Push is Faster

using std::cpp 2025

Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>

Madrid, March 2025

Data processing as (forward) range transformation

```cpp
for(auto x: rng) {
 if(x%2 == 0) std::cout << x*3 << " ";
}
```

Data processing as (forward) range transformation

```
for(auto x: rng) {
 if(x%2 == 0) std::cout << x*3 << " ";
}
```

```
auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
auto dst =     [](auto x) { std::cout << x << " "; };

for(auto x: rng) {
  if(is_even(x)) dst(x3(x));
}
```

Data processing as (forward) range transformation

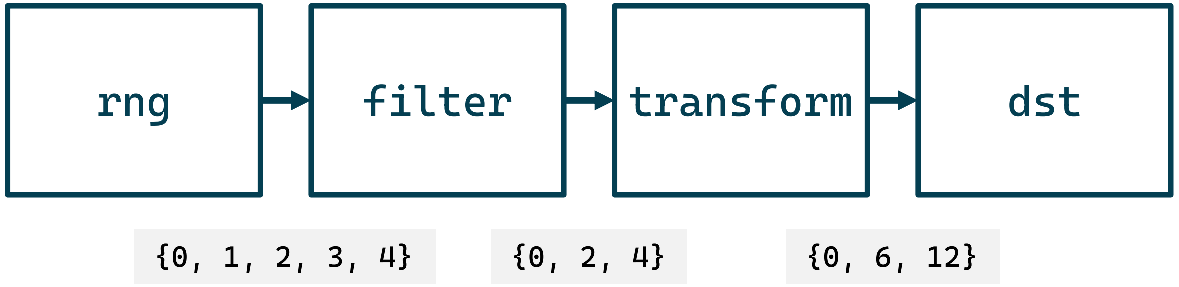
```
for(auto x: rng) {
 if(x%2 == 0) std::cout << x*3 << " ";
}
```

```
auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
auto dst =     [](auto x) { std::cout << x << " "; };

for(auto x: rng) {
  if(is_even(x)) dst(x3(x));
}
```

```
using namespace std::views;

for (auto x: rng | filter(is_even) | transform(x3)) {
  dst(x);
}
```



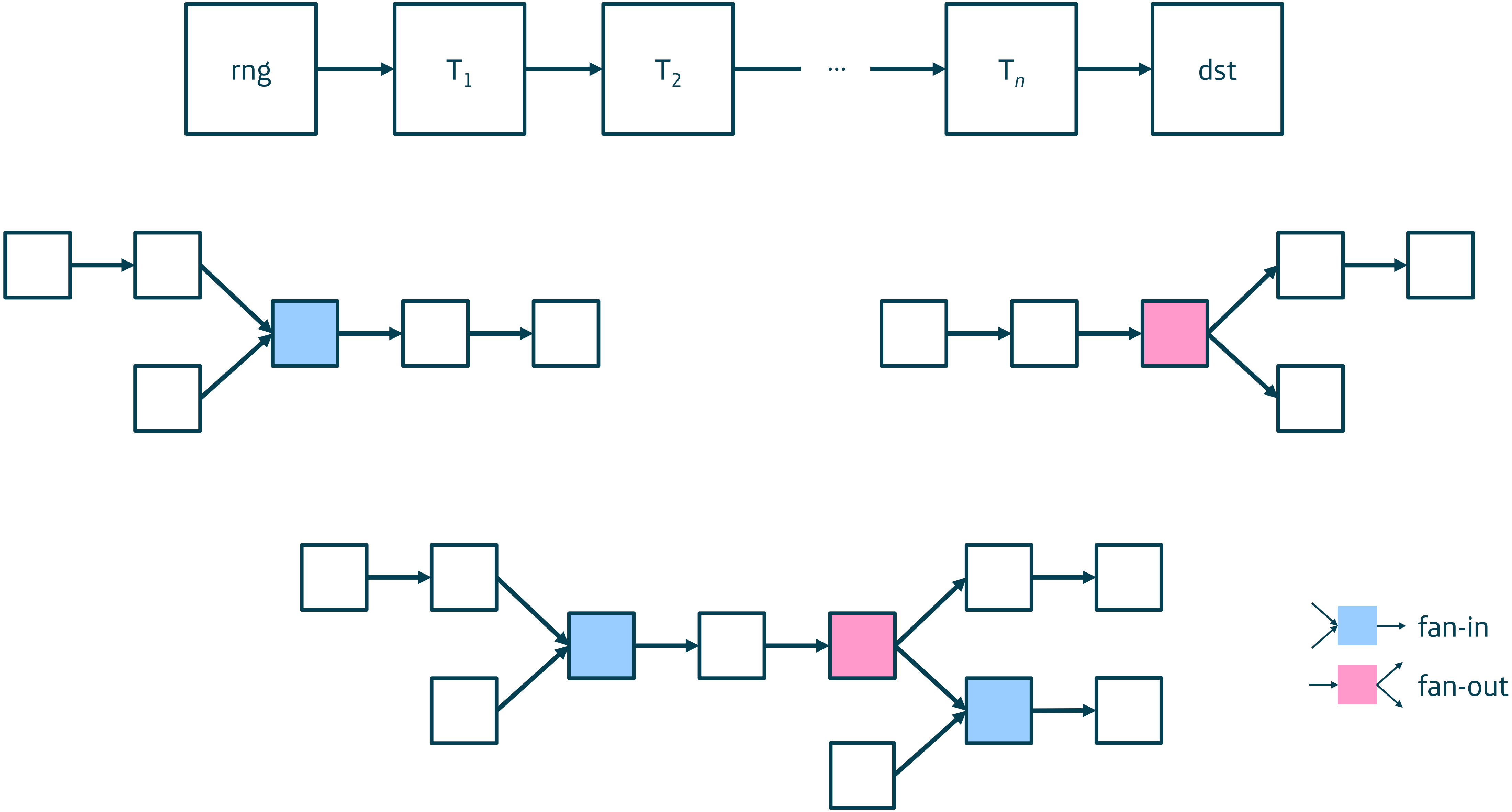
rng → filter → transform → dst

{0, 1, 2, 3, 4}

{0, 2, 4}

{0, 6, 12}

DAGs, fan-in, fan-out



Push or pull

Key differece

Push passes values

```
for(auto x: rng) {
  if(is_even(x)) dst(x3(x));
}
```

```
auto filter = [](auto pred, auto next) {
  return [=](auto x) { if(pred(x)) next(x); };
};
auto transform = [](auto f, auto next) {
  return [=](auto x) { next(f(x)); };
};

for(auto x: rng) {
  filter(is_even, transform(x3, dst))(x);
}
```

Pull retrieves values

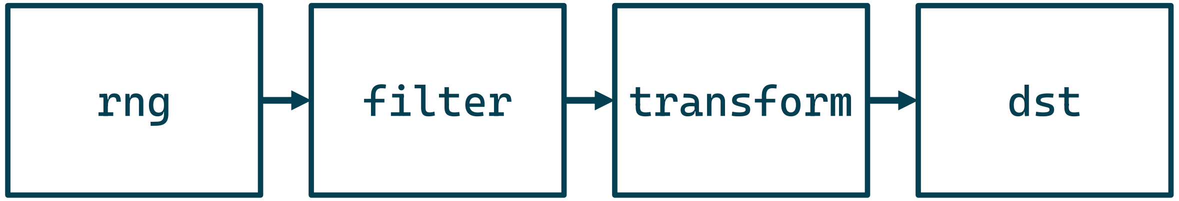
```
for (auto x:
     rng | filter(is_even) | transform(x3)) {
  dst(x);
}
```

```
for (auto x: transform(filter(rng, is_even), x3)) {
  dst(x);
}
```

Look for the control point

```
for(auto x: rng) {
  if(is_even(x)) dst(x3(x));
}
```

