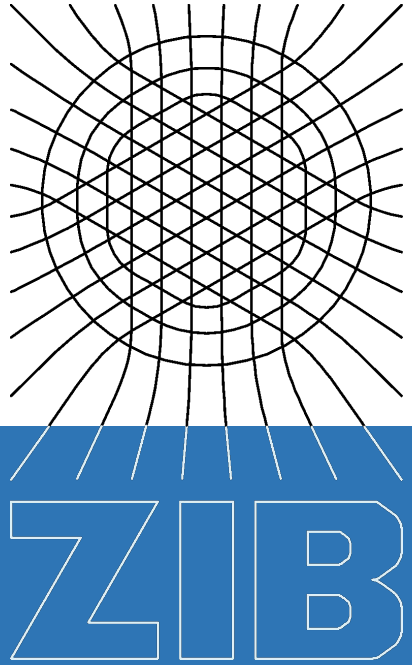
DHPCC++ 2018 Conference
St Catherine's College, Oxford, UK
May 14th, 2018

SYCL-based Data Layout Abstractions for CPU+GPU Codes



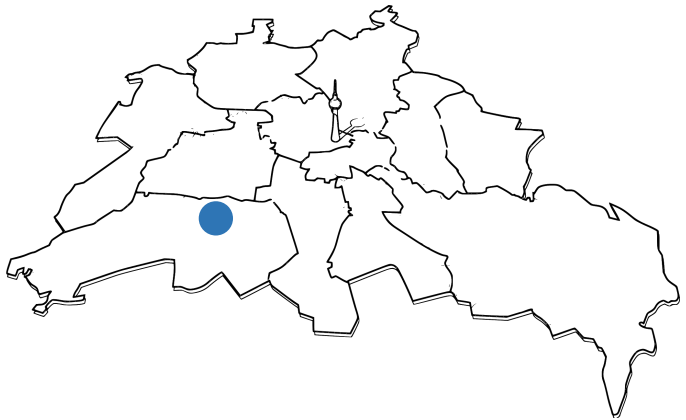
Florian Wende, Matthias Noack,
Thomas Steinke

Zuse Institute Berlin

Setting the context

Zuse Institute Berlin (ZIB)

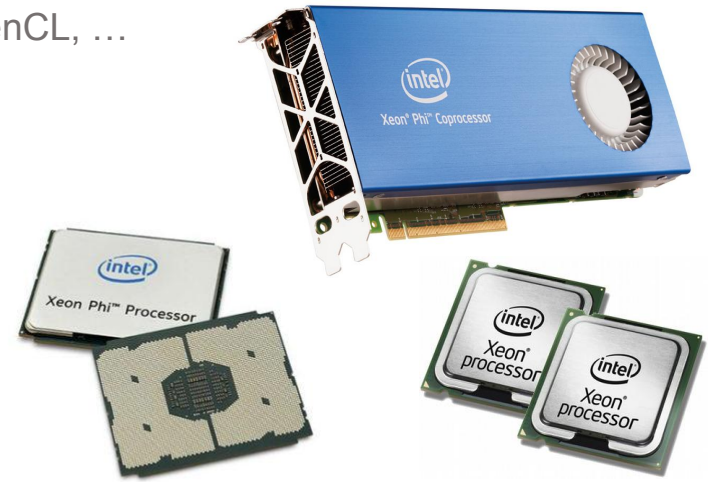
- Mathematics departments - Life and Material Science & Optimization
- **Computer science department - Algorithms for Innovative Architectures**
  - multi- & many-core and (GP)GPU computing in HPC
  - parallel programming models, languages and extensions
  - HPC user consulting



Setting the context

Zuse Institute Berlin (ZIB)

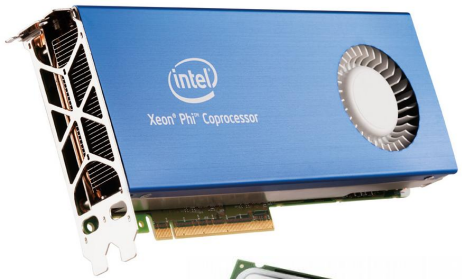
- Mathematics departments - Life and Material Science & Optimization
- **Computer science department - Algorithms for Innovative Architectures**
  - *Intel Parallel Computing Center (since 2013)*
    - Intel Xeon Phi programming: OpenMP, OpenCL, …



Setting the context

Zuse Institute Berlin (ZIB)

- Mathematics departments - Life and Material Science & Optimization
- **Computer science department - Algorithms for Innovative Architectures**
  - *Intel Parallel Computing Center (since 2013)*
    - Intel Xeon Phi programming: OpenMP, OpenCL, …
  - *HighPerMeshes (since 2017)*
    - domain specific programming on unstructured grids using a DSL: C++, SYCL, ...



Setting the context

HighPerMeshes - domain specific programming on unstructured grids

- DSL constructs
  - grid representation / distribution / access
  - mathematical operations
- target platform aware compiler infrastructure
  - C++, SYCL, code transformations (clang/llvm)
  - multi-processing / -threading and SIMD optimizations
  - heterogeneous platforms
- data structures
  - vector data types + operations
  - container / buffer data types

Setting the context

HighPerMeshes - domain specific programming on unstructured grids

- DSL constructs
  - grid representation / distribution / access
  - mathematical operations
- target platform aware compiler infrastructure
  - C++, SYCL, code transformations (clang/llvm)
  - multi-processing / -threading and SIMD optimizations
  - heterogeneous platforms
- data structures
  - vector data types + operations
  - **container / buffer data types** →

- **multi-dimensional fields**
- **different data layouts**
- **contiguous memory**
- **support for heterogeneous platforms**

