ViennaData 1.0.0

User Manual

Institute for Microelectronics
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria

Main Contributors:

Karl Rupp

Current Maintainers:

Karl Rupp

Institute for Microelectronics
Vienna University of Technology
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria/Europe

Phone	+43-1-58801-36001
FAX	+43-1-58801-36099
Web	`http://www.iue.tuwien.ac.at`

Contents

Introduction

When designing classes in object-oriented programming languages, one has to distinguish between intrinsic properties that are inherent to the modelled entity, and data that is associated with the entity. For example, a triangle is intrinsically defined by three disjoint points, while associated data can be color or texture information in graphical applications, or material properties for two-dimensional scientific simulations.

While associated data depends on the respective application, intrinsic properties do not. Therefore, when designing reusable classes, associated data must not be part of the class itself. Reconsidering the example of a triangle, a class member for color information would immediately result in a potentially significant overhead for scientific simulations and thus reduce the reuseability of the class. However, for graphical applications the color information for the triangle still has to be accessible. This is where `ViennaData` comes into play, because it provides a convenient means to store and access associated data for small, reusable classes with intrinsic data members only.

As an example, suppose there exists a minimalistic class for triangles called `Triangle`, which provides access to its vertices. For objects of this triangle class, color information modelled by a class `RGBColor` should be stored within some application. The following lines accomplish that for a triangle object `tri` using `ViennaData`:

```
//store RGB color information:
viennadata::access<string, RGBColor>("color")(tri) = RGBColor(255,42,0);

//get color information at some other point in the application:
RGBColor col = viennadata::access<string, Color>("color")(tri);
```

In `ViennaData` data is associated with objects by keys of user-defined type. In the example above, the color information is stored using a key `"color"` of type `std::string`. However, the user is free to use any key of any type[1]. Thus, any associated data object `data` of a generic type `DataType` can be stored and accessed for an object of any type using a key `key` of type `KeyType` by (cf. Chapter 2)

```
//store 'data' for obj
viennadata::access<KeyType, DataType>(key)(obj) = data;

//retrieve the stored data:
DataType data2 = viennadata::access<KeyType, DataType>(key)(obj);
```

Therefore, `ViennaData` allows to design and work with classes that focus on their intrinsic properties only, and ensures uniformity of associated data access over an arbitrarily large set of objects of different type. This is in particular important if several different libraries with different coding styles are used within a project.

[1]Some technical restrictions apply, see Chapter [MISSING].

With little additional configuration (cf. Chapter 3), the code for color information access using `ViennaData` can be reduced to

```
//store RGB color information
viennadata::access<color_info>()(tri) = RGBColor(255, 42, 0);

//get color information at some other point in the application:
RGBColor col = viennadata::access<color_info>()(tri);
```

This allows uniform, type-safe data access throughout the whole application. As shown in Chapter 4, there is virtually no abstraction penalty for this gain in flexibility and uniformity. Explicit use cases are given in Chapter 5.

Internally, `ViennaData` stores data in tree-like structures, for which the object address is used as key for storing the data. However, if objects provide a means to identify themselves by numbers in the range $1, \ldots, N$, the internal datastructure of `ViennaData` can be conveniently customized to a more appropriate storage scheme. Similarly, if it is a-priori known that for a particular key type only one key object is used, the internal datastructure of `ViennaData` can be adapted to that. Chapter 6 explains the internals.

Finally, a few words shall be spent on the pitfalls of `ViennaData`. Since data is associated with objects, object lifetimes have to be kept in mind. If an object, for which associated data is stored, reaches the end of its lifetime, the user has to care for the associated data separately. This can be very simple and fully transparent for the user in some cases, and rather tough in other cases. More details are given in Chapter 7.

Chapter 1

Installation

This chapter shows how ViennaData can be integrated into a project and how the examples are built. The necessary steps are outlined for several different platforms, but we could not check every possible combination of hardware, operating system and compiler. If you experience any trouble, please write to the maining list at

viennadata-support@lists.sourceforge.net

1.1 Dependencies

- A recent C++ compiler (e.g. GCC version 4.2.x or above and Visual C++ 2008 are known to work)
- CMake [1] as build system (optional, but highly recommended for building the examples)

1.2 Generic Installation of ViennaData

Since ViennaData is a header-only library, it is sufficient to copy the viennadata/ folder either into your project folder or to your global system include path. On Unix based systems, this is often /usr/include/ or /usr/local/include/.

On Windows, the situation strongly depends on your development environment. We advise users to consult the documentation of their compiler on how to set the include path correctly. With Visual Studio this is usually something like C:\Program Files\Microsoft Visual Studio 9.0\VC\include and can be set in Tools -> Options -> Projects and Solutions -> VC++-Directories.

1.3 Building the Examples and Tutorials

For building the examples, we suppose that CMake is properly set up on your system. The other dependencies are listed in Tab. 1.1.

