accu
2023

KEYNOTE:
C++ HORIZONS

BRYCE ADELSTEIN LELBACH

C++20 is in the field

- Modules
- Coroutines
- Concepts
- Ranges

C++23 is coming!

- More ranges
- Formatted output
- `mdspan`
- `expected`
- `import std`
- Deducing `this`

On The Horizon For

On The Horizon For C++

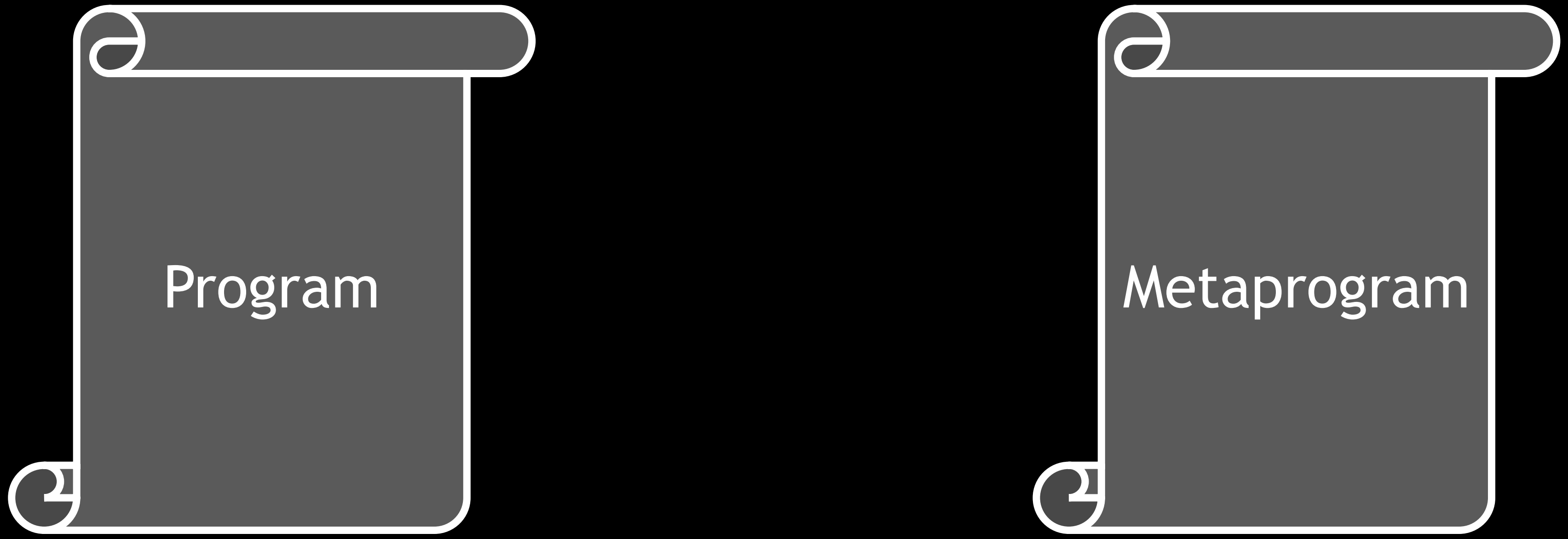
- Reflection

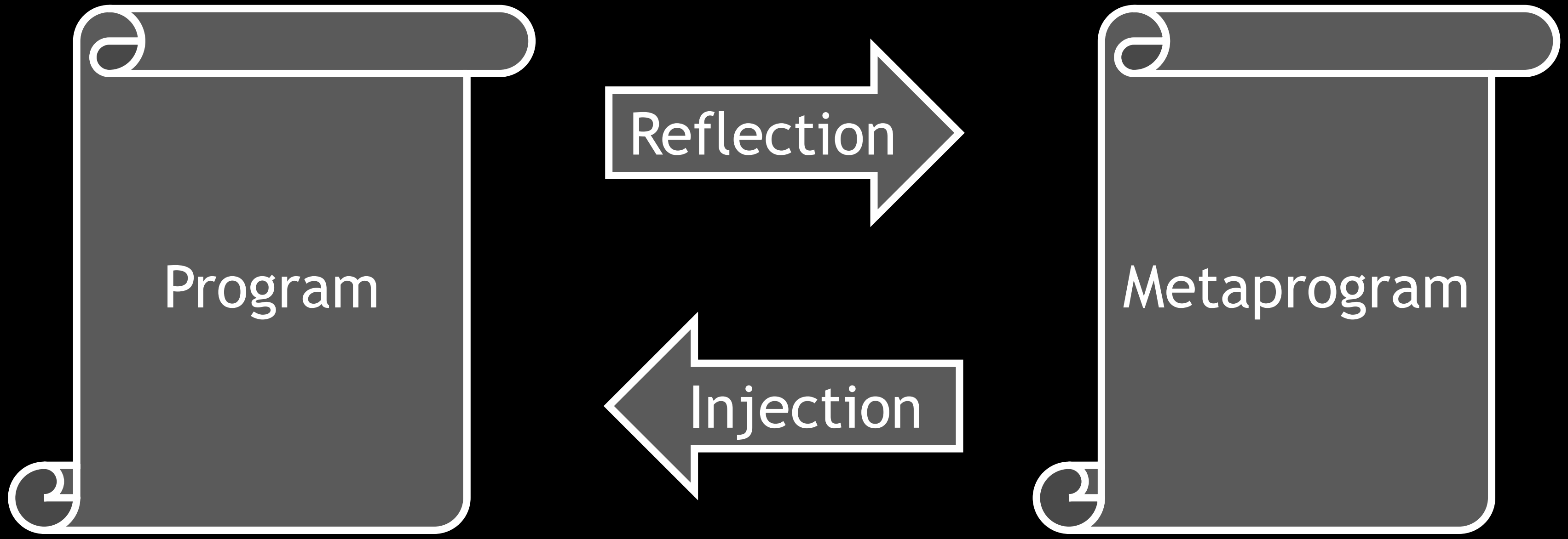
On The Horizon For C++

- Reflection
- Pattern Matching

On The Horizon For C++

- Reflection
- Pattern Matching
- Senders


Program
Metaprogram


Program
Reflection
Injection
Metaprogram

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
      return meta::name_of(e);
  return "<unnamed>";
}
```

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
      return meta::name_of(e);
  return "<unnamed>";
}
```

^*name-or-postfix-expr*

The reify operator.
entity -> reflection

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] =
      return me
  return "<unna
}
```

std::meta::info is the type for reflection objects.

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
      return meta::name_of(e);
  return "<unnamed>";
}
```

Reflections can be manipulated and transformed like any C++ objects.

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
                     me_of(e);
```

template for

Compile-time expansion.

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
      return meta::name_of(e);
  retu
}
```

[:*reflection*:]

The splice operator.
reflection -> entity

```
template <typename T> requires is_enum_v<T>
constexpr string to_string(T value) {
  template for (constexpr meta::info e : meta::members_of(^T))
    if ([:e:] == value)
      return meta::name_of(e);
  return "<unnamed>";
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```cpp
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename Hasher, typename T>
constexpr void hash_append(Hasher& hasher, T const& t) {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr meta::info member : data_members)
    hash_append(hasher, t.[:member:]);
}
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
                              , name_of(^T), name_of(e));
                              rward<[:parameter_types_of(e):]>(
                              f(e)#]...

    }
};
```

[#*reflection*#]

The identifier splice operator.
reflection -> identifier

```cpp
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
                          , name_of(^T), name_of(e));
                          rward<[:parameter_types_of(e):]>(
                          f(e)#]...
    }
};
```

...[:*reflection*:]
...[#*reflection*#]

Splice pack expansion
reflection -> pack

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```cpp
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

