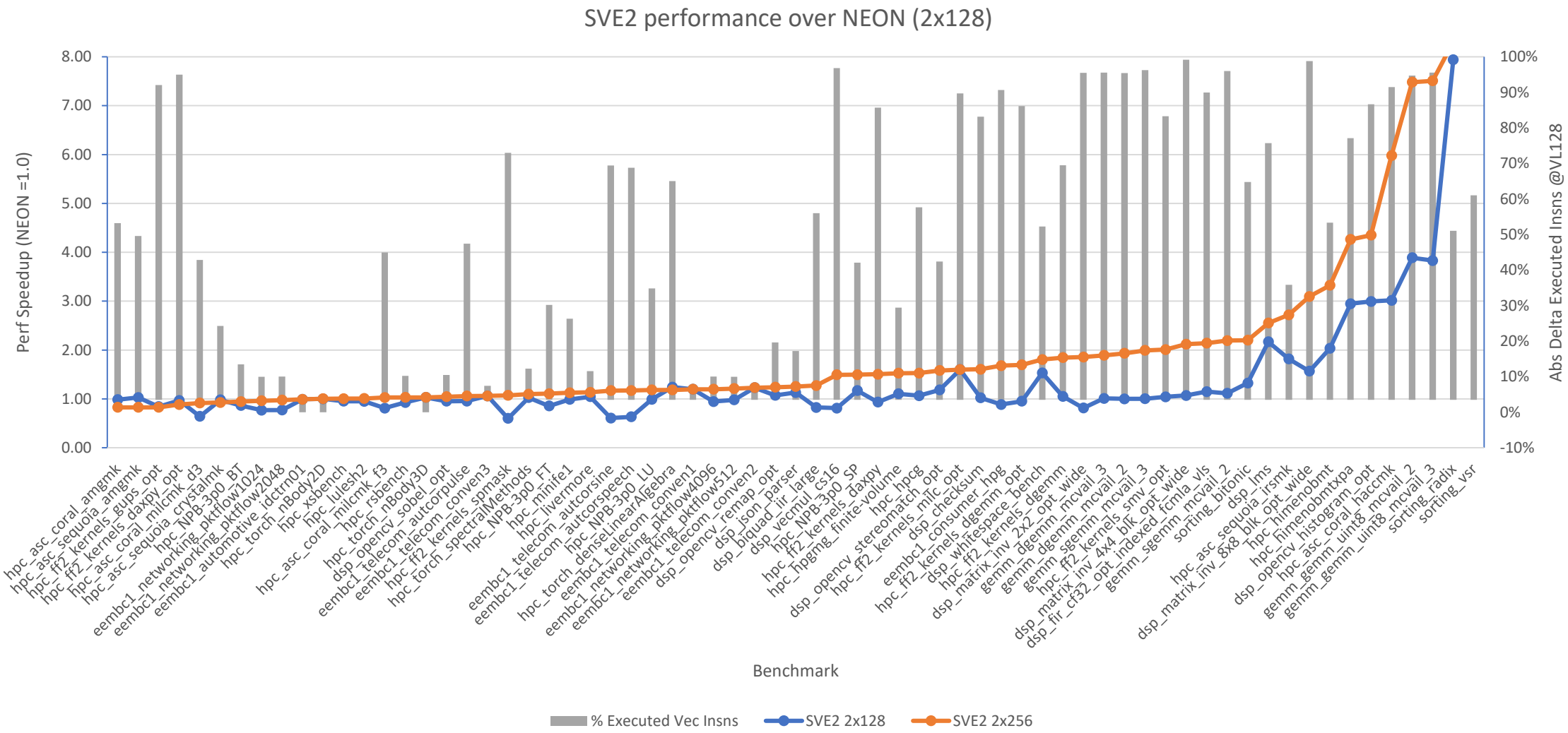
arm

General performance SVE2 vs. NEON



arm

General performance SVE2 vs. NEON



arm

Introducing TME

Arm Transactional Memory Extension (TME)

