METAPROGRAMMING FOR THE BRAVE

LOUIS DIONNE, C++NOW 2016

NO CUTE INTRO, THIS IS A SERIOUS TALK

LOGICAL OPERATIONS

```
template <typename ...Bools>
struct logical_and;

template <>
struct logical_and<> {
    static constexpr bool value = true;
};

template <typename Bool, typename ...Bools>
struct logical_and<Bool, Bools...> {
    static constexpr bool value = std::conditional<Bool::value,
                                      logical_and<Bools...>,
                                      std::false_type
                                  >::type::value;
};
```

AN EAGER VERSION

(ROLAND BOCK)

```
template <bool ...> struct bools;

template <typename ...Bools>
struct logical_and
    : std::is_same<
        bools<Bools::value...>,
        bools<(Bools::value, true)...>
    >
{ };
```

A SHORT CIRCUITING VERSION

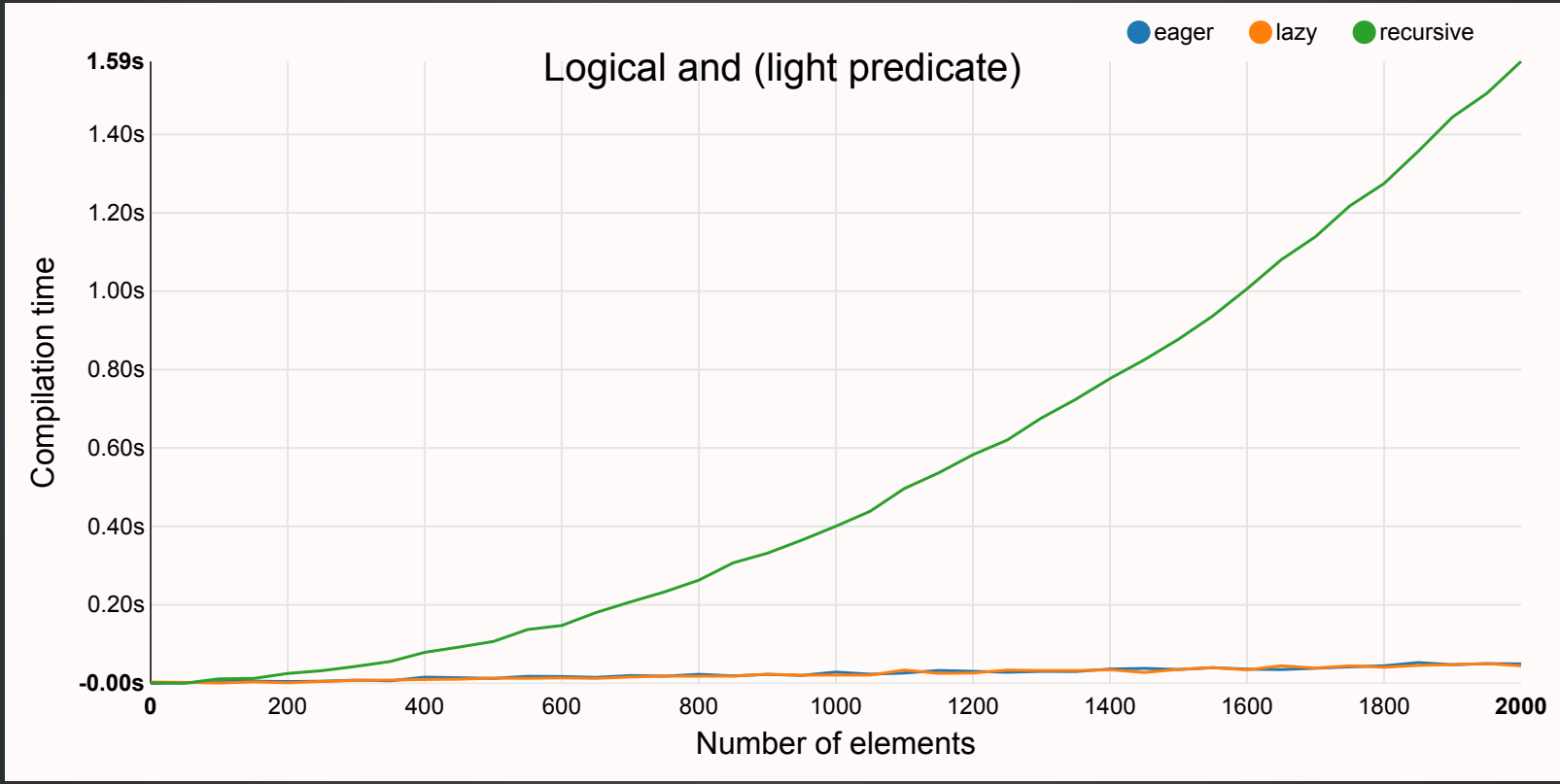
(ERIC FISELIER)

```
template <typename Condition, typename T = void>
struct lazy_enable_if
    : std::enable_if<Condition::value, T>
{ };

std::true_type expand(...);

template <typename ...Bools>
decltype(expand(typename lazy_enable_if<Bools, int>::type{}...))
and_impl(int);

template <typename ...Bools>
std::false_type and_impl(...);

template <typename ...Bools>
struct logical_and
    : decltype(and_impl<Bools...>(int{}))
{ };
```

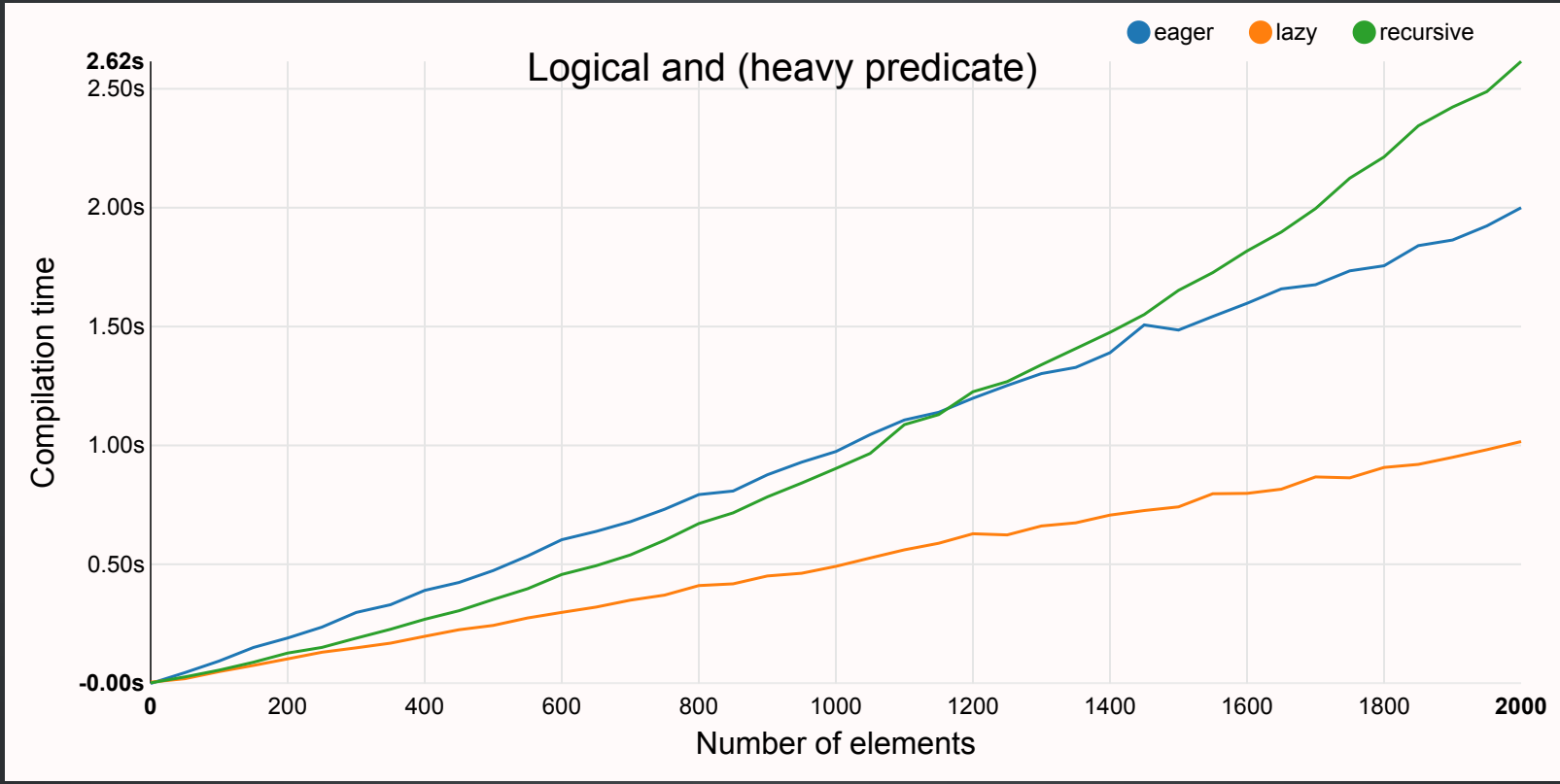
WHICH ONE IS FASTER?

