A Discussion of Selected Vienna-Libraries for Computational Science

Karl Rupp
MCS Division
Argonne National Laboratory
Bld. 240
Argonne IL, 60439, USA
rupp@mcs.anl.gov

Florian Rudolf
Institute for Microelectronics
Technische Universität Wien
Gußhausstraße 27-29/E360
A-1040 Wien, Austria
rudolf@iue.tuwien.ac.at

Josef Weinbub
Institute for Microelectronics
Technische Universität Wien
Gußhausstraße 27-29/E360
A-1040 Wien, Austria
weinbub@iue.tuwien.ac.at

ABSTRACT

We address the low popularity of C++ in computational science by introducing a set of orthogonal libraries: The CUDA-, OpenCL-, and OpenMP-enabled linear algebra library ViennaCL, the mesh datastructure library ViennaGrid, a data storage facility named ViennaData, and the symbolic math kernel ViennaMath. Finally, we discuss how these orthogonal components interact within the finite element package ViennaFEM. The main focus of the discussion is on the various programming techniques applied and on how C++ can be improved to better fulfill the demands of computational science.

1. INTRODUCTION

While C++ is evolving and the new features of the C++11 standard make their way into recent compilers, large parts of the computational science community in general and the high-performance-computing community in particular still hesitate to move from C or Fortran to C++. Among the many different reasons [3], the broad range of scientific software providing C or Fortran interfaces is certainly one with the highest weight. Also, a lower level of abstraction is often preferred over the higher complexity of C++ and its associated pitfalls when used in a naive way, for example the creation of expensive temporaries when implementing operator overloads carelessly. Furthermore, current parallelization and vectorization frameworks such as Cilk [9] or OpenMP [25] prefer low-level code with index-based loops rather than high-level code with iterator-based loops [14].

On the other hand, scripting languages such as Python, MATLAB, or Julia are preferred for prototyping purposes. Again, a far larger amount of libraries and tools such as IPython [16] or SciPy [27] is available in such frameworks, while the C++ ecosystem including the Boost libraries [5] does not provide a similar set of functionality even though it could in many aspects combine the best of both worlds. We present and discuss selected free open-source libraries (termed *Vienna-libraries* in the following), which extend the set of available C++ libraries for scientific software engineers. They are a subset of orthogonal tools primarily developed at the Vienna University of Technology and either provide orthogonal functionality or are by themselves composed of orthogonal components. This is in contrast to many application-oriented libraries or frameworks in computational science, where a-priori independent functionality has grown to an application-specific – yet inseparable – alloy over time. Consequently, special attention will be attributed to the way orthogonality is preserved and which C++ techniques are used to achieve this. In the order of discussion, the dependency-free, header-only and multi-platform libraries presented in this work are the following:

- ViennaCL [33]. A linear algebra library using CUDA [24], OpenCL [17], or OpenMP while providing an application programming interface (API) compatible with Boost.uBLAS.
- ViennaGrid [36]. A library providing highly versatile datastructures for the management of meshes in arbitrary geometrical and topological dimension. It can be seen as enriching Boost.Geometry by topology.
- ViennaData [34]. An abstract library for storing data for objects in a non-intrusive manner. The motivations are similar to that of Boost.PropertyMap, yet it satisfies additional requirements of computational science.
- ViennaMath [37]. A symbolic math kernel with a homogeneous interface for compile time and run time manipulations and evaluations. One way to think of it is to specialize Boost.Proto for computer algebra and to add a run time layer to circumvent certain disadvantages of evaluations at compile time.
- ViennaFEM [35]. A library combining the functionality offered by ViennaCL, ViennaGrid, ViennaData, and ViennaMath in order to offer a high-level API for the finite element method, which is very close to the mathematical formulation. In simple terms, ViennaFEM can be seen as one possible extension of Boost.odeint from ordinary differential equations to partial differential equations.

These libraries supplement material presented at C++Now 2012, where implementations aspects of the task execution framework ViennaX [38] have already been discussed [39].

2. VIENNACL

A good linear algebra library is indispensable for the computational scientist, because its performance is in many cases the main factor for the total execution time. This is reflected in several ways: For a single workstation, vendor-tuned libraries for central processing units (CPUs) and accelerators such as graphics processing units (GPUs) are available [2, 15, 23]. On a large scale, the benchmark used for the Top500 list of supercomputers [29] uses a dense matrix operation as a single criterion for assessing computational performance. Despite being a questionable metric, it is still the main metric in high performance computing [18]. For these reasons, one may assume Boost.uBLAS to be the ideal solution in a C++ environment.

2.1 Deficiencies of Boost.uBLAS

While Boost.uBLAS has certainly been a good step forward when it was included into Boost around the year 2000, it has not received the necessary attention to keep track of recent developments in computing hardware. Recent and not-so-recent developments such as multi-core CPUs, vector extensions such as MMX, SSE, or AVX, as well as GPU computing are not natively supported. Even though Boost.uBLAS does not claim to be a high-performance library, it should not ignore this major user requirement.

The Boost Numeric Bindings project [6] is an attempt to mitigate some of its deficiencies by making external algorithms available for Boost.uBLAS objects. Still, concerns have been expressed by several users about the lack of intrinsic support for novel computing hardware as well as advanced algorithms [7]. Also, colleagues expressed concerns in personal communication about Boost.uBLAS being too tightly integrated into the whole collection of Boost libraries, impeding the convenient and quick migration of their otherwise lightweight codes between computing clusters.