Convenient coding

Example: how the code should look like

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
buffer<vec3<float>, 3> field_1({nx, ny, nz}), field_2({nx, ny, nz});

for (k = 0; k < nk; ++k)
  for (j = 0; j < nj; ++j)
    for (i = 0; i < ni; ++i) {
      field_1[k][j][i] = some_scalar_func(i, j, k);
      field_2[k][j][i] = log(field_1[k][j][i].x);
      field_1[k][j][i].z *= -1.0F;
    }

field_1.memcpy_h2d();
// do some computation on the GPU using field_1
```

Convenient coding

Example: how the code should look like

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
buffer<vec<float>, 3> field_1({nx, ny, nz}), fi

for (k = 0; k < nk; ++k)
  for (j = 0; j < nj; ++j)
    for (i = 0; i < ni; ++i) {
      field_1[k][j][i] = some_scalar_func(i, j,
      field_2[k][j][i] = log(field_1[k][j][i].x
      field_1[k][j][i].z *= -1.0F;
    }

field_1.memcpy_h2d();
// do some computation on the GPU using field_1
```

- multi-dimensional fields

Convenient coding

Example: how the code should look like

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
buffer<vec<float>, 3> field_1({nx, ny, nz}), fi

for (k = 0; k < nk; ++k)
  for (j = 0; j < nj; ++j)
    for (i = 0; i < ni; ++i) {
      field_1[k][j][i] = some_scalar_func(i, j,
      field_2[k][j][i] = log(field_1[k][j][i].x
      field_1[k][j][i].z *= -1.0F;
    }

field_1.memcpy_h2d();
// do some computation on the GPU using field_1
```

- **multi-dimensional fields**
- **different data layouts: AoS vs. SoA**

  element access should be contiguous for SIMD optimizations

Convenient coding

Example: how the code should look like

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
buffer<vec<float>, 3> field_1({nx, ny, nz}), fi

for (k = 0; k < nk; ++k)
  for (j = 0; j < nj; ++j)
    for (i = 0; i < ni; ++i) {
      field_1[k][j][i] = some_scalar_func(i, j,
      field_2[k][j][i] = log(field_1[k][j][i].x
      field_1[k][j][i].z *= -1.0F;
    }

field_1.memcpy_h2d();
// do some computation on the GPU using field_1
```

- multi-dimensional fields
- **contiguous memory**

  just one data transfer

- **different data layouts: AoS vs. SoA**

  element access should be contiguous for SIMD optimizations

- **support for heterogeneous platforms**

Convenient coding

Multi-dimensional fields: data alignment

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[j][i] = log(a[j][i]);
```

Convenient coding

Multi-dimensional fields: data alignment

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[j][i] = log(a[j][i]);
```

Declaration 1 (straightforward):

```
constexpr size_t Align = 32; // e.g. AVX(2)
alignas(Align) float a[nj][ni];
```

only OK if *n* is a multiple of (`Align / sizeof(float)`)

Convenient coding

Multi-dimensional fields: data alignment

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[j][i] = log(a[j][i]);
```

Padding row length `ni` makes sure, each row starts aligned.

Declaration 2 (with padding rows length):

```
constexpr size_t Align = 32; // e.g. AVX(2)
constexpr size_t nAlign = Align / sizeof(float);
constexpr nip = ((ni + nAlign - 1) / nAlign) * nAlign;
alignas(Align) float a[nj][nip];
```

Convenient coding

Multi-dimensional fields: data alignment

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[j][i] = log(a[j][i]);
```

Declaration 3 (dynamically sized):

```
constexpr size_t Align = 32; // e.g. AVX(2)
using allocator = boost::alignment::aligned_allocator<float, Align>;
vector<vector<float, allocator>> a(nj, vector<float, allocator>(ni));
```

Convenient coding

Multi-dimensional fields: data alignment

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[j][i] = log(a[j][i]);
```

Declaration 3 (dynamically sized):

```
constexpr size_t Align = 32; // e.g. AVX(2)
using allocator = boost::alignment::aligned_allocator<float, Align>;
vector<vector<float, allocator>> a(nj, vector<float, allocator>(ni));
```

Nice! ... but not contiguous in memory

Convenient coding

Multi-dimensional fields: data alignment + contiguous memory

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[index(j, i, nip)] = log(a[index(j, i, nip)]);
```

Declaration 4 (dynamically sized):

```
...
vector<float, allocator> a(nj * nip);
auto index = [&](size_t j, size_t i, size_t n){ return j * n + i; } // index function
```

Contiguous in memory! … but not nice

Convenient coding

Multi-dimensional fields: data alignment + contiguous memory

```
for (j = 0; j < nj; ++j) // row
  for (i = 0; i < ni; ++i) // column
    a[index(j, i, nip)] = log(a[index(j, i, nip)]);
```

- **can be copied as a whole**

  e.g. to / from GPU or network

- **data locality: less TLB misses**

Declaration 4 (dynamically sized):

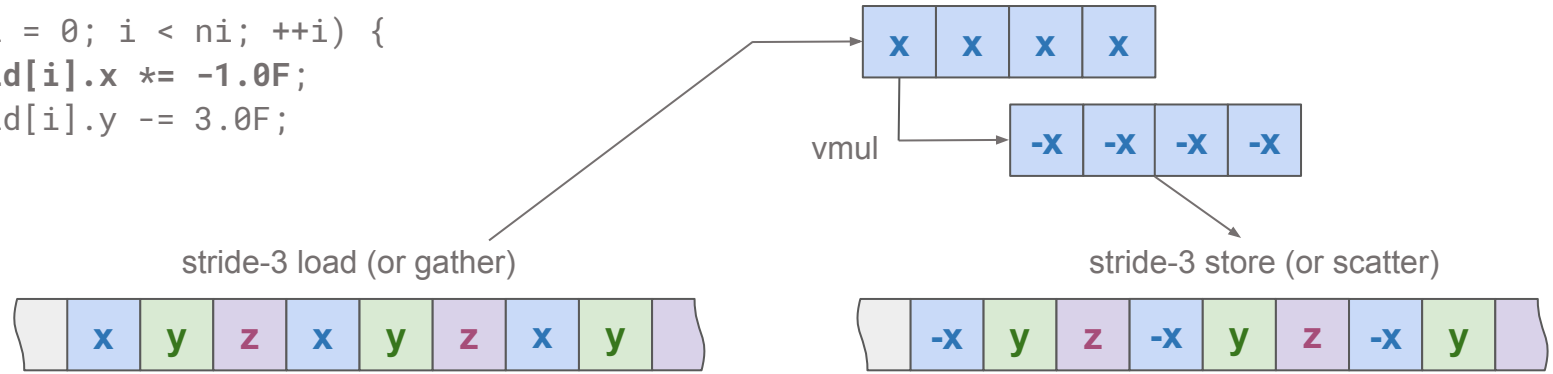
```
...
vector<float, allocator> a(nj * nip);
auto index = [&](size_t j, size_t i, size_t n){ return j * n + i; } // index function
```

