ViennaData 1.0.0

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

<div align="center">

`viennadata-support@lists.sourceforge.net`

</div>

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.

4

- 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 ⚠

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` that is 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`:

```cpp
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 `VeinnaData`, 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 `VeinnaData`. 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 `VeinnaData` to use this member function as identification mechanism for all objects

```cpp
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.

11

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::deque<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

```cpp
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

```cpp
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

```cpp
   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

```cpp
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

```cpp
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

4.1 ViennaData Storage Models

compare access times for sparse storage and dense storage, including various dispatches

4.2 ViennaData vs. Traditional OOP

Compare ViennaData with traditional OOP concepts where data is injected as class members

Chapter 5

Use Cases

In this chapter we discuss two use-cases for `ViennaData`. The two use-cases are intentionally chosen to be very distinct from each other. The first use-case is admittedly artificial, since the author of `ViennaData` has not been involved in programming computer games yet. The second use-case describes the solution of partial differential equations by means of discretization methods. Since the author is actively working on the simulation of semiconductor devices, this can be seen as the roots of `ViennaData`.

5.1   Human-Like Characters in Online Games

5.2   Solution of Partial Differential Equations

Chapter 6

Library Internals

What is going on internally.

6.1 Classes

Main classes and their interaction

6.2 viennadata::all

Explain effort required for making viennadata::all work.

6.3 Programming Techniques

A few words here

Chapter 7

Pitfalls

7.1 Object Lifetimes

Object gone, data gone...

7.2 Key-Data Registration

If you forget to register your data, you may not be able to find it again...

Chapter 8

Design Decisions

8.1   Key Type and Value Type Template Parameters

Why it is ¡KeyType, ValueType¿ all over the place...

8.2   Limitations of the all-type?

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/