Implementing C++26 `std::simd`

A Layered, Compiler-First Approach

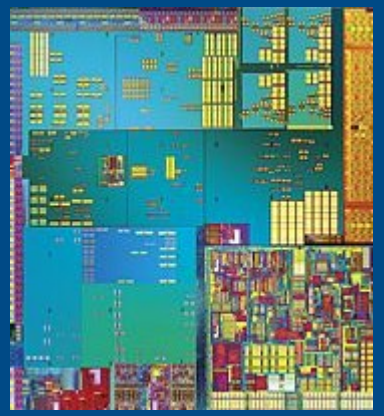
Dr Daniel Towner, Principal Systems Engineer

intel®

Motivation for `std::simd`

- For 3 decades or more, processors have delivered ever-increasing parallel capability
  - - not through threading, but through data-parallelism
- Data-parallelism now surrounds us, from phones to cloud compute.
- A myriad of C++ libraries support data-parallelism, but none are standardised, portable and performant everywhere…
- ….until now

Agenda



What is hardware SIMD?

```
auto f(vec<float> x) {
    return reduce(x + 1.0f);
}
```

What is `std::simd`?

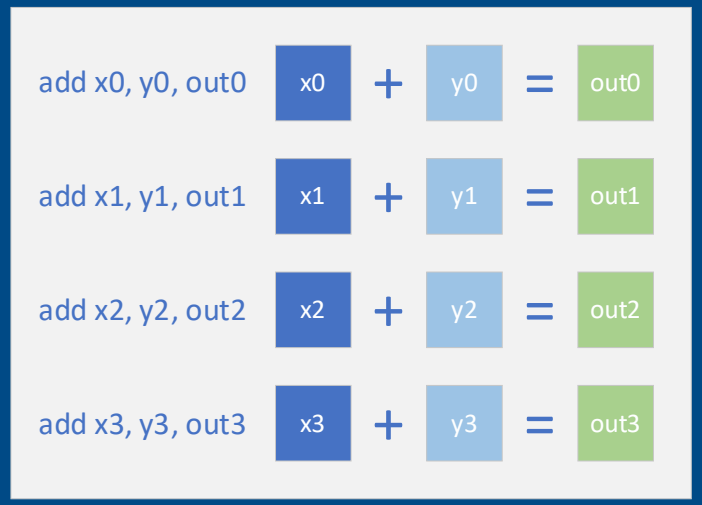
How is `std::simd` implemented in LLVM?

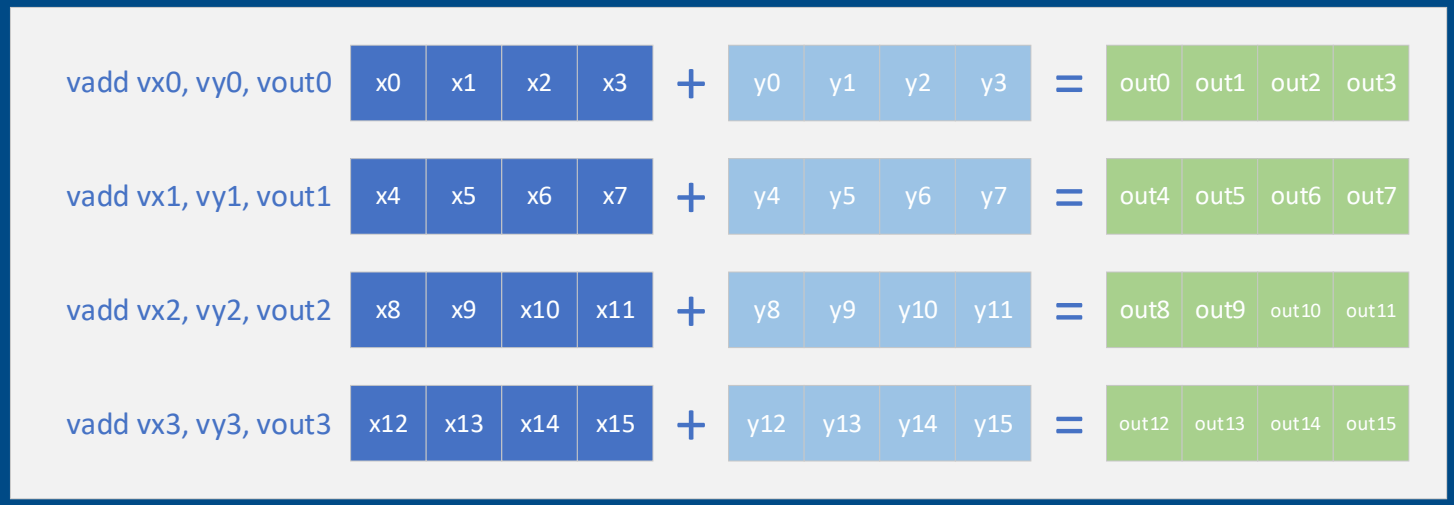
What is SIMD?

What can the hardware do, and how do we access it?

