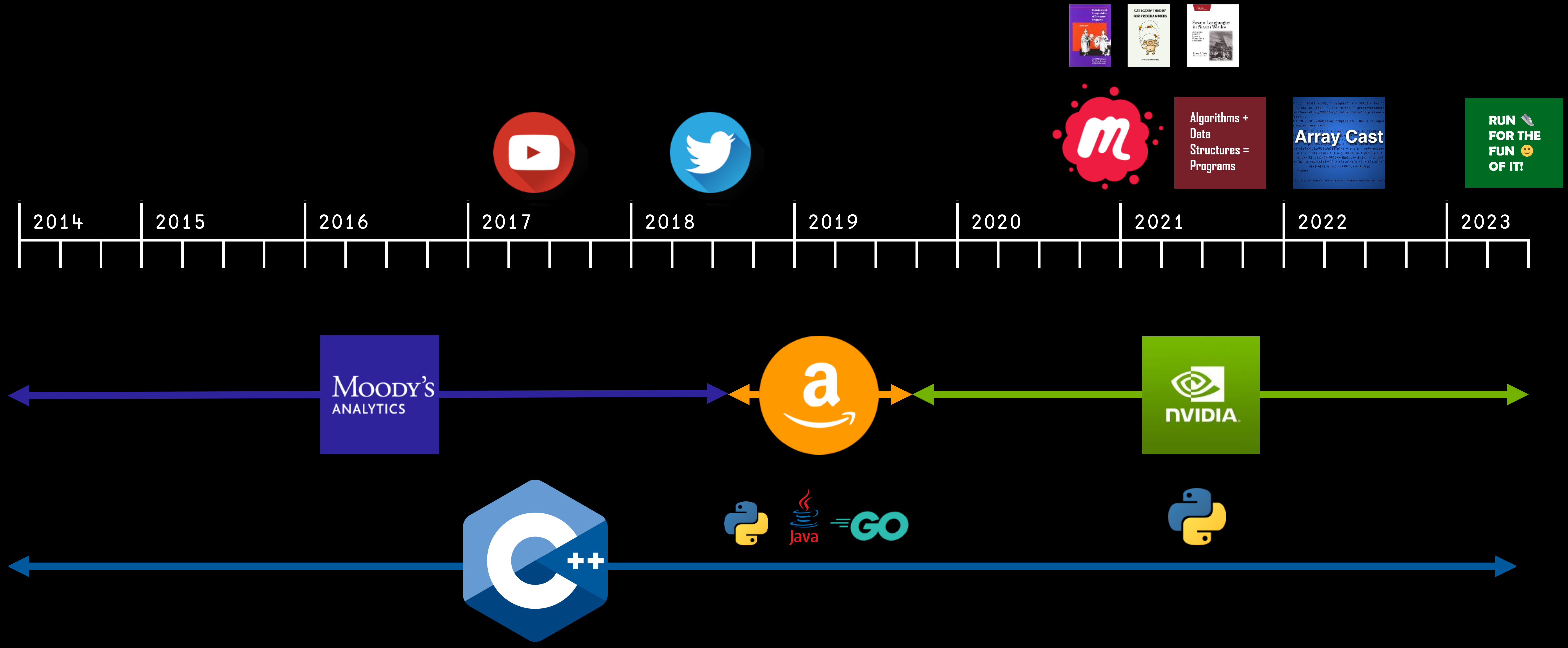
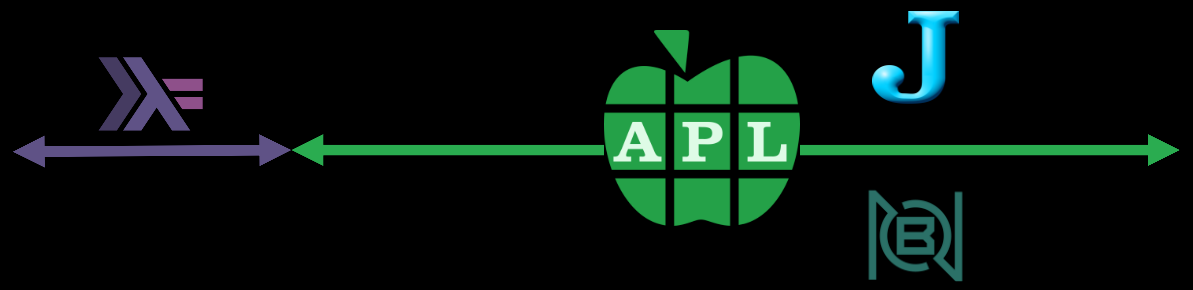
New Algorithms in C++23

