# std::simd

## how to express inherent parallelism efficiently via data-parallel types

Dr. Matthias Kretz

**GSI** Helmholtz Center for Heavy Ion Research

CppCon '23

# Warning: content overload

- I can talk about SIMD for days.
- I will have to scratch the surface.
- My goal is to share my vision.
- How to think / design…

# Motivation

`std::simd` is for you!

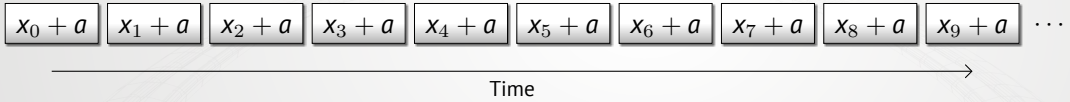
`std::simd` is for you!

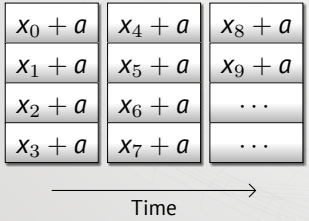
… unless you don't care for speed-up measured in factors not percentages

# SIMD — Single Instruction Multiple Data

in other words...

- multiple operations in one instruction, or
- execute the same work in less time

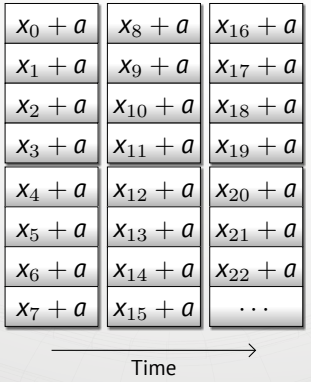


# ILP — Instruction Level Paralellism

in other words...

- multiple instructions in one CPU cycle, or
- execute the same work in even less time



`std::simd` expresses data-parallelism — SIMD and ILP

# Reaching Peak FLOP

Input: sequential scalar code
Output: no SIMD, no/little ILP

# single-precision multiply-add

```
1  void peak(benchmark::State& state) {
2    float x = 0;
3    fake_modify(x);
4    for (auto _ : state) {
5
6        x = x * 3.f + 1.f;
7
8    }
9    fake_read(x);
10 }
```

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports)

- `g++ -O3 -DNDEBUG`: 0.25 FLOP/cycle
- `g++ -O3 -DNDEBUG -march=native`: 0.5 FLOP/cycle

# single-precision SIMD multiply-add

```
1  void peak(benchmark::State& state) {
2    std::simd<float> x = 0;
3    fake_modify(x);
4    for (auto _ : state) {
5
6        x = x * 3.f + 1.f;
7
8    }
9    fake_read(x);
10 }
```

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports)

- `g++ -O3 -DNDEBUG`: 1 FLOP/cycle
- `g++ -O3 -DNDEBUG -march=native`: 8 FLOP/cycle

# single-precision SIMD multiply-add + ILP

```
1  void peak(benchmark::State& state) {
2    std::simd<float> x[8] = {};
3    fake_modify(x);
4    for (auto _ : state) {
5      for (auto& v : x) {
6        v = v * 3.f + 1.f;
7      }
8    }
9    fake_read(x);
10 }
```

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports)

- `g++ -O3 -DNDEBUG`: 8 FLOP/cycle
- `g++ -O3 -DNDEBUG -march=native`: 64 FLOP/cycle

# single-precision SIMD multiply-add + ILP

```
1  void peak(benchmark::State& state) {
2    std::simd<float> x[8] = {};
3    fake_modify(x);
4    for (auto _ : state) {
5      for (auto& v : x) {
6        v = v * 3.f + 1.f;
7      }
8    }
9    fake_read(x);
10 }
```

That's $\frac{64}{0.25} = 256$ times faster than the naive benchmark!

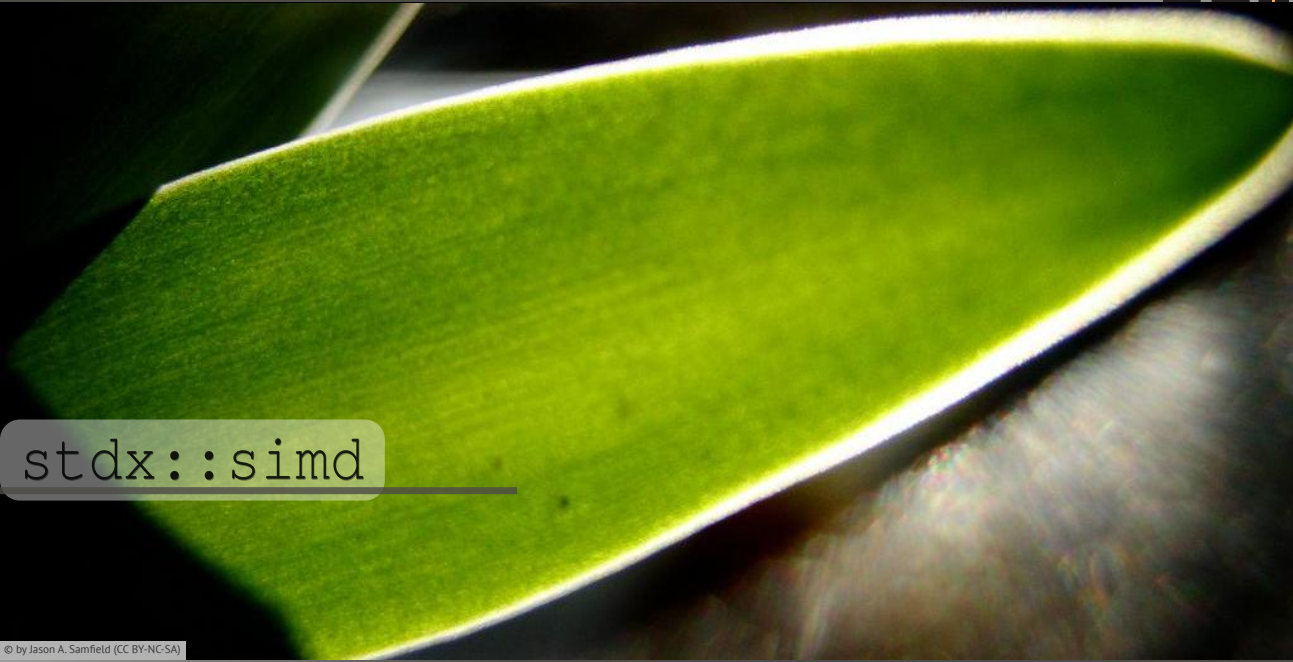
Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports)

- `g++ -O3 -DNDEBUG`: 8 FLOP/cycle
- `g++ -O3 -DNDEBUG -march=native`: 64 FLOP/cycle

# single-precision SIMD multiply-add + ILP (Outlook)

```
1  void peak(benchmark::State& state) {
2    std::simd<float, std::simd<float>::size() * 8> x = 0.f;
3    fake_modify(x);
4    for (auto _ : state) {
5
6        x = x * 3.f + 1.f;
7
8    }
9    fake_read(x);
10 }
```

- Using a data-parallel type that is wider than the default increases ILP!
- Not implemented (by me), so I won't give you benchmark results.
- I expect I can make the above equivalent to the previous slide.



# stdx::simd

# Disclaimer: this is not really a talk about `std::experimental::simd`

- If I write `simd<T>` I mean `std::simd<T>` as proposed in P1928, targeting C++26.
- If I show working code, I used the libstdc++ `std::experimental::simd` implementation of the Parallelism TS 2.

# Data-Parallel Types

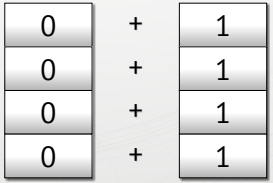
One variable stores $\mathcal{W}_{\mathtt{T}}$ values.
One operator signifies $\mathcal{W}_{\mathtt{T}}$ operations (element-wise).