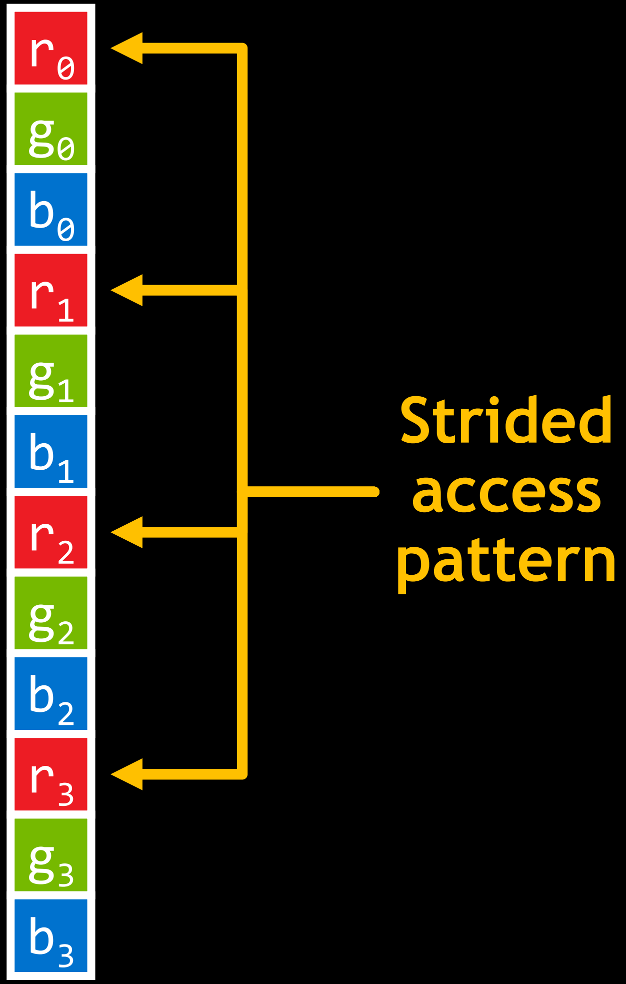
```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
template <typename T> requires is_class<T>
struct traced {
  private: T payload_;
  template for (constexpr auto e : member_functions_of(^T))
    [:protection_of(e):]: [:attributes_of(e):]
    [:return_of(e):] [#e#]
    (...[:parameter_types_of(e):] ...[#parameters_of(e)#]...)
    [:qualifiers_of(e):]
    {
      print("Calling {}::{}\n", name_of(^T), name_of(e));
      return payload_.[:e:](forward<[:parameter_types_of(e):]>(
        ...[#parameter_names_of(e)#]...
      ));
    }
};
```

```
struct color {
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

std::vector<color> image(…);

for (auto& [r, g, b] : image)
  r /= 2;
```

```
struct color {
  std::vector<uint8_t> red;
  std::vector<uint8_t> green;
  std::vector<uint8_t> blue;
};

color image(…);

for (auto& r : image.red)
  r /= 2;
```

```
struct color {
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

std::vector<color> image(…);

for (auto& [r, g, b] : image)
  r /= 2;
```



```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

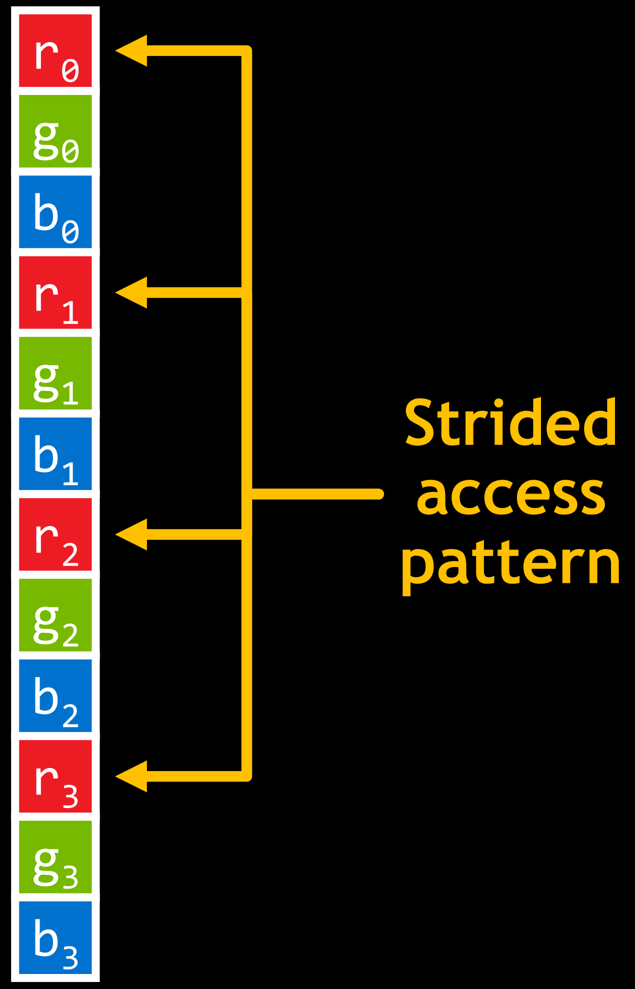
```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

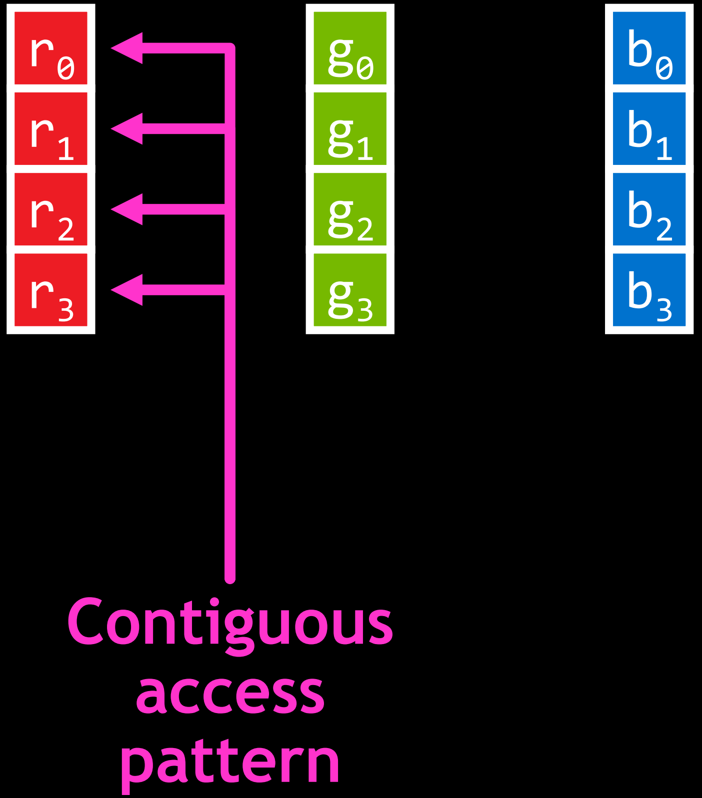
```
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

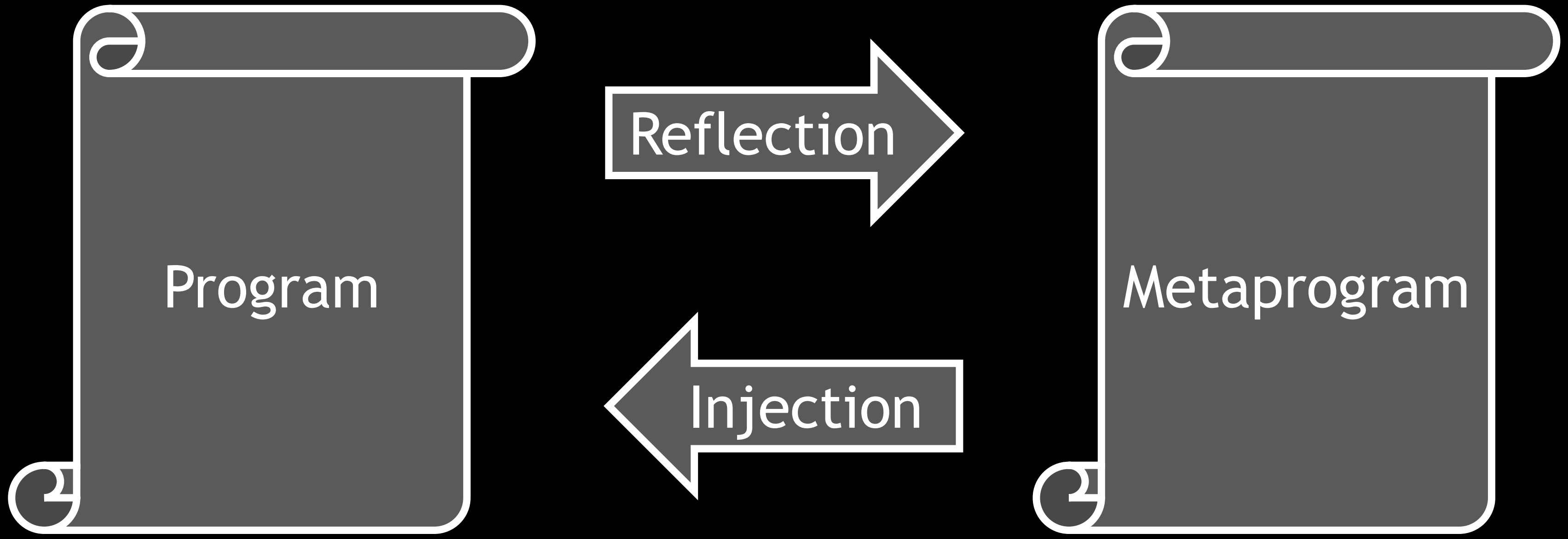
```cpp
template <typename T>
struct soa {
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);

