C++ Data Layout Abstractions through Proxy Types

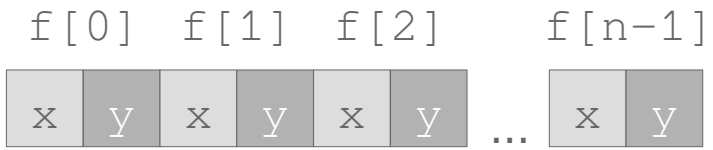
iWAPT2019, May 24th, Rio de Janeiro, Brazil

Florian Wende

Setting the context

Data layout of linearly indexed fields with structured element types

```
struct {
  double x,y;
} f[n];
```

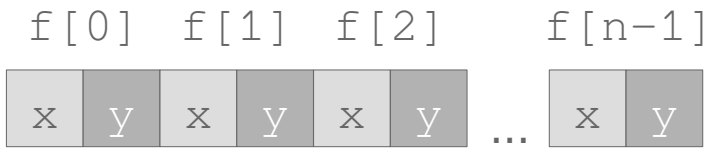


Setting the context

Data layout of linearly indexed fields with structured element types

```
struct {
  double x,y;
} f[n];
```

array-of-structures (AoS)
data layout



Setting the context

Data layout of linearly indexed fields with structured element types

```
struct {
  double x,y;
} f[n];
```

array-of-structures (AoS)
data layout


```
for(i=0;i<n;++i)
  f[i].x+=1.0;
```

strided data access

Setting the context

Data layout of linearly indexed fields with structured element types

- array-of-structures (AoS): natural layout in C/C++
  - the same members of successive elements are separated by the element size/extent
  - successive members of the same element are contiguous in memory

→ **performance penalty on SIMD platforms**

Setting the context

Data layout of linearly indexed fields with structured element types

- array-of-structures (AoS): natural layout in C/C++
  - the same members of successive elements are separated by the element size/extent
  - successive members of the same element are contiguous in memory

- **structure-of-arrays (SoA)**
  - the same members of all elements are placed in memory one after the other
  - successive members of the same element are separated by the field extent

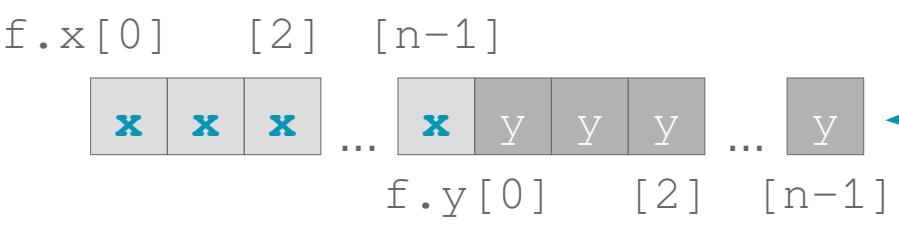
Setting the context

Data layout of linearly indexed fields with structured element types

```
struct {
  double x[n];
  double y[n];
} f;
```