$\mathcal{W}$ for "width"; depends on type `T`

```
int x = 0;
x += 1;
```


vs.

```
simd<int> x = 0;
x += 1;
```



## Example

One multiplication:

```
float f(float x) {
  return x * 2.f;
}
```

https://godbolt.org/z/1TY9jbqqj

Several multiplications in parallel:

```
simd<float> f(simd<float> x) {
  return x * 2.f;
}
```

## Example

One multiplication:

```
float f(float x) {
  return x * 2.f;
}
```

https://godbolt.org/z/1TY9jbqqj

```
; Intel, AVX-512:
f(float):
  vaddss xmm0, xmm0, xmm0
  ret

; aarch64:
f(float):
  fadd s0, s0, s0
  ret
```

Several multiplications in parallel:

```
simd<float> f(simd<float> x) {
  return x * 2.f;
}
```

```
; Intel, AVX-512:
f(std::simd<float, std::__detail::_VecBltnBtmsk<64> >):
  vaddps zmm0, zmm0, zmm0
  ret

; aarch64:
f(std::simd<float, std::__detail::_VecBuiltin<16> >):
  fadd v0.4s, v0.4s, v0.4s
  ret
```

# Pass by value or const-ref?

- You pass `int` and `float` by value not const-ref... 🤔
- ...so you pass `simd<int>` and `simd<float>` by value! (exceptions apply)

by reference

```
void f(simd<float>& result, const simd<float>& x) {
  result = x * 2.f;
}
```

```
vmovaps zmm0, ZMMWORD PTR [rsi]
vaddps zmm0, zmm0, zmm0
vmovaps ZMMWORD PTR [rdi], zmm0
vzeroupper
ret
```

by value

```
simd<float> f(simd<float> x) {
  return x * 2.f;
}
```

```
vaddps zmm0, zmm0, zmm0
ret
```

# Pass by value or const-ref?

- You pass `int` and `float` by value not const-ref... 🤔
- ...so you pass `simd<int>` and `simd<float>` by value! (exceptions apply)

## by reference 😱

```
1 void f(simd<float>& result, const simd<float>& x) {
2   result = x * 2.f;
3 }
```

```
1 vmovaps zmm0, ZMMWORD PTR [rsi]
2 vaddps zmm0, zmm0, zmm0
3 vmovaps ZMMWORD PTR [rdi], zmm0
4 vzeroupper
5 ret
```

## by value

```
simd<float> f(simd<float> x) {
  return x * 2.f;
}
```

```
1 vaddps zmm0, zmm0, zmm0
2 ret
```

# Pass by value or const-ref?

- You pass `int` and `float` by value not const-ref... 🤔
- ...so you pass `simd<int>` and `simd<float>` by value! (exceptions apply)

## by reference 😱

```
1 void f(simd<float>& result, const simd<float>& x) {
2   result = x * 2.f;
3 }
```

```
1 vmovaps zmm0, ZMMWORD PTR [rsi]
2 vaddps zmm0, zmm0, zmm0
3 vmovaps ZMMWORD PTR [rdi], zmm0
4 vzeroupper
5 ret
```

## by value 🏆

```
1 simd<float> f(simd<float> x) {
2   return x * 2.f;
3 }
```

```
1 vaddps zmm0, zmm0, zmm0
2 ret
```

# Data-Parallel Conditionals

## Example

One compare and 0 or 1 assignments:

```
float f(float x) {
  if (x > 0.f) { x *= 2.f; }
  return x;
}
```

$\mathcal{W}_{\texttt{float}}$ compares and $0$–$\mathcal{W}_{\texttt{float}}$ assignments in parallel:

```
simd<float> f(simd<float> x) {
  return simd_select(x > 0.f, x * 2.f, x);
}
```

`return x > 0.f ? x * 2.f : x;` — anyone?

The TS uses `where`-expressions instead.

- Compares yield $\mathcal{W}_{\texttt{T}}$ boolean answers
- Return type of compares: `std::simd_mask<T, N>`
- Reduction functions: `all_of`, `any_of`, `none_of`
- `simd` code typically uses no/few branches, relying on masked assignment instead

# Data-Parallel Conditionals

## Example

One compare and 0 or 1 assignments:

```
float f(float x) {
  return x > 0.f ? x * 2.f : x;
}
```

$\mathcal{W}_{\texttt{float}}$ compares and $0$–$\mathcal{W}_{\texttt{float}}$ assignments in parallel:

```
simd<float> f(simd<float> x) {
  return simd_select(x > 0.f, x * 2.f, x);
}
```

`return x > 0.f ? x * 2.f : x;` — anyone?

The TS uses `where`-expressions instead.

- Compares yield $\mathcal{W}_{\texttt{T}}$ boolean answers
- Return type of compares: `std::simd_mask<T, N>`
- Reduction functions: `all_of`, `any_of`, `none_of`
- `simd` code typically uses no/few branches, relying on masked assignment instead

# Data-Parallel Conditionals

## Example

One compare and 0 or 1 assignments:

```
float f(float x) {
  return x > 0.f ? x * 2.f : x;
}
```

$\mathcal{W}_{\texttt{float}}$ compares and $0–\mathcal{W}_{\texttt{float}}$ assignments in parallel:

```
simd<float> f(simd<float> x) {
  return simd_select(x > 0.f, x * 2.f, x);
}
```

`return x > 0.f ? x * 2.f : x;` — anyone?
The TS uses `where`-expressions instead.

- Compares yield $\mathcal{W}_{\texttt{T}}$ boolean answers
- Return type of compares: `std::simd_mask<T, N>`
- Reduction functions: `all_of`, `any_of`, `none_of`
- `simd` code typically uses no/few branches, relying on masked assignment instead

# Synopsis

```
template <typename T, int N>
  using simd = basic_simd<T, ...>;
```

```
template <typename T, int N>
  using simd_mask = basic_simd_mask<sizeof(T), ...>;
```

- `simd<T>` behaves just like `T` (as far as is possible)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `simd_mask<T>` behaves like `bool`
  In contrast to `bool`, there are many different mask types:
  - storage: bit-masks vs. element-sized masks (array of `bool` is not unheard of),
  - SIMD width `simd::size`
- `Abi` determines width and ABI (i.e. how parameters are passed to functions)
  - Example: `simd<int, 8>` on x86 can be two `xmm` registers (`alignof == 16`) or one `ymm` register (`alignof == 32`).

# Synopsis

```
template <typename T, int N>
  using simd = basic_simd<T, ...>;

template <typename T, int N>
  using simd_mask = basic_simd_mask<sizeof(T), ...>;
```

- `simd<T>` behaves just like `T` (as far as is possible)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `simd_mask<T>` behaves like `bool`
  In contrast to `bool`, there are many different mask types:
  - storage: bit-masks vs. element-sized masks (array of `bool` is not unheard of),
  - SIMD width `simd::size`
- `Abi` determines width and ABI (i.e. how parameters are passed to functions)
  - Example: `simd<int, 8>` on x86 can be two `xmm` registers (`alignof == 16`) or one `ymm` register (`alignof == 32`).

# Synopsis

```
template <typename T, typename Abi = ...>
  class basic_simd;

template <size_t Bytes, typename Abi>
  class basic_simd_mask;
```

```
template <typename T, int N>
  using simd = basic_simd<T, ...>;

template <typename T, int N>
  using simd_mask = basic_simd_mask<sizeof(T), ...>;
```

- `simd<T>` behaves just like `T` (as far as is possible)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `simd_mask<T>` behaves like `bool`
  In contrast to `bool`, there are many different mask types:
  - storage: bit-masks vs. element-sized masks (array of `bool` is not unheard of),
  - SIMD width `simd::size`
- `Abi` determines width and ABI (i.e. how parameters are passed to functions)
  - Example: `simd<int, 8>` on x86 can be two `xmm` registers (`alignof == 16`) or one `ymm` register (`alignof == 32`).

# ABI tag / width default

- Compiler flags determine the default ABI tag / SIMD width.
- `simd<T>` sets the ABI tag to the widest efficient $\mathcal{W}_{\mathrm{T}}$ for your `-march=` setting. It also influences the representation of `simd_mask` (i.e. `sizeof(mask)` may be very different).

  The TS uses the wrong default for the ABI tag. The TS gives you the lowest common denominator for all possible implementations of the target architecture. Use `native_simd<T>` with the TS!
- The ABI tag enables support for future ISA extensions without breaking existing code.

  The dreaded ABI break becomes an ABI addition…

**Consequence**

The `std::simd` and `simd_mask` ABI depends on `-m` flags!

# Constructors (simplified)

```
1 template <typename T, typename Abi = ...>
2 class basic_simd {
3   basic_simd() = default;
4   basic_simd(T);
5   basic_simd(std::contiguous_iterator auto, Flags = ...);
6   basic_simd(Generator);
7 }
```

- The defaulted *default* constructor allows uninitialized and zero-initialized objects.
- The *broadcast* constructor initializes all elements with the given value.

  requires a value-preserving conversion
- The *load* constructor reads $\mathcal{W}_{\mathrm{T}}$ elements starting from the given iterator.

  `Flags` can hint about alignment and opt in to non-value-preserving conversions
- The *generator* constructor initializes each element via the given generator function.

  The generator function is called with `std::integral_constant<std::size_t, i>`, where `i` is the index of the element to be initialized.

# Constructor examples

```
1  stdx::native_simd<int> x0; // uninitialized
2
3  stdx::native_simd<int> x1{}; // zero-initialized
4
5  stdx::native_simd<int> x2 = 1; // all elements are 1
6
7  stdx::native_simd<int> x3(addr, stdx::vector_aligned); // load from aligned address
8
9  stdx::native_simd<int> iota([](int i) { return i; }); // [0, 1, 2, 3, 4, ...]
```

# Loads & stores

`simd` needs to interact with the scalar world…

```
1 void f(std::vector<float>& data) {
2   using floatv = std::simd<float>;
3   for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4     floatv v(it);
5     v = std::sin(v);
6     v.copy_to(it);
7   }
8 }
```

- The member functions `copy_from` and `copy_to` allow "conversion" from/to contiguous ranges of `T`. `copy_from` is equivalent to the load constructor.
- The loads/stores are equivalent to dereferencing the iterator in the scalar case.