Contiguous in memory! … but not nice

Convenient coding

Data layout: Array of Structs (AoS)

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
vec3<float> field[ni];

for (i = 0; i < ni; ++i) {
  field[i].x *= -1.0F;
  field[i].y -= 3.0F;
}
```



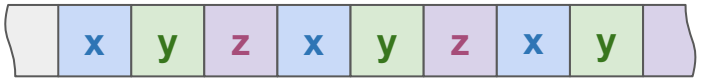
Convenient coding

Data layout: Array of Structs (AoS)

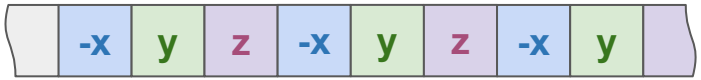
```
template <typename T>
using vec3 = struct { T x; T y; T z; };
vec3<float> field[ni];

for (i = 0; i < ni; ++i) {
  field[i].x *= -1.0F;
  field[i].y -= 3.0F;
}
```

- **definitely an optimization issue on standard CPUs (even with SLP SIMD)**
- **GPU: might help increase the ILP with multiple component accesses**

stride-3 load (or gather)



stride-3 store (or scatter)



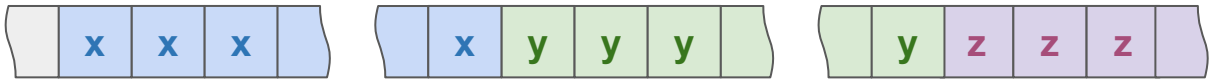
Convenient coding

Data layout: Struct of Arrays (SoA)

```
template <typename T>
using vec3 = struct { T x; T y; T z; };
vec3<float> field[ni];

for (i = 0; i < ni; ++i) {
  field[i].x *= -1.0F;
  field[i].y -= 3.0F;
}
```

At least on the CPU side, we would like to have something like this

How to?



Convenient coding

Data layout: observation (code from the field of electrical engineering)

About 3x better performance on GPU (AMD FirePro) with (Codeplay's) `cl::sycl::vec<T, D>` instead of our own `vec<T, D>` data type

- data layout then is AoS

  need to be compared against SoA

- element access through `.x()`, `.y()`, `.z()`, ..

  requires some code changes

Implementation: vector data type

Mixing our `vec<T, D>` with (Codeplay's) `cl::sycl::vec<T, D>`

```
template <typename T, size_t D>
class vec;
```

https://github.com/flwende/data_types/tree/master/include/vec

Implementation: vector data type

Mixing our `vec<T, D>` with (Codeplay's) `cl::sycl::vec<T, D>`

```
template <typename T>
class vec<T, 3> {
  ..
public:
  using fundamental_type = T;
  static constexpr size_t dim = 3;
  ..
  union {
    cl::sycl::vec<T, 3> sycl_vec;
    struct { T x; T y; T z; };
  };
  // our vector implementation
};
```

https://github.com/flwende/data_types/tree/master/include/vec

Implementation: vector data type

Mixing our `vec<T, D>` with (Codeplay's) `cl::sycl::vec<T, D>`

```
template <typename T>
class vec<T, 3> {
  ..
public:
  using fundamental_type = T;
  static constexpr size_t dim = 3;
  ..
  union {
    cl::sycl::vec<T, 3> sycl_vec;
    struct { T x; T y; T z; };
  };
  // our vector implementation
};
```

- **direct member access**

```
vec<float, 3> v;
v.x = 3.0F; // not v.x() = 3.0F
```

- **for SoA data layout, the `sycl_vec` member can be used on GPU**

https://github.com/flwende/data_types/tree/master/include/vec

Implementation: buffer data type for CPU

The buffer type

```
template <typename T, size_t D,.., data_layout Layout,..>
class buffer {
  using TT = typename type_info<T, Layout>::mapped_type;
  vector<TT,..> m_data; // 1D vector: contiguous memory allocation
  ..
public:
  sarray<size_t, D> size;
  buffer(const sarray<size_t, D>& size) : size(size) { ; }
  ..
  inline accessor<T, D, Layout> read() const {
    return accessor<T, D, Layout>(&m_data[0], size);
  }
  ..
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The buffer type

```
template <typename T, size_t D,.., data_layout Layout,..>
class buffer {
  using TT = typename type_info<T, Layout>::mapped_type;
  vector<TT,..> m_data; // 1D vector: contiguous memory allocation
  ..
public:
  sarray<size_t, D> size;
  buffer(const sarray<size_t, D>& size) : s
..
  inline accessor<T, D, Layout> read() cons
    return accessor<T, D, Layout>(&m_data[0
  }
};
```

Definition of `type_info<T, Layout>`

general **T**

- **AoS** → **mapped_type = T**
- **SoA** → **mapped_type = T**

T = vec<X, D>

- **AoS** → **mapped_type = T**
- **SoA** → **mapped_type = X**

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The accessor type (AoS data layout = default)

```
template <typename T, size_t D, data_layout Layout, typename Enabled=void>
class accessor<T, D, Layout, Enabled> {
  T* ptr;
  sarray<size_t, D> n;
  ..
public:
  accessor(T* ptr, sarray<size_t, D>& n) : ptr(ptr), n(n) { ; }
  ..
  inline accessor<T, D-1, Layout> operator[](size_t idx) {
    size_t offset = idx * n.reduce_mul<D-1>();
    return accessor<T, D-1, Layout>(&ptr[offset], n);
  }
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The accessor type (AoS data layout = default)