Tutorial No.	Dependencies
tutorial/tut1.cpp	OpenCL
tutorial/tut2.cpp	OpenCL, ublas
tutorial/tut3.cpp	OpenCL, ublas
tutorial/tut4.cpp	ublas
tutorial/tut5.cpp	OpenCL
benchmarks/vector.cpp	OpenCL
benchmarks/sparse.cpp	OpenCL, ublas
benchmarks/solver.cpp	OpenCL, ublas

Table 1.1: Dependencies for the examples in the `examples/` folder

1.3.1 Linux

To build the examples, open a terminal and change to:

```
$> cd /your-ViennaData-path/build/
```

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make
```

to build the examples. If desired, you can build each example separately instead:

```
$> make blas1            #builds the blas level 1 tutorial
$> make vectorbench      #builds vector benchmarks
```

Speed up the building process by using jobs, e.g. `make -j4`.

1.3.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the `GCC` compiler the Xcode [2] package has to be installed. To install `CMake`, external portation tools such as Fink [3], DarwinPorts [4] or MacPorts [5] have to be used.

The build process of `ViennaData` is similar to Linux.

1.3.3 Windows

In the following the procedure is outlined for `Visual Studio`: Assuming that an `OpenCL` SDK and `CMake` is already installed, Visual Studio solution and project files can be created using `CMake`:

- Open the `CMake` GUI.
- Set the `ViennaData` base directory as source directory.

- Set the `build/` directory as build directory.
- Click on 'Configure' and select the appropriate generator (e.g. `Visual Studio 9 2008`)
- Click on 'Generate' (you may need to click on 'Configure' one more time before you can click on 'Generate')
- The project files can now be found in the `ViennaData` build directory, where they can be opened and compiled with Visual Studio (provided that the include and library paths are set correctly, see Sec. 1.2).

The examples and tutorials should be executed from within the `build/` directory of `ViennaData`, otherwise the sample data files cannot be found.

Chapter 2

Basic Usage

In this Section the fundamental operations in `ViennaData` are covered in full detail. All functionality is available out of the box and operations are checked. If `ViennaData` access turns out to be performance critical, several optimizations can be triggered, which is discussed in Chapter 3.

2.1 Access

The main function in `ViennaData` is the `access()` function. From a user perspective, the call

```
viennadata::access<KeyType, ValueType>(key)(obj)
```

returns a reference to the data of type `ValueType` stored for an arbitrary object `obj` and identified by the `key` object of type `KeyType`. At first sight, the offered functionality can be seen as a `std::map<KeyType, ValueType>` for each object `obj`. However, the `access` function offers a lot more flexibility, which will be discussed in the following.

As a first example, to store RGB color information on objects `obj1`, `obj2` and `obj3` using a `std::string`, say `"color"`, as key, the code

```
viennadata::access<string, RGBColor>("color")(obj1) = RGBColor(255,42,0);
viennadata::access<string, RGBColor>("color")(obj2) = RGBColor(0,42,255);
viennadata::access<string, RGBColor>("color")(obj3) = RGBColor(10,20,33);
```

is sufficient. Note that `obj1`, `obj2`, and `obj3` can be of different type. The data can be recovered directly via

```
RGBColor col1 = viennadata::access<string, RGBColor>("color")(obj1);
RGBColor col2 = viennadata::access<string, RGBColor>("color")(obj2);
RGBColor col3 = viennadata::access<string, RGBColor>("color")(obj3);
```

In addition, several different colors can be associated with a single object `obj1` as

```
viennadata::access<string, RGBColor>("front")(obj1) = RGBColor(255,42,0);
viennadata::access<string, RGBColor>("back")(obj1) = RGBColor(12,21,255);
viennadata::access<string, RGBColor>("side")(obj1) = RGBColor(30,20,10);
```

and similarly for obj2 and obj3 if desired. Since any type that fulfills the LessThan-Comparable concept [1] [CITE STL] can be used as key type, one can also use keys of different types, e.g.

```
viennadata::access<string, RGBColor>("front")(obj1)  = RGBColor(255,42,0);
viennadata::access<long,   RGBColor>( 12345 )(obj1)  = RGBColor(12,21,255);
viennadata::access<double, RGBColor>( 42.42 )(obj1)  = RGBColor(30,20,10);
```

If color is to be saved in different color spaces, this could be accomplished by adjusting the data type accordingly:

```
viennadata::access<string, RGBColor>("front")(obj1)  = RGBColor(255,42,0);
viennadata::access<long,   HSVColor>( 12345 )(obj1)  = HSVColor(12,21,255);
viennadata::access<double, CMYKColor>(42.42 )(obj1)  = CMYKColor(0,1,2,3);
```

At this point, the second template argument appears to be somewhat redundant. However, please read about the design decisions in Chapter 8 in order to get more details on why the second template paramter is still required.

In the last examples, only one key object per key type has been used. As will be shown in Chapter 3, one can completely turn off run-time comparisions of keys, so that one can write

```
viennadata::access<front_color, RGBColor>()(obj1)  = RGBColor(255,42,0);
viennadata::access<back_color,  HSVColor>()(obj1)  = HSVColor(12,21,255);
viennadata::access<side_color, CMYKColor>()(obj1)  = CMYKColor(0,1,2,3);
```

where front_color, back_color and side_color are empty tag classes [CITE] that are solely used as a key type. By using the types of the key only, the run time key comparisons are modified to compile time dispatches. This results in considerably faster execution, provided that it is already known at compile time which key has to be used.

Compile time dispatches are disabled by default. On details how to enable them, refer to Sec. 3.4

With compile time dispatches enabled, the stored data can be retained in the same way:

```
RGBColor  col1 = viennadata::access<front_color, RGBColor>()(obj1);
HSVColor  col2 = viennadata::access<back_color,  HSVColor>()(obj1);
CMYKColor col3 = viennadata::access<side_color, CMYKColor>()(obj1);
```

For the case that it is a-priori known that for each KeyType only one ValueType will be used, ViennaData can be configured to explicitly bind the KeyType to a particular ValueType. In the previous code snippet, the front_color can be bound to the RGBColor type, and similarly the back_color to HSVColor and side_color to CMYKColor. In this case, the previous code snippet reduces to

