ViennaCL 1.0.4

User Manual

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

Copyright © 2010, Institute for Microelectronics, TU Vienna.

Main Contributors:

Florian Rudolf
Karl Rupp
Josef Weinbub

Current Maintainers:

Florian Rudolf
Karl Rupp
Josef Weinbub

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

The Vienna Computing Library (ViennaCL) is a scientific computing library written in C++. It allows simple, high-level access to the vast computing resources available on parallel architectures such as GPUs and multi-core CPUs. The primary focus is on common linear algebra operations (BLAS level 1 and 2) and the solution of large sparse systems of equations by means of iterative methods. In ViennaCL 1.0.x, the following iterative solvers are implemented (confer for example to the book of Y. Saad [1]):

- Conjugate Gradient (CG)
- Stabilized BiConjugate Gradient (BiCGStab)
- Generalized Minimum Residual (GMRES)

An optional ILU preconditioner can be used, which is in ViennaCL 1.0.4 precomputed and applied on a single CPU core and may thus not lead to overall performance gains over a purely CPU based implementation.

The solvers and preconditioners can also be used with different libraries due to a generic implementation. At present, it is possible to use the solvers and preconditioner directly with the ublas library from Boost [2].

Under the hood, ViennaCL uses OpenCL [3] for accessing and executing code on compute devices. Therefore, ViennaCL is not tailored to products from a particular vendor and can be used on many different platforms. At present, ViennaCL is known to work on modern GPUs from NVIDIA and ATI (see Tab. 1) as well as CPUs from AMD using the ATI Stream SDK. In principle, the ATI Stream SDK can also be used with ViennaCL on multi-core CPUs from Intel, even though this is not officially supported by ATI.

Double precision arithmetic on GPUs is only possible if it is provided by the GPU. There is no double precision emulation in ViennaCL.

Double precision arithmetic using the ATI Stream SDK is only exerimental. See Sec. 1.3.2 for details.

Compute Device	float	double
NVIDIA Geforce 86XX GT/GSO	ok	-
NVIDIA Geforce 88XX GTX/GTS	ok	-
NVIDIA Geforce 96XX GT/GSO	ok	-
NVIDIA Geforce 98XX GTX/GTS	ok	-
NVIDIA GT 230	ok	-
NVIDIA GT(S) 240	ok	-
NVIDIA GTS 250	ok	-
NVIDIA GTX 260	ok	ok
NVIDIA GTX 275	ok	ok
NVIDIA GTX 280	ok	ok
NVIDIA GTX 285	ok	ok
NVIDIA GTX 465	ok	ok
NVIDIA GTX 470	ok	ok
NVIDIA GTX 480	ok	ok
NVIDIA Quadro FX 46XX	ok	-
NVIDIA Quadro FX 48XX	ok	ok
NVIDIA Quadro FX 56XX	ok	-
NVIDIA Quadro FX 58XX	ok	ok
NVIDIA Tesla 870	ok	-
NVIDIA Tesla C10XX	ok	ok
NVIDIA Tesla C20XX	ok	ok
ATI Radeon HD 45XX	ok	-
ATI Radeon HD 46XX	ok	-
ATI Radeon HD 47XX	ok	-
ATI Radeon HD 48XX	ok	experimental
ATI Radeon HD 54XX	ok	-
ATI Radeon HD 55XX	ok	-
ATI Radeon HD 56XX	ok	-
ATI Radeon HD 57XX	ok	-
ATI Radeon HD 58XX	ok	experimental
ATI Radeon HD 59XX	ok	experimental
ATI FireStream V92XX	ok	experimental
ATI FirePro V78XX	ok	experimental
ATI FirePro V87XX	ok	experimental
ATI FirePro V88XX	ok	experimental

Table 1: Available arithmetics in `ViennaCL` provided by selected GPUs. At the release of `ViennaCL 1.0.4`, GPUs from ATI do not comply to `OpenCL` standard for double precision extensions and can thus only be used in single precision arithmetics in `ViennaCL` per default. Once the driver of these GPUs complies to the double precision extension standard of `OpenCL`, they can be used with `ViennaCL` immediately. An experimental support for ATI GPUs has to be enabled explicitly in `ViennaCL`, see Sec. 1.3.2.