```
template <typename T, data_layout Layout, typename Enabled>
class accessor<T, 1, Layout, Enabled> {
  T* ptr;
  sarray<size_t, 1> n;
  ..
public:
  accessor(T* ptr, sarray<size_t, 1>& n) : ptr(ptr), n(n) { ; }
  ..
  inline T& operator[](size_t idx) {
    return ptr[idx];
  }
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The accessor type (SoA data layout)

```
template <typename T, size_t D>
class accessor<T, D, SoA, typename enable_if<is_vec<T>::value>::type> {
  using TT = typename type_info<T, SoA>::mapped_type;
  TT* ptr;
  sarray<size_t, D> n;
  ..
public:
  accessor(TT* ptr, sarray<size_t, D>& n) : ptr(ptr), n(n) { ; }
  ..
  inline accessor<T, D-1, SoA> operator[](size_t idx) {
    size_t offset = idx * T::dim * n.reduce_mul<D-1>();
    return accessor<T, D-1, SoA>(&ptr[offset], n);
  }
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The accessor type (SoA data layout)

```
template <typename T>
class accessor<T, 1, SoA, typename enable_if<is_vec<T>::value>::type> {
  using TT = typename type_info<T, SoA>::mapped_type;
  TT* ptr;
  sarray<size_t, 1> n;
  ..
public:
  accessor(TT* ptr, sarray<size_t, 1>& n) : ptr(ptr), n(n) { ; }
  ..
  inline vec_proxy<TT, T::dim> operator[](size_t idx) {
    return vec_proxy<TT, T::dim>(&ptr[idx], n[0]);
  }
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU

The vector proxy

```
template <typename T, size_t D>
class vec_proxy;
```

https://github.com/flwende/data_types/tree/master/include/vec

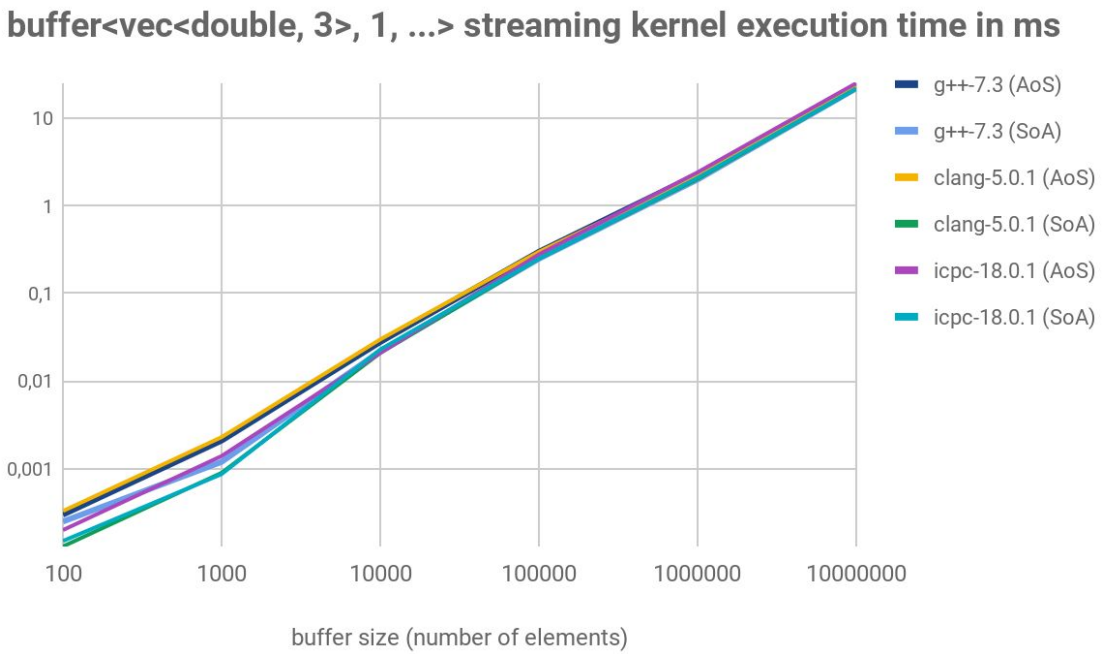
Implementation: buffer data type for CPU

The vector proxy

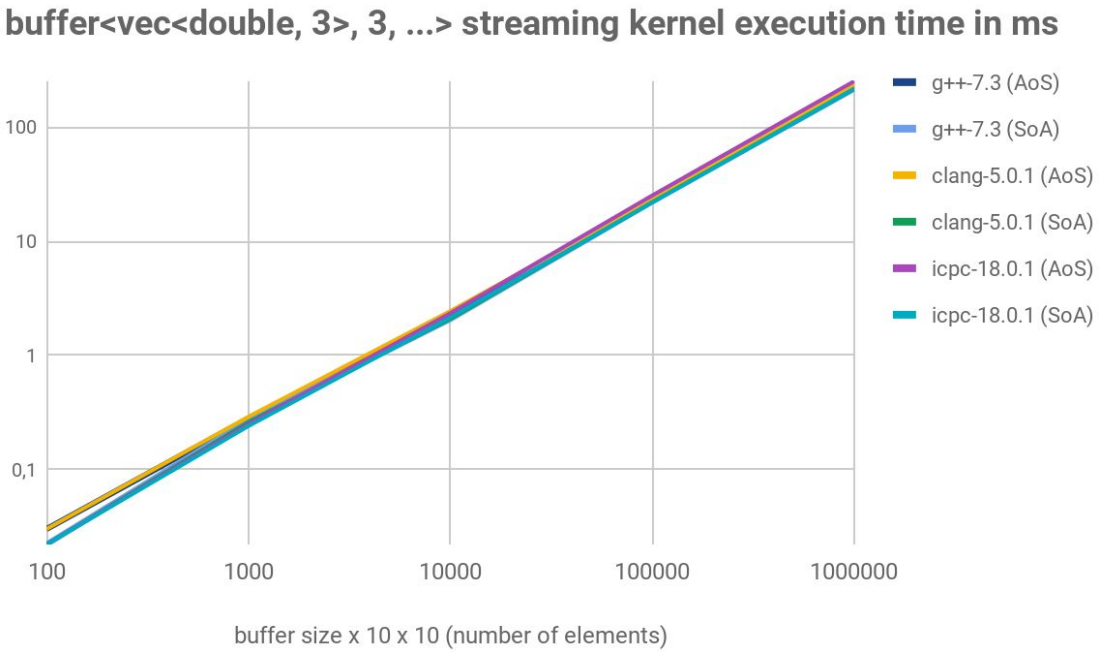
```
template <typename T>
class vec_proxy<T, 3> {
  T& x;
  T& y;
  T& z;
  ..
  vec_proxy(T* ptr, size_t n) : x(ptr[0]), y(ptr[n]), z(ptr[2 * n]) { ; }
  ..
};
```

