SIMD, stdx::simd, vir-simd, and merged GR4 blocks

maximize throughput, minimize latency, express parallelism via data-parallel types

Dr. Matthias Kretz

GSI Helmholtz Center for Heavy Ion Research

European GNU Radio Days ’24

@mkretz@floss.social

github.com/mattkretz

Goals and non-goals for this talk

- Introduce SIMD
- Introduce data-parallel types: `std::experimental::simd`[1]
- Introduce extensions: vir-simd
- Walk through a simple signal-processing example
- This talk wants to be a starting point for you to use `stdx::simd` in a GR4 block

[1] From now on shortened to: `stdx::simd` — via `namespace stdx = std::experimental;`

Motivation

SIMD – Single Instruction Multiple Data

in other words...

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

$x_0 + a$ | $x_1 + a$ | $x_2 + a$ | $x_3 + a$ | $x_4 + a$ | $x_5 + a$ | $x_6 + a$ | $x_7 + a$ | $x_8 + a$ | $x_9 + a$ | $\cdots$

$\xrightarrow{\hspace{6cm}}$ Time

$x_0 + a$	$x_4 + a$	$x_8 + a$
$x_1 + a$	$x_5 + a$	$x_9 + a$
$x_2 + a$	$x_6 + a$	$\cdots$
$x_3 + a$	$x_7 + a$	$\cdots$

$\xrightarrow{\hspace{2cm}}$ Time

ILP – Instruction Level Paralellism

in other words...

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

$x_0 + a$	$x_8 + a$	$x_{16} + a$
$x_1 + a$	$x_9 + a$	$x_{17} + a$
$x_2 + a$	$x_{10} + a$	$x_{18} + a$
$x_3 + a$	$x_{11} + a$	$x_{19} + a$
$x_4 + a$	$x_{12} + a$	$x_{20} + a$
$x_5 + a$	$x_{13} + a$	$x_{21} + a$
$x_6 + a$	$x_{14} + a$	$x_{22} + a$
$x_7 + a$	$x_{15} + a$	$\cdots$

$\longrightarrow$

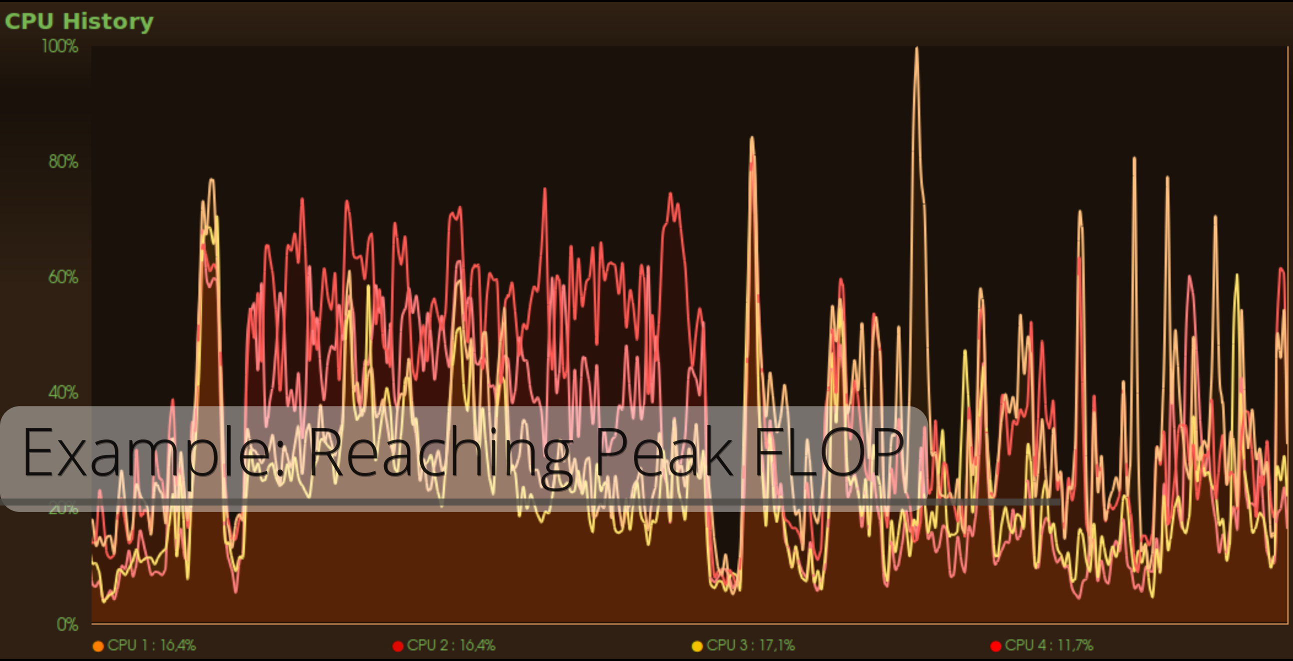
Time

Take-Away #1

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

again, `stdx` is short for `std::experimental`

Example: Reaching Peak FLOP



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

single-precision multiply-add

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports), C++26 `std::simd` prototype

```
void peak(benchmark::State& state) {
  float x = 0.f;
  fake_modify(x);
  for (auto _ : state) {
    x = x * 3.f + 1.f;
  }
  fake_read(x);
}
```

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

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

single-precision multiply-add

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports), C++26 `std::simd` prototype

```cpp
void peak(benchmark::State& state) {
  float x = 0.f;
  fake_modify(x);
  for (auto _ : state) {
    x = x * 3.f + 1.f;
  }
  fake_read(x);
}
```

```cpp
void peak(benchmark::State& state) {
  simd<float, simd<float>::size() * 8> x = 0.f;
  fake_modify(x);
  for (auto _ : state) {
    x = x * 3.f + 1.f;
  }
  fake_read(x);
}
```

	`g++ -O3 -DNDEBUG`	`g++ -O3 -DNDEBUG -march=native`
scalar	0.25 FLOP/cycle	0.5 FLOP/cycle
data-parallel	8 FLOP/cycle	64 FLOP/cycle

$\frac{64}{0.25} = 256$ times faster

A data-parallel type wider than the default can increase ILP!

single-precision multiply-add

Linux, GCC 13, Intel Xeon W-2145 (2 AVX-512 FMA ports), C++26 `std::simd` prototype

```
void peak(benchmark::State& state) {
  float x = 0.f;
  fake_modify(x);
  for (auto _ : state) {
    x = x * 3.f + 1.f;
  }
  fake_read(x);
}
```

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

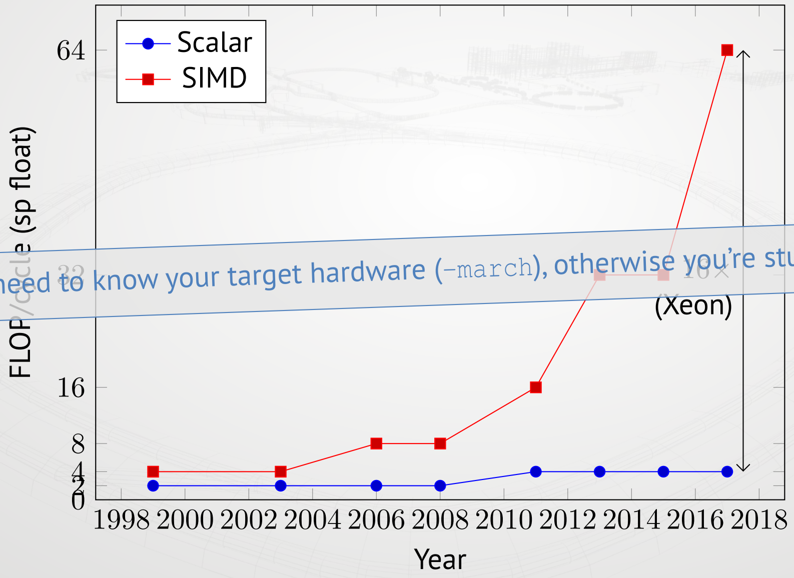
The naive benchmark was slow because the code didn't contain any parallelism:
one long dependency chain

	g++ -O3 -DNDEBUG	g++ -O3 -DNDEBUG -march=native
scalar	0.25 FLOP/cycle	0.5 FLOP/cycle
data-parallel	8 FLOP/cycle	64 FLOP/cycle

$\frac{64}{0.25} = 256$ times faster

A data-parallel type wider than the default can increase ILP!

x86



Compilers need to know your target hardware (`-march`), otherwise you're stuck in 2006!

Our non-parallel reality

We write and use code as impossible to optimize as this naive benchmark all the time!

- Example: `float std::cos(float)`
- This interface is bad for performance, so compilers replace `std::cos` calls with `__builtin_cosf`.
  - allows compile-time evaluation for constant inputs
  - enables vectorization if the caller does multiple calls
- Compilers will not be modified to replace your library functions with compiler builtins.

🤔

Calling a function (over library boundaries) with a single input/output value inhibits parallelization (SIMD & ILP).

An alternative

```
std::simd<float> std::cos(std::simd<float>)
```

Just sayin'

Consider using a "`T` or `simd<T>`" concept for more generality!

An alternative

```
std::simd<float> std::cos(std::simd<float>)
```

Just sayin'

Consider using a "`T` or `simd<T>`" concept for more generality!

`stdx::simd` Overview

Data-Parallel Types

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

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

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

0	+	1

vs.

0	+	1
0	+	1
0	+	1
0	+	1

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

stdx::simd::size()

- simd<T>::size() is a constant expression, denoting the number of elements.
- A default size() is chosen by the implementation, depending on the target.
- For most targets
  simd<T>::size() * sizeof(T) == simd<U>::size() * sizeof(U) holds:

float	float	float	float	float	float	float	float

double	double	double	double

If you really want to rely on it, add a static_assert

Synopsis

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

template <typename T>
  using native_simd = simd<T, ...>;
```

```
template <typename T, typename Abi>
  class simd_mask;

template <typename T>
  using native_simd_mask = simd_mask<T, ...>;
```

- `simd<T>` behaves just like `T` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool`
- `simd_mask<T>` behaves like `bool` (element-wise)
  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: `native_simd<int>` on x86 can be one `xmm` register (`sizeof == 16`), one `ymm` register (`sizeof == 32`), or one `zmm` register (`sizeof == 64`).

Synopsis

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

template <typename T>
  using native_simd = simd<T, ...>;
```

```
template <typename T, typename Abi>
  class simd_mask;

template <typename T>
  using native_simd_mask = simd_mask<T, ...>;
```

- `simd<T>` behaves just like `T` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool`
- `simd_mask<T>` behaves like `bool` (element-wise)
  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: `native_simd<int>` on x86 can be one `xmm` register (`sizeof == 16`), one `ymm` register (`sizeof == 32`), or one `zmm` register (`sizeof == 64`).