Scalar versus parallel – single-instruction, multiple-data

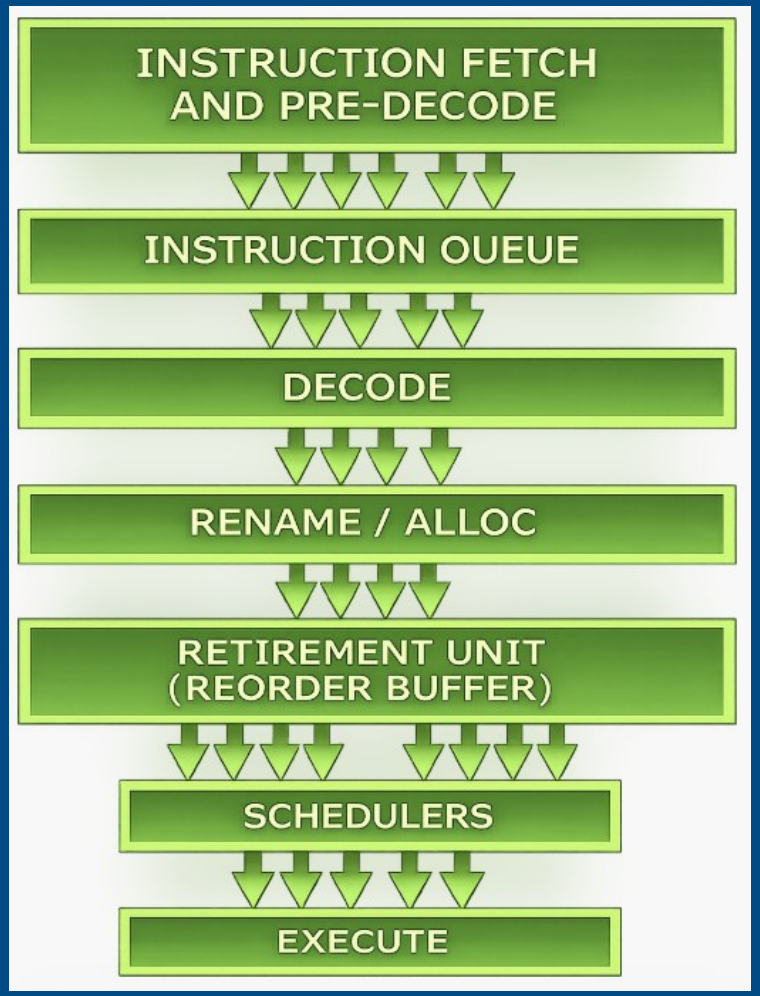
```
auto sum_arrays(const float* x, const float* y, float* out) {
    for (int i=0; i<2048; ++i)
      out[i] = x[i] + y[i];
}
```





Left: 4 instructions, 4 adds
Right: 4 instructions, 16 adds! 4x faster.

Why SIMD exists — amortizing the pipeline tax

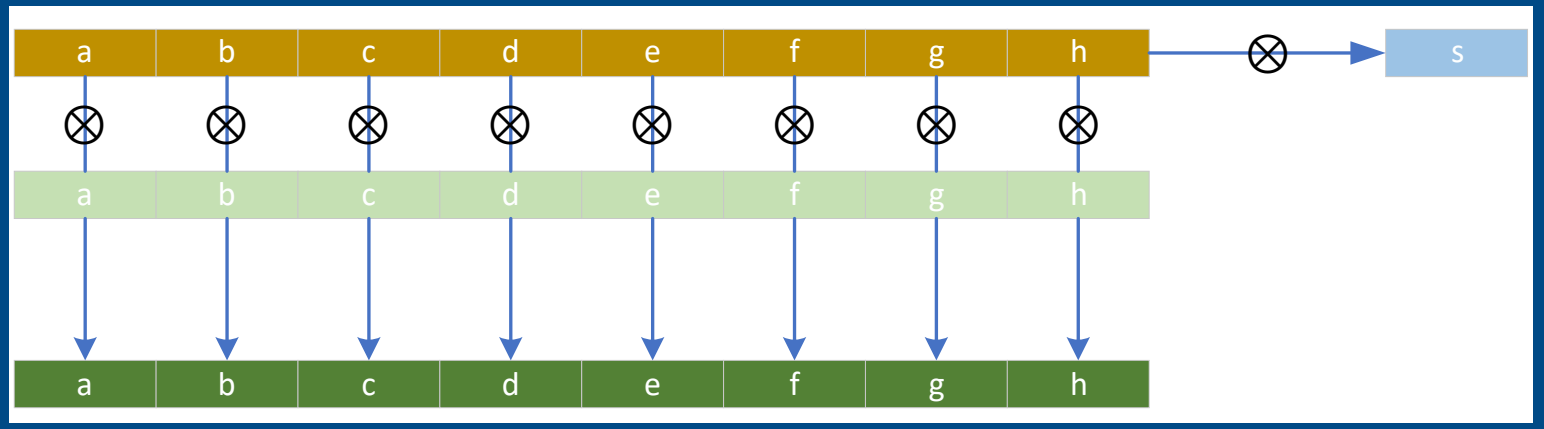


- Every instruction pays a cost:
  - fetch, decode, schedule, speculation, retire
- SIMD amortizes that cost by doing N operations for every instruction, drastically improving performance.
- Also cache-friendly by construction - data is packed together.

- SIMD in hardware is easy. The challenge has always been to make SIMD accessible to the programmer.

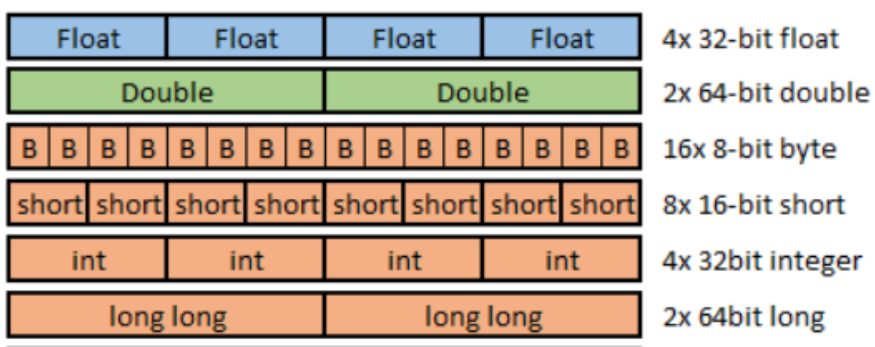
Modes of operations

- There are three major dimensions in which vector instructions can execute:

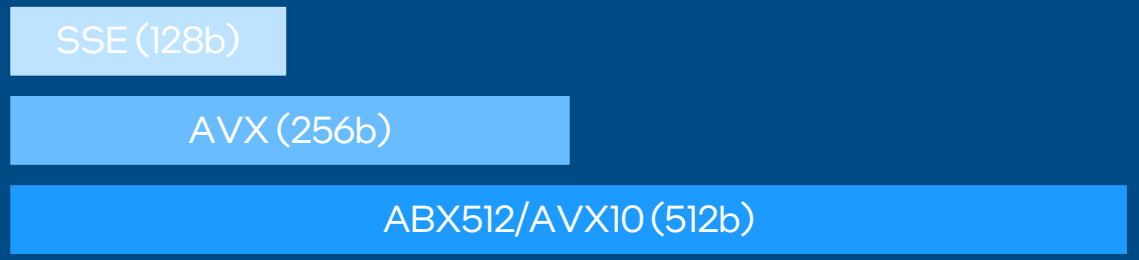


- Vertical/map: Each element generates a new element
- Horizontal/reduce: Each element generates a new element
- Shuffle/permute: reorder the elements

What SIMD can do — operations, widths, masking



The register can be divided up into different numbers and sizes of elements.



Registers have been getting wider over time, increasing efficiency, but also power and complexity.

	Add	Sub	Mul	Div	Shift
16 x 8-bit int					
8 x 16-bit int					
4 x 32-bit int					
4 x 32-bit float					
2 x 64-bit float					

Not all data types are equal – some lack support (SSE)

Masks can be used to deactivate some lanes.

Auto-vectorising

- Auto-vectorisation is able to detect and recover parallelism from sequential code:

```
auto sum_arrays(const float* x,
                const float* y,
                float* out) {
    for (int i=0; i<2048; ++i)
      out[i] = x[i] + y[i];
}
```

```
.L4:
        vmovups ymm1, YMMWORD PTR [rbp+4112+rax]
        vaddps  ymm0, ymm1, YMMWORD PTR [rbp+16+rax]
        add     rax, 32
        vmovaps YMMWORD PTR [rsp-32], ymm1
        vmovups YMMWORD PTR [rax-32+rdi], ymm0
        cmp     rax, 4096
        jne     .L4
```

- Auto-vectorisation doesn't always work
  - conservative transforms might not allow the parallelism to be exploited because they can't always be proved correct.
- Forcing the user to write sequential representations of parallel code is nasty.

Accessing simd hardware – compiler features

Explicit use of intrinsics of builtins allows parallelism to be directly expressed.

```
__m512i doSomething(__m512 value)
{
  constexpr __m512 k = _mm512_set1_ps(-0.0);
  const auto s0 = _mm512_andnot_epi16(k,
     _mm512_castps_si512(value));
  const auto s1 = _mm512_mul_ps(_mm512_set1_ps(2.0), s0);
  const auto t0 = _mm512_add_ps(_mm512_set1_ps(1.0), s1);

  const auto km = _mm512_cmp_mask_ps(t0,
     _mm512_set1_ps(threshold), _MM_CMP_GT);
  const auto t1 = _mm512_blend_ps(km, t0, t1);

  return _mm512_cvtps_epi32(t1, 0);
}
```

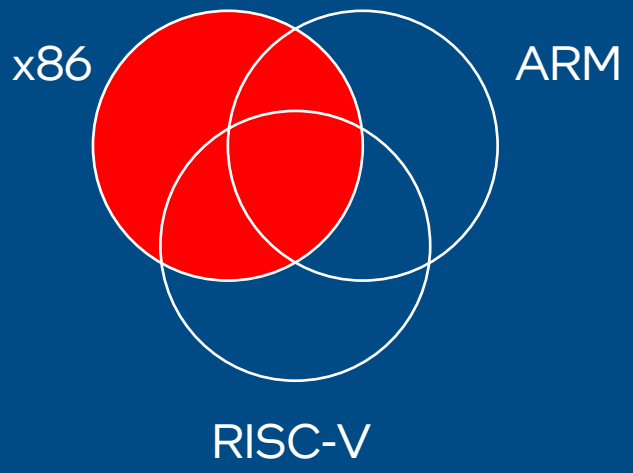
Intrinsics – verbose, non-portable, and obfuscated

```
v16i doSomething(v16f value) {
    // abs: clear sign bit
    v16u bits = (v16u)value & 0x7FFFFFFF;
    v16f s0 = (v16f)bits;

    // scale and offset
    v16f t0 = s0 * 2.0f + 1.0f;

    // blend: keep t0 where > threshold, else zero
    v16f t1 = t0 > value[0] ? t0 : (v16f){};

    // convert to int
    return __builtin_convertvector(t1, v16i);
}
```

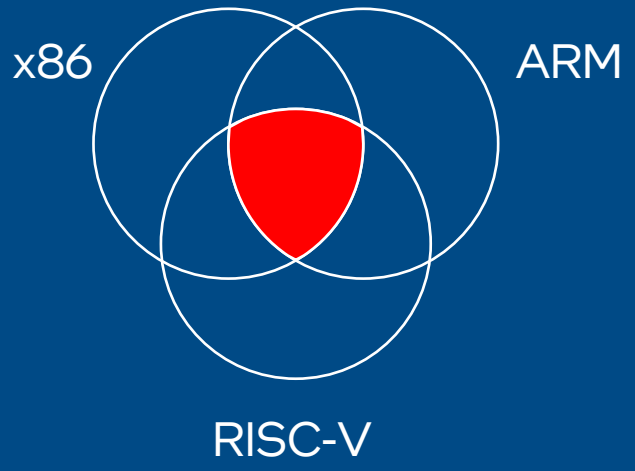
Compiler builtins – clearer and more portable (though not perfect)