2.2 Interface Similarity

The approach taken by ViennaCL is to provide a user API as close to Boost.uBLAS as possible, yet to make use of the computing power of accelerators and multiple cores. While all computations in ViennaCL were formerly based on OpenCL, CUDA and OpenMP computing backends were introduced with ViennaCL 1.4.0. As a short example, the following code snippet depicts simple vector operations:

```
1 using namespace boost::numeric::ublas;
2
3 std::size_t size = 1000;
4 vector<T>   x    =   zero_vector<T>(size);
5 vector<T>   y    = scalar_vector<T>(size, 42);
6
7 y = x + y;                // vector addition
8 y = element_prod(x, y); // elementwise product
```

where `T` is usually of type `double` when used by a computational scientist. Clearly, such a high-level representation is desirable irrespective of the underlying computing hardware. The same code is valid with ViennaCL after replacing line 1 by

```
1 using namespace viennacl;
2 using namespace viennacl::linalg;
```

where the first line makes the primitive objects (such as `vector`) and the second line all the linear algebra functions available. Thus, ViennaCL provides all the basic types from Boost.uBLAS such as `vector`, `matrix`, or `compressed_matrix`. By appropriate compile-time switches one enables the different computing backends available on the machine.

The use of accelerators has a detrimental effect to the performance of single element access. For example, the initialization code

```
1 std::fill(x.begin(), x.end(), rand_functor());
```

for filling a vector `x` with random numbers will perform well if `x` is from Boost.uBLAS. However, doing the same with ViennaCL, the code will be orders of magnitude slower when using CUDA or OpenCL, because each element access triggers a write operation via the PCI-Express interface. Even if OpenCL for the CPU is used, the management overhead of the OpenCL backend gives poor performance. As a remedy, ViennaCL provides a `copy()` routine, which is interface-compatible to `std::copy()`. It collects the data and then uses a single PCI-Express transfer, reducing overhead considerably. Moreover, it allows to manipulate only parts of a vector rather than the vector as a whole:

```
1 //batched copy from Boost.uBLAS to ViennaCL:
2 viennacl::copy(ublas_x.begin(),
3                 ublas_x.end(),
4                 viennacl_x.begin();
5
6 //batched copy from ViennaCL to Boost.uBLAS:
7 viennacl::copy(viennacl_x.begin(),
8                 viennacl_x.begin() + 100,
9                 ublas_x.begin();
```

Here, the contents of a Boost.uBLAS vector `ublas_x` is first transferred to a ViennaCL vector `viennacl_x`, which may reside on a GPU. Then, the first 100 entries are copied back. In this way, the latency of PCI-Express is avoided to the extent possible and data initialization is comparable in speed to Boost.uBLAS. However, this example shows one important aspect: Iterators cannot be applied to accelerators directly because of latency problems.

Elementary operations on matrices are dealt with in the same manner as for vectors. The interface for triangular solvers are also interface compatible:

```
1 // upper triangular solver for Ax = b:
2 vector<T> x = solve(A, b, upper_tag());
```

and similarly for the lower triangular solver as well as those with unit diagonal. However, ViennaCL not only provides dense triangular solvers, but also iterative solvers such as a conjugate gradient or a generalized minimum residual solver [26]. This additional functionality is available via an extension of the existing generic interface for the triangular solvers based on tag dispatching by providing a tag `cg_tag()` for the conjugate gradient solver and correspondingly for the other solvers:

```
1 // conjugate gradient solver for Ax = b:
2 vector<T> x = solve(A, b, cg_tag());
```

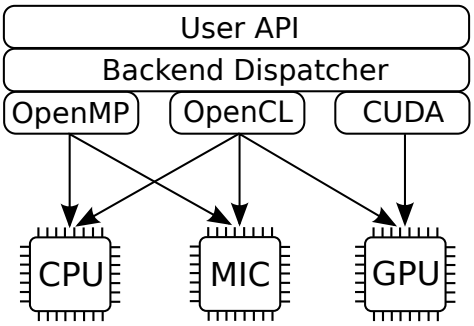

Figure 1: Architecture of ViennaCL. The user-API is compatible with Boost.uBLAS, while the dispatcher translates the respective computing kernels into calls to the respective computing backend.

where `A` is a sparse matrix and `b` is a dense vector. Thanks to the interface compatibility, the solver implementation accepts both Boost.uBLAS and ViennaCL objects and thus allows for code reuse beyond different computing architectures. Similarly, the generic interface accepts objects from the Eigen library [11] and MTL 4 [21]. Generic wrappers for directly passing containers from the C++ standard template library as well as VexCL [32] types are in preparation.

2.3 Selected Internals

Whenever an elementary operation such as `y = x + y` is encountered, both Boost.uBLAS and ViennaCL use the expression template technique [30, 31] to avoid spurious temporaries. While Boost.uBLAS can handle an arbitrary number of operands without introducing temporaries, ViennaCL needs to map the expression to a predefined set of computing kernels. Currently, up to two vector operands, possibly scaled by scalars, can be handled on the right hand side without introducing a temporary. Albeit this appears to be a rather small number, experience has shown that this handles most cases in linear algebra.

The operations defined in the computing backends are typically not called directly, but rather an intermediate dispatcher is called, cf. Fig. 1. This dispatcher inspects the memory handles holding the data for each of the objects such as the vectors `x` and `y`. Depending on the location of the data (main memory, CUDA memory, or OpenCL memory), the call is translated into calls of the respective computing backend. Currently there is mostly a one-to-one mapping between the functionality offered by the dispatcher backend and the operations defined for each backend. For the future we expect that the dispatcher layer offers a much larger set of operations than defined in the individual computing backends and translates the required operation into the respective calls to the kernels defined in each computing backend.

The generic implementation of algorithms in ViennaCL also requires another level of indirection. While functions in Boost.uBLAS such as `prod()` or `inner_prod()` only return the respective expression templates, ViennaCL overloads these functions for the various types in other libraries as well. Initially, interfaces based on the `enable_if<>` technique using SFINAE [30] have been used, e.g.