# Loads & stores

`simd` needs to interact with the scalar world…

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4      floatv v(it);
5      v = std::sin(v);
6      v.copy_to(it);
7    }
8  }
```

There's a bug, though…

- The member functions `copy_from` and `copy_to` allow "conversion" from/to contiguous ranges of `T`. `copy_from` is equivalent to the load constructor.
- The loads/stores are equivalent to dereferencing the iterator in the scalar case.

# Loads & stores

`simd` needs to interact with the scalar world…

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4      floatv v(it);
5      v = std::sin(v);
6      v.copy_to(it);
7    }
8  }
```

There's a bug, though…

- The member functions `copy_from` and `copy_to` allow "conversion" from/to contiguous ranges of `T`. `copy_from` is equivalent to the load constructor.
- The loads/stores are equivalent to dereferencing the iterator in the scalar case.

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4      floatv v(it);
5      v = std::sin(v);
6      v.copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4
5
6      std::sin(floatv(it)).copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it < data.end(); it += floatv::size()) {
4
5
6      std::sin(floatv(it)).copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it <= data.end() - floatv::size();
4         it += floatv::size()) {
5
6      std::sin(floatv(it)).copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    for (auto it = data.begin(); it <= data.end() - floatv::size();
4         it += floatv::size()) {
5
6      std::sin(floatv(it)).copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    auto it = data.begin();
4    for (; it <= data.end() - floatv::size();
5         it += floatv::size()) {
6      std::sin(floatv(it)).copy_to(it);
7    }
8
9
10
11 }
```

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    auto it = data.begin();
4    for (; it <= data.end() - floatv::size();
5         it += floatv::size()) {
6      std::sin(floatv(it)).copy_to(it);
7    }
8    for (; it < data.end(); ++it) {
9      *it = std::sin(*it);
10   }
11 }
```

- Having to write the epilogue every time is *error prone*.
- P1928 provides the low-level primitives, enabling *library-based high-level abstractions*. E.g.
  - SIMD iterator adaptor,
  - SIMD execution policy,
  - your ideas…

# Loads & stores (fixed)

SIMD code typically needs an *epilogue*:

```
1  void f(std::vector<float>& data) {
2    using floatv = std::simd<float>;
3    auto it = data.begin();
4    for (; it <= data.end() - floatv::size();
5         it += floatv::size()) {
6      std::sin(floatv(it)).copy_to(it);
7    }
8    for (; it < data.end(); ++it) {
9      *it = std::sin(*it);
10   }
11 }
```

- Having to write the epilogue every time is *error prone*.
- P1928 provides the low-level primitives, enabling *library-based high-level abstractions*. E.g.
  - SIMD iterator adaptor,
  - SIMD execution policy,
  - your ideas…

# P0350 Outlook — SIMD execution policy

```
1 void f(std::vector<float>& data) {
2   std::for_each(std::execution::simd, data.begin(), data.end(), [](auto& v) {
3     v = std::sin(v);
4   });
5 }
```

- Lambda called with `stdx::native_simd<float>`.
- Epilogue: called with `stdx::simd<float, Abi>` with different `Abi` so that the remainder of `data` is processed with minimal calls to the lambda.

**Take-Away #1**

For `simd` and scalar arrays to interact, you need an epilogue.

If your (internal) data structures use `simd` types, it's simpler, of course.

**Take-Away #2**

Design: Decouple `simd` access of contiguous ranges from the work applied to the data.

# Subscripting

Loads & stores are great, but sometimes you just want to access it like an array.

```
1  void f(std::simd<float> x) {
2    for (int i = 0; i < x.size(); ++i) {
3      x[i] *= 2.f;
4      x[i] = foo(x[i]);
5      auto ref = x[i];
6      ref = foo(x[i]); // ERROR: no assignment
7    }
8  }
```

- non-`const` subscripting returns a `basic_simd::reference`
- implements all non-const operators, i.e. (compound) assignment, increment and decrement, and also `swap`.

- all of the above functions are rvalue-ref qualified, i.e. are *only allowed on temporaries*
- For `std::vector<float> x` the type of `val` in `auto val = x` is `float`, not `float&`. I.e. we expect a *decay* of the reference (proxy) to the element type. Another paper I still have to write and defend in the committee. 😉

# Subscripting

Loads & stores are great, but sometimes you just want to access it like an array.

```
1  void f(std::simd<float> x) {
2    for (int i = 0; i < x.size(); ++i) {
3      x[i] *= 2.f;
4      x[i] = foo(x[i]);
5      auto ref = x[i];
6      ref = foo(x[i]); // ERROR: no assignment
7    }
8  }
```

- non-`const` subscripting returns a `basic_simd::reference`
- implements all non-const operators, i.e. (compound) assignment, increment and decrement, and also `swap`.

- all of the above functions are rvalue-ref qualified, i.e. are *only allowed on temporaries*
- For `std::vector<float> x` the type of `val` in `auto val = x` is `float`, not `float&`. I.e. we expect a *decay* of the reference (proxy) to the element type. Another paper I still have to write and defend in the committee. 😉

# Subscripting

Loads & stores are great, but sometimes you just want to access it like an array.

```
1  void f(std::simd<float> x) {
2    for (int i = 0; i < x.size(); ++i) {
3      x[i] *= 2.f;
4      x[i] = foo(x[i]);
5      auto ref = x[i];
6      ref = foo(x[i]); // ERROR: no assignment
7    }
8  }
```

- non-`const` subscripting returns a `basic_simd::reference`
- implements all non-const operators, i.e. (compound) assignment, increment and decrement, and also `swap`.

- all of the above functions are rvalue-ref qualified, i.e. are *only allowed on temporaries*
- For `std::vector<float> x` the type of `val` in `auto val = x` is `float`, not `float&`. I.e. we expect a *decay* of the reference (proxy) to the element type. Another paper I still have to write and defend in the committee. 😉

# Arithmetic & math

This is what you all came for, I guess

```
1 void f(std::simd<float> x, std::simd<float> y) {
2   x += y; // W_float additions
3   x = sqrt(x); // W_float square roots
4   ... // etc. all operators and <cmath>
5 }
```

- Operations act element-wise
- Speed-up is often a factor of $\mathcal{W}_{\mathrm{T}}$, but may be less, depending on hardware details.

# Same (element-wise) for compares

```
1 void f(std::simd<float> x, std::simd<float> y) {
2   if (x < y) {} // nonono, you don't write 'if (truefalsetruetrue)' either
3   x = simd_select(x < y, y, x); // x = y but only for the elements where x < y
4   if (all_of(x < y)) {} // this makes sense, yes
5 }
```

- Comparisons return a `simd_mask`.
- `simd_mask` is not convertible to `bool`.
- `simd_mask` can be *reduced* to `bool` via `all_of`, `any_of`, or `none_of`.

## Regularity

1. `T a = b; assert(a == b);`
2. `T a; a = b;` <=> `T a = b;`
3. `T a = c; T b = c; a = d; assert(b == c);`
4. `T a = c; T b = c; zap(a); assert(b == c && a != b);`

# Same (element-wise) for compares

```
1 void f(std::simd<float> x, std::simd<float> y) {
2   if (x < y) {} // nonono, you don't write 'if (truefalsetruetrue)' either
3   x = simd_select(x < y, y, x); // x = y but only for the elements where x < y
4   if (all_of(x < y)) {} // this makes sense, yes
5 }
```

- Comparisons return a `simd_mask`.
- `simd_mask` is not convertible to `bool`.
- `simd_mask` can be *reduced* to `bool` via `all_of`, `any_of`, or `none_of`.

## Regularity 😱 — the assertions are ill-formed

1. `T a = b; assert(a == b);`
2. `T a; a = b; <=> T a = b;`
3. `T a = c; T b = c; a = d; assert(b == c);`
4. `T a = c; T b = c; zap(a); assert(b == c && a != b);`

# Regularity

- A data-parallel type implements data-parallel semantics.
- So we get “data-parallel regularity” / “element-wise regularity”.
- Testing for regularity yields multiple answers, each saying that regularity held.
- To reduce multiple answers into one answer we need a reduction: `all_of`.

## Element-wise Regularity 🏆

1. `simd<T> a = b; assert(all_of(a == b));`
2. `simd<T> a; a = b; <=> simd<T> a = b;`
3. `simd<T> a = c; simd<T> b = c; a = d; assert(all_of(b == c));`
4. `simd<T> a = c; simd<T> b = c; zap(a); assert(all_of(b==c && a!=b));`

# Reductions

- We've already seen mask reductions: `all_of`, `any_of`, and `none_of`
- More mask reductions: `reduce_count`, `reduce_min_index`, and `reduce_max_index`
- For `simd` there's `reduce`, `reduce_min`, and `reduce_max`

**Example**

```
1 int f(std::simd<float> x) {
2   float sum = std::reduce(x);
3   float product = std::reduce(x, std::multiplies());
4   float sum_of_positive_x = std::reduce(x, x > 0);
5   float minimum = std::reduce_min(x);
6   return std::reduce_min_index(x == minimum); // (lowest) index of minimum
7 }
```

# Permutations

- There is no permutation API in the TS and P1928.
- The generator constructor can often be used instead.
- Intel is adding permutations to `simd` with P2664 (design still in review, may change).
- P2664 also explores how to add gather & scatter primitives.