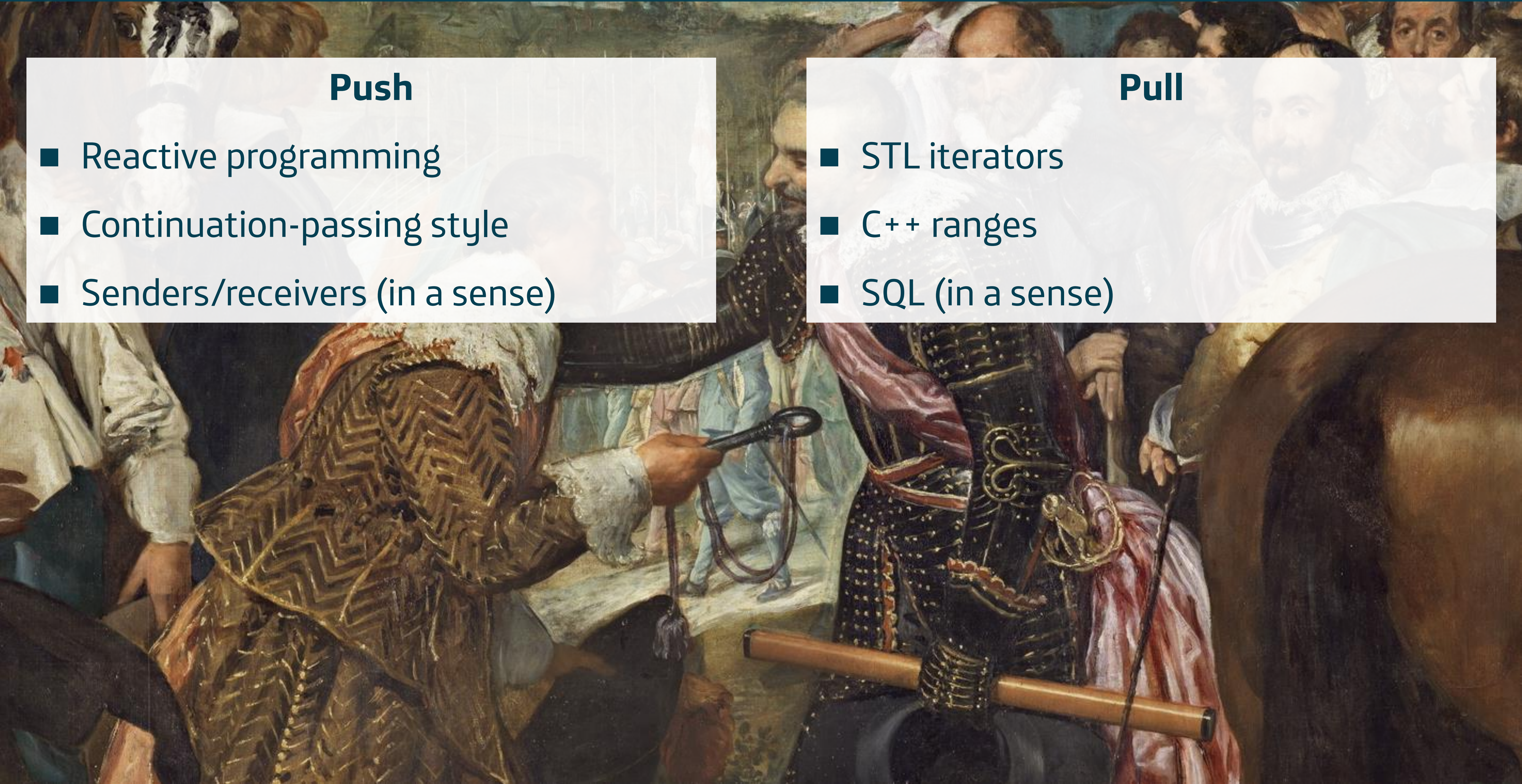
Push: control point at the beginning



Pull: control point at the end

```
for (auto x: rng | filter(is_even) | transform(x3)) {
  dst(x);
}
```

Some push and pull technologies



Push

- Reactive programming
- Continuation-passing style
- Senders/receivers (in a sense)

Pull

- STL iterators
- C++ ranges
- SQL (in a sense)

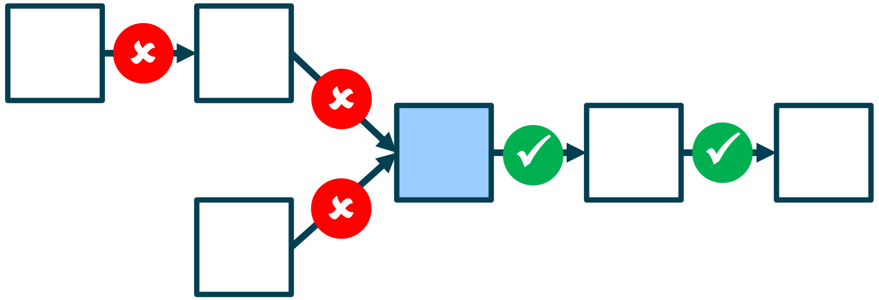
Push and pull are not equivalent

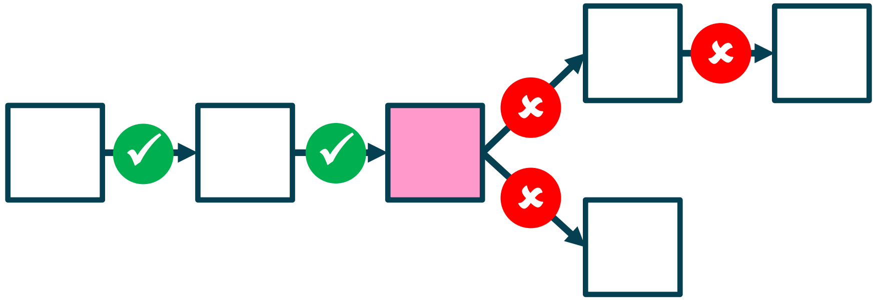
- All fan-in nodes must precede the control point
- All fan-out nodes must follow the control point

⇒ Push can't do DAGs with fan-in

⇒ Pull can't do DAGs with fan-out

⇒ Some DAGs are neither pushable nor pullable



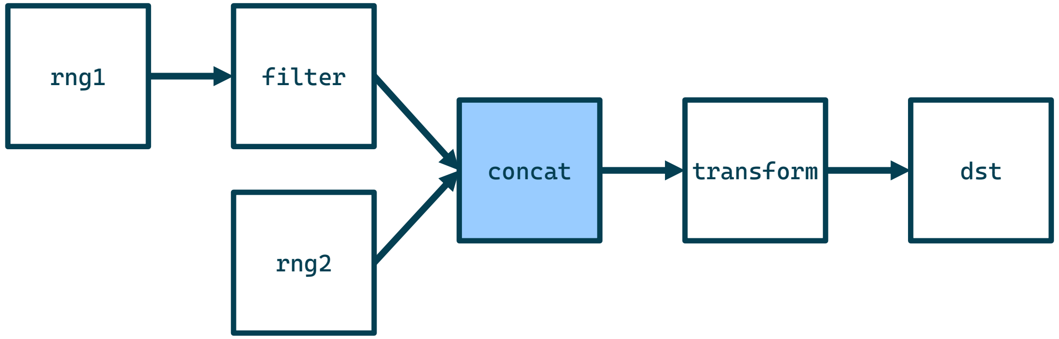


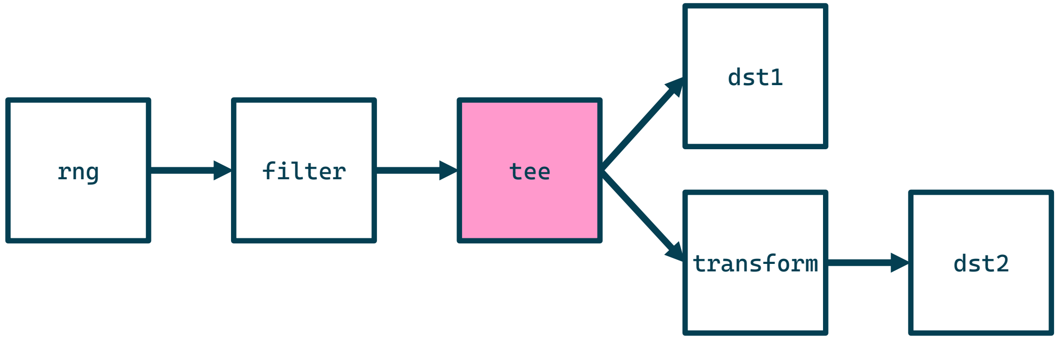
Push and pull are not equivalent

```
for (auto x:
     concat(rng1 | filter(is_even), rng2) |
     transform(x3)) {
  dst(x);
}
```

```
for(auto x: rng) {
  if(is_even(x)) {
    dst1(x);
    dst2(x3(x));
  }
}
```

```
for(auto x: rng) {
  filter(is_even,
         tee(dst1, transform(x3, dst2)))(x);
}
```





