ViennaMath 1.0.0

User Manual

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

Institute for Analysis and Scientific Computing
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

Author and Project Head:

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`

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

Phone	+43-1-58801-10101
Web	`http://www.asc.tuwien.ac.at`

Contents

Introduction

The beginning of the computing era is inherently linked with the use of computing devices for numerical algorithms. While the first implementations had been carried out in assembly language, the FORTRAN language then provided a means for efficiently abstracting the underlying hardware. This approach to programming is still common for the implementation of numerical algorithms today: Mathematical primitives such as polynomials or functions are in one way or another represented as an array of numbers already in user code. The mathematical meaning of these numbers is, if at all, only implicitly deducible for a programmer.

Abstraction facilities provided by modern programming languages such as C++ are much more powerful and mature compared to an abstraction at an array level. However, implementation guidelines in numerical textbooks written in the past cannot reflect the current state-of-the-art in programming, thus algorithms are often still implemented without making use of additional abstraction mechanisms. One of the reasons is that first attemps of using object-oriented programming have reported poor performance due to additional indirections at runtime. However, this issue (and several others) have been adressed by language improvements and/or programming techniques.

The aim of `ViennaMath` is to show by example that preserving the mathematical abstraction by means of a symbolic math engine can make code much more readable without sacrificing performance. Even though `ViennaMath` makes use of object-oriented programming, certain calculations and optimizations can already be carried out already at compile time, thus eliminating many expensive indirections at runtime. Consequently, the process of hand-tuning code, i.e. the use of information already available at compile time, is shifted from the user to the compiler. This increases productivity without reducing performance.

Chapter 1

Installation

This chapter shows how ViennaMath 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 mailing list at

viennamath-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 recommended for building the examples)

1.2 Generic Installation of ViennaMath

Since ViennaMath is a header-only library, it is sufficient to copy the `viennamath/` 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. Please consult the documentation of your compiler or development environment on how to set the include path correctly. The include paths in Visual Studio are 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

An overview of available examples and their purpose is given Tab. 1.1. For building the examples, we suppose that CMake is properly set up on your system. In the following, instructions on how to build the examples on different platforms are given.

File	Purpose
`basic.cpp`	Basic handling of `ViennaMath` expressions
`latex_output.cpp`	How to use and customize the LaTeXtranslator
`model_benchmark.cpp`	An example of how `ViennaMath` can eliminate dependencies in an expression
`newton_solve.cpp`	A Newton solver using `ViennaMath` expressions
`traversal.cpp`	How to traverse a `ViennaMath` expressions
`substitute.cpp`	Substitute terms in a `ViennaMath` expressions
`vector_expr.cpp`	Explains the use of vector expressions

Table 1.1: Overview of the examples in the `examples/` folder

1.3.1 Linux

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

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

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make
```

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

```
$> make basic              #builds the 'basic' tutorial
$> make substitute         #builds the 'substitute' tutorial
```

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 `ViennaMath` 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 `ViennaMath` 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 'Configure' again.
- Click on 'Generate' in order to let `CMake` generate the project files for you.
- The project files can now be found in the `ViennaMath` 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).

Chapter 2

Basic Types

Since C++ is a statically typed language [CITE], the basic mathematical building blocks such as constants or variables are represented as types. The possiblity of manipulations at compiletime or runtime is accomplished by essentially two different implementations of these primitives. Basic types for runtime evaluations are discussed first, since their interface and handling is potentially more familiar to average C++ programmers. Sec. 2.2 then provides an overview of the basic types used for compiletime manipulations.

The main include file for `ViennaMath` is `viennamath/expression.hpp` and includes all the types discussed in the remainder of this chapter.

> Include `viennamath/expression.hpp` to make all `ViennaMath` types available.

> Note that all types reside in namespace `viennamath`. The namespace is not written explicitly in the following, thus either `viennamath::` prefixes or certain **`using`** declarations need to be added by the user in order to make the code valid.

2.1 Types Evaluated at Runtime

Common to all types represented at runtime is that they inherit from the same abstract base class and can thus be accessed and manipulated using a pointer to that interface. The interface is not fixed a-priori and can be adjusted via a template parameter, which is in the following called `InterfaceType`. Library users should use the expression wrapper objects discussed next, because it provides an automatic memory management and does not involve complicated pointer manipulation.

2.1.1 Expression Wrapper `expr`

The main expression wrapper type in `ViennaMath` is `rt_expr<InterfaceType>`. The prefix `rt` refers to *runtime* and aids in distinguishing between types processed at runtime, and types processed at compiletime. In most cases, the default parameter for the runtime interface `InterfaceType` is used, in which case users would have to write