```
RGBColor  col1 = viennadata::access<front_color>()(obj1);
HSVColor  col2 = viennadata::access<back_color>()(obj1);
CMYKColor col3 = viennadata::access<side_color>()(obj1);
```

Similarly, no ValueType is used for storing data in this case.

[1] **operator**< can be used for comparisons

See Sec. 3.5 for details on how to specify a default `ValueType` for a particular `KeyType`.

2.2 Copy

In order to copy data of type `ValueType` stored using a `key` of type `KeyType` from an object `obj_src` to an object `obj_dest`, the `copy()` function can be used as follows;

```
viennadata::copy<KeyType, ValueType>(key)(obj_src, obj_dest);
```

In this case, only the particular data for the particular key is copied. If the `key` argument is omitted, i.e.

```
viennadata::copy<KeyType, ValueType>()(obj_src, obj_dest);
```

all data stored for `obj_src` using any key of type `KeyType` is copied to `obj_dest`. As an example, consider

```
viennadata::access<std::string, RGBColor>("front")(obj_src) = RGBColor
    (255,0,0);
viennadata::access<std::string, RGBColor>("back")(obj_src) = RGBColor
    (0,255,0);
viennadata::access<std::string, RGBColor>("side")(obj_src) = RGBColor
    (0,0,255);

//copy all data from obj_src to obj_dest:
viennadata::access<std::string, RGBColor>()(obj_src, obj_dest);

//access the front color (result will be RGBColor(255, 0, 0))
RGBColor red = viennadata::access<std::string, RGBColor>("front")(obj_dest
    );
```

In this example, the colors stored for `obj_src` are copied to `obj_dest` and can then be accessed using the keys `"front"`, `"back"` and `"side"`.

`obj_src` and `obj_dest` do not need to be of the same type for basic `copy()` functionality!

For technical reasons, `obj_src` and `obj_dest` have to be of the same type if `all` is used.

So far, every pair of `KeyType` and `ValueType` has to be copied by a separate function call. For larger projects where multiple different keys and data types are used, this is too tedious. To copy all data of possibly different type for a particular key, one can use the dedicated `all` specifier if :

```
viennadata::copy<KeyType, viennadata:all>()(obj_src, obj_dest);
```

Similarly, to copy all data of type `ValueType` stored for `obj_src` to `obj_dest`, the line

```
viennadata::copy<viennadata:all, ValueType>()(obj_src, obj_dest);
```

can be used. Finally, to copy all data of arbitrary type stored using any key of any type for `obj_src` to `obj_dest`, the line

```
viennadata::copy<viennadata:all, viennadata:all>()(obj_src, obj_dest);
```

is sufficient.

In order to have `viennadata:all` working correctly, every pair of `KeyType` and `ValueyType` in use must be registered. By default, this is done automatically in the background. If the library user decides to disable automatic registration, each such pair must be registered manually (cf. Section 3.3).

2.3 Move

Data of type `ValueType` stored using a `key` of type `KeyType` for an object `obj_src` can be transferred to another object `obj_dest` by using the `move()` function as follows:

```
viennadata::move<KeyType, ValueType>(key)(obj_src, obj_dest);
```

The difference to the `copy()` function in the previous section is that the data with `move()` the data is then only available for `obj_dest`.

Similar to `copy()`, the following variants are available:

```
viennadata::move<KeyType, ValueType>()(obj_src, obj_dest);
viennadata::move<KeyType, viennadata:all>()(obj_src, obj_dest);
viennadata::move<viennadata:all, ValueType>()(obj_src, obj_dest);
viennadata::move<viennadata:all, viennadata:all>()(obj_src, obj_dest);
```

Since their use is similar to that of the `copy()` variants, the reader is referred to the previous section.

For technical reasons, `obj_src` and `obj_dest` have to be of the same type if `all` is used.

2.4 Erase

In order to delete data of type `ValueType` stored using a `key` of type `KeyType` for an object `obj`, the `erase()` function can be used as follows:

```
viennadata::erase<KeyType, ValueType>(key)(obj);
```

Similar to `copy()` and `move()`, the following variants are available:

```
viennadata::erase<KeyType, ValueType>()(obj);
viennadata::erase<KeyType, viennadata:all>()(obj);
viennadata::erase<viennadata:all, ValueType>()(obj);
viennadata::erase<viennadata:all, viennadata:all>()(obj);
```

They allow to delete larger amounts of data associated with `obj` by a single call. For more explainations, the reader is referred to Section 2.2.

2.5 Query Data Availability

`viennadata::find()` allows to check whether data is stored for a particular object. If there is data stored, a pointer to the data is returned. Otherwise, a `NULL` pointer is returned. the following code snippets shows how to check for data of type `ValueType` associated with `obj` using a `key` of type `KeyType`:

```
if (viennadtaa::find<KeyType, ValueType>(key)(obj))
  /* data available */
else
  /* data not yet available */
```

`find` is very attractive if there is a large number of objects with only a few carrying data. Using `viennadata::access` to query for data existence by iterating through all objects would create data of type `ValueType` for every object. This overhead is avoided with `viennadata::find`.

Chapter 3

Advanced Usage

While Chapter 2 outlined the basic usage of `ViennaData`, this Chapter focuses on customizing the library to the user's demands and the particular environment. It is explained how the object identification can be changed, how the internal storage of data can be customized and data accesses can be tuned for maximum speed.

3.1 Retrieving IDs from Objects

In order to associate data with objects, one has to distinguish between different objects. The only portable identification mechanism is the address of an object, thus this is the default identification mechanism `ViennaData`. However, object addresses can be randomly scattered over the address space, which leads to tree-based data structures in order to provide a reasonably fast lookup of data.

However, it can be the case that there are additional means for identification provided. Let us consider a class `MyClass` equipped with a member function `id()` that is used to identify objects. The following code configures `ViennaData` to use this member function as identification mechanism for all objects

```
namespace viennadata
{
 namespace traits
 {
  template <>
  struct id<MyClass>
  {
   typedef long   id_type;
   static id_type get(MyClass const & obj) { return obj.id(); }
  };
 }
}
```

Summing up, the library user has to provide a specialization for the template class `id` for each class and provide a type definition for the ID-type `id_type` and provide a static function that returns the ID for an object of the respective class.

3.2 Dense Data

By default, `ViennaData` internally uses a tree-based layout similar to

```
std::map<ObjType *, DataContainer>
```

to store and retrieve data for objects of type `MyClass` by the use of a data container type `DataContainer`. This induces lookup-costs of the order $\log N$, where N denotes the number of objects for which data is stored.

However, if the range of IDs of objects of a type `MyClass` is known to be in the range $0, \ldots, N]$, a tree-based layout can be configured to be of type

```
std::vector<DataContainer>
```

This requires that a custom identification mechanism is set up as outlined in the previous section. Moreover, the custom `id_type` must be compatible with being used as an array index. Since such a storage scheme is only memory efficient if the number of objects is comparable to N, we refer to this scheme as *dense data storage*. To change the default tree-based data storage scheme to the dense storage scheme for objects of type `MyClass`, the lines

```
namespace viennadata
{
 namespace traits
 {
  template <typename KeyType, typename ValueType>
  struct storage<KeyType, ValueType, MyClass>
  {
   typedef dense_data_tag    tag;
  };
 }
}
```

are sufficient. Due to the template specialization, a fine-grained control over which storage to use for the various types of data and keys is readily provided.

While tree-based schemes offer automatic memory management, this is not the case for the dense data storage. In this case, the user has to reserve the required memory manually *prior to any other calls* of `ViennaData` functions. To reserve data memory for data of type `ValueType` with keys of type `KeyType` for 42 objects of the same type as `obj`, the code

```
viennadata::reserve<KeyType,ValueType>(42)(obj);
```

is required. Similarly, memory for other data, key and object types is reserved.

The dense storage scheme should only be used if either memory consumption or data access times turn out to be critical.

3.3 Register Data

The use of `vienndata::all` for the `copy()`, `move()` and `erase()` functions as described in Chapter 2 requires the tracking of types used at compile time during run time. This is by default handled automatically in `ViennaData` at the cost of a small overhead at data access. For performance-critical applications, this automatism can be disabled for each triple of key type `KeyType`, data type `ValueType` and object type `ObjectType` by

```
namespace viennadata
{
 namespace traits
 {
  template <>
  struct signup<KeyType, ValueType, MyClass>
  {
   typedef manual_signup_tag    tag;
  };
 }
}
```

Again, the partial specialization can be adjusted to match a larger class of types.

Once the automatic tracking of types at runtime is disabled, the correct use of `viennadata::all` requires that all each triple of key type `KeyType`, data type `ValueType` and object type `ObjectType` is registered manually as

```
  viennadata::signup<KeyType,ValueType>()(obj);
```

Here, `obj` is of type `ObjType`. Note that multiple calls of `viennadata::signup()` with the same types has the same effect as a single call.

3.4 Type-Based Key Dispatch

As already shown in the Chapter 2, `ViennaData` can be configured to perform a purely type-based key dispatch for data access. The impact on performance can be considerable, especially when compared to potentially costly string comparisons. To enable a compile time dispatch for a key of type `KeyType`, the lines

```
namespace viennadata
{
 namespace traits
 {
  template <>
  struct dispatch<KeyType>
  {
    typedef type_key_dispatch_tag    tag;
  };
 }
}
```

are required.

3.5 Default Data Types for Keys

For the cases where only one data type per key type is used, `ViennaData` can be provided with a default data type for each key type. This is especially common when using a type-based key dispatch for data access. Consider for example

```
viennadata::access<front_color, RGBColor>()(obj1) = RGBColor(255,42,0);
viennadata::access<back_color,  RGBColor>()(obj1) = RGBColor(0,42,255);
viennadata::access<edge_color,  RGBColor>()(obj1) = RGBColor(255,0,42);
```

Since the key classes are already named appropriately and always the some data type `RGBColor` is used, setting `RGBColor` as default data type for the respective keys allows to make the code more compact. This can be enabled along the lines

```
namespace viennadata
{
 namespace traits
 {
  template <>
  struct default_data_for_key<front_color>
  {
    typedef RGBColor    type;
  };
  // and similary for back_color, edge_color
 }
}
```

Now, the initial code snippet can be written more compactly as