structure-of-arrays (SoA)
data layout

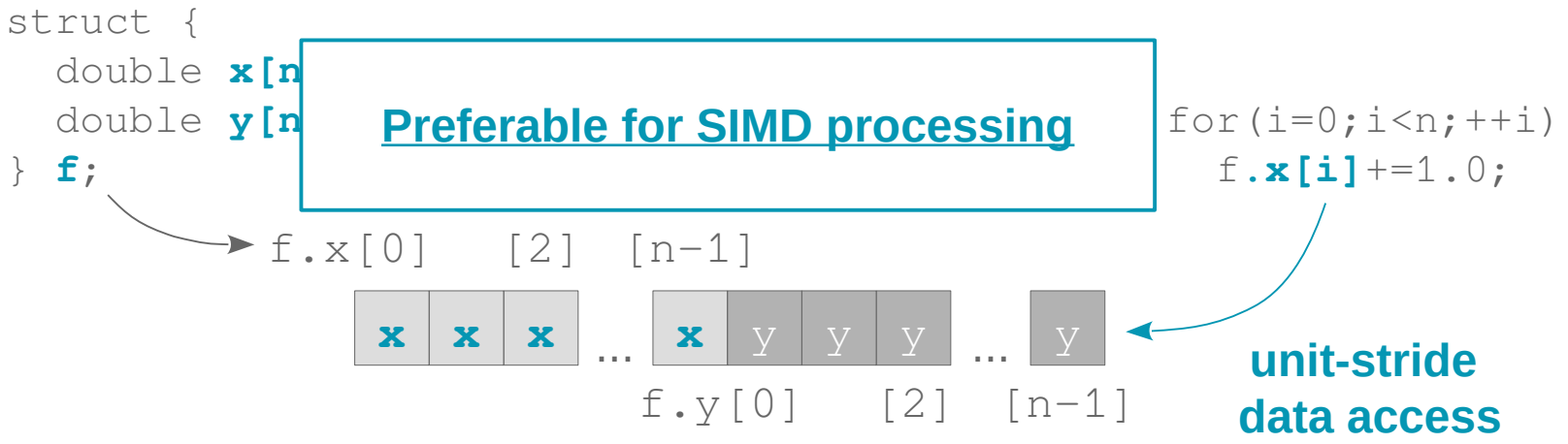
```
for(i=0;i<n;++i)
  f.x[i]+=1.0;
```



unit-stride data access

Setting the context

Data layout of linearly indexed fields with structured element types



Setting the context

Transition from AoS to SoA layout

- adapt field declarations &
  interchange data-member and field-element access

  `f[i].x` → `f.x[i]`

  `f[j][i].x` → `f.x[j][i]` or `f[j].x[i]`

Setting the context

Transition from AoS to SoA layout

- adapt field declarations &
  interchange data-member and field-element access

  `f[i].x` → `f.x[i]`

  `f[j][i].x` → `f.x[j][i]` or `f[j].x[i]`

→ **data-layout specific syntax conflicts with “convenient coding”**

Setting the context

Transition from AoS to SoA layout

- adapt field declarations &
  interchange data-member and field-element access

  `f[i].x` → `f.x[i]`

  `f[j][i].x` → `f.x[j][i]` or `f[j].x[i]`

- **transparent solution with AoS-like data access**
  - macro-based approaches (e.g. Intel SDLT)
    - definition/description of structured types through C-macros [& API calls for data access]

Setting the context

Transition from AoS to SoA layout

- adapt field declarations &
  interchange data-member and field-element access

  `f[i].x` → `f.x[i]`

  `f[j][i].x` → `f.x[j][i]` or `f[j].x[i]`

- **transparent solution with AoS-like data access**
  - macro-based approaches (e.g. Intel SDLT)
    - definition/description of structured types through C-macros [& API calls for data access]
  - **C++ proxy types**

Outline

- C++ proxy types
- A multi-dimensional AoS/SoA container
- Automated code generation with LibTooling
- Performance impact

Outline

- **C++ proxy types**
- A multi-dimensional AoS/SoA container
- Automated code generation with LibTooling
- Performance impact

Federal Ministry of Education and Research

C++ proxy types

act as a mediators for the actual structured types

- provide the same functionality

```
template <typename T>
struct A {
  T  x;
  T  y;
  ..
  A(T x,T y) : x(x),y(y) {}
  T foo(){..}
};
```

C++ proxy types

act as a mediators for the actual structured types

- provide the same functionality
- use **reference-valued members** to implement memory-access indirection

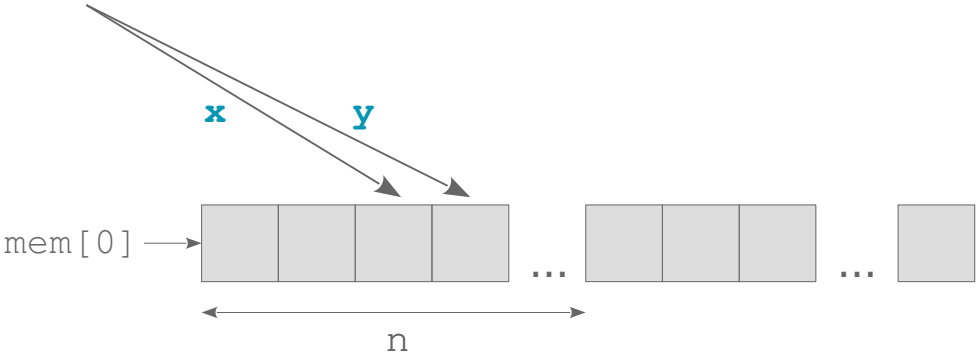
```
template <typename T>
struct A_proxy {
  T& x;
  T& y;
  ..
  A_proxy(T* base,int n) : x(base[0]),y(base[n]) {}
  T foo(){..}
};
```