NO IDEA, MUST BENCHMARK

WHEN THE CONDITIONS ARE LIGHT TO EVALUATE



WHEN THE CONDITIONS ARE HEAVY TO EVALUATE



TIP 1

PREFER VARIADIC EXPANSION TO RECURSION IN METAFUNCTIONS

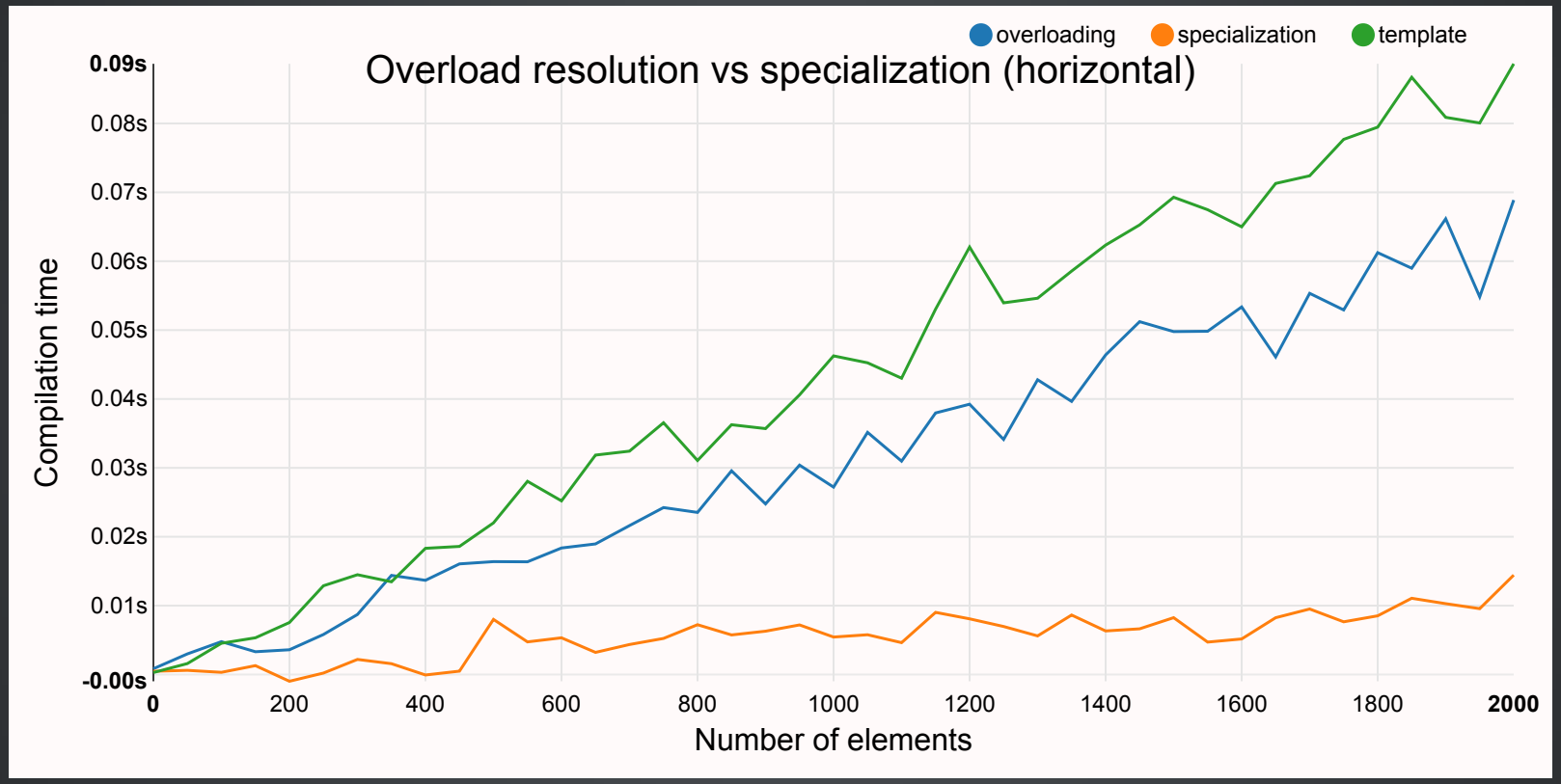
SPECIALIZATION VS OVERLOADING

HORIZONTAL

```
template <int> struct arg { };
void overloaded(arg<1>, ..., arg<n>) { }

int main() { overloaded(arg<1>{}, ..., arg<n>{}); }
```

VS

```
template <int>          struct arg { };
template <typename ...> struct specialized { };
template <>             struct specialized<arg<1>, ..., arg<n>> { };

int main() { specialized<arg<1>, ..., arg<n>>{}; }
```

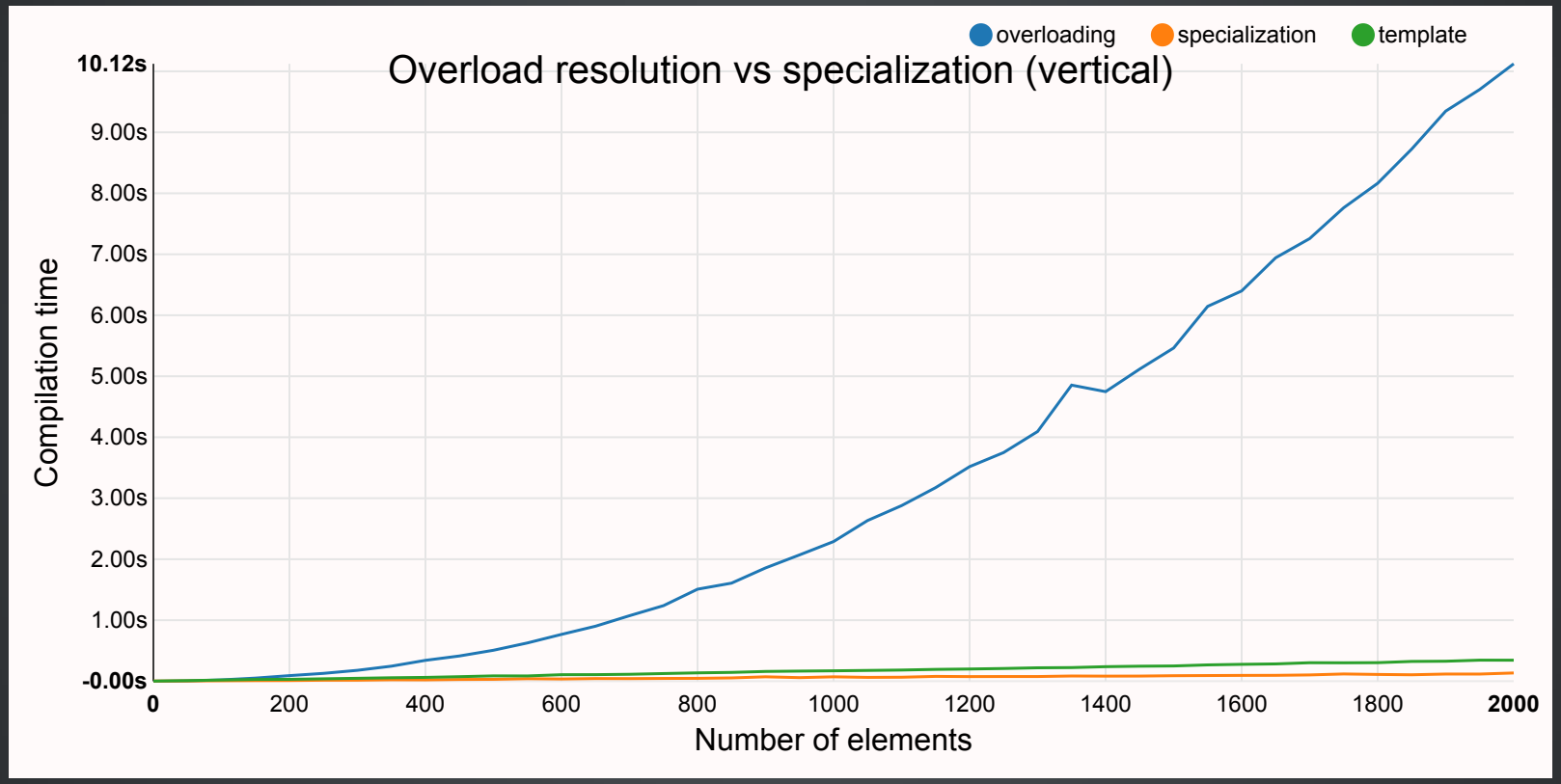

overloading
specialization
template
Overload resolution vs specialization (horizontal)
Compilation time
0.09s
0.08s
0.07s
0.06s
0.05s
0.04s
0.03s
0.02s
0.01s
-0.00s
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Number of elements

VERTICAL

```
template <int> struct arg { };
void overloaded(arg<1>) { }
...
void overloaded(arg<n>) { }

int main() {
    overloaded(arg<1>{});
    ...
    overloaded(arg<n>{});
}
```

VS

```
template <int>      struct arg { };
template <typename> struct specialized { };
template <>         struct specialized<arg<1>> { };
...
template <>         struct specialized<arg<n>> { };

int main() {
    specialized<arg<1>>{};
    ...
    specialized<arg<n>>{};
}
```


overloading
specialization
template
Overload resolution vs specialization (vertical)
10.12s
9.00s
8.00s
7.00s
6.00s
5.00s
4.00s
3.00s
2.00s
1.00s
-0.00s
Compilation time
0
200
400
600
800
1000
1200
1400
1600
1800
2000
Number of elements

TIP 2

AVOID OVERLOAD RESOLUTION WITH HUGE NUMBERS OF ARGUMENTS

LET'S TAKE A LOOK AT SYMBOL LENGTHS

EXAMPLE: EXPRESSION TEMPLATES

THANKS TO AGUSTIN BERGÉ FOR HELP ON THIS TEST CASE

```
template <typename F, typename Left, typename Right>
struct expr {
    F f; Left left; Right right;
    auto operator()() const { return f(left(), right()); }
};

template <typename Left, typename Right>
auto operator+(Left const& left, Right const& right) {
    return expr<std::plus<>, Left, Right>{std::plus<>{}, left, right};
}

template <int>
struct Leaf {
    int operator()() const { return 1; }
};

int main() {
    Leaf<1> a; Leaf<2> b; Leaf<3> c; Leaf<4> d;
    auto expr = (a + b) + (c + d);
    std::cout << expr() << std::endl;
}
```