```
viennadata::access<front_color>()(obj1)  = RGBColor(255,42,0);
viennadata::access<back_color>()(obj1)  = RGBColor(0,42,255);
viennadata::access<edge_color>()(obj1)  = RGBColor(255,0,42);
```

If no default data type is specified, but the second argument is nevertheless omitted, a compile time error is thrown.

Chapter 4

Benchmark Results

Since `ViennaData` introduces a layer for handling associated data, it is rather natural to ask for any potential runtime overhead. The aim of this Chapter is to give a precise answer.

4.1 ViennaData Storage Models

There are the following three different object-to-data models available in `ViennaData`:

- Object identification via object addresses, sparse storage.
- Custom identification via dedicated `id()` member, sparse storage.
- Custom identification via dedicated `id()` member, dense storage (cf. Sec. 3.2).

In addition, the following data key dispatch mechanisms are compared in the following:

- Key dispatch at runtime for keys of type `std::string`.
- Key dispatch at runtime for keys of type `long`.
- Key dispatch at compile time (cf. Sec. 3.4).

This results in nine different configurations for the subsequent tests. All comparisions are carried out for objects of the following class:

```
class SlimClass
{
  public:
    SlimClass(double v = 1.0, size_t i = 0) : value_(v), id_(i) {}

    double value() const { return value_; }
    size_t id() const { return id_; }
  private:
    double value_;
    size_t id_;
};
```

The first test with execution times given in Tab. 4.1 compares the execution times of summing up random values stored and retrieved for 1 000 objects of type `SlimClass` using `viennadata::access<>()`, repeated 1 000 times.

	Key Dispatch		
Identification	`std::string`	`long`	compile time
`&obj, sparse`	84	53	43
`obj.id(), sparse`	83	52	43
`obj.id(), dense`	48	15	**4**

Table 4.1: Execution time for summing up data of 1 000 objects of type `SlimClass`, repeated 1 000 times. Execution times in milliseconds.

	Key Dispatch		
Identification	`std::string`	`long`	compile time
`&obj, sparse`	333	247	173
`obj.id(), sparse`	318	228	178
`obj.id(), dense`	151	73	**5**

Table 4.2: Execution time for summing up data of 1 000 000 objects of type `SlimClass`. Execution times in milliseconds.

Let us first consider the differences among the three rows. As can clearly be seen, a change of the object identification from address-based to using the member function `id()` does not change execution speed, if the internal storage scheme remains tree-based (*sparse*). However, if the `id()` member is used for a dense storage scheme as described in Section 3.2, execution times are reduced by around 40 milliseconds irrespective of the key dispatch method employed.

Comparing the different colums in 4.1, it can be seen that a string-based dispatch is most expensive as expected. A dispatch based on integers is faster, but the differences only become significant if a dense storage scheme is used. The compile-time dispatch is notably faster than dispatches at run time. In particular, for the dense storage scheme execution times differ by a factor of four and twelve respectivly.

In a second test, the same procedure is carried out for 1 000 000 objects without repetition. Thus, a naive guess is that execution times should be roughly the same. However, as Tab. 4.2 shows, the logarithmic lookup times in a sparse, tree-based storage layout leads to much higher execution times than for the first test. This may be also partly due to less efficient CPU cache use caused by the larger amount of data involved.

Finally, it should be emphasized that in real applications there is typically much less emphasis on data retrieval as compared to our benchmarks. Thus, even though a sparse storage scheme is slower than a dense storage scheme, it may be fast enough for most applications.

4.2 ViennaData vs. Traditional OOP

As discussed in the introduction already, `ViennaData` allows to design small, lightweight classes with high reusability. In this section we compare data access times of `ViennaData` with other classes that stem from traditional OOP design and thus have a higher number of class member variables.

The objects for which data is attached using `ViennaData` are of type `SlimClass` as intro-

Setup	Exec. Time (us), 10^3 Objects	Exec. Time (ms), 10^6 Objects
`ViennaData`, config 1	84	333
`ViennaData`, config 2	52	228
`ViennaData`, config 3	**4**	**5**
OOP, `FatClass<10>`	**1.3**	**4**
OOP, `FatClass<100>`	2.1	11
OOP, `FatClass<1000>`	2.5	11

Table 4.3: Execution time for summing up one `double` value from each object.

duced in the previous section. The following three access models are used:

1. Object identification via object addresses, sparse storage, key dispatch at runtime for keys of type `std::string`.
2. Custom identification via dedicated `id()` member, sparse storage, key dispatch at runtime for keys of type `long`.
3. Custom identification via dedicated `id()` member, dense storage, key dispatch at compile time.

Note that the three models represents the diagonals in Tab. 4.1 and Tab. 4.2.

For the traditional OOP classes, we use the following template class

```
template <size_t num_bytes>
class FatClass
{
  public:
    FatClass(double v = 1.0, size_t i = 0) : value_(v), id_(i) {}

