std::simd:

simd: How to Express Inherent Parallelism Efficiently Via Data-parallel Types

MATTHIAS KRETZ

std::simd

how to express inherent parallelism efficiently via data-parallel types

Dr. Matthias Kretz

GSI Helmholtz Center for Heavy Ion Research

CppCon '23

@mkretz@floss.social

github.com/mattkretz

Goals and non-goals for this talk

- This is *not a tutorial*!

  You won't really know how to use the `std::simd` API after this talk.
- I will have to scratch the surface.

  Most examples/topics have many interesting tangents to take; we don't have that time.
- My goal is to share my vision.

  Take your view from SIMD registers up to designing efficient software.
- How to think / design…
- I might have promised too much content in the talk's abstract.

  Sorry.

Motivation

`std::simd` is for you!

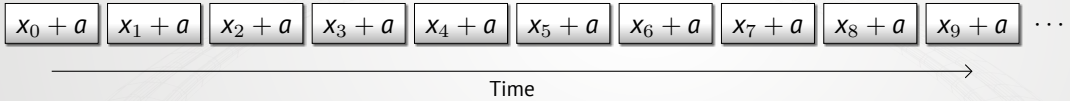
`std::simd` is for you!

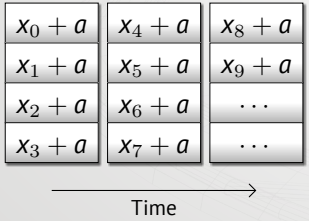
… unless you don't care for speed-up measured in factors not percentages

SIMD — Single Instruction Multiple Data

in other words...

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

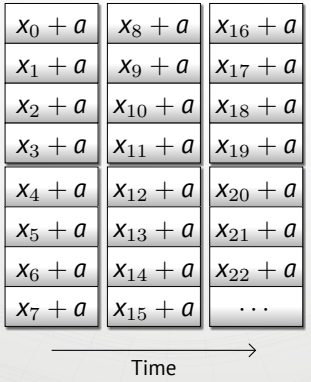




ILP — Instruction Level Paralellism

in other words...

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



Take-Away #1

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

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)

```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);
}
```

	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)

```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)

```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);
}
```

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!

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 language support.

🤔

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!


std::simd Overview

Data-Parallel Types

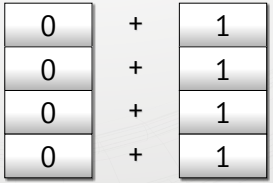
One variable stores $\mathcal{W}_{\mathtt{T}}$ values.
One operator signifies $\mathcal{W}_{\mathtt{T}}$ operations (element-wise).

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

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

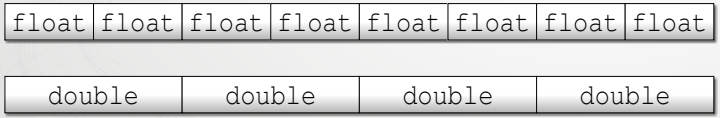
vs.

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



std::simd::size

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



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

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` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `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: `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` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `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: `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), ...>;
```

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

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

- `simd<T>` behaves just like `T` (element-wise)
- `T` must be a *vectorizable type* — all arithmetic types except `bool` or `long double`
- `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: `simd<int, 8>` on x86 can be two `xmm` registers (`alignof == 16`) or one `ymm` register (`alignof == 32`).

Constructor examples

```
std::simd<int> x0;
```


```
std::simd<int> x1{};
```


```
std::simd<int> x2 = 1;
```


```
std::simd<int> x3(it);
```


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

Constructor examples

```
std::simd<int> x0;
```


```
std::simd<int> x1{};
```


```
std::simd<int> x2 = 1;
```


```
std::simd<int> x3(it);
```

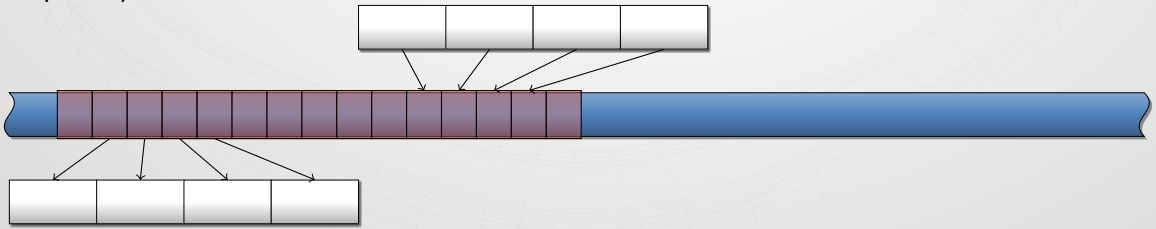

```
std::simd<int> iota([](int i) { return i; });
```

Loads & stores

```
1  class simd {
2    simd(contiguous_iterator auto it);
3
4    void
5    copy_from(contiguous_iterator auto it);
6
7    void
8    copy_to(contiguous_iterator auto it);
9  };
```

(simplified)

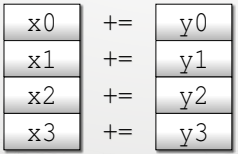
- Loads copy `simd<T>::size()` elements from a contiguous range into the `simd<T>` elements.
- Stores do the reverse.
- Consider loads & stores equivalent to dereferencing an iterator in the scalar case.



Arithmetic & math

```
1  void f(std::simd<float> x,
2         std::simd<float> y) {
3    x += y;       // x.size() additions
4    x = sqrt(x); // x.size() square roots
5    ...
   // etc. all operators and <cmath>
6  }
```

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



Same for compares (element-wise)

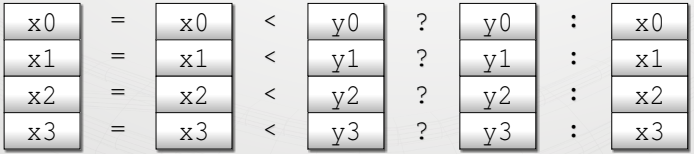
```
1 void f(std::simd<float> x, std::simd<float> y) {
2   if (x < y) { x = y; } // nonono, you don't write 'if (truefalsetruetrue)' either
3   x = simd_select(x < y, y, x); // x = y but only for the elements where x < y
4   if (all_of(x < y)) {...}      // this makes sense, yes
5 }
```

- Comparisons return `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)

```
1  void f(std::simd<float> x, std::simd<float> y) {
2    if (x < y) { 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 `simd_mask`.
- `simd_mask` is not convertible to `bool`.
- `simd_mask` can be *reduced* to `bool` via `all_of`, `any_of`, or `none_of`.



Permutations

- Permutations enable reductions and vectorization of some loops with dependent interations.
- Permutation API paper (authored by Intel) progressing in Library Evolution (LEWG).
- That same paper also explores how to add gather & scatter primitives.

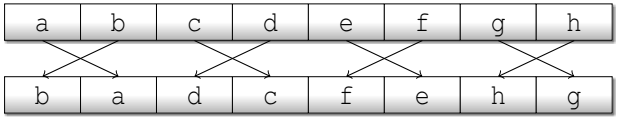
  like loads & stores doing random access rather than contiguous access

Example

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

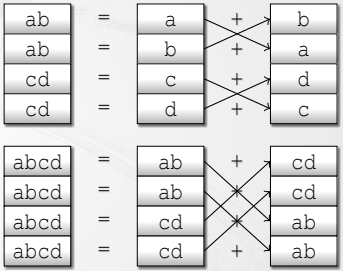
with AVX, compiles to:

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



Reductions

- `simd_mask` **reductions:**
  `all_of`, `any_of`, `none_of`, `reduce_count`, `reduce_min_index`, `reduce_max_index`
- `simd` **reductions:**
  `reduce`, `reduce_min`, `reduce_max`



Example

```
1  void 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    int min_idx = std::reduce_min_index(x == minimum);
7  }
```

Example: Image Processing

Example: image processing

- Let's explore usage of `std::simd` in image processing.
- Task "paint it gray":
  - Given a 32-bit ARGB image (`0xAARRGGBB`)
  - Compute gray as `(11 * red + 16 * green + 5 * blue) / 32`
  - Store the gray value back to the RGB components, leaving the alpha channel unchanged.
- We start with a `struct Pixel { std::uint8_t b, g, r, a; };` (little-endian),
- combined to an `using Image = std::vector<Pixel>;`.

Task

Write a function `void to_gray(Image& img) { ... }` that modifies the image in place.

to_gray — scalar solution

```
1  void to_gray(Image& img) {
2    for (Pixel& pixel : img) {
3      const auto gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32;
4      pixel.r = pixel.g = pixel.b = gray;
5    }
6  }
```

or

```
1  void to_gray(Image& img) {
2    std::for_each(execution::unseq, img.begin(), img.end(), [](Pixel& pixel) {
3      const auto gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32;
4      pixel.r = pixel.g = pixel.b = gray;
5    });
6  }
```

to_gray — scalar solution

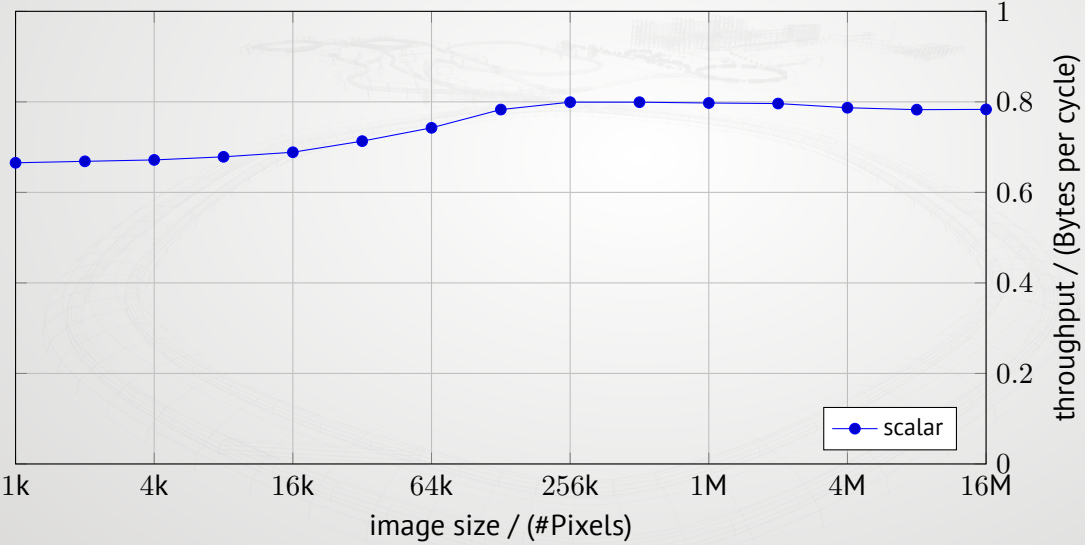
```
1  void to_gray(Image& img) {
2    for (Pixel& pixel : img) {
3      const auto gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32;
4      pixel.r = pixel.g = pixel.b = gray;
5    }
6  }
```

or