Synopsis

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

template <typename T>
  using native_simd = simd<T, ...>;
```

```
template <typename T, typename Abi>
  class simd_mask;

template <typename T>
  using native_simd_mask = simd_mask<T, ...>;
```

- `simd<T>` behaves just like `T` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool`
- `simd_mask<T>` behaves like `bool` (element-wise)
  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: `native_simd<int>` on x86 can be one `xmm` register (`sizeof == 16`), one `ymm` register (`sizeof == 32`), or one `zmm` register (`sizeof == 64`).

Constructor examples

```cpp
stdx::simd<int> x0;

stdx::simd<int> x1{};

stdx::simd<int> x2 = 1;

stdx::simd<int> x3(p, stdx::element_aligned);

stdx::simd<int> iota([](int i) { return i; });
```

?	?	?	...
0	0	0	...
1	1	1	...
p[0]	p[1]	p[2]	...
0	1	2	...

Constructor examples

`stdx::simd<int> x0;`	?	?	?	...
`stdx::simd<int> x1{};`	0	0	0	...
`stdx::simd<int> x2 = 1;`	1	1	1	...
`stdx::simd<int> x3(p, stdx::element_aligned);`	p[0]	p[1]	p[2]	...
`stdx::simd<int> iota([](int i) { return i; });`	0	1	2	...

Loads & stores

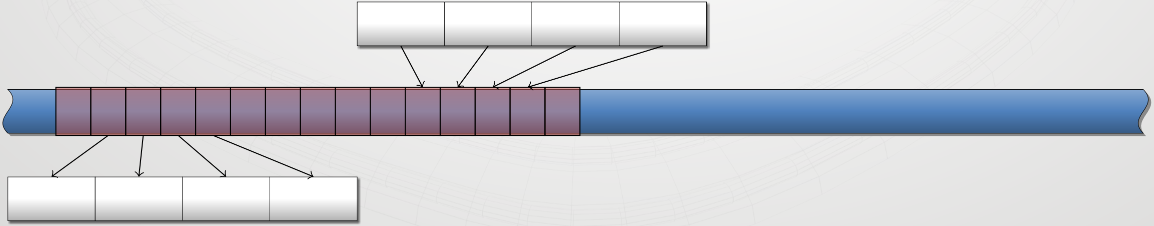
```
class simd {
  simd(const auto* ptr, auto flags);

  void
  copy_from(const auto* ptr, auto flags);

  void
  copy_to(auto* ptr, auto flags);
};
```

(simplified)

- Loads copy simd<T>::size() elements from a contiguous array chunk into the simd<T> elements.
- Stores do the reverse.
- Consider loads & stores equivalent to dereferencing an iterator in the scalar case.



Arithmetic & math

```cpp
void f(stdx::simd<float> x,
       stdx::simd<float> y) {
  x += y;       // x.size() additions
  x = sqrt(x); // x.size() square roots
  ...
// etc. all operators and <cmath>
}
```

- Operations act element-wise
- Speed-up is often a factor of `simd<T>::size()`, but may be less, depending on hardware details.

x0	+=	y0
x1	+=	y1
x2	+=	y2
x3	+=	y3

Same for compares (element-wise)

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

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

Same for compares (element-wise)

```cpp
void f(stdx::simd<float> x, stdx::simd<float> y) {
  if (x < y) { x = y; } // nonono, you don't write 'if (truefalsetruetrue)' either
  where(x < y, x) = y; // x = y but only for the elements where x < y
  if (all_of(x < y)) {...}       // this makes sense, yes
}
```

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

if (	x0	<	y0	)	x0	=	y0	;
if (	x1	<	y1	)	x1	=	y1	;
if (	x2	<	y2	)	x2	=	y2	;
if (	x3	<	y3	)	x3	=	y3	;

Permutations (vir-simd)

Permutations enable

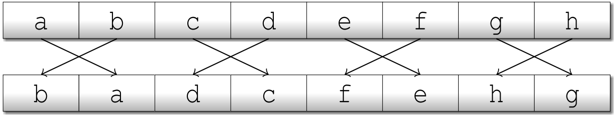
- reductions,
- vectorization of loops with dependent iterations, and
- more efficient permutations of larger arrays.

Example

```
stdx::native_simd<float> permute_even_odd(stdx::native_simd<float> x) {
  return vir::simd_permute(x, [](auto idx) { return idx ^ 1; });
}
```

with AVX, compiles to:

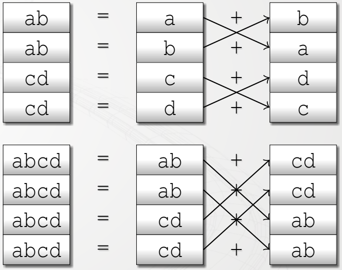
```
vpermilps ymm0, ymm0, 177
```



Reductions

- `simd_mask` **reductions:**
  `all_of`, `any_of`, `none_of`, `popcount`, `find_first_set`, `find_last_set`
- `simd` **reductions:**
  `reduce`, `hmin`, `hmax`

SIMD tree reduction:



Example

```cpp
void f(stdx::simd<float> x) {
  float sum = stdx::reduce(x);
  float product = stdx::reduce(x, std::multiplies());
  float sum_of_pos_x = stdx::reduce(where(x > 0, x));
  float minimum = stdx::hmin(x);
  int min_idx = stdx::find_first_set(x == minimum);
}
```

Automatic type vectorization (vir-simd)

```
struct Point {
  float x, y, z;
};
using PointV = vir::simdize<Point>;

// equivalent to
struct PointV {
  simd<float> x, y, z;

  // stdx::simd-like interface (loads & stores, broadcast, size, subscripting, ...)
};
```

A much better solution should be possible after *reflection* lands in the C++ standard.

Documentation

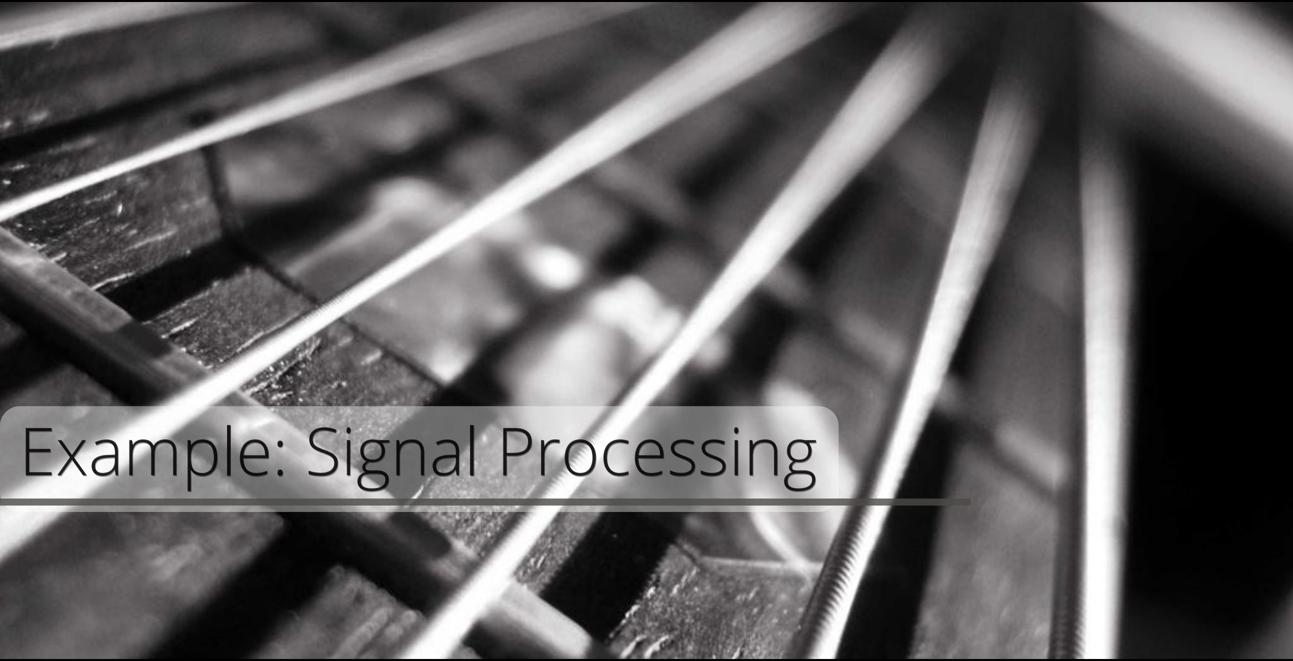
`std::experimental::simd`

`https://en.cppreference.com/w/cpp/experimental/simd`

vir-simd

`https://github.com/mattkretz/vir-simd/blob/master/README.md`

or just `https://github.com/mattkretz/vir-simd`



Example: Signal Processing

Example: signal processing

Let's start really simple, with a gain function:

```
void apply_gain(std::span<const float> in, std::span<float> out, float gain) {
  assert(in.size() == out.size());
  // implement here
}
```

Implementation using a for-loop:

```
  for (std::size_t i = 0; i < in.size(); ++i) {
    out[i] = in[i] * gain;
  }
```

Implementation using a standard algorithm:

```
  std::transform(in.begin(), in.end(), out.begin(), [&](float sample) {
      return sample * gain;
  });
```

Example: signal processing

Let’s start really simple, with a gain function:

```
void apply_gain(std::span<const float> in, std::span<float> out, float gain) {
  assert(in.size() == out.size());
  // implement here
}
```

Implementation using a for-loop:

```
  for (std::size_t i = 0; i < in.size(); ++i) {
    out[i] = in[i] * gain;
  }
```

Implementation using a standard algorithm:

```
  std::transform(in.begin(), in.end(), out.begin(), [&](float sample) {
      return sample * gain;
  });
```

Example: signal processing

Let's start really simple, with a gain function:

```
void apply_gain(std::span<const float> in, std::span<float> out, float gain) {
  assert(in.size() == out.size());
  // implement here
}
```

Implementation using a for-loop:

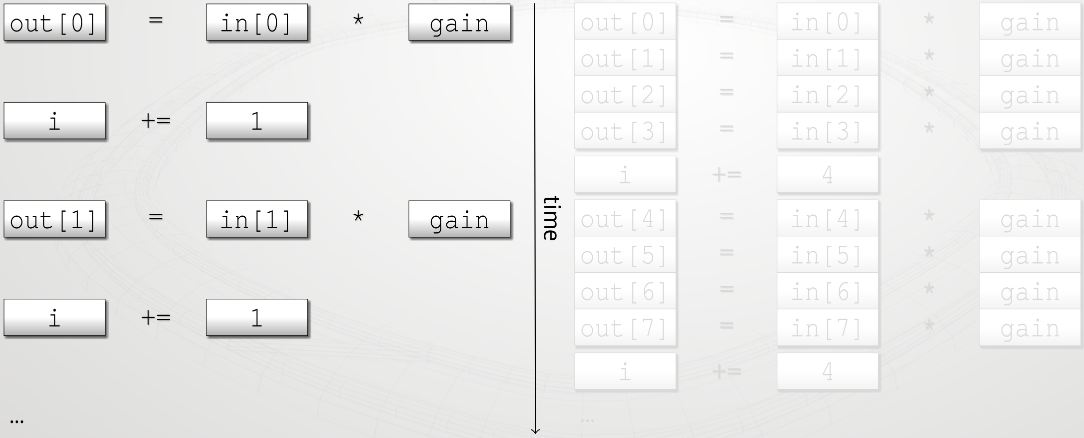
```
  for (std::size_t i = 0; i < in.size(); ++i) {
    out[i] = in[i] * gain;
  }
```

Implementation using a standard algorithm:

```
  std::transform(in.begin(), in.end(), out.begin(), [&](float sample) {
      return sample * gain;
  });
```

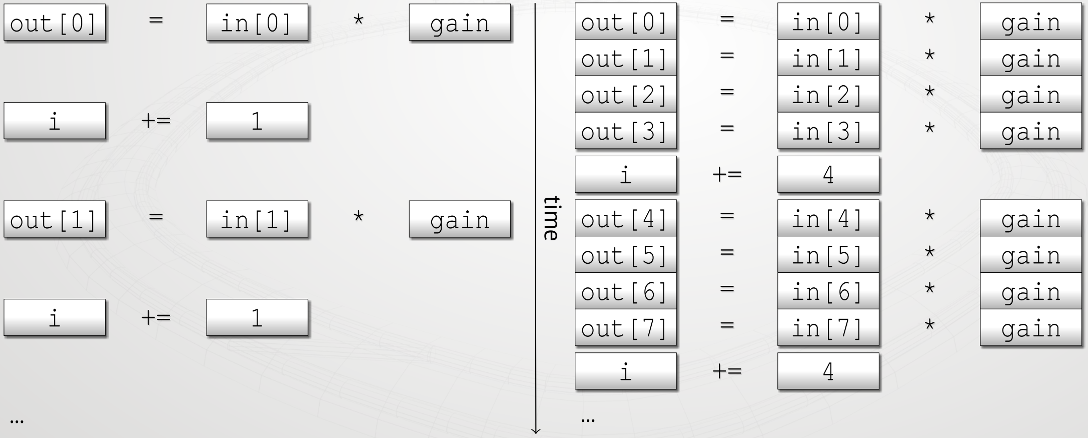
How can we use `stdx::simd` here?

Look at loops for data-parallelism.



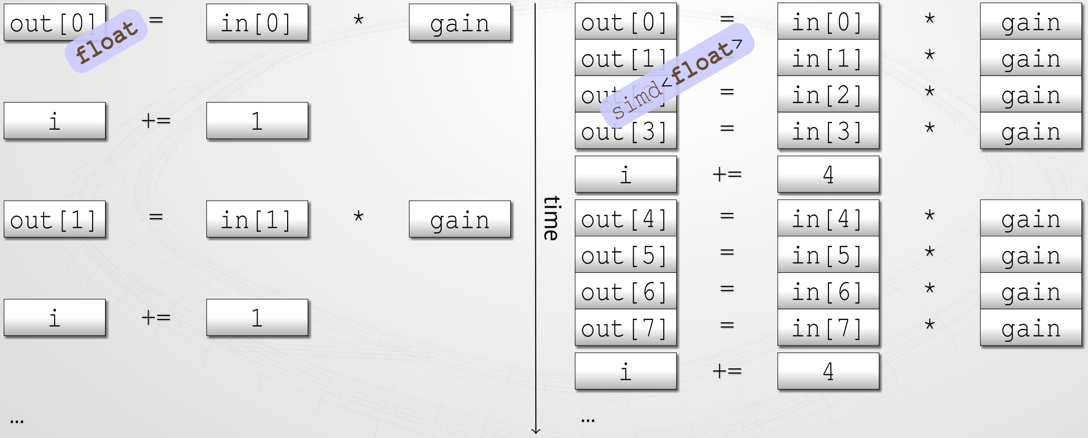
How can we use `stdx::simd` here?

Look at loops for data-parallelism.



How can we use stdx::simd here?

Look at loops for data-parallelism.



Apply gain using stdx::simd

```
for (std::size_t i = 0; i < in.size(); ++i) {
  out[i] = in[i] * gain;


}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. What is missing?
5. Now we can go faster. But it's possible we invoke UB. Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```cpp
for (std::size_t i = 0; i < in.size(); ++i) {
  float sample = in[i];
  sample *= gain;
  out[i] = sample;
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. What is missing?
5. Now we can go faster. But it's possible we invoke UB. Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i < in.size(); ++i) {
  simd<float> sample = in[i]; // wrong: copies in[i] to all elements of the simd
  sample *= gain;  // OK
  out[i] = sample; // Error: cannot convert simd<float> to float
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. What is missing?
5. Now we can go faster. But it's possible we invoke UB. Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using stdx::simd

```
for (std::size_t i = 0; i < in.size(); ++i) {
  auto sample = simd<float>(&in[i], stdx::element_aligned); // load
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);           // store
}
```

1. Where's the float?
2. Let's turn the temporary into an lvalue.
3. We start with changing the float into a simd<float>. Not correct yet.
4. Introduced load and store. It compiles. What is missing?
5. Now we can go faster. But it's possible we invoke UB. Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i < in.size(); ++i) {
  auto sample = simd<float>(&in[i], stdx::element_aligned); // load
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);           // store
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. Every sample is processed multiple times.
5. Now we can go faster. But it's possible we invoke UB.
   Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i < in.size(); i += simd<float>::size()) {
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. Every sample is processed multiple times.
5. Now we can go faster. But it's possible we invoke UB.
   Why?
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i < in.size(); i += simd<float>::size()) {
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. Every sample is processed multiple times.
5. Now we can go faster. But it's possible we invoke UB.
   `4 < 7` is `true`, but accessing `in[4 + simd::size - 1]` is out of bounds.
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i + simd<float>::size() <= in.size(); i += simd<float>::size()
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
```

1. Where's the `float`?
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. Every sample is processed multiple times.
5. Now we can go faster. But it's possible we invoke UB.
   `4 < 7` is `true`, but accessing `in[4 + simd::size - 1]` is out of bounds.
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd`

```
for (std::size_t i = 0; i + simd<float>::size() <= in.size(); i += simd<float>::size()
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
```

Sigh … and we're running out of slide space

1. Where's the float
2. Let's turn the temporary into an lvalue.
3. We start with changing the `float` into a `simd<float>`. Not correct yet.
4. Introduced load and store. It compiles. Every sample is processed multiple times.
5. Now we can go faster. But it's possible we invoke UB.
   `4 < 7` is `true`, but accessing `in[4 + simd::size - 1]` is out of bounds.
6. This condition avoids the out-of-bounds. But we might skip samples at the end.

Apply gain using `stdx::simd` with epilogue

```
using floatv = stdx::simd<float>;
std::size_t i = 0;
for (; i + floatv::size() <= in.size(); i += floatv::size()) {
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
//
// TODO: process samples in range [i, in.size())
//
```

1. Introduce an alias for our `simd` type. Recommended practice anyway.
2. In SIMD programming, this “handle the remainder” is called an *epilogue*.

Apply gain using `stdx::simd` with epilogue

```
using floatv = stdx::simd<float>;
std::size_t i = 0;
for (; i + floatv::size() <= in.size(); i += floatv::size()) {
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
for (; i < in.size(); ++i) {
  out[i] = in[i] * gain;
}
```

1. Introduce an alias for our `simd` type. Recommended practice anyway.
2. In SIMD programming, this “handle the remainder” is called an *epilogue*.

Apply gain using `stdx::simd` with epilogue

```
using floatv = stdx::simd<float>;
std::size_t i = 0;
for (; i + floatv::size() <= in.size(); i += floatv::size()) {
  auto sample = simd<float>(&in[i], stdx::element_aligned);
  sample *= gain;
  sample.copy_to(&out[i], stdx::element_aligned);
}
for (; i < in.size(); ++i) {
  out[i] = in[i] * gain;
}
```

Phew. Now that you've seen this… I have good news…

1. Introduce an alias for our `simd` type. Recommended practice anyway.
2. In SIMD programming, this "handle the remainder" is called an *epilogue*.

Good news 👍

**GR4 provides the above for you,
simply implement `processOne` and you're done!**

Example

```cpp
template <gr::meta::t_or_simd<float> V>
V processOne(V samples) {
  return samples * gain;
}
```

You can also overload on `simd`:

```cpp
float processOne(float x) { return x * gain; }

template <typename Abi>
simd<float, Abi> processOne(simd<float, Abi> x) { return x * gain; }
```

Good news 👍

**GR4 provides the above for you,
simply implement `processOne` and you're done!**

Example

```cpp
template <gr::meta::t_or_simd<float> V>
V processOne(V samples) {
  return samples * gain;
}
```

You can also overload on `simd`:

```cpp
float processOne(float x) { return x * gain; }

template <typename Abi>
simd<float, Abi> processOne(simd<float, Abi> x) { return x * gain; }
```

More good news 👍👍

vir-simd provides algorithms that implement the above for you!

Example

Start with:

```
void apply_gain(std::span<const float> in, std::span<float> out, float gain) {
  std::transform(std::execution::seq,
                 in.begin(), in.end(), out.begin(), [&](float x) {
      return x * gain;
  });
}
```

More good news 👍👍

vir-simd provides algorithms that implement the above for you!

Example

A minor tweak with huge effect:

```
void apply_gain(std::span<const float> in, std::span<float> out, float gain) {
  std::transform(vir::execution::simd,
                 in.begin(), in.end(), out.begin(), [&](auto x) {
      return x * gain;
  });
}
```

A Benchmark

In Code

trivial data-parallelism

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

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

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

`vir::execution::simd` is part of the vir-simd library.

In Code

trivial data-parallelism

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

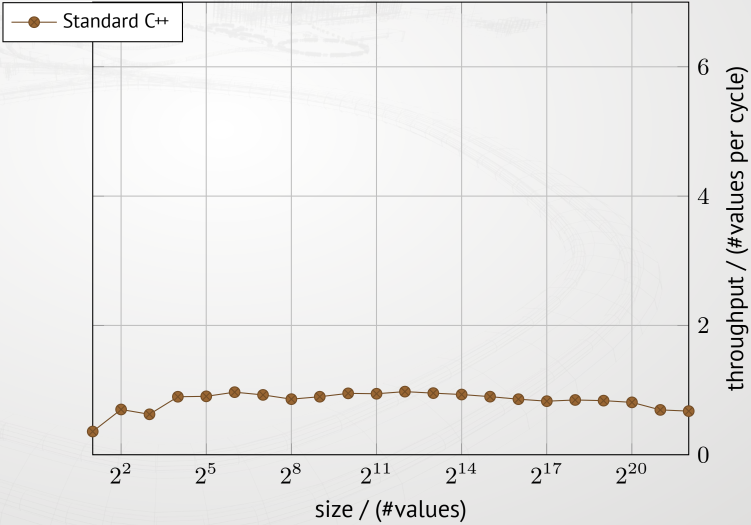
```
std::for_each(std::execution::unseq,
  data.begin(), data.end(),
  [](int& x) {
    x += 1;
  });
```

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

`vir::execution::simd` is part of the vir-simd library.

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

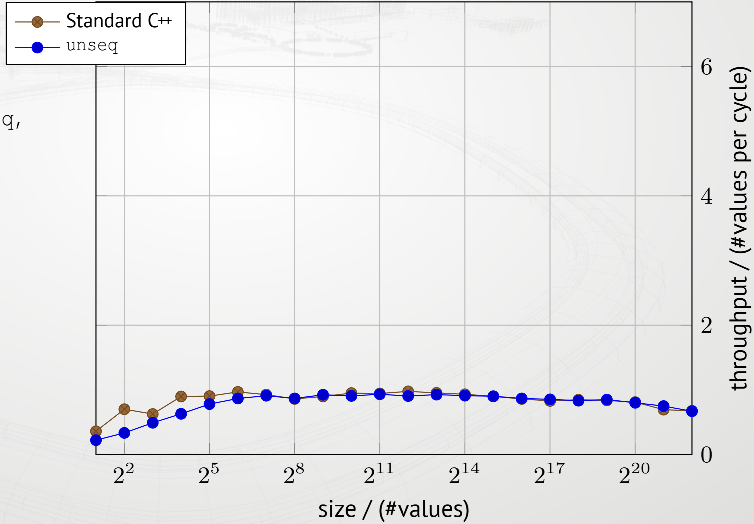
```
for (int& x : data) {
  x += 1;
}
```



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

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

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

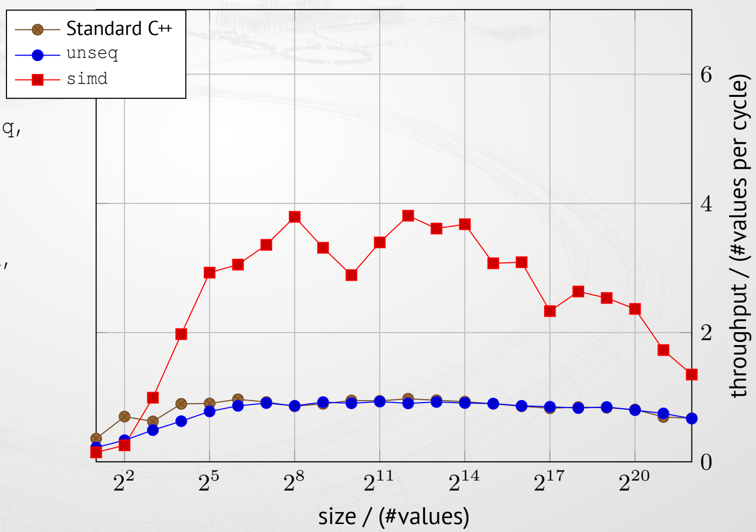


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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

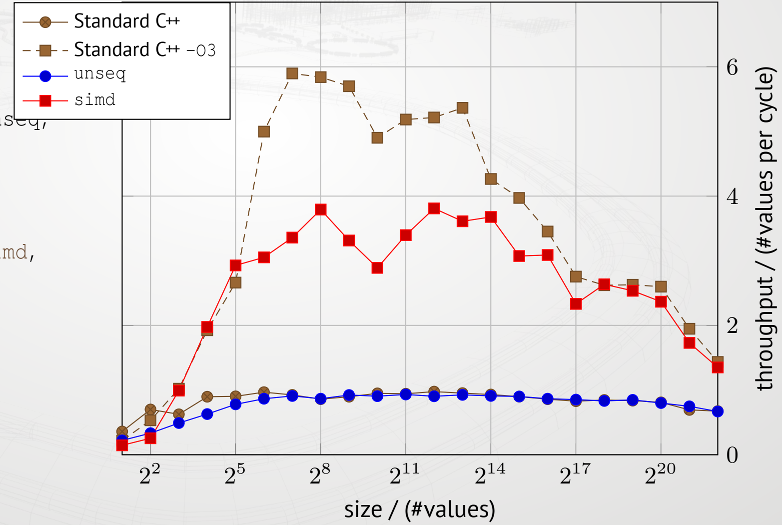


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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

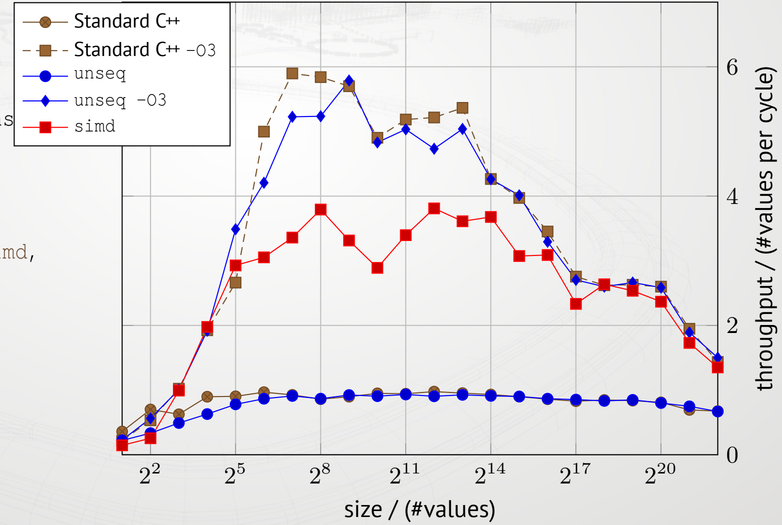


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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

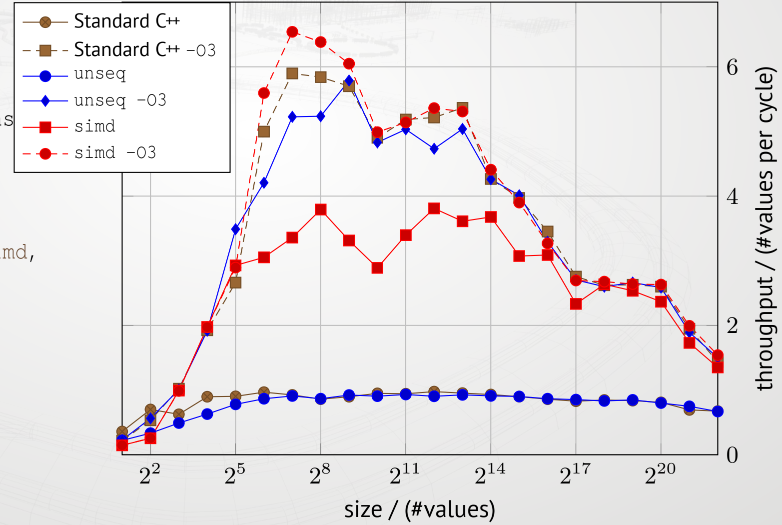


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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



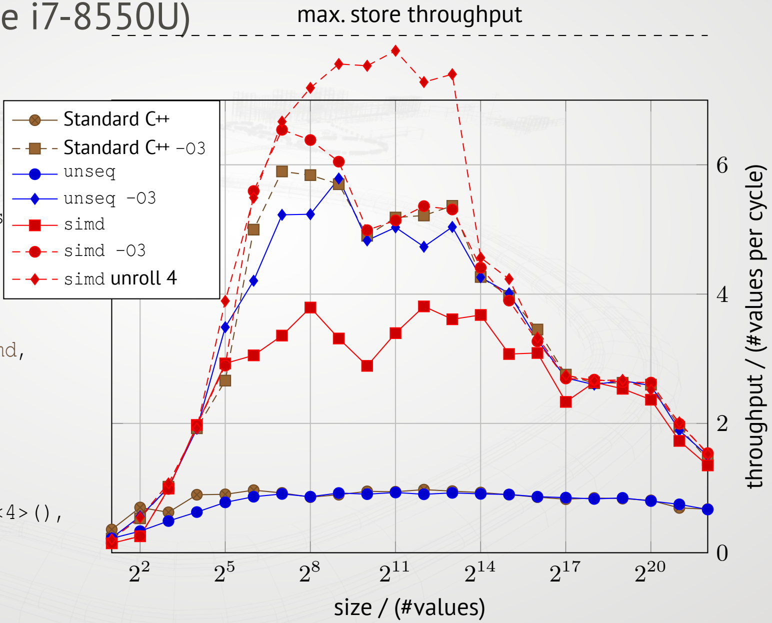
Results (GCC 13, Intel Skylake i7-8550U)

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

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