**Example**

```
1 std::simd<float> permute_even_odd(std::simd<float> x) {
2   return std::simd_permute(x, [](auto idx) { return idx ^ 1; });
3 }
```

with AVX, compiles to:

```
vpermilps ymm0, ymm0, 177
```

# Programming Models

# Abstract

- Conceptually: SIMD types express data-parallelism.
- Wrong mindset: SIMD types are specific SIMD registers.

Which is why I like to call them “data-parallel types”.

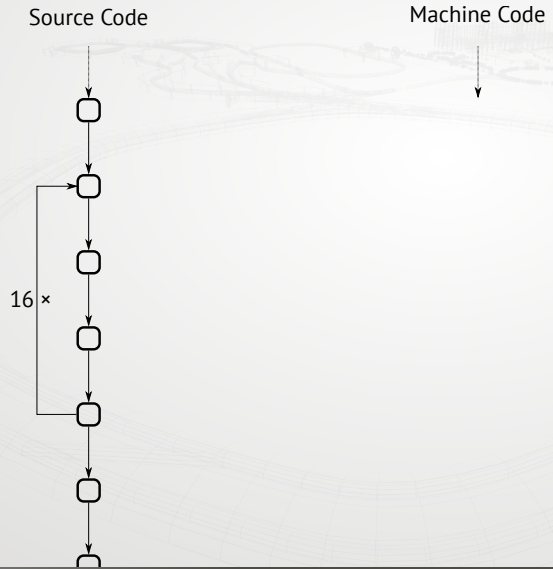
# Programming Models

**a small (abstract) example**

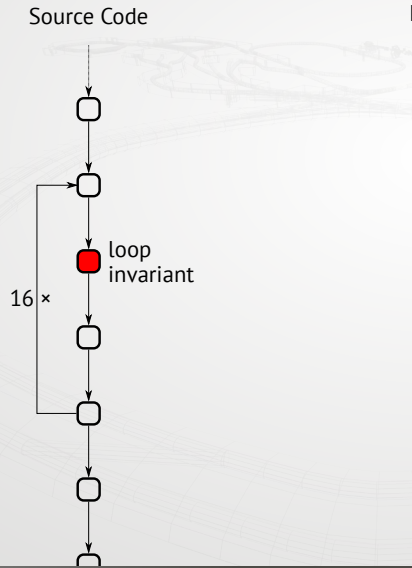
consider a function where 16 different values have to be processed in the same way

- the developer notices each of the values can be processed *independently* of the other 15
- in other words: the problem is inherently parallel