```
1  void to_gray(Image& img) {
2    std::for_each (std::execution::unseq, img.begin(), img.end(), [](Pixel& pixel) {
3      const auto gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32;
4      pixel.r = pixel.g = pixel.b = gray;
5    });
6  }
```

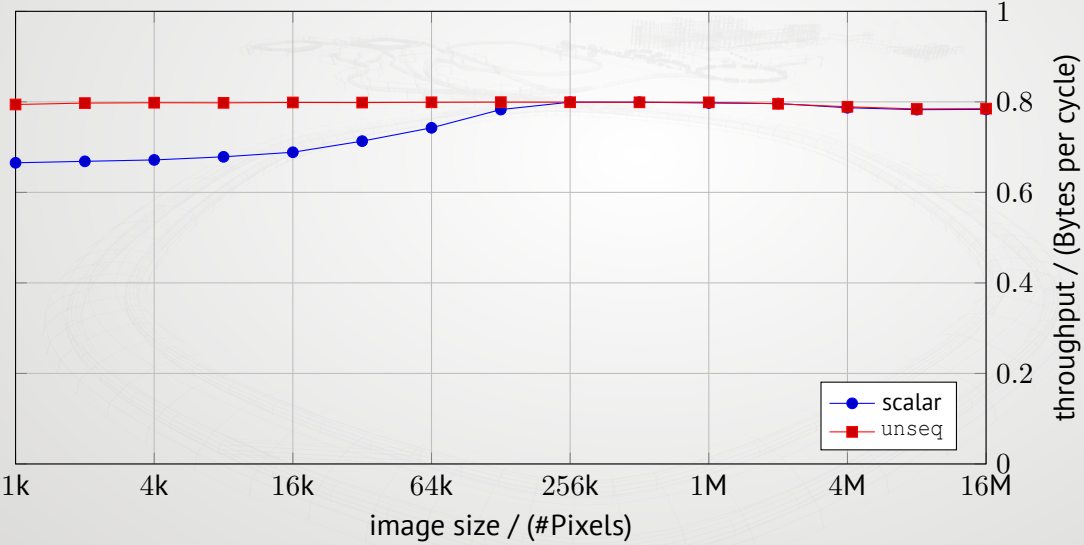
`to_gray` — scalar solution

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



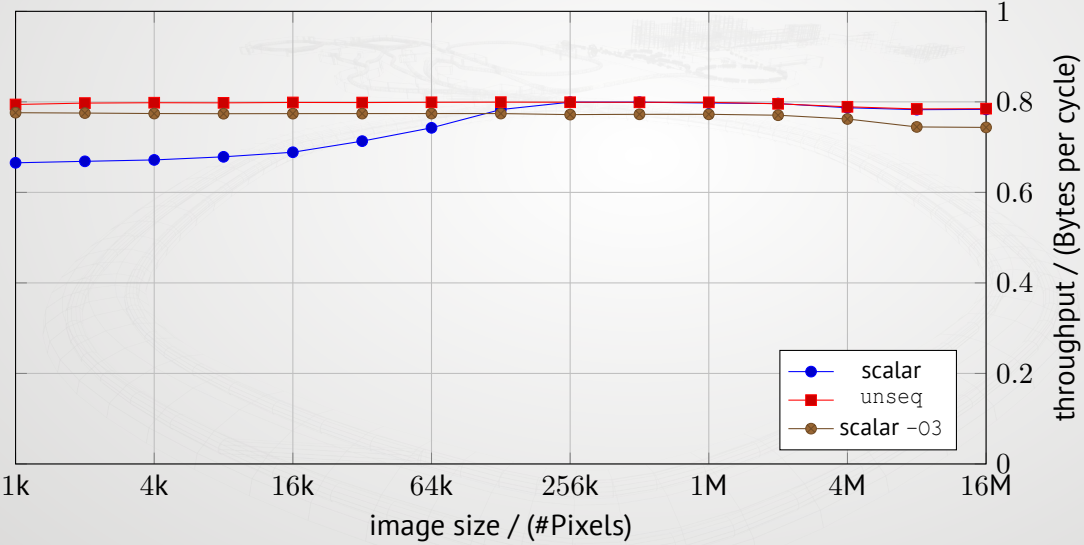
`to_gray` — scalar solution

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



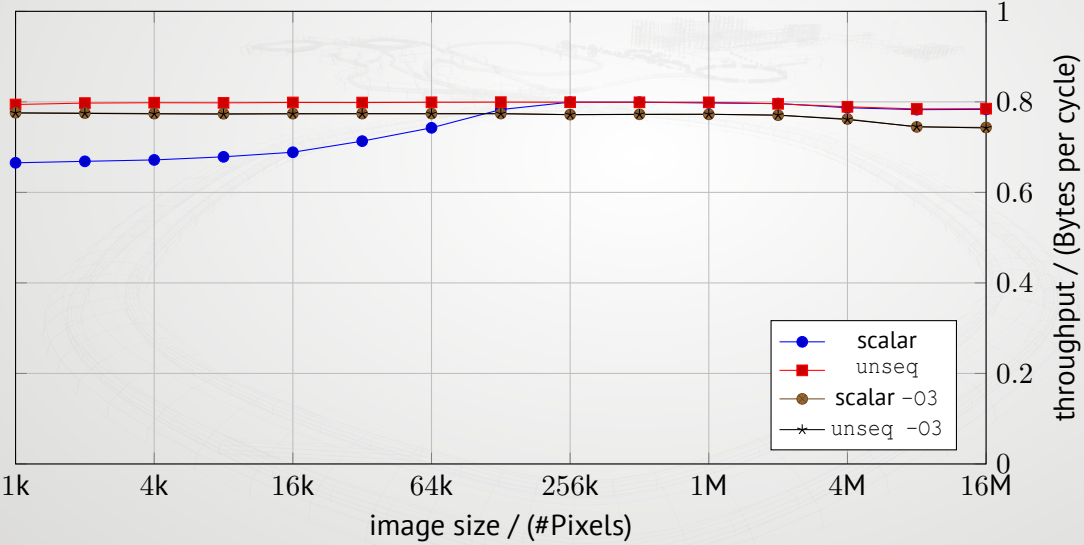
`to_gray` — scalar solution

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



`to_gray` — scalar solution

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



`to_gray` — How to vectorize?

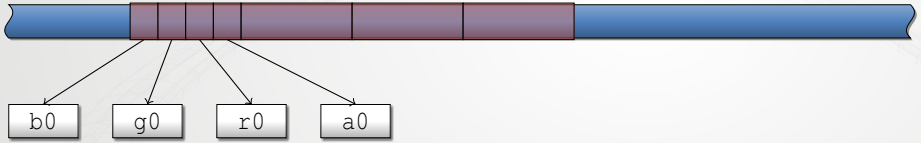
A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…

`to_gray` — How to vectorize?

A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…

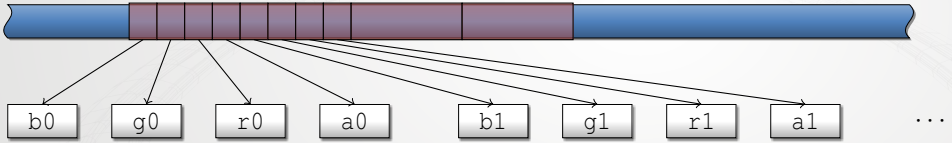
`to_gray` — How to vectorize?

A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…



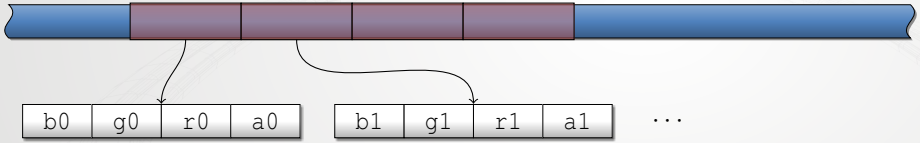
`to_gray` — How to vectorize?

A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…



`to_gray` — How to vectorize?

A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…

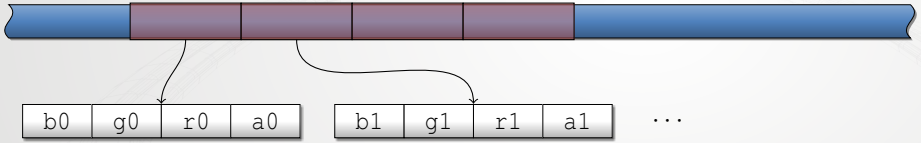



```
using Pixel = std::simd<std::uint8_t, 4>;
using Image = std::vector<Pixel>;
```

```
std::reduce(  b  *  5
              g  *  16
              r  *  11
              a  *  0  ) / 32
```

`to_gray` — How to vectorize?

A typical request when using SIMD is to keep it “localized” / non-invasive. The goal is to divide the code into expert code (SIMD) and average coder / business logic. Let’s try that…




```
using Pixel = std::simd<std::uint8_t, 4>;
using Image = std::vector<Pixel>;
```

```
std::reduce( b * 5
             g * 16
             r * 11
             a * 0 ) / 32
```

to_gray — non-invasive SIMD solution

```
1  using Pixel = std::simd<std::uint8_t, 4>;
2  using Image = std::vector<Pixel>;
3  using Pixel32 = std::simd<std::uint32_t, 4>;
4  using Pixel32Mask = Pixel32::mask_type;
5
6  constexpr std::uint32_t gray_coeff[4] = {5, 16, 11, 0};
7
8  void to_gray(Image& img) {
9    constexpr Pixel32Mask mask([](auto i) { return i < 3; });
10   for (Pixel& p8 : img) {
11     const Pixel32 pixel = p8;
12     const std::uint32_t gray = std::reduce(pixel * Pixel32(gray_coeff)) / 32u;
13     p8 = static_cast<Pixel>(std::simd_select(mask, gray, pixel));
14   }
15 }
```

to_gray — non-invasive SIMD solution

```
1  using Pixel = std::simd<std::uint8_t, 4>;
2  using Image = std::vector<Pixel>;
3  using Pixel32 = std::simd<std::uint32_t, 4>;
4  using Pixel32Mask = Pixel32::mask_type;
5
6  constexpr std::uint32_t gray_coeff[4] = {5, 16, 11, 0};
7
8  void to_gray(Image& img) {
9    constexpr Pixel32Mask mask([](auto i) { return i < 3; });
10   for (Pixel& p8 : img) {
11     const Pixel32 pixel = p8;
12     const std::uint32_t gray = std::reduce(pixel * Pixel32(gray_coeff)) / 32u;
13     p8 = static_cast<Pixel>(std::simd_select(mask, gray, pixel));
14   }
15 }
```

to_gray — non-invasive SIMD solution

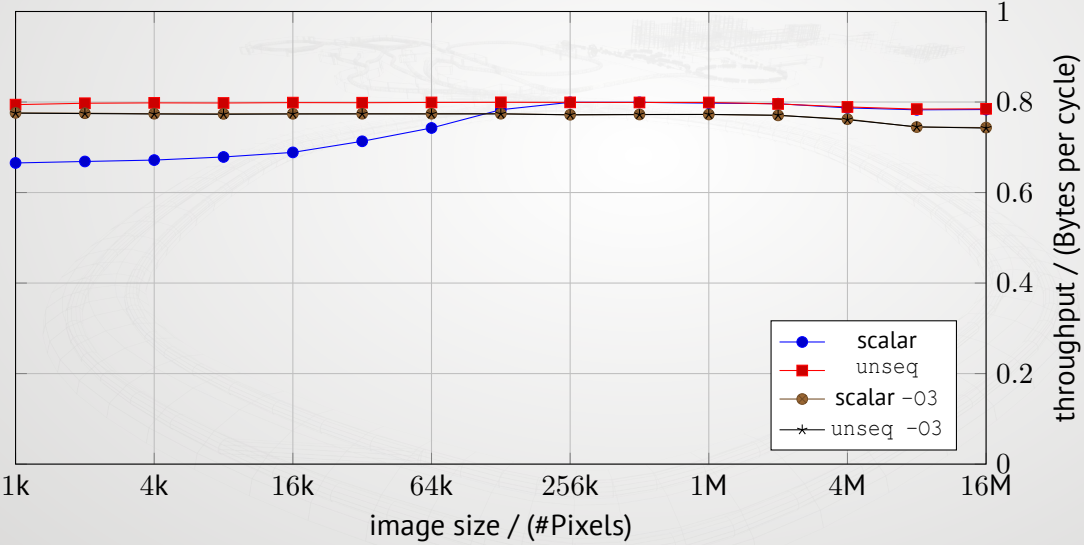
```
1  using Pixel = std::simd<std::uint8_t, 4>;
2  using Image = std::vector<Pixel>;
3  using Pixel32 = std::simd<std::uint32_t, 4>;
4  using Pixel32Mask = Pixel32::mask_type;
5
6  constexpr std::uint32_t gray_coeff[4] = {5, 16, 11, 0};
7
8  void to_gray(Image& img) {
9    constexpr Pixel32Mask mask([](auto i) { return i < 3; });
10   for (Pixel& p8 : img) {
11     const Pixel32 pixel = p8;
12     const std::uint32_t gray = std::reduce(pixel * Pixel32(gray_coeff)) / 32u;
13     p8 = static_cast<Pixel>(std::simd_select(mask, gray, pixel));
14   }
15 }
```

to_gray — non-invasive SIMD solution