```
rt_expr<> my_expression = /* any expression here */;
```

for instantiating an expression wrapper object `my_expression`. In order to avoid users from having to write the `rt_` and the lower-than and greater-than signs, there is a convenience shortcut `expr` provided. The previous code line thus becomes

```
expr my_expression = /* any expression here */;
```

The `expr`-type can be evaluated and manipulated using operator overloads. For example, the addition of two expressions is accomplished by

```
expr ex1 = /* any expression here */;
expr ex2 = /* any expression here */;
expr result = ex1 + ex2;
```

The initalization of expression objects is accomplished by any of the fundamental types discussed in the next subsections. Note that objects of type `expr` are default-constructible, yet they can only be used after an expression has been assigned to them.

2.1.2 Constant

Constants in C++ have their own types **double**, **long**, etc. These types can be used with `ViennaMath` directly. In order to also represent constants using a pointer to the runtime interface, a separate class `rt_constant<NumericT, InterfaceType>` is provided. The template parameter `NumericT` denotes the underlying numerical type such as **double**, **long**, or high precision types. There is again a convenience shortcut `constant` provided for the case of the commonly used `rt_constant<double>`, hence a user can write code such as

```
constant pi = 3.1415;
constant pi_squared = pi * pi;
```

An exemplary use with the expression wrapper `expr` is

```
constant pi = 3.1415;
expr pi_squared = pi * pi;
expr result = pi + pi_squared;
```

2.1.3 Variable

A mathematical variable in `ViennaMath` is modeled by `rt_variable<InterfaceType>`. and refers to the mapping

$$(x_0, x_1, \ldots, x_{N-1}) \mapsto x_j \ ,$$

where the value of j is provided to the constructor of the variable. By default, the index $j = 0$ is used. Any vector type offering access to its values using **operator**`[]` such as `std::vector<T>` can be used for an evaluation of the variable or a compounded expression.

A simple example leading to the mapping $(x, y) \mapsto x(y + \pi)$ using the types introduced so far is as follows:

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

Name	ViennaMath Function	Name	ViennaMath Function
Exponential	`exp()`	Modulus	`fabs()`
Sine	`sin()`	Square Root	`sqrt()`
Cosine	`cos()`	Natural Logarithm	`log()`
Tangent	`tan()`	Logarithm, Base 10	`log10()`

Table 2.1: Overview of unary functions defined in `ViennaMath`.

An evaluation of `f` at $(1, 2)$ can be accomplished by using evaluation overload the parenthesis operator and the `ViennaMath` helper function `make_vector()`, which conveniently creates a suitable vector for evaluation.

```
std::cout << f( make_vector(1,2) ) << std::endl;  //prints 5.1415
```

2.1.4 Unary Expression

Mappings of the form $x \mapsto \sin(x)$ are modeled by the `rt_unary_expr<InterfaceType>` class. Thus, they represent a unary function acting on a constant, a variable or an expression. An overview of the unary functions provided with `ViennaMath` is given in Tab. 2.1.

Function names in Tab. 2.1 are intentionally chosen such that they coincide with the standard functions for floating point types. When calling these functions with floating point types, compilation might fail due to ambiguity. In such case the namespace should be specified explicitly.

Typically, unary expressions are not instantiated explicitly by the library user. Instead, they are generated implicitly by one of the unary functions and then assigned to an object of type `expr` as in the following example:

```
variable x;
expr g = sin(2.0 * x);  // wraps a unary expression into 'g'
```

2.1.5 Binary Expression

Similar to unary expressions, binary expressions at runtime are mostly handled in the background only. They are created whenever one of the operator overloads for addition, subtraction, multiplication, or division is triggered. In particular, the argument `2.0 * x` to `sin()` in

```
expr g = sin(2.0 * x);
```

is a binary expression. Binary expressions are central for compile time evaluations in Sec. 2.2.

2.1.6 Expression Vector

For the cases where a vector-valued expression is required, a user can either instantiate a vector of `expr`, which allows for storing multiple scalar-valued function only, or use the `rt_vector_expr<InterfaceType>` class provided by `ViennaMath`. A convenience shortcut `vector_expr` is provided. The benefit of using the `vector_expr` class is that it provides the usual operator overloads directly:

```
variable x(0), y(1);
vector_expr vec(3); vec[0] = x; vec[1] = y; vec[2] = x + y;
vector_expr vec2 = x * vec + y * vec;
```

The dot-product of two vector-valued expressions is provided as well:

```
expr h = vec * vec2;
```

2.2 Types Evaluated at Compiletime

The runtime types discussed in the previous section enable a convenient handling of expressions. However, there are numerous runtime dispatches required when evaluating such runtime expressions, which are too costly in a high performance setting. The compiletime types discussed in this section avoid any additional runtime dispatches and their use thus result in faster code in general. This gain in performance comes at the price of a few additional restrictions: Since the expression is entirely encoded in the type, there is no equivalent to `expr` in order to assign an expression to a another object[1]. Furthermore, compilation times increase due to the additional work to be done for the compiler. Excessive use of compiletime evaluations and manipulations can even result in minutes to hours of compilation time, even though this is rarely encountered in practice. Another complication stems from the fact that no floating point template arguments are allowed, thus reducing any compiletime calculations to integer calculations. Fractional numbers can be emulated this way, but they cannot resolve all problems.

2.2.1 Constant

Since no floating point type is allowed as template argument, only integer values `val` are represented by the class `ct_constant<val>`. Operators are overloaded in the same way as for the runtime evaluation types in Sec. 2.1. One example of a compiletime calculation is given as follows:

```
ct_constant<2> c2;   //the constant '2'
ct_constant<5> c5;   //the constant '5'
std::cout << c2 + c5 << std::endl;  //prints '7' (computed at compiletime)
```

Note that `ct_constant<>` can in principle also be mixed with ordinary constants such as

```
std::cout << 2 + c5 << std::endl;  //prints '7'
```

However, depending on the optimization capabilities of the C++ compiler used, ordinary constants may or may not be used for compiletime computations, while the compiler is forced to do it in the introductory snippet.

[1]The new C++11 standard addresses this issue and provides the **auto** keyword for automatic type deduction. However, `ViennaMath` intentionally does not use any C++11 features yet.

A general guideline is to use `ct_constant<val>` for encoding an integer `val` already known at compile time rather than writing the value explicitly in code.

2.2.2 Variable

A mathematical variable for compiletime manipulations is represented by `ct_variable<id>`, where `id` refers to the coordinate entry in the evaluation vector. The meaning of `id` is identical to the constructor argument of a `variable` in the runtime case.

Operators are again overloaded as usual. For example, consider

```
ct_variable<0> x;
ct_variable<1> y;
std::cout << x * y << std::endl;
```

2.2.3 Unary Expression

The unary functions in Tab. 2.1 can also be called with compiletime types. The corresponding type for the compiletime representation is provided by `ct_unary_expr<E, OP>`, where `E` refers to the expression on which the unary function encoded by the tag `OP` acts. Unary operation tags start with `op_` and are defined in `viennamath/compiletime/unary_op_tags.hpp`. Their type name can be deduced from the function names in Tab. 2.1 by adding the prefix. Note that all unary functions are evaluated at runtime, because the underlying C-functions are called for evaluation. For example, the type `T` of the compiletime unary expression

```
ct_variable<0> x;
T t = sin(x);
```

is `ct_unary_expr< ct_variable<0>, op_sin<NumericT> >`, where `NumericT` is the floating point type used for the evaluation at runtime (typically `double`).

2.2.4 Binary Expression

The binary expression `ct_binary_expr<L, OP, R>` with left hand side expression `L`, operation tag `OP` and right hand side expression `R` are the main types for building more complex expressions. Currently, four binary operations are supported: addition (with tag `op_plus<NumericT>`), subtraction (`op_minus<NumericT>`), multiplication (`op_mult<NumericT>`), and division (`op_div<NumericT>`). Similar to unary expressions, binary expressions are seldomly set up by hand. Two examples of binary expressions are as follows:

```
ct_binary_expr< ct_variable<0>,    // x
                op_plus<double>,   // +
                ct_variable<1> >   // y

ct_binary_expr< ct_constant<1>,    // 1
                op_div<double>,    // /
                ct_variable<0> >   // x
```

Typical uses of binary expressions are within the manipulation of compiletime expressions in metafunctions. As an example, outputting the first term of a polynomial is considered:

```
ct_variable<0> x;
ct_variable<1> y;
print_first( x*y + x*x*y - y*y );
```

Only two versions of the `print_first` function are required. The first one recursively traverses the binary expression along the left hand side argument:

```
template <typename L, typename OP, typename R>
print_first(ct_binary_expr<L, OP, R> const & b)
{ print_first(b.lhs()); } //recursion along left hand side
```

The recursion terminates with a general implementation for printing the left-most entry:

```
template <typename T>
print_first(T const & t)
{ std::cout << t << std::endl; }
```

If a binary operation consists of one object for compiletime and one for runtime evaluation, the compiletime object is converted to a runtime object and then processed as usual in the runtime setting.

Chapter 3

Expression Manipulation

The basic description of the types in Chap. 2 allows for defining expressions and evaluating them. However, for most algorithms expressions need to be manipulated in one way or another, which is the topic of this chapter. Unless otherwise noted, all manipulations considerered in the following can be used for both compiletime and runtime expressions using the same interface.

Manipulation functionality resides in folder `viennamath/manipulation/`. The respective header files are not included automatically with `viennamath/expression.hpp` and need to be included as required.

3.1 Evaluation

All `ViennaMath` expressions can be evaluated to a floating point number using the parenthesis operator. Depending on the number of variables in the expression, either a scalar or a vector needs to be passed for evaluation.

Using `operator()`, however, is possibly not an option for a generic interface with non-ViennaMath types. For this reason, `viennamath::eval()` provides a generic evaluation interface. The first argument is the expression to be evaluated, and the second argument is the tuple with the values to be substituted for the variables. For example, the expression x^2 is defined and evaluated at $x = 2$ as follows:

```
ct_constant<2> c2;
ct_variable<0> x;
eval( x*x, 2.0 ); //      runtime evaluation
eval( x*x,  c2 ); // compiletime evaluation
```

Note that compiletime evaluation is only performed when both arguments are fully compiletime compatible. As soon as one part of the expression cannot be handled at compile time, a fallback to runtime evaluation is carried out. A hybrid evaluation in such cases is postponed to future releases of `ViennaMath`.

A vector of values needs to be passed as second argument, if a variable formally refers to any other than the first coordinate in a vector. Let us consider several use-cases of `eval()` consisting of various combinations of compiletime and runtime expressions:

```
ct_constant<3> c3;
ct_variable<1> y;
expr f = x*x + y;      // conversion to runtime expression
eval( f, 2.0 );        // runtime exception: insufficient values provided
eval( x*x + y, c2 );  // compilation error: insufficient values provided
eval( x*x + y,
      make_vector(2.0, 3.0) ); //    runtime evaluation
eval( x*x + y,
      make_vector(c2, c3) );  // compiletime evaluation
eval( 2.0, 0.0 );             // possible thanks to overloading
```

Since the runtime wrapper `expr` hides information from the compiler, an exception is thrown at runtime if insufficient values are provided for evaluation. For a full compiletime evaluation, insufficient parameters are already detected at an earlier stage. The helper function `make_vector()` generates a suitable vector type both for the runtime and the compiletime case. Instead of using `make_vector()`, a STL vector (`std::vector<double>`) or any compatible type can also be passed. Also note that the last line in the code snippet shows the benefit of using `eval()` instead of `operator()`: Scalars can also be 'evaluated' and are thus reinterpreted as constant functions.

3.2 Substitution

Formally, the evaluation of an expression can be seen as a substitution of the variables with values. A generalization is to replace arbitrary expressions with another expression in third expression. This is accomplished by the function `substitute()` defined in `viennamath/manipulation/substitute.hpp`:

```
expr f = (x + y) * (x - y);
substitute(x, y, f);          // returns (y + y) * (y - y) = 0
substitute(x, 1, f);          // returns (1 + y) * (1 - y)
substitute(x + y, x - y, f); // returns (x - y) * (x - y)
```

As with `eval`, substitutions are carried out at compiletime if all parameters are compiletime expressions.

3.3 Expansion

It is often desired to expand an expression given as a product of other terms. For example, instead of $2(x + y)$ one may want to have $2x - 2y$. Such a functionality is provided by the function `expand()` defined in `viennamath/manipulation/expand.hpp`:

```
expand( c2 * (x+y) );
```

where `c2` is a compiletime constant and `x`, `y` are compiletime variables. Note that `ViennaMath 1.0.0` does not support the expansion of runtime expressions yet.

`ViennaMath 1.0.0` supports compiletime expansion only.

3.4 Simplification

In the course of manipulating expressions, simple operations such as $x + 0$ or $x/1$ may appear. However, such terms constitute unnecessary overhead for later evaluations, thus it is desirable to have these operations dropped. Such a simplification of the expression can be achieved with the function `simplify()` defined in `viennamath/manipulation/simplify.hpp`:

```
simplify( x + 1.0 * y - 0.0 );  // returns x
simplify( x * (2.0 + 3.0) + (y * 0) / (x * 1) ); // returns 5x
```

`simplify()` is available both for runtime and compiletime manipulation.

3.5 Differentation

3.6 Integration

3.7 Extract Coefficient

`coefficient()` is in `ViennaMath 1.0.0` available for compiletime types only.

3.8 Drop Dependent Terms

`drop_dependent_terms()` is in `ViennaMath 1.0.0` available for compiletime types only.

Chapter 4

Advanced Features

4.1 Function Symbols

4.2 Integration Symbols

4.3 LaTeX Output

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/