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.

> 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

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

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);
```

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. The following unary functions are provided:

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()

Function names 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

2.2.1 Constant

2.2.2 Variable

2.2.3 Unary Expression

2.2.4 Binary Expression

2.2.5 Expression Vector

Chapter 3

Expression Manipulation

3.1 Evaluation

3.2 Substitution

3.3 Expansion

3.4 Simplification

3.5 Differentation

3.6 Integration

3.7 Extract Coefficient

3.8 Drop Dependent Terms

Chapter 4

Advanced Features

4.1 Function Symbols

4.2 Integration Symbols

4.3 LaTeX Output

Chapter 5

Benchmark Results

Give an idea of compile time performance.

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/