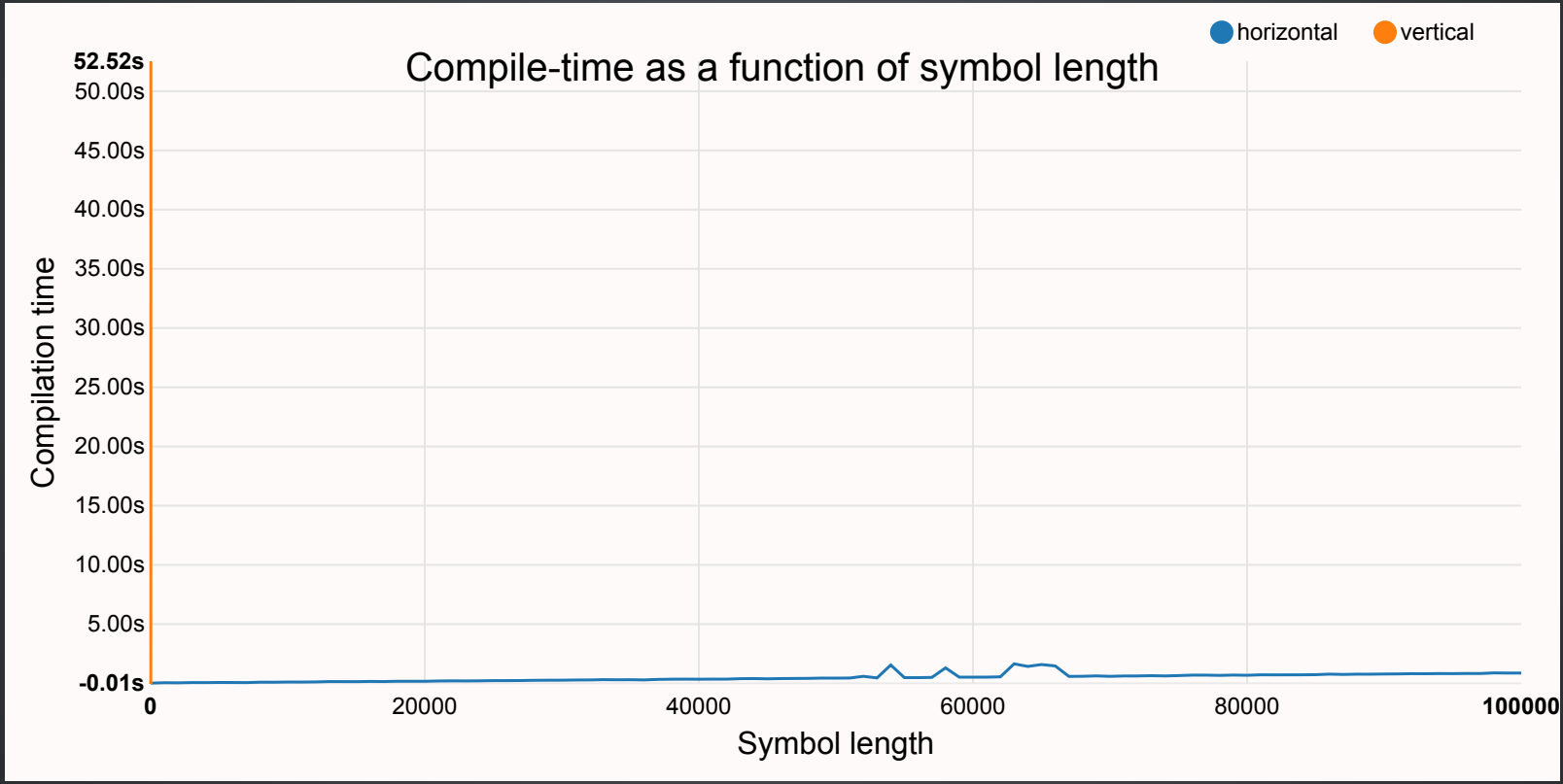
VERTICAL EXPLOSION

```
Leaf<1> exp0;
Leaf<2> exp1;
auto exp2 = exp0 + exp1;
auto exp3 = exp2 + exp2;
auto exp4 = exp3 + exp3;
...
auto expN = expN-1 + expN-1;

std::cout << expN() << std::endl;
```

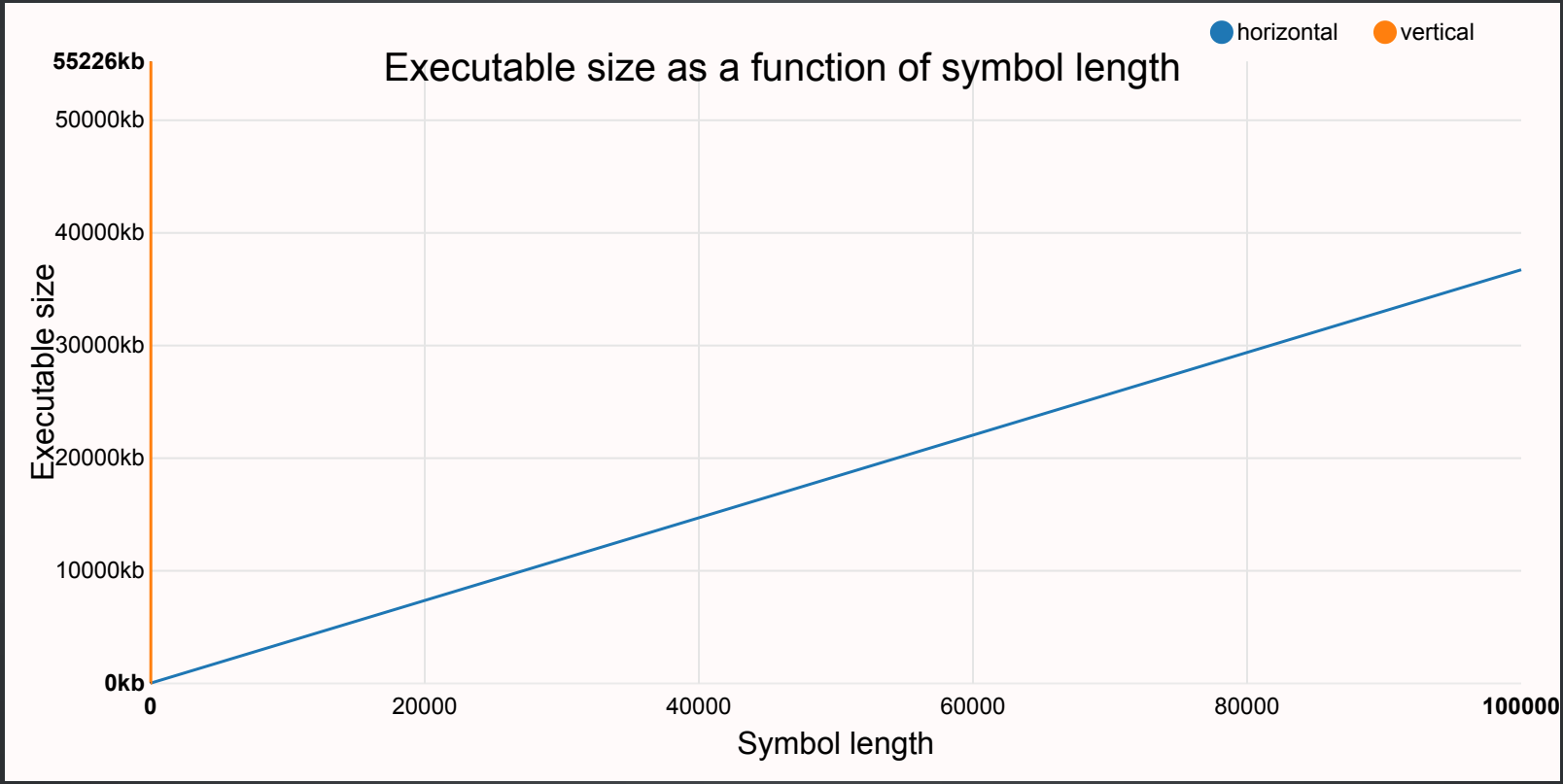
HORIZONTAL GROWTH

```
Leaf_symbol-of-length-n<1> exp0;
Leaf_symbol-of-length-n<2> exp1;
auto exp2 = exp0 + exp1;
auto exp3 = exp2 + exp2;
auto exp4 = exp3 + exp3;

std::cout << exp4() << std::endl;
```

FIRST PROBLEM: COMPILE-TIMES



SECOND PROBLEM: HUGE EXECUTABLES



BUT SURELY -O3 FIXES THAT?