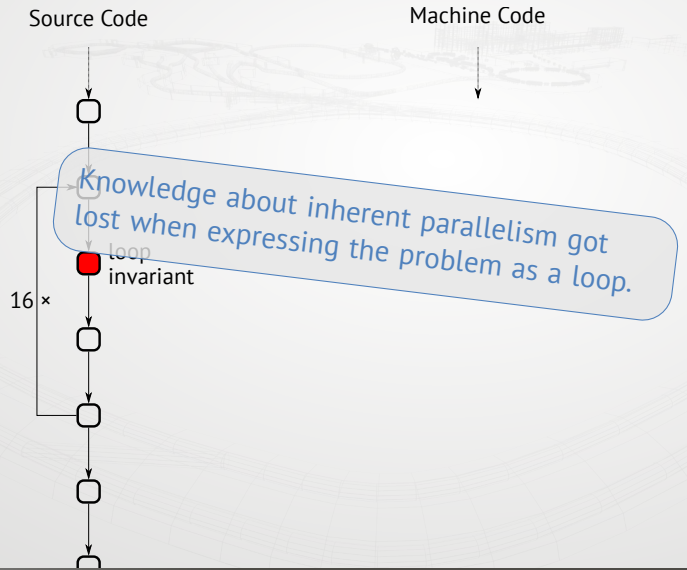
# Solution 1: Standard C++



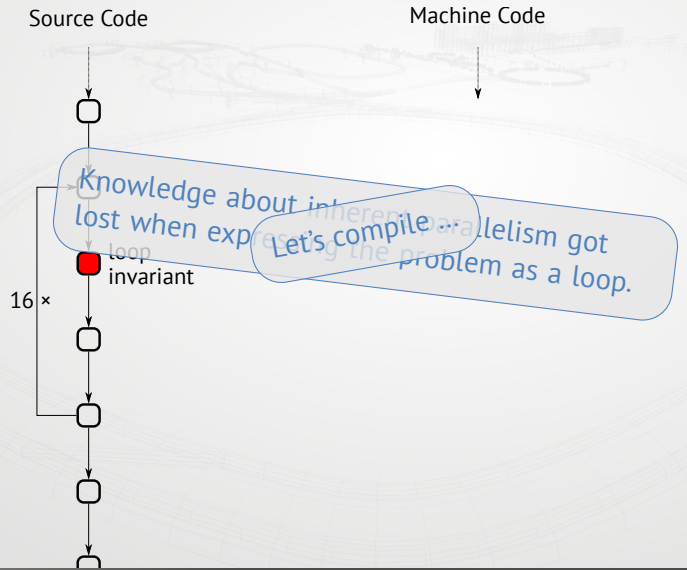
# Solution 1: Standard C++



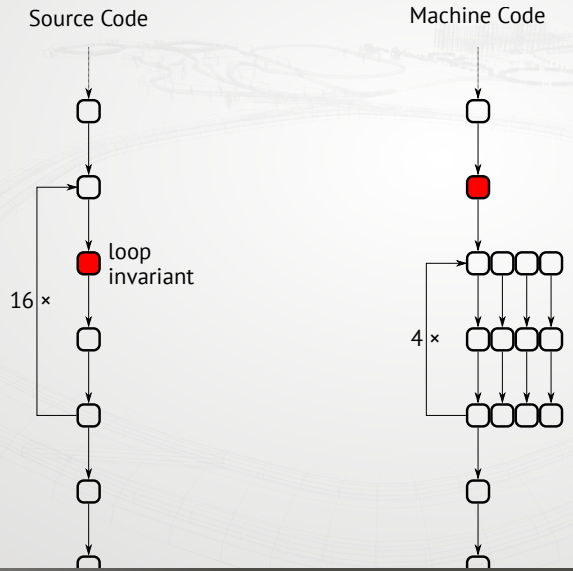
# Solution 1: Standard C++



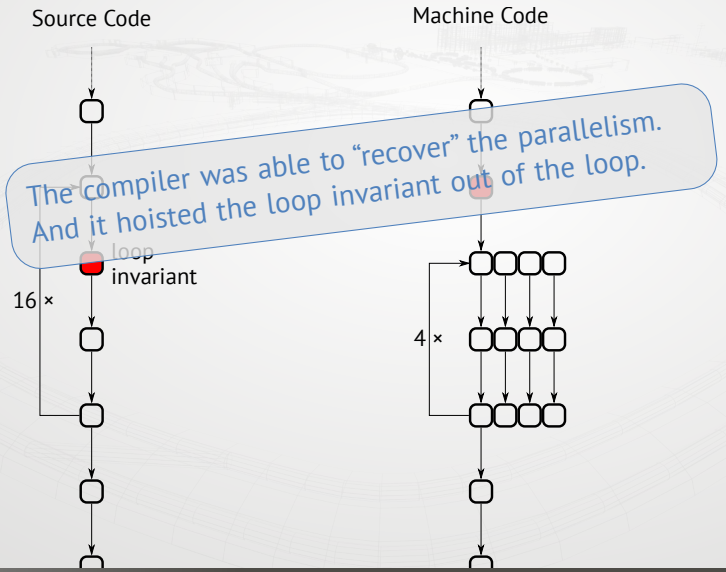
# Solution 1: Standard C++



# Solution 1: Standard C++



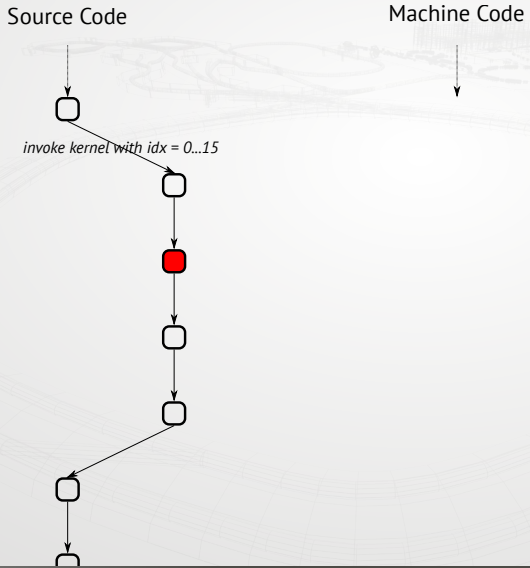
# Solution 1: Standard C++



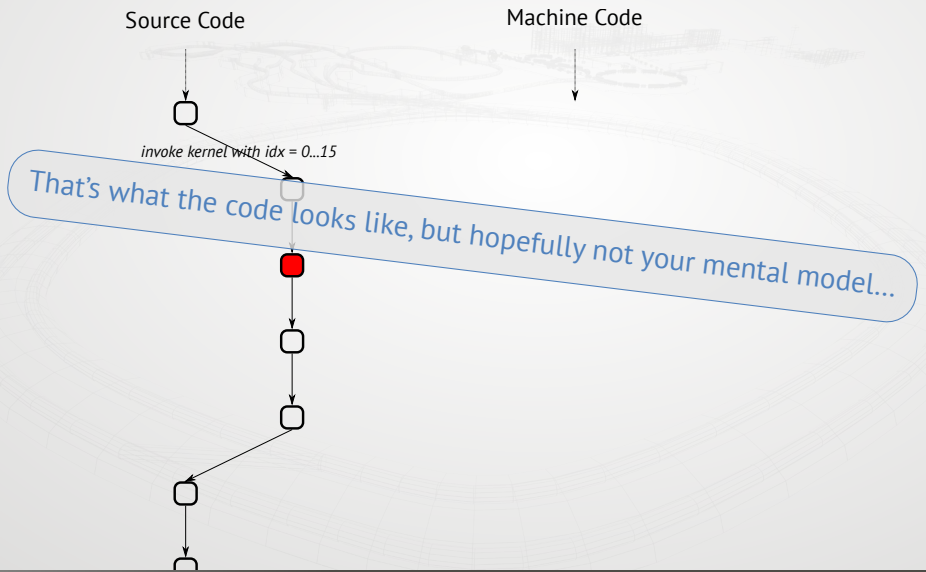
# Solution 2: SIMT (simplified)

Single Instruction, Multiple Threads

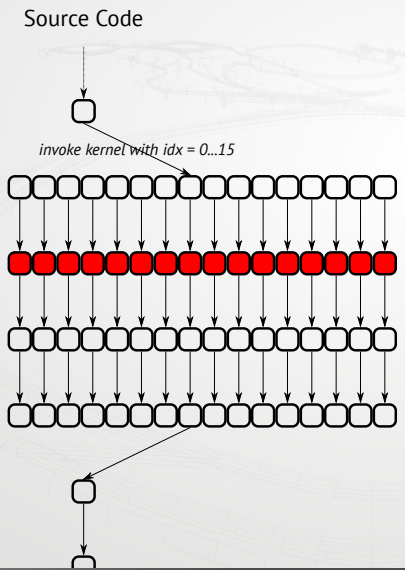
# Solution 2: SIMT (simplified)



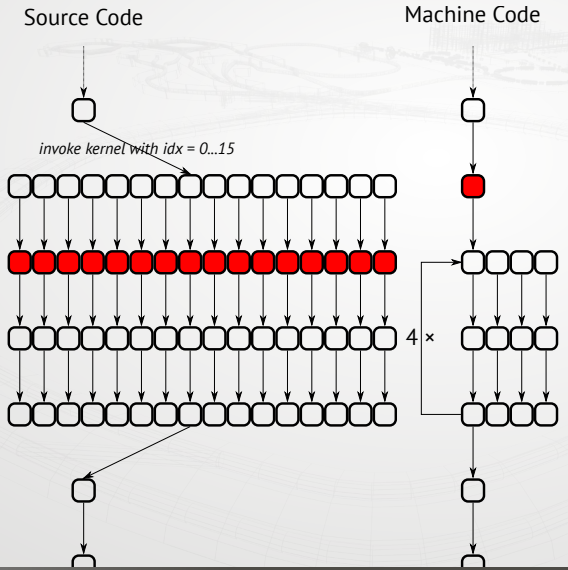
# Solution 2: SIMT (simplified)



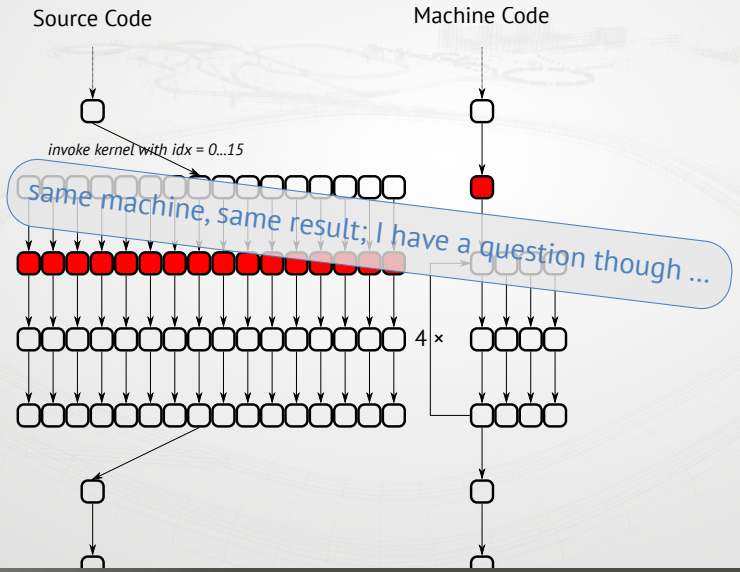
# Solution 2: SIMT (simplified)



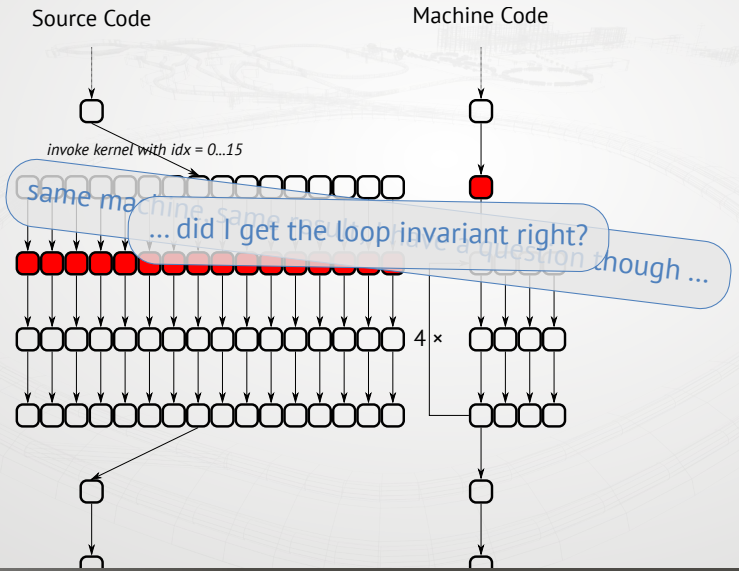
# Solution 2: SIMT (simplified)



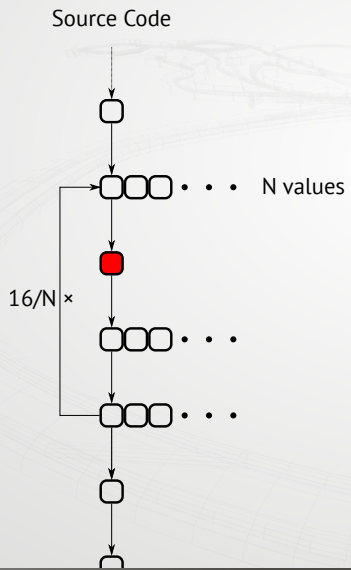
# Solution 2: SIMT (simplified)



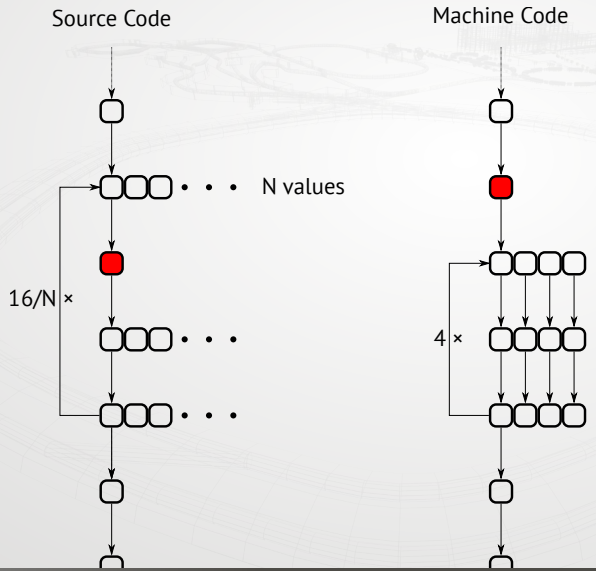
# Solution 2: SIMT (simplified)



# Solution 3: Data-Parallel Types



# Solution 3: Data-Parallel Types



# Solution 1': Standard C++ execution policies

**loop with different semantics**

```
std::for_each(std::execution::unseq, /*...*/)
```

- vector semantics: similar to SIMT (Solution 2: simplified SIMT)
  - Differs from *sequenced before* semantics you are used to.
  - "The behavior of a program is undefined if it invokes a vectorization-unsafe standard library function [...]."
  - "A standard library function is vectorization-unsafe if it is specified to synchronize with another function invocation, or another function invocation is specified to synchronize with it, [...]."
  - Uncaught exceptions invoke `std::terminate`.
- `unseq` does not come with the parallelization overhead of `par` and `par_unseq`.
- However, that's the *theory* — check whether it actually helps in practice (example follows).