```
1  using Pixel = std::simd<std::uint8_t, 4>;
2  using Image = std::vector<Pixel>;
3  using Pixel32 = std::simd<std::uint32_t, 4>;
4  using Pixel32Mask = Pixel32::mask_type;
5
6  constexpr std::uint32_t gray_coeff[4] = {5, 16, 11, 0};
7
8  void to_gray(Image& img) {
9    constexpr Pixel32Mask mask([](auto i) { return i < 3; });
10   for (Pixel& p8 : img) {
11     const Pixel32 pixel = p8;
12     const std::uint32_t gray = std::reduce(pixel * Pixel32(gray_coeff)) / 32u;
13     p8 = static_cast<Pixel>(std::simd_select(mask, gray, pixel));
14   }
15 }
```

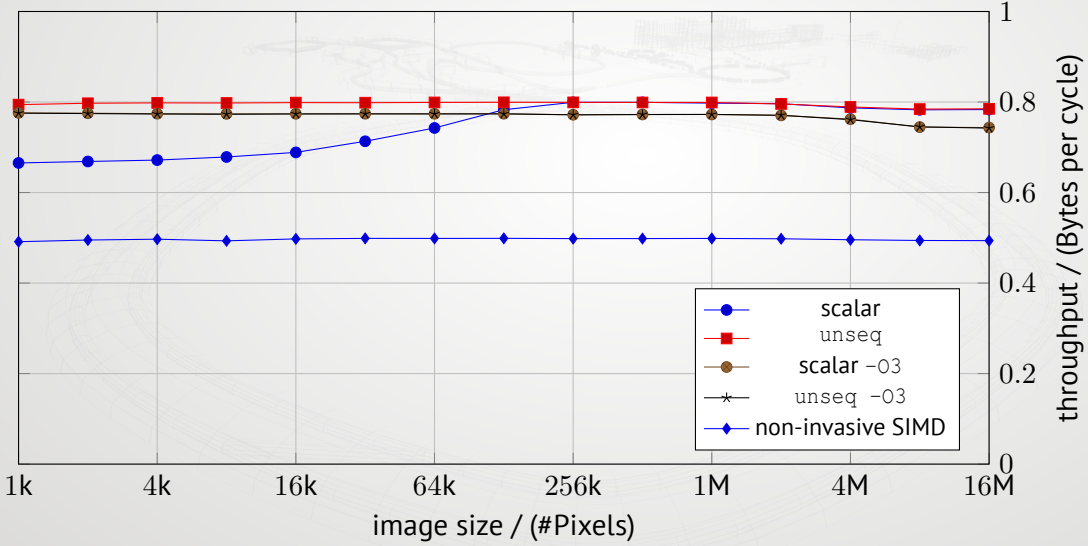
`to_gray` — non-invasive SIMD solution

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



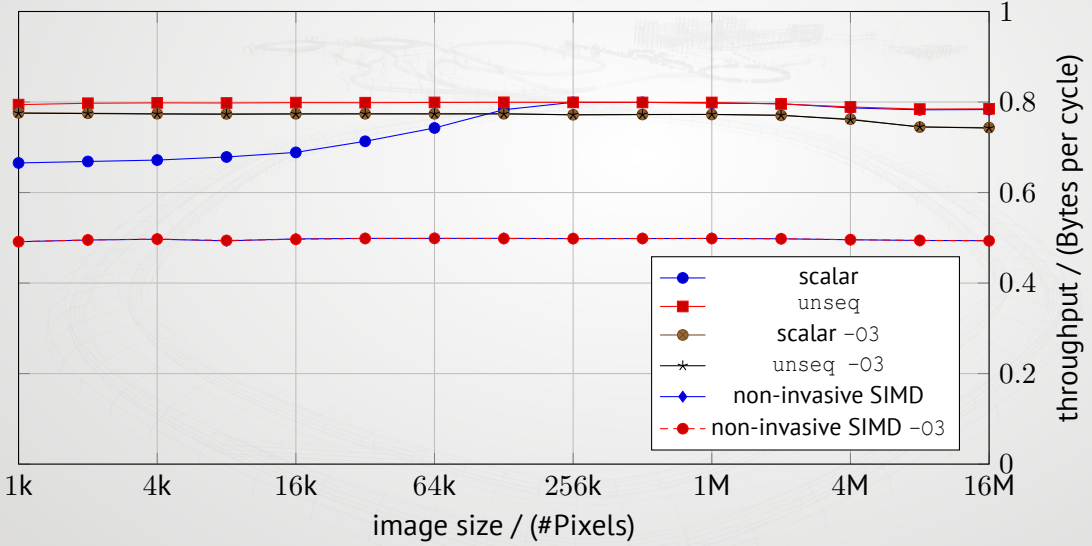
`to_gray` — non-invasive SIMD solution

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



`to_gray` — non-invasive SIMD solution

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



😱 SIMD made our program slower!?

😱 SIMD made our program slower!?

Take-Away #3

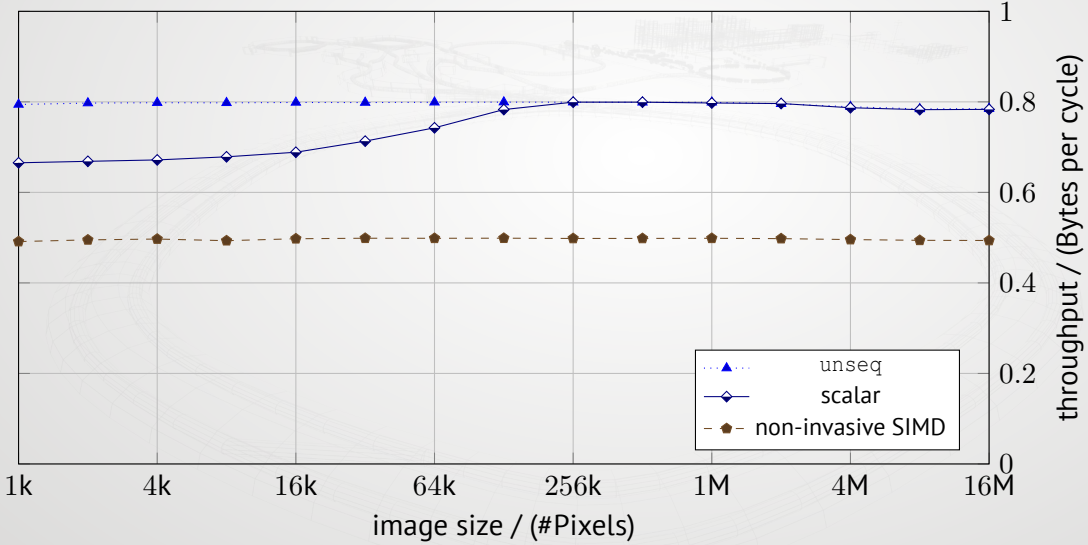
🏆 focus on data-parallelism not SIMD instructions/registers!

consider SIMD an implementation-detail of data-parallelism

We failed to express the “interesting” parallelism.
Instead we expressed only 4 independent values and had to reduce to a scalar after a single SIMD op.

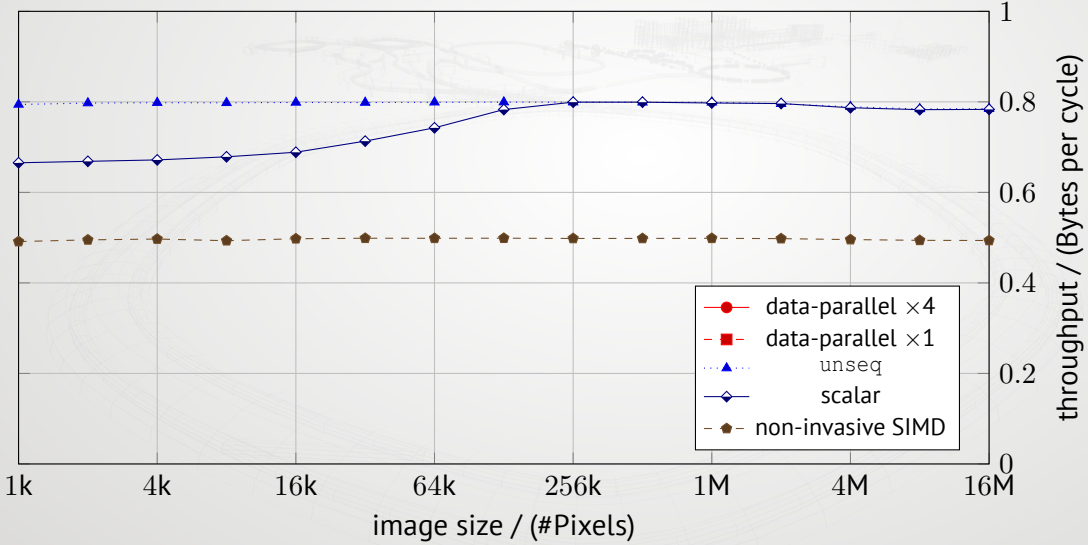
`to_gray` — data-parallel solution

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



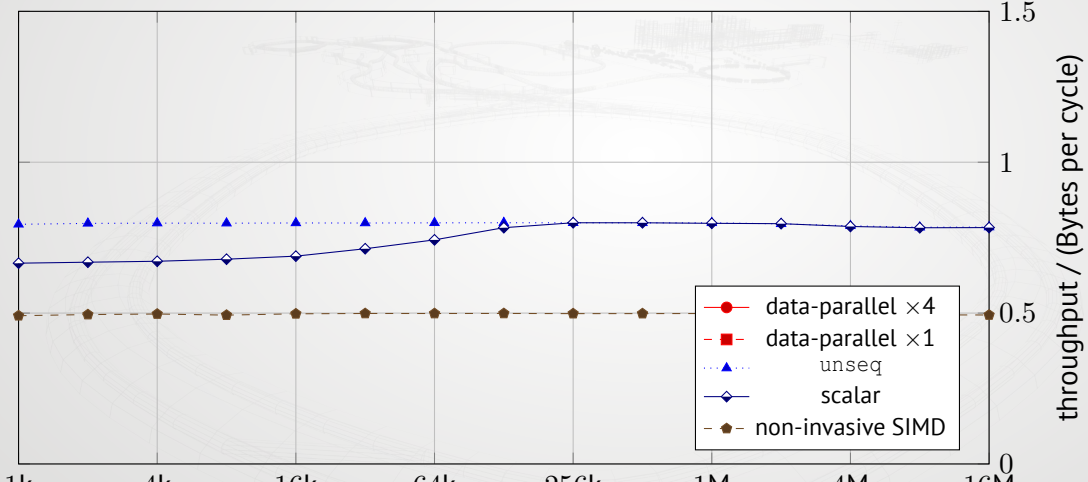
to_gray — data-parallel solution

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



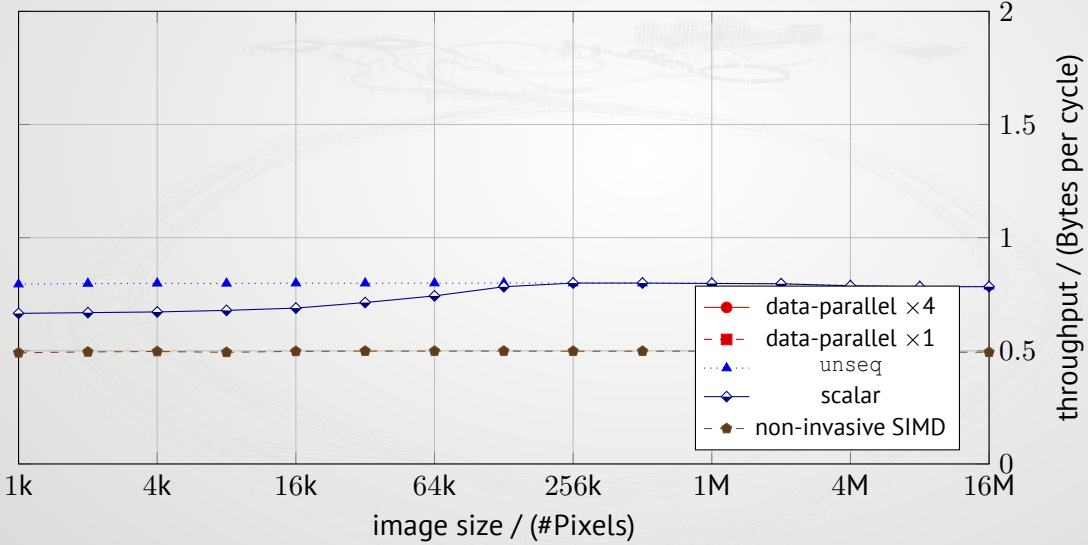
`to_gray` — data-parallel solution

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



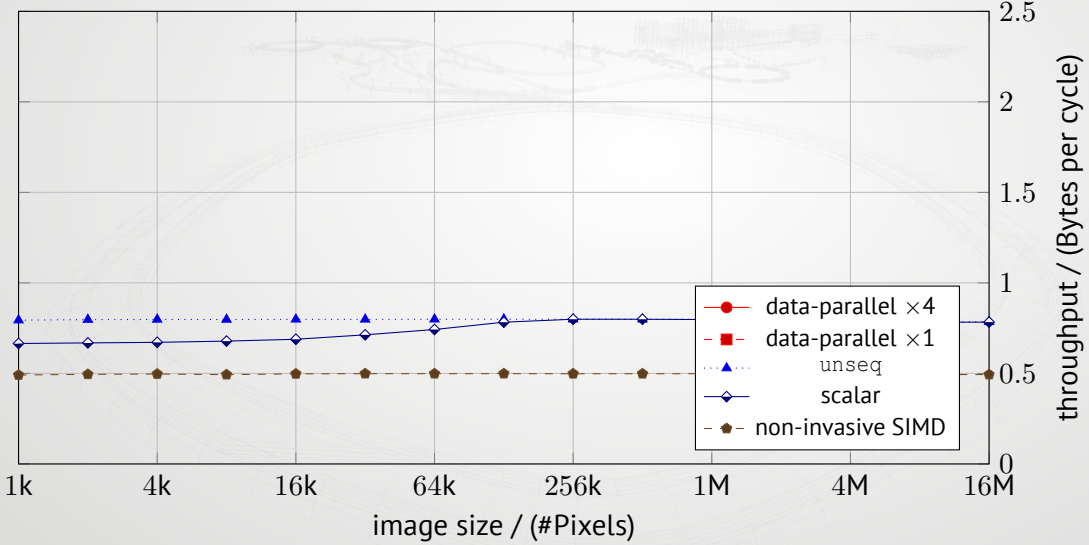
`to_gray` — data-parallel solution

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



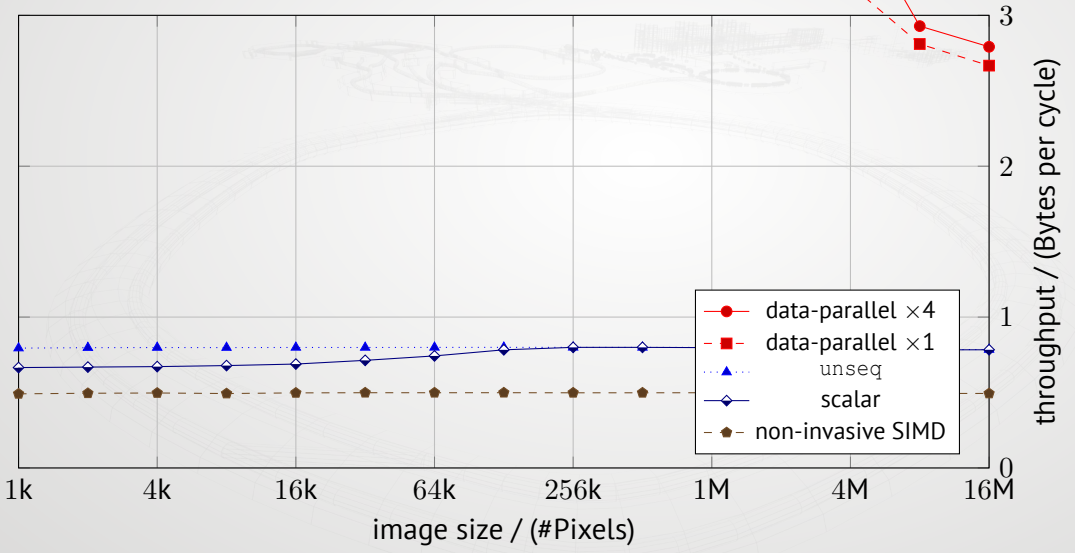
`to_gray` — data-parallel solution

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



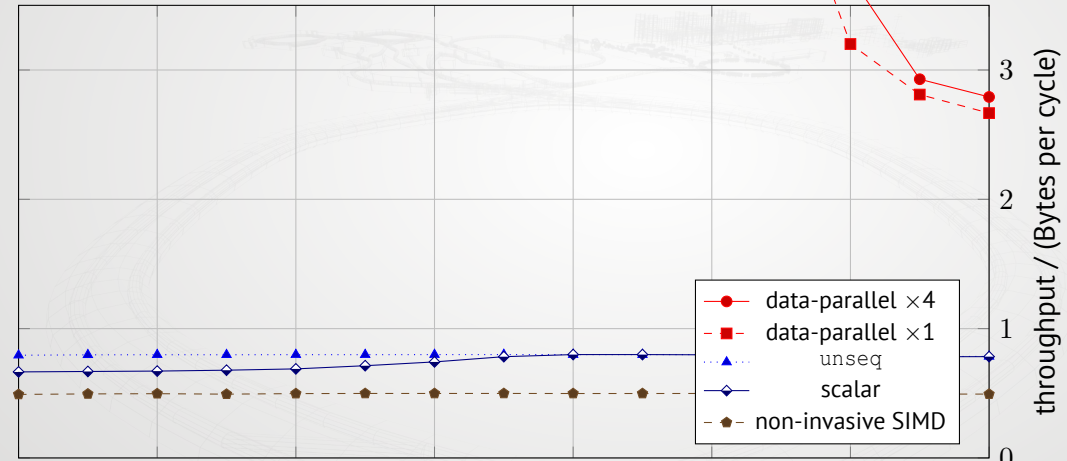
`to_gray` — data-parallel solution

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



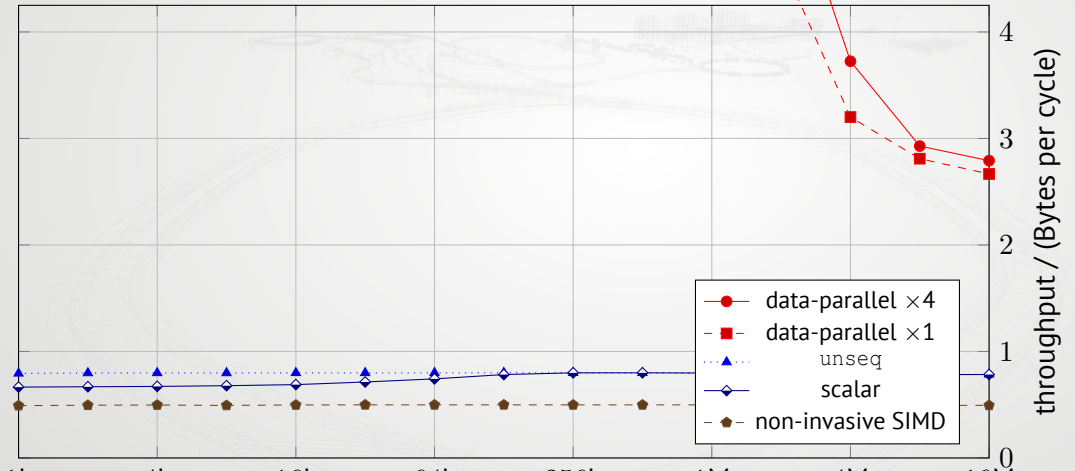
`to_gray` — data-parallel solution

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



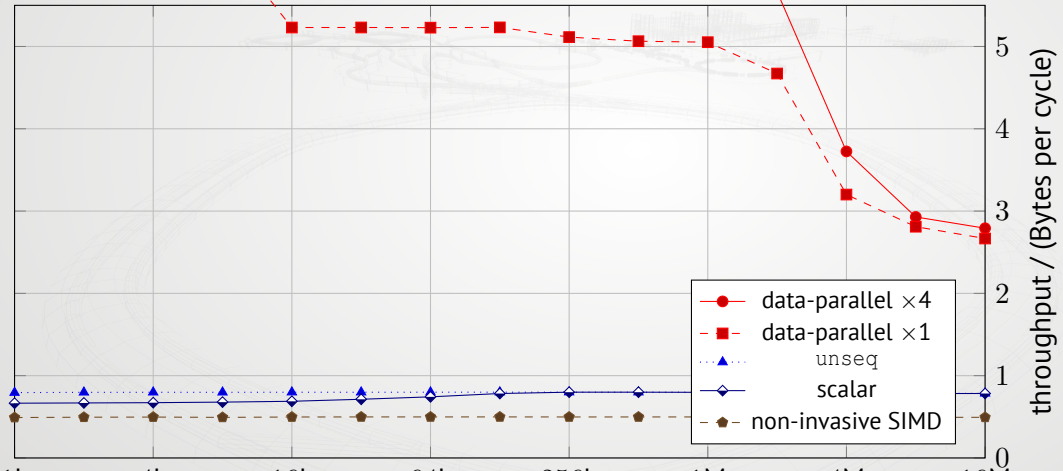
`to_gray` — data-parallel solution

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



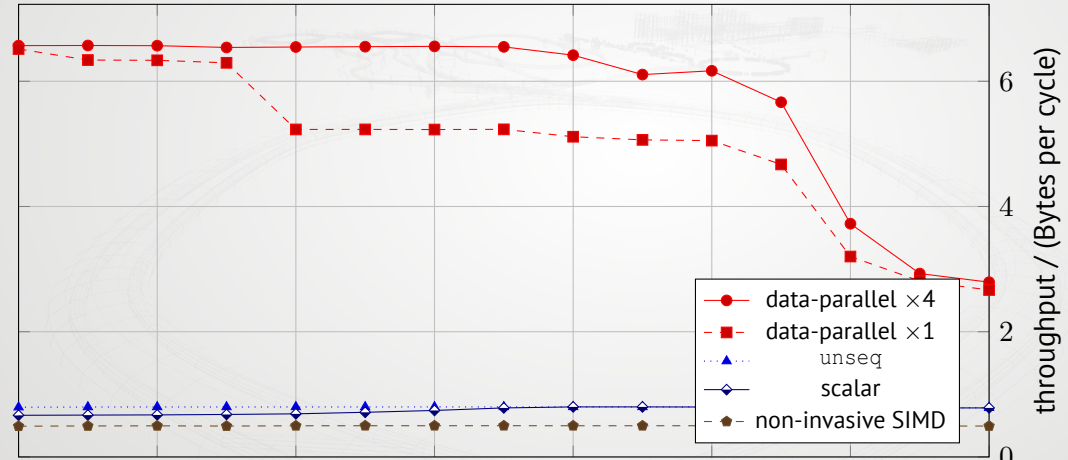
to_gray — data-parallel solution

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



`to_gray` — data-parallel solution

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



How to express data-parallelism?

Idea: Every pixel can be processed independently.
⇒ Process as many pixels in parallel as possible.

Solution: Replace the types in