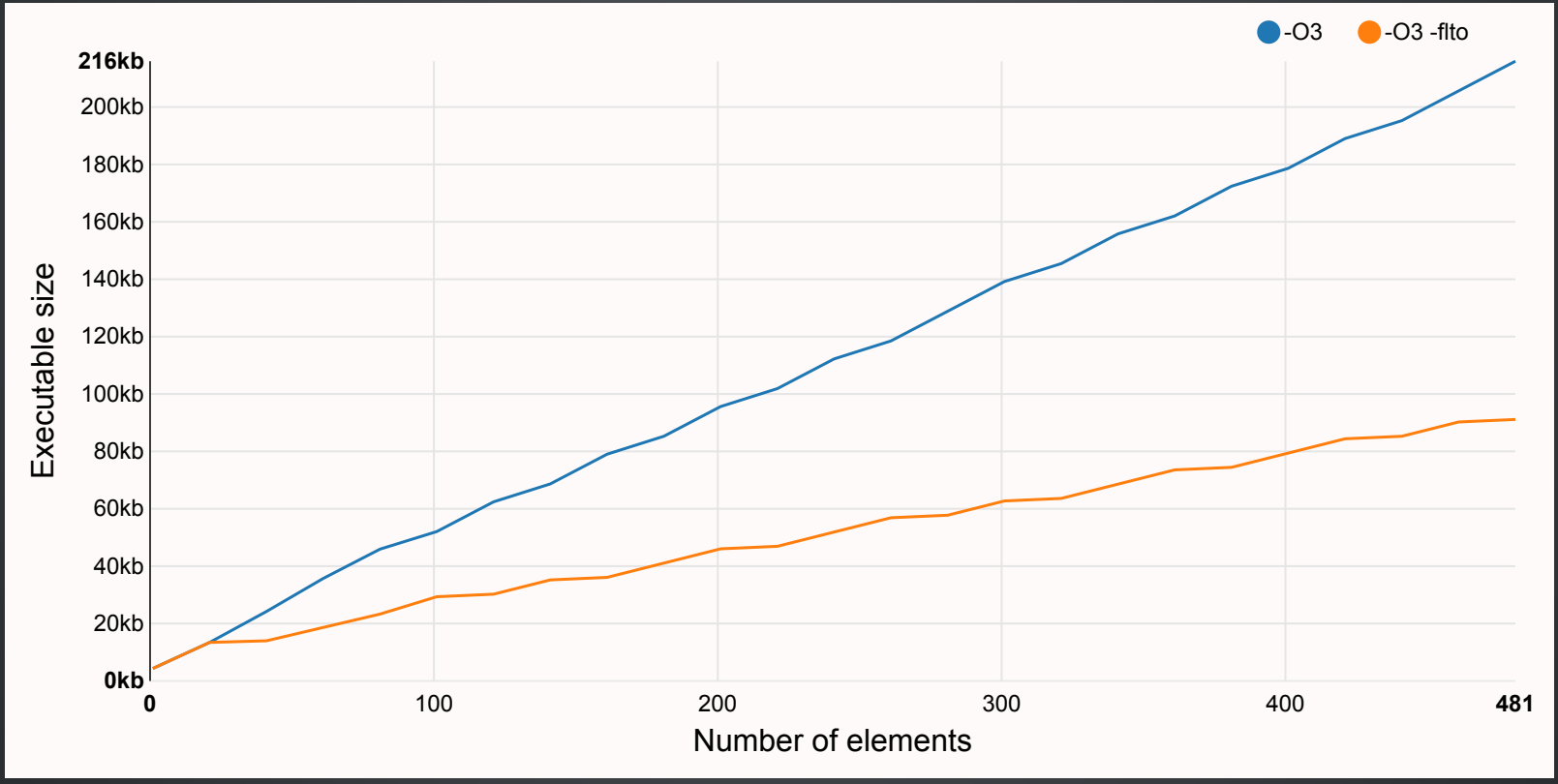
SOMETIMES, BUT NOT ALWAYS

THANKS TO JASON RICE FOR THIS TEST CASE

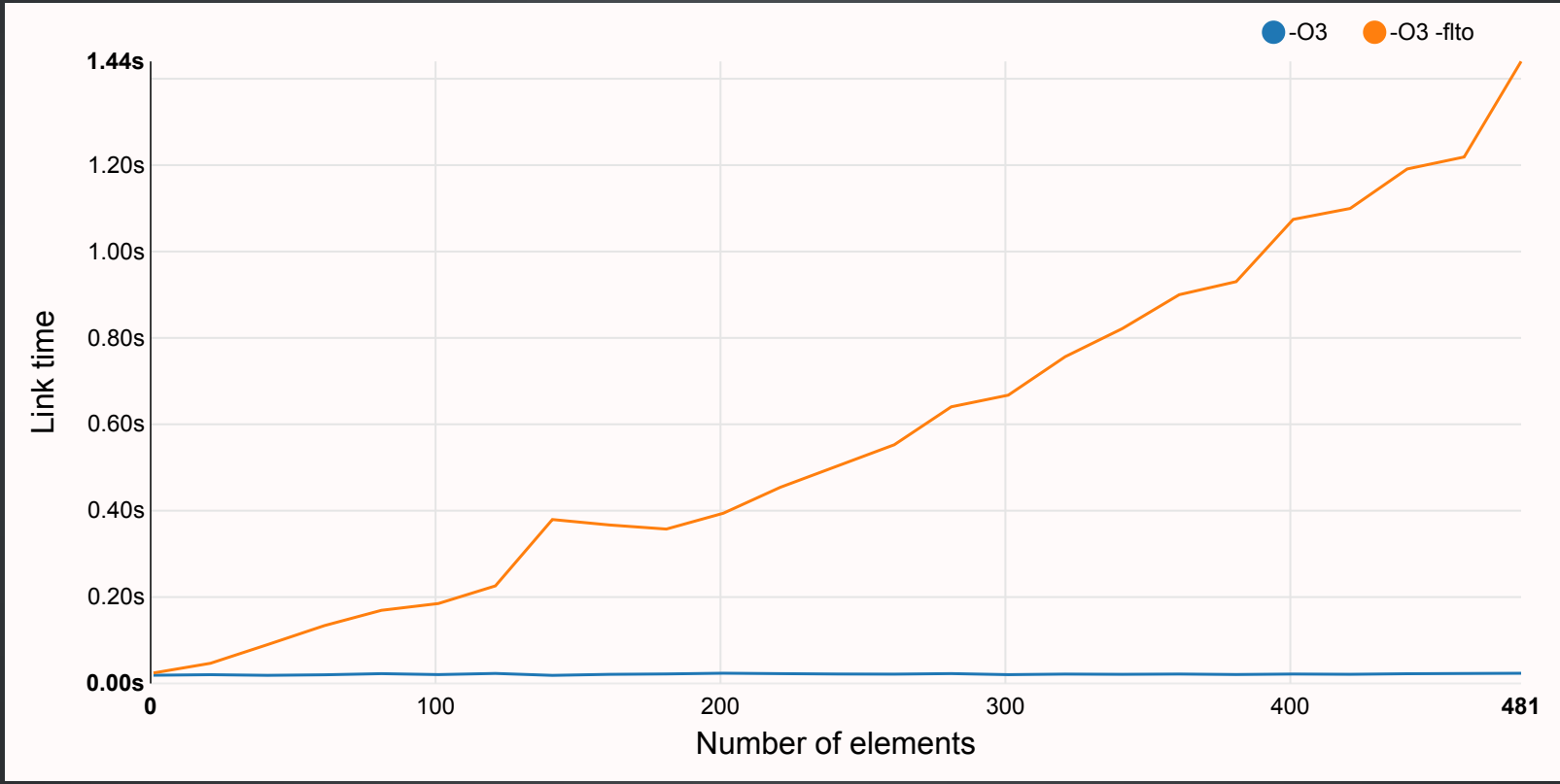
```
template <int i>
auto buildLargeMap() {
    return hana::unpack(hana::range_c<int, 0, i>, [](auto ...x) {
        return hana::make_map(
            hana::make_pair(x, std::vector<int>{})...
        );
    });
}

int main() {
    auto x = buildLargeMap<<%= n %>>();
}
```

HAVE TO USE LTO

(AND IT'S STILL PRETTY BLOATED)



OF COURSE, LTO IS NOT FREE



TIP 3

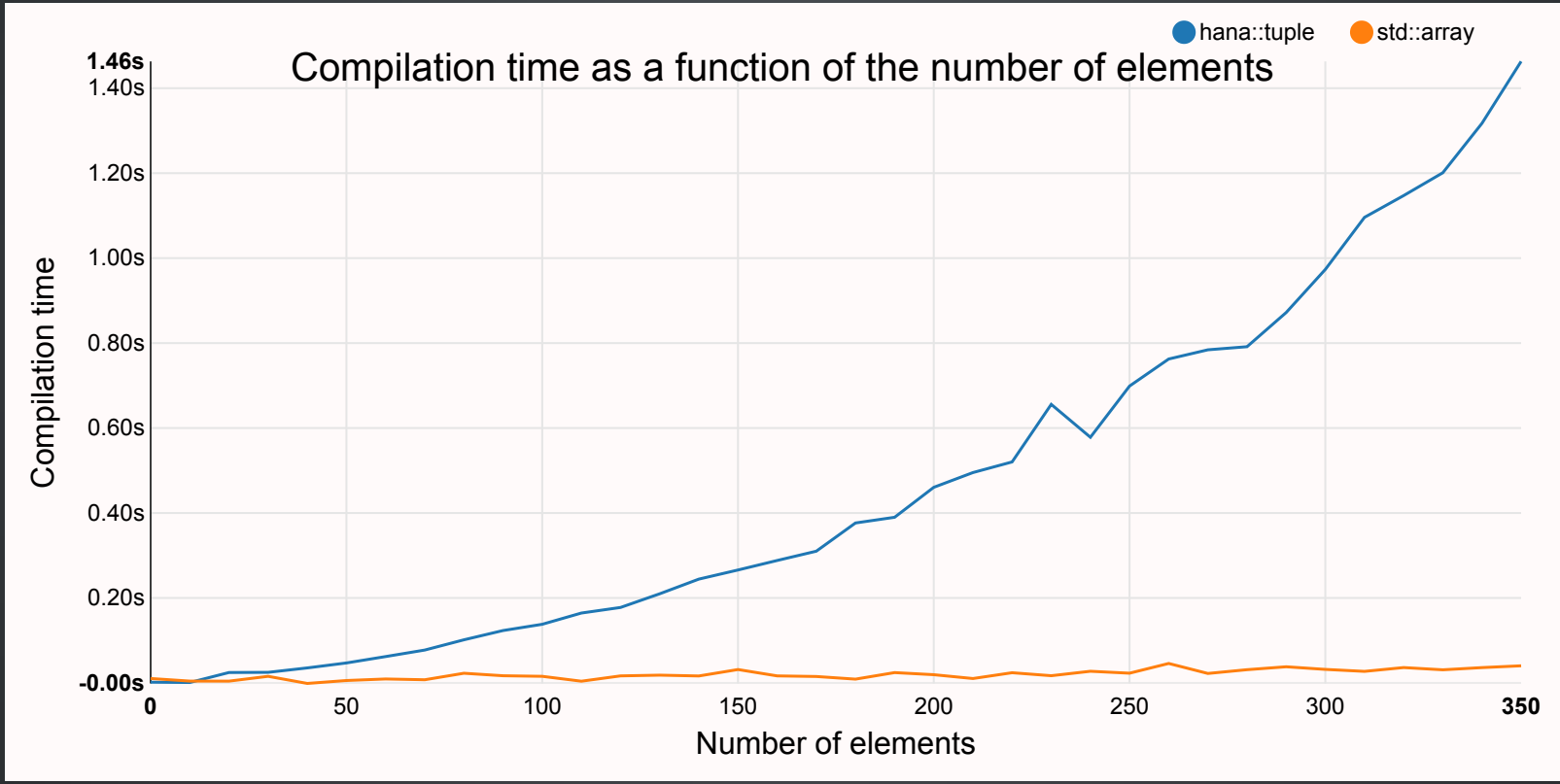
USE LTO AND WATCH OUT FOR LONG SYMBOLS CAUSING BLOAT

SHOULD I USE A TUPLE OR AN ARRAY?