# In Code

## trivial data-parallelism

given `data` of type `std::vector<int>`

```
1 std::for_each(std::execution::unseq,
2   data.begin(), data.end(),
3   [](int& x) {
4     x += 1;
5   });
```

```
1 std::for_each(vir::execution::simd,
2   data.begin(), data.end(),
3   [](auto& x) {
4     x += 1;
5   });
```

I'm using `vir::execution::simd` instead of `std::execution::simd` because the former is available as a library, the latter is only a proposal.

# In Code

## trivial data-parallelism

given `data` of type `std::vector<int>`

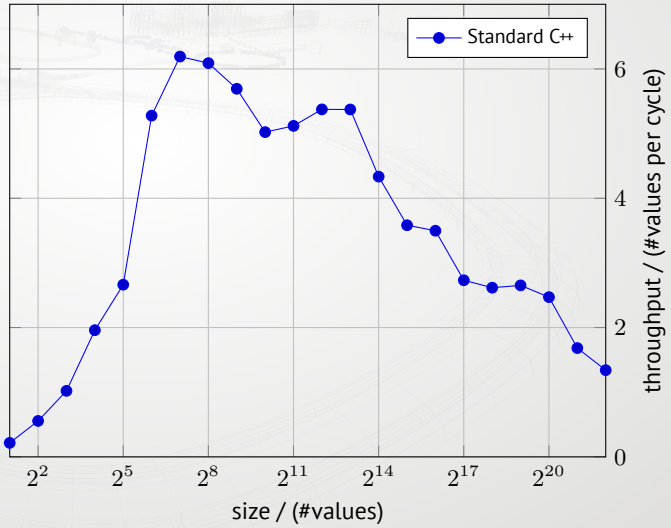
```
1 std::for_each(std::execution::unseq,
2   data.begin(), data.end(),
3   [](int& x) {
4     x += 1;
5   });
```

```
1 std::for_each(vir::execution::simd,
2   data.begin(), data.end(),
3   [](auto& x) {
4     x += 1;
5   });
```

I'm using `vir::execution::simd` instead of `std::execution::simd` because the former is available as a library, the latter is only a proposal.

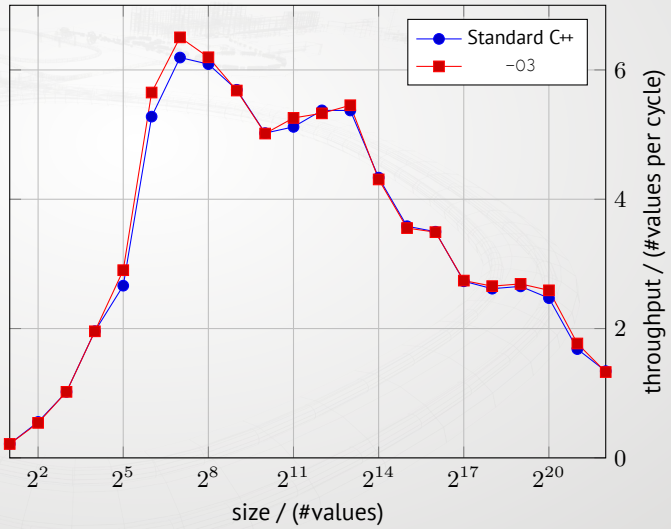
# Results (GCC 13, Intel Skylake i7-8550U)

```
1 for (int& x : data) {
2   x += 1;
3 }
```



# Results (GCC 13, Intel Skylake i7-8550U)

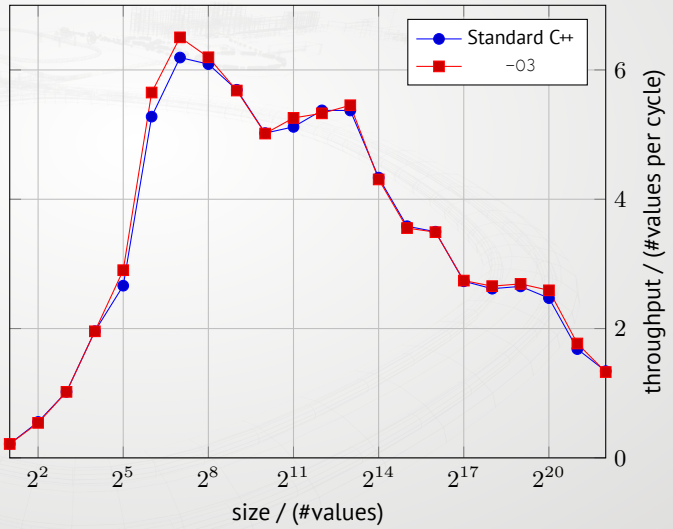
```
1 for (int& x : data) {
2   x += 1;
3 }
```



# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

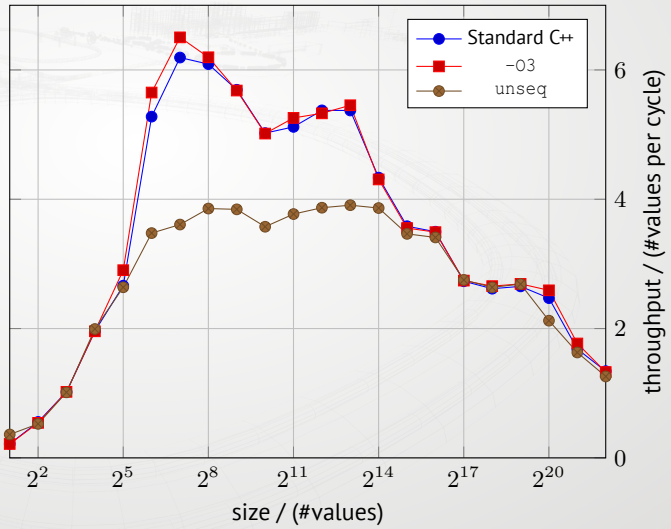
```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```



# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

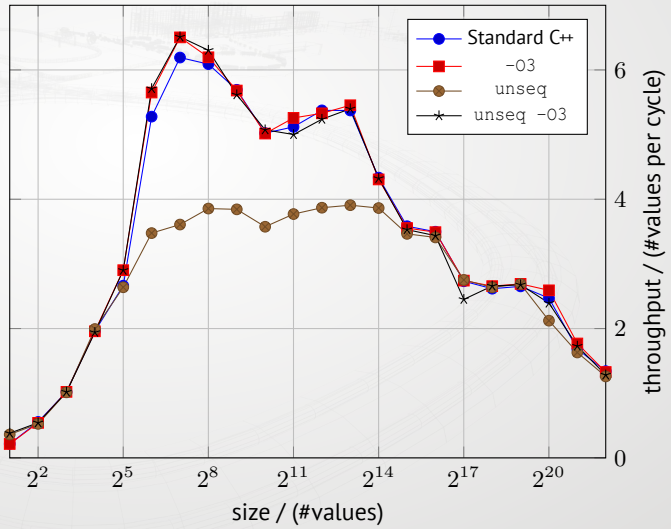
```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```



# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```

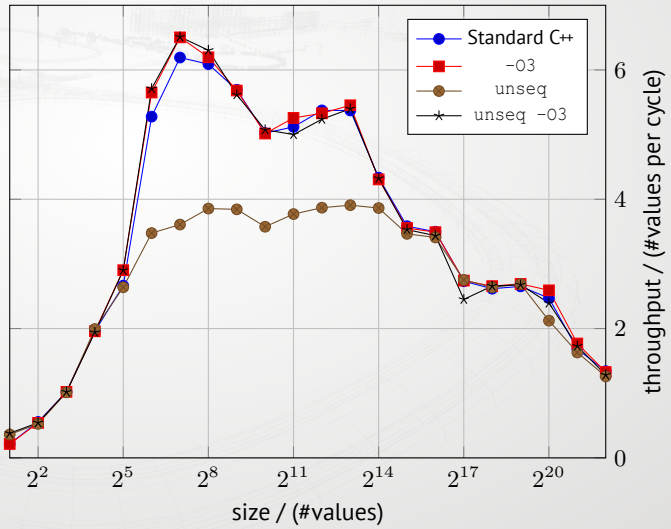


# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```

```cpp
std::for_each(vir::execution::simd,
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```

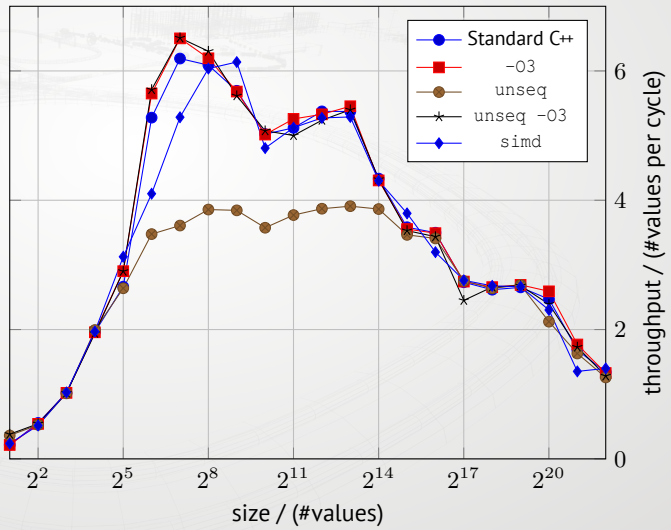


# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```