C++ proxy types

act as a mediators for the actual structured types

- provide the same functionality
- use **reference-valued members** to implement memory-access indirection

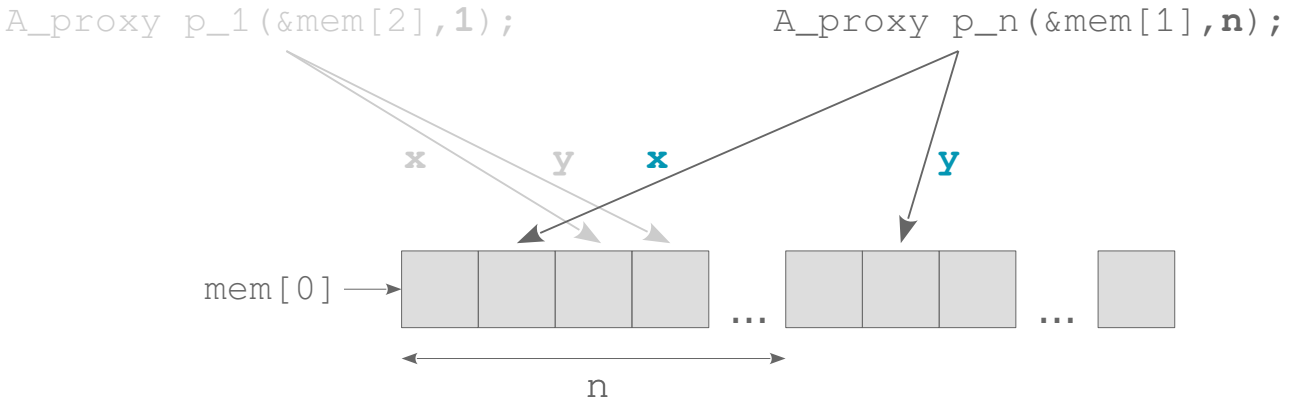
```
A_proxy p_1(&mem[2],1);
```



C++ proxy types

act as a mediators for the actual structured types

- provide the same functionality
- use **reference-valued members** to implement memory-access indirection



C++ proxy types

Proxy type definition: `A` → `A_proxy` (→ `A`)

members m_i of `A` with fundamental type `T`

- change "`T` m_i" → "`T&` m_i" in `A_proxy`

C++ proxy types

Proxy type definition: `A` → `A_proxy` (→ `A`)

member functions "`ret_t foo(T[&] arg)`"

- parameter(s): `T` equals `A` → add "`ret_t foo(A_proxy[&] arg)`" to `A` and `A_proxy`
- return type: to be adapted in `A_proxy`
  - `ret_t` equals `A` → change to `A_proxy` only if `*this` or any of foo's parameters is returned
  - `ret_t` equals `A&` → change to `A_proxy&`

C++ proxy types

Proxy type definition: `A` → `A_proxy` (→ `A`)

constructor(s)

- `A` is a "*homogeneous structured type*" (all members have the same type)
  - add "`A(A_proxy& a)..`" in `A`
  - replace all constructors in `A_proxy` by

```
A_proxy(T* base,int n)
    :m0(base[0*n]),m1(base[1*n]),m2(base[2*n]),..{}
```

C++ proxy types

Proxy type definition: `A` → `A_proxy` (→ `A`)

constructor(s)

- `A` is a "*homogeneous structured type*" (all members have the same type)
  - add "`A(A_proxy& a)..`" in `A`
  - replace all constructors in `A_proxy` by

```
A_proxy(T* base,int n)
    :m0(base[0*n]),m1(base[1*n]),m2(base[2*n]),..{}
```