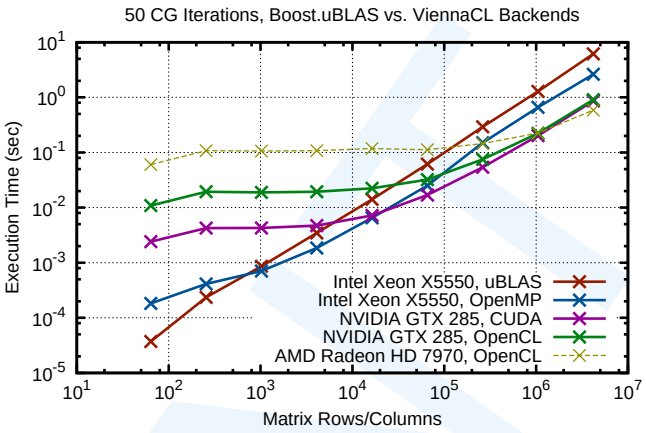

Figure 2: Performance comparison of Boost.uBLAS for 50 iterations of a conjugate gradient solver for a system of equations derived from the two-dimensional discretization of the Poisson equation on the unit square using finite differences.

```
template< typename MatrixT, typename VectorT>
typename enable_if< is_eigen<MatrixT>,
                    VectorT>::type
prod(MatrixT const& A, VectorT const& x)
{
  return A * x;
}
```

to obtain a unified interface for the matrix-vector product `A * x` in Eigen and `prod(A, x)` from Boost.uBLAS. Due to insufficient compiler support, a direct, albeit repetitive, function overloading is used. The `decltype` operator in C++11 helps when it comes to the automatic deduction of the return type of these generic overloads (note that the above snippet can be improved by returning the appropriate expression template), yet its adaption in ViennaCL is hindered by the small availability compilers for C++11.

2.4 Performance

We consider the linear system of equations obtained from the discretization of the Poisson equation on the unit square using finite differences. A comparison of execution times for a fixed number of 50 iterations of a conjugate gradient solver without preconditioner is given in Fig. 2. The optimized sparse matrix-vector multiplication function `axpy_prod()` has been used for Boost.uBLAS, even though the less optimized function `prod()` would presumably be used by a less experienced user.

At large problem sizes, the GPU backends of ViennaCL are by about an order of magnitude faster than Boost.uBLAS. For system sizes below 10^4 unknowns, the PCI-Express latency as well as OpenCL and CUDA overheads of the GPU backends are clearly visible, leading to constant execution times. Still, in such case the OpenMP backend of ViennaCL outperforms Boost.uBLAS for problem sizes down to 100×100 by about a factor of two. Only for very small operations, the rigorous use of expression templates gives better performance of Boost.uBLAS over ViennaCL.

2.5 Conclusions

ViennaCL readily shows that many of the deficiencies of Boost.uBLAS in terms of performance can be overcome without changing the API significantly. Thus, the same high level of abstraction is possible even if accelerators are used. However, current and future heterogeneous computing environments do not allow for a neat abstraction of linear algebra operations using iterators, but instead require the use of a reasonable set of well-tuned computing kernels operating on memory buffers directly.

3. VIENNAGRID

In order to set up the discrete system of equations as in Sec. 2.4, a discrete representation of the computational domain is required and central to areas such as fluid dynamics or mechanical engineering. This is achieved by representing the computational domain by a set of points, which in addition to their geometric location have a topological structure imposed on them. A set of vertices makes up a cell, typically given by simple shapes such as tetrahedra or hexahedra in three dimensions. The union of all the cells, which are allowed to touch, but not to overlap, represents the computational domain. For a more stringent definition of these topological concepts we refer to the literature [19].

Boost.Geometry is able to deal well with the geometric part of meshes, i.e. points, by providing a wrapper for containers, various coordinate systems, and algorithms defined therein. Only a few C++ libraries dealing with the topological aspect are available, of which we mention CGAL [8] and DUNE [10]. CGAL's focus, however, is on geometrical algorithms and uses topological information only as a means to serve the needs of its many geometric algorithms. DUNE, on the other hand, explicitly aims at numerical application and allows for dimension-independent programming thanks to a heavy use of templates. However, a design decision of DUNE is that the internal datastructure is not flexible, i.e. the user has no control over which topological information is stored internally, and thus inefficient for certain problems.

Our approach to mesh handling is ViennaGrid, which provides a very flexible, highly configurable and dimension-independent datastructure. It also supports advanced features such as uniform and non-uniform mesh refinement and the computation of Voronoi quantities. Cell types are restricted to be homogeneous over the whole mesh in the current release 1.0.1. Even though this is usually the case in practice, support for hybrid meshes is currently under development, but not further discussed in this work for the sake of clarity.

3.1 Main Entities

A *domain* is the main container for the mesh and holds the global representation of all elements. Vertices are zero-dimensional topological elements and refer to a geometric *point* inside the computational domain. An *edge* is a one-dimensional topological element and connects two vertices. Conversely, a *cell* is a topological element of maximum dimension, say N. For example, a triangle in a triangular hull mesh in a three-dimensional Euclidean space is of topological dimension two. A *facet* is a topological element of dimension $N-1$, i.e. a triangle in a tetrahedral mesh, or

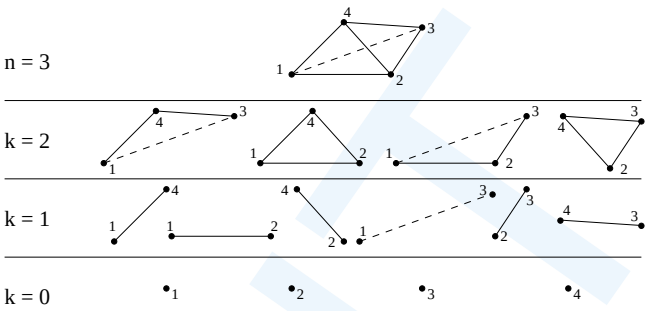

Figure 3: Illustration of the boundary k-cells of a tetrahedron (3-simplex).

a quadrilateral in a hexahedral mesh. Because of this parameterization, it is convenient to use the notion of *n-cells*, where n is the topological dimension. Consequently, we refer to vertices are 0-cells, to edges as 1-cells, and so on.