```cpp
std::for_each(vir::execution::simd,
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```



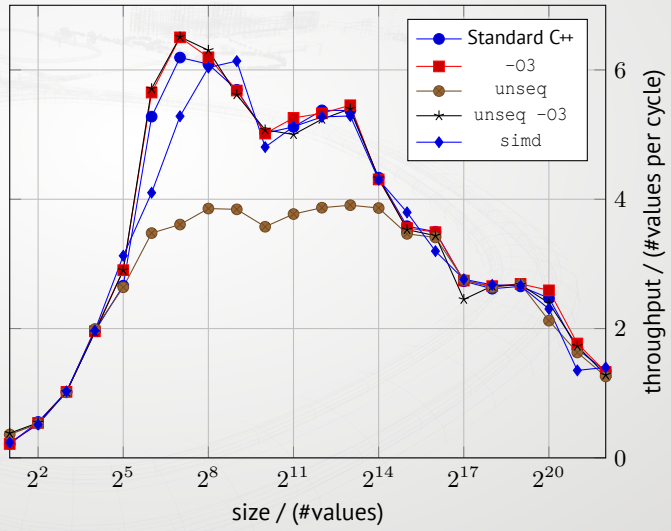
# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```

```cpp
std::for_each(vir::execution::simd,
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```

```cpp
std::for_each(
  vir::execution::simd.unroll_by<4>(),
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```



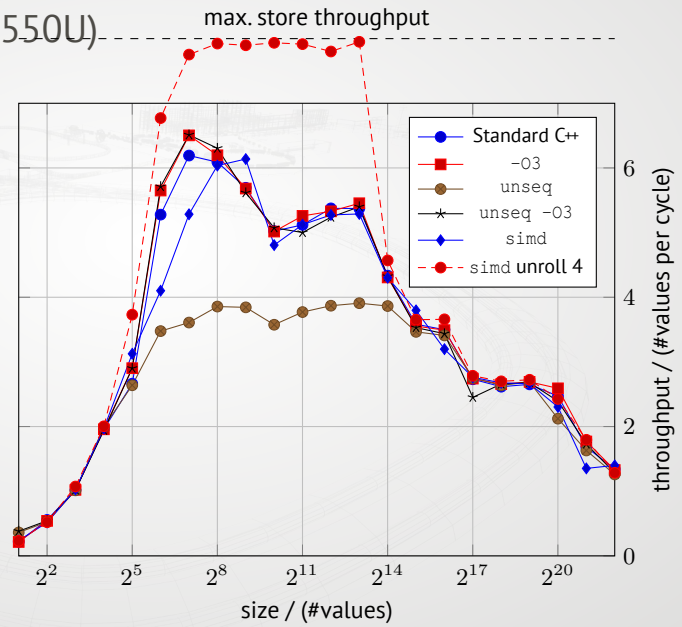
# Results (GCC 13, Intel Skylake i7-8550U)

```cpp
for (int& x : data) {
  x += 1;
}
```

```cpp
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) { x += 1; }
);
```

```cpp
std::for_each(vir::execution::simd,
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```

```cpp
std::for_each(
  vir::execution::simd.unroll_by<4>(),
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```



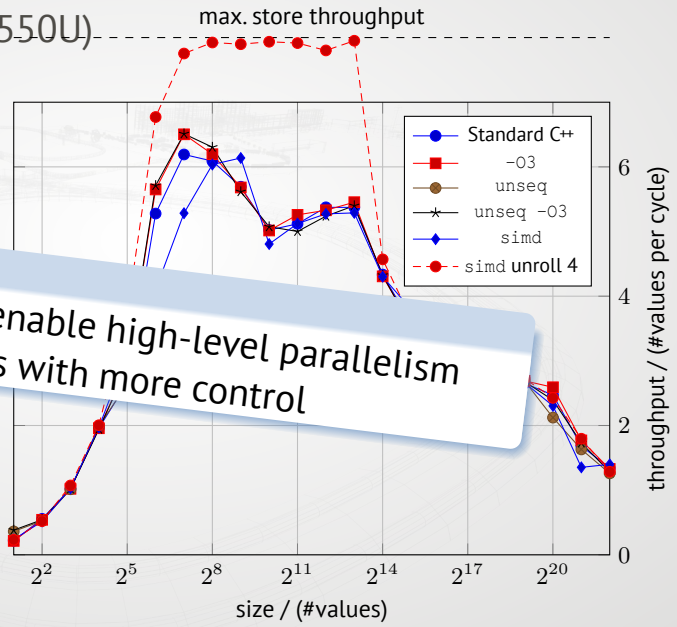
# Results (GCC 13, Intel Skylake i7-8550U)

```
for (int& x : data) {
  x += 1;
}
```

```
std::for_
  data.be
  [](int
);
```

```
std::for_each(vir::
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```

```
std::for_each(
  vir::execution::simd.unroll_by<4>(),
  data.begin(), data.end(),
  [](auto& x) { x += 1; }
);
```




Take-Away #3

Data-parallel types enable high-level parallelism abstractions with more control

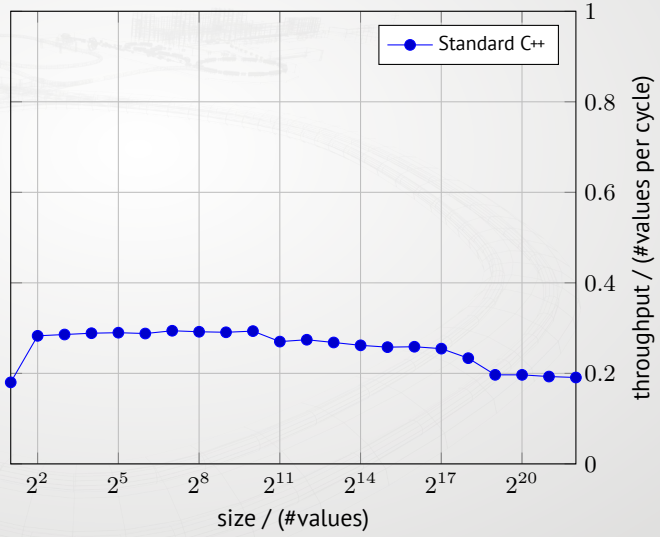
# One more teaser

SIMD … when all you have is AoS (Array of Struct)