Accessing hardware – abstraction libraries

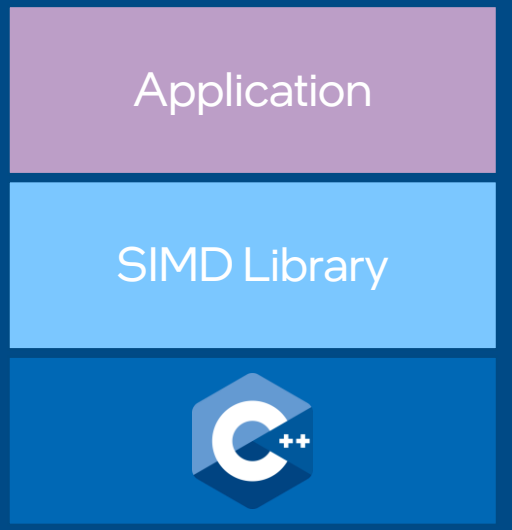
- Many excellent, and widely used libraries available (VCL, Highway, EVE, dozens more)



Exposes target specific features. Not portable.



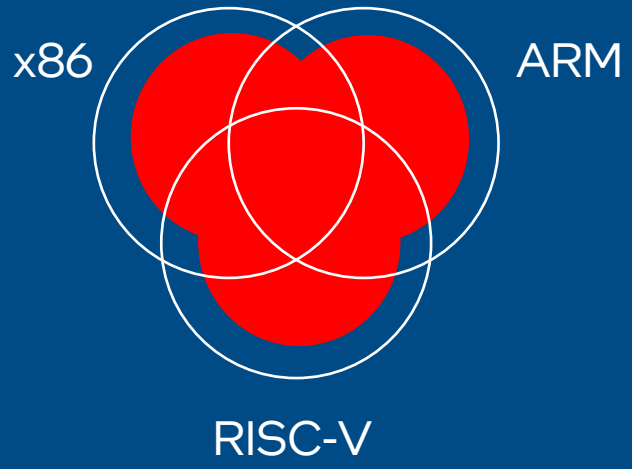
Portability through `intersection`. Only support features available everywhere.



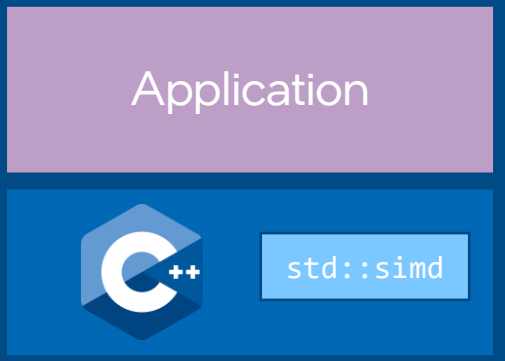
Libraries will never fully integrate with C++, giving missed opportunity.

Enter `std::simd`

Use LLVM to do as much of the hard work as possible.



Union of all targets to achieve portability.



Fully integrated with C++ (and its compilers)

Overview of `std::simd`

A quick tutorial on what C++26 brings...

Simplicity and readability – std::simd style

```
__m512i doSomething(__m512 value)
{
  constexpr __m512 k = _mm512_set1_ps(-0.0);
  const auto s0 = _mm512_andnot_epi16(k,
     _mm512_castps_si512(value));
  const auto s1 = _mm512_mul_ps(_mm512_set1_ps(2.0), s0);
  const auto t0 = _mm512_add_ps(_mm512_set1_ps(1.0), s1);

  const auto km = _mm512_cmp_mask_ps(t0,
     _mm512_set1_ps(threshold), _MM_CMP_GT);
  const auto t1 = _mm512_blend_ps(km, t0, t1);

  return _mm512_cvtps_epi32(t1, 0);
}
```

Native vec of floats, chosen at compile time

Operator overloading and scalar promotion

Overloads for standard libraries (in std)

Element indexing

New simd-specific functions to express common patterns

Conversion to a different element type

```
auto doSomething(simd::vec<float> value)
{
  auto t0 = abs(value) * 2 + 1;

  auto t1 = select(t0 >= value[0], t0);

  return simd::vec<int32_t>(t1);
}
```

One source, many targets

```
auto doSomething(vec<float> value)
{
  auto t0 = abs(value) * 2 + 1;

  auto t1 = select(t0 >= threshold, t0);

  return simd::vec<int32_t>(t1);
}
```

```
doSomething(simd<float, simd_abi::fixed
        xorps   xmm2, xmm2
        movaps  xmm3, xmmword ptr [rip
        movaps  xmm1, xmm0
        cmpltps xmm0, xmm2
        xorps   xmm3, xmm1
        blendvps        xmm1, xmm3, xmm
        movaps  xmm0, xmmword ptr [rip
        addps   xmm1, xmm1
        addps   xmm1, xmmword ptr [rip
        cmpleps xmm0, xmm1
        andps   xmm0, xmm1
        cvttps2dq       xmm0, xmm0
        ret
```

Goldmont Atom

```
doSomething(simd<float, simd_abi::fixed_size<8
        vxorps  xmm1, xmm1, xmm1
        vcmpltps        ymm1, ymm0, ymm1
        vbroadcastss    ymm2, dword ptr [rip
        vxorps  ymm2, ymm0, ymm2
        vblendvps       ymm0, ymm0, ymm2, ymm1
        vbroadcastss    ymm1, dword ptr [rip
        vbroadcastss    ymm2, dword ptr [rip
        vfmadd231ps     ymm2, ymm0, ymm1
        vbroadcastss    ymm0, dword ptr [rip
        vcmpleps        ymm0, ymm0, ymm2
        vandps  ymm0, ymm0, ymm2
        vcvttps2dq      ymm0, ymm0
        ret
```