The curse of pull

```
std::vector<int> rng;
int res;

auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
void dst(int);

void push()
{
  for(auto x: rng) {
    if(is_even(x)) dst(x3(x));
  }
}

void pull()
{
  for (auto x:
       rng | filter(is_even) | transform(x3)) {
    dst(x);
  }
}
```

https://godbolt.org/z/o4KTKef59

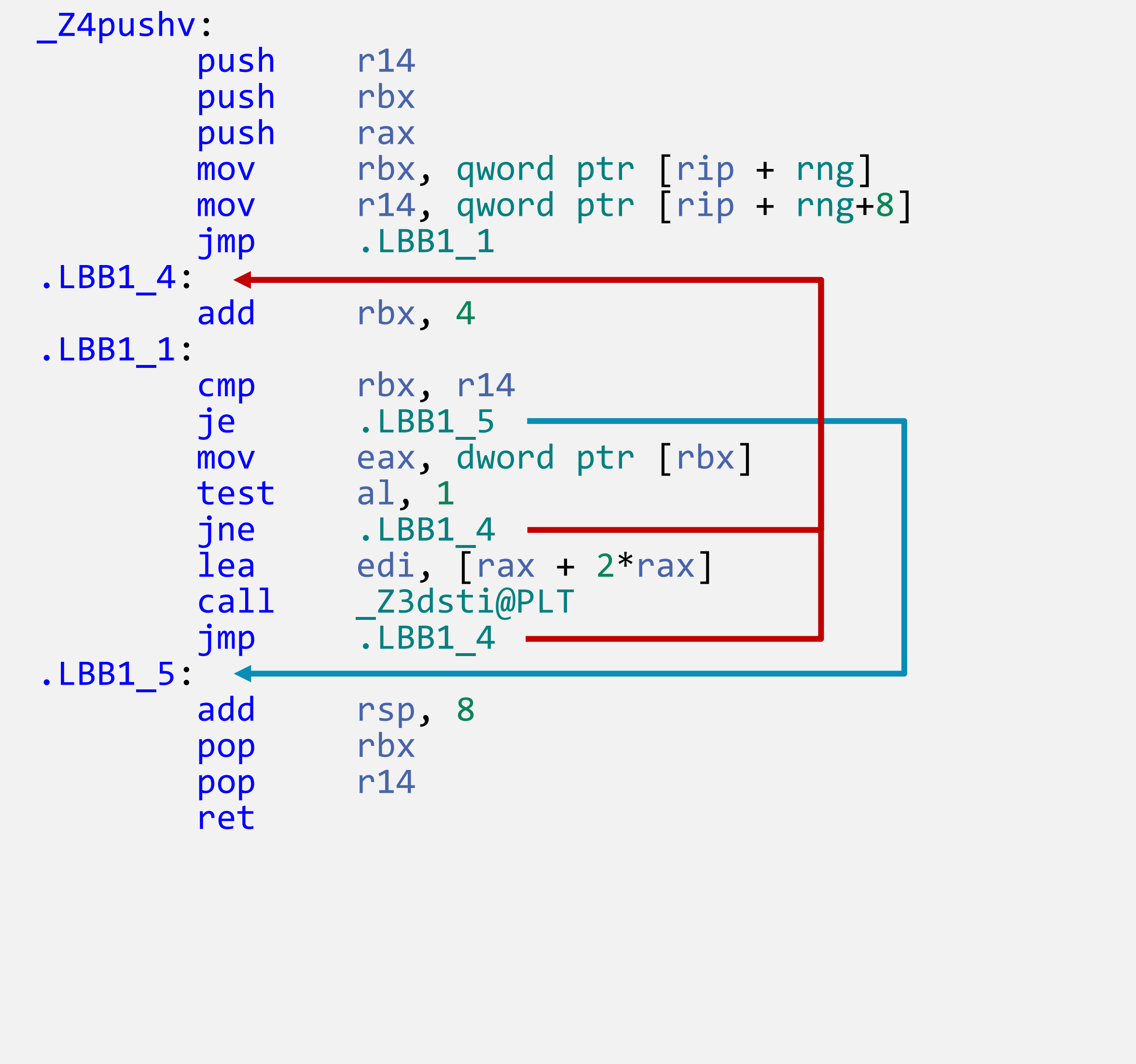
The curse of pull

```
std::vector<int> rng;
int res;

auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
void dst(int);

void push()
{
  for(auto x: rng) {
    if(is_even(x)) dst(x3(x));
  }
}

void pull()
{
  for (auto x:
       rng | filter(is_even) | transform(x3)) {
    dst(x);
  }
}
```



```
_Z4pushv:
        push    r14
        push    rbx
        push    rax
        mov     rbx, qword ptr [rip + rng]
        mov     r14, qword ptr [rip + rng+8]
        jmp     .LBB1_1
.LBB1_4:
        add     rbx, 4
.LBB1_1:
        cmp     rbx, r14
        je      .LBB1_5
        mov     eax, dword ptr [rbx]
        test    al, 1
        jne     .LBB1_4
        lea     edi, [rax + 2*rax]
        call    _Z3dsti@PLT
        jmp     .LBB1_4
.LBB1_5:
        add     rsp, 8
        pop     rbx
        pop     r14
        ret
```

https://godbolt.org/z/o4KTKef59

The curse of pull

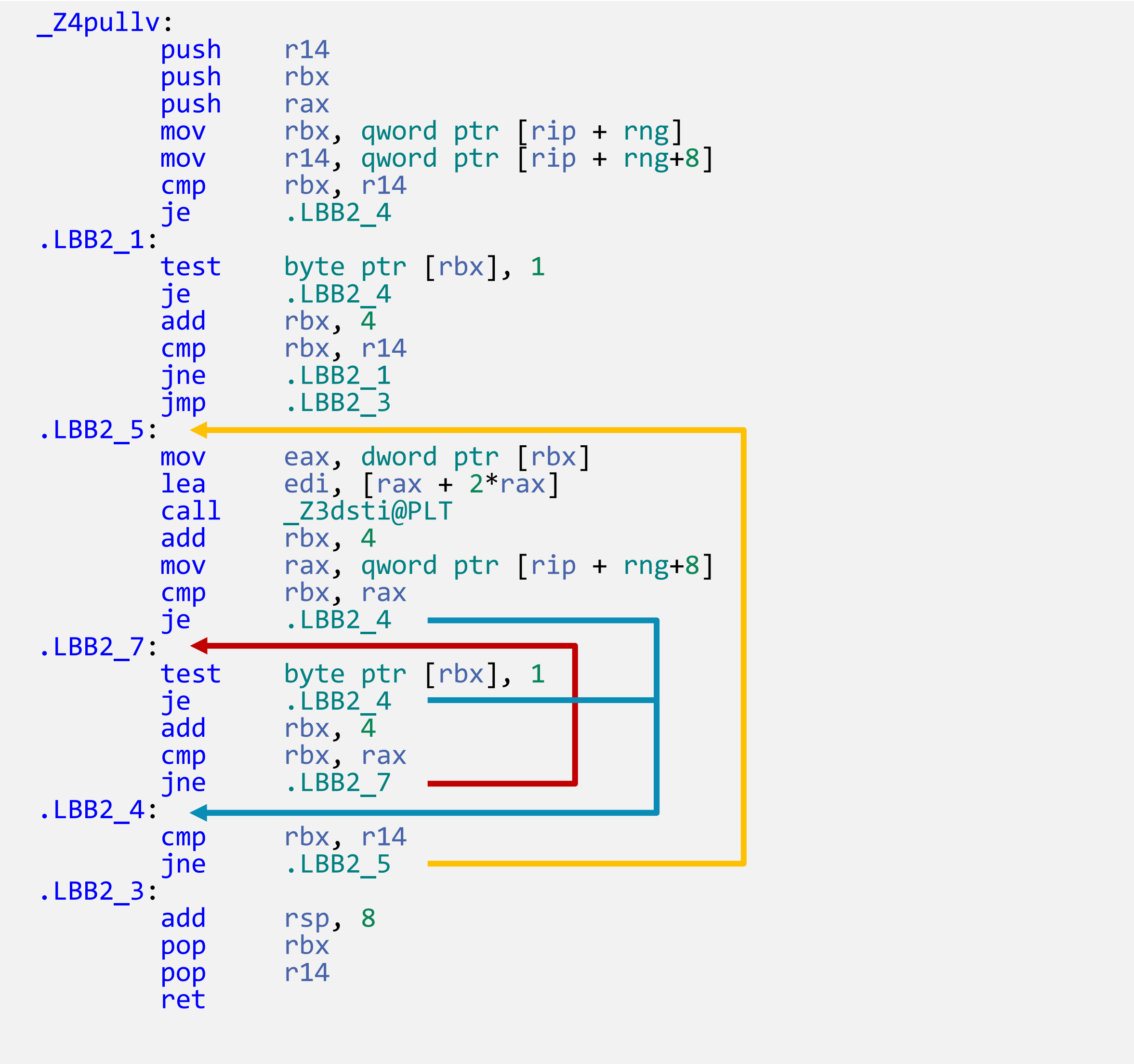
```
std::vector<int> rng;
int res;

auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
void dst(int);

void push()
{
  for(auto x: rng) {
    if(is_even(x)) dst(x3(x));
  }
}

void pull()
{
  for (auto x:
       rng | filter(is_even) | transform(x3)) {
    dst(x);
  }
}
```