Conor Hoekstra

code_report





About Me

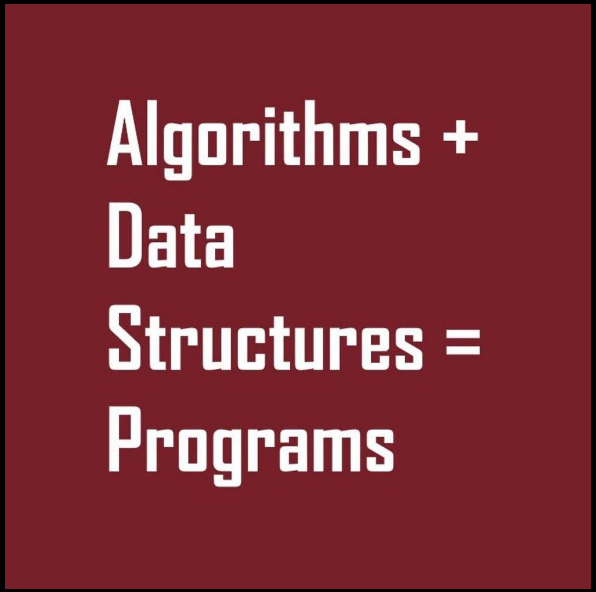
Conor Hoekstra / @code_report

Algorithms +
Data
Structures =
Programs

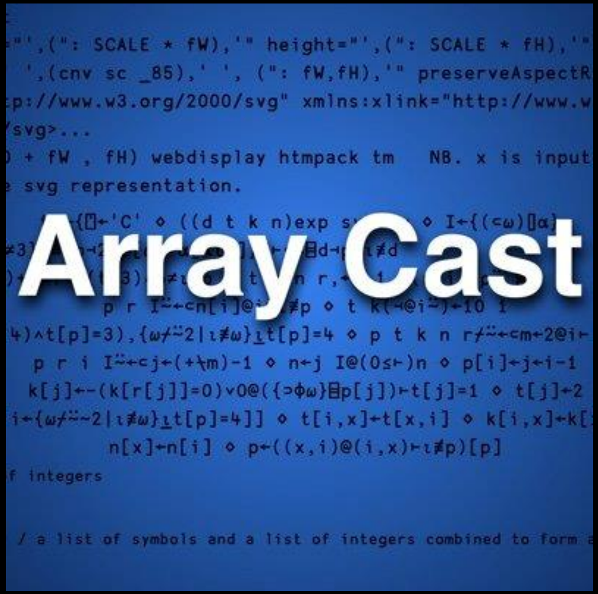
Array Cast

319 Videos

32 (20) Talks



132 Episodes
@adspthepodcast



54 Episodes
@arraycast

9 Episodes
@conorhoekstra

https://github.com/codereport/Content

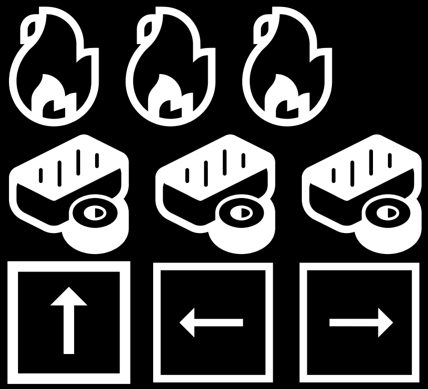
Code Links

Example	Language	GitHub Link	Godbolt Link
Warm Up: Negatives	C++	Link	https://godbolt.org/z/8TEevabqd
Sushi for Two	C++	Link	https://godbolt.org/z/69Kh8baz3
Sushi for Two	Circle	Link	https://godbolt.org/z/P6PxMnhMz
Max Gap	C++	Link	https://godbolt.org/z/P43EhYcsj
Max Gap	Circle	Link	https://godbolt.org/z/3K598jMMa
`filter_out_html_tags`	Circle	Link	https://godbolt.org/z/on5xMG5ax

Algorithm Land Overview

Problems:

- Warm Up
- Sushi for Two
- Max Gap
- filter_html_tags



C++ Algorithm Land Overview

C++98 Iterator Algorithms
`std::find_if(a.begin(), a.end(), f)`
`std::count(b.begin(), b.end(), 3)`
`std::all_of(c.begin(), c.end(), f)`
`std::sort(d.begin(), d.end())`
...

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms
`std::find_if(a.begin(), a.end(), f)`	`std::ranges::find_if(a, f)`
`std::count(b.begin(), b.end(), 3)`	`std::ranges::count(b, 3)`
`std::all_of(c.begin(), c.end(), f)`	`std::ranges::all_of(c, f)`
`std::sort(d.begin(), d.end())`	`std::ranges::sort(d)`
...	...

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms	C++20/23 Range Adaptors & Factories
`std::find_if(a.begin(), a.end(), f)`	`std::ranges::find_if(a, f)`	`std::views::take`
`std::count(b.begin(), b.end(), 3)`	`std::ranges::count(b, 3)`	`std::views::drop`
`std::all_of(c.begin(), c.end(), f)`	`std::ranges::all_of(c, f)`	`std::views::transform`
`std::sort(d.begin(), d.end())`	`std::ranges::sort(d)`	`std::views::filter`
...	...	`std::views::chunk_by`
		...

C++ Algorithm Land Overview

C++98 Iterator Algorithms	C++20 Range Algorithms	C++20/23 Range Adaptors & Factories
std::find_if(a.begin(), a.end(), f)	std::ranges::find_if(a, f)	std::views::take
std::count(b.begin(), b.end(), 3)	std::ranges::count(b, 3)	std::views::drop
std::all_of(c.begin(), c.end(), f)	std::ranges::all_of(c, f)	std::views::transform
std::sort(d.begin(), d.end())	std::ranges::sort(d)	std::views::filter
...	...	std::views::chunk_by
		...