  vector<...[:meta::type_of(data_members):]> ...[#data_members#]...;

  soa(ranges::forward_range<T>&& aos) {
    template for (constexpr meta::info member : data_members) {
      [#member#].resize(ranges::size(aos));
      ranges::transform(aos, [#member#].begin(),
        [] (auto&& v) { return v.[:member:]; });
    }
  }

  auto operator[](size_t i) { return tie(...[#data_members#][i]...); }
};
```

```
struct color {
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

std::vector<color> image(…);

for (auto& [r, g, b] : image)
  r /= 2;
```



```
struct color {
  uint8_t red;
  uint8_t green;
  uint8_t blue;
};

soa<color> image(…);

for (auto& [r, g, b] : image)
  r /= 2;
```




Program
Reflection
Injection
Metaprogram

Selection in C++

Selection in C++

`switch`: Operates on a single integral value. Often too limited.

Selection in C++

`switch`: Operates on a single integral value. Often too limited.

`if`: Operates on arbitrary Boolean expressions. Often too complex.

Selection in C++

`switch`: Operates on a single integral value. Often too limited.

`inspect`: Matches values against patterns; binds variables on success.

`if`: Operates on arbitrary Boolean expressions. Often too complex.

```
switch (i) {
  case 0:  print("got zero"); break;
  case 1:  print("got one"); break;
  default: print("don't care");
}
```

```
inspect (i) {
  0  => { print("got zero"); }
  1  => { print("got one"); }
  __ => { print("don't care"); }
};
```

`inspect (`*expr*`) { … }`

Expression, not a statement.

```
switch (i) {
  case 0:  print("got zero"); break;
  case 1:  print("got one"); break;
  default: print("don't care");
}
```

```
inspect (i) {
  0  => { print("got zero"); }
  1  => { print("got one"); }
  __ => { print("don't care"); }
};
```

pattern => *statement*

There's a variety of different patterns available.

```
switch (i) {
  case 0:  print("got zero"); break;
  case 1:  print("got one"); break;
  default: print("don't care");
}
```

```
inspect (i) {
  0  => { print("got zero"); }
  1  => { print("got one"); }
  __ => { print("don't care"); }
};
```

inspect stops at the first match, not the best match.

Constant patterns.

```
switch (i) {
  case 0:  print("got zero"); break;
  case 1:  print("got one"); break;
  default: print("don't care");
}
```

```
inspect (i) {
  0  => { print("got zero"); }
  1  => { print("got one"); }
  __ => { print("don't care"); }
};
```

Wildcard pattern.

```
switch (i) {
  case 0:  print("got zero"); break;
  case 1:  print("got one"); break;
  default: print("don't care");
}
```

```
inspect (i) {
  0  => { print("got zero"); }
  1  => { print("got one"); }
  __ => { print("don't care"); }
};
```

```
if (s == "foo") {
  print("got foo“);
} else if (s == "bar") {
  print("got bar");
} else {
  print("don't care");
}
```

```
inspect (x) {
  "foo" => { print("got foo"); }
  "bar" => { print("got bar"); }
  __    => { print("don't care"); }
};
```

Constant patterns aren’t limited to integrals or enums!

```
unsigned fibonacci(unsigned n) {
  return inspect (n) {
    0 => 0;
    1 => 1;
    e if (e > 47) => { throw overflow_error("too large"); }
    a => fibonacci(a - 1) + fibonacci(a - 2);
};
```

```
unsigned fibonacci(unsigned n) {
  return inspect (n) {
    0 => 0;
    1 => 1;
    e if (e > 47) => { throw overflow_error("too large"); }
    a => fibonacci(a - 1) + fibonacci(a - 2);
};
```

Identifier patterns bind the value to a name.

```
unsigned fibonacci(unsigned n) {
  re
  (
```

Pattern guards can perform complex tests that cannot be performed within the *pattern*.

```
    e if (e > 47) => { throw overflow_error("too large"); }
    a => fibonacci(a - 1) + fibonacci(a - 2);
};
```

```
auto&& [x, y, z] = p;
if (x == 0 && y == 0 && z == 0) {
  print("on origin");
} else if (x == 0) {
  print("on y-axis");
} else if (y == 0) {
  print("on x-axis");
} else if (z == 0) {
  print("on z-axis");
} else {
  print("{}, {}, {}", x, y, z);
}
```

```
inspect (p) {
  [0, 0] => { print("on origin"); }
  [0, y] => { print("on y-axis"); }
  [x, 0] => { print("on x-axis"); }
  [x, y] => { print("{}, {}, {}", x, y, z); }
};
```

[*pattern*, *pattern*, …] => …

Compound patterns can be used to decompose objects.

```
struct visitor {
  void operator()(int i) const {
    print("got int: {}", i);
  }
  void operator()(float f) const {
    print("got float: {}", f);
  }
};

visit(visitor{}, v);
```

```
inspect (v) {
  <int>   i => { print("got int: {}", i); }
  <float> f => { print("got float: {}", f); };
}
```

<type> pattern => …

Alternative patterns match different types.

```
struct phone_number_extractor {
  optional<array<string_view, 3>>
  try_extract(string_view sv) const;
};
inline constexpr phone_number_extractor phone_number;
```

```
struct email_extractor {
  optional<array<string_view, 2>>
  try_extract(string_view sv) const;
};
inline constexpr email_extractor email;

inspect (s) {
  (phone_number?) ["212", __, __]   => { print("got a New York phone number"); }
  (email?)        [address, domain] => { print("got an email"); }
};
```

(expr?) pattern => …

Extractor patterns allow you to customize matching and decomposition.

```
struct phone_number_extractor {
  optional<array<string_view, 3>>
  try_extract(string_view sv) const {
    auto s = views::split(sv, "-");
    if (ranges::size(s) != 3) return nullopt;
    for (auto&& c: s) if (ranges::find_if_not(c, isdigit)) return nullopt;
    return {begin(s), begin(s) + 1, begin(s) + 2};
  }
};
inline constexpr phone_number_extractor phone_number;

struct email_extractor {
  optional<array<string_view, 2>>
  try_extract(string_view sv) const;
};
inline constexpr email_extractor email;

inspect (s) {
  (phone_number?) ["212", __, __]   => { print("got a New York phone number"); }
  (email?)        [address, domain] => { print("got an email"); }
};
```

(expr?) pattern => …

Extractor patterns allow you to customize matching and decomposition.

```
struct phone_number_extractor {
  optional<array<string_view, 3>>
  try_extract(string_view sv) const {
    auto s = views::split(sv, "-");
    if (ranges::size(s) != 3) return nullopt;
    for (auto&& c: s) if (ranges::find_if_not(c, isdigit)) return nullopt;
    return {begin(s), begin(s) + 1, begin(s) + 2};
  }
};
inline constexpr phone_number_extractor phone_number;

struct email_extractor {
  optional<array<string_view, 2>>
  try_extract(string_view sv) const;
};
inline constexpr email_extractor email;

inspect (s) {
  (phone_number?) ["212", __, __]   => { print("got a New York phone number"); }
  (email?)        [address, domain] => { print("got an email"); }
};
```

(expr?) pattern => …

Extractor patterns allow you to customize matching and decomposition.

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```cpp
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```cpp
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}", p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}“, p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}“, p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}“, p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}“, p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

```
template <typename T>
constexpr std::string save_json(T const& t) {
  std::string out;
  constexpr meta::info data_members =
    meta::members_of(^T, meta::is_nonstatic_data_member);
  template for (constexpr auto i : views::iota(0, data_members.size()) {
    constexpr meta::info member = data_members[i];
    out += format("{{\"{}\":", meta::name_of(member));
    inspect (t.[:member:]) {
      <bool> b => { out += b ? "true" : "false"; }
      <numeric> n => { out += format("{}", n); }
      <pointer> nullptr => { out += "null"; }
      <pointer> p => { out += format("{}“, p); }
      <string_view> s => { out += format("\"{}\"", s); }
      <range> rng => { out += format("{}", rng | transform(save_json)); }
      obj => { out += format("{{{}}}", save_json(obj)); }
    }
    out += "}";
    if (i != data_members.size() - 1)
      out += ",";
  }
  return out;
}
```

Selection in C++

`switch`: Operates on a single integral value. Often too limited.

`inspect`: Matches values against patterns; binds variables on success.

`if`: Operates on arbitrary Boolean expressions. Often too complex.

Today, C++ has:

- No standard model for asynchrony.
- No standard way to express where things should execute.

Today, C++ has:

- No standard model for asynchrony.
- No standard way to express where things should execute.

The solution is coming soon:

Senders

Schedulers are handles to execution contexts.

Schedulers are handles to execution contexts.

Senders represent asynchronous work.

Schedulers are handles to execution contexts.

Senders represent asynchronous work.

Receivers process asynchronous signals.