https://github.com/flwende/data_types/tree/master/include/vec

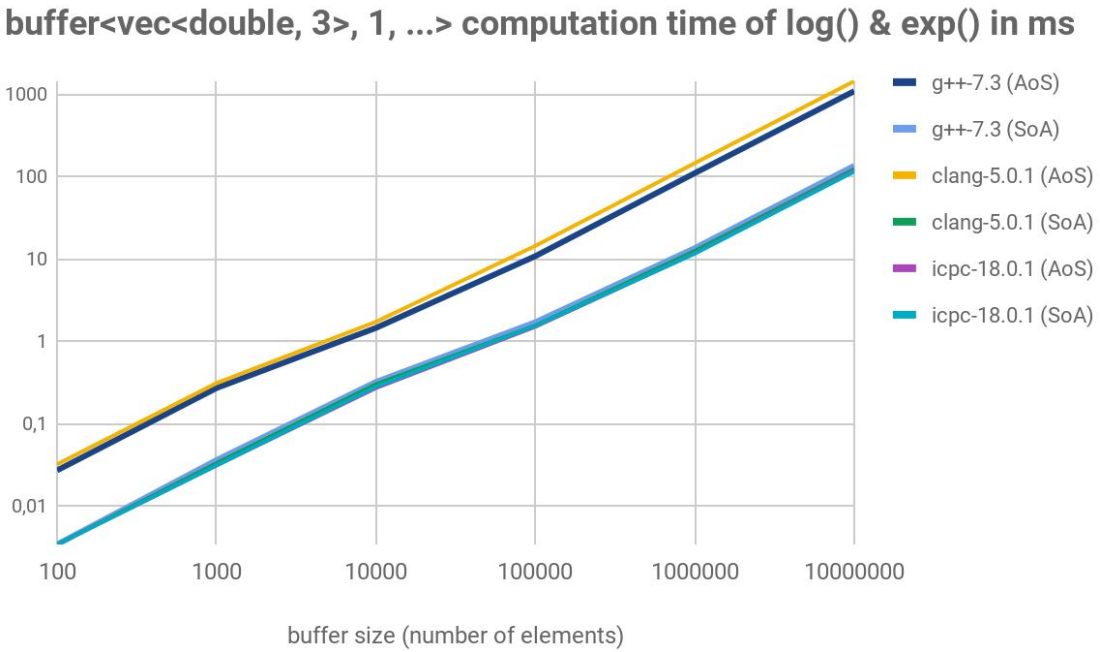
Performance: buffer data type for CPU



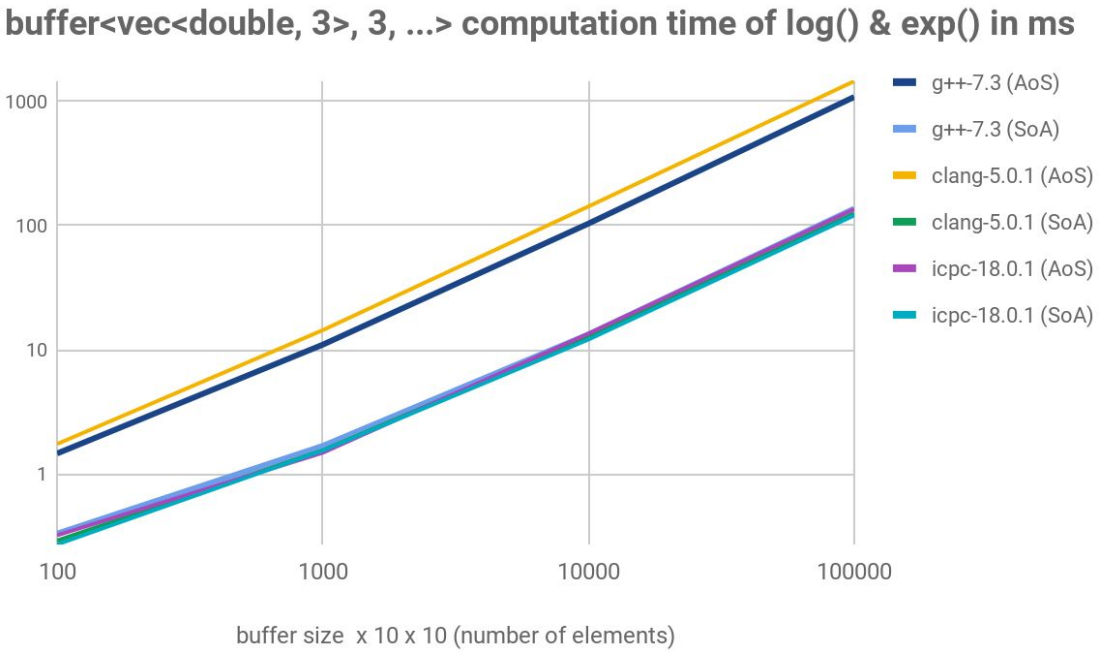
Performance: buffer data type for CPU



Performance: buffer data type for CPU



Performance: buffer data type for CPU



Implementation: buffer data type for CPU+GPU

```
template <typename T, size_t D, buffer_type Buffer_type=host,...>
class buffer { /* sketch: see previous slides */ };
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU+GPU