```
std::for_each(vir::execution::simd,
  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; }
);
```



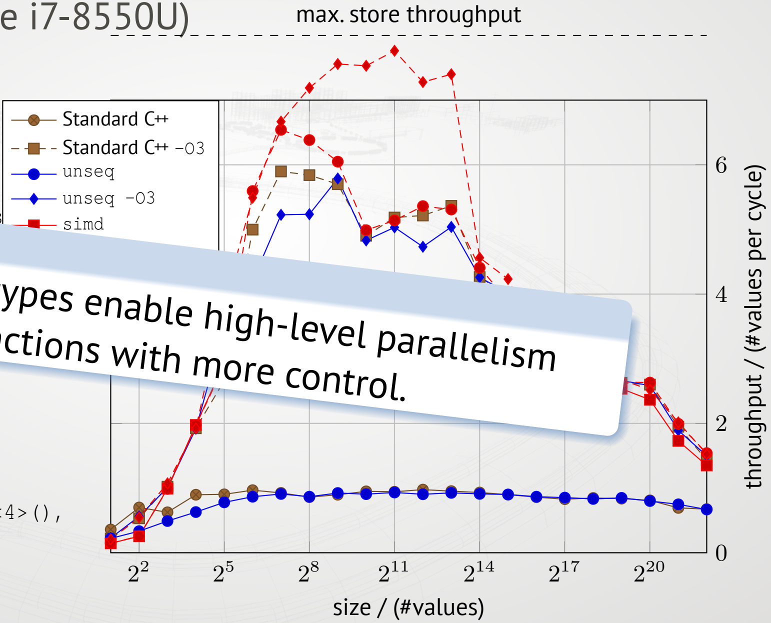
Results (GCC 13, Intel Skylake i7-8550U)

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

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

```
std::for_each(vir::execution ...
  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 #2

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

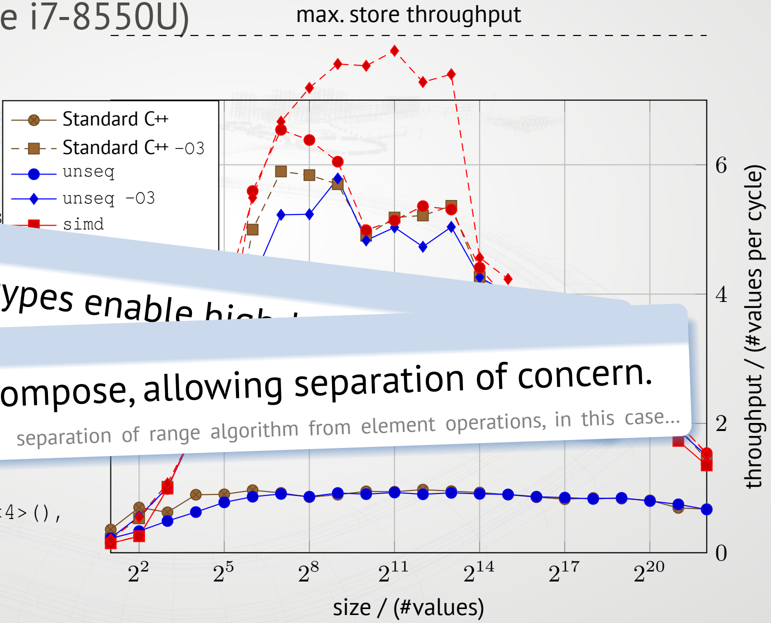
Results (GCC 13, Intel Skylake i7-8550U)

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

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

```
std:
  da
  []
);
```

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



Take-Away #2

Data-parallel types enable high... abst...

Take-Away #3

Data-parallel types compose, allowing separation of concern.

separation of range algorithm from element operations, in this case...

SIMD and Merged Blocks

Amdahl’s law

> “the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used.”
>
> [Reddy, Martin (2011). API Design for C++. doi:10.1016/2010-0-65832-9]

In other words:
The overall speedup from using SIMD of a flow graph may be frustratingly low, even though SIMD improves throughput & latency per block.

Because …

Amdahl's law

"the overall performance improvement gained by optimizing a single part of a system is limited by the fraction of time that the improved part is actually used."

[Reddy, Martin (2011). API Design for C++. doi:10.1016/2010-0-65832-9]

In other words:
The overall speedup from using SIMD of a flow graph may be frustratingly low,
even though SIMD improves throughput & latency per block.

Because ...

time

invoke block a	a: process samples	invoke block b	b: process samples

Run-time configurable communication between blocks has a cost:

- block output written to memory,
- block scheduling
- buffer management (avoid over- and underruns),
- block input read from memory
- (additional cache/memory latencies)

So, we need to write our application as one loop? 🤔 😱

Alternative:

- let 'b' process the results from 'a' immediately.
- This requires connection at compile-time but does not require implementing a new block that merges their functionality.

time

invoke block a	a: process samples	invoke block b	b: process samples

Run-time configurable communication between blocks has a cost:

- block output written to memory,
- block scheduling
- buffer management (avoid over- and underruns),
- block input read from memory
- (additional cache/memory latencies)

So, we need to write our application as one loop? 🤔 😱

Alternative:

- let 'b' process the results from 'a' immediately.
- This requires connection at compile-time but does not require implementing a new block that merges their functionality.

time

invoke block a | a: process samples | invoke block b | b: process samples

after compile-time connection:

invoke block a+b | a: process samples | b: process samples

with a good chance of interleaving computation:

invoke block a+b | a: process samples
b: process samples

- a mergable block implements `processOne` instead of `processBulk`
- “`One`” means “one object”, *not* “one sample”
- “`One`” can be a `simd<T>` object (multiple samples) or a `T` object (one sample)
- thus, a better picture looks like:

time

invoke block a

`a.work()` loops over

`a.processOne(...)`

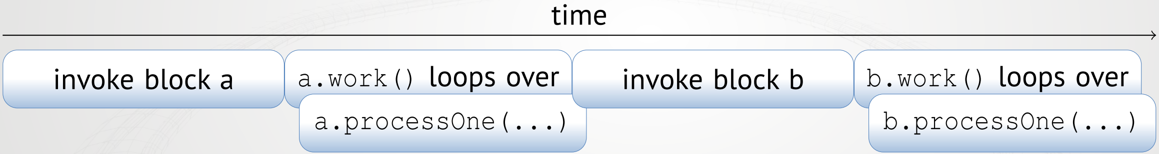
invoke block b

`b.work()` loops over

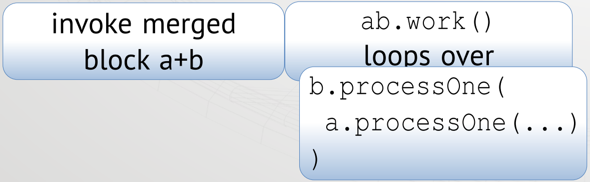
`b.processOne(...)`

and turns into:

- a mergable block implements `processOne` instead of `processBulk`
- “`One`” means “one object”, *not* “one sample”
- “`One`” can be a `simd<T>` object (multiple samples) or a `T` object (one sample)
- thus, a better picture looks like:



and turns into:



🏆 Performance

A merged block is an efficient loop implementation

...

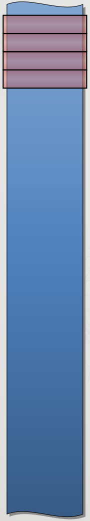
🏆 Performance

A merged block is an efficient loop implementation

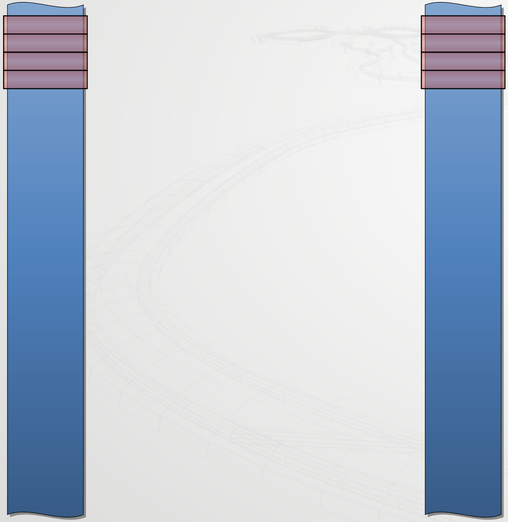
🏆 Abstraction

… via composition of existing components!