Broadwell

```
doSomething(simd<float, simd_abi::fixed_size<1
        vxorps  xmm1, xmm1, xmm1
        vcmpltps        k1, zmm0, zmm1
        vxorps  zmm0 {k1}, zmm0, dword ptr [r
        vbroadcastss    zmm1, dword ptr [rip
        vfmadd213ps     zmm1, zmm0, dword ptr
        vcmpgeps        k1, zmm1, dword ptr [
        vmovaps zmm0 {k1} {z}, zmm1
        vcvttps2dq      zmm0, zmm0
        ret
```

Skylake

Recompiling generates code which uses different native register sizes. The width does not change at run-time.

Portability in type and size

```
template<class T, int N>
auto doSomething(simd::vec<T, N> value)
{
  auto t0 = abs(value) * 2 + 1;

  auto t1 = select(t0 >= value[0], t0);

  return simd::vec<int32_t, N>(t1);
}
```

<int, 4>

```
fn(xvec::simd::basic_vec<int, xvec::simd::simd_fixed_size_abi
        vpabsd  xmm1, xmm0
        vpaddd  xmm1, xmm1, xmm1
        vpord   xmm1, xmm1, dword ptr [rip + .LCPI0_0]{1to4}
        vpbroadcastd    xmm2, xmm0
        vpcmpgtd        k1, xmm2, xmm1
        vmovdqa32       xmm1 {k1}, xmm0
        vmovdqa xmm0, xmm1
        ret
```

<float, 16>

```
fn(xvec::simd::basic_vec<float, xvec::simd::simd_fixed_size_abi<16, 
        vandps  zmm1, zmm0, dword ptr [rip + .LCPI1_0]{1to16}
        vbroadcastss    zmm2, dword ptr [rip + .LCPI1_1] # zmm2 = [2
        vfmadd213ps     zmm2, zmm1, dword ptr [rip + .LCPI1_2]{1to16
        vbroadcastss    zmm1, xmm0
        vcmpleps        k1, zmm1, zmm2
        vmovaps zmm0 {k1}, zmm2
        vcvttps2dq      zmm0, zmm0
        ret
```

<short, 12>

```
fn(xvec::simd::basic_vec<short, xvec::simd::simd_fix
        vpabsw  ymm1, ymm0
        vpaddw  ymm1, ymm1, ymm1
        vpord   ymm1, ymm1, dword ptr [rip + .LCPI2_
        vpbroadcastw    ymm2, xmm0
        vpcmpnltw       k1, ymm1, ymm2
        vmovdqu16       ymm0 {k1}, ymm1
        vpmovsxwd       zmm0, ymm0
        ret
```

The declarative contract

- Imperative programming: how to achieve something

```
const auto as64 = _mm512_castps_epi64(values); // Treat as 64-bit units
const auto t = _mm512_cvtepi64_epi32(as64);    // Pull out the lower 32-bits
const auto evens = _mm256_castepi32_ps(t);     // Treat as float again.
```

- Declarative programming: what to achieve

```
auto evens = permute(values, [](auto idx) { return idx * 2; });
```

- `std::simd` is declarative – tell it where the parallelism is, how much is available, and what to do.

Construction and conversion

- Iota- or generator-based construction:

```
// LUTs can be defined using lambda generators
constexpr auto lut0 = simd::vec<int>([](auto i) { return i * 7 % 3; });


// ...or direct operations on iota (0, 1, 2, ...)
constexpr auto lut1 = simd::iota<simd::vec<int>> * 7 % 3;
```

- From spans:

```
std::span<float, 8> sf = …;
simd::basic_vec v = sf;
```

- Modern C++ brings new safety guarantees:

```
simd::vec<short> x = lut0;                          // Not value-preserving
simd::vec<short, 16> x = simd::vec<short, 16>(lut0);  // use explicit to force
```