    double value() const { return value_; }
    size_t id() const { return id_; }
  private:
    double value_;
    char payload[num_bytes];
    size_t id_;
};
```

where the template parameter `num_bytes` denotes the number of bytes used for other data members. `FatClass<10>`, `FatClass<100>` and `FatClass<1000>` are used.

As can be seen in Tab. 4.3, access times with dense storage are by a factor of two larger for 1 000 objects, but by a factor of two smaller when using 1 000 000 objects. Thus, `ViennaData` allows to achieve even better execution performance than traditional OOP for a large number of objects. This is mostly due to better caching possibilities for the dense storage layout, where similar data for different objects is located on a linear piece of memory.

The sparse storage layout is not competitive in terms of access times as the number of objects grows. However, data access times are often negligible compared to data manipulation, thus the increased data access time will be in many cases still negligible on the overall.

Chapter 5

Use Case: Scientific Simulation

In this chapter we discuss the origins of ViennaData, namely the use for scientic simulations. More precisely, the use of ViennaData for the solution of partial differential equations by means of discretization methods such as the Finite Element Method or the Finite Volume Method are discussed.

In the following it is assumed that the simulation domain, which can be one-, two- or three-dimensional, is decomposed into small elements, say lines, triangles and tetrahedrons respectively. The simulation domains can represent arbitrary objects, for instance buildings, human bones or microelectronic devices, and the physical effects simulated can be displacements, stresses, heat, charge densities, etc.

5.1 Handling Material Properties

Let us consider the classes Line, Triangle and Tetrahedron that reflect the respective geometric primitives. When running a two-dimensional simulation, material properties have to be associated with triangles, thus one might be tempted to add a material member variable of some type, say, Material to the Triangle class. However, reusing the same class for facets of a Tetrahedron in three-dimensional simulations would result in an unused member variable material in Triangle, because materials have to be associated with a Tetrahedron in three dimensions. Similarly, material information should not be a member of a line.

In order to preserve reusability of the Line, Triangle and Tetrahedron classes, ViennaData can be used as follows for the three-dimensional case:

```
Tetrahedron tet;
/* initialize tet */

//store material on tet using ViennaData:
viennadata::access<string, Material>("material")(tet) = Material("wood");
```

Then, at any later point, the material for tet can be conveniently retrieved by

```
Material tet_mat = viennadata::access<string, Material>("material")(tet);
```

Other keys can be used as described in Chapter 2. Note that one will typcially have thousands of cells, not just a single one.

In order to allow a uniform material retrieval independent of the spatial dimension used for the simulation, one can provide a **typedef** for the respective cell type:

```
typedef Tetrahedron     CellType;  //for 3d
//typedef Triangle      CellType;  //for 2d
//typedef Line          CellType;  //for 1d

CellType cell;

/* Initialize cell here */

//store data on cell:
viennadata::access<string, Material>("material")(cell) = Material("wood");

//retrieve material from cell at some later point:
Material c_mat = viennadata::access<string, Material>("material")(cell);
```

Thus, in this setting `ViennaData` provides unified access for objects of type `Line`, `Triangle` and `Tetrahedron`. Moreover, one is not restricted to a particular implementation and can still exchange the implementations for the geometric primitives at some later point without affecting the storage of materials.

By using `ViennaData`, one can easily exchange the implementation of the objects the data is associated with.

5.2 Use with Discretization Methods

The various discretization schemes such as the Finite Volume Method or the Finite Element Method associate unknowns with vertices, lines, facets and/or cells within a mesh. A similar reasoning as for the storage of material properties applies to the storage of unknowns (or their identification indices) as well. Moreover, another aspect is that different effects are simulated. For instance, one may want to simulate mechanical stresses as well as heat flow in a building. Thus, data for the different simulations has to be separated in addition. Such a scenario would be almost impossible to handle if the unknown IDs were members of the respective classes `Line`, `Triangle` and `Tetrahedron`. However, with `ViennaData` unknown indices can be stored and retrieved for a point object `vertex` as

```
//store unknown index id_1 for simulation 1 (long sim_1 = 1;):
viennadata::access<long, UnknownIndex>(sim_1)(vertex) = id_1;
//store unknown index id_2 for simulation 2 (long sim_2 = 2;):
viennadata::access<long, UnknownIndex>(sim_2)(vertex) = id_2;

/* lots of other computations done here */

//during simulation 1:
UnknownIndex id_1 = viennadata::access<long, UnknownIndex>(sim_1)(vertex);
//during simulation 2:
UnknownIndex id_2 = viennadata::access<long, UnknownIndex>(sim_2)(vertex);
```

Thus, not only remains the implementation of geometric primitives light-weight and reusable, the resulting code due to `ViennaData` is very compact and expressive. To obtain high execution performance, a dense storage scheme should be employed for accessing the unknown indices, cf. Sec. 3.2.

Another important aspect for scientific simulation is the handling of boundary conditions. Typically only a subset of elements on the boundary is effected by a certain type of boundary conditions, thus it would be highly inefficient to use a boolean member variable for all elements in the mesh. Here, the default sparse storage scheme of ViennaData turns out to be memory-efficient when for instance flagging boundary triangles:

```
// tag tri1, tri42 and tri360 for boundary conditions in simulation 1:
viennadata::access<long, bool>(sim_1)(tri1)   = true;
viennadata::access<long, bool>(sim_1)(tri42)  = true;
viennadata::access<long, bool>(sim_1)(tri360) = true;
// tag only tri1 and tri360 for boundary conditions in simulation 1:
viennadata::access<long, bool>(sim_2)(tri1)   = true;
viennadata::access<long, bool>(sim_2)(tri360) = true;

/* lots of other computations done here */

// iterate over triangles using tri_iter in simulation 1
// and check for boundary conditions:
if ( viennadata::find<long, bool>(sim_1)(*tri_iter) )
  //boundary triangle for simulation 1
else
  //no boundary

