Vectorizing a CFD Code With \`std::simd\` Supplemented by (Almost) Transparent Loading and Storing

OLAF KRZIKALLA

VECTORIZING A CFD CODE WITH STD::SIMD SUPPLEMENTED BY (ALMOST) TRANSPARENT LOADING AND STORING

German Aerospace Center (DLR)

Institute of Software Methods for Product Virtualization

DLR

Motivation: The Origin of the Talk

The task:

- Vectorization of time-consuming parts of a complex, existing code
- no revolutionary approach, please

The idea:

- use type deduction to load and store `std::simd` and scalar variables
- syntactically equalize scalar and vectorized code

The talk:

- share experience with vectorization using `std::simd`
- introduce the SIMD_ACCESS library

Background: Vectorization

Nowadays, all your CPUs can compute four times faster

$$\begin{array}{|c|}\hline x_1\\\hline x_2\\\hline x_3\\\hline x_4\\\hline\end{array} \quad \begin{bmatrix}+\\-\\*\\/\end{bmatrix} \quad \begin{array}{|c|}\hline y_1\\\hline y_2\\\hline y_3\\\hline y_4\\\hline\end{array} \quad = \quad \begin{array}{|c|}\hline z_1\\\hline z_2\\\hline z_3\\\hline z_4\\\hline\end{array}$$

One CPU instruction adds/multiplies/… multiple set of operands at once

→ Single Instruction Multiple Data (**SIMD**)

For more details Matthias Kretz‘ Cppcon talk about std::simd:
https://youtu.be/LAJ_hywLtMA

Background: Vectorization

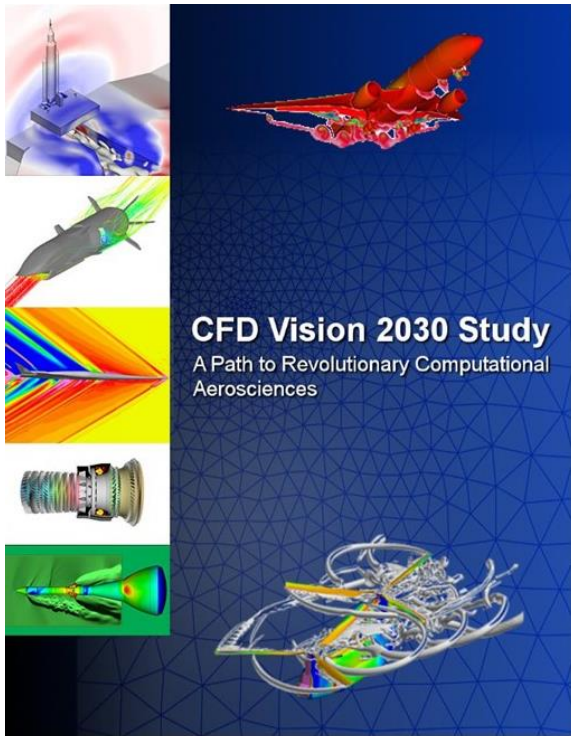
```
void add_array(double* x, double* y, double* z, int size)
{
  for (int i = 0; i < size; ++i)
    z[i] = x[i] + y[i];
}
```

Loop transformation
→ **Vectorization**
(explicit, guided or implicit)

```
void add_array(double* x, double* y, double* z, int size)
{
  for (int i = 0; i < size; i += simd_size)
    simd_value(z[i]) = simd_value(x[i]) simd_op(+) simd_value(y[i]);
}
```

Background: Cooperation CODA (CFD for ONERA, DLR, Airbus)

- Requirement to fundamentally re-engineer CFD software for extreme-scale parallel computing platforms was reported on a world-wide basis (cf. **CFD Vision 2030**):
- This necessity was also observed by ONERA, DLR and Airbus
- ONERA, DLR and Airbus decided in **2017** to jointly develop the new *next generation CFD* code **CODA** with a focus on:
  - modern algorithms
  - exploiting current and future HPC hardware architectures
  - making use of modern software engineering and code design

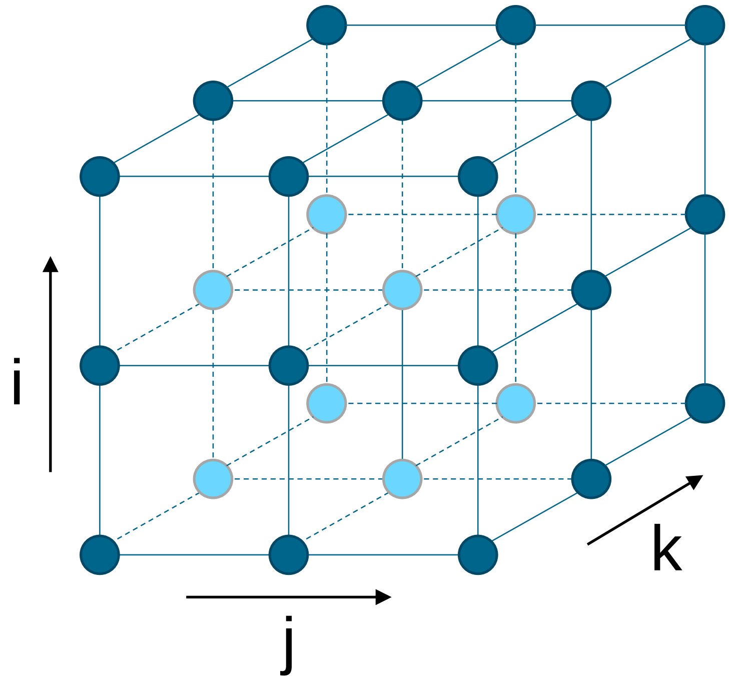

CFD Vision 2030 Study
A Path to Revolutionary Computational Aerosciences

DLR, ONERA and Airbus partnership
11 December 2017
Developing the next generation of Computational Fluid Dynamics together