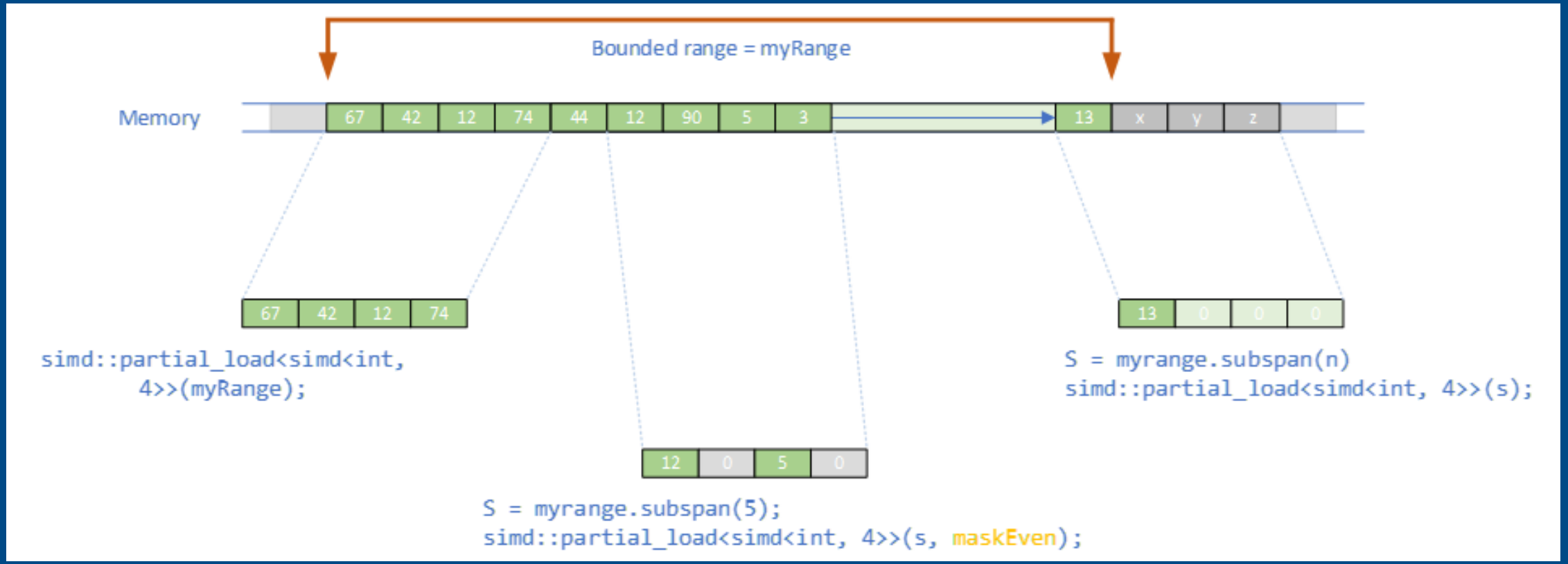
Masks

```
// Create a mask from relational operator
// classification, generator, etc.
const auto m0 = x > 34;
const auto m1 = isinf(x);
const auto m2 =
  simd::mask<float>([](auto idx) { return idx % 2 == 0; });
simd::mask<float> m3 = 0b10100101;

// Combine masks using logical operators
const auto m4 = m0 & m1 | m2;

// Use masks to operate on vectors
const auto conditionalAdd = select(isinf(x), x, x + 1.0f);

// Extract properties
auto numPositiveValues = reduce_count(x > 0);
```

Note there are two representations of masks – wide and dense. The library hides this.

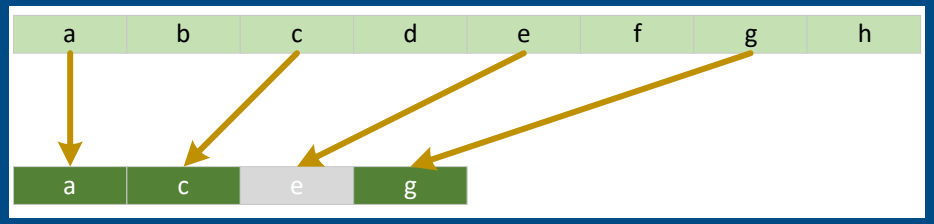
Memory safety with spans

- C++26 emphasizes safety - raw pointers/iterators discouraged



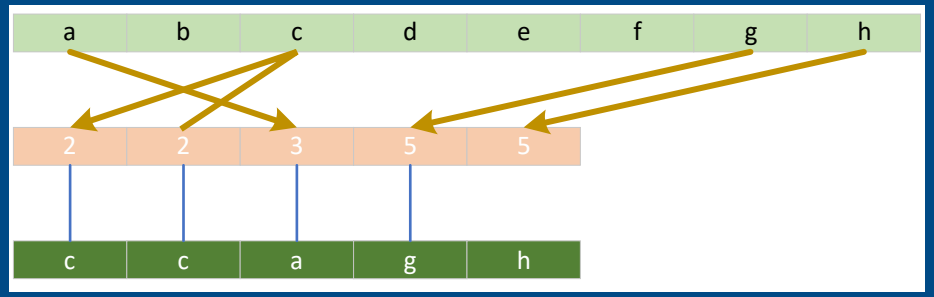
- (for performance, there is an unchecked version too)

Permutations

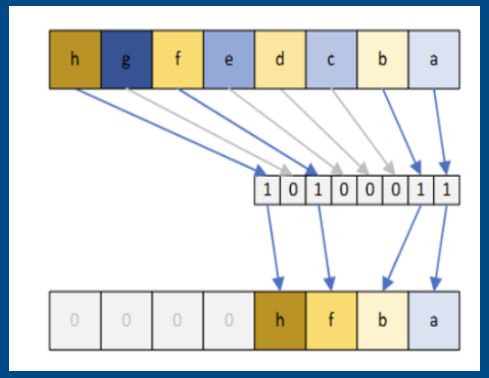



```
permute<4>([](auto i) { return i * 2; })
```

Static permute

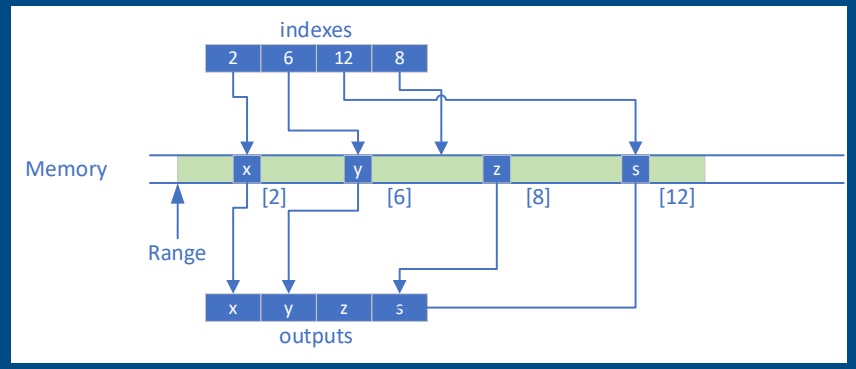


Dynamic permute (one vec/mask by another vec)




```
simd::vec<float> x;

simd::mask<float> m;

auto out = compress(x, m);
```

Compress/expand using masks



Memory gather/scatter

Other features

- Comprehensive standard library overloads:
  - <cmath> (sin, cos, exp, log, etc.)
  - <algorithm> (min, max, clamp, etc.)
  - <bit> (popcount, byteswap, rot, etc.)
  - <complex> (real, imag, conj, etc.)
- Reduction operations
  - Choose from min, max, sum, logical_or, etc.
  - Can extend to user-defined operations
  - With and without masking (i.e., only reduce active elements)

Implementing `std::simd`

You've seen what the library does. Here's how it's built.