https://godbolt.org/z/o4KTKef59



```
_Z4pullv:
        push    r14
        push    rbx
        push    rax
        mov     rbx, qword ptr [rip + rng]
        mov     r14, qword ptr [rip + rng+8]
        cmp     rbx, r14
        je      .LBB2_4
.LBB2_1:
        test    byte ptr [rbx], 1
        je      .LBB2_4
        add     rbx, 4
        cmp     rbx, r14
        jne     .LBB2_1
        jmp     .LBB2_3
.LBB2_5:
        mov     eax, dword ptr [rbx]
        lea     edi, [rax + 2*rax]
        call    _Z3dsti@PLT
        add     rbx, 4
        mov     rax, qword ptr [rip + rng+8]
        cmp     rbx, rax
        je      .LBB2_4
.LBB2_7:
        test    byte ptr [rbx], 1
        je      .LBB2_4
        add     rbx, 4
        cmp     rbx, rax
        jne     .LBB2_7
.LBB2_4:
        cmp     rbx, r14
        jne     .LBB2_5
.LBB2_3:
        add     rsp, 8
        pop     rbx
        pop     r14
        ret
```

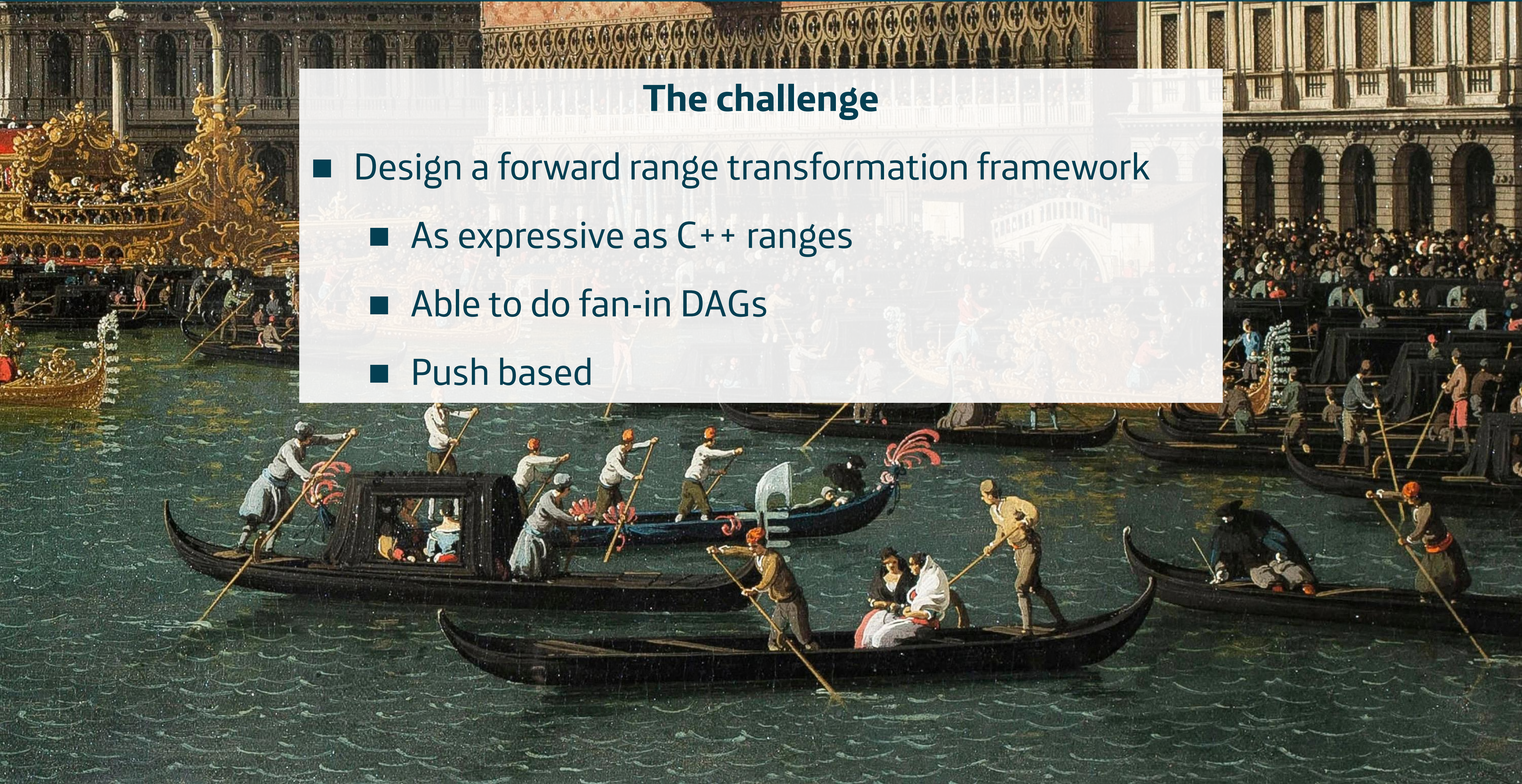
The curse of pull

```
auto v = filter(rng, is_even);
auto first = v.begin();
auto last = v.end();

while(first != last)
{
  dst(*first);
  ++first;
}
```

- Transformation is not 1:1 → double end-check
- `concat`, `join`, etc. have problems of their own
  - Context must be restored on each iteration
  - Fat iterators
- Compilers don't always see through all this
  - Much better at optimizing argument passing → context kept on call stack
  - Inversion of control → push

Transrangers



The challenge

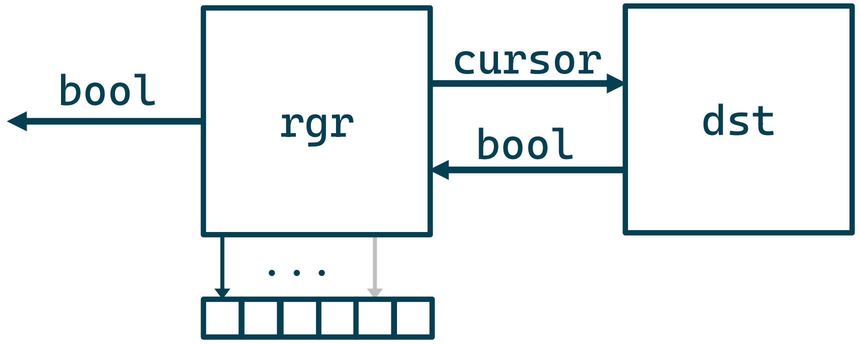
- Design a forward range transformation framework
  - As expressive as C++ ranges
  - Able to do fan-in DAGs
  - Push based

Concepts

```
template<typename Cursor>
concept cursor = requires(Cursor p)
{
  { *p } -> can_reference;
};

template<typename Dst, typename Cursor>
concept destination_function = requires(Dst dst, Cursor p)
{
  { dst(p) } -> std::convertible_to<bool>;
};

template<typename Ranger,typename Dst>
concept ranger = requires(Ranger rgr, Dst dst)
{
  typename Ranger::cursor;
  requires cursor<typename Ranger::cursor>;
  requires destination_function<Dst, typename Ranger::cursor>;
  { rgr(dst) } -> std::convertible_to<bool>;
};
```



- Traverses a range
- Passes cursors to **`dst`** while it returns **`true`**
- Returns **`false`** if there are still elements to process
- May process the range with several destination functions

Concepts

```
#include <transrangers.hpp>

using namespace transrangers;

auto rgr = all(rng); // plain ranger over range rng
auto b = rgr([n = 10](auto p) mutable { // print 10 elements
  std::cout<< *p << " ";
  return --n != 0;
});
if(!b) { // still elements in the range
  rgr([](auto p) { // print remaining in brackets
    std::cout<< "[" << *p << "] ";
    return true;
  });
}
```

- Push based
- Yet control is done at the end point!