```
gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32
```

with data-parallel types.

```
1 simd<std::uint32_t> r, g, b;
2 auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
```

Problem: Where do `r`, `g`, and `b` come from?

How to express data-parallelism?

Idea: Every pixel can be processed independently.
⇒ Process as many pixels in parallel as possible.

Solution: Replace the types in `gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32` with data-parallel types.

```
1 std::simd<std::uint32_t> r, g, b;
2 auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
```

Problem: Where do `r`, `g`, and `b` come from?

How to express data-parallelism?

Idea: Every pixel can be processed independently.
⇒ Process as many pixels in parallel as possible.

Solution: Replace the types in
`gray = (pixel.r * 11 + pixel.g * 16 + pixel.b * 5) / 32` with data-parallel types.

```
1 std::simd<std::uint32_t> r, g, b;
2 auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
```

Problem: Where do `r`, `g`, and `b` come from?

to_gray — corresponding scalar solution

```
1  using Pixel = std::uint32_t; // 0xAARRGGBB
2  using Image = std::vector<Pixel>;
3
4
5  for (auto it = img.begin(); it < img.end(); ++it) {
6    Pixel p = *it;
7    const auto a =  p >> 24;
8    const auto r = (p >> 16) & 0xFFu;
9    const auto g = (p >> 8) & 0xFFu;
10   const auto b =  p & 0xFFu;
11   const auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
12   p = gray | (gray << 8) | (gray << 16) | (a << 24);
13   *it = p;
14 }
```

to_gray — corresponding scalar solution

```
1  using Pixel = std::uint32_t; // 0xAARRGGBB
2  using Image = std::vector<Pixel>;
3
4
5  for (auto it = img.begin(); it < img.end(); ++it) {
6    Pixel p = *it;
7    const auto a =  p >> 24;
8    const auto r = (p >> 16) & 0xFFu;
9    const auto g = (p >> 8) & 0xFFu;
10   const auto b =  p & 0xFFu;
11   const auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
12   p = gray | (gray << 8) | (gray << 16) | (a << 24);
13   *it = p;
14 }
```

to_gray — corresponding scalar solution

```
1  using Pixel = std::uint32_t; // 0xAARRGGBB
2  using Image = std::vector<Pixel>;
3
4
5  for (auto it = img.begin(); it < img.end(); ++it) {
6    Pixel p = *it;
7    const auto a =  p >> 24;
8    const auto r =  (p >> 16) & 0xFFu;
9    const auto g =  (p >> 8) & 0xFFu;
10   const auto b =  p & 0xFFu;
11   const auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
12   p = gray | (gray << 8) | (gray << 16) | (a << 24);
13   *it = p;
14 }
```

Now: replace `Pixel` with `simd<Pixel>`, the rest follows

to_gray — data-parallel solution