```
1  template <typename T> struct Vec3d
2  {
3    T x, y, z;
4
5    friend constexpr T operator*(Vec3d a, Vec3d b)
6    { return a.x * b.x + a.y * b.y + a.z * b.z; }
7  };
8
9  float inner_product(std::vector<Vec3d<float>> const& a,
10                     std::vector<Vec3d<float>> const& b) {
11   return std::transform_reduce(vir::execution::simd, a.begin(), a.end(),
12                                b.begin(), 0.f);
13 }
```

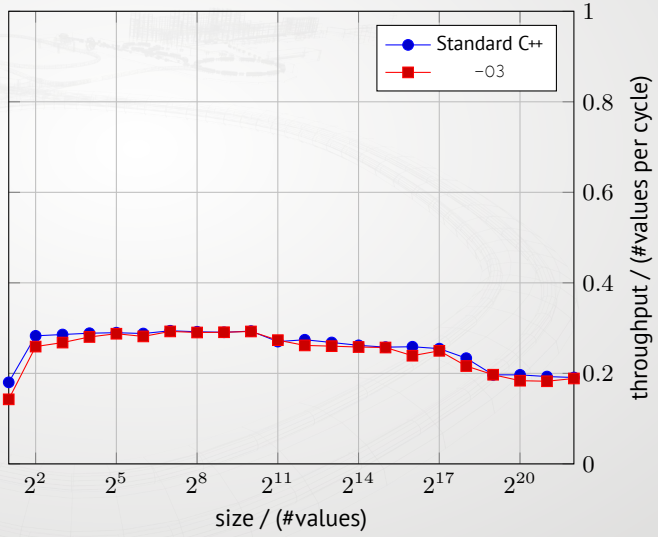
# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy



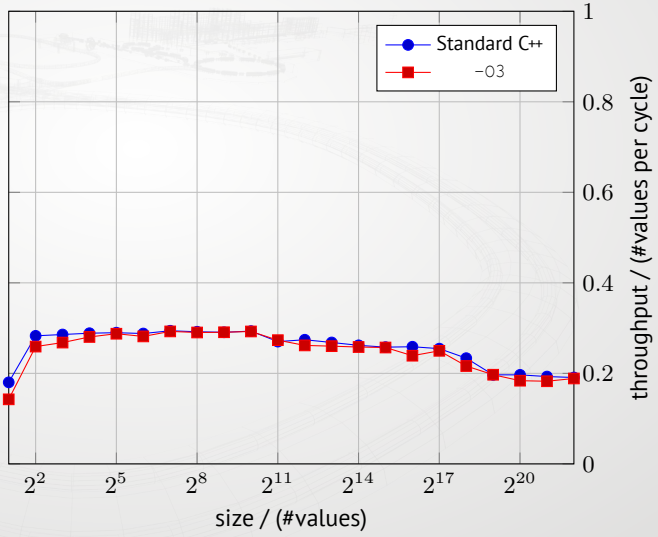
# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize



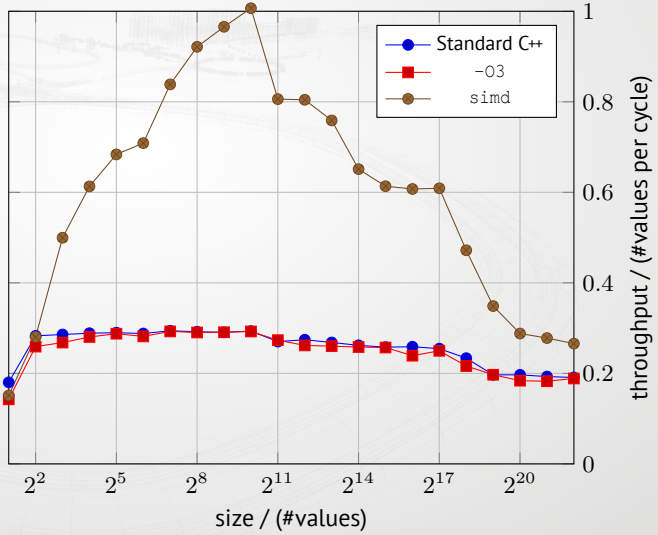
# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- `std::execution::unseq` doesn't compile (would fall back to `seq` anyway)



# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- `std::execution::unseq` doesn't compile (would fall back to `seq` anyway)
- another talk/paper…

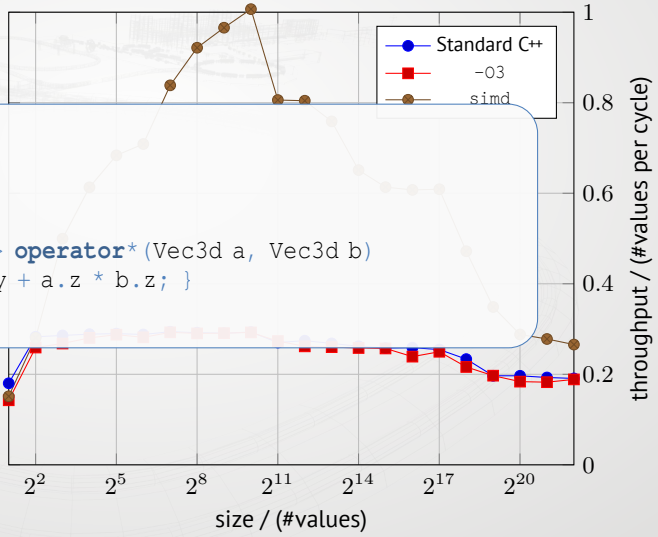


# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- std::experimental::simd doesn't compile (would fall back to seq anyway)
- another talk/paper

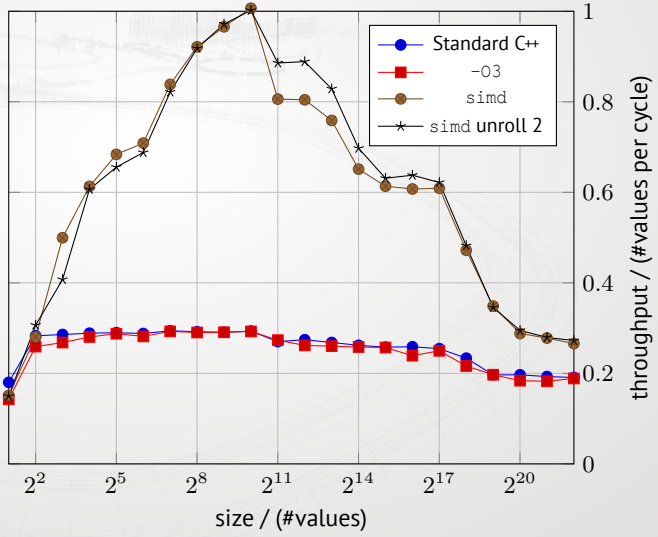
```
struct Vec3d<simd<float>>
{
  simd<float> x, y, z;

  friend constexpr simd<float> operator*(Vec3d a, Vec3d b)
  { return a.x * b.x + a.y * b.y + a.z * b.z; }
};
```



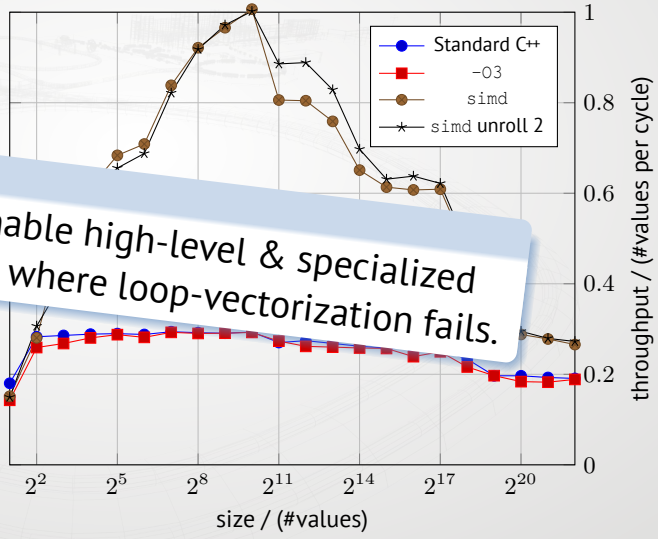
# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- `std::execution::unseq` doesn't compile (would fall back to `seq` anyway)
- another talk/paper…
- unrolling doesn't help: transforming AoS into three SIMD registers is the bottleneck



# Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- `std:` `tion::unseq` doesn't com
  any
- and
- unrolling doesn't help.
  AoS into three SIMD registers is the bottleneck




**Take-Away #4**

Data-parallel types enable high-level & specialized parallelism abstractions where loop-vectorization fails.

The semantics of data-parallel types
teach developers to design scalable and portable parallelization.

Target-dependent $\mathcal{W}_T$
Conversions between scalar and vector objects
Conditional assignment instead of branching

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy … to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience

The semantics of data-parallel types
teach developers to design scalable and portable parallelization.

Target-dependent $\mathcal{W}_T$
Conversions between scalar and vector objects
Conditional assignment instead of branching

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy ... to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience

The semantics of data-parallel types
teach developers to design scalable and portable parallelization.

Target-dependent $\mathcal{W}_T$
Conversions between scalar and vector objects
Conditional assignment instead of branching

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy ... to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience

The semantics of data-parallel types
teach developers to design scalable and portable parallelization.

Target-dependent $\mathcal{W}_T$
Conversions between scalar and vector objects
Conditional assignment instead of branching

- It becomes clear that data structures are the main challenge
  - Translating an inherently data-parallel algorithm to data-parallel types is often trivial
  - However, where do `simd` objects come from, and where can you put them?
  - With vector loops and SIMT it is easy … to write inefficient memory access patterns.
- Using SIMD types makes the design challenges wrt. efficient vectorization obvious
- Subsequent designs can profit from this experience

- In contrast to SIMT and vector loops, the chunk size is given as a constant expression.
- Working on a larger index space than $\mathcal{W}_{\mathrm{T}}$ requires a loop and/or multiple threads.
- Clear separation of serial, SIMD-parallel, and thread parallel execution.
- No restriction on I/O, exceptions, function calls, and synchronization.
- API & ABI for vectorization across multiple translation units (and library boundaries).

# SIMD and memory

# Outlook

# vectorizable types

- TS ( C++17 )
  - arithmetic types other than `bool`
- P1928 ( C++26, in library wording )
  - arithmetic types other than `bool` and `long double`
- P2663 ( C++26, in library evolution, thank you Intel )
  - adds `std::complex` to vectorizable types (`simd<complex>` *not* `complex<simd>`)
- TODO ( Intel is preparing papers in this area )
  - add scoped and unscoped enums to vectorizable types
  - add user-defined numeric types to vectorizable types (fixed-point, saturating integers, etc.)

# More interoperability

- Conversion to/from `std::array`, `std::span`, contiguous ranges?
- Mask conversion to/from `std::bitset`.
- Initializer list constructor for `simd`.
- Make `simd` and `simd_mask` ranges.
  - Formatting falls out.
  - Makes it easy to flatten a `vector<simd<T>` into a range of `T`.

# automatic type vectorization

```
1  struct Point {
2    float x, y, z;
3  };
4  using PointV = simdize<Point>;
5
6  // equivalent to
7  struct PointV {
8    simd<float> x, y, z;
9
10 };
```