Layered library

X86, ARM,
RISC-V, etc.

Optimisations for specific target features.

Complete, functional implementation of everything from std::simd, using Compiler and Data layers.

Data storage/layout, conversion, type safety, API and enforcement, mask wrappers

simd builtins and features

Target
(optional optimisations)

Operations

Data + API

Compiler

Code generation

Core functional library

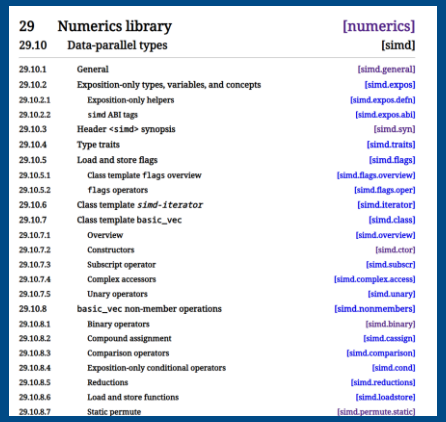
- The entire library works on top of the compiler, with any target which supports the basic simd features
- Target-specific optimisations are optionally applied at the top-layer, not littered throughout the codebase. Adding a new target only requires extending the top layer.

Layer 1 – LLVM target-agnostic simd features

```
// 8 floats = 256 bits
typedef float v8f __attribute__((vector_size(32)));

v8f process(v8f a, v8f b) {
    // operator overloading
    v8f sum = a + b;

    // Builtin simd functions
    v8f clamped = __builtin_elementwise_min(sum, (v8f){1,1,1,1,1,1,1,1});

    // builtin conversion - float to int
    auto converted = __builtin_convertvector(clamped, int __attribute__((vector_size(32))));

    // Builtin permutation
    return __builtin_shufflevector(converted, converted, 1,0, 3,2, 5,4, 7,6); // swap pairs
}
```

- We've already seen that this gives us access to the underlying simd capabilities of a target in a portable manner.
- It's great as a foundation on which to build std::simd itself, by abstracting the target's simd capabilities.

Layer 2 — the wrapper and element storage



29 Numerics library [numerics]
29.10 Data-parallel types [simd]

29.10.1	General	[simd.general]
29.10.2	Exposition-only types, variables, and concepts	[simd.expos]
29.10.2.1	Exposition-only helpers	[simd.expos.defn]
29.10.2.2	*simd* ABI tags	[simd.expos.abi]
29.10.3	Header <simd> synopsis	[simd.syn]
29.10.4	Type traits	[simd.traits]
29.10.5	Load and store flags	[simd.flags]
29.10.5.1	Class template flags overview	[simd.flags.overview]
29.10.5.2	flags operators	[simd.flags.oper]
29.10.6	Class template *simd-iterator*	[simd.iterator]
29.10.7	Class template basic_vec	[simd.class]
29.10.7.1	Overview	[simd.overview]
29.10.7.2	Constructors	[simd.ctor]
29.10.7.3	Subscript operator	[simd.subscr]
29.10.7.4	Complex accessors	[simd.complex.access]
29.10.7.5	Unary operators	[simd.unary]
29.10.8	basic_vec non-member operations	[simd.nonmembers]
29.10.8.1	Binary operators	[simd.binary]
29.10.8.2	Compound assignment	[simd.cassign]
29.10.8.3	Comparison operators	[simd.comparison]
29.10.8.4	Exposition-only conditional operators	[simd.cond]
29.10.8.5	Reductions	[simd.reductions]
29.10.8.6	Load and store functions	[simd.loadstore]
29.10.8.7	Static permute	[simd.permute.static]

C++ mechanics. Provide all the concepts, utilities and traits needed to make std::simd work.

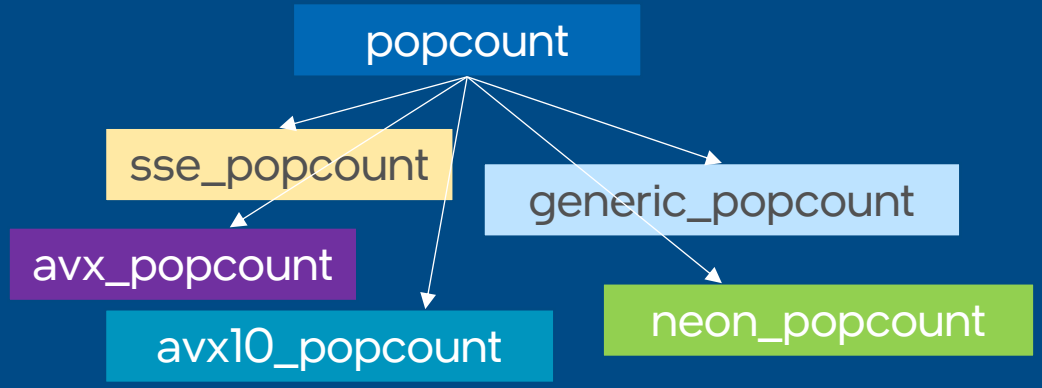
```
typedef element_type_t<T> v8f
  __attribute__((vector_size(32)));
```

Data storage. Provide appropriately sized element containers for the data, allowing extensibility to complex, and user-defined types.

```
101101001
```

```
11111111 00000000 11111111 11111111
```

Masking. Both verbose (SSE, AVX, etc) and dense (AVX-512, ARM) masks are available and vary by target. Abstract them away.