```
1  using Pixel = std::uint32_t; // 0xAARRGGBB
2  using Image = std::vector<Pixel>;
3  using PixelV = std::simd<Pixel, std::simd<Pixel>::size() * ILP>;
4
5  for (auto it = img.begin(); it < img.end(); it += PixelV::size()) {
6    PixelV p(it); // loads {it[0], it[1], it[2], ...}
7    const auto a =  p >> 24;
8    const auto r = (p >> 16) & 0xFFu;
9    const auto g = (p >> 8) & 0xFFu;
10   const auto b =  p & 0xFFu;
11   const auto gray = (r * 11u + g * 16u + b * 5u) / 32u;
12   p = gray | (gray << 8) | (gray << 16) | (a << 24);
13   p.copy_to(it);
14 }
```

- 🏆 The code doing data-parallel computation of `gray` is nice!
- 🏆 Performance is great!
- 😱 The code for read-/writing color components is awful!

TODO

Break the code in two parts:

1. Iteration over an image, producing vectorized access to pixels
2. Operations on vectorized pixels

- Ideally, the first part is generalized to:
  Iteration over a range, providing vectorized access to its elements.
- A generic solution should go into the standard library.


Programming Models

Abstract

🏆 Conceptually: `std::simd` types express data-parallelism.

🤔 Wrong mindset: `std::simd` types are specific SIMD registers.

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

`simd` types is so much shorter, though

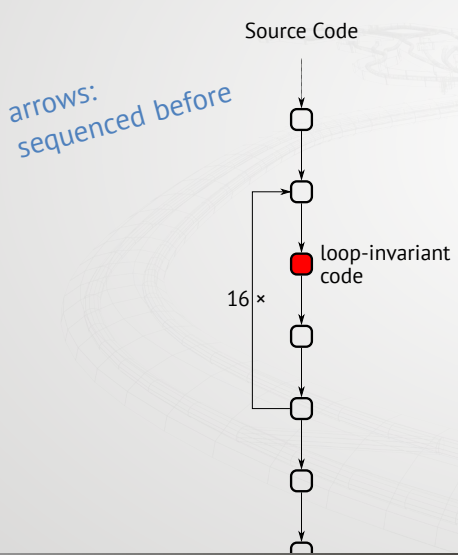
Programming Models

a small (abstract) example

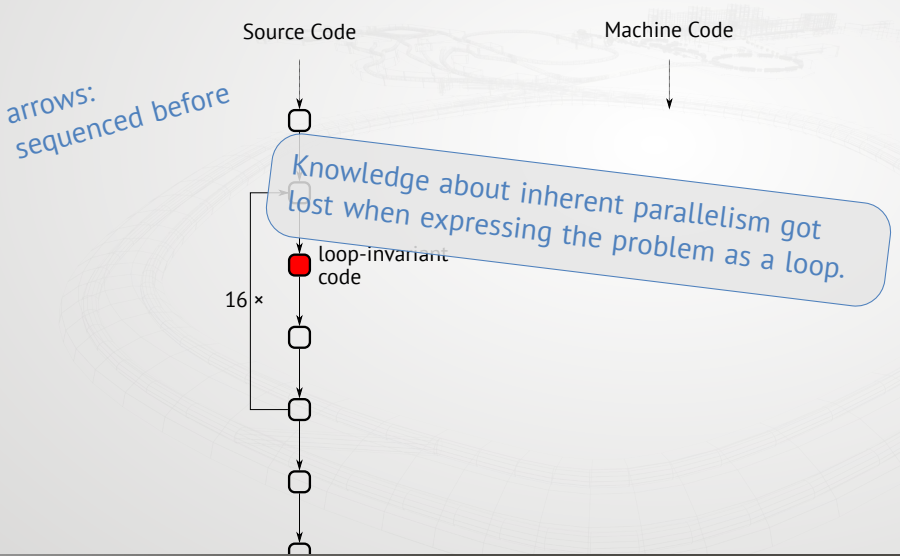
consider a function where 16 different values have to be processed in the same way

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

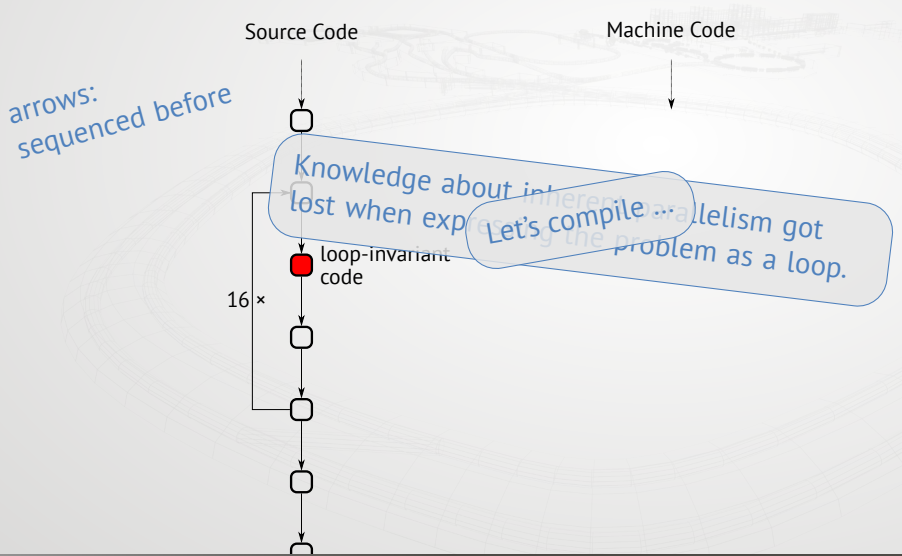
Solution 1: Standard C++



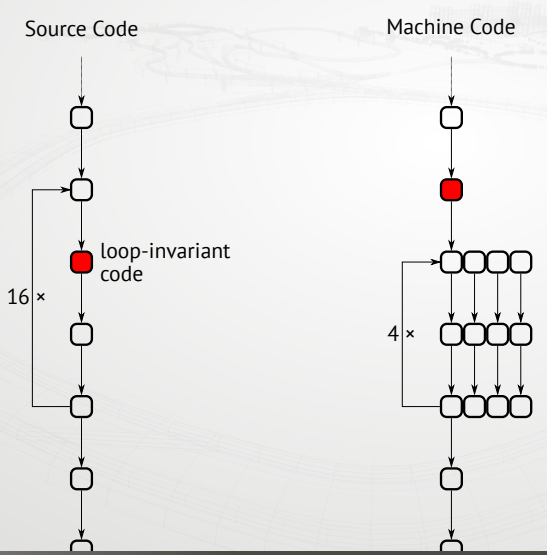
Solution 1: Standard C++



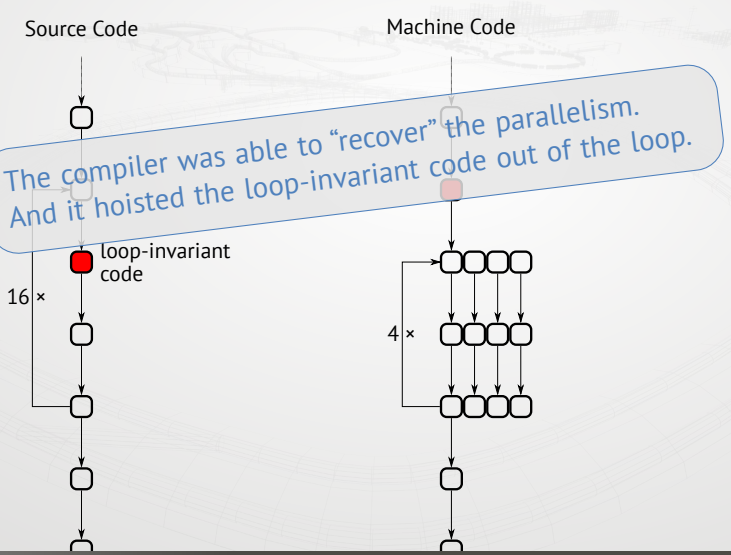
Solution 1: Standard C++



Solution 1: Standard C++



Solution 1: Standard C++

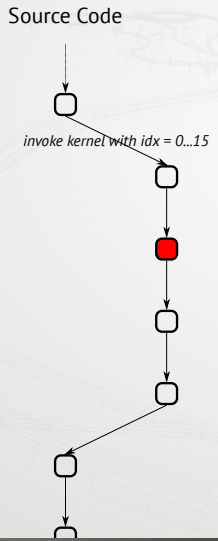


Solution 2: SIMT (simplified)

Single Instruction, Multiple Threads

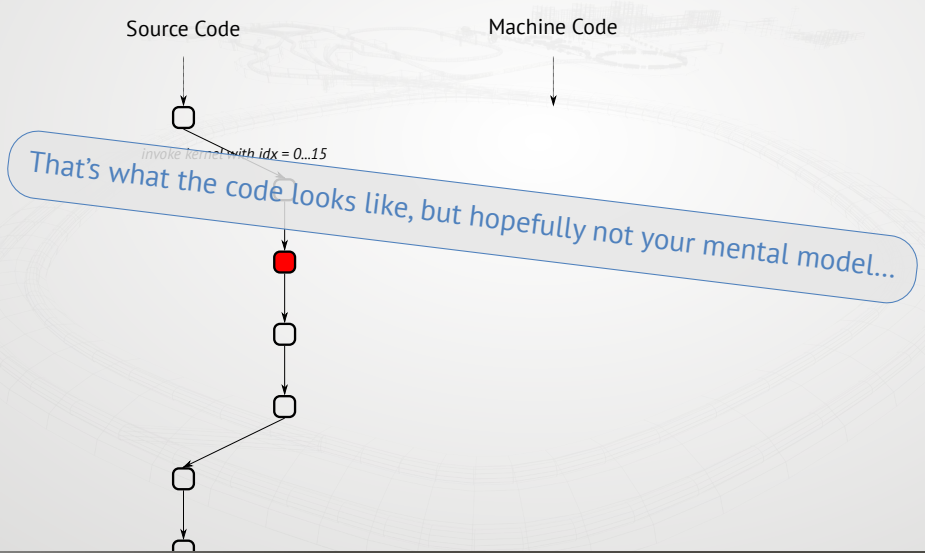
Solution 2: SIMT (simplified)

The code the developer sees



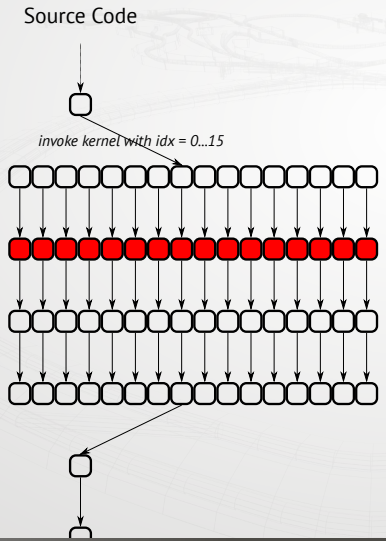
Solution 2: SIMT (simplified)

The code the developer sees



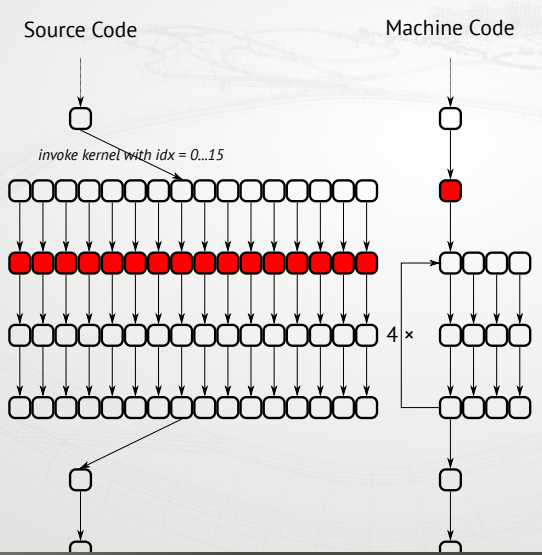
Solution 2: SIMT (simplified)

The code the compiler sees



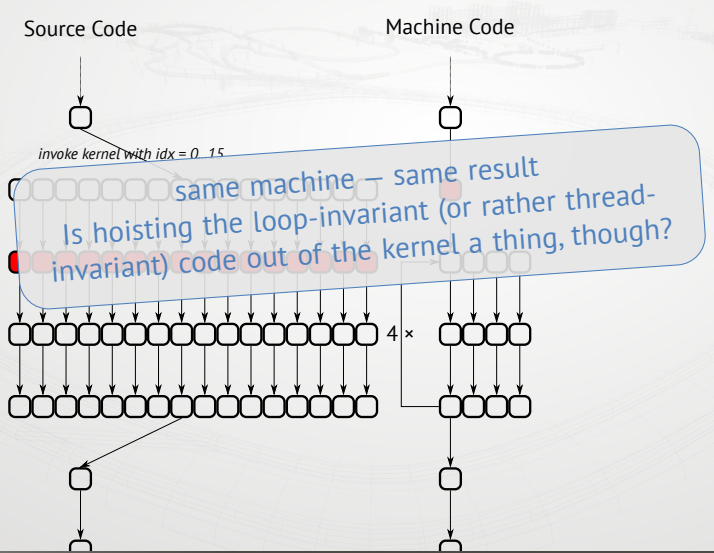
Solution 2: SIMT (simplified)

The code the compiler sees



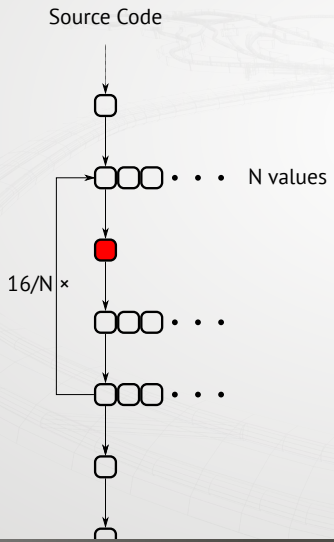
Solution 2: SIMT (simplified)

The code the compiler sees



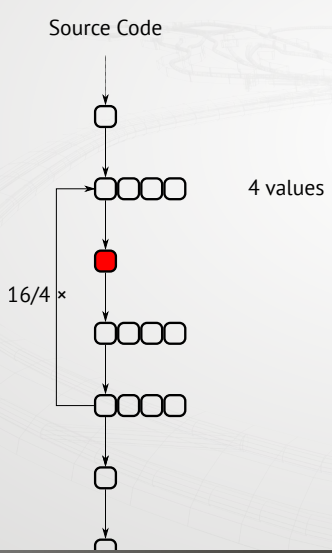
Solution 3: Data-Parallel Types

The code the developer sees



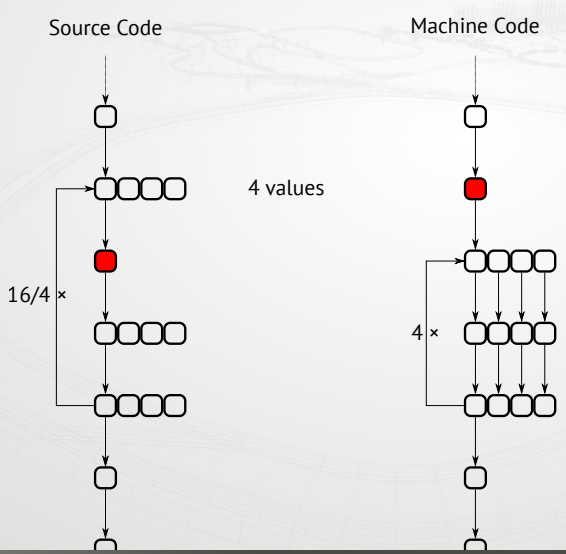
Solution 3: Data-Parallel Types

The code the compiler sees



Solution 3: Data-Parallel Types

The code the compiler sees



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).

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 8>`, `simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`,
`simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

out of bounds!

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`,
`simd<T, 8>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 4>` works

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`,
`simd<T, 8>`, `simd<T, 4>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 2>` is out of bounds

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`,
`simd<T, 8>`, `simd<T, 4>`

Epilogues

🤔 In general, our problems are not "`16 / 4`" — i.e. range size divisible by `simd::size()`.

Example: iterating `range.size() == 45` with `simd<T, 8>`

`simd<T, 1>` works

`simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`, `simd<T, 8>`,
`simd<T, 8>`, `simd<T, 4>`, `simd<T, 1>`

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. HPX also has an implementation.

In Code

trivial data-parallelism

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

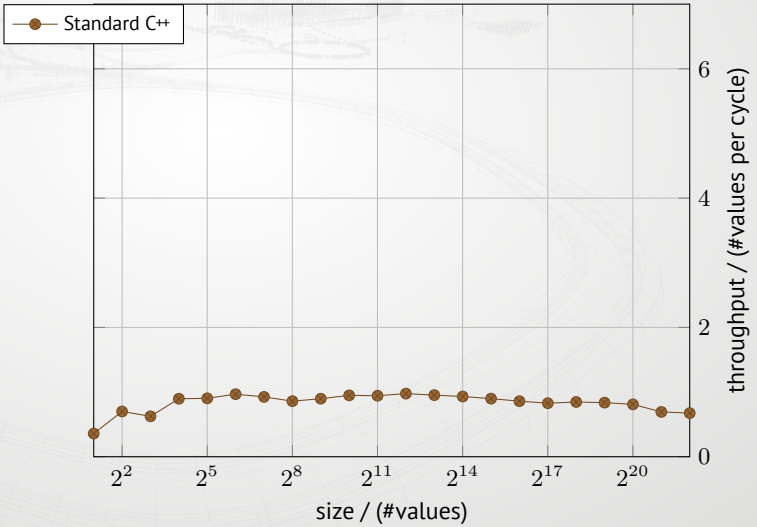
```
1 std::for_each(std::execution::unseq,
2   data.begin(), data.end(),
3   [](int& x) {
4     x += 1;
5   });
```