Chapter 1

Installation

This chapter shows how ViennaCL 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

viennacl-support@lists.sourceforge.net

1.1 Dependencies

ViennaCL uses the CMake build system for multi-platform support. Thus, before you proceed with the installation of ViennaCL, make sure you have a recent version of CMake installed.

- A recent C++ compiler (e.g. GCC version 4.2.x or above and Visual C++ 2008 are known to work)
- OpenCL [3, 4] for accessing compute devices (GPUs); see Section 1.3 for details. (optional, since iterative solvers can also be used with e.g. ublas)
- CMake [5] as build system (optional, but highly recommended for building the examples)
- ublas (shipped with Boost [2]) provides the same interface as ViennaCL and allows to switch between CPU and GPU seamlessly, see the tutorials.

1.2 Generic Installation of ViennaCL

Since ViennaCL is a header-only library, it is sufficient to copy the folder viennacl/ 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`. The include and library directories of your `OpenCL` SDK should also be added there.

If multiple `OpenCL` libraries are available on the host system, one has to ensure that the intended one is used.

1.3 Get the `OpenCL` Library

The development of `OpenCL` applications requires a corresponding library (e.g. `libOpenCL.so` under Unix based systems) and a suitable driver if used on GPUs. This section describes how these can be acquired.

Note, that for Mac OS X systems there is no need to install an `OpenCL` capable driver and the corresponding library. The `OpenCL` library is already present if a suitable graphics card is present. The setup of `ViennaCL` on Mac OS X is discussed in Section 1.4.2.

1.3.1 NVIDIA Driver

NVIDIA provides the `OpenCL` library with the GPU driver. Therefore, if a NVIDIA driver is present on the system, the library is too. However, not all of the released drivers contain the `OpenCL` library. A driver which is known to support `OpenCL`, and hence providing the required library, is 195.36.24.

1.3.2 ATI Stream SDK

ATI provides the `OpenCL` library with the Stream SDK [6]. At the release of `ViennaCL 1.0.4`, the latest Stream SDK was 2.1. If used with ATI GPUs, the ATI GPU drivers of version at least 10.4[1] are required. If `ViennaCL` is to be run on multi-core CPUs, no additional GPU driver is required. The installation notes of the SDK provides guidance throughout the installation process [7].

If the SDK is installed in a non-system wide location, the include and library paths have to be specified in the `CMakeLists.txt` files. An example of how to set these paths is provided in the file `CMakeLists.txt` around line 18. Be sure to add the `OpenCL` library path to the `LD_LIBRARY_PATH` environment variable. Otherwise, linker errors will occur as the required library cannot be found.

It is important to note that the ATI Stream SDK does not provide full double precision support [8] on CPUs and GPUs, so it is only experimentally available in `ViennaCL`. This experimental mode is disabled by default, but can be enabled for either CPUs or GPUs by defining one of the preprocessor constants

[1] Current ATI drivers may not work with current kernel version. The presented tests are based on the 2.6.33 kernel.

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

```
// for CPUs:
#define VIENNACL_EXPERIMENTAL_DOUBLE_PRECISION_WITH_STREAM_SDK_ON_CPU
// for GPUs:
#define VIENNACL_EXPERIMENTAL_DOUBLE_PRECISION_WITH_STREAM_SDK_ON_GPU
```

prior to any inclusion of ViennaCL header files.

Some compute kernels may not work as expected in the experimental double precision mode in the ATI Stream SDK. Moreover, the functions norm_1, norm_2, norm_inf and index_norm_inf are not available on GPUs in double precision using ATI Stream SDK.

1.4 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.

1.4.1 Linux

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

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

Execute

```
$> cmake ..
```

to obtain a Makefile. Executing

```
$> make
```

builds the examples. If some of the dependencies in Tab. 1.1 are not fulfilled, you can build each example separately:

```
$> make tut1              #builds tutorial 1
$> make vectorbench       #builds vector benchmarks
```

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

1.4.2 Mac OS X

The tools mentioned in Section 1.1 are available on macintosh platforms too. For the GCC compiler the Xcode [9] package has to be installed. To install CMake and Boost external portation tools have to be used, for example, Fink [10], DarwinPorts [11] or MacPorts [12]. Such portation tools provide the aforementioned packages, CMake and Boost, for macintosh platforms.