*CODA is co-owned by ONERA, DLR and Airbus

Background: A particular CFD task in CODA

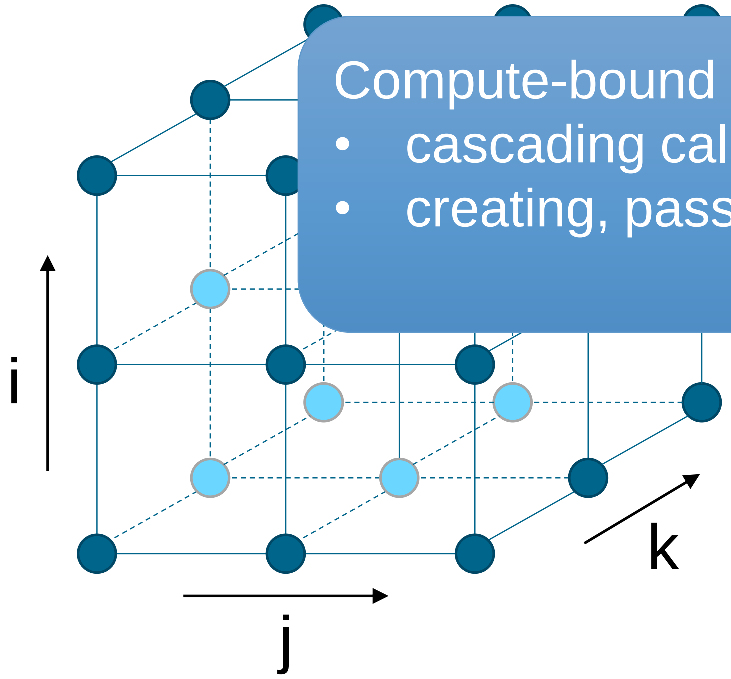
Compute flows in a cube



```
for dir : (0, 3)
  for i : (0, size)
    for j : (0, size)
      for k : (0, size)
        for c : (i|j|k + 1, size)
          compute_flow(point(...),
                       point(...));
```

Background: A particular CFD task in CODA

Compute flows in a cube



Compute-bound → worthwhile vectorization target

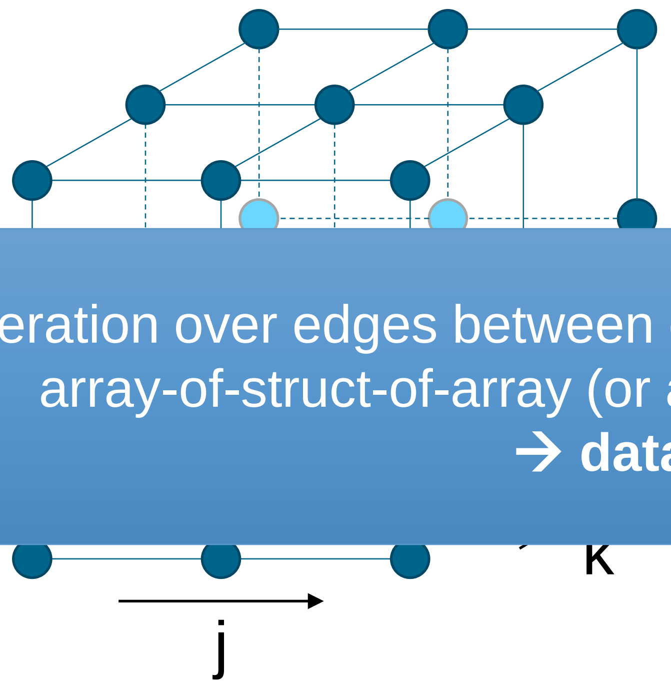
- cascading calls ending up partly in external libraries
- creating, passing and returning data structs

→ **explicit vectorization**

```
    for     (i|j|k + 1, size)
      compute_flow(point(...),
                   point(...));
```

Background: A particular CFD task in CODA

Compute flows in a cube in every spatial direction



```
for dir : (0, 3)
  fo        (0, size)
                                         e)
                                         .),
                                         .));
```

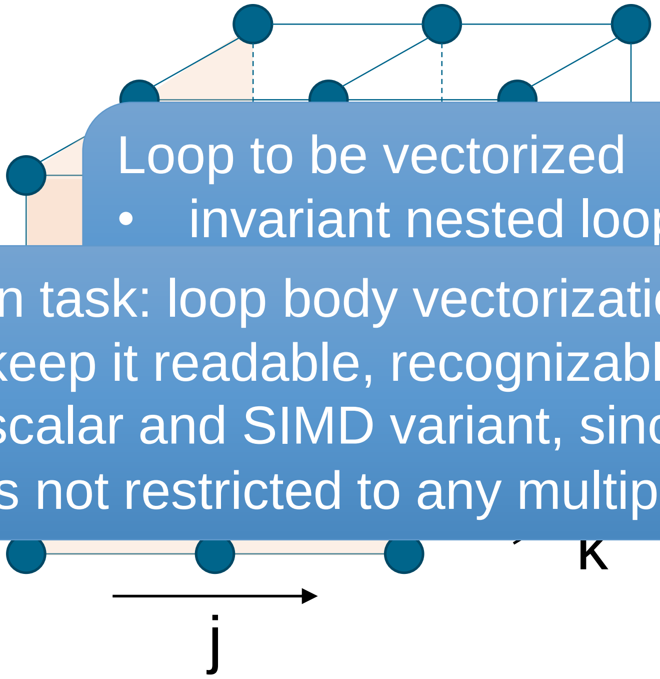
Iteration over edges between cube points in three directions: i, j, k

- array-of-struct-of-array (or array-of-struct-of-simd) data layout not viable

→ **data layout remains scalar**

Background: A particular CFD task in CODA

Compute flows in a cube in every spatial direction: loop preparation



Loop to be vectorized
- invariant nested loop length

Main task: loop body vectorization
- keep it readable, recognizable
- scalar and SIMD variant, since `size` is not restricted to any multiples

Loops rearranged and fusioned
- point coordinates computed using `dir`, `v` and `i|j|k`

```
for dir : (0, 3)
  for i|j|k : (0, size)
    for v : (0, size * size)
      for c : (i|j|k + 1, size)
        compute(point(...),
                point(...));
```