```
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. HPX also has an implementation.

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

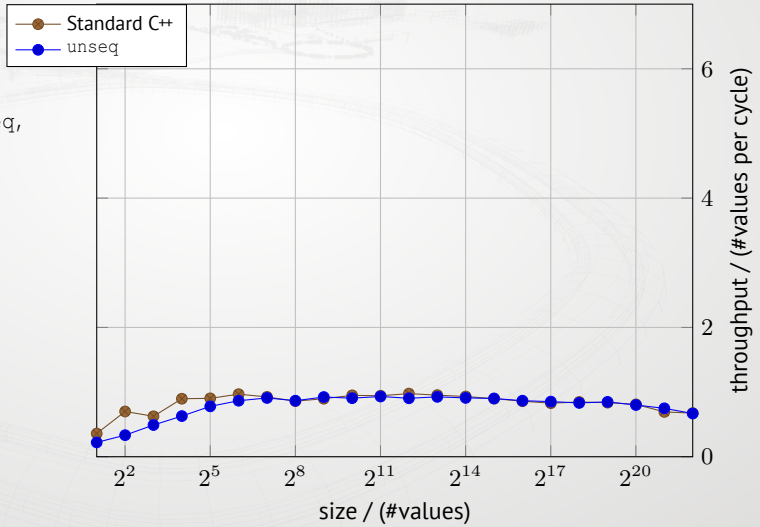
```
1 for (int& x : data) {
2   x += 1;
3 }
```



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; }
);
```

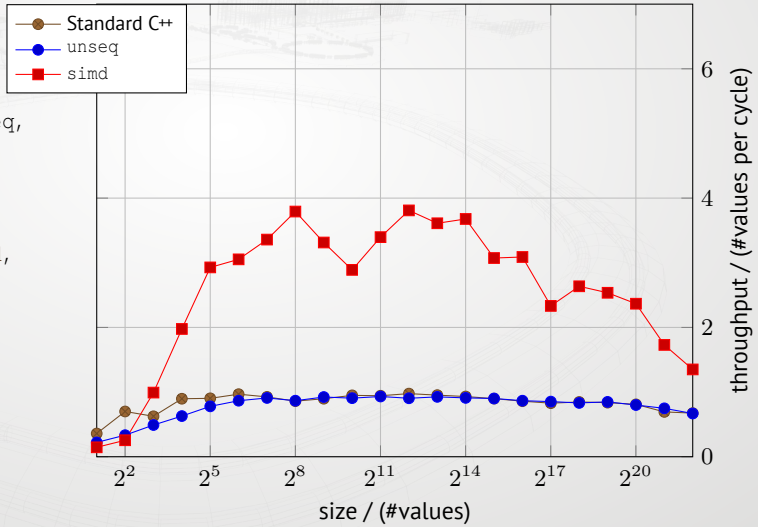


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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

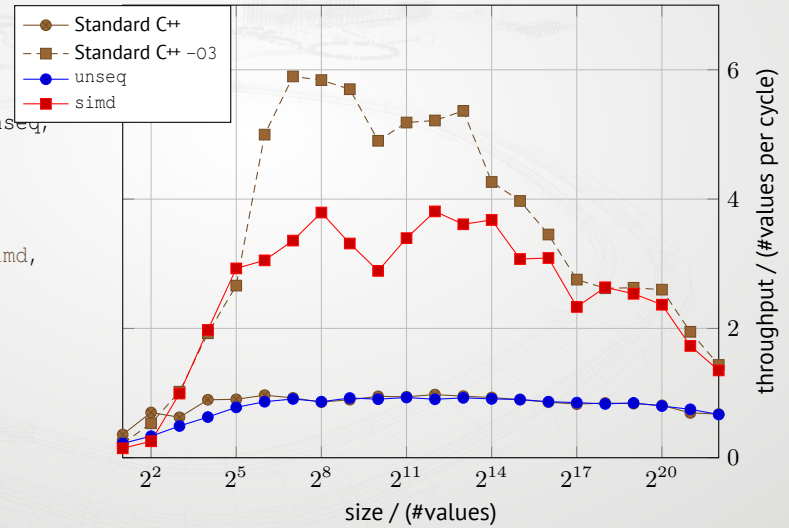


Results (GCC 13, Intel Skylake i7-8550U)

```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; }
);
```

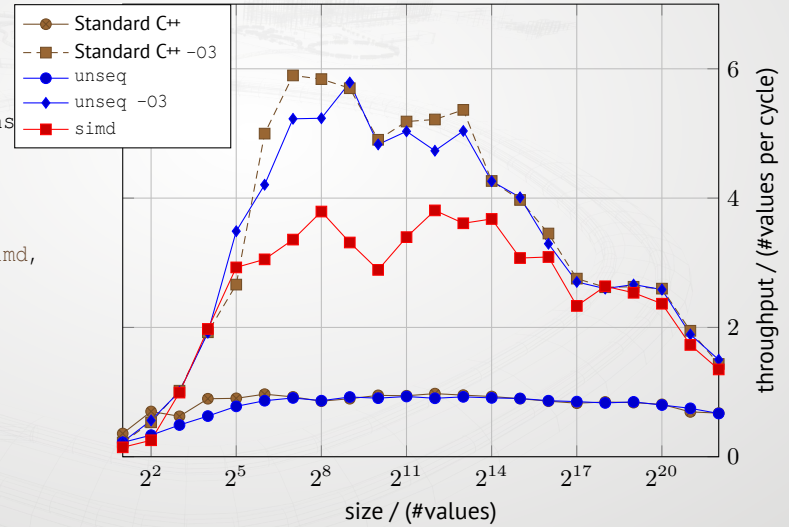


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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

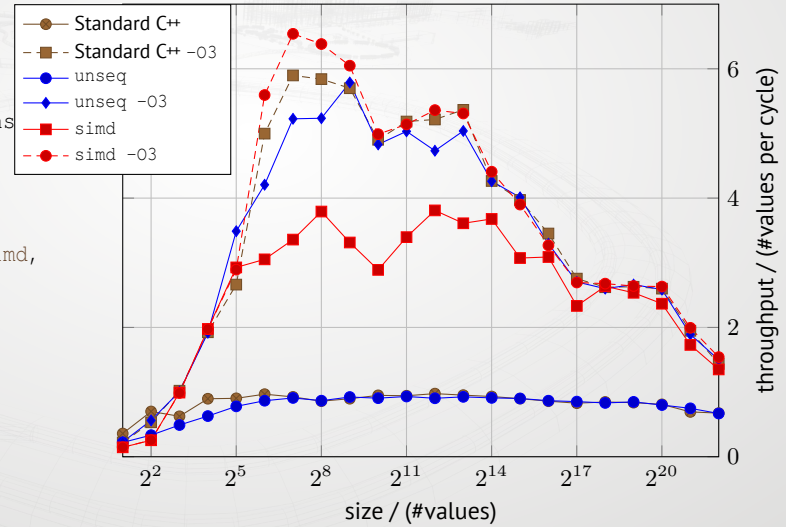


Results (GCC 13, Intel Skylake i7-8550U)

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

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

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



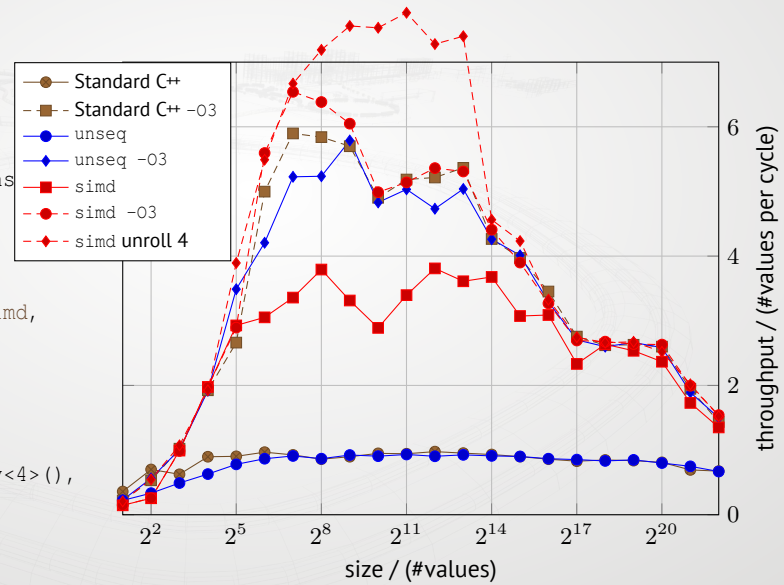
Results (GCC 13, Intel Skylake i7-8550U)

```
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)

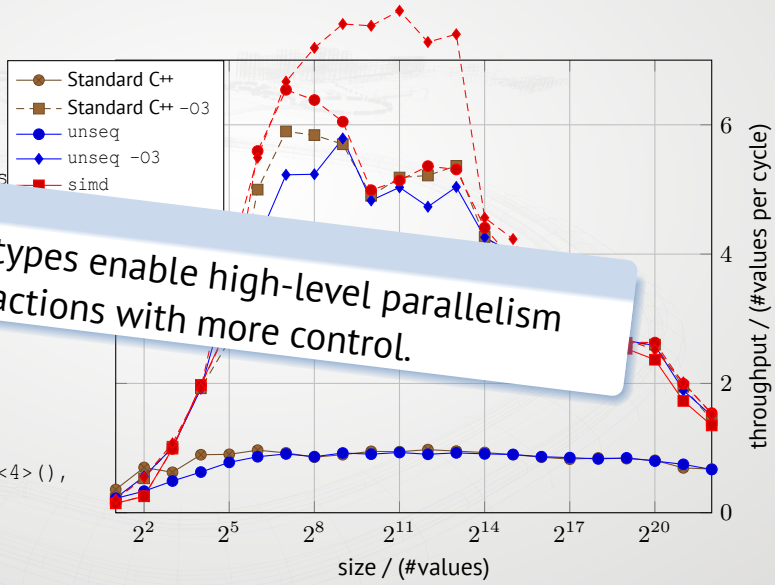
max. store throughput

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

```
std::for_
  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 #4

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

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

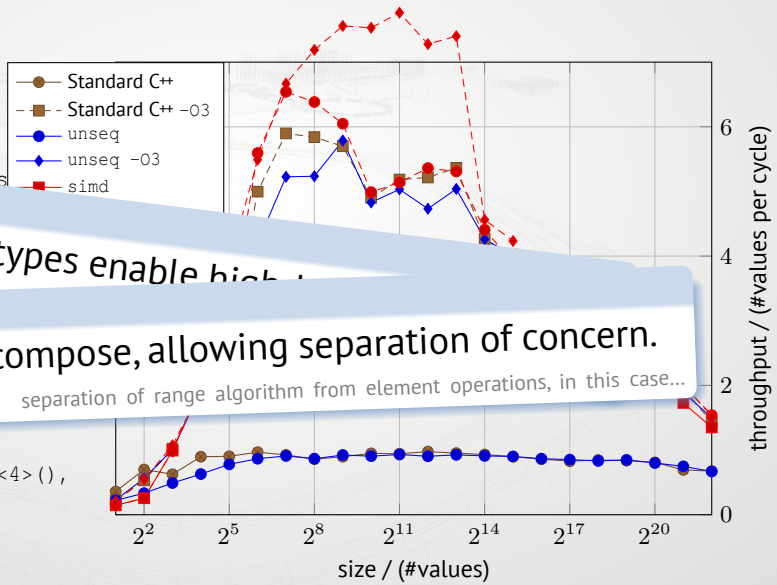
max. store throughput

```
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 #4

Data-parallel types enable high

Take-Away #5

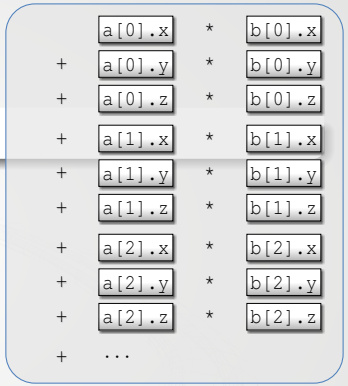
Data-parallel types compose, allowing separation of concern.