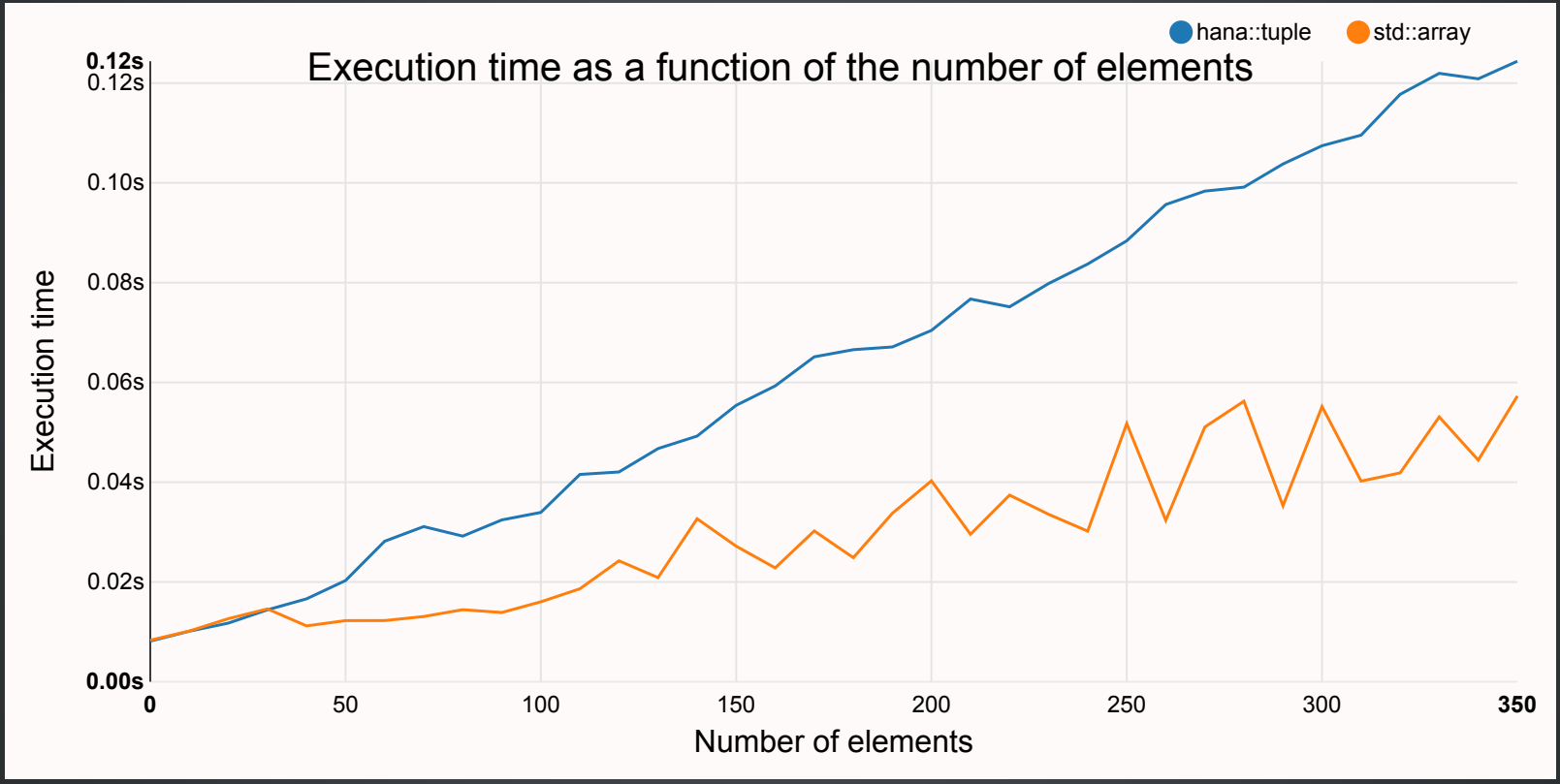
```
hana::tuple<int, int, int, int, int> ints{1, 2, 3, 4, 5};
int sum = 0;
hana::for_each(ints, [&](int i) { sum += i; });
```

```
std::array<int, 5> ints{{1, 2, 3, 4, 5}};
int sum = 0;
for (int i : ints)
    sum += i;
```

COMPILE-TIME; DEFINITELY THE ARRAY



RUNTIME; THAT DEPENDS ON THE NUMBER OF ELEMENTS



WHY? LOOK AT THE ASSEMBLY

FOR TUPLE

```
callq   _rand                   #
movl    %eax, -60(%rbp)         # Call std::rand() a bunch of times
...                             #

addl    -60(%rbp), %ebx         #
addl    -64(%rbp), %ebx         #
addl    -68(%rbp), %ebx         #
...                             #
addl    -196(%rbp), %ebx        # Add a bunch of times
addl    %r13d, %ebx             #
addl    %r14d, %ebx             #
addl    %r15d, %ebx             #
addl    %eax, %ebx              #
movslq  %ebx, %rsi
addq    -56(%rbp), %rsi
```

FOR ARRAY

```
callq   _rand                   #
movl    %eax, -176(%rbp)        # Call std::rand() a bunch of times
...                             #

movdqu  -176(%rbp), %xmm0       #
movdqu  -160(%rbp), %xmm1       #
movdqu  -144(%rbp), %xmm2       #
movdqu  -128(%rbp), %xmm3       #
paddd   %xmm0, %xmm1            #
paddd   %xmm2, %xmm1            #
paddd   %xmm3, %xmm1            #
movdqu  -112(%rbp), %xmm0       # Pack and vectorize-add a bunch of times
paddd   %xmm1, %xmm0            #
...                             #
movdqu  -32(%rbp), %xmm1        #
paddd   %xmm0, %xmm1            #
movdqa  %xmm1, %xmm0            #
movhlps %xmm0, %xmm0            #
paddd   %xmm1, %xmm0            #
phaddd  %xmm0, %xmm0            #
movd    %xmm0, %eax
movslq  %eax, %rsi
addq    %r14, %rsi
```

TIP 4

PREFER HOMOGENEOUS CONTAINERS WHEN POSSIBLE

IMPLEMENTING TUPLE ALGORITHMS

for_each

```
template <typename ...T, typename F, std::size_t ...i>
void for_each_impl(std::tuple<T...> const& tuple, F const& f,
                   std::index_sequence<i...>)
{
    int expand[] = {0, (f(std::get<i>(tuple)), void(), 0)...};
    (void)expand;
}

template <typename ...T, typename F>
void for_each(std::tuple<T...> const& tuple, F const& f) {
    for_each_impl(tuple, f, std::make_index_sequence<sizeof...(T)>{});
}
```

transform

```
template <typename ...T, typename F, std::size_t ...i>
auto transform_impl(std::tuple<T...> const& tuple, F const& f,
                    std::index_sequence<i...>)
{
    return std::make_tuple(f(std::get<i>(tuple))...);
}

template <typename ...T, typename F>
auto transform(std::tuple<T...> const& tuple, F const& f) {
    return transform_impl(tuple, f,
                std::make_index_sequence<sizeof...(T)>{});
}
```

select

```
template <typename Tuple, std::size_t ...i>
auto select(Tuple&& tuple, std::index_sequence<i...> const&) {
    using Result = std::tuple<
        std::tuple_element_t<i, std::remove_reference_t<Tuple>>...
    >;
    return Result{std::get<i>(static_cast<Tuple&&>(tuple))...};
}
```

EASY SO FAR

remove_if

```
template <typename ...T, typename Pred>
auto remove_if(std::tuple<T...> const& tuple, Pred const& pred) {
    return detail::remove_if_impl(tuple, pred,
                          std::make_index_sequence<sizeof...(T)>{});
}
```

BASE CASE

```
struct detail {

template <typename Tuple, typename Pred>
static auto
remove_if_impl(Tuple const& tuple, Pred const& pred,
               std::index_sequence<>)
{
    return std::tuple<>{};
}
```

NOT TAKING THE ELEMENT

```
template <typename Tuple, typename Pred, std::size_t i,
                                         std::size_t ...is>
static auto
remove_if_impl(Tuple const& tuple, Pred const& pred,
               std::index_sequence<i, is...>,
               std::enable_if_t<
                   decltype(pred(std::get<i>(tuple)))::value
               >* = nullptr)
{
    return detail::remove_if_impl(tuple, pred,
                                  std::index_sequence<is...>{});
}
```

TAKING THE ELEMENT

```
template <typename Tuple, typename Pred, std::size_t i,
                                         std::size_t ...is>
static auto
remove_if_impl(Tuple const& tuple, Pred const& pred,
               std::index_sequence<i, is...>,
               std::enable_if_t<
                   !decltype(pred(std::get<i>(tuple)))::value
               >* = nullptr)
{
    return std::tuple_cat(
        select(tuple, std::index_sequence<i>{}),
        detail::remove_if_impl(tuple, pred,
                               std::index_sequence<is...>{})
    );
}

}; // end struct detail
```

THIS IS WRONG!

TECHNICALLY, WHAT'S THE MINIMAL AMOUNT OF RUNTIME WORK WE HAVE TO DO?

LET'S PRETEND WE KNOW WHICH INDICES TO KEEP

```
template <typename ...T, typename Pred>
auto remove_if(std::tuple<T...> const& tuple, Pred const& pred) {
    using Indices = typename magic<Pred, T...>::type;
    return select(tuple, Indices{});
}
```