Concepts

```
using namespace std::views;

for (auto x: transform(filter(rng, is_even), x3)) {
  dst(x);
}
```

```
using namespace transrangers;

transform(x3, filter(is_even, all(rng)))([](auto p){
  dst(*p);
  return true;
});
```

- A *transranger* takes a ranger and returns an adapted ranger
- They compose the same way as C++ range views (last to first)

Clojure	C++ ranges	Transrangers
reducer	view	ranger
transducer	range adaptor	transranger

Some transrangers

```
template<typename Pred,typename Ranger>
auto filter(Pred pred,Ranger rgr)
{
  using cursor=typename Ranger::cursor;

  return ranger<cursor>(
    [=](auto dst) mutable {
    return rgr([&](const auto& p) {
      return pred(*p)?dst(p):true;
    });
  });
}
```

```
template<typename Ranger>
auto unique(Ranger rgr)
{
  using cursor=typename Ranger::cursor;

  return ranger<cursor>(
    [=,start=true,p=cursor{}](auto dst) mutable {
    if(start){
      start=false;
      if(rgr([&](const auto& q) {
        p=q;
        return false;
      }))return true;
      if(!dst(p))return false;
    }
    return rgr([&,prev=p](const auto& q) mutable {
      if((*prev==*q)||dst(q)){prev=q;return true;}
      else{p=q;return false;}
    });
  });
}
```

```
template<typename F,typename Ranger>
auto transform(F f,Ranger rgr)
{
  using cursor=deref_fun<typename Ranger::cursor,F>;

  return ranger<cursor>([=](auto dst) mutable {
    return rgr([&](const auto& p) {
      return dst(cursor{p,&f});
    });
  });
}
```

```
template<typename Ranger>
auto concat(Ranger rgr)
{
  return rgr;
}

template<typename Ranger,typename... Rangers>
auto concat(Ranger rgr,Rangers... rgrs)
{
  using cursor=typename Ranger::cursor;

  return ranger<cursor>(
    [=,cont=false,next=concat(rgrs...)]
    (auto dst) mutable {
      if(!cont){
        if(!(cont=rgr(dst)))return false;
      }
      return next(dst);
    }
  );
}
```

Some transrangers

- Currently provided by `<transrangers.hpp>`
  - `filter`, `transform`, `take`, `concat`, `unique`, `join`, `zip`
- Theorem: for any (forward) range adaptor `ra` an equivalent transranger `tr` can be written
  - Proof: too long to fit in this slide
  - Corollary: transrangers are as expressive as C++ (forward) range adaptors