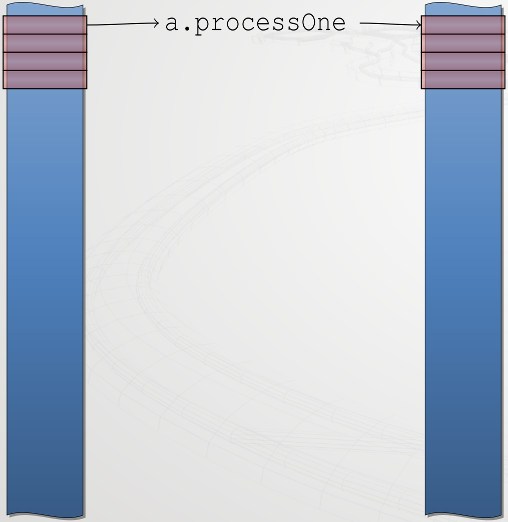
In Pictures: not merged, no SIMD



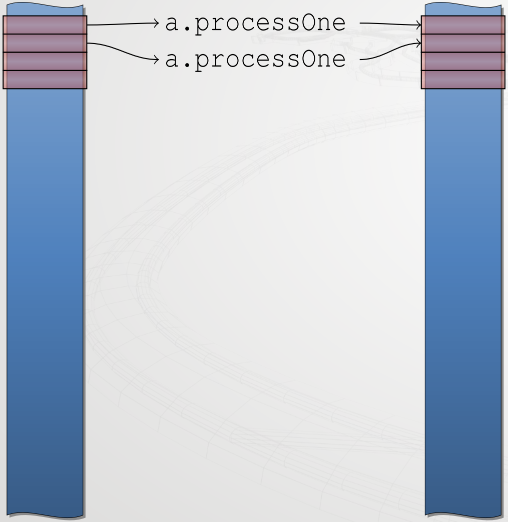
In Pictures: not merged, no SIMD



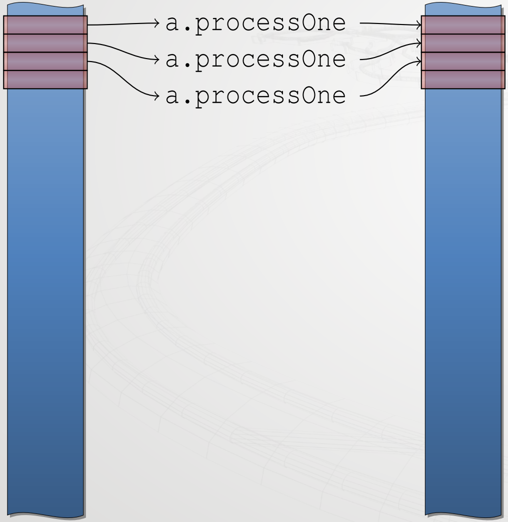
In Pictures: not merged, no SIMD



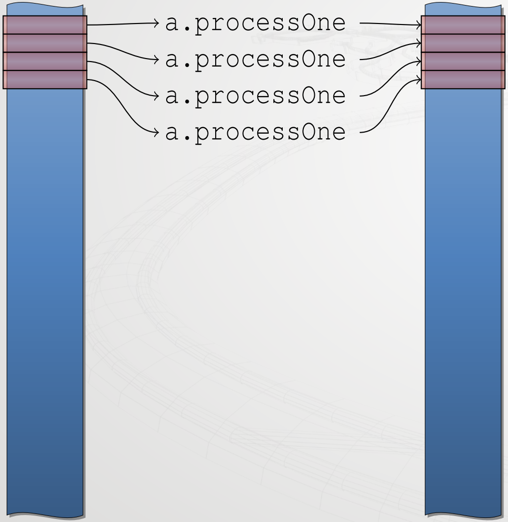
In Pictures: not merged, no SIMD



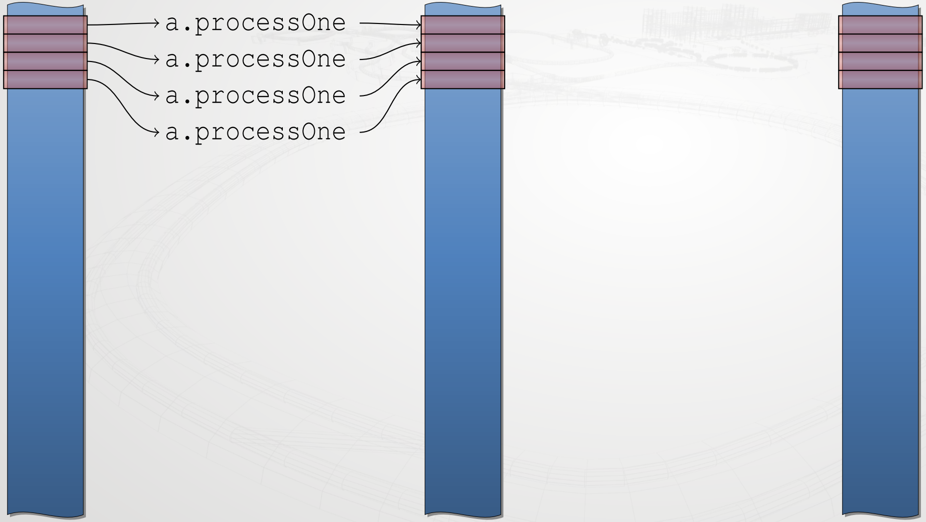
In Pictures: not merged, no SIMD



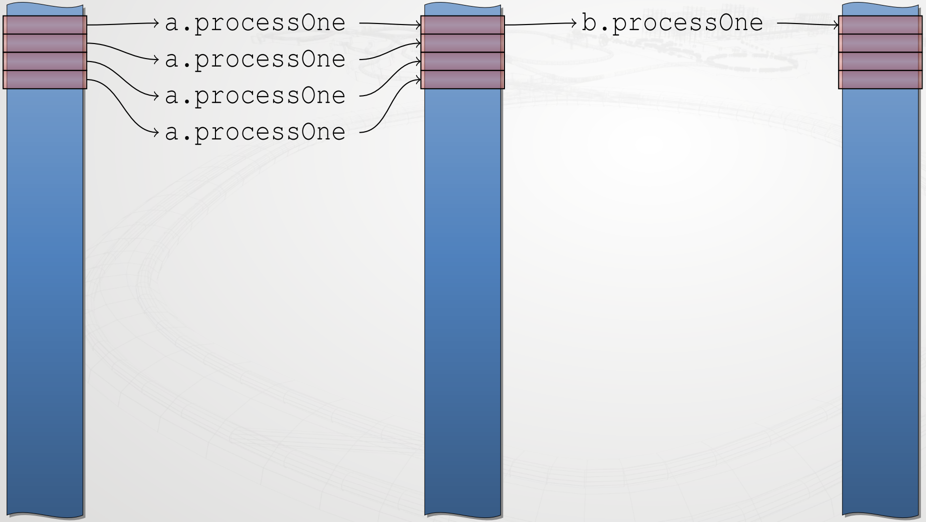
In Pictures: not merged, no SIMD



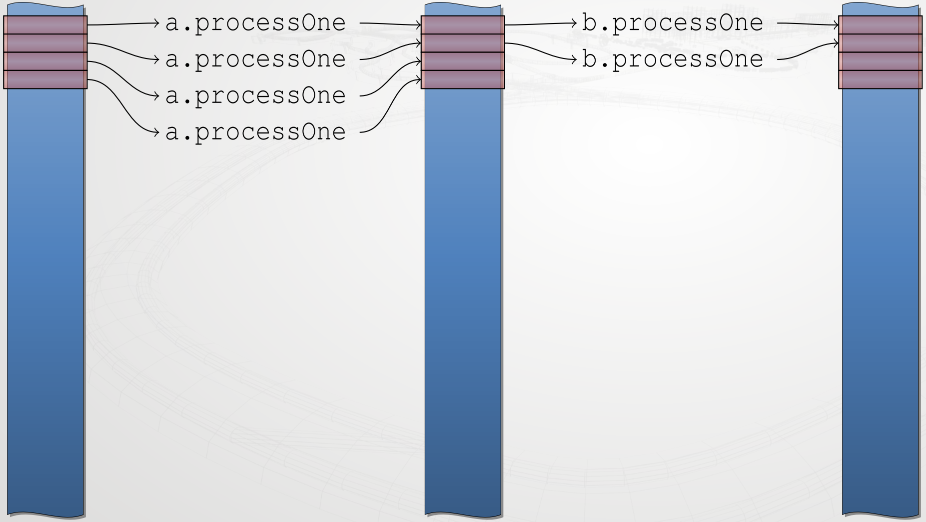
In Pictures: not merged, no SIMD



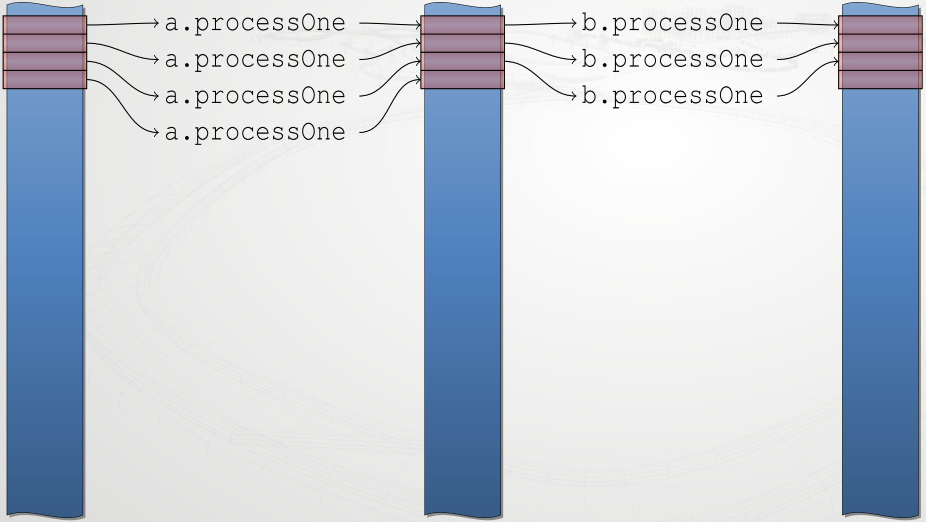
In Pictures: not merged, no SIMD



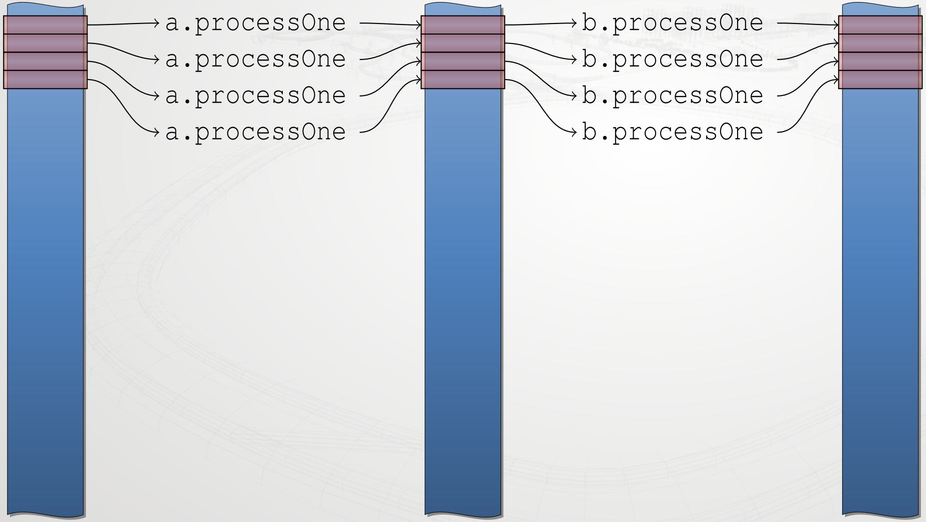
In Pictures: not merged, no SIMD



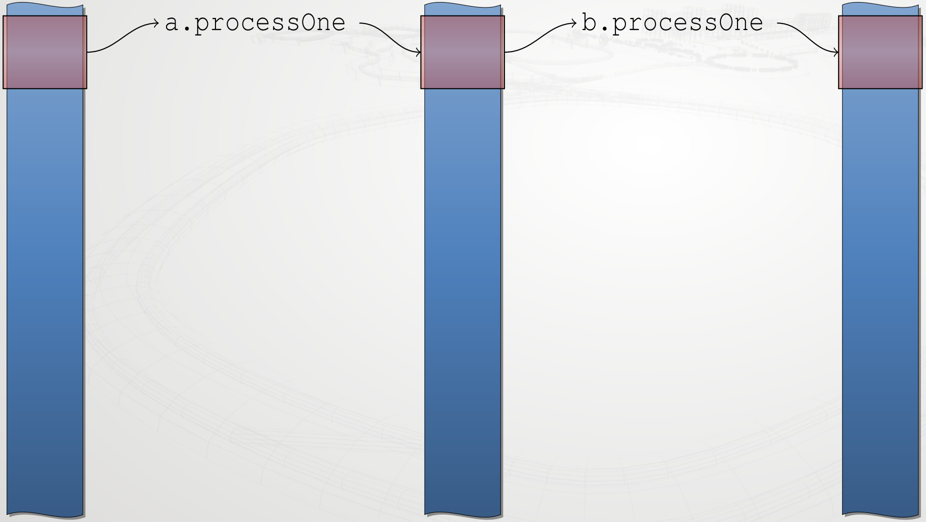
In Pictures: not merged, no SIMD



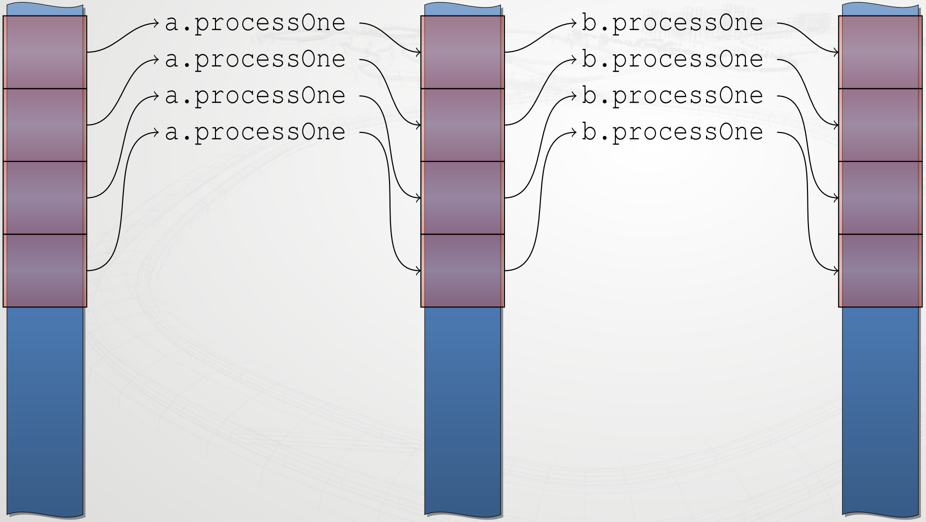
In Pictures: not merged, no SIMD



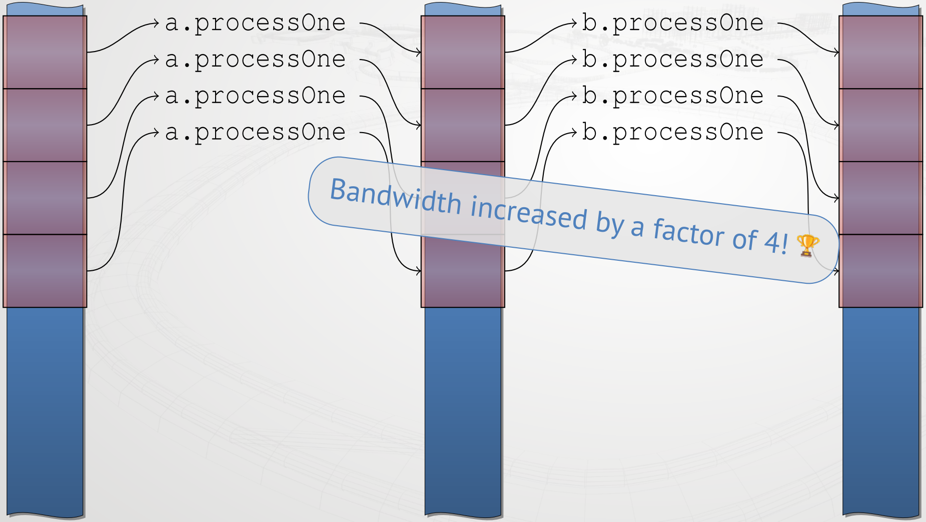
In Pictures: not merged, with 4-wide SIMD



In Pictures: not merged, with 4-wide SIMD

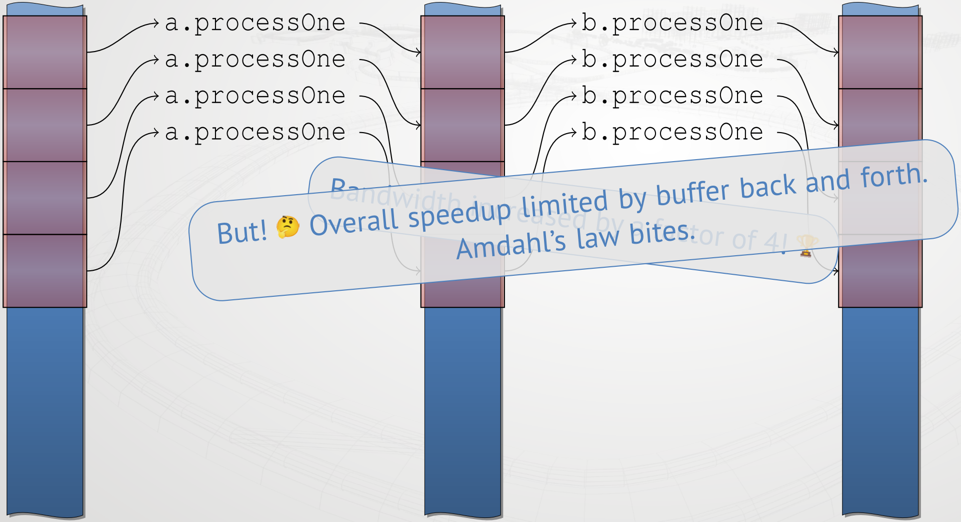


In Pictures: not merged, with 4-wide SIMD



Bandwidth increased by a factor of 4! 🏆

In Pictures: not merged, with 4-wide SIMD

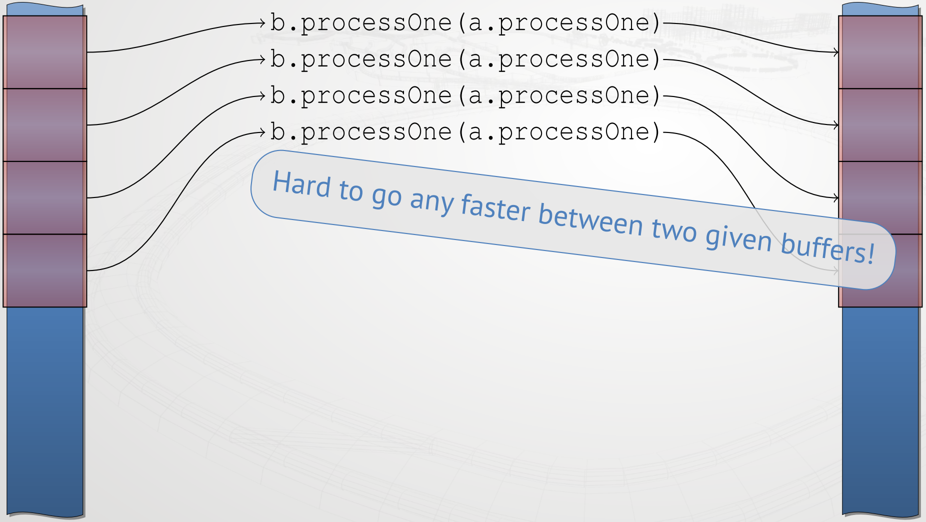


Bandwidth increased by a factor of 4! 🏆

But! 🤔 Overall speedup limited by buffer back and forth.
Amdahl's law bites.

In Pictures: merged, with 4-wide SIMD

In Pictures: merged, with 4-wide SIMD



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 wording, waiting on the above)
  - adds `std::complex` to vectorizable types (`simd<complex>` *not* `complex<simd>`)