```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

```cpp
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

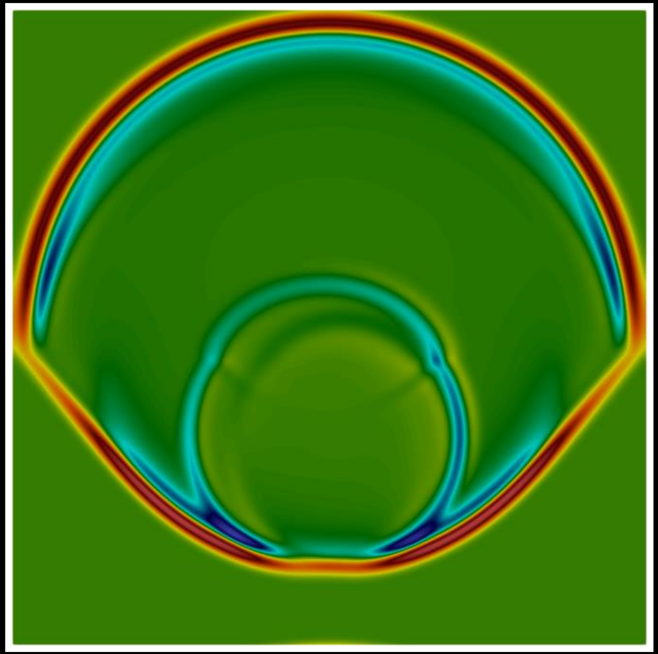
```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

```
ex::scheduler auto sch = thread_pool.scheduler();

ex::sender auto begin = ex::schedule(sch);
ex::sender auto hi    = ex::then(begin, [] { return 13; });
ex::sender auto add   = ex::then(hi, [] (int a) { return a + 42; });

auto [i] = this_thread::sync_wait(add).value();
```

```
std::vector<std::string_view> v{…};

ex::sender auto s = ex::transfer_just(gpu_stream_scheduler{}, v)
                  | sort_async
                  | unique_async
                  | ex::transfer(thread_pool.scheduler())
                  | for_each_async([] (std::string_view e)
                                   { std::print(file, "{}\n", e); });

this_thread::sync_wait(s);
```

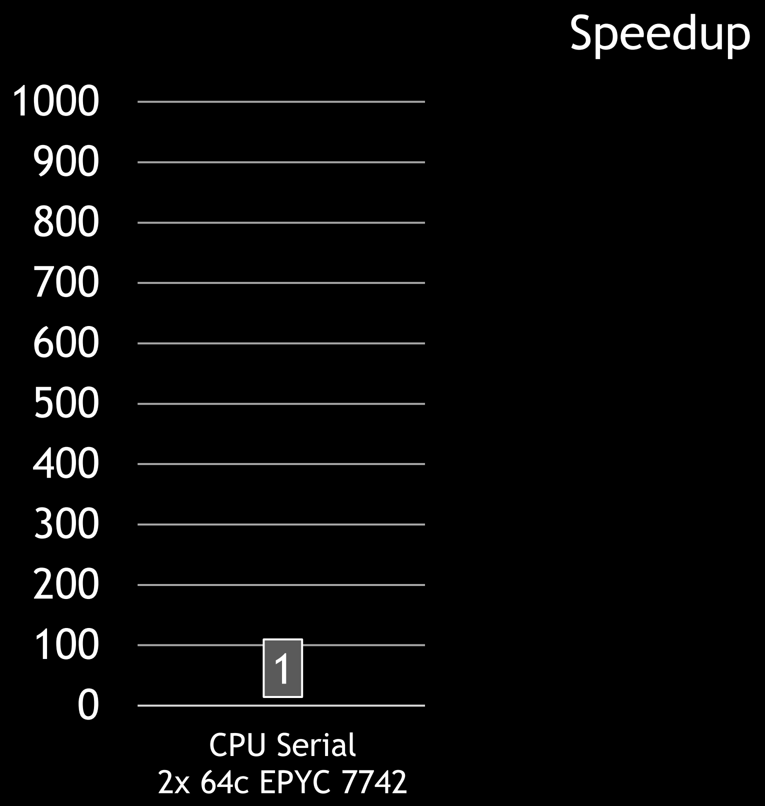
Maxwell's Equations

```
sender auto maxwell_eqs(scheduler auto &compute,
                        grid_accessor A, …) {
  return repeat_n(n_outer_iterations,
           repeat_n(n_inner_iterations,
               schedule(compute)
             | bulk(G.cells, update_h(G))
             | halo_exchange(G, hx, hy)
             | bulk(G.cells, update_e(time, dt, G))
             | halo_exchange(G, hx, hy))
           | transfer(cpu_serial_scheduler)
           | then(output_results))
         );
}
```



Maxwell's Equations

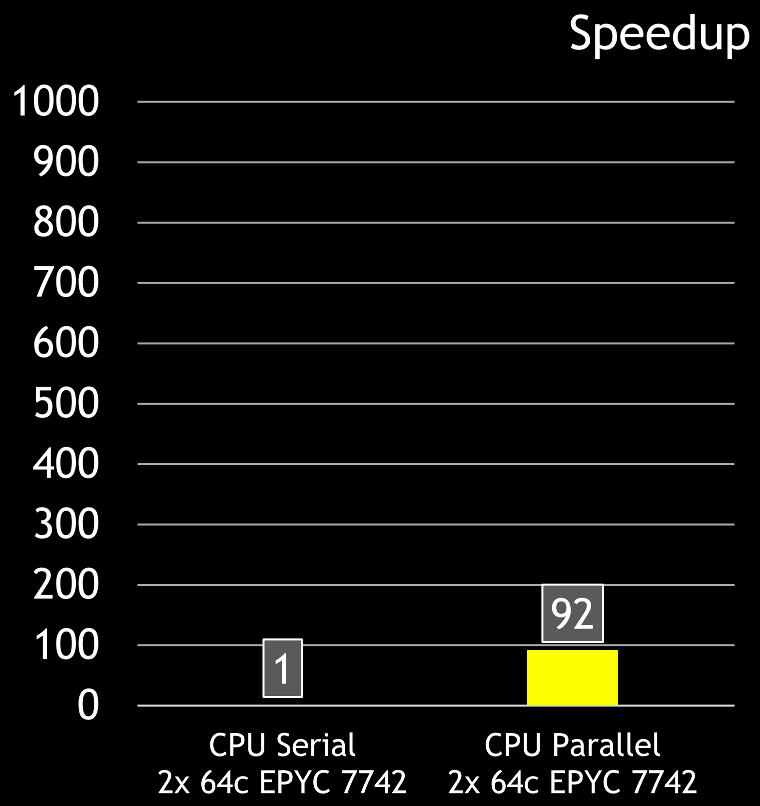
Change one line of code and scale from a single CPU thread…




```
sync_wait(maxwell_eqs(cpu_serial_scheduler), …);
```