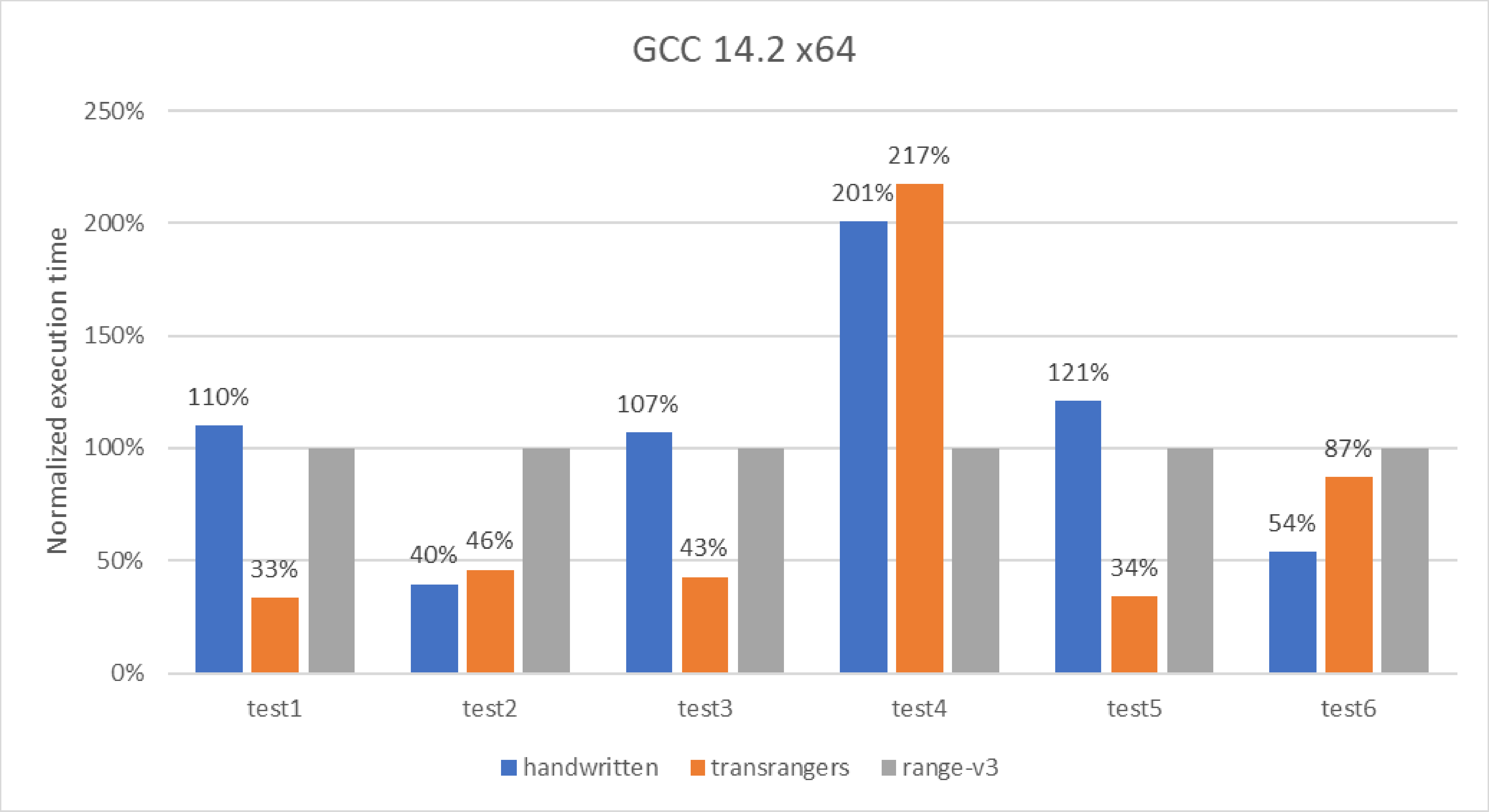
Performance

Performance tests

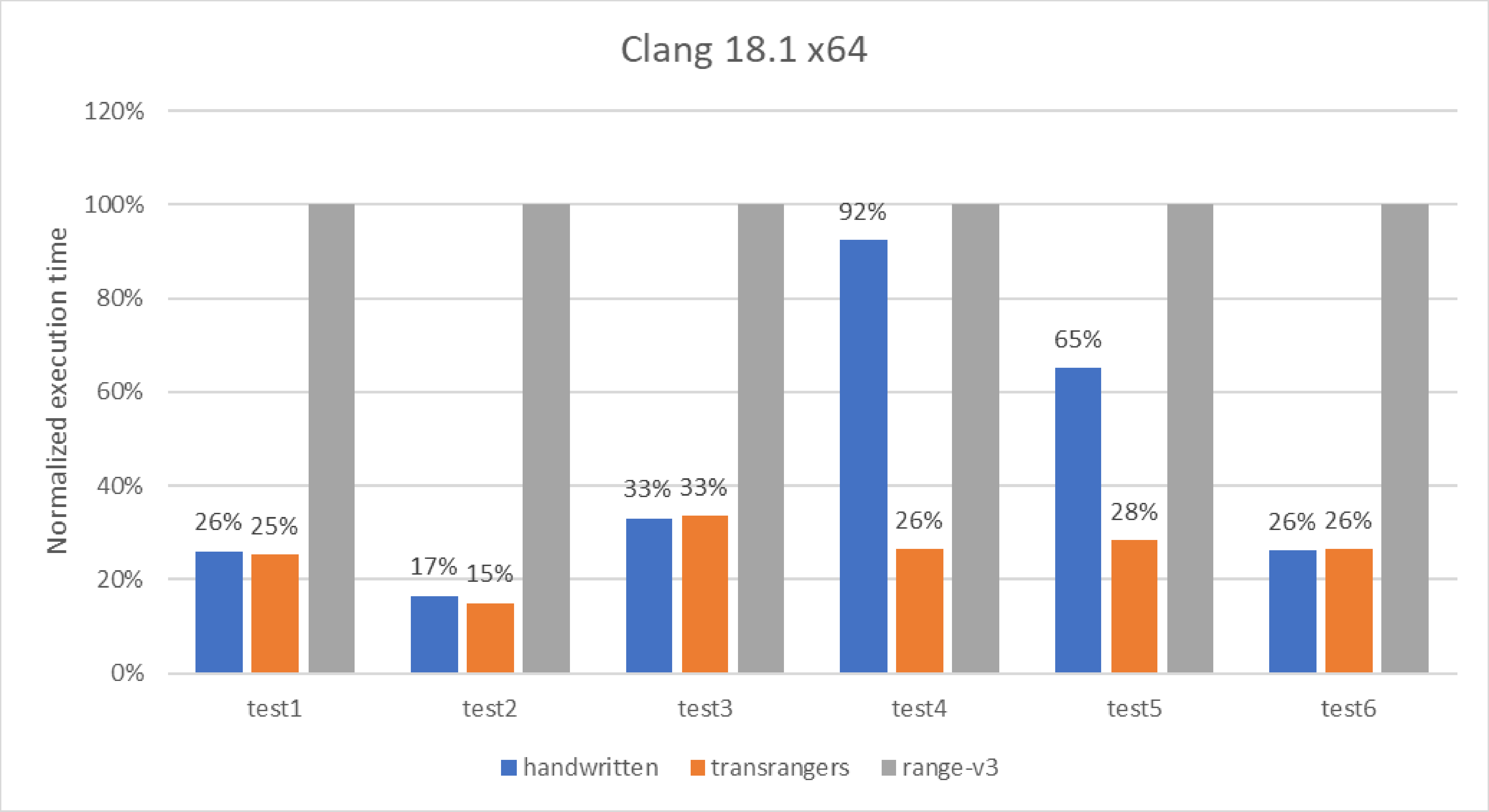
Name	Operation	Data
test1	`filter|transform`	1M integers
test2	`concat|take(1.5M)|filter|transform`	two vectors of 1M integers each
test3	`unique|filter`	100k integers
test4	`join|unique|filter|transform`	collection of 10 vectors of 100k integers each
test5	`transform(unique)|join|filter|transform`	collection of 10 vectors of 100k integers each
test6	`zip(·,·|transform)|transform(sum)|filter`	two vectors of 1M integers each

- Run as GitHub Actions
- GCC 14.2 x64 Ubuntu 24.04
- Clang 18.1 x64 Ubuntu 24.04
- Visual Studio 2022 19.43 x64 Microsoft Windows Server 10
- Clang 18.1 ARM64 macOS 14.7

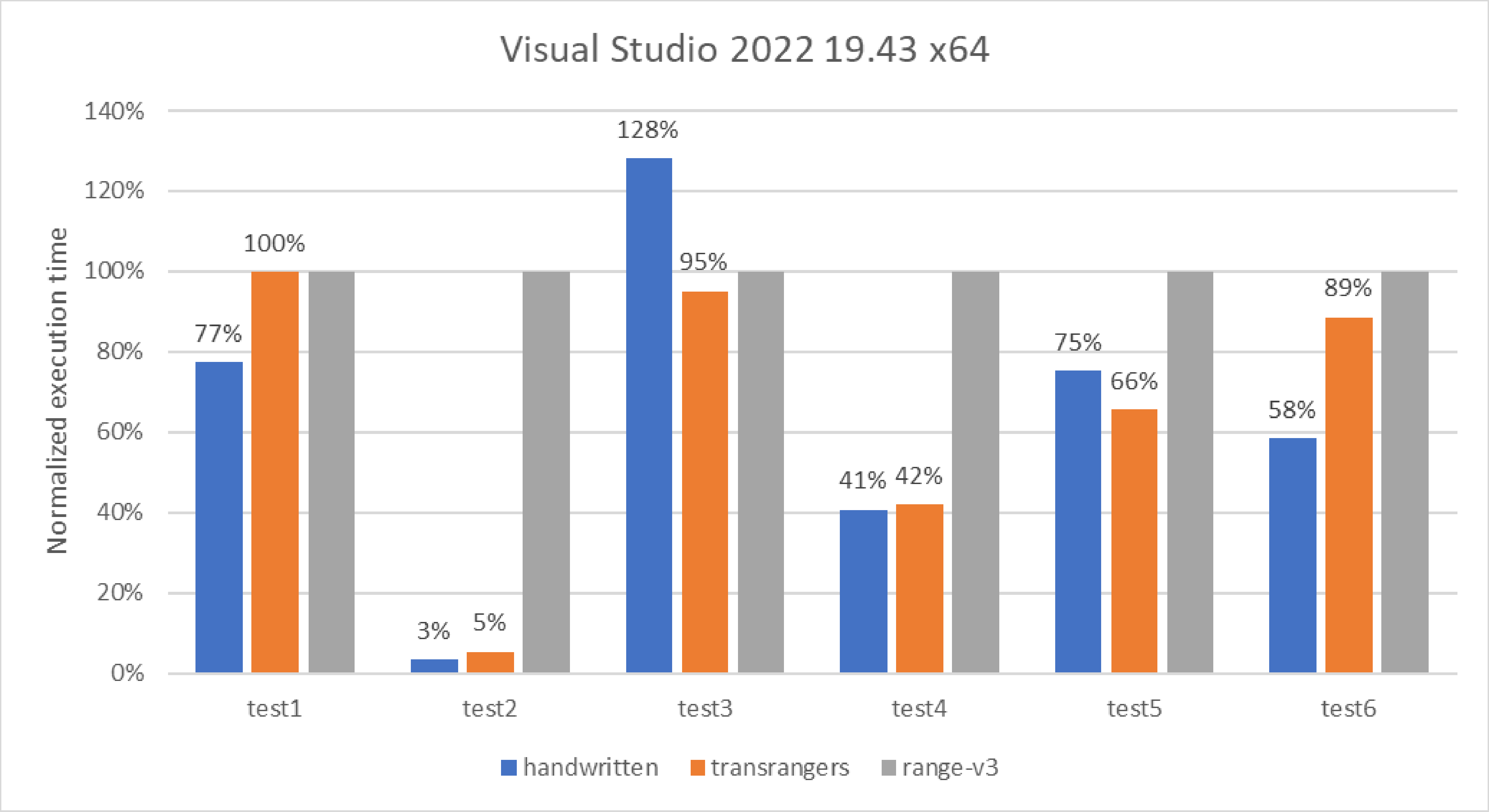
Results


GCC 14.2 x64
Normalized execution time
250%
200%
150%
100%
50%
0%
110%
33%
40%
46%
107%
43%
201%
217%
121%
34%
54%
87%
test1
test2
test3
test4
test5
test6
handwritten
transrangers
range-v3