- P2964 (C++26 is unlikely, scope & design review)
  - add scoped and unscoped enums to vectorizable types (e.g. `std::byte`)
  - add user-defined numeric types to vectorizable types (fixed-point, saturating integers, etc.)

We want more interoperability

- Conversion to/from `std::array`, `std::span`, or generally any contiguous range.
- Mask conversion to/from `std::bitset`.
- Initializer list constructor for `simd`.

  Tricky because of portability concerns.

- Make `simd` and `simd_mask` (read-only) ranges.
  - ⇒ Formatting (`std::format`).
  - ⇒ Makes it easy to flatten a `vector<simd<T>>` into a range of `T`.

Better gather & scatter integration

Example

```
void f(std::vector<float>& data, t_or_simd<int> auto indexes) {
  data[indexes] = std::sin(data[indexes]);
}
```

1. `stdx::simd` is for you – because you care about latency & throughput!
2. `stdx::simd` expresses data-parallelism – compiled to SIMD (and ILP[2])!
3. Focus on data-parallelism not SIMD instructions/registers!
4. `stdx::simd` enables separation of
   - serial execution,
   - synchronously parallel execution (SIMD and ILP), and
   - asynchronously parallel execution (threads).
5. `stdx::simd` guides you to design scalable and portable parallelization.
6. vir-simd implements high-level standard algorithms to simplify SIMD access to ranges of scalars.
7. vir-simd implements high-level & specialized parallelism abstractions with more control where loop-vectorization fails.
8. `stdx::simd` provides an API & ABI for vectorization across translation units (including library boundaries).