- `A` is an "*inhomogeneous structured type*" → see the paper

Outline

- C++ proxy types
- **A multi-dimensional AoS/SoA container**
- Automated code generation with LibTooling
- Performance impact

Federal Ministry of Education and Research

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

```
          f[i].x += 1.0;
f[k]..[j][i].y += 1.0;
```

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

```
          f[i].x += 1.0;
f[k]..[j][i].y += 1.0;
```

container definition

- multi-dimensional: subscript operator chaining
- member access: right-most subscript operator `[]` returns proxy types for non-AoS layout

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

container definition

```
template <typename E,int D,layout L>
class container {
  typename traits<E,L>::mem_type* mem;
  extent<int,D> n;
  ..
  auto operator[](int i) {
    return accessor<E,D,L>(mem,n)[i];
  }
};
```

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

accessor definition (recursive)

```
template <typename E,int D,layout L,typename Enabled=void>
class accessor {
  ..
  accessor<E,D-1,L> operator[](int i) {
    int offset = some_func(i);
    return accessor<E,D-1,L>(&mem[offset],n);
  }
};
```

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

accessor definition (recursive, anchor)

```
template <typename E,layout L,typename Enabled>
class accessor<E,1,L,Enabled> {
  ..
  E& operator[](int i) {
    return mem[i];
  }
};
```

A multi-dimensional AoS/SoA container

Transparent AoS-like data access with different layouts:

accessor definition (recursive, anchor for SoA)

```
template <typename E>
class accessor<E,1,layout::SoA,enable_if<provides_proxy<E>::value>::type> {
  ..
  traits<E,layout::SoA>::proxy_type operator[](int i) {
    return traits<E,layout::SoA>::proxy_type(&mem[i],n[0]);
  }
};
```

A multi-dimensional AoS/SoA container

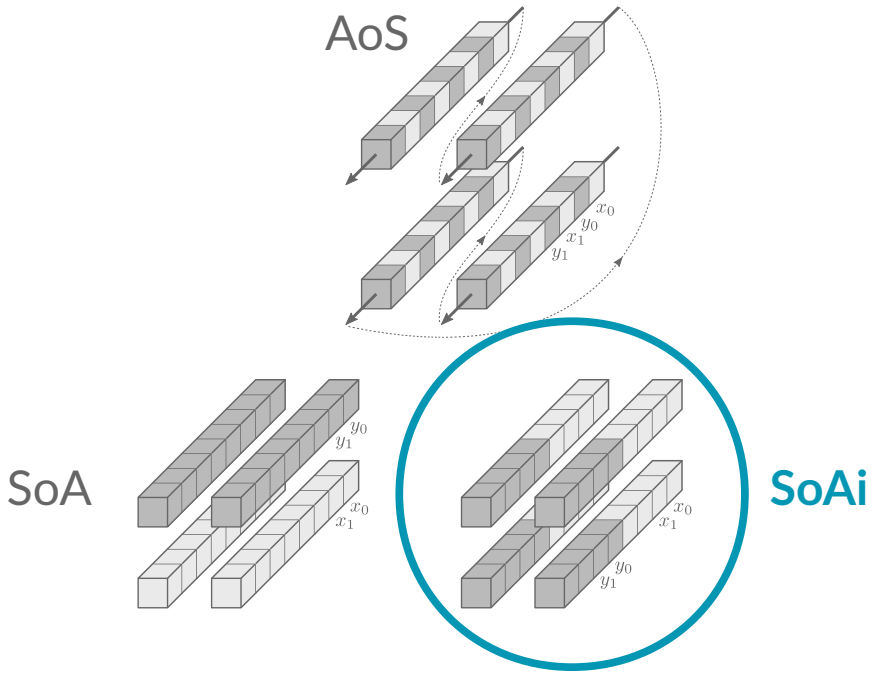
Transparent AoS-like data access with different layouts:

- core components of the container: accessor type + proxy types
  - unified implementation for “homogeneous” and “inhomogeneous” structured types
    → see the paper
  - additional features: data alignment, padding