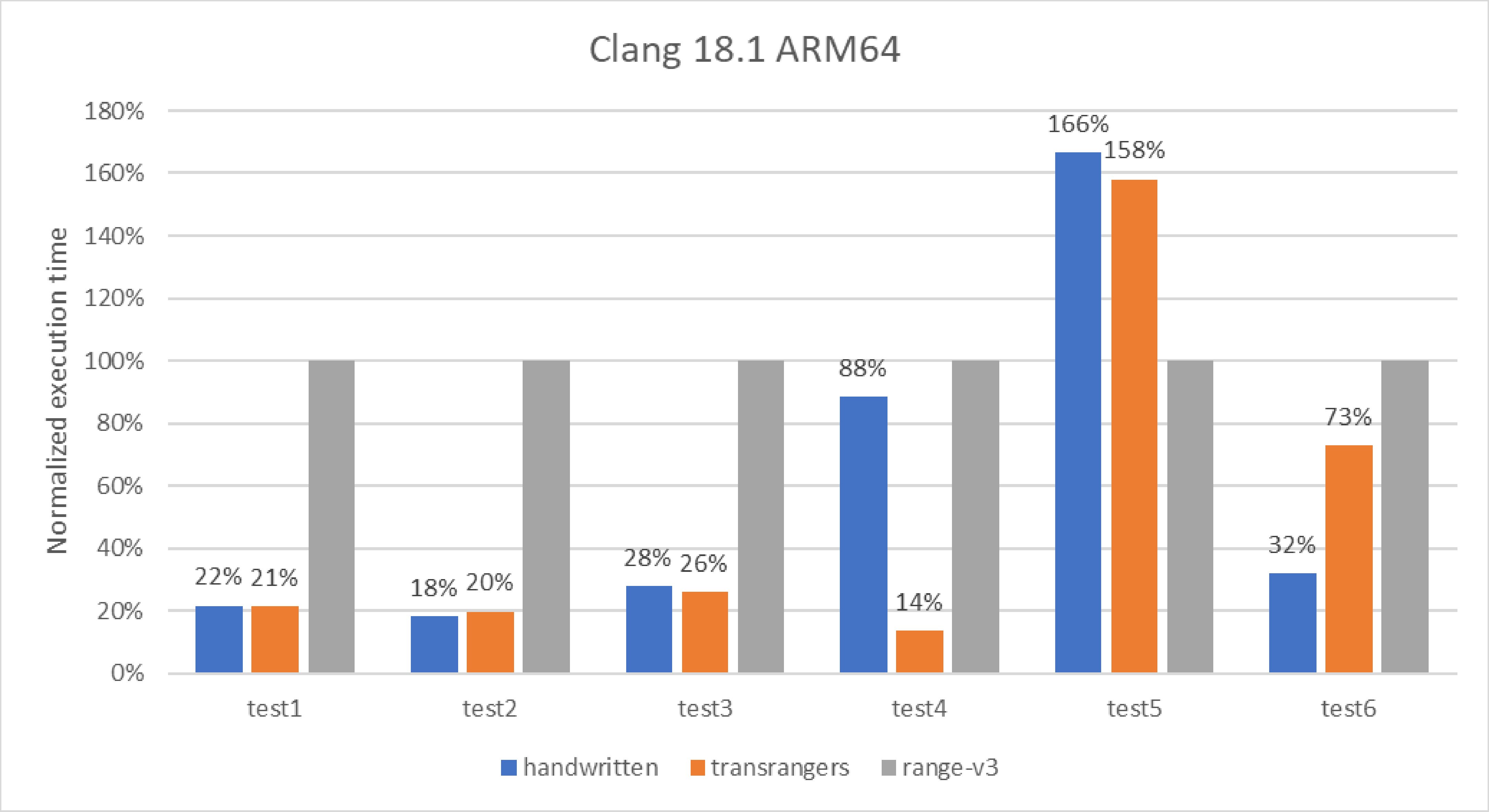
Results


Clang 18.1 x64
Normalized execution time
120%
100%
80%
60%
40%
20%
0%
26%
25%
17%
15%
33%
33%
92%
26%
65%
28%
26%
26%
test1
test2
test3
test4
test5
test6
■ handwritten
■ transrangers
■ range-v3

Results


Visual Studio 2022 19.43 x64
Normalized execution time
140%
120%
100%
80%
60%
40%
20%
0%
77%
100%
3%
5%
128%
95%
41%
42%
75%
66%
58%
89%
test1
test2
test3
test4
test5
test6
■ handwritten
■ transrangers
■ range-v3

Results

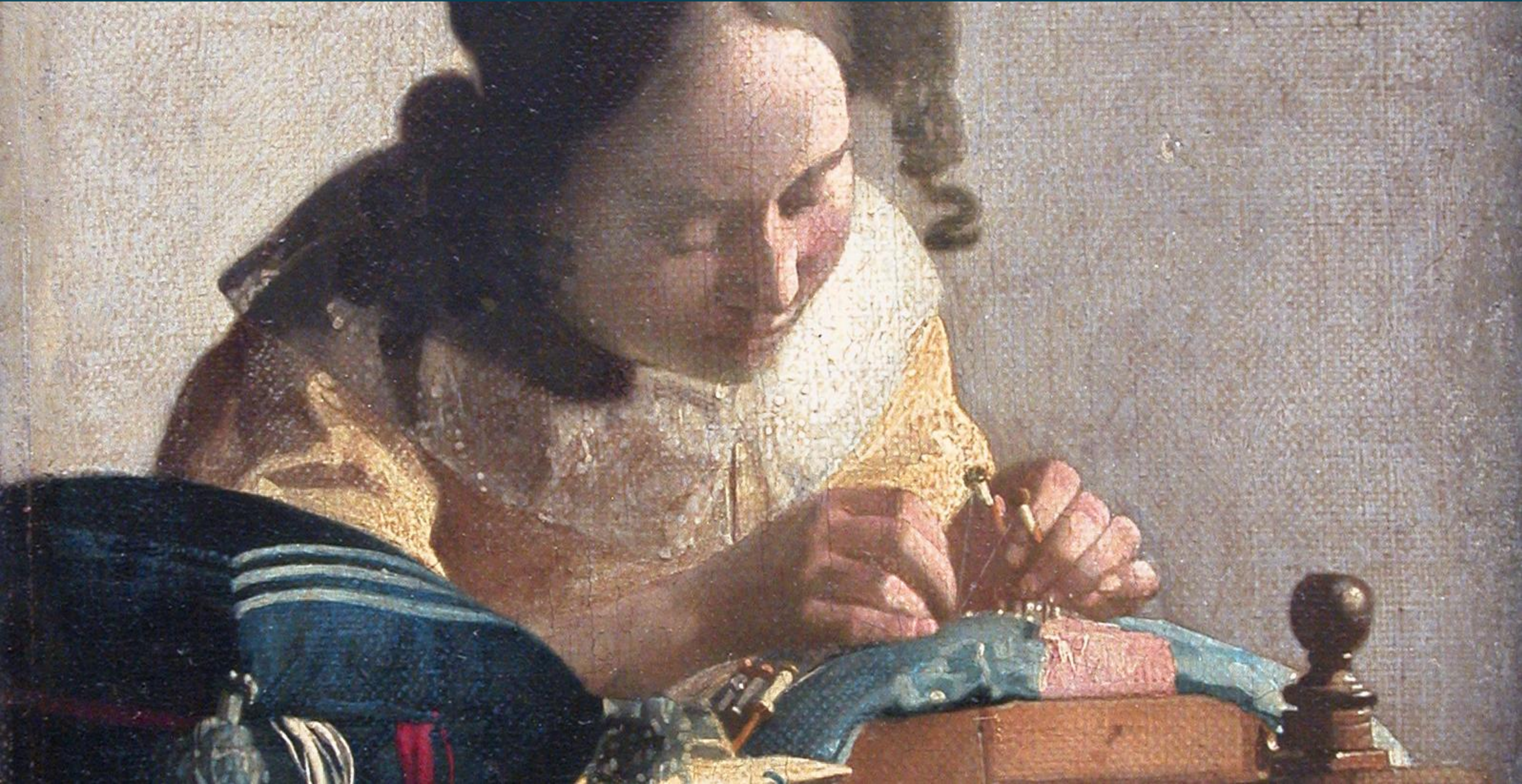

Clang 18.1 ARM64
Normalized execution time
180%
160%
140%
120%
100%
80%
60%
40%
20%
0%
22%
21%
18%
20%
28%
26%
88%
14%
166%
158%
32%
73%
test1
test2
test3
test4
test5
test6
handwritten
transrangers
range-v3

What from here

What we've learnt

- Transrangers are a faster alternative to C++ (forward) range adaptors
- Could be used as a transparent backend implementation for C++ ranges
  - Contact your local stdlib provider
- Similar approaches
  - Barry Revzin's Rivers: https://github.com/brevzin/rivers
  - Rust's `try_for_each`
    - Not for the reasons you may think