C++20/23 Range
Adaptors & Factories

std::views::take

std::views::drop

std::views::transform

std::views::filter

std::views::chunk_by

C++20/23 Range Adaptors & Factories

std::views::take
std::views::drop
std::views::transform
std::views::filter

std::views::chunk_by

C++20/23 Range Adaptors & Factories

drop
drop_while
elements (keys | values)
filter
iota
join
reverse
split
take
take_while
transform

adjacent (pairwise)
adjacent_transform (pairwise_transform)
cartesian_product
chunk
chunk_by
enumerate
join_with
slide
stride
zip
zip_transform

[[digression]]

Different Programming Paradigms

- Collection Oriented Programming ☆
  - Functional-Style
- Function Programming
- Object-Oriented Programming
- Imperative Programming

Libraries		Languages
C++	Ranges	APL
Rust	Iterators	
Java	Streams	Pharo

[[end of digression]]

Warm Up

🔥🔥🔥

```cpp
int num_negatives(std::vector<int> nums) {
    int count = 0;
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] < 0) ++count;
    }
    return count;
}
```

```cpp
int num_negatives(std::vector<int> nums) {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```

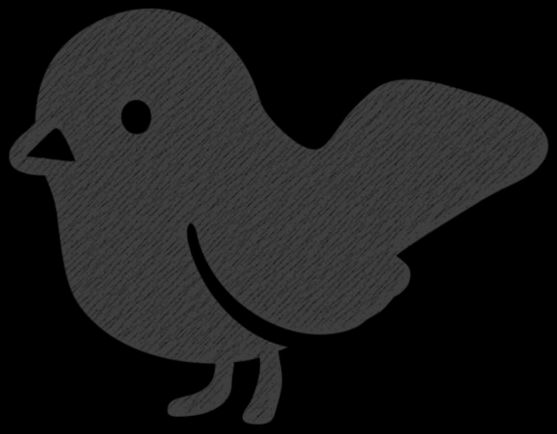
```cpp
auto num_negatives(std::vector<int> nums) {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```

```cpp
auto num_negatives(std::vector<int> nums) -> int {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```

```cpp
auto num_negatives(std::vector<int> nums) -> int {
    return std::count_if(nums.cbegin(), nums.cend(),
        [](auto e) { return e < 0; });
}
```

```cpp
auto num_negatives(std::vector<int> nums) -> int {
    return std::ranges::count_if(nums,
        [](auto e) { return e < 0; });
}
```

```cpp
auto num_negatives(std::vector<int> nums) -> int {
    return std::ranges::count_if(nums, lt_(0));
}
```



```cpp
using namespace combinators;

auto num_negatives(std::vector<int> nums) -> int {
    return std::ranges::count_if(nums, lt_(0));
}
```

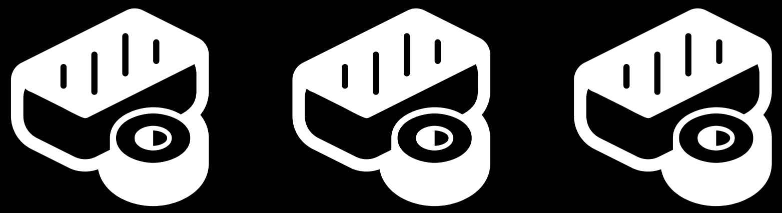
inevitably someone says…

Loops are easier to read
/ understand … everyone
knows them

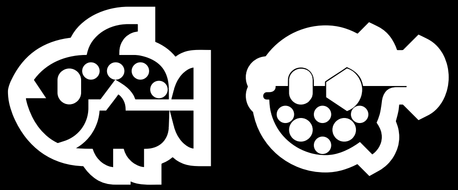
```cpp
int num_negatives(std::vector<int> nums) {
    int count = 0;
    for (auto const num : nums) {
        if (num < 0) ++count;
    }
    return count;
}
```

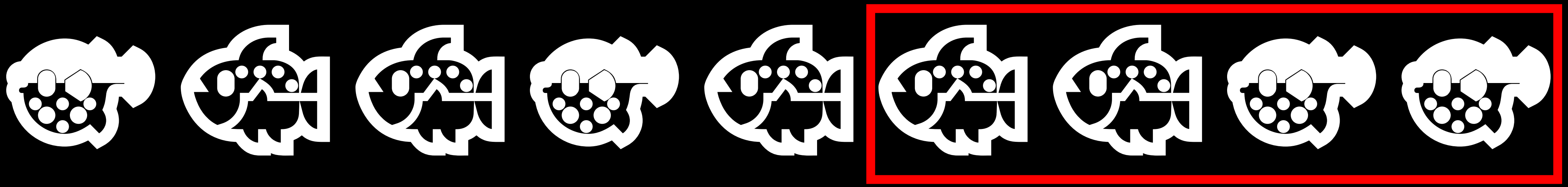
introducing one of my favorite problems of all time…

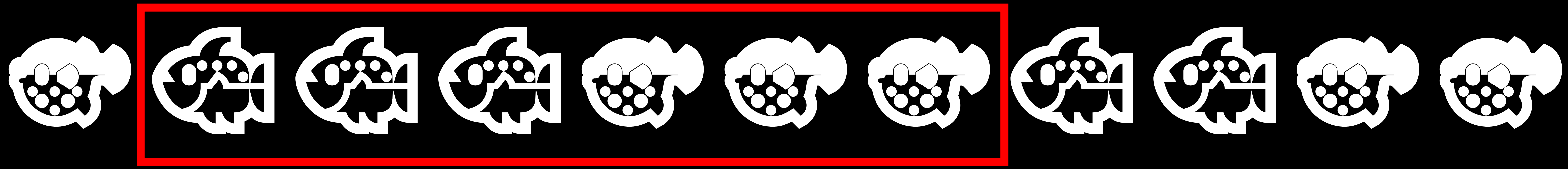
Sushi for Two



https://codeforces.com/contest/1138/problem/A