A multi-dimensional AoS/SoA container

SoAi data layout

- innermost dimension uses the SoA layout
  - increased spatial locality for mixed member access within loops



Outline

- C++ proxy types
- A multi-dimensional AoS/SoA container
- **Automated code generation with LibTooling**
- Performance impact

Federal Ministry of Education and Research

Automated code generation with LibTooling

The proxy type approach allows us to **stay with the actual code base** mostly

- source code modifications comprise
  - definition of the proxy types
  - adaptation of field declarations: use our AoS/SoA container type
  - change the signature of all functions that are affected by the former two points

Automated code generation with LibTooling

The proxy type approach allows us to **stay with the actual code base** mostly

- source code modifications comprise
  - definition of the proxy types
  - adaptation of field declarations: use our AoS/SoA container type
  - change the signature of all functions that are affected by the former two points

→ automated adaptation of the source code

- no user intervention
- high-level source-to-source code transformation

Automated code generation with LibTooling

LibTooling: a library for writing standalone tools **based on Clang**

- operates on the **abstract syntax tree** (AST) of a C/C++ program (section)
  - code analysis & navigation

- **frontend actions**
  - match AST nodes and perform actions on them
  - modify text buffers referencing the source files

Federal Ministry of Education and Research

Automated code generation with LibTooling

AST matchers:

```
auto matcher = fieldDecl(allOf(
                         isPublic(),
                         hasName("x"),
                         hasParent(recordDecl().bind("target"))));
```

- node matcher (match a specific node type)
- narrowing matcher (match attributes of a node)
- traversal matcher (relation ships and traversal between nodes)

Automated code generation with LibTooling

AST matchers:

```
auto matcher = fieldDecl(..);

auto callback = [](const MatchFinder::MatchResult& result) {
  if(RecordDecl* decl = result.Nodes.getNodesAs<RecordDecl>("target"))
    cout << "class " << decl->getNameAsString() << " has public member x";
};

MatchFinder matchFinder;
matchFinder.addMatcher(matcher,callback);
matchFinder.matchAST(context);
```

Automated code generation with LibTooling

Code-trafo frontend action step 1: field declarations

- match **AST nodes of type `VarDecl`** that have element type `CXXRecordDecl`, e.g.

```
std::vector<std::array<A,4>> f_1;
std::array<std::array<A,5>,12> f_2;
```

or `ConstantArrayType`, e.g.

```
A f_3[35][3];
```

- **determine the element type** (here `A`) and field extent (if possible)
  - add the name of the element type to a **list of potential proxy type candidates**

Automated code generation with LibTooling

Code-trafo frontend action step 2: proxy candidate detection

- match all declarations/definitions of structured types that are on the "potential candidate" list

```
CXXRecordDecl(hasName(“..”))
ClassTemplate[PartialSpecialization]Decl(hasName(“..”))
```

  - collect meta data: proxy candidate requirements
    - fundamental member types only
    - no polymorphism (not abstract)

Automated code generation with LibTooling

Code-trafo frontend action step 3a: adapt original structured type(s)

- for all proxy candidates `A`, modify the containing source file as follows
  - add forward declaration of `A_proxy`
  - add constructor "`A(A_proxy& a)..`" to `A`
  - for member functions taking parameters of type `A[&]`,
    add equivalent definitions taking parameters of type `A_proxy[&]` instead
  - ...

Automated code generation with LibTooling

Code-trafo frontend action step 3b: create proxy types

- for all proxy candidates `A`, create the proxy type `A_proxy`
  - duplicate the declaration/definition of A
  - replace all constructors by "`A_proxy(T* base,int n)..`"
  - for all members m_i change "`T` m_i" → "`T&` m_i"
  - for member functions taking/returning parameters of type `A[&]`, add equivalent definitions taking/returning parameters of type `A_proxy[&]` instead
  - ...

Automated code generation with LibTooling

Code-trafo frontend action step 3b: create proxy types

- for all proxy candidates `A`, create the proxy type `A_proxy`
  - duplicate the declaration/definition of A
  - replace all constructors by "`A_proxy(T* base,int n)..`"
  - for all members m_i change "`T` m_i" → "`T&` m_i"
  - for member functions taking/returning parameters of type `A[&]`, add equivalent definitions taking/returning parameters of type `A_proxy[&]` instead
  - ...