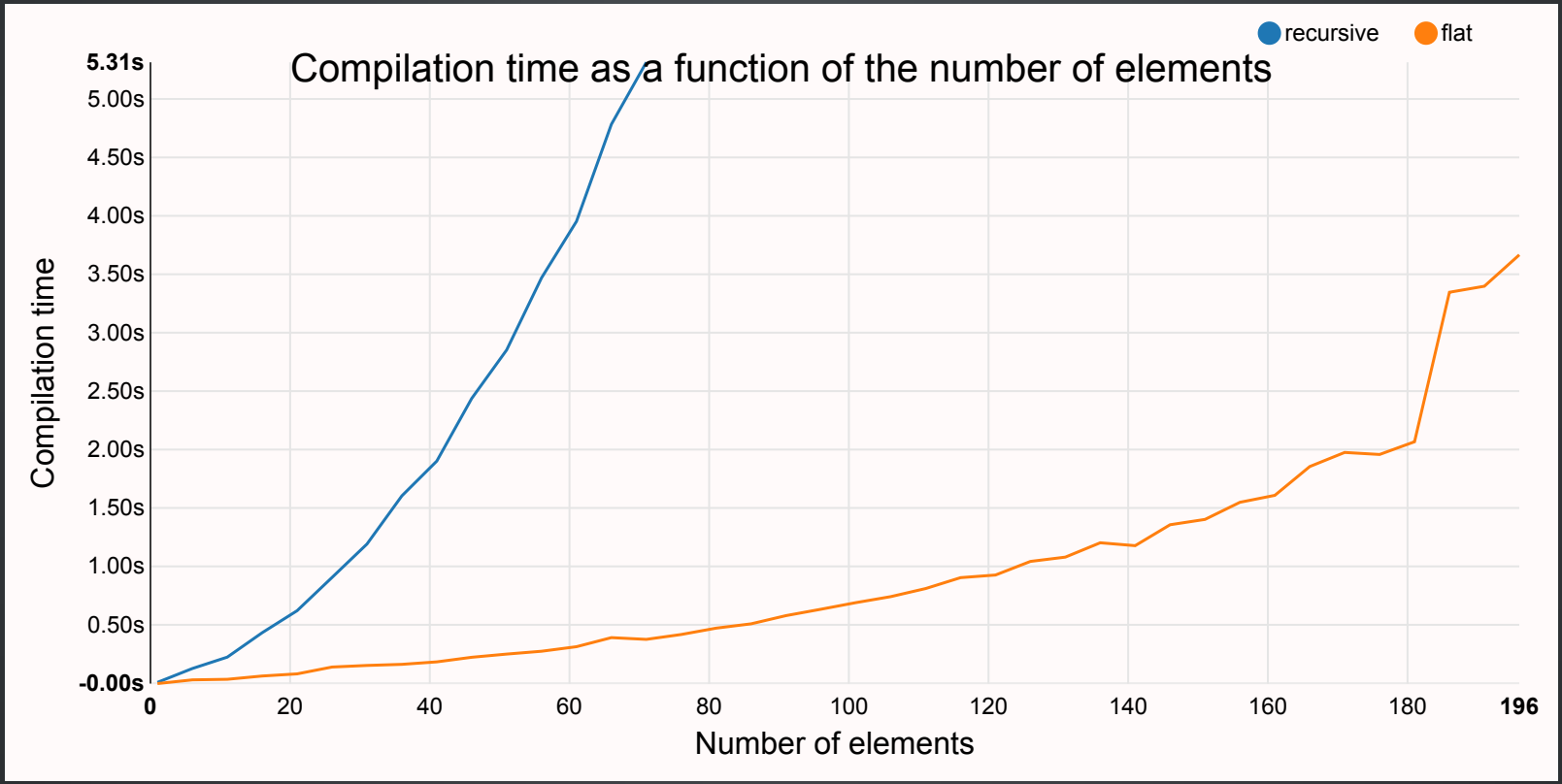
NOW ALL THAT'S LEFT IS TO IMPLEMENT `magic`

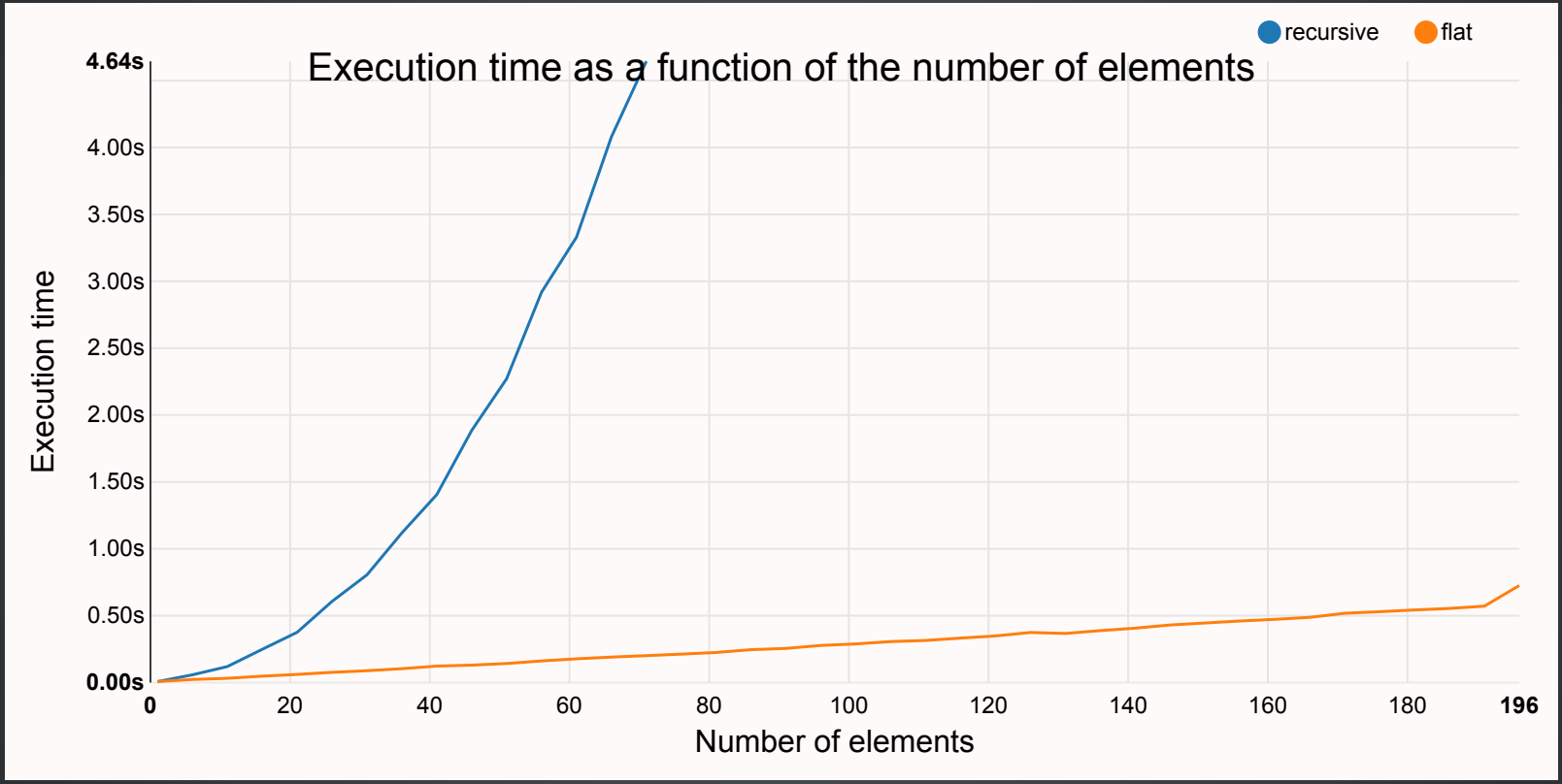
```
template <typename Pred, typename ...T>
struct magic {
    static constexpr bool results[] = {
        decltype(std::declval<Pred>()(std::declval<T>()))::value...
    };

    static constexpr std::size_t N_keep = detail::count(
        std::begin(results), std::end(results), false
    );

    using Indices = std::array<std::size_t, N_keep>;
    static constexpr Indices indices = compute_indices(); // ...
```

```
    static constexpr Indices compute_indices() {
        Indices indices{};
        std::size_t* out = &const_cast<std::size_t&>(
            static_cast<Indices const&>(indices)[0]
        );
        for (std::size_t i = 0; i < sizeof...(T); ++i) {
            if (!results[i]) {
                *out++ = i;
            }
        }

        return indices;
    }
```

```
    template <std::size_t ...i>
    static std::index_sequence<indices[i]...>
    as_index_sequence(std::index_sequence<i...>);

    using type = decltype(
        as_index_sequence(std::make_index_sequence<N_keep>{})
    );
}; // end struct magic
```

DID THAT PAY OFF?




recursive
flat
Execution time as a function of the number of elements
Execution time
4.64s
4.00s
3.50s
3.00s
2.50s
2.00s
1.50s
1.00s
0.50s
0.00s
0
20
40
60
80
100
120
140
160
180
196
Number of elements

THIS IS HOW LIBC++ IMPLEMENTS `tuple_cat`

```
constexpr boost::hana::tuple<> tuple_cat() {
    return boost::hana::tuple<>();
}

template <typename Tuple0, typename ...Tuples>
constexpr auto tuple_cat(Tuple0&& t0, Tuples&& ...tpls) {
    using T0 = std::remove_reference_t<Tuple0>;
    return tuple_cat_impl<
        boost::hana::tuple<>,
        std::index_sequence<>,
        std::make_index_sequence<tuple_size<T0>::value>
    >{}(
        boost::hana::tuple<>{},
        std::forward<Tuple0>(t0),
        std::forward<Tuples>(tpls)...
    );
}
```

```
template <typename Types, typename Is, typename Js>
struct tuple_cat_impl;

template <typename ...T, size_t ...I, size_t ...J>
struct tuple_cat_impl<boost::hana::tuple<T...>,
                      std::index_sequence<I...>,
                      std::index_sequence<J...>>
{
    template <typename Tuple0>
    constexpr auto operator()(boost::hana::tuple<T...> t, Tuple0&& t0)
    {
        return ::forward_as_tuple(
            std::forward<T>(boost::hana::at_c<I>(t))...,
            boost::hana::at_c<J>(std::forward<Tuple0>(t0))...
        );
    }
```