```cpp
template <int N>
constexpr auto sushi_for_two(std::array<int, N> sushi) {
    int current_sushi       = 0;
    int sushi_in_a_row      = 0;
    int prev_sushi_in_a_row = 0;
    int max_of_mins         = 0;
    for (auto const s : sushi) {
        if (current_sushi != s) {
            current_sushi = s;
            if (prev_sushi_in_a_row == 0) {
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row      = 1;
            } else {
                auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
                max_of_mins         = std::max(max_of_mins, min);
                prev_sushi_in_a_row = sushi_in_a_row;
                sushi_in_a_row      = 1;
            }
        } else {
            sushi_in_a_row += 1;
        }
    }
    auto const min = std::min(sushi_in_a_row, prev_sushi_in_a_row);
    max_of_mins    = std::max(max_of_mins, min);
    return max_of_mins * 2;
}
```

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(_eq_)
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```

q

```
sf2: { x }

[1, 2, 2, 1, 2, 2, 2, 1, 1]
```

```
sf2: { chunk x }

[[1], [2, 2], [1], [2, 2, 2], [1, 1]]
```

```
sf2: { count each chunk x }

[1, 2, 1, 3, 2]
```

```
sf2: { (&) prior count each chunk x }
```

```
[1, 1, 1, 1, 2]
```

Hoogle Translate

prior

CUDA	**adjacent_difference**	Thrust	Doc
C++	**adjacent_difference**	<numeric>	Doc
APL	**/ (n-wise reduce)**	-	Doc
Haskell	**mapAdjacent**	Data.List.HT	Doc
Kotlin	**zipWithNext**	collections	Doc
q	**prior**	-	Doc
C++	**adjacent_transform**	<ranges>	Doc

```
sf2: { (&) prior count each chunk x }
```

```
[1, 1, 1, 1, 2]
```

```
sf2: { 1 _ (&) prior count each chunk x }
```

```
[1, 1, 1, 2]
```

```
sf2: { max 1 _ (&) prior count each chunk x }
```

2

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
```

2

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
```

4

```
sf2: 2 * max 1 _ (&) prior count each chunk ::
```

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi;
}
```

[1, 2, 2, 1, 2, 2, 2, 1, 1]

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{});
}
```

[[1], [2, 2], [1], [2, 2, 2], [1, 1]]

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance);
}
```

[1, 2, 1, 3, 2]

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); });
}
```

[1, 1, 1, 2]

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); }));
}
```

2

2 very irritating things about this code

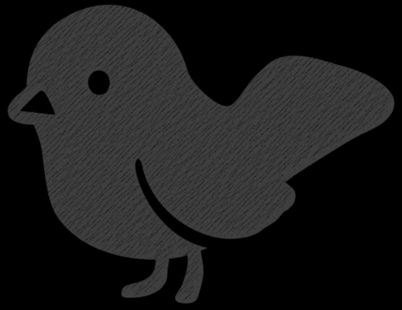
```
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(
            [](auto a, auto b) { return std::min(a, b); }));
}
```

4

```cpp
using namespace std::views;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```

```cpp
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(std::equal_to{})
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```

```cpp
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return 2 * std::ranges::max(sushi
        | chunk_by(_eq_)
        | transform(std::ranges::distance)
        | adjacent_transform<2>(_min_));
}
```



```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
           |> chunk_by($, _eq_)
           |> transform($, std::ranges::distance)
           |> adjacent_transform<2>($, _min_)
           |> std::ranges::max($) * 2;
}
```

CIRCLE

COMPILER EXPLORER

-std=c++2b --gcc-toolchain /opt/compiler-explorer/gcc-13.1.0/ -Wl,-rpath,/opt/compiler-explorer/gcc-13.1.0/lib64/

```cpp
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
           |> chunk_by($, _eq_)
           |> transform($, std::ranges::distance)
           |> adjacent_transform<2>($, _min_)
           |> std::ranges::max($) * 2;
}
```

Code Links

Example	Language	GitHub Link	Godbolt Link
Warm Up: Negatives	C++	Link	https://godbolt.org/z/8TEevabqd
Sushi for Two	C++	Link	https://godbolt.org/z/69Kh8baz3
Sushi for Two	Circle	Link	https://godbolt.org/z/P6PxMnhMz
Max Gap	C++	Link	https://godbolt.org/z/P43EhYcsj
Max Gap	Circle	Link	https://godbolt.org/z/3K598jMMa
`filter_out_html_tags`	Circle	Link	https://godbolt.org/z/on5xMG5ax

https://github.com/codereport/Content/Talks

```
using namespace std::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
           |> chunk_by($, _eq_)
           |> transform($, std::ranges::distance)
           |> adjacent_transform<2>($, _min_)
           |> std::ranges::max($) * 2;
}
```

```cpp
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
           |> chunk_by($, _eq_)
           |> transform($, std::ranges::distance)
           |> adjacent_transform<2>($, _min_)
           |> std::ranges::max($) * 2;
}
```

```haskell
sushiForTwo :: [Int] -> Int
sushiForTwo = (*2)
            . maximum
            . mapAdjacent min
            . map length
            . group
```