Programmer writes

```
tstart  x0
cbnz    x0, fallback

// transactional code here

tcommit
```

Hardware provides

- Strong isolation.
  - Non-interference & containment from both transactional and non-transactional code.
- Failure atomicity.
  - Architectural changes discarded on failure.
  - All instructions commit or none.
- Best-effort transactions.
  - No forward progress guarantee, SW must provide non-transactional fallback path.
  - Good for multi-client “server” applications with large, rarely contended data structures.

arm

TME transaction failure causes

- Causes
  - **DBG** A debug-related exception was encountered but not raised.
  - **NEST** The maximum transactional nesting depth was exceeded.
  - **SIZE** The transactional read set limit or the transactional write set limit was exceeded.
  - **ERR** An operation architecturally not permitted in Transactional state was attempted.
  - **MEM** A transactional memory conflict was detected.
  - **CNCL** The transaction was canceled by a TCANCEL instruction.
  - **TRIVIAL** The system is currently running with the trivial TM implementation enabled.
  - **IMP** Any other failure cause.
- Other bits
  - **RTRY** Only set if the transaction might succeed on retrying.
  - **INT** An unmasked interrupt was delivered.
  - **REASON** From the TCANCEL operand.

arm

Transactional Lock Elision (TLE)

Killer use-case for Hardware Transactional Memory (HTM)

- Execute a lock-protected critical section transactionally.
  - Replaces the lock/unlock by TSTART/TCOMMIT.
  - Same programming model as locks, but the critical sections can now run concurrently.
  - Only falls back to a lock/unlock if there is a conflict (after user attempts any retries).
  - Fallback code is the same as the transactional code but runs with the mutex locked.
  - GNU C library already supports this for `pthread_mutex_lock()` on other architectures.

- Benefits of TLE.
  - Concurrent execution of critical regions (when no conflict) gives higher throughput.
  - Lock thrashing: lock state is not modified, so each processor can hold a local copy.
  - Lock preemption: swapping out a thread that “holds” an elided lock does not block other threads.

Example TLE implementation

lock(x)

```
do {
    while (is_locked(x)) /* wait for legacy */;
    status = __tstart();
    if (status == IN_TRANSACTION) {
        if (is_locked(x))
            /* lost race with legacy */
            __tcancel(LOCKED);
        return;
    }
    /* transaction failed */
} while (ok_to_retry(status));
legacy_lock(x);
```

Heuristic based on failure cause, retry count, previous behavior, etc.

unlock(x)

```
if (is_locked(x)) {
    legacy_unlock(x);
} else {
    __tcommit();
}
```