The domain is implemented via a class `domain<Config>`, where the template argument `Config` specifies the properties of the domain as well as its datastructure. Only three type definitions are required:

- `numeric_type`: Numeric type used for the coordinates of the geometric space, usually `double`.
- `coordinate_system_tag`: One out of several tags for different coordinate systems available, e.g. `cartesian_cs<2>` for a two-dimensional Cartesian coordinate system.
- `cell_tag`: A tag for the cell type within the mesh, e.g. `tetrahedron_tag` for tetrahedral meshes.

Note that due to the use of homogeneous meshes in ViennaGrid 1.0.1, the n-cell type for each topological dimension n can be obtained directly by the specification of the cell type only. The individual n-cells are instances of type `element_t<Config, Tag>`, where `Config` is again the domain configuration type, and `Tag` is a suitable element tag. Two different families of element tags are supported: Simplex elements of dimension k are obtained from `simplex_tag<k>`, and hypercubes of dimension k from `hypercube_tag<k>`. For example, `simplex_tag<2>` refers to triangles and `simplex_tag<3>` refers to tetrahedra, while `hypercube_tag<2>` denotes quadrilaterals and `hypercube_tag<3>` hexahedra. Because of this parameterization of the element tag, template metaprogramming can be applied to compose the required datastructures at compile time.

3.2 Configurable Datastructure

An n-cell consists of k-cells with $0 \leq k < n$ at its boundary, cf. Fig. 3, to which we refer to as *boundary k-cells* in the following. An algorithm might need to operate on each k-cell in the mesh, but also on each boundary k-cells of an n-cell. Let us first consider a datastructure where all boundary k-cells in a mesh are available both locally for each cell and globally within the domain. In such case, each instance of an n-cell carries references or pointers to each of its k-cells. For example, a tetrahedron holds references to its

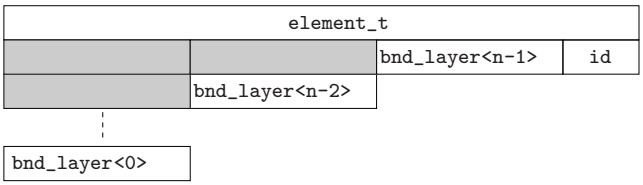

Figure 4: Illustration of recursive inheritance for the n-cell class `element_t`. An optional identification variable (id) can be attached to `element_t`.

	Amount	Bytes/Object	Total Memory
Vertices	4913	24	115 KB
Edges	31024	16	485 KB
Facets	50688	48	2376 KB
Cells	24576	112	2688 KB
Total			**5664 KB**

Table 1: Minimum memory consumption for a tetrahedral mesh where all n-cells are stored explicitly and boundary information on each n-cell is stored. Vertices use `double`-precision coordinates and elements are linked with 64-bit-pointers.

four facets, its six edges, and its four vertices, leading to a total number of 14 references to boundary k-cells. These references are obtained by `element_t` inheriting from a helper class `bnd_layer<k>`, which has the sole purpose of holding references and providing access to the boundary k-cells. With an identification class `id` being either empty or holding a numerical identifier, the implementation of `element_t` becomes

```
template <typename ConfigT,
          typename Tag>
class element_t :
 public bnd_layer<ConfigT, Tag, Tag::dim-1>,
 public id<ConfigT, Tag> { ... }
```

Here, `bnd_layer<Config, Tag, k>` inherits recursively from `bnd_layer<ConfigT, Tag, k-1>` until $k = 0$, cf. Fig. 4. Since the number of boundary k-cells for each k is known at compile time, a static array can be used for the references, allowing the compiler to unroll loops.

If only a minimal topological information is required by a particular algorithm, for example when implementing lowest-order finite element methods, a tetrahedron needs to hold references to its vertices only, thus only 4 instead of 14 references are required. In ViennaGrid, this setting is enabled by providing specializations of `bnd_layer<ConfigT, Tag, k>` depending on `ConfigT`, `Tag`, and `k`, for which a metafunction `bnd_handling` is overloaded suitably. For example, to selectively disable the storage of boundary triangles for tetrahedra, the necessary overload is obtained via the convenience macro

```
VIENNAGRID_DISABLE_BOUNDARY_NCELL(
 MyConfig, viennagrid::tetrahedron_tag, 2)
```

Similarly, the storage of boundary k-cells within the domain can be enabled or disabled, allowing the library user to tune the datastructure to a particular application.

	Amount	Bytes/Object	Total Memory
Vertices	4913	24	115 KB
Edges	0	-	0 KB
Facets	0	-	0 KB
Cells	24576	32	768 KB
Total			**883 KB**

Table 2: Minimum memory consumption for a tetrahedral mesh where only 0-cells and 3-cells are stored explicitly. Vertices use `double`-precision coordinates and elements are linked with 64-bit-pointers.

Tab. 1 and Tab. 2 compare the memory requirements of a tetrahedral domain on a 64-bit system, where references to all boundary k-cells are kept, with a domain where only the vertices of each cell are referenced. The first datastructure is commonly provided statically in higher-order finite element packages, while the latter datastructure is found in lowest-order finite element implementations. The dynamic datastructure in ViennaGrid allows an efficient datastructure for both scenarios, leading to a difference in memory footprint of more than a factor of six.

3.3 Iterations

So far only the internal configurable datastructure of ViennaGrid has been discussed. Let us now consider the iteration over k-cells, for which the concept of ranges as in Boost.Range is adapted. Thus, iterators are instantiated from a dedicated and light-weight range object, which is derived from the domain or the respective n-cell holding the boundary k-cells. A sample code snippet for iteration over all edges (1-cells) in the domain is as follows:

```
typedef result_of::ncell_range<DomainType,
                               1>::type  EdgeRange;
typedef result_of::iterator<EdgeRange>::type
                                    EdgeIterator;

EdgeRange edges = ncells<1>(domain);
for (EdgeIterator eit  = edges.begin();
                  eit != edges.end();
                ++eit)
{ /* do something with *eit */ }
```

First, the range type is retrieved by passing the domain type (`DomainType`) and the topological dimension (1 for edges) to the metafunction `ncell_range`. Then, the iterator type is obtained and the range instantiated using the function `ncells` by passing the topological dimension as template parameter. Finally, iteration over all edges in the domain is carried out and each edge can be obtained by dereferencing the iterator as usual.