Note: this holds for homogeneous structured types
for inhomogeneous structured types → see the paper

Automated code generation with LibTooling

Code-trafo frontend action step 4: adapt field declarations

e.g. "`vector<array<A,4>> f_4(n)`" → "`container<A,2,layout::SoA> f_4{{4,n}}`"

Automated code generation with LibTooling

Code-trafo frontend action step 4: adapt field declarations

e.g. "`vector<array<A,4>> f_4(n)`" → "`container<A,2,layout::SoA> f_4{{4,n}}`"

Code-trafo frontend action step 5: add "`#include`" lines to all relevant files

Automated code generation with LibTooling

Code-trafo frontend action step 4: adapt field declarations

e.g. "`vector<array<A,4>> f_4(n)`" → "`container<A,2,layout::SoA> f_4{{4,n}}`"

Code-trafo frontend action step 5: add "`#include`" lines to all relevant files

Code-trafo frontend action step 6: write all changes back to source files

Outline

- C++ proxy types
- A multi-dimensional AoS/SoA container
- Automated code generation with LibTooling
- **Performance impact**

Federal Ministry of Education and Research

Performance impact

Prototype implementation of the proxy generator and source-to-source transformation tool

`https://github.com/flwende/code_transformation`
(see the `test` branch for the latest status)

Performance impact

Prototype implementation of the proxy generator and source-to-source transformation tool

Test cases

- math kernel: `log`, `exp` function calls in 3d loop
- real-world code: MIDG2 (electrical engineering) → see the paper

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

```
for(int k=0;k<n[2];++k)
  for(int j=0;j<n[1];++j)
    for(int i=0;i<n[0];++i)
      y[k][j][i]   = [log|exp](x[k][j][i]))
```

full element processing

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

```
for(int k=0;k<n[2];++k)
  for(int j=0;j<n[1];++j)
    for(int i=0;i<n[0];++i)
      y[k][j][i].y = [log|exp](x[k][j][i])).y
```

element member processing

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

```
for(int k=0;k<n[2];++k)
  for(int j=0;j<n[1];++j)
    for(int i=0;i<n[0];++i)
      y[k][j][i].y = [log|exp](x[k][j][i])).y
```

element types

`vec<double,3>` and `tuple<uint16,double,uint32>`

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

```
for(int k=0;k<n[2];++k)
  for(int j=0;j<n[1];++j)
    for(int i=0;i<n[0];++i)
      y[k][j][i].y = [log|exp](x[k][j][i])).y
```

element types

`vec<double,3>` and `tuple<uint16,double,uint32>`

compiler
gnu-8.2 + libmvec (glibc-2.27)
clang++-9 + SVML (Intel 2019)

system
Intel Xeon Gold 6138 (Skylake, AVX512)
CentOS-7.5 with Linux kernel 3.10

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

AoS

```
.L55:
 vmovsd 16(%rbx), %xmm0
 addq   $24, %rbx
 call   __log_finite
 vmovsd %xmm0, 16(%rsp)
 vmovsd -16(%rbx), %xmm0
 call   __log_finite
 vmovsd %xmm0, 8(%rsp)
 vmovsd -24(%rbx), %xmm0
 call   __log_finite
 ..
 cmpq   %rbx, %rbp
 jne    .L55
```

SoA[i]

```
.L55:
 vmovupd (%r14,%r15), %zmm0
 call    _ZGVeN8v___log_finite
 vmovapd %zmm0, -176(%rbp)
 vmovupd 0(%r13,%r15), %zmm0
 call    _ZGVeN8v___log_finite
 vmovapd %zmm0, -112(%rbp)
 vmovupd (%rbx,%r15), %zmm0
 call    _ZGVeN8v___log_finite
 vmovapd -112(%rbp), %zmm2
 ..
 cmpq    -184(%rbp), %r15
 jne     .L55
```