// iterate over triangles using tri_iter in simulation 1
// and check for boundary conditions:
if ( viennadata::find<long, bool>(sim_2)(*tri_iter) )
  //boundary triangle for simulation 2
else
  //no boundary
```

Note that by using viennadata::access in the if-clauses, a boolean would be default-constructed and stored on each triangle. By using viennadata::find, only three and two booleans are stored for the two simulations respectively.

ViennaData provides a flexible, unified interface for both sparse and dense storage schemes.

Chapter 6

Library Internals

In order to configure ViennaData in the best way for the respective project and to avoid common pitfalls, it is of advantage to understand some of the library-internals. The focus is on explaining the *big picture* rather than in going through all the details. These details can either be found in the Doxygen generated documentation, or directly in the source code.

6.1 Main Classes

All data resides in the instances of the template class

```
template <typename KeyType,
          typename ValueType,
          typename ObjectType>
class data_container;
```

located in data_container.hpp. As can be seen from the list of template parameters, each triple of key type, data type and object type lets the compiler create another instance of data_container. Thus, compilation times will increase with the number of different types involved.

data_container uses a singleton pattern and is directly interfaced by the interface functions

```
viennadata::access<KeyType, ValueType>()()
viennadata::copy<KeyType,   ValueType>()()
viennadata::move<KeyType,   ValueType>()()
viennadata::erase<KeyType,  ValueType>()()
viennadata::find<KeyType,   ValueType>()()
```

The two parenthesis arguments are due to the following:

1. The inner parenthesis belong to a function returning a proxy class
2. The outer parenthesis denote the parenthesis operator `operator()` of the proxy class

Within the parenthesis operator, the appropriate functions of data_container are called.

The storage type in data_container is obtained from a traits class container_traits, which provides the correct storage scheme. By default, the storage is equivalent to

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

Thus, the first call call

```
viennadata::access<KeyType, ValueType>(key)(obj) = data;
```

for `obj` of type `Object` is equivalent to

```
std::map<ObjectType *, std::map<KeyType, ValueType> >  storage;
storage[&obj][key] = data;
```

and subsequent calls reuse the `storage` object. If a dense storage scheme is used (cf. Sec. 3.2 and `storage_traits.hpp`), the storage type is equivalent to

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

and the first call to `viennadata::access` can be seen as

```
std::vector< std::map<KeyType, ValueType> >  storage;
storage.resize(some_size);
storage[obj.id()][key] = data;
```

The configuration of the ID retrieval (here: `obj.id()`) is discussed in Sec. 3.1 and implemented in `object_identifier.hpp`, while the need to call `viennadata::reserve` in order to provide a value for `some_size` is discussed in Sec. 3.2. Finally, a compile time key dispatch (cf. Sec. 3.4) eliminates the inner map of the storage scheme, hence the data is organized as

```
std::vector< ValueType >             dense_storage;
```

when using dense storage, and as

```
std::map<ObjectType *, ValueType >  sparse_storage;
```

for sparse storage. The traits classes in `data_container_taits.hpp` ensure that the correct access mechanism is used for the various operations.

6.2 viennadata::all

In order to make the following code lines

```
viennadata::access<std::string, long>("foo")(obj1) = 42;
viennadata::access<long,        double>(360)(obj1) = 3.1415;
viennadata::copy<viennadata::all, viennadata::all>()(obj1, obj2);
```

work as expected, a number of coding tricks have to be applied. As outlined in the previous section, the first two lines create two storage objects

```
std::map<ObjectType *, std::map<std::string, long> >
std::map<ObjectType *, std::map<long, double> >
```

for which the data needs to be transferred. However, there is no means to iterate over compiler generated types, therefore the generated types have to be tracked at runtime.

A technique called *type erasure* [CITE] is used to store pairs of different (!) `KeyType`s and `ValueType`s in an array, see `key_value_pair.hpp` for the type erasure and `key_value_registration.hpp` for the array. When using the `viennadata::all` type for copy, move or erase operations, the type-erasured `KeyType`s and `ValueType`s are iterated and compared. If the types match, a virtual function call unwraps the types and calls the appropriate `data_container`-member. This virtual function call is the reason why the source and the destination object have to be of the same type.

Chapter 7

Pitfalls

The increased reusability of classes due to a separation of intrinsic and associated data (cf. Introduction) introduces a few pitfalls one should be aware of. Depending on the code design and the application, these pitfalls can be somewhere between completely unrealistic and painful. In the following, we discuss the most important of them.

7.1 Object Lifetimes

Consider the following code, where the class `A` is defined somewhere else:

```
A * myA = new A();
viennadata::access<long, double>(42)(*myA) = 3.1415;
delete myA;
```

The problem here is that data associated with `A` is not erased prior to the deletion of `myA`. Thus, there is still the double value 3.1415 stored using the address of `myA`, but there is no direct way to access that data. To avoid confusion: The data is not lost in a sense of memory leak, because the tree managing the data is still consistent and properly free'd at program exit. However, the user has no direct way to retain the an object that has the same address as the one pointed to by `myA`.

In order to ensure that all data is properly deleted when the respective object is destroyed, the following options are available:

- Manually call `viennadata::erase<KeyType, ValueType>()(*myA)` for all pairs of `KeyType` and `ValueType` used.
- Manually ensure that `viennadata::erase<all,all>()(*myA)` is called prior to deleting `*myA`. This requires that all data is registered properly.
- Use `viennadata::free(*myA)` instead of `delete`. This will call `viennadata::erase<all,all>()(*myA)` and then `delete myA`. It will not work if `myA` points to an array, for which `delete[]` has be called.
- If the sources of `myA` can be modified, then