Vectorzation: The loop body

```
simd_value(z[i]) = simd_value(x[i]) simd_op(+) simd_value(y[i]);
```

Vector type information encoded in operand type
→ Correct implementation selected by
C++ operator overloading

```
simd_type operator+(simd_type x, simd_type y)
{
  return simd_add(x, y);
}
```

Already provided by std::simd (among other SIMD libraries)

Vectorzation: The loop body

```
simd_value(z[i]) = simd_value(x[i]) simd_op(+) simd_value(y[i]);
```

No vector type information in
`x, y, z` or `i`
→ Explicit load and store operations

```
auto result = simd_load(x, i) + simd_load(y, i)
simd_store(z, i, result);
```

Explicit access provided by std::simd

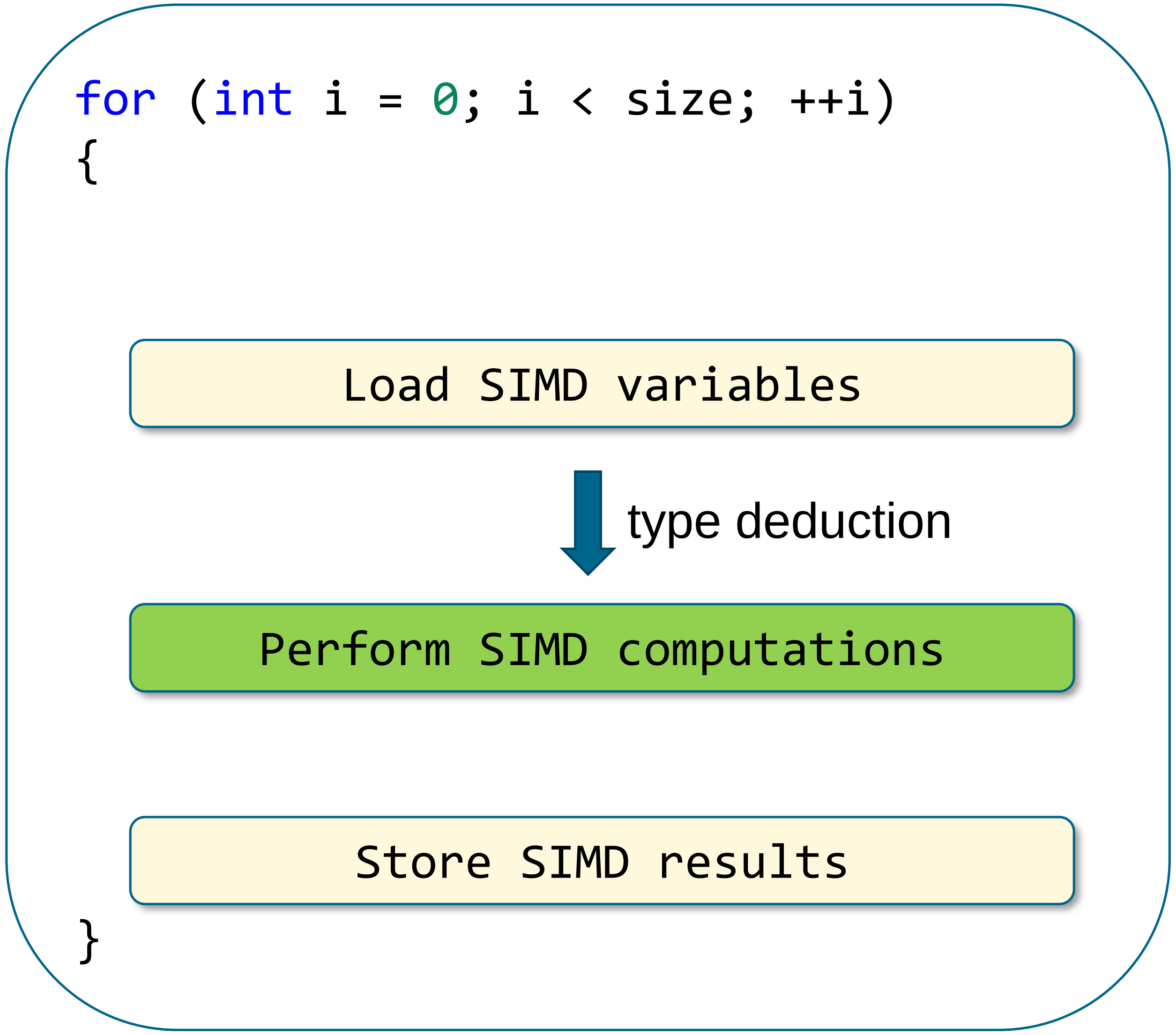
- Requires rearranging and rewriting the loop body
- Different scalar version

Vectorzation: The loop body

```
for (int i = 0; i < size; ++i)
{

    Load SIMD variables

    Perform SIMD computations

    Store SIMD results
}
```