Assembly output
full `vec<double,3>`

Performance impact

Test case **math kernel**: `log`, `exp` function calls in 3d loop

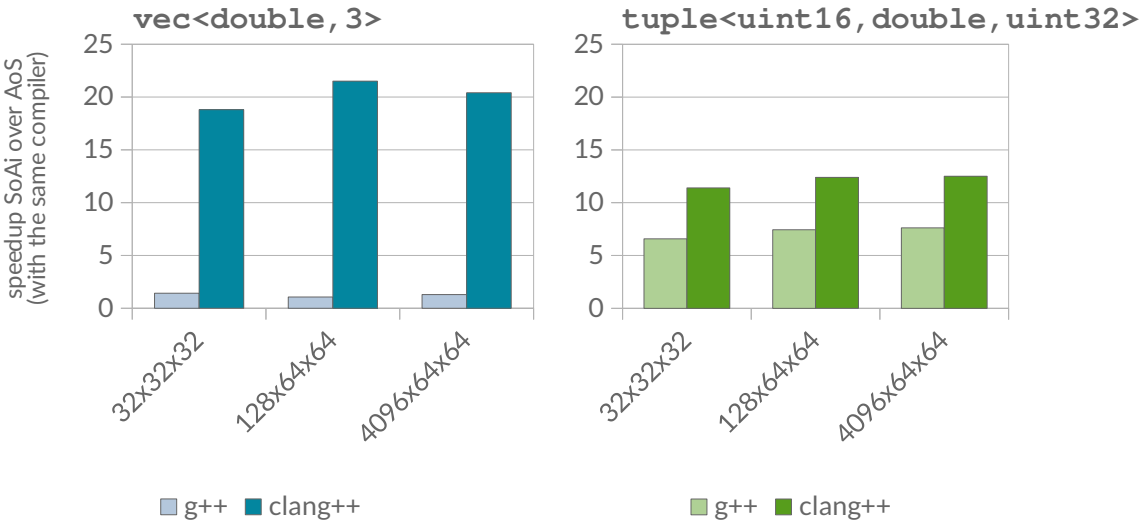
AoS (OpenMP SIMD pragma)

```
.L55:
 ..
 vpermt2pd -128(%rbx), %zmm19, %zmm6
 vmovapd   %zmm4, %zmm5
 vmovapd   %zmm6, %zmm0
 vpermt2pd -64(%rbx), %zmm18, %zmm5
 vpermt2pd -64(%rbx), %zmm20, %zmm0
 vmovapd   %zmm5, -176(%rbp)
 call      _ZGVeN8v___log_finite
 vmovapd   %zmm0, -240(%rbp)
 vmovapd   -176(%rbp), %zmm0
 call      _ZGVeN8v___log_finite
 vmovapd   %zmm0, -176(%rbp)
 vmovapd   -112(%rbp), %zmm0
 call      _ZGVeN8v___log_finite
 vmovapd   -176(%rbp), %zmm2
 ..
```