  ```
  A::~A()
  {
    viennadata::erase<all,all>()(*this);
  }
  ```

will automatically remove all data stored for the object, unless automatic registration of key and data types is turned off.

- Provide another identification mechanism that is not based on object lifetimes, see Sec. 3.1. Thus, if you create an object with the same ID at some later time, the data can be accessed again.

7.2 Polymorphism

Since ViennaData relies on a static type system, it cannot reasonably deal with polymorphism. Consider

```
class Base { };
class Derived : public Base {};

Derived d;
Base * pb = &d;
viennadata<long, double>(360)(*pb) = 3.1415;
std::cout << viennadata<long, double>(360)(d) << std::endl;
```

Even though data is stored for the same object, the resulting program will still print a zero value. The reason is that data is stored for an object of type Base, while it is retrieved for an object of type Derived.

At present there is no direct way to overcome this issue. Future versions of ViennaData may provide means to work with pointers to base classes.

The use of ViennaData with polymorphism may lead to unexpected results!

Chapter 8

Design Decisions

Every now and then a programmer is faced with the question "Why have you implemented it that way?". This is then the chance to outline the various pros and cons of different approaches and assign appropriate weights such that the approach taken turns out to be the best. The first question is sometimes followed or replaced by "Why haven't you done it like this: ....?". In this case, the various pros and cons of different approaches should be outlined again and the taken weights have to be justified - even if it often tends be become a rather emotional debate.

In this chapter the various design considerations are presented and the reasons why `ViennaData` is the way it is are given. Remaining questions or inputs for further improvements should be sent to the mailinglist at

`viennadata-support@lists.sourceforge.net`

8.1 Two Pairs of Parenthesis

Probably the most immediate question relates to the two pairs of parenthesis as in

```
viennadata::access<long, double>(42)(obj) = 360.0;
```

One could have just used a single pair as follows:

```
viennadata::access<long, double>(42, obj) = 360.0;
```

However, the situation is more delicate with other functions of `ViennaData`. Consider the alternatives

```
//reserve space for 100 objects, keys of type long:
viennadata::reserve<long, DataType>(100, obj);

//copy data stored at key '42' for obj1 to obj2:
viennadata::copy<long, DataType>(42, obj1, obj2);

//copy all data with keys of type long:
viennadata::copy<long, DataType>(obj1, obj2);

//erase all data with keys of type long for obj:
viennadata::erase<long, DataType>(obj);
```

Even though only three different functions were used, there are four different meanings of the first parameters: First, the parameter denotes the number of objects, then it denotes the data, after that the source object, and finally the object for which data is to be erased. In particular the last line is not intuitive - it suggests that `obj` is erased. Compare with the previous snippet with the actual code:

```
viennadata::reserve<long, DataType>(100)(obj);
viennadata::copy<long,  DataType>(42)(obj1, obj2);
viennadata::copy<long,  DataType>()(obj1, obj2);
viennadata::erase<long, DataType>()(obj);
```

Here, the first pair of parenthesis always refers to the set of data manipulated, while the parameters in the second pair always refer to the objects the data refers to. In the library author's point of view, the second version is more intuitive and has thus been selected.

> The first pair of parenthesis always refers to the data being manipulated (either key or number of objects to be reserved), while the second pair of parenthesis always refers to the objects the data is associated with.

8.2 KeyType and DataType Template Parameters

Considering the code

```
//store '360.0' for 'obj' using key 42
viennadata::access<long, double>(42)(obj) = 360.0;
```

one is tempted to ask whether the template parameters are redundant. In this example, this is certainly the case. However, a slight variation of the code above looks like this:

```
//store '360.0' for 'obj' using key 42
viennadata::access<std::string, double>("42")(obj) = 360.0;
```

Clearly, as soon as character strings are used as key, type deduction fails. Moreover, when using compile time key dispatch, one has to supply the key type anyway.

As for the data type, which could be automatically deduced to be of type `double`, there are two problems involved: The one is that for accessing data with possibly implicit conversion, there is no other option but to supply the data type:

```
//store '360.0' for 'obj' using key 42
int value = viennadata::access<long, double>(42)(obj);
```

The other problem is due to unwanted side-effects when changing data types while refactoring code. If the `DataType` were optional, the code

```
long data = 42;
/* maybe some other code here */

//store data for obj:
viennadata::access<std::string>("data")(obj) = data;

//at some other place, maybe even a different source file:
long data2 = viennadata::access<std::string, long>("data")(obj);
```

were legal. However, when changing the type of `data` from **`long`** to, say, **`int`**, the data retrieval would be broken. Thus, to unify the interface for access, copy, move, erase and find, the key type and the data type always have to be supplied. The only possible exception has to be explicitly enabled by the user, cf. Sec. 3.5.

Always specify the key type and the data type when calling functions from `ViennaData`

8.3 Memory Management for Dense Data Storage

Dense data storage is only configured for optimization purposes, thus no runtime penalty acceptable.

Change Logs

Version 1.0.0

First release

License

Bibliography

[1] "CMake." [Online]. Available: http://www.cmake.org/

[2] "Xcode Developer Tools." [Online]. Available: http://developer.apple.com/technologies/tools/xcode.html

[3] "Fink." [Online]. Available: http://www.finkproject.org/

[4] "DarwinPorts." [Online]. Available: http://darwinports.com/

[5] "MacPorts." [Online]. Available: http://www.macports.org/