CIRCLE

```cpp
auto sushi_for_two(std::vector<int> sushi) {
    return sushi
             |> chunk_by($, _eq_)
             |> transform($, std::ranges::distance)
             |> adjacent_transform<2>($, _min_)
             |> std::ranges::max($) * 2;
}
```

```haskell
sushiForTwo :: [Int] -> Int
sushiForTwo = (*2)
            . maximum
            . mapAdjacent min
            . map length
            . group
```

q

```q
sf2: 2 * max 1 _ (&) prior
     count each chunk ::
```

q

```
sf2: 2 * max 1 _ (&) prior
count each chunk ::
```

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
```

```
sf2: { 2 * max 1 _ (&) prior count each chunk x }
chunk: { (where differ x) cut x }
```

```
sf2: { 2 * max 1 _ (&) prior deltas where differ x }
```

```
sf2: { 2 * max 1 _ (&) prior deltas (where differ x) , count x }
```

CIRCLE

```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi;
}
```

[1, 2, 2, 1, 2, 2, 2, 1, 1]

CIRCLE

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1));
}
```

```
[1, 0, 1, 1, 0, 0, 1, 0]
```

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1));
}
```

```
[(1,1), (0,2), (1,3), (1,4), (0,5), (0,6), (1,7), (0,8)]
```

```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst);
}
```

[(1,1), (1,3), (1,4), (1,7)]

CIRCLE

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> transform($, _snd);
}
```

[1, 3, 4, 7]

```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($);
}
```

[1, 3, 4, 7]

```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()));
}
```

[0, 1, 3, 4, 7, 9]

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $);
}
```

[1, 2, 1, 3, 2]

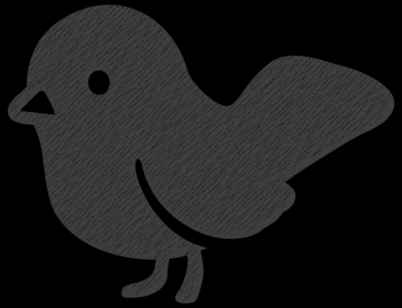
```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1));
}
```

[1, 1, 1, 2]

CIRCLE

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```

CIRCLE



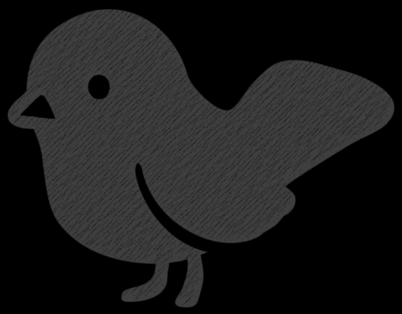
```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```

```
auto adjacent_transform_2(auto&& rng, auto op) {
    return rng |> zip_with(op, $, $ | drop(1));
}
auto differ(auto&& rng) { return adjacent_transform_2(rng, _neq_); }
auto deltas(auto&& rng) { return adjacent_transform_2(rng, _c(_sub_)); }

auto indices(auto&& rng) {
    return rng
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $);
}
```

CIRCLE

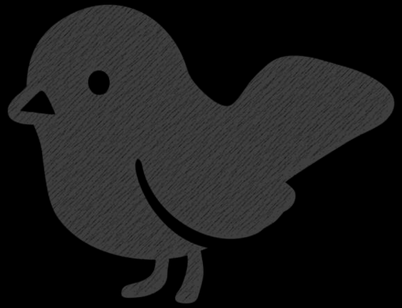
```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```

CIRCLE



```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
           |> differ($)
           |> indices($)
           |> concat($, single(sushi.size()))
           |> deltas($)
           |> adjacent_transform_2($, _min_)
           |> std::ranges::max($) * 2;
}
```

```
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
    return sushi
        |> zip_with(_neq_, $, $ | drop(1))
        |> zip($, iota(1))
        |> filter($, _fst)
        |> values($)
        |> concat(single(0), $, single(sushi.size()))
        |> zip_with(_sub_, $ | drop(1), $)
        |> zip_with(_min_, $, $ | drop(1))
        |> std::ranges::max($) * 2;
}
```



```cpp
using namespace ranges::views;
using namespace combinators;

auto sushi_for_two(std::vector<int> sushi) {
  auto indices = concat(                                      //
      concat(single(0),                                       //
             zip(zip_with(_neq_, sushi, sushi | drop(1)),     //
                 iota(1))                                     //
                 | filter(_fst)                               //
                 | values),                                   //
      single(sushi.size()));                                  //
  auto deltas = zip_with(_sub_, indices | drop(1), indices);
  return 2 * ranges::max(zip_with(_min_, deltas, deltas | drop(1)));
}
```

Max Gap

https://leetcode.com/problems/maximum-gap/

[8, 4, 1, 3, 10]

[1, 3, 4, 8, 10]

[1, 3, 4, 8, 10]
[2, 1, 4, 2]

[1, 3, 4, 8, 10]
[2, 1, **4**, 2]

```cpp
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    return nums;
}
```

[8, 4, 1, 3, 10]

```cpp
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums;
}
```

[1, 3, 4, 8, 10]

```cpp
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums
      | adjacent_transform<2>(std::minus{});
}
```

[2, 1, 4, 2]

anyone notice the bug?