Coroutines



Using std::generator for range transformation

```cpp
#include <generator>
#include <iostream>
#include <vector>

std::vector<int> rng={0,1,2,3,4};

auto is_even = [](auto x) { return x%2 == 0; };
auto x3 =      [](auto x) { return x*3; };
auto dst =     [](auto x) { std::cout << x << " "; };

template<typename Range>
std::generator<typename Range::value_type>
all(Range& rng)
{
  for(auto x: rng) co_yield x;
}

template<typename T, typename Pred>
std::generator<T>
filter(Pred pred, std::generator<T> rng)
{
  for(auto x: rng) if(pred(x)) co_yield x;
}
```

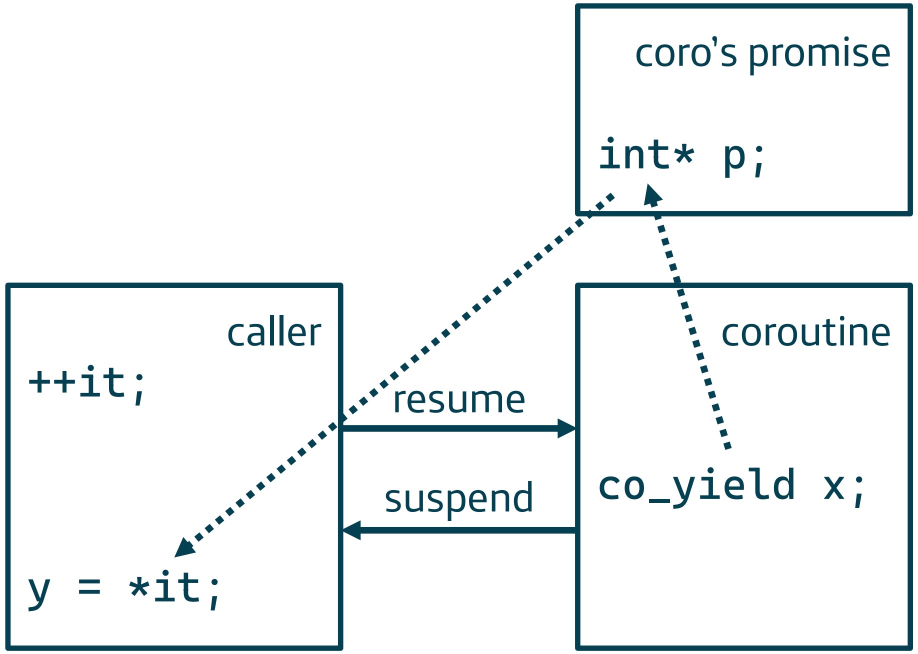
```cpp
template<typename T, typename F>
std::generator<T>
transform(F f, std::generator<T> rng)
{
  for(auto x: rng) co_yield f(x);
}

int main()
{
  for(auto x:
      transform(x3, filter(is_even, all(rng)))) {
    dst(x);
  }
}
```

https://godbolt.org/z/vMMY4TYnG

Are coroutines push or pull?

- Well, neither
  - `co_yield` looks like push
  - `co_await` looks like pull
  - Both boil down to coroutine suspension and value transfer through the coroutine's promise
- A `std::generator` is-a C++ range
  - So, it suffers also from the curse of pull
  - In the limit, it can only get as fast as C++ ranges

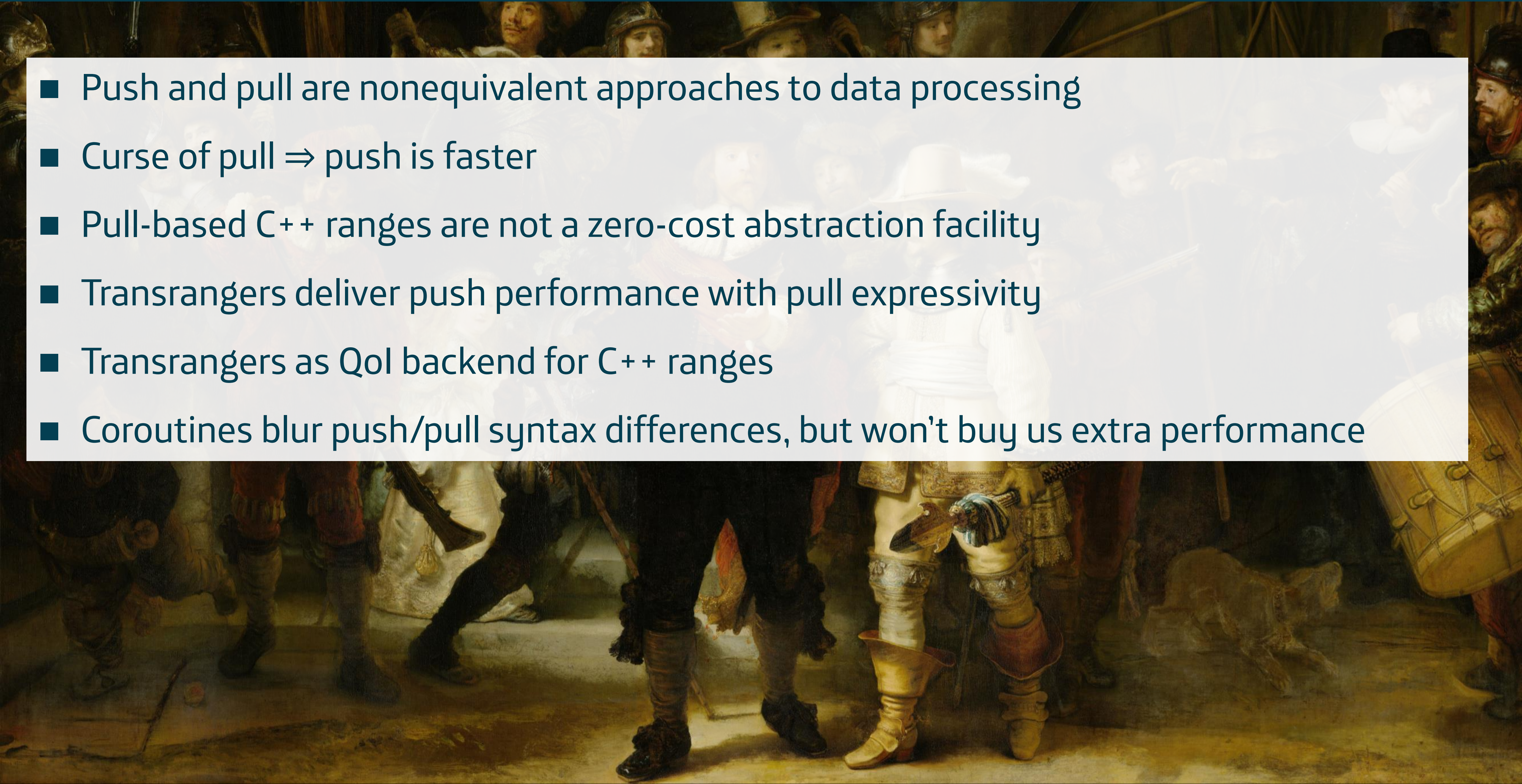


Coroutines and performance

- When stars align, coroutines can be as fast as regular code
  - Data exchange through promise ⇒ argument passing / value return
  - Frame dynamic allocation ⇒ use of stack (HALO)
  - Make coroutines local and scaffolding available for inlining (P0981R0, P1365R0)
- Ultimately, coroutines are not prime candidates for top performance
- But they provide incredible syntax (which is what they're about)

Calling it a night

Takeaways



- Push and pull are nonequivalent approaches to data processing
- Curse of pull ⇒ push is faster
- Pull-based C++ ranges are not a zero-cost abstraction facility
- Transrangers deliver push performance with pull expressivity
- Transrangers as QoI backend for C++ ranges
- Coroutines blur push/pull syntax differences, but won't buy us extra performance

Push is Faster

Thank you

github.com/joaquintides/usingstdcpp2025
github.com/joaquintides/transrangers

"Rain, Steam and Speed – The Great Western Railway", © 1844 J. M. W. Turner; "La rendición de Breda", © 1634-35 Diego Velázquez; "Il Bucintoro", © 1745-50 Giovanni Antonio Canal *Canaletto*; "Avant la représentation", © 1895-1900 Edgar Degas; "La Madeleine à la veilleuse", © c. 1640 Georges de La Tour; "De Kantwerkster", © 1669-70 Johannes Vermeer; "De Nachtwacht", © 1642 Rembrandt van Rijn

using std::cpp 2025
Joaquín M López Muñoz <joaquin.lopezmunoz@gmail.com>
Madrid, March 2025