Scalar versions often intertwined in one expression

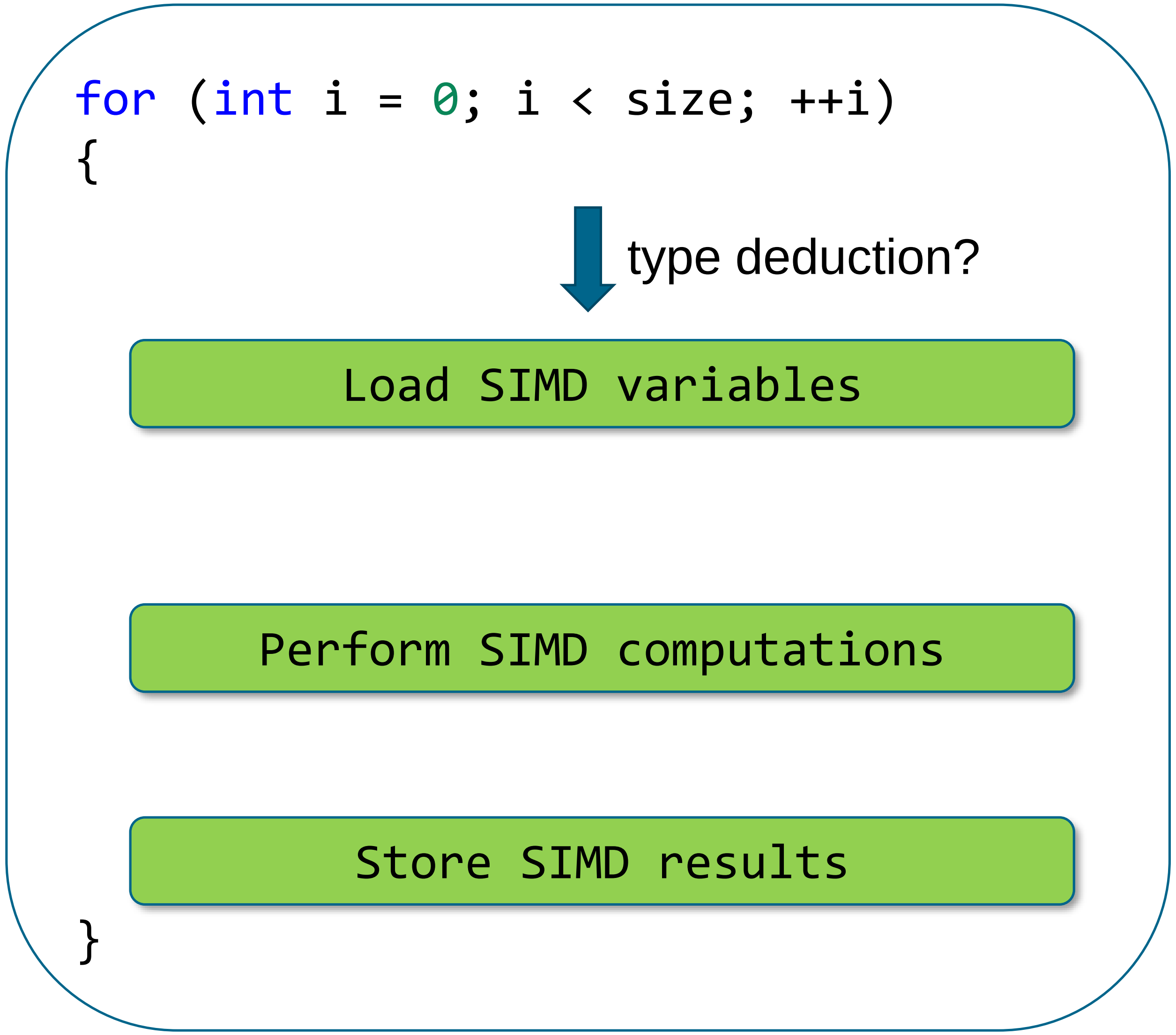
Vectorzation: The loop body



```
for (int i = 0; i < size; ++i)
{
```

```
}
```

Vectorzation: The loop body



```
for (int i = 0; i < size; ++i)
{
    Load SIMD variables

    Perform SIMD computations

    Store SIMD results
}
```

Vectorzation: The loop body

The core idea: encode the SIMD type information in the loop index

```
simd_value(z[i]) = simd_value(x[i]) simd_op(+) simd_value(y[i]);
```

- If `i` is an simd index (a newly introduced class) → SIMD version
- If `i` is an integral type (e.g. an integer) → scalar version

→ Loaded types deduced by SIMD index type, deduction propagates forward to operators

Vectorized loop body with overloaded `operator[](simd_index)`:

```
z[i] = x[i] + y[i];
```

- Same code for scalar and vector version
- No rewrite of the loop body required

Vectorzation: The loop body

A minor C++ detail: `operator[]` not yet globally overloadable

- Currently no general solution, since `index` is the second argument
- Possible workaround: macro

```
SIMD_ACCESS(z, i) = SIMD_ACCESS(x, i) + SIMD_ACCESS(y, i);
```

- Rewrite (but no rearrangement) necessary
- Same code for scalar and vector versions

SIMD_ACCESS Library

https://github.com/dlr-sp/simdize

Supplements `std::experimental::simd` (`std::simd` from C++26 onwards)

- Namespace `simd_access`

Proof-of-concept for simd indexing

- Demand-driven implementation
- Collection of the generalizable part of our vectorization

Platform for experiments and discussions of future developments

- Single header include provided

SIMD_ACCESS Basic Features: SIMD Index Types

Defining simd index types

- `concept simd_access::is_index`

```
simd_access::index<simd_size>
```

- consecutive sequence of elements starting at a given index

```
std::simd<std::is_integral, simd_size>
```

- Indirectly indexed elements

```
simd_access::index_array<simd_size, ArrayType>
```

- Indirectly indexed elements
- `ArrayType` may be a pointer or iterator to an array of indices

SIMD_ACCESS Basic Features: The SIMD_ACCESS Macro

Using index types

```
#define SIMD_ACCESS(array, index)
```

Replacement for the expression `array[index]`

- `index`: simd or scalar version possible
- Implementation expects a contiguous `array` with valid element at index 0
- Usable as lvalue: `SIMD_ACCESS(data, i) = some_value;`

SIMD_ACCESS Basic Features: The SIMD_ACCESS Macro

Using index types

```
#define SIMD_ACCESS(array, index)
```

Replacement for the expression `array[index]`

- `index`: simd or scalar version possible
- Implementation expects a contiguous `array` with valid element at index 0
- Usable as lvalue
  - internal type yielded
  - Use `SIMD_ACCESS_V` in deduced contexts:

```
template<is_simd T>
void foo(const T& simd_value);

foo(SIMD_ACCESS_V(data, i));
```

SIMD_ACCESS Basic Features: The SIMD_ACCESS Macro

Using index types

```
#define SIMD_ACCESS(array, index, expr)
```

Optional accessor to data members or elements of a subarray

- Replacement for the expression `array[index]expr`

```
// sub_arr[i][1]:
double sub_arr[100][3]                 -> SIMD_ACCESS(sub_arr, i, [1])
// pair_arr[i].first:
std::pair<double, int> pair_arr[100] -> SIMD_ACCESS(pair_arr, i, .first)
// pnt_sub_arr[i][0].first:
std::pair<...> pair_sub_arr[100][2]  -> SIMD_ACCESS(pair_sub_arr, i, [0].first)
```