```
template <typename T, size_t D>
class buffer<T, D, device, AoS> {
  cl::sycl::buffer<T, D>* m_data;
  ..
public:
  buffer(const sarray<size_t, D>& size) : size(size) {
    cl::sycl::range<D> n = “size”;
    m_data = new cl::sycl::buffer<T, D>(n);
  }
  ..
  inline auto read(cl::sycl::handler& h) {
    return m_data->template get_access<cl::sycl::access::mode::read,
                                       cl::sycl::access::target::global_buffer>(h);
  };
  ..
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU+GPU

```
template <typename T, size_t D>
class buffer<T, D, device, AoS> {
  cl::sycl::buffer<T, D>* m_data;
  ..
public:
  ..
  inline void memcpy_h2d(T* ptr, sarray<size_t, D>& n,..) {
    accessor<T, D, AoS> src(ptr, n);
    auto a_data = m_data->template get_access<.., cl::sycl::access::target::host_buffer>();
    accessor<T, D, AoS> dst(a_data.get_pointer(), size);
    memcpy(dst, src, n);
  };
  ..
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU+GPU

```
template <typename T, size_t D, data_layout Layout_host, data_layout Layout_device>
class buffer<T, D, host_device,..> : public buffer<T, D, host, Layout_host>,
                                     buffer<T, D, device, Layout_device> {
  ..
  using host_buffer = buffer<T, D, host, Layout_host>;
  using device_buffer = buffer<T, D, device, Layout_device>;
  ..
  inline void memcpy_h2d() {
    device_buffer::memcpy_h2d(host_buffer::data, host_buffer::size);
  }
  ..
};
```

https://github.com/flwende/data_types/tree/master/include/buffer

Implementation: buffer data type for CPU+GPU

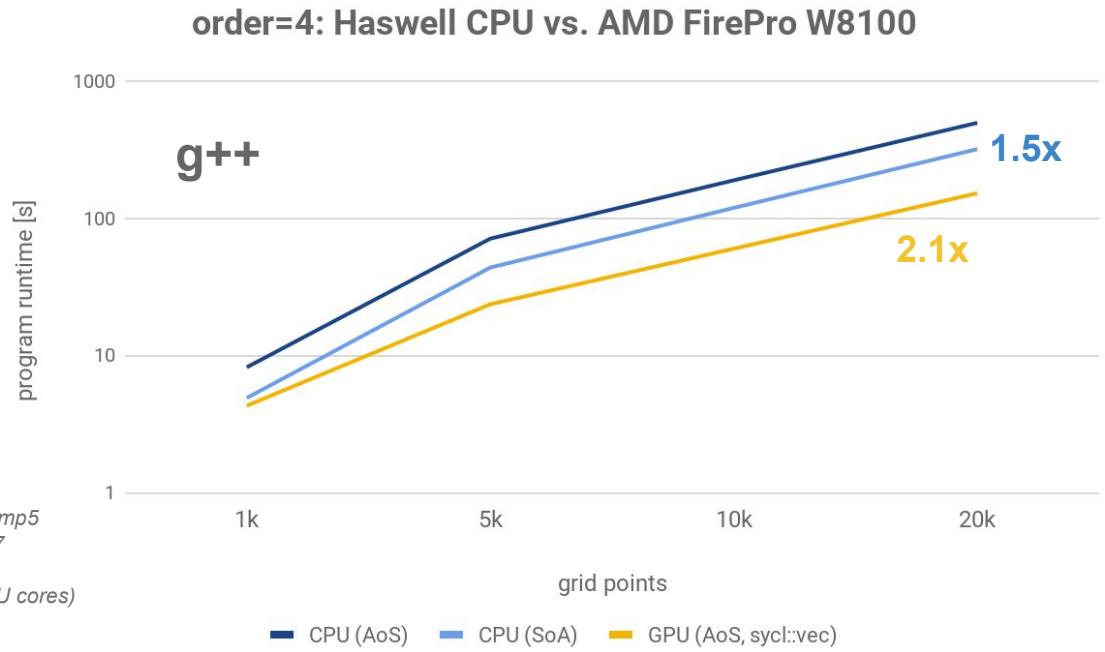
```
template <typename T, size_t D, data_layout Layout_host, data_layout Layout_device>
class buffer<T, D, host_device,..> : public buffer<T, D, host, Layout_host>,
                                     buffer<T, D, device, Layout_device> {
  ..
  using host_buffer = buffer<T, D, host, Layout_host>;
  using device_buffer = buffer<T, D, device, Layout_device>;
  ..
  inline void memcpy_h2d() {
    device_buffer::memcpy_h2d(host_buffer::data, host_buffer::size);
  }
  ..
};
```

implicit data layout transformation!

https://github.com/flwende/data_types/tree/master/include/buffer

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)

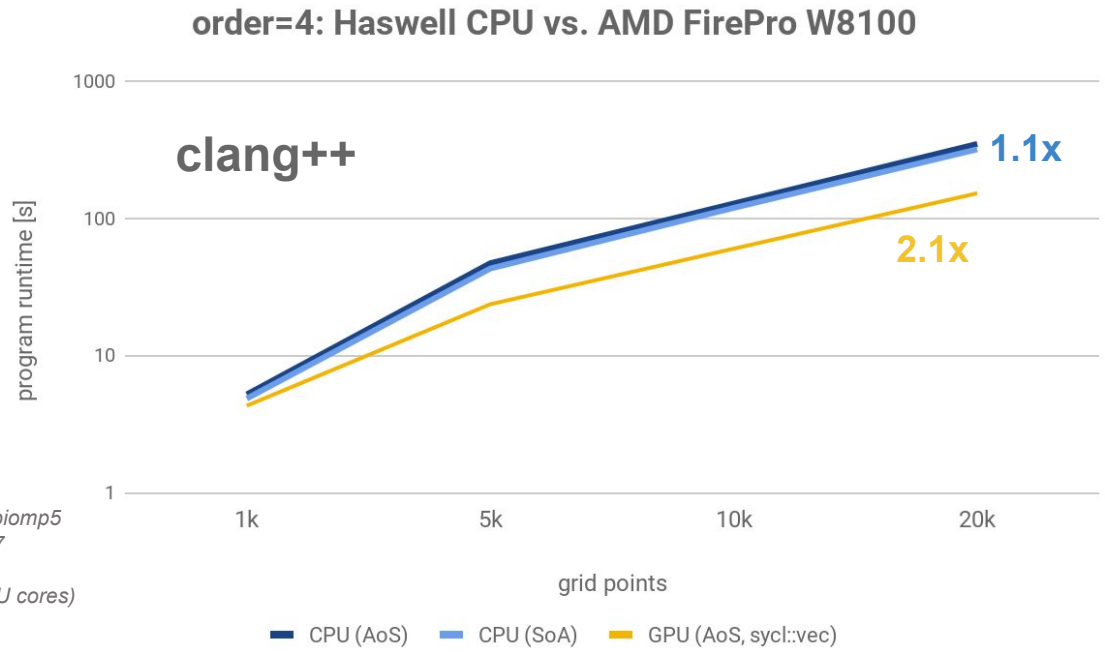