ABI. Create a layer which dispatches from a `std::simd` function to an internal implementation function.

Layer 2 - Dispatching to target implementations

- We want to differentiate and optimise for:
  - different targets (x86, ARM, etc.)
  - different instruction sets for a target (SSE, AVX, AVX512, AVX10)
- Traditionally handled as:

```
auto popcount(vec_type auto v) {
  if constexpr (hasSse)
    // Do SSE specific code
  else if constexpr (hasAvx || hasAvx2)
    // etc.
}
```

```
auto popcount(vec_type auto v) {
#if defined(__AVX512F__)
  // AVX-512 specific code
#else if defined(__AVX512FP16__)
  // AVX2 specific code
}
```

- But this leads to verbose code which is difficult to maintain and understand, with target-specific code scattered everywhere.

Target dispatch by overloading

- We realised that we could encode the main axes of code generation – target and operation – using tags:

```
struct generic_tag {};
struct x86_tag          : generic_tag {};
struct x86_sse_tag      : x86_tag {};
struct x86_avx_tag      : x86_sse_tag {};
struct x86_avx2_tag     : x86_avx_tag {};
struct x86_avx512_tag   : x86_avx2_tag {};
// ... x86_avxsnc_tag, x86_avx10_tag, etc..

inline constexpr auto vendor = vendor_tag{};
```

```
std::plus<>
std::minus<>
std::less<>
std::equals<>
etc.
```

- We then dispatch to a named implementation function:

```
basic_vec::operator*(auto lhs, auto rhs)
 { return do_operator(vendor, lhs, rhs, std::multiplies<>()); }
```

```
basic_vec::popcount(auto v) { return details::popcount(vendor, v); }
```

Overload resolution for operators

Generic implementations

```
template<vec_type V>
auto do_operator(generic_tag, V lhs, V rhs, auto OP)
{
  // Generic op using builtin operator overloads
  return OP(lhs, rhs);
}

template<vec_complex V>
auto do_operator(generic_tag, V lhs, V rhs, std::multiplies<>)
{
  return swap(lhs) * rhs + conj(lhs) * rhs;
}
```

Complex-isa implementation

```
template<vec_complex V>
auto
do_operator(x86_avx512_tag, V lhs, V rhs, std::multiplies<>)
{
  // Use AVX-512 specific instruction for complex
multiplication
  return _mm512_complex_mul(lhs, rhs);
}
```

Dispatcher

```
basic_vec::operator*(auto lhs, auto rhs)
 { return do_operator(vendor, lhs, rhs,
                      std::multiplies<>()); }
```

Outcome:

On SSE for multiply
Call compilers multiply

On AVX for complex multiply:
Call generic complex multiply

On recent Xeon:
Call complex-multiply intrinsic

Overload resolution for named functions

Overloaded implementations

```
template<vec_integral V>
auto popcount(generic_tag, V v) {
  // Use a generic algorithm for integral types
}

template<vec_integral V>
requires (sizeof(typename V::value_type) == 1)
auto popcount(x86_avx_tag, V v) {
  // Use custom version for bytes on AVX
}

template<vec_integral V>
auto popcount(x86_avxsnc_tag, V v) {
  return __mm512_popcount_epiXX(v);
}
```

Dispatcher

```
basic_vec::popcount(auto v) {
  return details::popcount(vendor, v);
}
```

Outcome:

On SSE:

Call generic synthesised popcount

On AVX for vec<uint8_t>:

Call special case AVX

On recent Xeon:

Call native popcount

Layer 4 – Target optimisations

- Layer 4 is completely optional
  - any machine with simd support in LLVM itself will just work…
  - …but maybe not efficiently
- Handles:
  - Special instructions
  - special cases of inputs (e.g., vec of bytes),
  - special sizes (e.g., in-lane permutes are faster)
  - Much more…
- Implemented as overloads of the base functions, which can work with parameters, constraints or more.
- The core library doesn't know or care how the target layer works

Rough edges – where LLVM could do better

- _BitInt(N):
  - Used to represent compact masks (e.g., ARM, AVX-512) of arbitrary size
  - Certain sizes bring code generation issues
- Lack of `constexpr`:
  - Now fixed but was a long-time issue for the vector built-ins, preventing much of the library from being **`constexpr`**-friendly..
  - Still present for functions like sin (https://godbolt.org/z/rc5rYT735)
- Dynamic permutes:
  - Scary and complicated. Many special cases to get right.
  - We have builtin_shufflevector (which is superb!)
  - Can we do the same for dynamic permutes?
- Gather/scatter builtins:
  - Similar to dynamic permutes – we can probably do better moving these into the compiler.

Closing

Status

- C++26:
  - Core features from Matthias Kretz from GSI.
  - Extended features from Daniel Towner and Ruslan Arutyunyan (Intel)
  - Everything described today is in the working draft.
- GCC 16
  - std::simd implementation from Matthias Kretz .
- C++29:
  - User-defined element types (`vec<saturating_int>`)
  - Named permutes (stride, transpose, zip, unzip)
  - `chunked_invoke` for interoperability with legacy or intrinsic-based code
  - Concepts for detecting common simd types to simplify generic programming.
  - More standard function overloads
  - Numerous small tweaks

Summary

- SIMD hardware gives us data-parallelism
- std::simd gives programmers declarative access without sacrificing portability
- The implementation:
  - Works with the compiler at low or even zero overhead
  - Provides functional implementation for any simd-enabled target that LLVM already supports
  - Allows new targets to be easily added and optimised.

Questions