If the CMake build system has problems detecting your Boost libraries, determine the location of your Boost folder. Open the CMakeLists.txt file in the root directory of ViennaCL and add your Boost path after the following entry: `IF($CMAKE_SYSTEM_NAME MATCHES "Darwin")`

The build process of ViennaCL is similar to Linux.

1.4.3 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 ViennaCL 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 ViennaCL 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 ViennaCL, otherwise the sample data files cannot be found.

Chapter 2

Basic Types

This chapter provides a brief overview of the basic interfaces and usage of the provided data types. The term *GPU* refers here and in the following to both GPUs and multi-core CPUs accessed via `OpenCL` and managed by `ViennaCL`. Operations on the various types are explained in Chap. 3. For full details, refer to the reference pages in the folder `doc/doxygen`.

2.1 Scalar Type

The scalar type `scalar<T>` with template parameter T denoting the underlying CPU scalar type (float and double, if supported - see Tab. 1) represents a single scalar value on the GPU. `scalar<T>` is designed to behave much like a scalar type on the CPU, but library users have to keep in mind that every operation on `scalar<T>` requires to launch the appropriate compute kernel on the GPU and is thus much slower then the CPU equivalent.

Be aware that operations between objects of type `scalar<T>` (e.g. additions. comparisons) have large overhead. For every operation, a separate compute kernel launch is required.

2.1.1 Example Usage

The scalar type of `ViennaCL` can be used just like the built-in types, as the following snippet shows:

```
float cpu_float = 42.0f;
double cpu_double = 13.7603;
ViennaCL::scalar<float>  gpu_float(3.1415f);
ViennaCL::scalar<double> gpu_double = 2.71828;

//conversions and t
cpu_float = gpu_float;
gpu_float = cpu_double;  //automatic transfer and conversion

cpu_float = gpu_float * 2.0f;
cpu_double = gpu_float - cpu_float;
```

Interface	Comment
`v.handle()`	The GPU handle

Table 2.1: Interface of `vector<T>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

Mixing built-in types with the `ViennaCL` scalar is usually not a problem. Nevertheless, since every operation requires `OpenCL` calls, such arithmetics should be used sparsingly.

In the present version of `ViennaCL`, it is not possible to assign a `scalar<float>` to a `scalar<double>` directly.

2.1.2 Members

Apart from suitably overloaded operators that mimic the behavior of the respective CPU counterparts, only a single public member function `handle()` is available, cf. Tab. 2.1.

2.2 Vector Type

The main vector type in `ViennaCL` is `vector<T, alignment>`, representing a chunk of memory on the compute device. `T` is the underlying scalar type (either `float` or `double` if supported, cf. Tab. 1, complex types are not supported in `ViennaCL 1.0.4`) and the optional argument `alignment` denotes the memory the vector is aligned to (in multiples of `sizeof(T)`). For example, a vector with a size of 55 entries and an alignment of 16 will reside in a block of memory equal to 64 entries. Memory alignment is fully transparent, so from the end-user's point of view, `alignment` allows to tune `ViennaCL` for maximum speed on the available compute device.

At construction, `vector<T, alignment>` is initialized to have the supplied length, but the memory is not initialized to zero. Another difference to CPU implementations is that accessing single vector elements is very costly, because every time an element is accessed, it has to be transferred from the CPU to the compute device or vice versa.

2.2.1 Example Usage

The following code snippet shows the typical use of the vector type provided by `ViennaCL`. The overloaded function `copy()` function, which is used as in the STL, should be used for the initialization of vector entries:

```
std::vector<ScalarType>      stl_vec(10);
viennacl::vector<ScalarType> vcl_vec(10);

//fill the STL vector:
for (unsigned int i=0; i<vector_size; ++i)
  stl_vec[i] = i;

//copy content to GPU vector (recommended initialization)
copy(stl_vec.begin(), stl_vec.end(), vcl_vec.begin());
```