Host: g++-7.3 + glibc-2.25 + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)

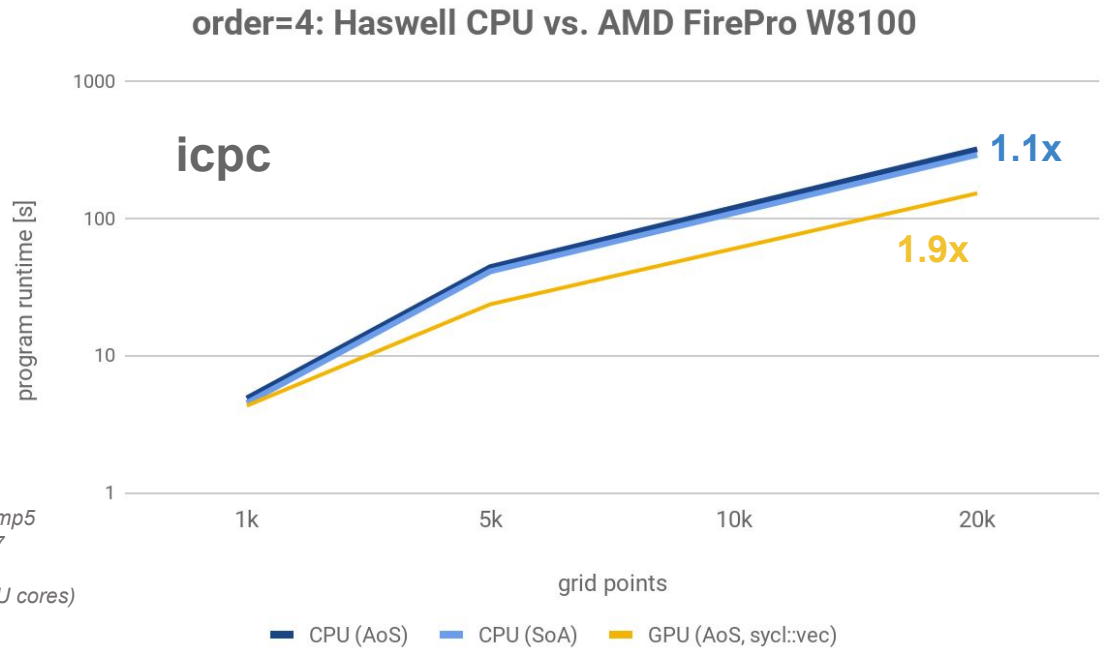


Host: clang++-5.0.1 + SVML + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)

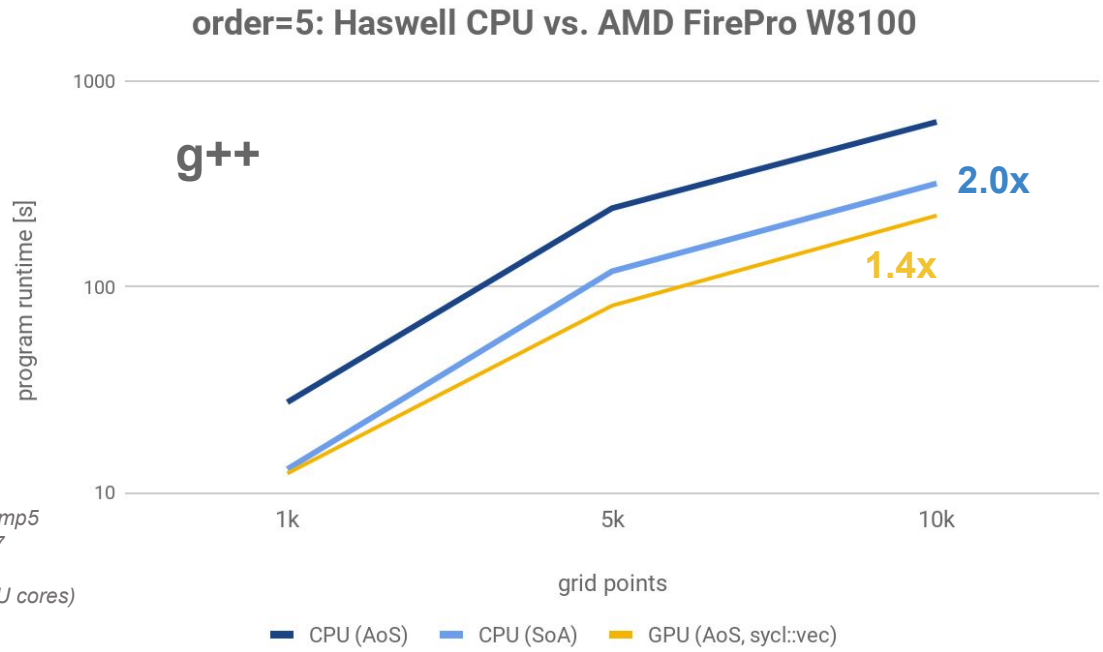


Host: icpc-18.0.1 + SVML + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)

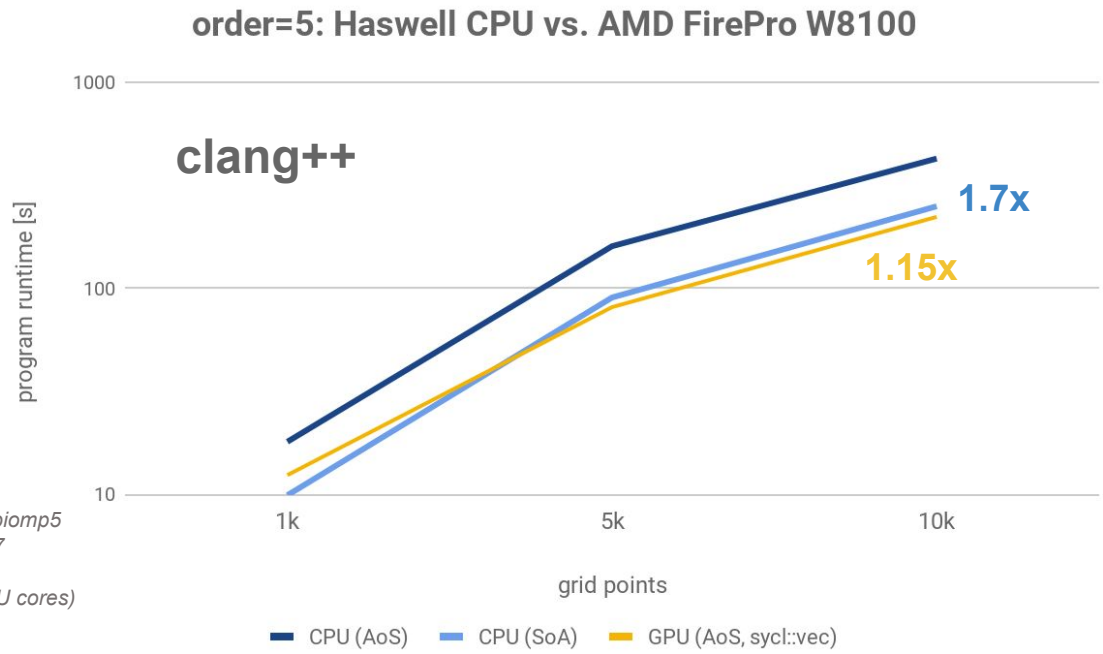


Host: g++-7.3 + glibc-2.25 + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)

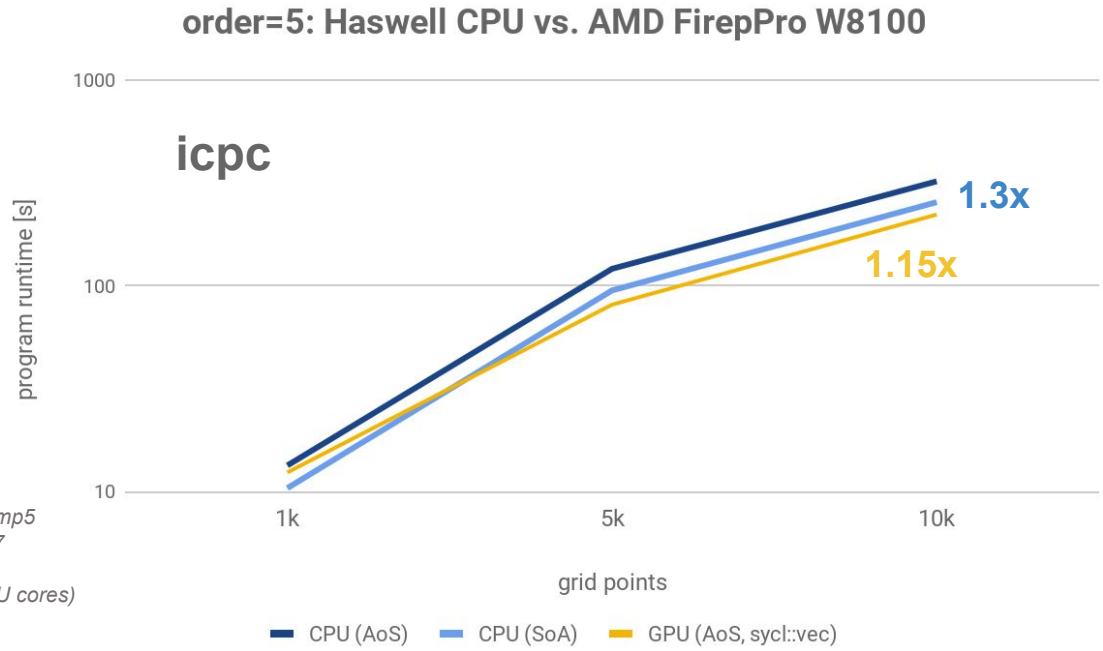


Host: clang++-5.0.1 + SVML + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Performance: buffer data type for CPU+GPU

Electrical engineering: solving for Maxwell's equations (Discontinuous Galerkin)



Host: icpc-18.0.1 + SVML + libiomp5
GPU: Codeplay ComputeCpp 0.7

2x Intel Xeon E5-2630v3 (16 CPU cores)
AMD FirePro W8100

Next steps

- SoA data layout on GPU
  - currently some issues with references to the different address spaces
  - multi-dimensional fields
- SoAoS data layout
- integration of network functionality
  - distributed buffers

EOP

Funding: **HighPerMeshes** (http://highpermeshes.info)
(BMBF grant 01IH16005)