```cpp
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(nums
      | adjacent_transform<2>(std::minus{}));
}
```

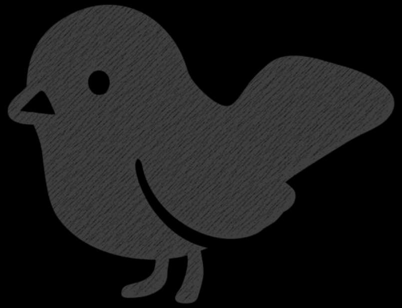
4

```cpp
using namespace std::views;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(nums
      | reverse
      | adjacent_transform<2>(std::minus{}));
}
```

4

```cpp
using namespace std::views;
using namespace combinators;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return std::ranges::max(nums
      | adjacent_transform<2>(_c(_sub_)));
}
```

CIRCLE



```cpp
using namespace std::views;
using namespace combinators;

auto max_gap(std::vector<int> nums) {
    std::ranges::sort(nums);
    return nums
           |> adjacent_transform<2>($, _c(_sub_))
           |> std::ranges::max($);
}
```

filter_out_html_tags

around first-parameter-passing. It would be nice to not have to pay more syntax when we don't need to.

On the whole though the argument seems to strongly favor placeholders, and if anything exploring a special case of the pipeline operator that does left-threading only and has a more restrictive right-hand side to avoid potential bugs. That might still allow the best of both worlds.

A recent Conor Hoekstra talk has a nice example that I'll present multiple different ways (in all cases, I will not use the `|` from Ranges).

With left-threading, the example looks like:

```
auto filter_out_html_tags(std::string_view sv) -> std::string {
  auto angle_bracket_mask =
    sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

  return std::views::zip_transform(
      std::logical_or(),
      angle_bracket_mask,
      angle_bracket_mask |> rv::scan_left(std::not_equal_to{})
    |> std::views::zip(sv)
    |> std::views::filter([](auto t){ return not std::get<0>(t); })
    |> std::views::values()
    |> std::ranges::to<std::string>();
}
```

Notably here we run into both of the limitations of left-threading: we need to pipe into a parameter other than the first and we need to pipe more than once. That requires introducing a new named variable, which is part of what this facility is trying to avoid the need for. This is not a problem for either of the placeholder-using models, as we'll see shortly.

With the placeholder-mandatory model, we don't need that temporary, since we can select which parameters of `zip_transform` to pipe into, and indeed we can pipe twice (I'll have more to say about nested placeholders later):

```
auto filter_out_html_tags(std::string_view sv) -> std::string {
  return sv
    |> std::views::transform(%, [](char c){ return c == '<' or c == '>'; })
    |> std::views::zip_transform(std::logical_or{}, %, % |> rv::scan_left(%, std::not_equal_to{}))
    |> std::views::zip(%, sv)
    |> std::views::filter(%, [](auto t){ return not std::get<0>(t); })
    |> std::views::values(%)
    |> std::ranges::to<std::string>(%);
```

https://wg21.link/p2672

```cpp
auto filter_out_html_tags(std::string_view sv) -> std::string {
  auto angle_bracket_mask =
    sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

  return std::views::zip_transform(
      std::logical_or(),
      angle_bracket_mask,
      angle_bracket_mask |> rv::scan_left(std::not_equal_to{})
    |> std::views::zip(sv)
    |> std::views::filter([](auto t){ return not std::get<0>(t); })
    |> std::views::values()
    |> std::ranges::to<std::string>();
}
```

```
'<div>Hello <b>C++North!</b></div>'
```

```
'<'='<div>Hello <b>C++North!</b></div>'
```

```
'<'='<div>Hello <b>C++North!</b></div>'

1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
```

```
'<'='<div>Hello <b>C++North!</b></div>'
     100000000001000000000001000100000
```

```
'<'='<div>Hello <b>C++North!</b></div>'

1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
```

```
'<'=
    '<div>Hello <b>C++North!</b></div>'

1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
```

```
                '<'=
'<div>Hello <b>C++North!</b></div>'

1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
```

```
{'<'=⍵}
'<div>Hello <b>C++North!</b></div>'

1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0
```

```
{ω∊'<>'}
'<div>Hello <b>C++North!</b></div>'

1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 1
```

```
{≠\⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

1 1 1 1 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 1 1 0 1 1 1 1 1 0
```

```
{~≠\⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1
```

```
{⍵/⍨~≠\⍵∊'<>'}
      '<div>Hello <b>C++North!</b></div>'

0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1
```

```
{⍵/⍨~≠\⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

< d i v > H e l l o   < b > C + + N o r t h ! < / b > < / d i v >
0 0 0 0 1 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 0 0 0 1 0 0 0 0 0 1
```