Iterating over boundary k-cells of an n-cell is provided by the same generic interface, where the n-cell is used instead of the domain. Since the resulting for-loops are short (for example, a tetrahedron is known to have six edges only), index-based iteration is supported as well in order to assist the compiler in detecting unrollable loops:

```
EdgeOnCellRange edges = ncells<1>(cell);
for (std::size_t i=0; i<edges.size(); ++i)
{ /* do something with edges[i] */ }
```

Here, `cell` is the n-cell over whose edges to iterate, and the type `EdgeOnCellRange` is obtained using the metafunction `ncell_range` again. `edges.size()` returns a value known at compile time.

Coboundary-iteration, i.e. iteration over n-cells sharing a certain k-cell, $k < n$, is achieved using the same interface. An example for coboundary-iteration is the traversal of all tetrahedra sharing a common edge. However, the range retrieval function `ncells()` needs to be supplied with the enclosing cell complex as second parameter in such case. This can either be the full domain, or only a view of the domain, called *segment* in ViennaGrid. Also, boundary and co-boundary iterations can be nested arbitrarily or combined with STL algorithms such as `for_each()`, leading to dimension-independent and highly expressive code.

3.4 Conclusions

ViennaGrid applies generic programming and metaprogramming techniques for handling meshes with a datastructure adjustable at compile time. It naturally extends the ideas of Boost.Geometry from geometry to topology and decouples the two. Its generic iterator-interface is designed with high usability in mind and leads to very expressive code.

4. VIENNADATA

Many algorithms acting on the n-cells of ViennaGrid need to store and access data on some or all of the elements. For example, an application may store a vector field on each vertex or on each cell of a mesh. The typical approach using object-oriented programming is to add data members to the respective classes such as `element_t` in ViennaGrid. However, such an intrusive approach, albeit often encountered in practice, would restrict the reusability of a generic mesh datastructure to a particular use-case. The approach taken by DUNE is to defer all data handling to the user and only provide numerical identifiers for each object. From the usability point of view, this is not a satisfactory approach either, because in this case the challenge of storing data on the mesh is only deferred to the library user. The rationale behind ViennaData is to provide an orthogonal, generic storage layer for ViennaGrid, DUNE and all other scenarios where many object of the same type are encountered.

4.1 Attaching Data to Objects

Without further knowledge about the internals of an arbitrary object `obj` of arbitrary type `ObjectType`, one way to store data (of type `DataType`) for `obj` is to use a map such as

```
1 std::map<ObjectType *, DataType> data;
```

Since different data of the same type might be stored for `obj`, one may use different keys (of type `KeyType`) in order to organize the data. Because the number of different keys used within a large application might be unknown at compile time and difficult to determine in advance at run time, one could use

```
1 std::map<ObjectType *,
2          std::map<KeyType, DataType> > data;
```

The complexity of inserts as well as accesses is logarithmic in the number of keys and the number of objects, which can already be acceptable for a number of applications. What remains is to provide an interface which releases the user from the burden of instantiating and passing the data containers for all combinations of `ObjectType`, `KeyType`, and `DataType` around. By using Meyer's Singletons [20] for the data containers and a number of proxy-classes for wrapping the access accordingly, one ultimately obtains the interface used by ViennaData:

```
1 viennadata::access<KeyType,DataType>(key)(obj);
```

where `key` is an object of type `KeyType` used for accessing the data for `obj`. The two template arguments are intentionally in the same order as in `std::map<KeyType, DataType>` and mandatory. Thus, in order to store and access data of type `long` using a key of type `std::string`, one can simply write (namespace `viennadata` omitted)

```
1 access<std::string, long>("my_key")(obj) = 42;
2 long data
3    = access<std::string, long>("my_key")(obj);
```

and by default the data is accessed by internally using the datastructure of two nested maps as shown above.

4.2 A Limitation of C++

A call to `access` for key type `KeyType`, data type `DataType`, and object type `ObjectType` internally forwards the call to a class `container<KeyType, DataType, ObjectType>`, for which functions such as `reserve` and `erase` are defined. For example, to erase data of type `double` stored for an object `obj` using a `key` of type `long`, one writes

```
1 erase<KeyType, DataType>(key)(obj);
```

However, if an object is destroyed, data for the object stored in any instance of `container<>` needs to be erased. If `key` is omitted, all data of type `DataType` stored for `obj` using any keys of type `KeyType` is erased. Thus, a library user would have to keep track of all key types and data types used for each object type and call `erase` for each triple before the object is destroyed. Since this is highly error prone, ViennaData internally tracks all instances of `container<>` using type erasure [1]. On first access to a new instance of `container<>`, a reference is added to a type tracker at run time. A special type `viennadata::all` indicates that an operation should act on all matching instances of `container<>`, so that a user can simply call

```
1 erase<viennadata::all,
2       viennadata::all>()(obj);
```

Internally, the container of type-erased references identifying the various instances of `container<>` is traversed and `erase(obj)` is called on each matching container, erasing the data stored for `obj`. If a large number of different key types, data types and object types is involved, this workaround can become very costly. Note that if the call to `erase` is placed within the destructor of the objects for which data is stored,

the data is automatically deleted at the end of the object's lifetime.