Assembly output
full `vec<double,3>`

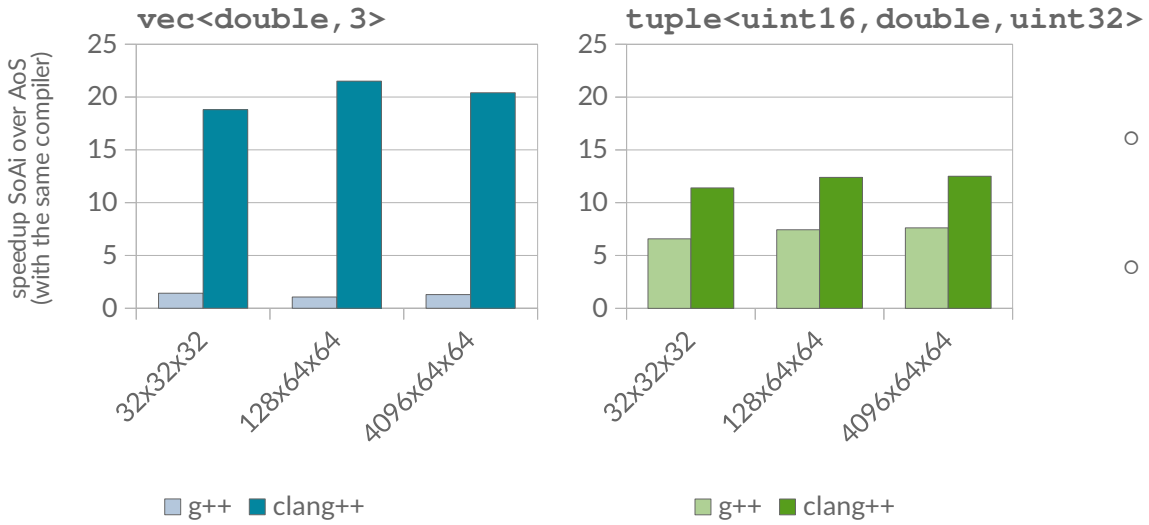
Performance impact

Test case **math kernel**: `log`, `exp` function calls on **full elements** in 3d loop



Performance impact

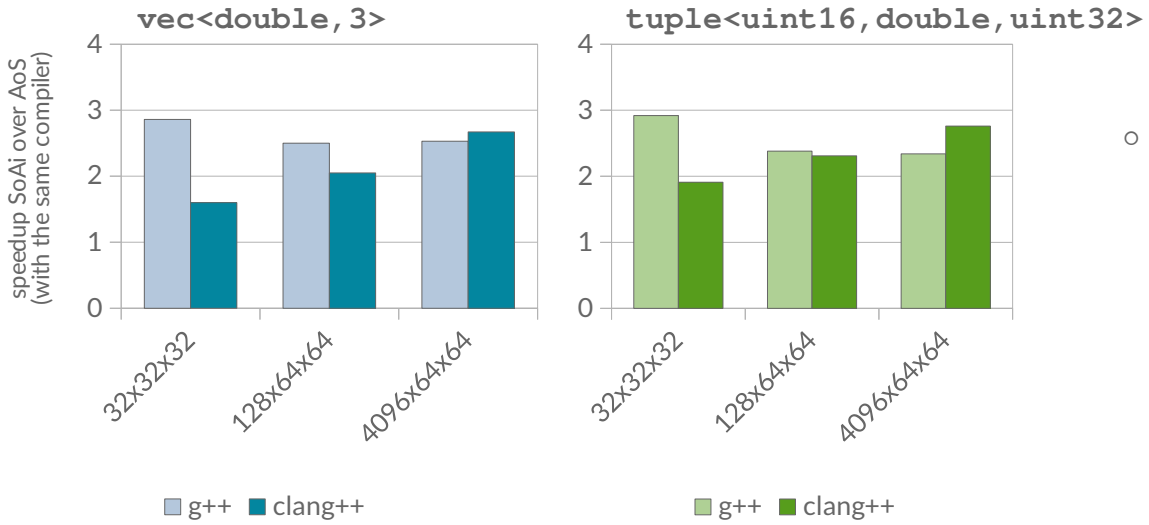
Test case **math kernel**: `log`, `exp` function calls on **full elements** in 3d loop



- `g++` vectorizes the AoS case with OpenMP SIMD pragma
- `clang++` uses SIMD match calls from Intel's SVML for SoA[i] and serial math calls for AoS → larger speedups

Performance impact

Test case **math kernel**: `log`, `exp` function calls on **element member** in 3d loop



- both `g++` and `clang++` generate SIMD vector instructions and SIMD math calls for AoS and SoAi
  → gains reflect the impact of the contiguous memory access

The big picture

- preferable data layout for SIMD processing: SoA[i]
- proxy approach for SoA[i] layout with AoS-like data access
  - works with both `g++` and `clang++`
    - Intel compiler: work in progress (Intel does not fully support C++-17 yet)
  - not macro-based: programmer does not need to code against an API
- automated proxy-type generator and source-to-source transformation tool
  - generates high-level C++ code: can be compiler with any C++-17 compiler
    - existing code base remains as is mostly
    - no transformation(s) on the IR-level

Thanks for listening

Contact: Florian Wende
wende@zib.de
Zuse Institute Berlin (ZIB)

Git repo: `https://github.com/flwende/code_transformation`

Funding: BMBF grant 01IH16005 (HighPerMeshes)

Federal Ministry of Education and Research