Maxwell's Equations

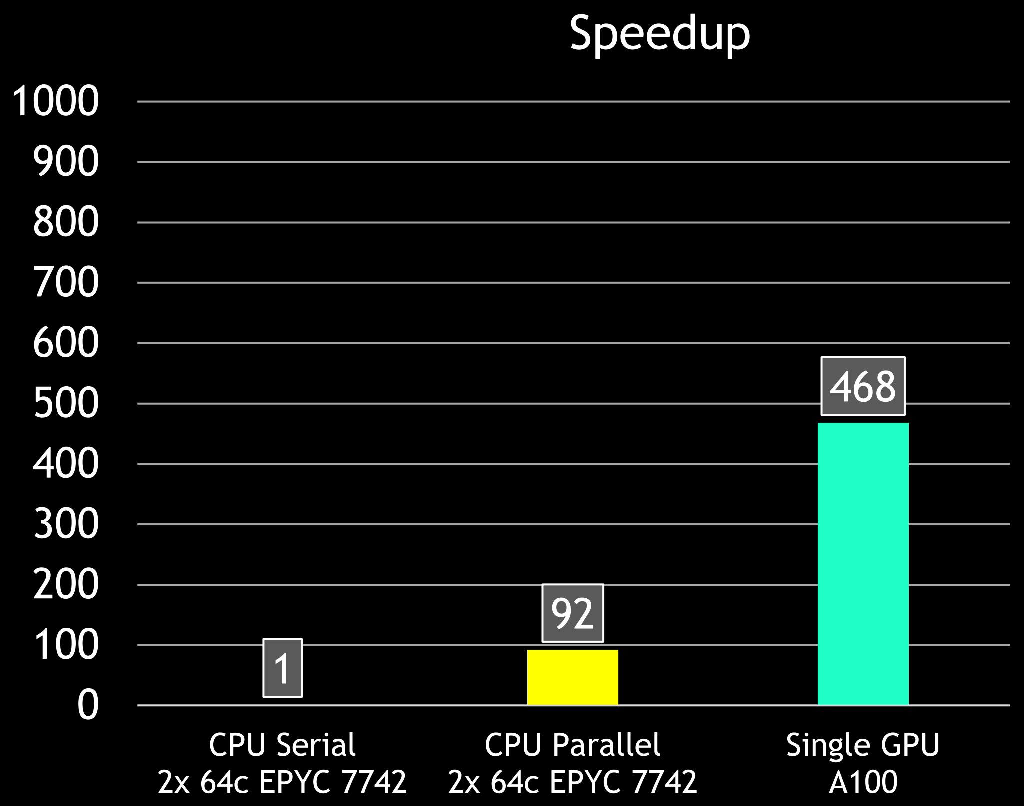
Change one line of code and scale from a single CPU thread up to multiple CPU threads…




```
sync_wait(maxwell_eqs(cpu_parallel_scheduler), …);
```

Maxwell's Equations

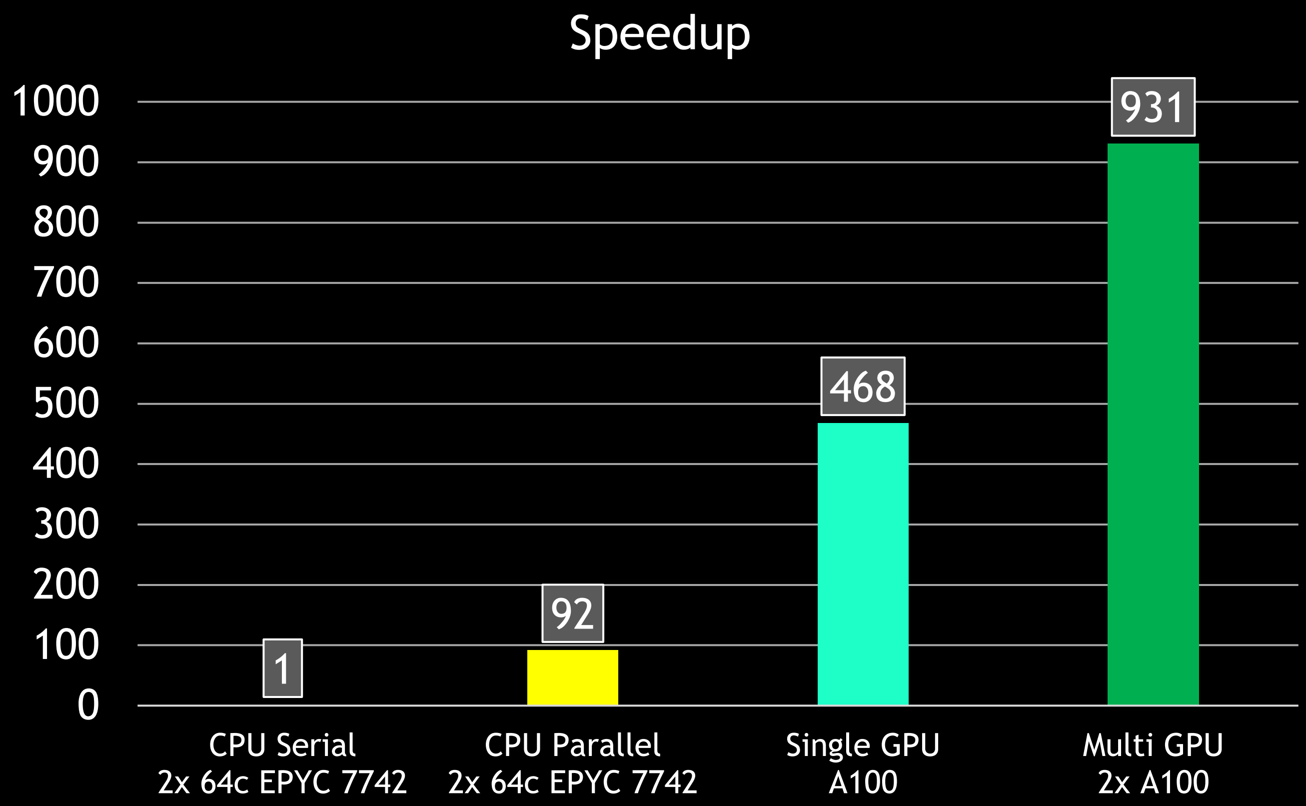
Change one line of code and scale from a single CPU thread up to a GPU…




```
sync_wait(maxwell_eqs(single_gpu_scheduler), …);
```

Maxwell's Equations

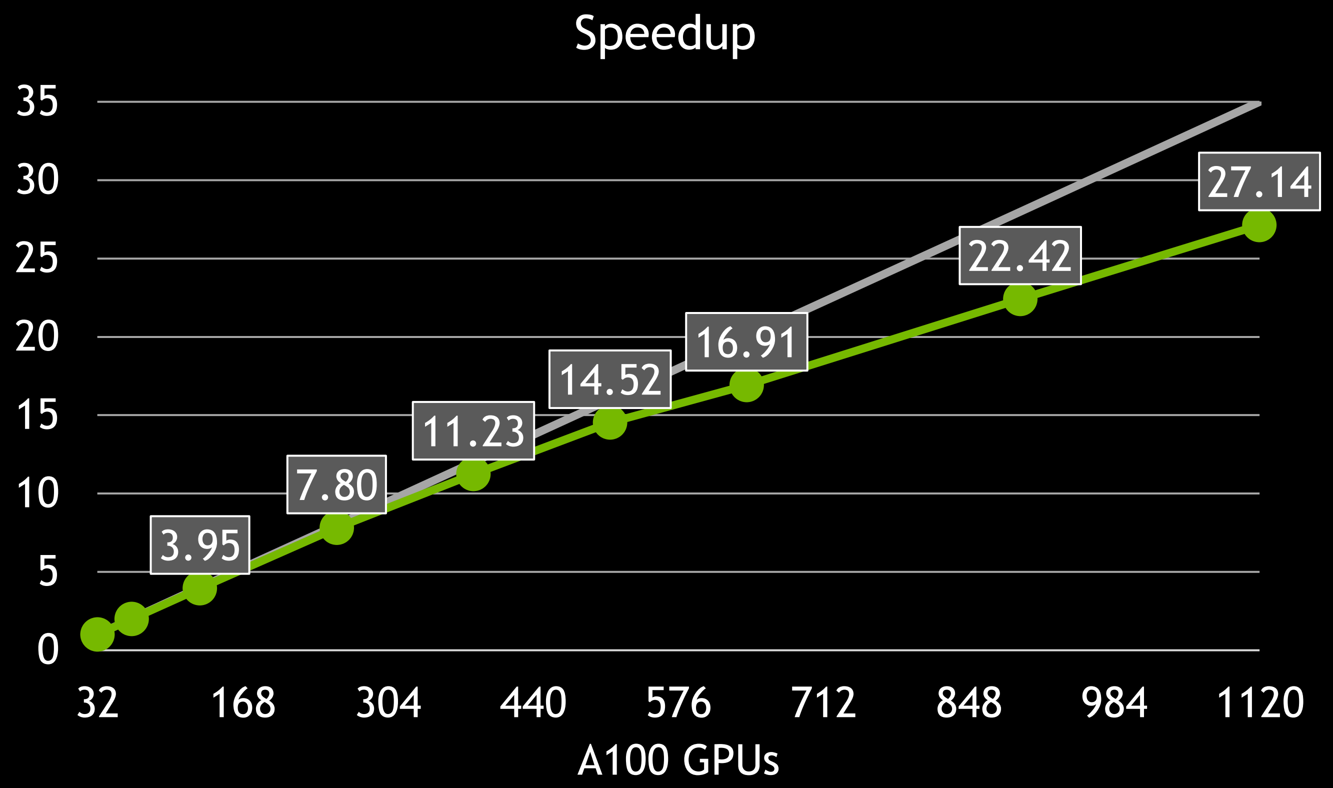
Change one line of code and scale from a single CPU thread up to multiple GPUs…




```
sync_wait(maxwell_eqs(multi_gpu_scheduler), …);
```