SIMD_ACCESS Basic Features: Loops

Getting index types

```
template<int simd_size>
void loop(std::integral auto start, std::integral auto end, auto&& functor);
```

Linearly indexed vector loop

- Iteration over consecutive sequence `[start, end)`
- Generic functor called with `index<simd_size>` or for the residual iterations with a scalar index

SIMD_ACCESS Basic Features: Loops

Getting index types

```
template<int simd_size>
void loop(std::random_access_iterator auto start,
          std::random_access_iterator auto end, auto&& functor);
```

Indirectly indexed vector loop

- `static_assert(std::is_integral_v<decltype(*start)>);`
- Iteration over the range `[start, end)` of indices
- Generic functor called with `index_array<simd_size>`

SIMD_ACCESS Basic Features

Unified source code for vectorized and residual scalar iterations

```
void add_array(double* x, double* y, double* z, int size)
{
  const auto vector_size = std::native_simd<double>::size();
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    SIMD_ACCESS(z, i) = SIMD_ACCESS(x, i) + SIMD_ACCESS(y, i);
  });
}
```

Extended Features: Reflections

Passing around data structs

```
struct Point
{
  double x, y;
  Point operator+(const Point& other) const;
};


void add_pts(Point* x, Point* y, Point* z, int size)
{
  for (int i = 0; i < size; ++i)
  {
    z[i] = x[i] + y[i];
  }
}
```

	a		b
$Point_1$	x_1	$+$	x_1
	y_1	$+$	y_1
$Point_2$	x_2	$+$	x_2
	y_2	$+$	y_2
$Point_3$	x_3	$+$	x_3
	y_3	$+$	y_3
$Point_4$	x_4	$+$	x_4
	y_4	$+$	y_4

Extended Features: Reflections

Passing around data structs

- Conversion to structs-of-simd with the SIMD_ACCESS library

```
struct Point
{
  double x, y;
  Point operator+(const Point& other) const;
};

void add_pts(Point* x, Point* y, Point* z, int size)
{
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    SIMD_ACCESS(c,i) = SIMD_ACCESS(a,i)+SIMD_ACCESS(b,i);
  });
}
```

	a		b
SIMD $Point_1$	x_1	+	x_1
	x_2		x_2
	y_1	+	y_1
	y_2		y_2
SIMD $Point_2$	x_3	+	x_3
	x_4		x_4
	y_3	+	y_3
	y_4		y_4

Extended Features: Reflections

Step 1: define a simdized data type with **`simdized_value`**

- Works best with templated structs
- Note the recursive implementation

```
template<class T>
struct Point
{
  T x, y;
  Point operator+(const Point& other) const;
};

template<int SimdSize, class T>
inline auto simdized_value(const Point<T>& t)
{
  using simd_access::simdized_value;
  return Point{simdized_value<SimdSize>(t.x), simdized_value<SimdSize>(t.y)};
}
```

Extended Features: Reflections

Step 2: specify the simdized members with **`simd_members`**

- Again note the recursive implementation

```
template<class T>
struct Point
{
  T x, y;
  Point operator+(const Point& other) const;
};

template<class DestType, class SrcType, class FN>
inline void simd_members(Point<DestType>& d, const Point<SrcType>& s, FN&& func)
{
  using simd_access::simd_members;
  simd_members(d.x, s.x, func);
  simd_members(d.y, s.y, func);
}
```

Extended Features: Reflections

```
template<class T>
struct Point {
  T x, y;
  auto operator+(const Point& op2) const { return Point{ x + op2.x, y + op2.y }; }
};

template<int SimdSize, class T>
inline auto simdized_value(const Point<T>& t)
{
  using simd_access::si
  return Point{simdized
}

template<class DestType, class SrcType, class FN>
inline void simd_members(Point<DestType>& d, const Point<SrcType>& s, FN&& func)
{
  using simd_access::simd_members;
  simd_members(d.x, s.x, func);
  simd_members(d.y, s.y, func);
}

void add_point_array(const Point<double>* x, const Point<double>* y, Point<double>* z, int size)
{
  const auto vector_size = std::native_simd<double>::size();
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    SIMD_ACCESS(z, i) = SIMD_ACCESS(x, i) + SIMD_ACCESS(y, i);
  });
}
```

https://godbolt.org/z/s5cP7ehTj

Extended Features: Universal SIMD

```
void ComputeDataPoints(const SomeData& data)
{
  simd_access::loop<vector_size>(0, n, [&](auto i)
  {
    const auto point = data.Generate(i);
    for (int j = 0; j < d; ++j)
    {
      Compute( point.GetVelocity(j) );
    }
  });
}
```

- No indexed access
- complex struct, only few members used
- accessed in nested loop

Extended Features: Universal SIMD

SIMD-like class for non-arithmetic types

```
template<class T, int SimdSize>
struct universal_simd : std::array<T, SimdSize>
{
  static constexpr auto size() { return SimdSize; }   // ‘static’ missing in std::array

  template<class G>
  explicit universal_simd(G&& generator); // passing std::integral_constant like std::simd
};


template<class IndexType> requires(!std::integral<IndexType>)
inline decltype(auto) generate_universal(const IndexType& idx, auto&& generator)
{
  return universal_simd<decltype(generator(0)), idx.size()>(...);
}
```

Extended Features: Universal SIMD

Generator function for simd and scalar index:

```
void ComputePoints(const SomeData& data)
{
  loop<vec_size>(0, n, [&](auto i)
  {
    auto point = data.Generate(i);
    //...
  }
}
```

```
void ComputePoints(const SomeData& data)
{
  loop<vec_size>(0, n, [&](auto i)
  {
    auto point = generate_universal(i, [&](auto sc_i)
    { return data.Generate(sc_i); });
    //...
  });
}
```