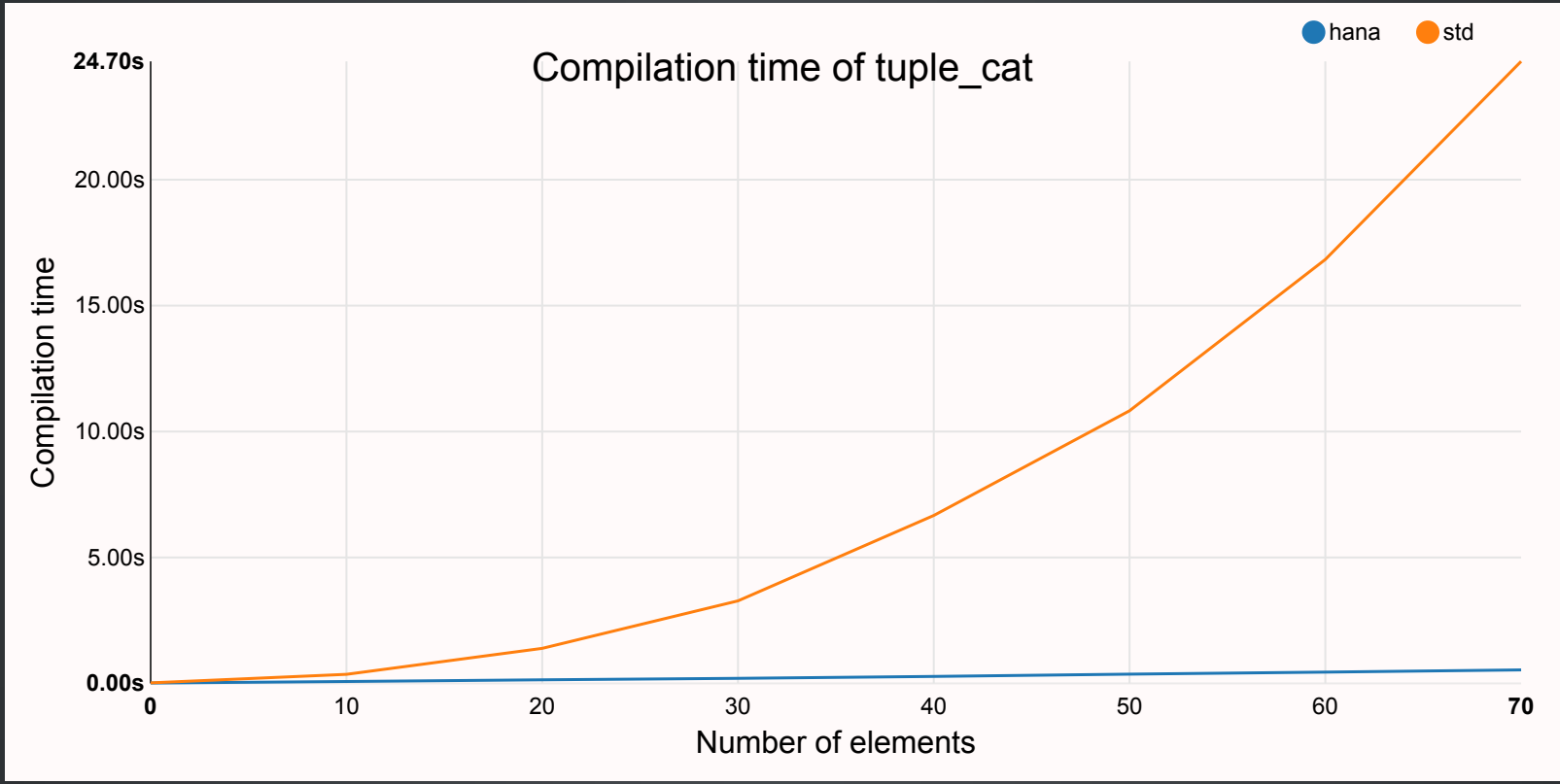
```
template <typename Tuple0, typename Tuple1, typename ...Tuples>
constexpr auto operator()(boost::hana::tuple<T...> t,
                          Tuple0&& t0, Tuple1&& t1,
                          Tuples&& ...tpls)
{
    using T0 = std::remove_reference_t<Tuple0>;
    using T1 = std::remove_reference_t<Tuple1>;
    return tuple_cat_impl<
        boost::hana::tuple<T...,
            typename apply_cv<Tuple0,
                typename tuple_element<J, T0>::type
            >::type&&...
        >,
        std::make_index_sequence<sizeof ...(T) +
                                 tuple_size<T0>::value>,
        std::make_index_sequence<tuple_size<T1>::value>
    >{}(
        ::forward_as_tuple(
            std::forward<T>(boost::hana::at_c<I>(t))...,
            boost::hana::at_c<J>(std::forward<Tuple0>(t0))...
        ),
        std::forward<Tuple1>(t1), std::forward<Tuples>(tpls)...
    );
}
```

THIS IS HOW HANA DOES IT

```
template <typename ...Tuples>
constexpr auto tuple_cat(Tuples&& ...tuples) {
    using Indices = flatten_indices<
        tuple_size<std::remove_reference_t<Tuples>>::value...
    >;
    using Result = typename tuple_cat_type<
        std::remove_reference_t<Tuples>...
    >::type; // implementation not relevant
    return Indices::template apply<Result>(
        ::forward_as_tuple(std::forward<Tuples>(tuples)...),
        std::make_index_sequence<Indices::flat_length>{}
    );
}
```

```cpp
template <std::size_t ...Lengths>
struct flatten_indices {
    static constexpr std::size_t lengths[] = {Lengths..., 0};
    static constexpr auto flat_length =
        mystd::accumulate(lengths, lengths + sizeof...(Lengths), 0);

    template <bool Inner>
    static constexpr auto compute() {
        mystd::array<std::size_t, flat_length> indices{};
        for (std::size_t index = 0, i = 0; i < sizeof...(Lengths); ++i)
            for (std::size_t j = 0; j < lengths[i]; ++j, ++index)
                indices[index] = (Inner ? i : j);
        return indices;
    }

    static constexpr auto inner = compute<true>();
    static constexpr auto outer = compute<false>();
```

```
    template <typename Result, typename Tuples, std::size_t ...i>
    static constexpr Result
    apply(Tuples tuples, std::index_sequence<i...>) {
        return Result{
            boost::hana::at_c<outer[i]>(
                boost::hana::at_c<inner[i]>(tuples)
            )...
        };
    }
};
```


hana
std
Compilation time of tuple_cat
24.70s
20.00s
15.00s
10.00s
5.00s
0.00s
Compilation time
0
10
20
30
40
50
60
70
Number of elements

TIP 5

OFFLOAD COMPUTATIONS TO COMPILE-TIME BY WORKING ON INDICES

LET'S TAKE A LOOK AT TUPLES

RECURSIVE IMPLEMENTATION

Do not do this!

stdlibc++, I'm looking at you!

MULTIPLE INHERITANCE

```
template <typename ...T>
struct tuple
    : tuple_impl<std::make_index_sequence<sizeof...(T)>, T...>
{
    using tuple_impl<std::make_index_sequence<sizeof...(T)>,
                     T...>::tuple_impl;
};

template <>
struct tuple<> { };
```

```cpp
template <std::size_t i, typename T>
struct element {
    element() = default;
    explicit constexpr element(T const& t) : value_(t) { }
    T value_;
};

template <typename Indices, typename ...T>
struct tuple_impl;

template <std::size_t ...i, typename ...T>
struct tuple_impl<std::index_sequence<i...>, T...>
    : element<i, T>...
{
    constexpr tuple_impl() = default;

    constexpr explicit tuple_impl(T const& ...t)
        : element<i, T>(t)...
    { }
};
```

ACCESSING TUPLE ELEMENTS

```
template <std::size_t n, typename T>
constexpr T& get(element<n, T>& t) {
    return t.value_;
}
```

ALIGNED STORAGE (EXPERIMENTAL)

COMPUTING THE OFFSET OF MEMBERS

```
template <typename ...T>
struct tuple {
    using SizeArray = std::array<std::size_t, sizeof...(T)>;
    static constexpr SizeArray sizes = {{sizeof(T)...}};
    static constexpr SizeArray alignments = {{alignof(T)...}};
    static constexpr SizeArray offsets = compute_offsets(); // ...
```

CREATING THE STORAGE

```
static constexpr std::size_t total_size = offsets[sizeof...(T)-1] +
                                          sizes[sizeof...(T)-1];

typename std::aligned_storage<total_size>::type storage_;
```

ACCESSING THE STORAGE

```
constexpr void* get_raw(std::size_t i) {
    if (i >= sizeof...(T))
        throw "out of bounds access in a tuple";
    return ((char*)&storage_) + offsets[i];
}
```

DEFAULT CONSTRUCTOR

```
tuple() {
    std::size_t i = 0;
    void* expand[] = {::new (this->get_raw(i++)) T()...};
    (void)expand;
}
```

VARIADIC CONSTRUCTOR

```
explicit tuple(T const& ...args) {
    std::size_t i = 0;
    void* expand[] = {::new (this->get_raw(i++)) T(args)...};
    (void)expand;
}
```

DESTRUCTOR

```
    ~tuple() {
        std::size_t i = sizeof...(T);
        int expand[] = {
            (static_cast<T*>(this->get_raw(--i))->~T(), int{})...
        };
        (void)expand;
    }
};
```

TYPED ACCESS TO THE STORAGE

```
template <std::size_t i, typename ...T,
    typename Nth = typename nth_type<i, T...>::type>
constexpr Nth& get(tuple<T...>& ts) {
    return *static_cast<Nth*>(ts.get_raw(i));
}
```

BENEFITS OF ALIGNED STORAGE

SOME ALGORITHMS ARE KILLER

FIRST, ADD THE FOLLOWING CONSTRUCTOR

```
explicit constexpr tuple(uninitialized const&) { }
```

NOW, LET'S IMPLEMENT PARTIAL SUMS

```
template <typename ...T, typename State, typename F>
auto partial_sum(tuple<T...> const& ts, State const& state, F const& f)
{
    return partial_sum_impl(ts, state, f,
                        std::make_index_sequence<sizeof...(T)+1>{});
}
```

```
template <typename T, typename ...U, typename State, typename F,
          std::size_t ...i,
          typename Result = tuple<
              State, fold_left_nth<i, State, F, T, U...>...
          >
>
Result partial_sum_impl(tuple<T, U...> const& ts, State const& state,
                        F const& f, std::index_sequence<0, i...>)
{
    Result result{uninitialized{}};

    void* expand[] = {
        ::new (result.get_raw(0)) State(state),
        ::new (result.get_raw(i)) fold_left_nth<i, State, F, T, U...>(
            f(::get<i-1>(result), ::get<i-1>(ts))
        )...
    };
    (void)expand;

    return result;
}
```

CAN DO AGGRESSIVE OPTIMIZATIONS EASILY

SHOULD BE EASY ON THE OPTIMIZER

POSSIBILITY OF PRE-ALLOCATING STORAGE

```
tuple<int, char, double> tuple{uninitialized};
tuple.push_back<int>(1);
tuple.push_back<char>('2');
tuple.push_back<double>(3.3);
```