Interface	Comment
`CTOR(n)`	Constructor with number of entries
`v(i)`	Access to the i-th element of v (slow!)
`v[i]`	Access to the i-th element of v (slow!)
`v.clear()`	Initialize v with zeros
`v.resize(n, bool preserve)`	Resize v to length n. Preserves old values if bool is true.
`v.begin()`	Iterator to the begin of the matrix
`v.end()`	Iterator to the end of the matrix
`v.size()`	Length of the vector
`v.swap(v2)`	Swap the content of v with v2
`v.internal_size()`	Returns the number of entries allocated on the GPU (taking alignment into account)
`v.empty()`	Shorthand notation for `v.size() == 0`
`v.clear()`	Sets all entries in v to zero
`v.handle()`	Returns the GPU handle (needed for custom kernels, see Chap. 5)

Table 2.2: Interface of `vector<T>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

```
//manipulate GPU vector here

//copy content from GPU vector back to STL vector
copy(vcl_vec.begin(), vcl_vec.end(), stl_vec.begin());
```

The function copy() does not assume that the values of the supplied CPU object are located in a linear memory sequence. If this is the case, the function `fast_copy` provides better performance.

Once the vectors are set up on the GPU, they can be used like objects on the CPU (cf. Chap. 3):

```
// let vcl_vec1 and vcl_vec2 denote two vector on the GPU
vcl_vec1 *= 2.0;
vcl_vec2 += vcl_vec1;
vcl_vec1 = vcl_vec1 - 3.0 * vcl_vec2;
```

2.2.2 Members

At construction, `vector<T, alignment>` is initialized to have the supplied length, but memory is not initialized. If initialization is desired, the memory can be initialized with zero values using the member function `clear()`. See Tab. 2.2 for other member functions.

Accessing single elements of a vector using operator() or operator[] is very slow! Use with care!

One important difference to CPU implementations is that the bracket operator (as well as

the parenthesis operator) is very slow, because for each access on the GPU a data transfer has to be initiated. The overhead of this transfer is orders of magnitude. For example:

```
// fill a vector on CPU
for (long i=0; i<cpu_vector.size(); ++i)
    cpu_vector(i) = 1e-3f;

// fill a vector on GPU - VERY SLOW!!
for (long i=0; i<gpu_vector.size(); ++i)
    gpu_vector(i) = 1e-3f;
```

The difference in execution speed is typically several orders of magnitude, therefore direct vector element access should be used only if a very small number of entries is accessed in this way. A much faster initialization is as follows:

```
// fill a vector on CPU
for (long i=0; i<cpu_vector.size(); ++i)
    cpu_vector(i) = 1e-3f;

// fill a vector on GPU with data from CPU - fast version
copy(cpu_vector, gpu_vector);
```

In this way, setup costs for the CPU vector and the GPU vector are comparable.

2.3 Dense Matrix Type

`matrix<T, F, alignment>` represents a dense matrix with interface listed in Tab. 2.3. The second optional template argument `F` specifies the storage layout and defaults to `row_major`, which is the only one provided in `ViennaCL 1.0.x`. The third template argument `alignment` denotes an alignemnt for the number of rows (cf. `alignment` for the `vector` type).

2.3.1 Example Usage

The use of `matrix<T, F>` is similar to that of the counterpart in `ublas`. The operators are overloaded similarly.

```
//set up a 3 by 5 matrix:
viennacl::matrix<float>  gpu_matrix(3, 5);

//fill it up:
gpu_matrix(0,2) = 1.0;
gpu_matrix(1,2) = -1.5;
gpu_matrix(3,0) = 4.2;
```

Accessing single elements of a matrix using `operator()` is very slow! Use with care!