Extended Features: Universal SIMD

References: use `std::ref`/`std::cref`

```
void ComputePoints(const SomeData& data)
{
  loop<vec_size>(0, n, [&](auto i)
  {
    const auto& point = data.Generate(i);
    //...
  }
}
```

```
void ComputePoints(const SomeData& data)
{
  loop<vec_size>(0, n, [&](auto i)
  {
    auto point = generate_universal(i, [&](auto sc_i)
    { return std::cref(data.Generate(sc_i)); });
    //...
  });
}
```

Extended Features: Universal SIMD

Accessing simdized members of universal_simd via macro:

```
#define SIMD_UNIVERSAL_ACCESS(value, expr)
```

- `value`: universal_simd or scalar variable
- `expr`: any expression valid in `value[0] expr`

Extended Features: Universal SIMD

Accessing simdized members

```
void ComputeDataPoints(const SomeData& data)
{
  for (int i = 0; i < n; ++i)
  {
    const auto point = data.Generate(i);
    for (int j = 0; j < d; ++j)
    {
      Compute(point.GetVelocity(j));
    }
  }
}
```

```
void ComputeDataPoints(const SomeData& data)
{
  simd_access::loop<vec_size>(0, n, [&](auto i)
  {
    auto point = generate_universal(i,
      [&](auto sc_i) { return data.Generate(sc_i); });
    for (int j = 0; j < d; ++j)
    {
      Compute(
        SIMD_UNIVERSAL_ACCESS(point, .GetVelocity(j)));
    }
  });
}
```

Extended Features: Handling of Conditions

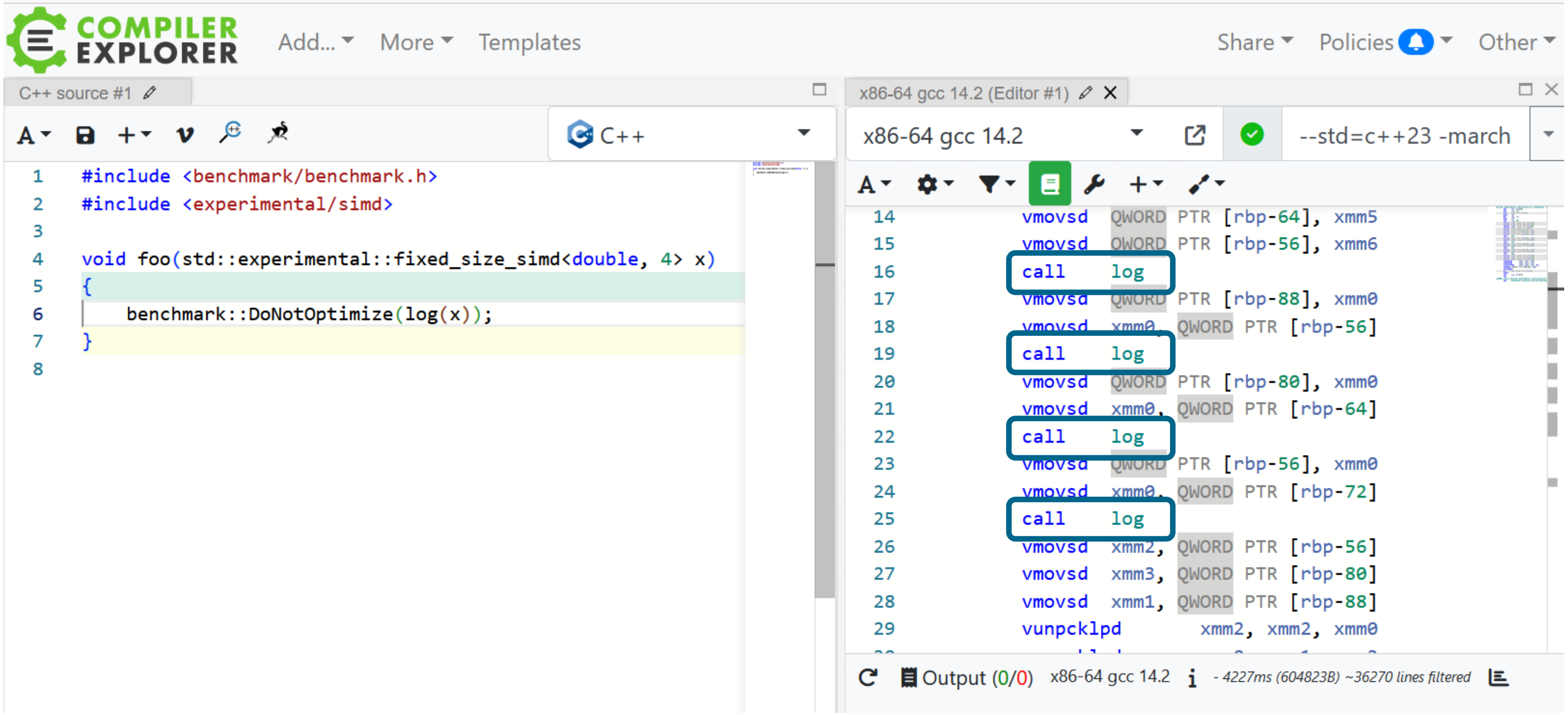
```
template<typename T>
T LogMean(const T &val1, const T &val2)
{
  // ...
  if (u < thres)
    twoF = 2.0 + u * (2.0 / 3.0 + u *
      (0.4 + 2.0 / 7.0 * u));
  else
    twoF = log(zeta) / f;
  // ...
}
```

```
template<typename T>
T LogMean(const T &val1, const T &val2)
{
  // ...
  auto mask = u < thres;
  if (stdx::any_of(mask))
    twoF = 2.0 + u * (2.0 / 3.0 + u *
      (0.4 + 2.0 / 7.0 * u));
  if (!stdx::all_of(mask))
    where(!mask, twoF) = log(zeta) / f;
  // ...
}
```