TIP 6

CHOOSE THE RIGHT REPRESENTATION FOR THE TASK AT HAND

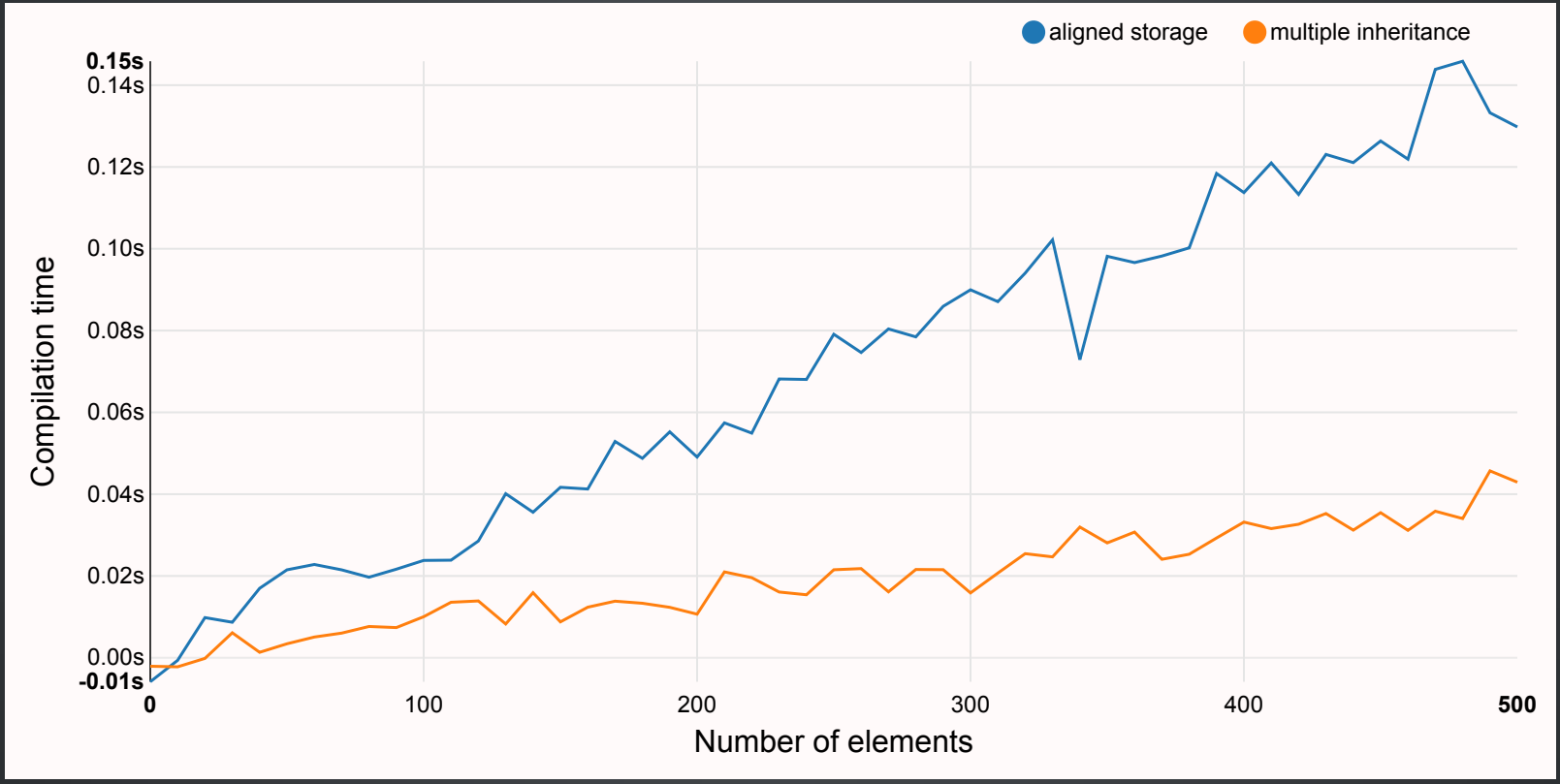
DOWNSIDES OF ALIGNED STORAGE

CAN'T BE CONSTEXPR (IN-PLACE `new` & DESTRUCTOR)

```
constexpr tuple<int, char> t{1, '2'};
```

```
[snip].cpp:5:28: error: constexpr variable cannot have non-literal type
                        'const tuple<int, char>'
constexpr tuple<int, char> t{1, '2'};
                           ^
[snip].hpp:90:5: note : 'tuple<int, char>' is not literal because it
                        has a user-provided destructor
    ~tuple() {
    ^
1 error generated.
```

SLOWER THAN MULTIPLE INHERITANCE



TIP 7

NEVER ASSUME, ALWAYS BENCHMARK!

- Different compilers behave differently
- The execution model is hard to predict
- Simply counting instantiations doesn't work

INTRODUCING METABENCH

NOT ANYMORE!

GOALS

- Simple
- Self-contained

1. WRITE YOUR BENCHMARK

std.cpp.erb:

```
#include <tuple>

int main() {
#if defined(METABENCH)
    auto tuple = std::make_tuple(<%= (1..n).to_a.join(', ') %>);
#endif
}
```

hana.cpp.erb:

```
#include <boost/hana/tuple.hpp>
namespace hana = boost::hana;

int main() {
#if defined(METABENCH)
    auto tuple = hana::make_tuple(<%= (1..n).to_a.join(', ') %>);
#endif
}
```

2. SETUP METABENCH

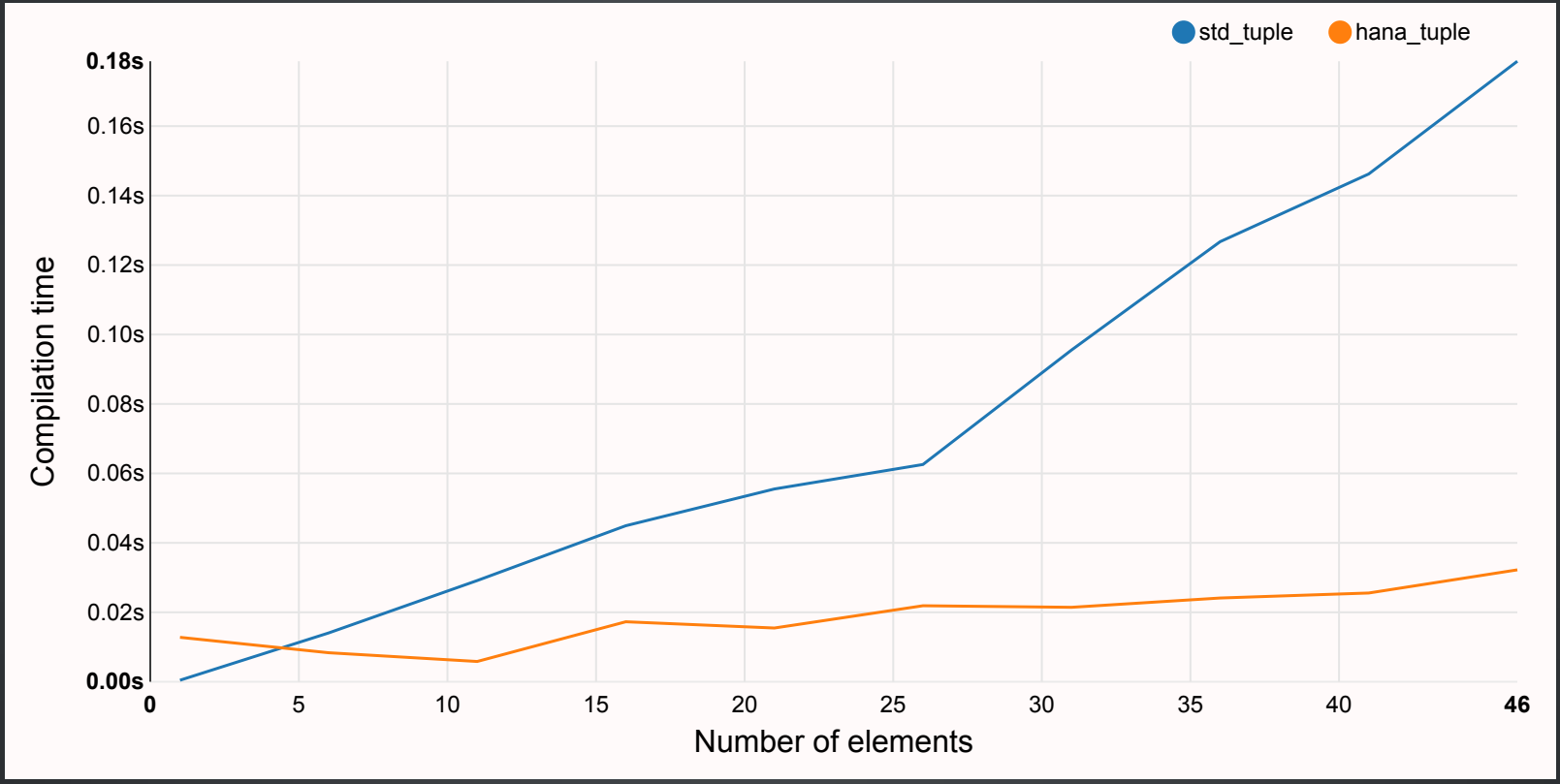
CMakeLists.txt:

```
include(metabench)

# Add new datasets
metabench_add_dataset(std_tuple  "std.cpp.erb" "(1..50).step(5)")
metabench_add_dataset(hana_tuple "hana.cpp.erb" "(1..50).step(5)")

# Add a new chart
metabench_add_chart(benchmark.make_tuple
    DATASETS std_tuple hana_tuple
    OUTPUT ${CMAKE_CURRENT_SOURCE_DIR}/make_tuple.html
)
```

3. BUILD THE BENCHMARK

```
ldionne in ~/code/cppnow-2016-metaprogramming-for-the-brave/build %
```

4. TADAM!


std_tuple
hana_tuple
0.18s
0.16s
0.14s
0.12s
0.10s
0.08s
0.06s
0.04s
0.02s
0.00s
Compilation time
0
5
10
15
20
25
30
35
40
46
Number of elements

KEY TAKEAWAYS

- Write simple code (for the compiler)
- Be data-driven when you optimize

THANK YOU

http://ldionne.com
http://github.com/ldionne
http://github.com/ldionne/metabench