A more efficient implementation would be possible if C++ provided a traversal mechanism for types. In pseudo-code this would read for an object `obj` of type `ObjectType`:

```
for_each [T, U] in container<T, U, ObjectType>
{ container<T, U, ObjectType>::erase(obj); }
```

Such a functionality could be provided without significant additional compiler implementation effort, because a compiler internally needs to keep track of all instantiations of template classes anyway.

4.3 Fast Data Access

Thinking of large meshes with millions of n-cells, logarithmic access complexity in the number of keys and the number of objects implies considerable overheads, particularly when compared to a direct index-based access. In terms of datastructure, it is desirable to use faster datastructures than

```
std::map<ObjectType *,
         std::map<KeyType, DataType> >
```

First, instead of dispatching data of the same type for different key objects at run time, the dispatch can be transferred to compile time based on the key type only:

```
struct FastKeyType {};
VIENNADATA_ENABLE_TYPE_BASED_KEY_DISPATCH(
  FastKeyType);
access<FastKeyType, long>()(obj) = 42;
```

Here, a dedicated key type is defined first. Then, a convenience macro provides the respective template specializations in order to enable type-based dispatch for `FastKeyType`. Finally, data is written using the function `access`. Note that no key object is required and hence the first pair of parentheses can be left empty. With this type-based key dispatch, the internal datastructure is effectively reduced to

```
std::map<ObjectType *, DataType>
```

and thus eliminates the access overhead with respect to the key dispatch at run time.

In order to eliminate the logarithmic access times with respect to the number of objects for which data is attached, one needs to provide a mechanism to extract an index such that one can directly access an array. Suitable identification integers are available in ViennaGrid and DUNE, but this may not be the case in general. To make the identification mechanism for objects of type `ObjectType` known to ViennaData, a template specialization for class `object_identifier` in namespace `viennadata::config` is required, e.g.

```
template <>
struct object_identifier<ObjectType> {
  typedef object_provided_id tag;
  typedef long id_type;
  static id_type get(ObjectType const & obj)
  { return obj.id(); }
};
```

The first `typedef` is used to indicate that the object provides integers for identification. The second `typedef` specifies the integer type, and finally `get()` provides the generic wrapper for extracting the identifier from the object. With the object identification in place, the convenience macro

```
VIENNADATA_ENABLE_DENSE_DATA_STORAGE_FOR_OBJECT
  (ObjectType)
```

then applies the necessary template specializations in order to use an internal datastructure of type

```
std::vector<DataType>
```

when using a type-based key dispatch, and

```
std::vector<std::map<KeyType, DataType> >
```

otherwise. Note that the object identifier and the dense data storage are two distinct concepts: On the one hand one may want to provide an object identification mechanism for all data stored for the object, but on the other hand store data using a certain key type for a few objects only. One example is the storage of data only for vertices on the boundary of a mesh, in which case a `map` is a better choice than a `vector`.

4.4 Benchmarks

Object identifiers and type-based key dispatch allow to reduce the logarithmic access complexity to constant access complexity. In order to quantify the overhead of ViennaData with respect to direct class member access, we compare data access for data attached to a lightweight class `LightWeight` with data attached to classes holding many members:

```
struct LightWeight
{ std::size_t id; };

template <std::size_t s>
struct FatClass
{ std::size_t id; char payload[s]; double d; };
```

Table 3 compares the execution times obtained on a Linux machine equipped with and AMD Athlon II X2 255 running a 64-bit Ubuntu 10.10. While direct member access is by a factor of about two faster for small payload in `FatClass` and 10^3 objects, ViennaData used for `LightWeight` is comparable or even faster than direct member access for 10^6 objects. This is because the larger payload for `FatClass` congests caches, while ViennaData intrinsically groups data of the same kind in memory, allowing for better cache reuse and a more efficient use of memory bandwidth.

	10^3 Objects (us)	10^6 Objects (ms)
`LightWeight`	**4**	**5**
`FatClass<10>`	**1.3**	**4**
`FatClass<100>`	2.1	11
`FatClass<1000>`	2.5	11

Table 3: Comparison of execution time for summing up one `double` value from each object using ViennaData on objects of type `LightWeight` and direct member access to objects of type `FatClass`.

4.5 Conclusions

ViennaData provides a generic and unified interface for attaching data to objects in a non-intrusive manner and provides fine-grained control of the internal storage schemes. This allows for tailoring the data structure to application-specific requirements on the manipulated data in a transparent way. We also highlighted a shortcoming of C++, for which a remedy using type-erasure is provided.

5. VIENNAMATH

Quantity distributions over a mesh are commonly modeled by a discrete space using spline interpolations or approximations. Since splines are piecewise polynomials, we draw our attention to the symbolic handling of polynomials and mathematical expressions within C++.

For evaluations at run time, GiNaC [13] provides the basic functionality of computer algebra systems directly within C++ [4]. On the other hand, expression templates have been shown to allow for certain mathematical manipulations at compile time, e.g. [22], using either Boost.Proto or custom implementations. With ViennaMath we provide both a symbolic math kernel for evaluations using traditional run time polymorphism, as well as compile time evaluations based on metaprogramming techniques such as expression templates using a unified interface. A full discussion of the all features of ViennaMath such as the transformation of mathematical expression in C++ to LaTeX code, however, is beyond the scope of this work.

5.1 Basic Types

ViennaMath provides two families of types, one for the representation of mathematical expressions at run time, and one for the representation at compile time. Compile time representations can be converted to run time representations, while the converse is obviously not possible.

The following snippet shows an exemplary use of the run time system:

```
constant pi = 3.1415;
variable x(0);
variable y(1);
expr p = x * (y + pi);
```

First, a constant `pi` is defined. Then, objects `x` and `y` representing mathematical variables x and y are defined. The constructor argument denotes the (zero-based) index when evaluating an expression for a tuple of numbers, which is termed *vector* in the following. Finally, the polynomial $p(x, y, \ldots) = x(y + \pi)$ is defined and the result stored in an object of type `expr`, which allows to capture all mathematical expression using a common type at run time. To evaluate the polynomial, one can simply write

```
p( make_vector(1, 2) );
```

to obtain the result 5.1415. `make_vector` is a convenience routine for creating a vector in-place and returns a vector of static size. The function name is chosen in similarity to various functions available in e.g. Boost.Fusion. As an alternative, objects of type `std::vector<T>` can be passed to the functor interface, with `T` being the underlying numerical type (`double` by default). ViennaMath also provides vector-valued expressions and overloads for common unary functions such as `expr()` or `sqrt()` for use within its symbolic math engine.