My assumption: time for all vector lanes == time for one vector lane

→ compute all vector lanes, if at least one is requested

Internally unrolled Log



```cpp
#include <benchmark/benchmark.h>
#include <experimental/simd>

void foo(std::experimental::fixed_size_simd<double, 4> x)
{
    benchmark::DoNotOptimize(log(x));
}
```

```
        vmovsd  QWORD PTR [rbp-64], xmm5
        vmovsd  QWORD PTR [rbp-56], xmm6
        call    log
        vmovsd  QWORD PTR [rbp-88], xmm0
        vmovsd  xmm0, QWORD PTR [rbp-56]
        call    log
        vmovsd  QWORD PTR [rbp-80], xmm0
        vmovsd  xmm0, QWORD PTR [rbp-64]
        call    log
        vmovsd  QWORD PTR [rbp-56], xmm0
        vmovsd  xmm0, QWORD PTR [rbp-72]
        call    log
        vmovsd  xmm2, QWORD PTR [rbp-56]
        vmovsd  xmm3, QWORD PTR [rbp-80]
        vmovsd  xmm1, QWORD PTR [rbp-88]
        vunpcklpd       xmm2, xmm2, xmm0
```

Internally unrolled Log

- Pragmatic solution necessary:

```
inline auto MaskedLog(const auto& mask, const auto& val)
{
  auto result = val;
  for (decltype(mask.size()) i = 0; i < mask.size(); ++i)
  {
    if (mask[i])
    {
      simd_members(result, result, [&](auto& d, const auto& s) { d[i] = log(s[i]); });
    }
  }
  return result;
}

{
  if (!stdx::all_of(mask))
    where(!mask, twoF) = MaskedLog(mask, zeta) / f;
}
```

Extended Features: Handling of Conditions

`simd_access::where_expression` for reflection support

```
template<class M, class T> requires(!is_stdx_simd<T>)
inline auto where(const M& mask, T& dest)
{
  return simd_access::where_expression<M, T>(mask, dest);
}

template<class M, class T>
struct where_expression {
  auto& operator=(const T& source) &&
  {
    simd_members(destination_, source, [&](auto& d, const auto& s) {
      using stdx::where, simd_access::where;
      where(mask_, d) = s;
    });
    return *this;
  }
};
```

Extended Features: Handling of Conditions

```
template<typename T>
T LogMean(const T &val1, const T &val2)
{
  // ...
  if (u < thres)
    twoF = 2.0 + u * (2.0 / 3.0 + u *
      (0.4 + 2.0 / 7.0 * u));
  else
    twoF = log(zeta) / f;
  // ...
}
```

```
template<typename T>
T LogMean(const T &val1, const T &val2)
{
  using stdx::where;
  using simd_access::where;
  // ...
  auto mask = u < thres;
  if (stdx::any_of(mask))
    twoF = 2.0 + u * (2.0 / 3.0 + u *
      (0.4 + 2.0 / 7.0 * u));
  if (!stdx::all_of(mask))
    where(!mask, twoF) = log(zeta) / f;
  // ...
}
```

Possible Extensions: gather/scatter Support

```
Point* dataPtr = //...;
simd_access::loop<vec_size>(0, size, [&](auto i)
{
  auto result = SIMD_ACCESS(dataPtr, i, .x);
  //...;
});
```

SIMD_ACCESS resolves to (casts omitted, `ElementSize==sizeof(Point)`)

```
template<size_t ElementSize, int SimdSize>
inline auto linear_load(const double* base)
{
  return stdx::fixed_size_simd<double, SimdSize>([&](auto i)
  { return *(base + ElementSize * i); });
}
```

also implementable as gather operation

```
template<size_t ElementSize, int SimdSize>
inline auto linear_load(const double* base)
{
  __m128i vindex {0x100000000ul, 0x300000002ul};
  return _mm256_i32gather_pd(base, vindex * ElementSize, 1);
}
```

Possible Extension: gather/scatter Support

```
loop<vec_size>(0, size, [&](auto i)
{
  auto result = SIMD_ACCESS_V(dataPtr, i,
   .first);
  benchmark::DoNotOptimize(result);
}, VectorResidualLoop);
```

```
loop<vec_size>(0, size, [&](auto i)
{
  stdx::fixed_size_simd<double, 4> result;
  __m128i vindex {0x200000000ul, 0x600000004ul};
  stdx::__data(result) = _mm256_i32gather_pd(
    &(dataPtr + i.scalar_index(0))->first,
    vindex, 8);
  benchmark::DoNotOptimize(result);
}, VectorResidualLoop);
```

	Broadwell	Skylake	Zen II
Intrinsic	~14 GB/s	~4 GB/s	~6 GB/s
SIMD_ACCESS	~29 GB/s	~40 GB/s	~28 GB/s

g++(11.4, 12.2) -g -march=core-avx2 -O3 -std=gnu++20; Broadwell: Intel(R) Xeon(R) CPU E5-1650 v4 @ 3.60GHz (AVX2); Skylake: Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz; Zen II: AMD EPYC 7702 64-Core Processor

Possible Extension: gather/scatter Support

```
.L144:
  vmovsd        32(%rax), %xmm1
  vmovsd        (%rax), %xmm0
  vmovhpd       48(%rax), %xmm1, %xmm1
  vmovhpd       16(%rax), %xmm0, %xmm0
  vinsertf128   $0x1, %xmm1, %ymm0, %ymm0
  vmovapd       %ymm0, 64(%rsp)
  addq          $4, %rdx
  addq          $64, %rax
  cmpq          %rdx, %r13
  ja  .L144
```

```
.L447:
  vmovapd       %ymm1, %ymm3
  vgatherdpd    %ymm3, (%rdx,%xmm2,8), %ymm0
  vmovapd       %ymm0, 64(%rsp)
  addq          $4, %rax
  addq          $64, %rdx
  cmpq          %rax, %r13
  ja  .L447
```

Possible reasons:
- Address computation (immediate vs. register addition)
- Parallel loads (2 or 3 per cycle)
- Downfall: Gather Data Sampling vulnerability CVE-2022-40982

g++(11.4, 12.2) -g -ma z (AVX2);
Skylake: Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz; Zen II: AMD EPYC 7702 64-Core Processor

Possible Extension: globally overloadable operator[]

```
namespace simd_access {

auto operator[](auto&& base, const is_index auto& index) { ... }

}
```

enables

```
void add_array(double* x, double* y, double* z, int size)
{
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    z[i] = x[i] + y[i];
  });
}
```