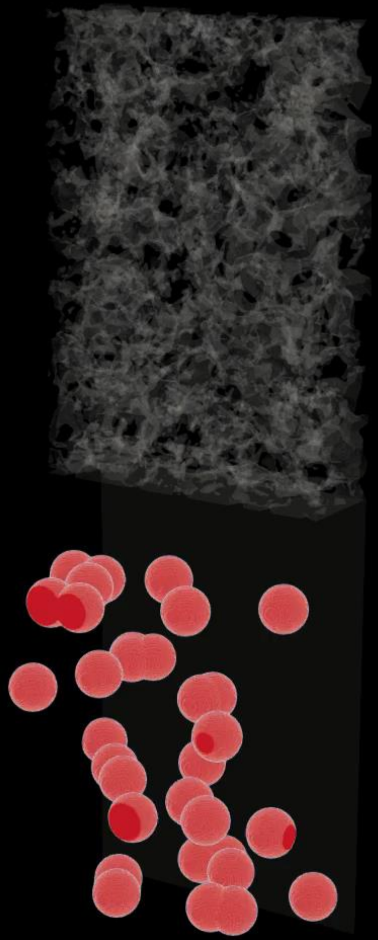
Maxwell's Equations

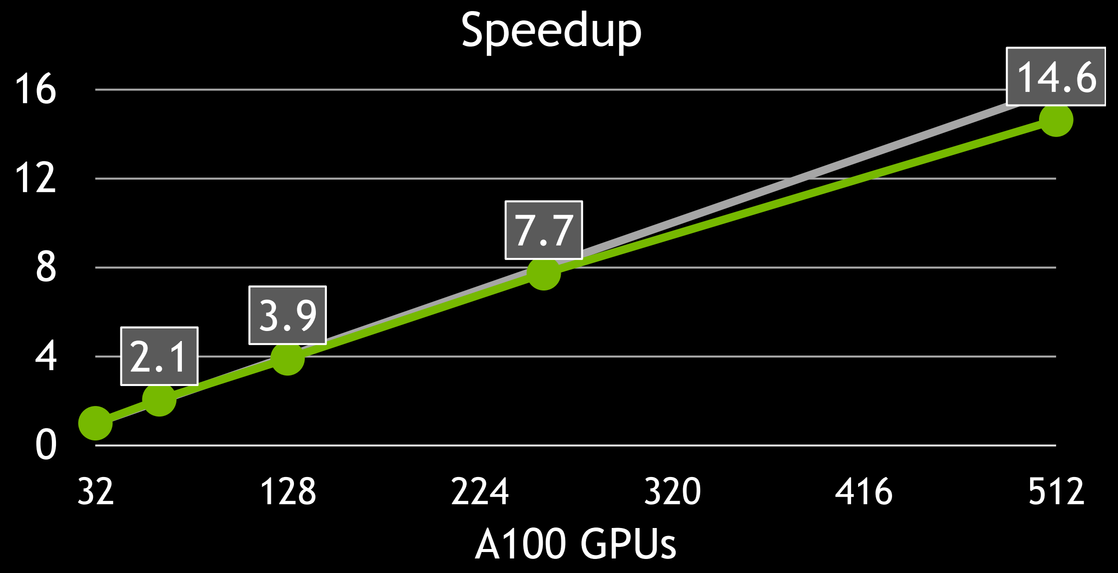
Change one line of code and scale from a single CPU thread up to a cluster of GPUs!




```
sync_wait(maxwell_eqs(multi_node_gpu_scheduler), …);
```

Palabos Carbon Sequestration





- Palabos is a framework for parallel computational fluid dynamics simulations using the Lattice-Boltzmann method.
- Code for multi-component flow through a porous media ported to C++ Senders and Receivers.
- Application: simulating carbon sequestration in sandstone.

Jonas Latt (University of Geneva), Christian Huber (Brown University)
Georgy Evtushenko (NVIDIA), Gonzalo Brito (NVIDIA)

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                      assert(bytes_read == sizeof(buf.size));
                      buf.data = make_unique<byte[]>(buf.size);
                      return span(buf.data.get(), buf.size);
                    })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                      assert(bytes_read == buf.size);
                      return move(buf);
                    });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```cpp
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                       assert(bytes_read == sizeof(buf.size));
                       buf.data = make_unique<byte[]>(buf.size);
                       return span(buf.data.get(), buf.size);
                     })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                       assert(bytes_read == buf.size);
                       return move(buf);
                     });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```cpp
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1)))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
struct dynamic_buffer {
  unique_ptr<byte[]> data;
  size_t size;
};

sender_of<set_value_t(dynamic_buffer)> auto async_read_array(auto handle) {
  return just(dynamic_buffer{})
       | let_value([handle] (dynamic_buffer& buf) {
           return just(as_writeable_bytes(span(&buf.size, 1))
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == sizeof(buf.size));
                    buf.data = make_unique<byte[]>(buf.size);
                    return span(buf.data.get(), buf.size);
                  })
                | async_read(handle)
                | then([&buf] (size_t bytes_read) {
                    assert(bytes_read == buf.size);
                    return move(buf);
                  });
         });
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (…) -> ex::sender auto {
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  …
}
```

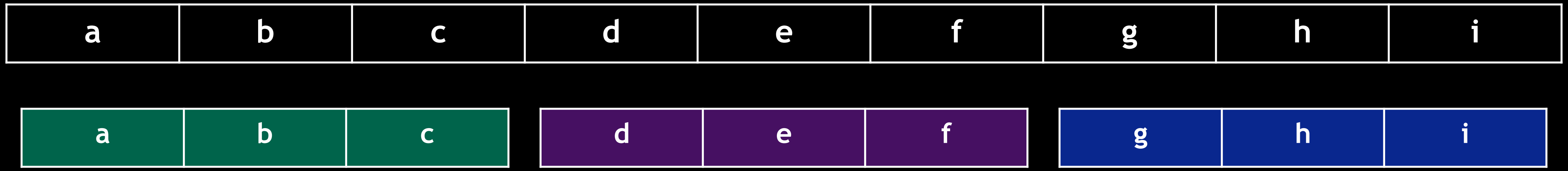
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    …
                  })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                      std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                      partials[0] = init;
                      return send_values(input, std::move(partials));
                    })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                     std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                     partials[0] = init;
                     return send_values(input, std::move(partials));
                   })
       | ex::bulk(tile_count,
                   [=] (std::size_t i, auto input, auto partials) {
                     …
                   })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                   std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                   partials[0] = init;
                   return send_values(input, std::move(partials));
                 })
       | ex::bulk(tile_count,
                 [=] (std::size_t i, auto input, auto partials) {
                   auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                   auto start      = i * tile_size;
                   auto end        = std::min(input.size(), (i + 1) * tile_size);
                   …
                 })
  …
}
```