The available types for compile time manipulations are similar to their run time equivalents and prefixed with `ct_`:

```
ct_constant<2> c2;
ct_variable<0> x;
```

The constant 2 is encoded as template argument, where due to restrictions in the C++ type system only integers are allowed. Floating point template arguments would be extremely helpful in this context, yet we understand the technical difficulties in providing such a feature. Similarly, a compile time variable is obtained by providing the constructor argument from the run time case as a template argument for the compile time case. To assign the operation `x + c2` to an object, one either needs to encode the expression explicitly in the type, or use the `auto`-keyword from C++11:

```
auto p = x + c2;
```

Evaluation of the compile time expression is again possible via the parentheses operator just as for run time evaluations:

```
p( c2 );
```

A mechanism for making the more intuitive

```
p( 2 );
```

available for explicit compile time evaluations is, however, not yet available in C++, thus one has to rely on compiler optimizations. Moreover, there is no way to make

```
auto p = x + 2;
```

accessible for subsequent compile time manipulation, because the integer constant enforces a run time type for `p`.

5.2 Expression Manipulation

ViennaMath not only provides expressions formed by using unary and binary operator overloads, but also symbolic manipulations. Besides the evaluation of expressions, the most commonly used functions are:

```
substitute(x, y, f);
expand(c2*(x+y));
simplify(x + 1.0 * y - 0);
diff(x + y, x);
```

In the first line, the variable `x` is substituted for `y` in the expression `f`. The second lines expands the supplied expression to `c2*x + c2*y`. The third line simplifies the provided expression to `x + y`, while the last line differentiates `x+y` with respect to `x`, resulting in the constant 1. If all terms passed to the manipulation functions are compile time evaluable, then they are actually manipulated at compile time and only the result is considered in the final executable.

Figure 5: ViennaFEM is composed of four orthogonal libraries.

ViennaMath also provides support for integrations, which makes it very attractive for the use within finite element or finite volume methods. Evaluation of the integral

$$\int_0^1 x^2 \, \mathrm{d}x$$

can be performed at compile time via

```
integrate( make_interval(c0, c1),
           x * x, x);
```

provided that `c0`, `c1` are compile time representation of 0 and 1 and that `x` is of type `ct_variable<0>`. Integrals can also be nested and evaluated at run time using numerical integration routines by feeding the integral expression to either one of the quadrature rules provided by ViennaMath, or a user-defined quadrature rule.

5.3 Conclusions

ViennaMath provides a symbolic engine for the evaluation and manipulation of mathematical expression both at run time and at compile time. This is achieved by blending well-established expression template techniques with traditional dynamic polymorphism in order to give the user the ability to balance run time efficiency with compilation times. Moreover, with just-in-time compilers becoming more popular, ViennaMath provides an ideal intermediate mathematical layer for code generation.

6. VIENNAFEM

The last library discussed in this work, ViennaFEM, combines functionality offered by the previous libraries to a more application-oriented finite element library. This design is in contrast to other modern C++ libraries for the finite element method such as Feel++ [12] or Sundance [28], where functionality is not separated as explicitly and cleanly as in ViennaFEM.

Due to the mathematical complexity of the underlying mathematical algorithms, we refrain from an in-depth discussion and merely focus on the interaction of the libraries ViennaFEM is based on. Thanks to the combination of ViennaData and ViennaGrid, the implementation of ViennaFEM mostly deals with the manipulation of the weak form of the underlying mathematical problem based on the features provided by ViennaMath. Once the system of equations is assembled, a GPU-accelerated solution is provided by ViennaCL.

6.1 Execution Flow

To see how the various libraries interact within ViennaFEM, a discussion of the numerical solution of the Poisson equation

$$\Delta u = 1 \tag{1}$$

on a computational domain Ω with Dirichlet boundary conditions is given. Using ViennaMath, the problem description transferred to code reads

```
function_symbol u(0, unknown_tag<>());
equation eq = make_equation(laplace(u), -1);
```

`u` represents a symbolic function, while `eq` encodes the Poisson equation (1). Since overloading `operator=` to behave like a mathematical equality leads to collisions with the assignments to objects in C++, a generator function `make_equation` is used instead for setting up the equation.

First, the mesh is imported in ViennaGrid. Depending on the spatial dimension and the finite element method used, certain k-cells can be disabled by the user for memory efficiency.

Next, the location and the value of the boundary conditions are written to the ViennaGrid domain using ViennaData. This is either achieved by reusing data read from file via the mesh file reader, or by iterating over the n-cells on the boundary of the mesh and storing boundary conditions explicitly.

Then, an equation assembler object is instantiated and the equation is passed to the functor interface:

```
pde_assembler fem_assembler;
fem_assembler( make_linear_pde_system(eq, u),
               my_domain,
               my_matrix, my_load_vector);
```

where `my_domain` is the ViennaGrid domain, and `my_matrix` and `my_load_vector` are generic matrix and vector objects offering parenthesis access. The assembler object `fem_assembler` then deals with all the details such as the enumeration of unknowns and the transformation of the strong form to the weak form. In addition to the equation `eq` and the unknown `u`, a configuration object can be supplied as third argument to `make_linear_pde_system`, customizing details such as the finite element spaces. Further arguments to `fem_assembler` even allow for coupled systems of partial differential equations. Note that this is a generic, dimension-independent interface, since the mathematical formulation is only internally mapped to the spatial dimension of the mesh supplied with `my_domain`.