Not a new idea, mentioned e.g. in
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2128r6.pdf

Possible Extension: globally overloadable operator[]

```
namespace simd_access {

auto operator[](auto&& base, const is_index auto& index) { ... }

}
```

Workarounds provided by SIMD_ACCESS:

1. Named global function: `simd_access::sa(auto&& base, const is_index auto& index)`
2. Macro: `SIMD_ACCESS(base, index)`
3. Operator injection by `index_operator`:

```
template<typename... Args>
using vector = simd_access::index_operator<std::vector<Args...>>;

void add(vector<double>& x, vector<double>& y, vector<double>& z, int size)
{
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    z[i] = x[i] + y[i];
  });
}
```

Possible Extension: overloadable operator.()

```
template<typename T>
struct member_overload
{
  template<auto T::*Member>
  auto operator.();
};
```

enables

```
void add_array(Point* x, Point* y, Point* z, int size)
{
  simd_access::loop<vector_size>(0, size, [&](auto i)
  {
    z[i].x = x[i].x + y[i].x;
  });
}
```

- Deduced pointer-to-member template parameter `Member`
- The type of the object, whose member is accessed, must be known to enable the compiler to look up for the appropriate member name in the expression

Possible Extension: overloadable operator.()

```
template<typename T>
struct member_overload
{
  template<auto T::*Member>
  auto operator.();
};
```

Workarounds provided by SIMD_ACCESS: :

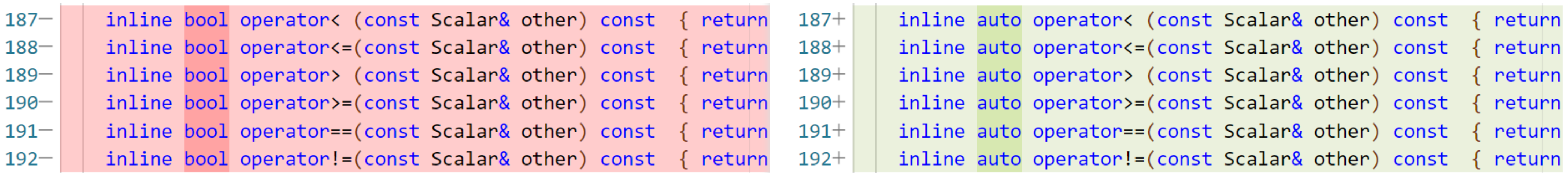
1. Named operator: `data[i].template dot<&Point::x>()`
2. Macro: `SIMD_ACCESS(data, i, .x)`
3. Named indexable using `std::tuple`: `data[i][nX]`

See also `test/potential_operator_overload.cpp`

Software Engineering Results

About 200 lines of code changed

- Expressions, control and data flow remain recognizable
- +120 lines added for reflections
- +20 lines patched in the Eigen library:



```
-    inline bool operator< (const Scalar& other) const  { return
-    inline bool operator<=(const Scalar& other) const  { return
-    inline bool operator> (const Scalar& other) const  { return
-    inline bool operator>=(const Scalar& other) const  { return
-    inline bool operator==(const Scalar& other) const  { return
-    inline bool operator!=(const Scalar& other) const  { return
```

```
+    inline auto operator< (const Scalar& other) const  { return
+    inline auto operator<=(const Scalar& other) const  { return
+    inline auto operator> (const Scalar& other) const  { return
+    inline auto operator>=(const Scalar& other) const  { return
+    inline auto operator==(const Scalar& other) const  { return
+    inline auto operator!=(const Scalar& other) const  { return
```

- + SIMD_ACCESS library

→ A lot of initial effort, but smooth vectorization afterwards

Performance Results: CODA test case

Wallclock time speedup, lower is better, <1 is an actual speedup

- AVX2, data type `double` (native vector size of 4)

speedup ($time_{SIMD}/time_{scalar}$)	Skylake	Broadwell
fixed_simd_size	4	4
test case 1 (loop count 9)	1,11	1,05
test case 2 (loop count 49)	1,39	1,07

Performance Results: Investigations

Vector vs. scalar instruction time

- AVX2, data type `double`

$time_{SIMD}/time_{scalar}$	Zen II	Skylake	Broadwell
Add	1,00	1,23	1,00
Mul	1,00	1,17	1,65
Div	1,00	1,98	4,01

→ single instruction in SIMD != single time step

Performance Results: CODA test case

Wallclock time speedup, lower is better, <1 is an actual speedup

- AVX2, data type `double` (native vector size of 4)

speedup ($time_{SIMD}/time_{scalar}$)	Skylake	Broadwell
fixed_simd_size	4	4
test case 1 (loop count 9)	1,11	1,05
test case 2 (loop count 49)	1,39	1,07

Performance Results: CODA test case

Wallclock time speedup, lower is better, <1 is an actual speedup

- AVX2, data type `double` (native vector size of 4)

speedup ($time_{SIMD}/time_{scalar}$)	Skylake	Broadwell	Broadwell-unroll
fixed_simd_size	4	4	8
test case 1 (loop count 9)	1,11	1,05	0,94
test case 2 (loop count 49)	1,39	1,07	0,91

→ Use std::simd‘s unroll feature!

Conclusion: Lessons Learned

std::simd is up and running

- Compilers make a decent optimization job
- Multiply your native simd size with a factor depending on your algorithm for best results

Vectorize conditions carefully

- Unused lanes may consume time

Use SIMD_ACCESS for easy vectorization

- Use auto everywhere
- Vectorization was smooth sailing once all required features had been implemented

Conclusion: Outlook

SIMD_ACCESS development

- Make it feature-complete.
- Reflection API

Discussion about globally overloadable operator[]()

- Who is interested in the topic?

Discussion about overloadable operator.()

- Could it hide too much complexity?

Imprint

Topic:	**Vectorizing a CFD Code with ‘std::simd‘ Supplemented by (Almost) Transparent Loading and Storing**
Date:	2024-09-17
Author:	Olaf Krzikalla
Institute:	DLR SP
Image sources:	All images “DLR (CC BY-NC-ND 3.0)” unless otherwise stated