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

One more teaser

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

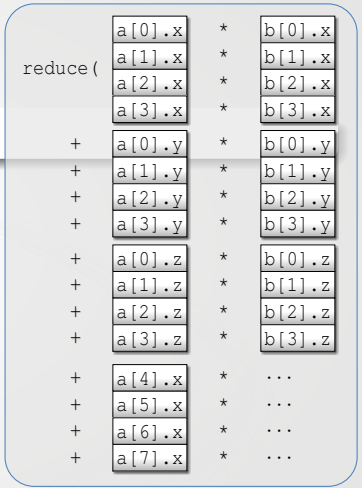
```
1  template <typename T> struct Vec3d
2  {
3    T x, y, z;
4
5    friend constexpr T operator*(Vec3d a, Vec3d b)
6    { return a.x * b.x + a.y * b.y + a.z * b.z; }
7  };
8
9  float inner_product(std::vector<Vec3d<float>> const& a,
10                     std::vector<Vec3d<float>> const& b) {
11  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



One more teaser

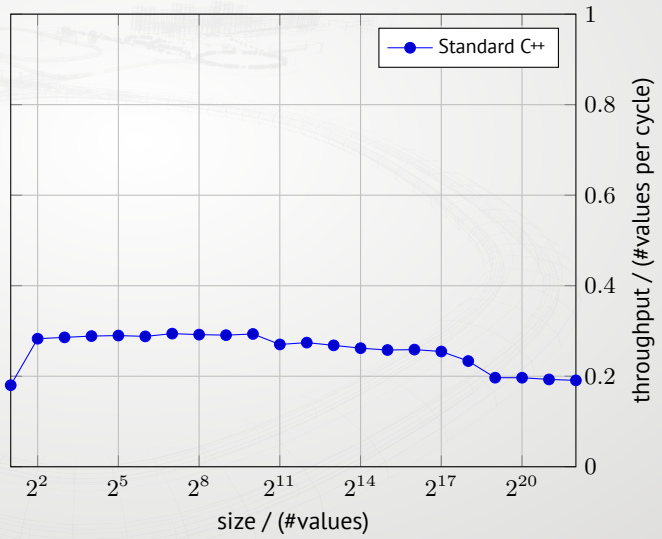
SIMD … when all you have is AoS (Array of Struct)

```
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 }
```



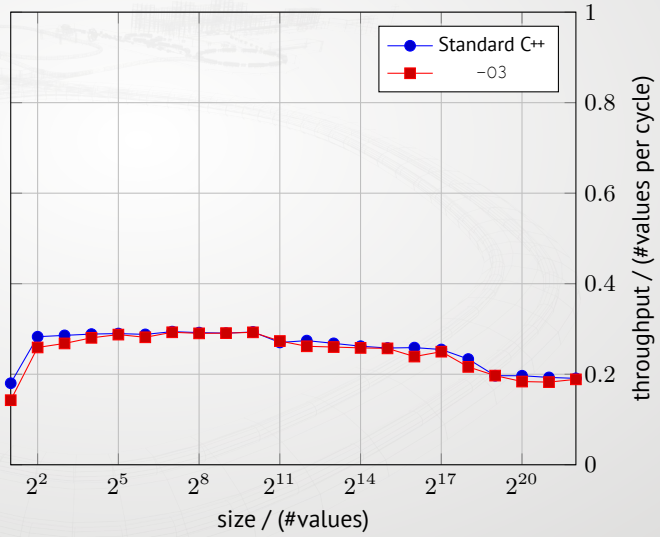
Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy



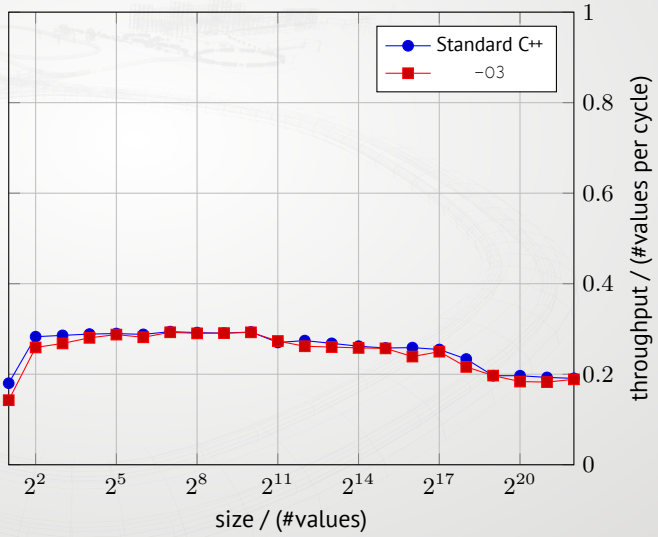
Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize



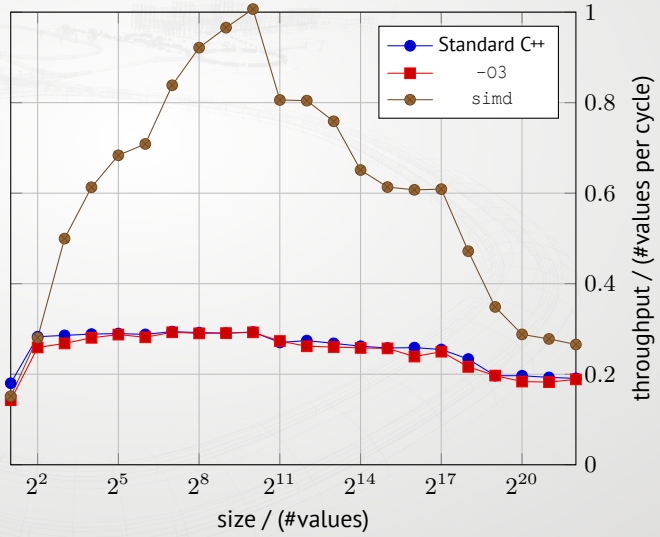
Teaser Results (GCC 13, Intel Skylake i7-8550U)

- 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)
- `vir::execution::simd`: another talk/paper…

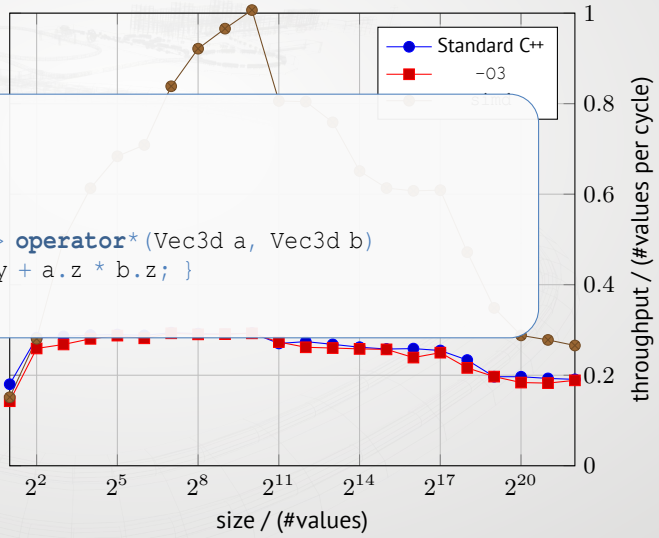


Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- st… co… (would fall back to seq) any…
- vi… tal…

```
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; }
};
```

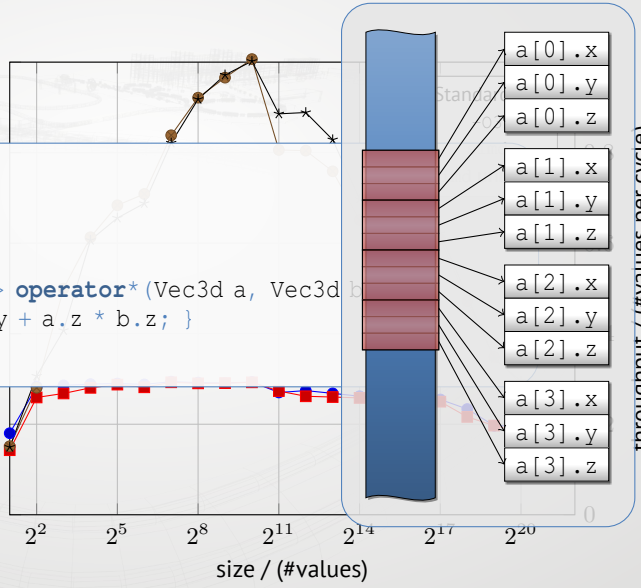


Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- st… co… any…
- vi… tal…

```
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; }
};
```

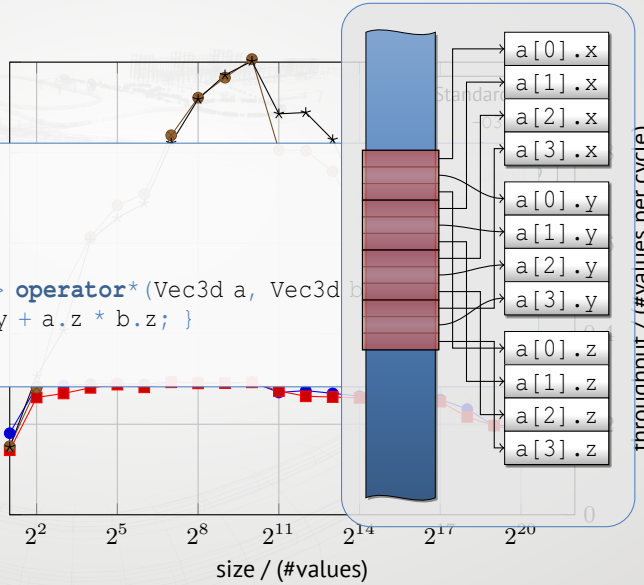


Teaser Results (GCC 13, Intel Skylake i7-8550U)

- no execution policy
- pretty please auto-vectorize
- st… co… an…
- vi… tal…

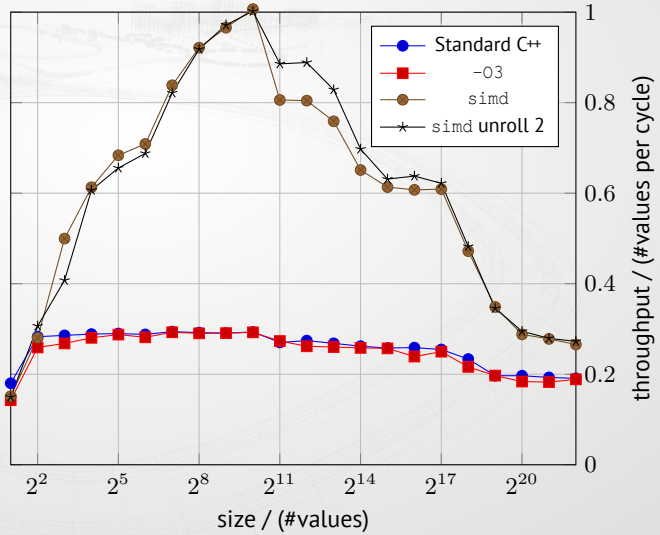
```
struct Vec3d<simd<float>>
{
  simd<float> x, y, z;

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



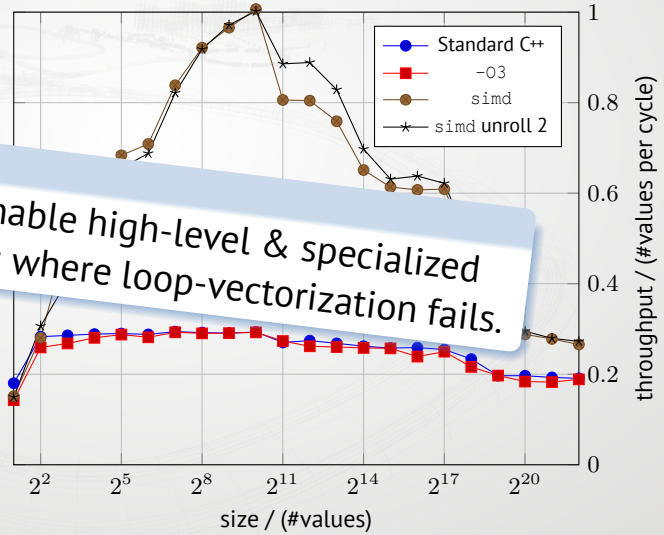
Teaser Results (GCC 13, Intel Skylake i7-8550U)

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



Take-Away #6

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

Take-Away #7

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

- Target-dependent width
- Visible "conversion" between scalars and `simd` objects (loads & stores vs. gathers & scatters)
- Conditional assignment instead of branching

Consequently …

… data-parallel types make the design challenges wrt. efficient vectorization more obvious.