Finally, a solver from ViennaCL solves the resulting system of equations. If desired, the solution is written to files for visualization using the IO functionality of ViennaGrid.

6.2 Conclusions

Thanks to the rigorous composition of ViennaFEM from orthogonal libraries, the core implementation is extremely lightweight and far easier to maintain than other monolithic finite element libraries. In particular, contributions to e.g. ViennaGrid will automatically be available in ViennaFEM, even if the contributor does not know anything about the finite element method. This is particularly important for the high complexity of research code, where the overall mathematical complexity is effectively split into units of smaller complexity.

7. SUMMARY

An overview of selected Vienna-Libraries has been given. Each of the libraries focuses on a particular task, aiming at providing functionality for a broad range of applications. ViennaCL has shown that the API of Boost.uBLAS can be used for accelerator and multi-core architectures. ViennaGrid offers a highly flexible mesh datastructure with a convenient interface, which enables dimension-independent programming and high computational efficiency. ViennaData allows for attaching data to arbitrary objects in a generic, non-intrusive way. We have also discussed its application for ViennaGrid in order to select the best data storage for the particular needs of different mesh algorithm. The symbolic math kernel library ViennaMath provides the ability to manipulate mathematical expressions either at run time or at compile time by means of a unified interface. Finally, ViennaFEM combines the orthogonal functionalities of ViennaCL, ViennaGrid, ViennaData, and ViennaMath to a modern, high-level finite element library.

Most importantly, we showed that the rigorous application of orthogonal software and library design is also well applicable to the domain of computational science, particularly the numerical solution of partial differential equations. The benefit of such a decomposition is exchangeability, a reduction of complexity into smaller, independent units, and thus increased maintainability. This might ultimately lead to a broader acceptance of C++ in this community.

8. ACKNOWLEDGMENTS

This work has been supported by the European Research Council (ERC) through the grant #247056 MOSILSPIN, the Austrian Science Fund (FWF) through the grant P23598, the Graduate School PDETech at the TU Wien, the Department of Energy ASCR SciDAC project FASTMath, Google via the Google Summer of Code 2011 and the Google Summer of Code 2012, as well as NVIDIA and AMD through hardware donations.

9. REFERENCES

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[2] AMD Accelerated Parallel Processing Math Libraries. URL: `http://developer.amd.com/tools/`.

[3] V. R. Basili, J. C. Carver, D. Cruzes, L. M. Hochstein, J. K. Hollingsworth, F. Shull, and M. V. Zelkowitz. Understanding the high-performance-computing community: A software engineer's perspective. *IEEE Software*, 25(4):29–36, 2008.

[4] C. Bauer, A. Frink, and R. Kreckel. Introduction to the ginac framework for symbolic computation within the c++ programming language. *Journal of Symbolic Computations*, 33(1):1–12, 2002.

[5] Boost C++ Libraries. URL: `http://www.boost.org/`.

[6] Boost Numeric Bindings. URL: `http://mathema.tician.de/software/boost-numeric-bindings`.

[7] Boost.uBLAS Mailing list: Anybody still there?, February 2012. URL: `http://boost.2283326.n4.nabble.com/Anybody-still-there-td4348187.html`.

[8] CGAL. URL: `http://www.cgal.org/`.

[9] Cilk. URL: `http://supertech.csail.mit.edu/cilk/`.

[10] DUNE. URL: `http://www.dune-project.org/`.

[11] Eigen. URL: `http://eigen.tuxfamily.org/`.

[12] Feel++. URL: `http://code.google.com/p/feelpp/`.

[13] GiNaC. URL: `http://www.ginac.de/`.

[14] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Inc., 1st edition, 2010.

[15] Intel Math Kernel Library. URL: `http://software.intel.com/en-us/intel-mkl`.

[16] IPython. URL: `http://ipython.org/`.

[17] Khronos Group. OpenCL. URL: `http://www.khronos.org/opencl/`.

[18] W. T. Kramer. Top500 versus sustained performance: the top problems with the top500 list - and what to do about them. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 223–230, New York, NY, USA, 2012. ACM.

[19] A. Logg. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering*, (4):283–295, 2009.

[20] S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley, 2005.

[21] MTL 4. URL: `http://www.mtl4.org/`.

[22] E. Niebler. Expressive C++: Expression Optimization.

[23] NVIDIA CUBLAS. URL: `https://developer.nvidia.com/cublas`.

[24] NVIDIA CUDA. URL: `http://www.nvidia.com/`.

[25] OpenMP. URL: `http://openmp.org/`.

[26] Y. Saad. *Iterative Methods for Sparse Linear Systems, 2nd edition*. SIAM, 2003.

[27] SciPy. URL: `http://scipy.org/`.

[28] Sundance. URL: `http://www.math.ttu.edu/~kelong/Sundance/html/`.

[29] Top500 Supercomputing Sites. URL: `http://top500.org/`.

[30] D. Vandevoorde and N. Josuttis. *C++ Templates*. Addison-Wesley, 2002.

[31] T. Veldhuizen. Expression Templates. *C++ Report*, 7(5):26–31, 1995.

[32] VexCL. URL: `https://github.com/ddemidov/vexcl/`.

[33] ViennaCL. URL: `http://viennacl.sourceforge.net/`.

[34] ViennaData. URL: `http://viennadata.sourceforge.net/`.

[35] ViennaFEM. URL: `http://viennafem.sourceforge.net/`.

[36] ViennaGrid. URL: `http://viennagrid.sourceforge.net/`.

[37] ViennaMath. URL: `http://viennamath.sourceforge.net/`.

[38] ViennaX. URL: `http://viennax.sourceforge.net/`.

[39] J. Weinbub, K. Rupp, and S. Selberherr. Utilizing Modern Programming Techniques and the Boost Libraries for Scientific Software Development. In *Proceedings of C++Now 2012*, 2012.