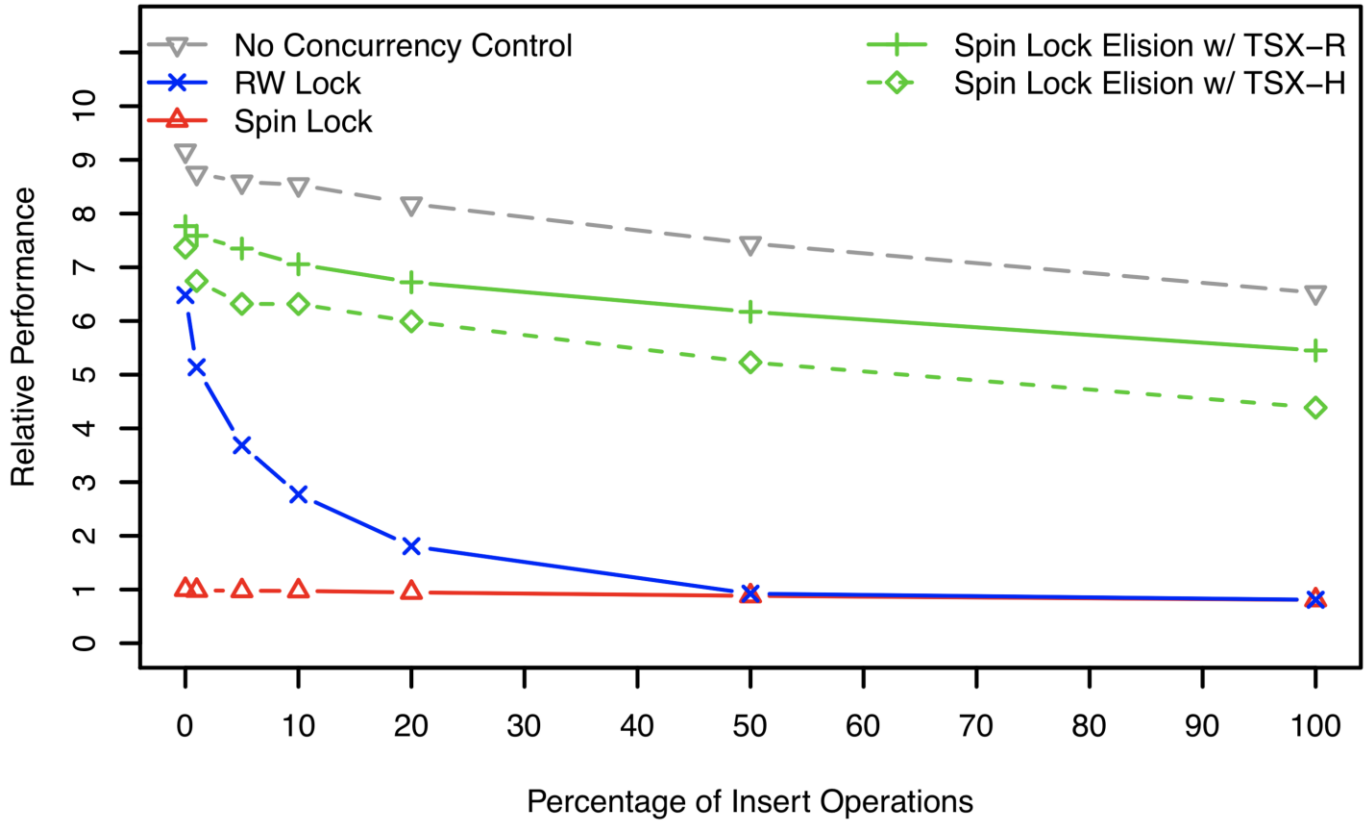
Unlocking an untaken lock generates an exception**

** “Fixed a bug in the EGL driver where a mutex was unlocked more than once. […] if lock elision is enabled in glibc, may result in a segmentation fault.” https://devtalk.nvidia.com/default/topic/908423

arm

SAP-HANA: B+Tree / Delta Storage

Real-world example using hardware transactions for database indexing



Intel(R) Core(TM) i7-4770 @ 3.4 GHz (8 threads, 4 processors)

"Perhaps the most urgently needed future direction is simplification. Functionality and code for concurrency control and recovery are too complex to design, implement, test, debug, tune, explain, and maintain."