a b c d e f g h i
a b c
d e f
g h i

a	b	c	d	e	f	g	h	i

a	ab	abc

`std::inclusive_scan`

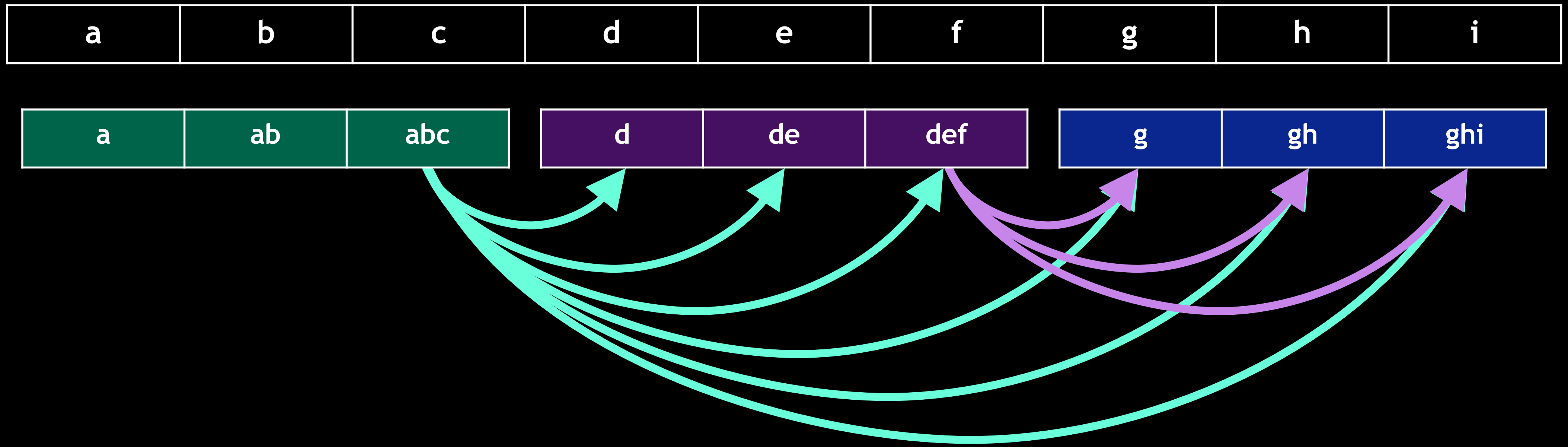
d	de	def

`std::inclusive_scan`

g	gh	ghi

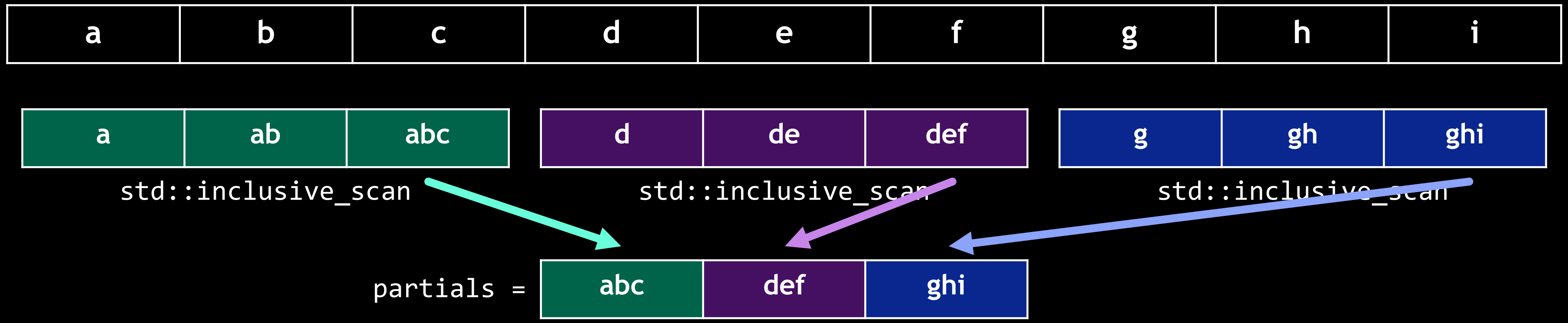
`std::inclusive_scan`

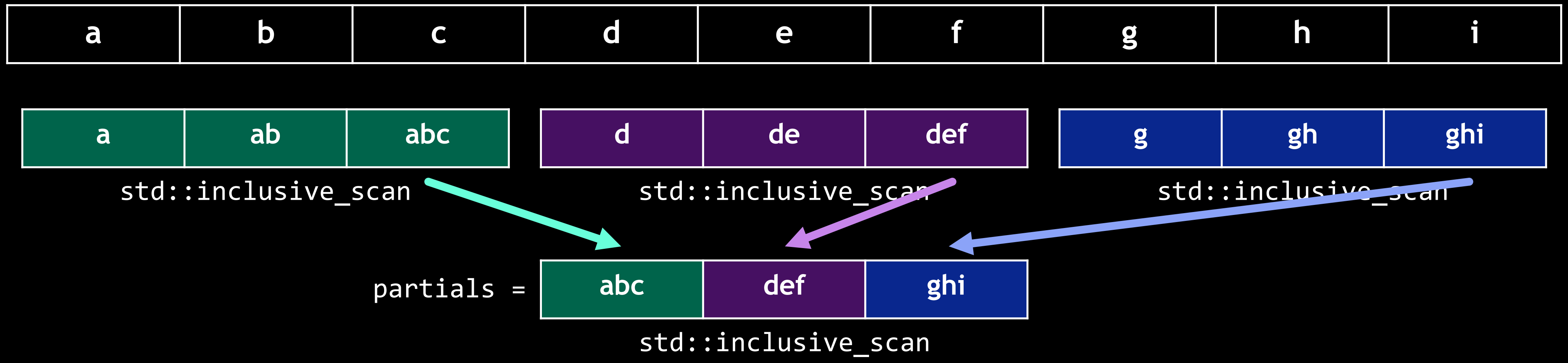
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                  std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                  partials[0] = init;
                  return send_values(input, std::move(partials));
                })
       | ex::bulk(tile_count,
                [=] (std::size_t i, auto input, auto partials) {
                  auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                  auto start      = i * tile_size;
                  auto end        = std::min(input.size(), (i + 1) * tile_size);
                  …                    std::inclusive_scan(begin(input) + start,
                                                           begin(input) + end,
                                                           begin(input) + start);

                })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                    auto start      = i * tile_size;
                    auto end        = std::min(input.size(), (i + 1) * tile_size);
                    …               = *--std::inclusive_scan(begin(input) + start,
                                                             begin(input) + end,
                                                             begin(input) + start);

                  })
  …
}
```

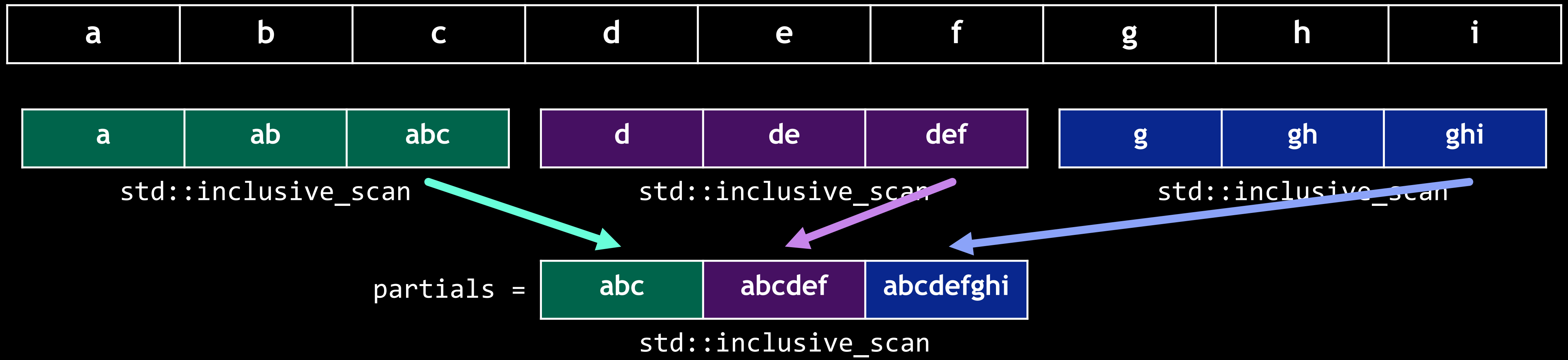
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
      | ex::then([=] (stdr::random_access_range auto input) {
                   std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                   partials[0] = init;
                   return send_values(input, std::move(partials));
                 })
      | ex::bulk(tile_count,
                 [=] (std::size_t i, auto input, auto partials) {
                   auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                   auto start      = i * tile_size;
                   auto end        = std::min(input.size(), (i + 1) * tile_size);
                   partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                            begin(input) + end,
                                                            begin(input) + start);
                 })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
def
ghi


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
def
ghi
std::inclusive_scan

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                     std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                     partials[0] = init;
                     return send_values(input, std::move(partials));
                   })
       | ex::bulk(tile_count,
                   [=] (std::size_t i, auto input, auto partials) {
                     auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                     auto start      = i * tile_size;
                     auto end        = std::min(input.size(), (i + 1) * tile_size);
                     partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                              begin(input) + end,
                                                              begin(input) + start);
                   })
       | ex::then([] (auto input, auto partials) {
                     std::inclusive_scan(begin(partials), end(partials), begin(partials));
                     …
                   })
  …
}
```