```
{⍵/⍨~≠\⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

>Hello >C++North!>>
```

```
{⍵/⍨(~⊢∨≠\)⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

Hello C++North!
```

```
{⍵/⍨(⊢⍱≠\)⍵∊'<>'}
'<div>Hello <b>C++North!</b></div>'

Hello C++North!
```

```cpp
auto filter_out_html_tags(std::string_view sv) -> std::string {
  auto angle_bracket_mask =
    sv |> std::views::transform([](char c){ return c == '<' or c == '>'; });

  return std::views::zip_transform(
      std::logical_or(),
      angle_bracket_mask,
      angle_bracket_mask |> rv::scan_left(std::not_equal_to{})
    |> std::views::zip(sv)
    |> std::views::filter([](auto t){ return not std::get<0>(t); })
    |> std::views::values()
    |> std::ranges::to<std::string>();
}
```

CIRCLE

```cpp
using namespace std::views;

auto filter_out_html_tags(std::string_view sv) {
    return sv
        |> transform($, [](auto e) { return e == '<' or e == '>'; })
        |> zip_transform(std::logical_or{}, $, scan_left($, true, std::not_equal_to{}))
        |> zip($, sv)
        |> filter($, [](auto t) { return not std::get<0>(t); })
        |> values($)
        |> ranges::to<std::string>($);
}
```

CIRCLE

```
using namespace std::views;
using namespace combinators;