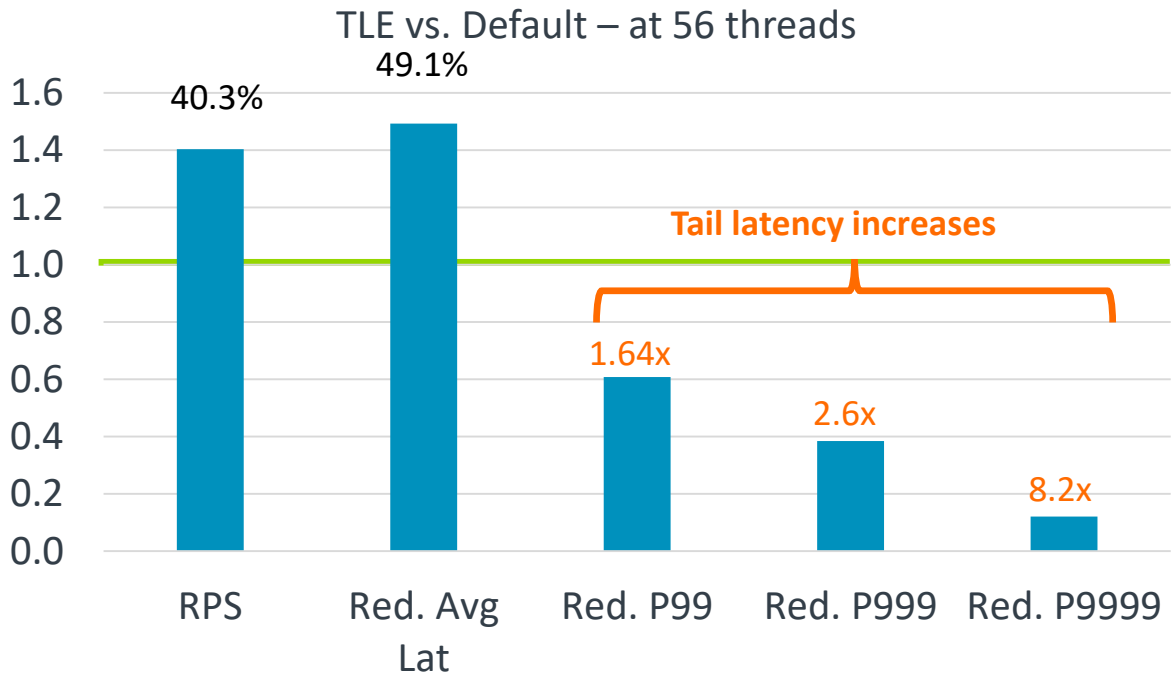
T. Karnagel, R. Dementiev, R. Rajwar, K. Lai, T. Ledger, B. Schlegel, W. Lehner.
Improving In-Memory Database Index Performance with Intel® Transactional Synchronization Extensions. *HPCA'14*

arm

Twemcache: Twitter Memcached

Using Glibc's Pthread mutex locks with TLE – no software changes required

- Twemcache[1] is a heavily modified *memcached*
  - Memcached is a high-performance distributed memory object caching system.
- Local test setup.
  - 56 thread Broadwell server; 28 thread Haswell client (not isolated).
- Throughput at scale with TLE.
  - At 56 threads: 40% more requests/second, 49% reduction in average request latency
  - Some increase in tail latencies, under investigation.



Intel(R) Xeon(R) CPU E5-2690 v4 @ 2.60GHz
(56 threads, 28 processors, 2 sockets)