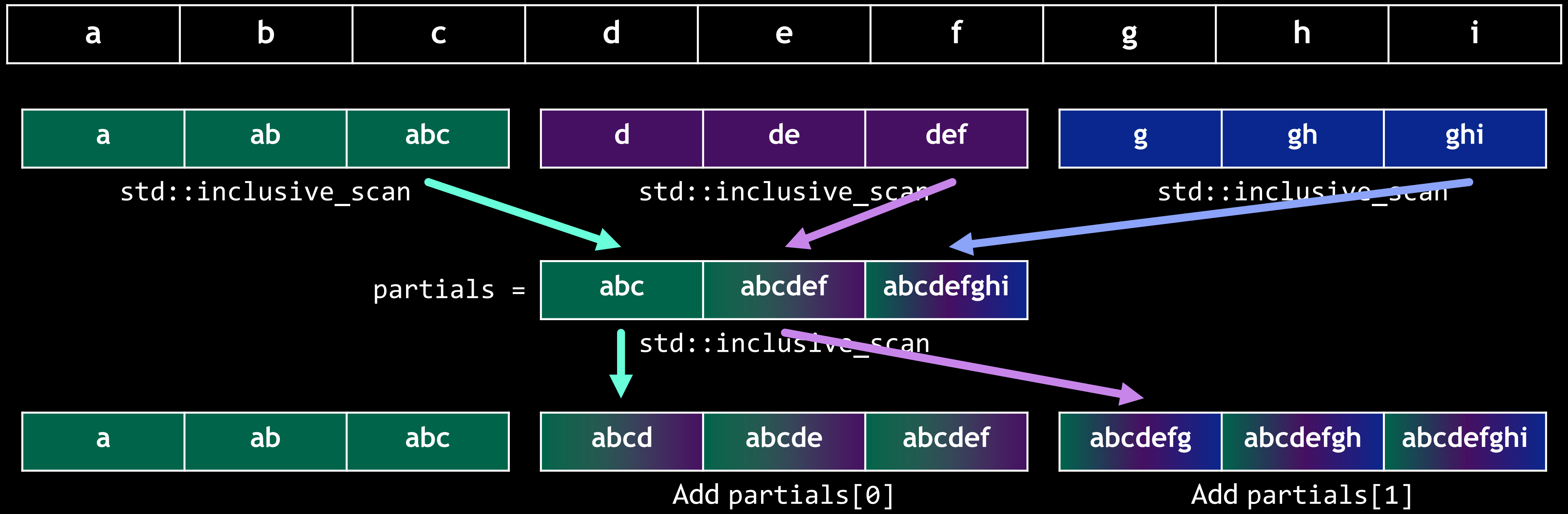
```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                  std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                  partials[0] = init;
                  return send_values(input, std::move(partials));
                })
       | ex::bulk(tile_count,
                [=] (std::size_t i, auto input, auto partials) {
                  auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                  auto start      = i * tile_size;
                  auto end        = std::min(input.size(), (i + 1) * tile_size);
                  partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                           begin(input) + end,
                                                           begin(input) + start);
                })
       | ex::then([] (auto input, auto partials) {
                  std::inclusive_scan(begin(partials), end(partials), begin(partials));
                  return send_values(input, std::move(partials));
                })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
d
de
def
g
gh
ghi
std::inclusive_scan
std::inclusive_scan
std::inclusive_scan
partials =
abc
abcdef
abcdefghi
std::inclusive_scan

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                    auto start      = i * tile_size;
                    auto end        = std::min(input.size(), (i + 1) * tile_size);
                    partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                             begin(input) + end,
                                                             begin(input) + start);
                  })
       | ex::then([] (auto input, auto partials) {
                    std::inclusive_scan(begin(partials), end(partials), begin(partials));
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    …
                  })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                    auto start      = i * tile_size;
                    auto end        = std::min(input.size(), (i + 1) * tile_size);
                    partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                             begin(input) + end,
                                                             begin(input) + start);
                  })
       | ex::then([] (auto input, auto partials) {
                    std::inclusive_scan(begin(partials), end(partials), begin(partials));
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size = (input.size() + tile_count - 1) / tile_count;
                    auto start     = i * tile_size;
                    auto end       = std::min(input.size(), (i + 1) * tile_size);
                    …
                  })
  …
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
      | ex::then([=] (stdr::random_access_range auto input) {
                  std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                  partials[0] = init;
                  return send_values(input, std::move(partials));
                })
      | ex::bulk(tile_count,
                [=] (std::size_t i, auto input, auto partials) {
                  auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                  auto start      = i * tile_size;
                  auto end        = std::min(input.size(), (i + 1) * tile_size);
                  partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                           begin(input) + end,
                                                           begin(input) + start);
                })
      | ex::then([] (auto input, auto partials) {
                  std::inclusive_scan(begin(partials), end(partials), begin(partials));
                  return send_values(input, std::move(partials));
                })
      | ex::bulk(tile_count,
                [=] (std::size_t i, auto input, auto partials) {
                  auto tile_size = (input.size() + tile_count - 1) / tile_count;
                  auto start     = i * tile_size;
                  auto end       = std::min(input.size(), (i + 1) * tile_size);
                  std::for_each(begin(input) + start, begin(input) + end,
                                [&] (auto& e) { e = partials[i] + e; });
                })
  …
}
```


a
b
c
d
e
f
g
h
i
a
ab
abc
std::inclusive_scan
d
de
def
std::inclusive_scan
g
gh
ghi
std::inclusive_scan
partials =
abc
abcdef
abcdefghi
std::inclusive_scan
a
ab
abc
abcd
abcde
abcdef
Add partials[0]
abcdefg
abcdefgh
abcdefghi
Add partials[1]

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                    auto start      = i * tile_size;
                    auto end        = std::min(input.size(), (i + 1) * tile_size);
                    partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                             begin(input) + end,
                                                             begin(input) + start);
                  })
       | ex::then([] (auto input, auto partials) {
                    std::inclusive_scan(begin(partials), end(partials), begin(partials));
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size = (input.size() + tile_count - 1) / tile_count;
                    auto start     = i * tile_size;
                    auto end       = std::min(input.size(), (i + 1) * tile_size);
                    std::for_each(begin(input) + start, begin(input) + end,
                                  [&] (auto& e) { e = partials[i] + e; });
                  })
       | ex::then([=] (auto input, auto partials) { return input; });
}
```

```
inline constexpr sender_adaptor auto
async::inclusive_scan = [] (ex::sender auto last, auto init, std::size_t tile_count) -> ex::sender auto {
  return last
       | ex::then([=] (stdr::random_access_range auto input) {
                    std::vector<stdr::range_value_t<decltype(input)>> partials(tile_count + 1);
                    partials[0] = init;
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size  = (input.size() + tile_count - 1) / tile_count;
                    auto start      = i * tile_size;
                    auto end        = std::min(input.size(), (i + 1) * tile_size);
                    partials[i + 1] = *--std::inclusive_scan(begin(input) + start,
                                                             begin(input) + end,
                                                             begin(input) + start);
                  })
       | ex::then([] (auto input, auto partials) {
                    std::inclusive_scan(begin(partials), end(partials), begin(partials));
                    return send_values(input, std::move(partials));
                  })
       | ex::bulk(tile_count,
                  [=] (std::size_t i, auto input, auto partials) {
                    auto tile_size = (input.size() + tile_count - 1) / tile_count;
                    auto start     = i * tile_size;
                    auto end       = std::min(input.size(), (i + 1) * tile_size);
                    std::for_each(begin(input) + start, begin(input) + end,
                                  [&] (auto& e) { e = partials[i] + e; });
                  })
       | ex::then([=] (auto input, auto partials) { return input; });
}
```

On The Horizon For C++

- Reflection
- Pattern Matching
- Senders

accu 2023

KEYNOTE:
C++ HORIZONS

BRYCE ADELSTEIN LELBACH

Thanks

- Jeff Garland
- Herb Sutter
- Andrei Alexandrescu
- Louis Dionne
- Daveed Vandevoorde
- Michael Park
- Michał Dominiak
- Georgy Evtushenko
- Lewis Baker
- Lucian Radu Teodorescu
- Lee Howes
- Kirk Shoop
- Michael Garland
- Eric Niebler
- Sean Baxter
- Kristen Shaker