[2] much better with C++26

High-throughput computing without overhead

```
mov     eax, DWORD PTR [rdi]
imul    eax, eax
mov     DWORD PTR [rdi], eax
```

- src/dst: array of integers
- throughput: 0.5/1/1 cycles (Intel)
- integer multiplications: 1

```
vmovdqu32   zmm0, ZMMWORD PTR [rdi]
vpmulld     zmm0, zmm0, zmm0
vmovdqu32   ZMMWORD PTR [rdi], zmm0
```

- src/dst: array of integers
- throughput: 0.5/1/1 cycles (Intel)
- integer multiplications: 16

Take-Away #4

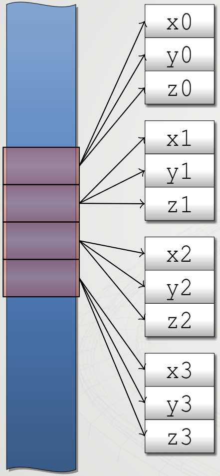
SIMD is relevant for low-latency, not only high-throughput

Data Structures for SIMD Processing

Consider this class template definition:

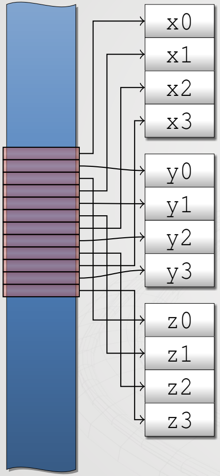
```
template <typename T> struct Point {
  T x, y, z;

  Point normalized() const {
    using std::sqrt;
    const auto scale = 1 / sqrt(x * x + y * y + z * z);
    return {x * scale, y * scale, z * scale};
  }
};
```



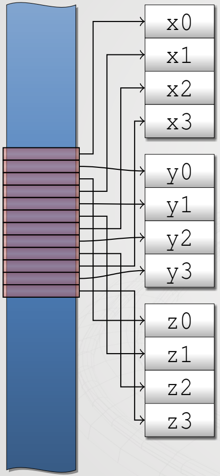
Array of Struct

`vector<Point<float>>`



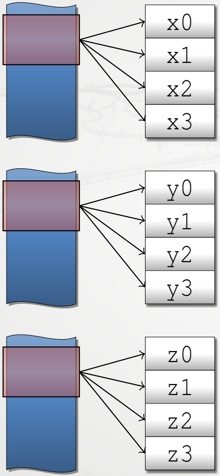
Array of Struct

`vector<Point<float>>`



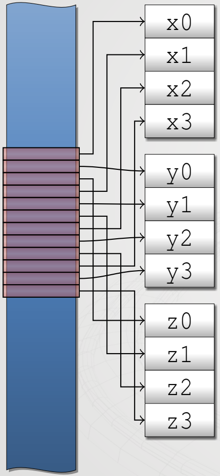
Array of Struct

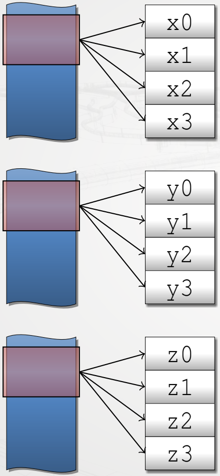
`vector<Point<float>>`

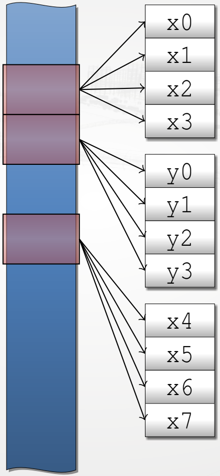


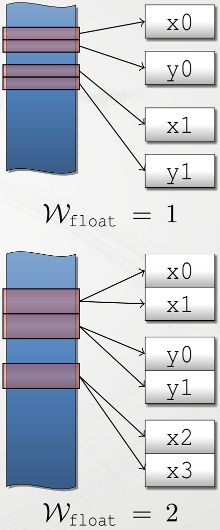
Struct of Array

`Point<valarray<float>>`









Array of Struct

`vector<Point<float>>`

Struct of Array

`Point<valarray<float>>`

Array of vectorized Struct

`vector<Point<simd<float>>>`

Example: normalize *N* 3-D points

```cpp
template <typename T> struct Point {
  T x, y, z;

  Point normalized() const {
    using std::sqrt;
    const auto scale
      = 1 / sqrt(x*x + y*y + z*z);
    return {x * scale,
            y * scale,
            z * scale};
  }
};
```

```cpp
void aos(const std::vector<Point<float>>& points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

```cpp
void soa(const Point<std::valarray<float>>& points) {
  points = points.normalized();
}
```

```cpp
void aovs(const std::vector<Point<      simd<float>>>&
          points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

Example: normalize *N* 3-D points

```
template <typename T> struct Point {
  T x, y, z;

  Point normalized() const {
    using std::sqrt;
    const auto scale
      = 1 / sqrt(x*x + y*y + z*z);
    return {x * scale,
            y * scale,
            z * scale};
  }
};
```

```
void aos(const std::vector<Point<float>>& points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

```
void soa(const Point<std::valarray<float>>& points) {
  points = points.normalized();
}
```

```
void aovs(const std::vector<Point<      simd<float>>>&
                points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

Example: normalize *N* 3-D points

```
template <typename T> struct Point {
  T x, y, z;

  Point normalized() const {
    using std::sqrt;
    const auto scale
      = 1 / sqrt(x*x + y*y + z*z);
    return {x * scale,
            y * scale,
            z * scale};
  }
};
```

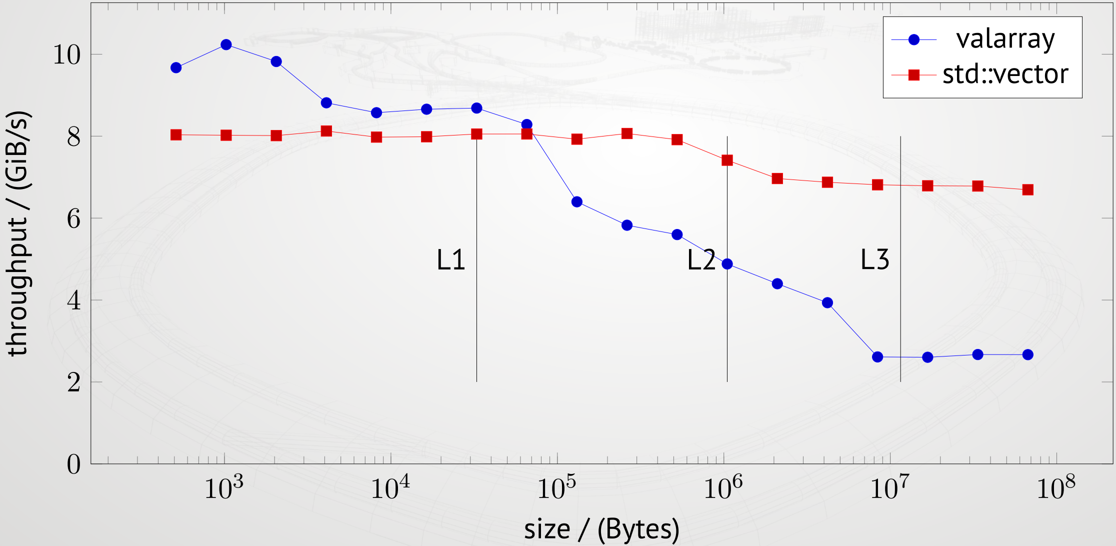
```
void aos(const std::vector<Point<float>>& points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

```
void soa(const Point<std::valarray<float>>& points) {
  points = points.normalized();
}
```

```
void aovs(const std::vector<Point<std::simd<float>>>&
              points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

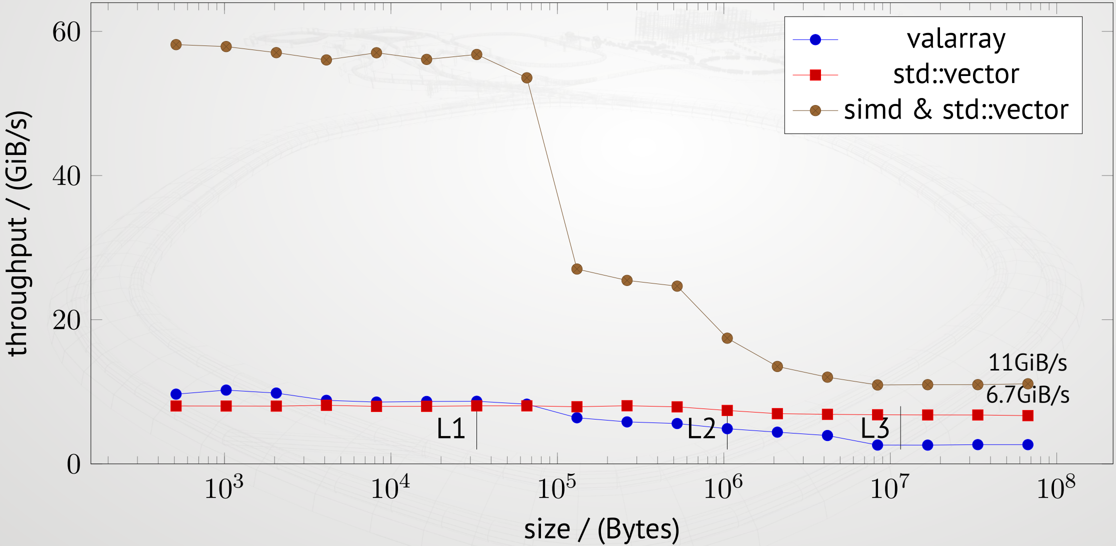
Normalization benchmark

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



Normalization benchmark

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



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;
}
```

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

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

```
; 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
```

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 $\mathcal{W}_{\mathrm{T}}$ 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 $\mathcal{W}_{\mathrm{T}}$ 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 $\mathcal{W}_{\mathrm{T}}$ 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

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)

```
template <typename T, typename Abi = ...>
class basic_simd {
  basic_simd() = default;
  basic_simd(T);
  basic_simd(std::contiguous_iterator auto, Flags = ...);
  basic_simd(Generator);
}
```

- 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 $\mathcal{W}_T$ 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.

Loads & stores (epilogue)

SIMD code typically needs an *epilogue*:

```
void f(std::vector<float>& data) {
  using floatv = std::simd<float>;
  auto it = data.begin();
  for (; it <= data.end() - floatv::size();
         it += floatv::size()) {
    std::sin(floatv(it)).copy_to(it);
  }
  for (; it < data.end(); ++it) {
    *it = std::sin(*it);
  }
}
```

- 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

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

- 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.

Subscripting

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

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

- 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.

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

- 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.

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

- 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. 😉