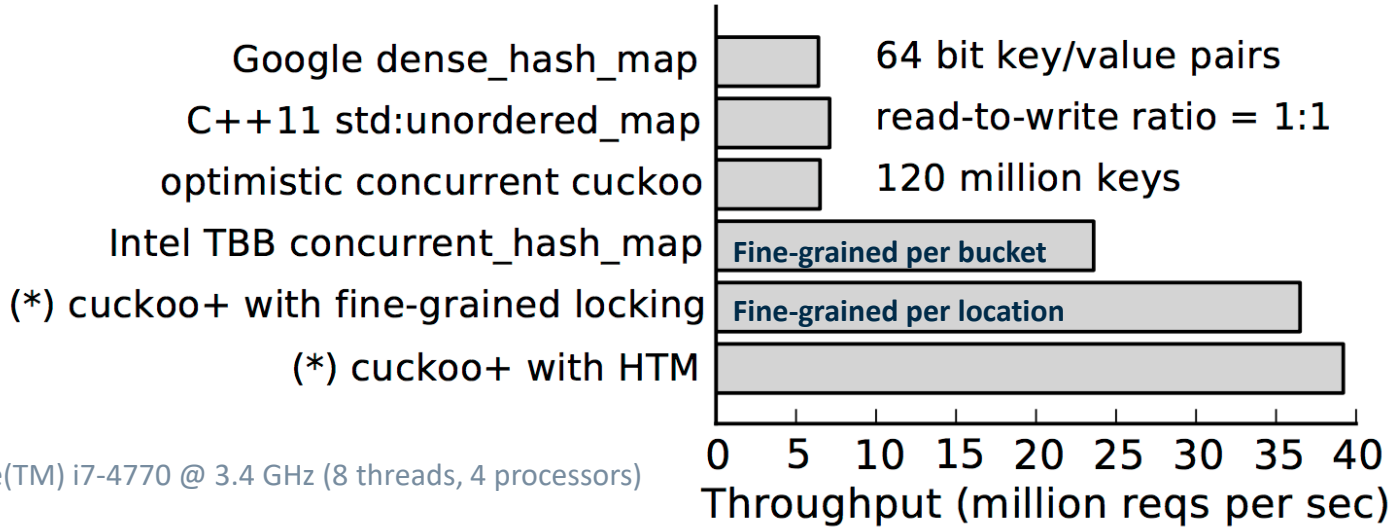
[1] https://github.com/twitter/twemcache

arm

DPDK cuckoo+

Hardware transactions for network data-plane processing

DPDK's flow classification index uses a concurrent cuckoo+ hash



Intel(R) Core(TM) i7-4770 @ 3.4 GHz (8 threads, 4 processors)

- "Our results about TSX can be interpreted in two ways. On one hand, in almost all of our experiments, hardware transactional memory provided a modest but significant speedup over either global locking or our best-engineered fine-grained locking, and it was easy to use. This confirms other recent results showing, e.g., a "free" 1.4x speedup from using TSX in HPC workloads [22]. On the other hand, the benefits of data structure engineering for efficient concurrent access contributed substantially more to improving performance, but also required deep algorithmic changes to the point of being a research contribution on their own."

Li et al. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. *EUROSYS'14*

arm

arm

Conclusions

Conclusions

- Arm is announcing two new technologies for the Arm architecture.
  - New ways to scale performance and exploit additional transistors with Moore's Law slowing.
  - Extracting more parallelism from existing software.
- SVE2: improved auto-vectorization with support for DSP/Media hand-coded SIMD.
  - Scalable vectorization for increased fine-grain Data Level Parallelism (DLP).
  - More work done per instruction.
- TME: easier lock-free programming for lightly-contended shared data structures.
  - Scalable concurrency to increase coarse-grain Thread Level Parallelism (TLP).
  - More done work per thread.
- SVE2 and TME may be able to combine for even greater performance scaling.
  - Tackling Amdahl's law on multiple fronts with a mix of DLP and TLP in multi-threaded applications.

Enabling tools and documentation

Upstreaming to begin immediately

LLVM

- Upstreaming of SVE2/TME assembly support to begin immediately.
- Goal of initial SVE2 auto-vectorization & ACLE upstream by end Q1 CY20.

GNU Tools

- Upstreaming of SVE2/TME assembly, initial SVE2 auto-vectorization & ACLE to begin immediately. (SVE autovec present since GCC8).
- Targeting GCC10 release at end Q1 CY20.

Glibc

- Aiming for Transactional Lock Elision support in upstream Glibc by Q3 CY19.

Arm Tools

- SVE2/TME support in Arm compiler, debugger and fast models planned for H2 CY2019.

Documentation

- SVE2/TME ISA XML available by 15 April 2019.
  - developer.arm.com/architectures.
- SVE literature at developer.arm.com/hpc.
  - No ABI changes required by SVE2.
- SVE2 literature – including VLA programmer's guide with code examples – available soon.

arm

arm

Thank You
Danke
Merci
谢谢
ありがとう
Gracias
Kiitos
감사합니다
धन्यवाद
شكرًا
תודה