auto filter_out_html_tags(std::string_view sv) {
    return sv
        |> transform($, _phi(_eq('<'), _or_, _eq('>')))
        |> zip_transform(_or_, $, scan_left($, true, _neq_))
        |> zip($, sv)
        |> filter($, _b(_not, _fst))
        |> values($)
        |> ranges::to<std::string>($);
}
```

[[digression]]

Parallel Block-Delayed Sequences

Sam Westrick
Carnegie Mellon University
Pittsburgh, PA, USA
swestric@cs.cmu.edu

Mike Rainey
Carnegie Mellon University
Pittsburgh, PA, USA
me@mike-rainey.site

Daniel Anderson
Carnegie Mellon University
Pittsburgh, PA, USA
dlanders@cs.cmu.edu

Guy E. Blelloch
Carnegie Mellon University
Pittsburgh, PA, USA
guyb@cs.cmu.edu

Abstract

Programming languages using functions on collections of values, such as map, reduce, scan and filter, have been used for over fifty years. Such collections have proven to be particularly useful in the context of parallelism because such functions are naturally parallel. However, if implemented naively they lead to the generation of temporary intermediate collections that can significantly increase memory usage and runtime. To avoid this pitfall, many approaches use "fusion" to combine operations and avoid temporary results. However, most of these approaches involve significant changes to a compiler and are limited to a small set of functions, such as maps and reduces.

In this paper we present a library-based approach that fuses widely used operations such as scans, filters, and flattens. In conjunction with existing techniques, this covers most of the common operations on collections. Our approach is based on a novel technique which parallelizes over blocks, with streams within each block. We demonstrate the approach by implementing libraries targeting multicore parallelism in two languages: Parallel ML and C++, which have very different semantics and compilers. To help users understand when to use the approach, we define a cost semantics that indicates when fusion occurs and how it reduces memory allocations. We present experimental results for a dozen benchmarks that demonstrate significant reductions in both time and space. In most cases the approach generates code that is near optimal for the machines it is running on.

CCS Concepts: • **Software and its engineering** → **Parallel programming languages**; *Functional languages*; • **Theory of computation** → *Parallel algorithms*.

Keywords: parallel programming, fusion, collections, functional programming

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea

ACM ISBN 978-1-4503-9204-4/22/04.
https://doi.org/10.1145/3503221.3508434

1 Introduction

Collection-oriented programming is a style of programming in which programs use operations over collections of values, such as map, reduce, filter, and scan. Languages supporting this style date back to 1960s with APL (arrays) [18], SETL (sets and maps) [29], Codd's relational algebra (relations) [11] and FP (sequences) [3]. These languages allowed a particularly simple and elegant way to work with collections. With the advent of highly parallel machines in the mid 80s, there was a significant increase in interest in this style of programming. The observation is that by raising the level of abstraction, sequential loops go away, and code often becomes inherently parallel. Furthermore, working with collections promotes a functional style of programming, and hence mostly avoids mutation and, in the context of parallelism, the potential dangerous data races they cause. Early such *data parallel* languages include CM-Lisp [19], C* [28], and Nesl [4]. Later ones used in a distributed setting include map-reduce [14] and Spark [37].

It was quickly noted, however, that collections can incur large overheads due to the generation of intermediate results. For example, a map squaring every element of a vector, followed by a reduce summing the results would naïvely generate an intermediate vector of the products before summing them. A loop, on the other hand, would multiply and add as it went along. The generation of this intermediate vector wastes not only space but also time, due to additional reads and writes in situations where memory bandwidth is often the bottleneck. This problem of avoiding intermediate results has been studied extensively since the 1970s [1, 35] and is often referred to as "loop-fusion", just "fusion", or originally "jamming". Fusion has been applied to data-parallel languages since the start of the 90s with dozens of papers on the topic (e.g., [9, 12, 13, 17, 20–22, 24, 26, 31]). Most of these techniques rely on compiler transformations.

Interestingly, however, Keller et al. [20] were able to show that by taking advantage of standard compiler optimizations, fusion can be implemented efficiently for sequences as a

guyb@cs.cmu.edu

llections of
e been used
to be partic-
e such func-
nted naively
mediate col-
y usage and
use "fusion"
sults. How-
nt changes
ctions, such

proach that
ers, and flat-
this covers

Keywords: parallel programming, fusion, collections, functional programming

1 Introduction

Collection-oriented programming is a style of programming in which programs use operations over collections of values, such as map, reduce, filter, and scan. Languages supporting this style date back to 1960s with APL (arrays) [18], SETL (sets and maps) [29], Codd's relational algebra (relations) [11] and FP (sequences) [3]. These languages allowed a particularly simple and elegant way to work with collections. With the advent of highly parallel machines in the mid 80s, there was a significant increase in interest in this style of programming. The observation is that by raising the level of abstraction, sequential loops go away, and code often becomes inherently parallel. Furthermore, working with or reduce can be composed. For example, a map followed by a reduce would never instantiate the intermediate list, only instantiating one element at a time, requiring $O(1)$ additional memory beyond the input to run.

Collection-oriented programming with stream fusion has gained recent popularity in the programming community, as demonstrated for example by the C++20 ranges library [27]. It promotes collection-oriented programming by supplying a variety of generic algorithms for ranges of elements in the form of so-called *view adapters*. What makes them interesting for us is that they are implemented using stream fusion.[1] The library allows operations such as maps, filters, and flattens to be composed and fused, and is entirely library-based, requiring no language extensions or specialized compiler support. However, it is designed for sequential computation, and does not support parallelism, except in easy cases. Similar tools exist in other languages, such as the `java.util.stream` library introduced in Java 8.

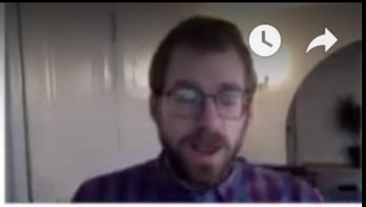
To make stream-fusion efficient requires the compiler to get rid of (possibly multiple) function calls on each iteration.

previous elements a
hand, stream fusio
is therefore not use
fusion supports ran
context. This sugge
be powerful.

2.1 Stream-of-bl

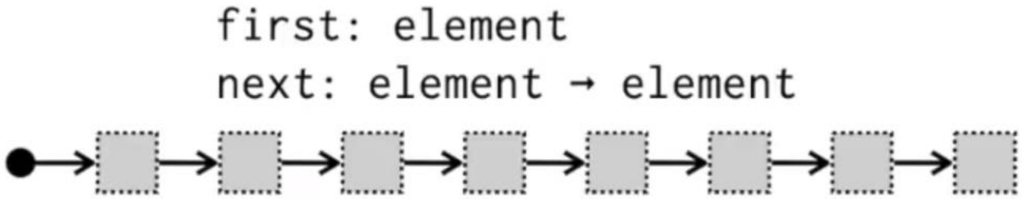
Previous work has
fusion with paralle
approach. The idea
fixed length such th
instantiate a whole
exploited within blo
apply f to each elem
proach works well w
as with GPUs or ve
correspond to the si
not well suited for c
would have to be gig

https://dl.acm.org/doi/pdf/10.1145/3503221.3508434
https://www.youtube.com/watch?v=jWaG90FbWKY

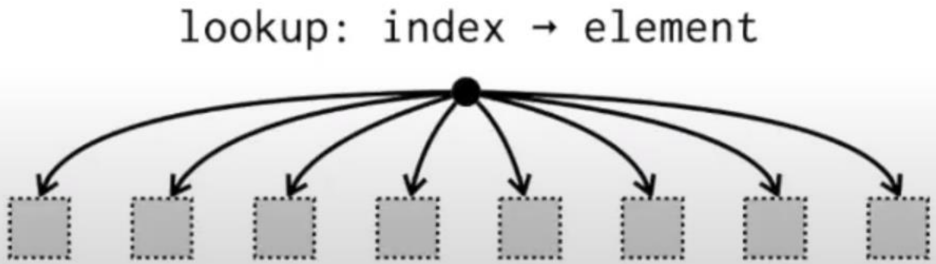


Fusion Breakdown

- **Stream fusion**
  - naturally sequential
  - e.g. lazy lists, Java streams, C++20 ranges/views, Rust iterators, ...



- **Index fusion**
  - naturally parallel
  - elements have to independent
  - good for map/zip/reduce fusion
  - e.g. Repa [1]



[1] **Regular, Shape-polymorphic, Parallel Arrays in Haskell.**
Gabriele Keller, Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier.
ICFP 2010.

Libraries		Languages
Ranges		APL
Iterators		
Java Streams		Pharo

Different Programming Paradigms

- Collection Oriented Programming ☆
  - aka Functional-Style (via T. Brindle)
- Function Programming
- Object-Oriented Programming
- Imperative Programming

[[end of digression]]

Thank You

https://github.com/codereport/Content/Talks

Conor Hoekstra

code_report

codereport

Questions?

https://github.com/codereport/Content/Talks

Conor Hoekstra

code_report

codereport