… subsequent designs can profit from this experience.

Data structures are the main challenge for efficient data-parallel execution.

- 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.

Granted, the same is true for the SIMD execution

Data structures are the main challenge for efficient data-parallel execution.

- 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.

Granted, the same is true for the SIMD execution policy on an AoS[1] range

[1] Array of Struct

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, a contribution from 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`.

Tricky because of portability concerns.

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

Better gather & scatter integration

Example

```
1 void f(std::vector<float>& data, simd_or<int> auto indexes) {
2   data[indexes] = std::sin(data[indexes]);
3 }
```

automatic type vectorization

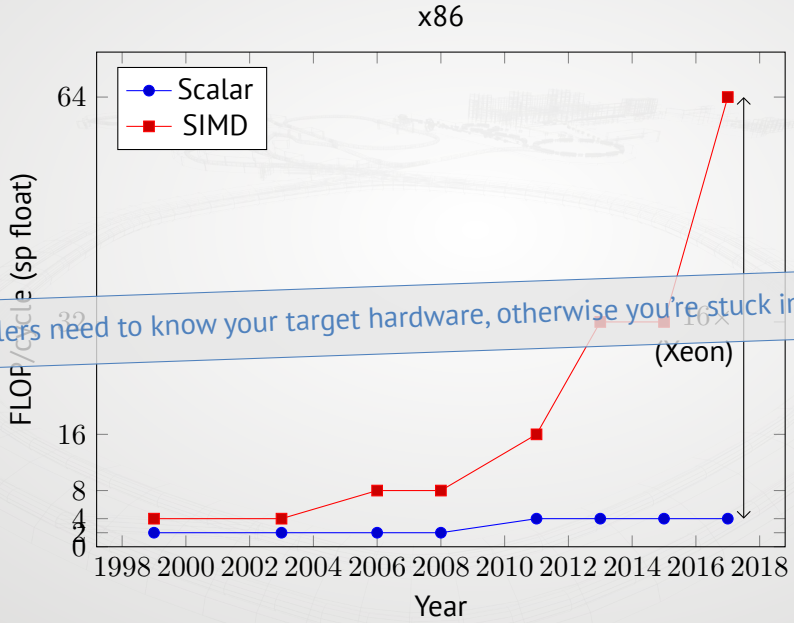
```cpp
struct Point {
  float x, y, z;
};
using PointV = simdize<Point>;

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

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

3rd-party solution until C++ reflection lands?

1. `std::simd` is for you — because you care about performance!
2. `std::simd` expresses data-parallelism — compiled to SIMD and ILP!
3. Focus on data-parallelism not SIMD instructions/registers!
4. `std::simd` enables separation of
   - serial execution,
   - synchronously parallel execution (SIMD and ILP), and
   - asynchronously parallel execution (threads).
5. `std::simd` guides you to design scalable and portable parallelization.
6. `std::simd` enables high-level & specialized parallelism abstractions with full control where loop-vectorization fails.
7. Consider type-vectorization not only loop-vectorization.
8. `std::simd` provides an API & ABI for vectorization across translation units (including library boundaries).



Compilers need to know your target hardware, otherwise you're stuck in 2006!

High-throughput computing without overhead

```
1 mov     eax, DWORD PTR [rdi]
2 imul    eax, eax
3 mov     DWORD PTR [rdi], eax
```

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

```
1 vmovdqu32    zmm0, ZMMWORD PTR [rdi]
2 vpmulld      zmm0, zmm0, zmm0
3 vmovdqu32    ZMMWORD PTR [rdi], zmm0
```

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

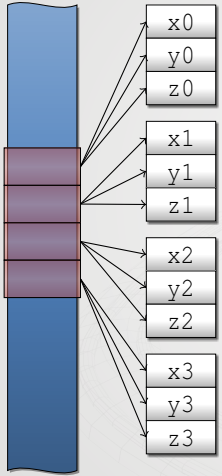
Take-Away #8

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

Data Structures for SIMD Processing

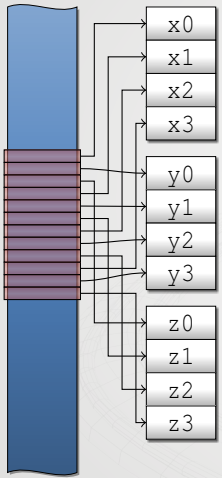
Consider this class template definition:

```
1  template <typename T> struct Point {
2    T x, y, z;
3
4    Point normalized() const {
5      using std::sqrt;
6      const auto scale = 1 / sqrt(x * x + y * y + z * z);
7      return {x * scale, y * scale, z * scale};
8    }
9  };
```



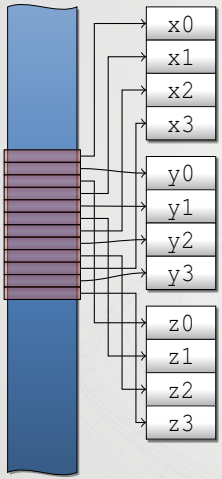
Array of Struct

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



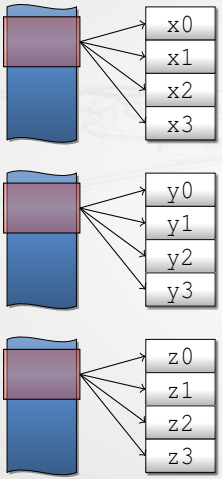
Array of Struct

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



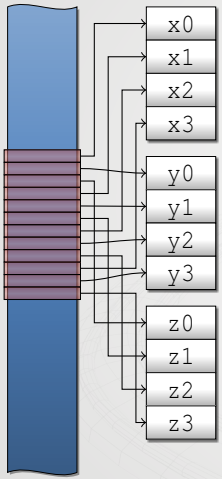
Array of Struct

`vector<Point<float>>`



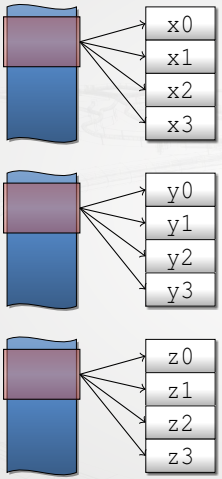
Struct of Array

`Point<valarray<float>>`



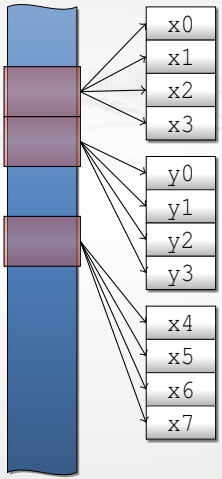
Array of Struct

`vector<Point<float>>`



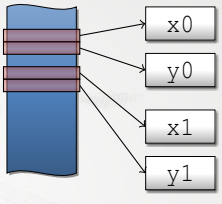
Struct of Array

`Point<valarray<float>>`

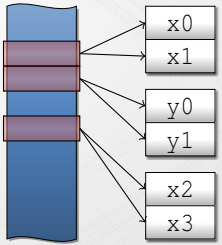


Array of vectorized Struct

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



$\mathcal{W}_{\texttt{float}} = 1$



$\mathcal{W}_{\texttt{float}} = 2$

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<stdx::simd<float>>>&
          points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

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

```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};
  }
};
```

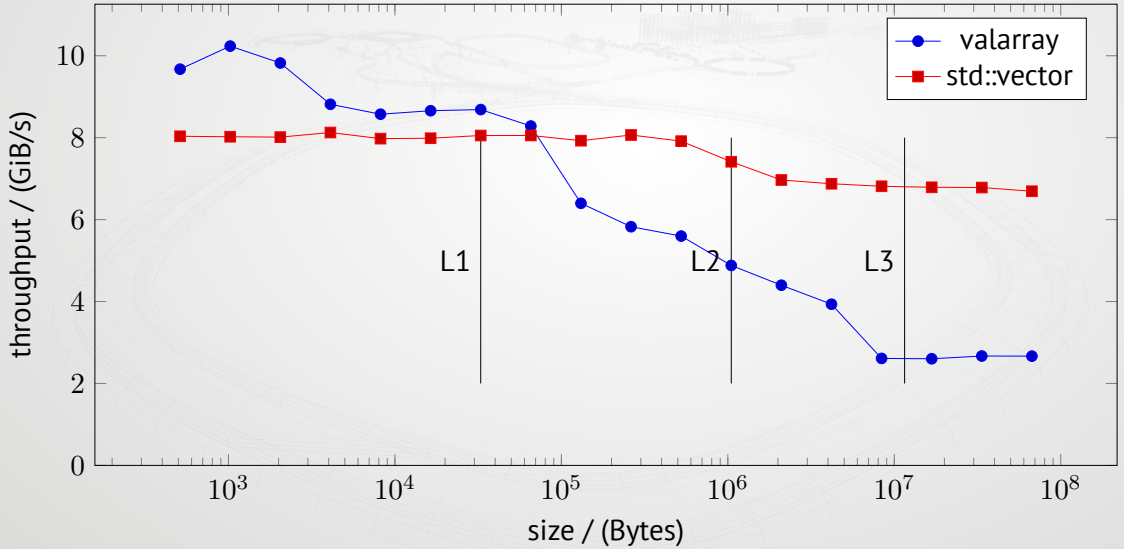
```cpp
void aos(const std::vector<Point<float>>& points) {
  for (auto&p : points) {
    p = p.normalized();
  }
}
```

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

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

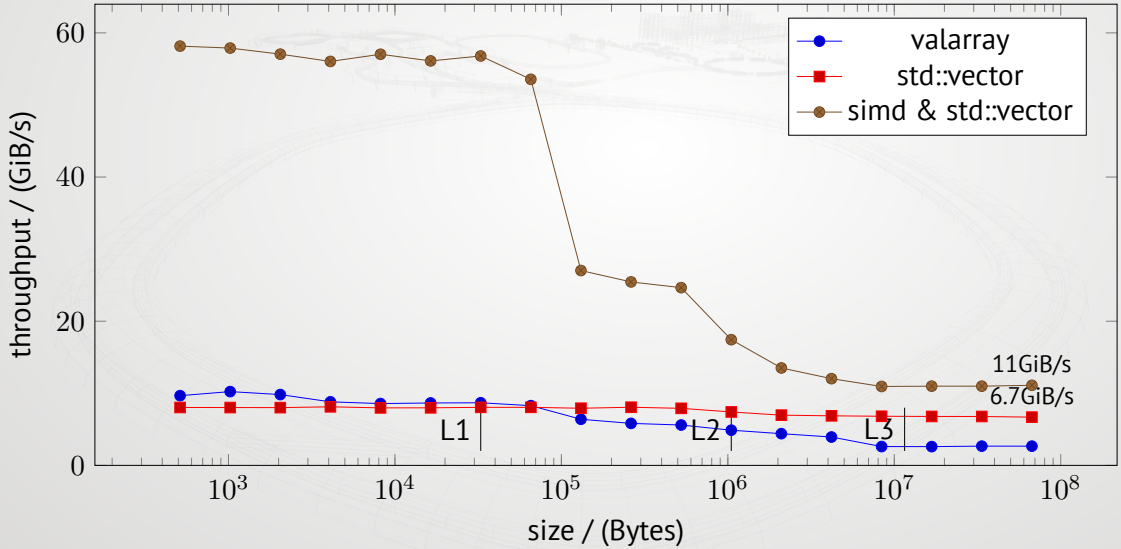
Normalization benchmark

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

```
; 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

```
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
```

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
```

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

ABI tag / width default

- Compiler flags determine the default ABI tag / SIMD width.
- `simd<T>` sets the ABI tag to the widest efficient $\mathcal{W}_{\mathtt{T}}$ 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 $\mathcal{W}_{\mathtt{T}}$ 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*:

```
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.

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. 😉

Regularity

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);`

- 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`.

Regularity

Regularity 🏆 — 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));`

- 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`.