A much better way is to initialize a dense matrix using the provided `copy()` function:

```
//copy content from CPU matrix to GPU matrix
copy(cpu_matrix, gpu_matrix);
```

Interface	Comment
`CTOR(nrows, ncols)`	Constructor with number of rows and columns
`mat(i,j)`	Access to the element in the i-th row and the j-th column of mat
`mat.resize(m, n, bool preserve)`	Resize mat to m rows and n columns. Currently, the boolean flag is ignored and entries always discarded.
`mat.size1()`	Number of rows in mat
`mat.internal_size1()`	Internal number of rows in mat
`mat.size2()`	Number of columns in mat
`mat.internal_size2()`	Internal number of columns in mat
`mat.clear()`	Sets all entries in v to zero
`mat.handle()`	Returns the GPU handle (needed for custom kernels, see Chap. 5)

Table 2.3: Interface of the dense matrix type `matrix<T, F>` in `ViennaCL`. Constructors, Destructors and operator overloads for BLAS are not listed.

```
//copy content from GPU matrix to CPU matrix
copy(gpu_matrix, cpu_matrix);
```

The type requirement on the `cpu_matrix` is that `operator()` can be used for accessing entries, that a member function `size1()` returns the number of rows and that `size2()` returns the number of columns.

2.3.2 Members

The members are listed in Tab. 2.3. The usual operator overloads are not listed explicitly

2.4 Sparse Matrix Types

There are two different sparse matrix types provided in `ViennaCL`, `compressed_matrix` and `coordinate_matrix`.

In `ViennaCL 1.0.4`, the use of `compressed_matrix` is encouraged over `coordinate_matrix`

2.4.1 Compressed Matrix

`compressed_matrix<T, alignment>`, represents a sparse matrix using a compressed sparse row scheme. Again, `T` is the floating point type. `alignment` is the alignment and defaults to `1` at present. In general, sparse matrices should be set up on the CPU and then be pushed to the compute device using `copy()`, because the management of map based sparse matrices is inefficient on most compute devices such as GPUs.

Interface	Comment
`CTOR(nrows, ncols)`	Constructor with number of rows and columns
`mat.set()`	Initialize mat with zeros
`mat.reserve(num)`	Reserve memory for up to `num` nonzero entries
`mat.size1()`	Number of rows in mat
`mat.size2()`	Number of columns in mat
`mat.nnz()`	Number of nonzeroes in mat
`mat.resize(m, n, bool preserve)`	Resize mat to m rows and n columns. Currently, the boolean flag is ignored and entries always discarded.
`mat.handle1()`	Returns the GPU handle holding the row indices (needed for custom kernels, see Chap. 5)
`mat.handle2()`	Returns the GPU handle holding the column indices (needed for custom kernels, see Chap. 5)
`mat.handle()`	Returns the GPU handle holding the entries (needed for custom kernels, see Chap. 5)

Table 2.4: Interface of the sparse matrix type `compressed_matrix<T, F>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

2.4.1.1 Example Usage

The use of `compressed_matrix<T, alignment>` is similar to that of the counterpart in `ublas`. The operators are overloaded similarly. There is a direct interfacing with the standard implementation using a vector of maps from the STL:

```
//set up a sparse 3 by 5 matrix on the CPU:
std::vector< std::map< unsigned int, float> > cpu_sparse_matrix(3);

//fill it up:
cpu_sparse_matrix[0][2] = 1.0;
cpu_sparse_matrix[1][2] = -1.5;
cpu_sparse_matrix[3][0] = 4.2;

//set up a sparse GPU matrix:
viennacl::compressed_matrix<float>  gpu_sparse_matrix(3, 5);

//copy to GPU:
copy(cpu_sparse_matrix, gpu_sparse_matrix);

//copy back to CPU:
copy(gpu_sparse_matrix, cpu_sparse_matrix);
```

The `copy()` functions can also be used with a generic sparse matrix data type fulfilling the following requirements:

- The `const_iterator1` type is provided for iteration along increasing row index
- The `const_iterator2` type is provided for iteration along increasing column index

- `.begin1()` returns an iterator pointing to the element with indices $(0, 0)$.
- `.end1()` returns an iterator pointing to the end of the first column
- When copying to the cpu type: Write operation via `operator()`
- When copying to the cpu type: `resize(m,n,preserve)` member (cf. Tab. 2.4)

The iterator returned from the cpu sparse matrix type via `begin1()` has to fulfill the following requirements:

- `.begin()` returns an column iterator pointing to the first nonzero element in the particular row.
- `.end()` returns an iterator pointing to the end of the row
- Increment and dereference

For the sparse matrix types in `ublas`, these requirements are all fulfilled.

2.4.1.2 Members

The interface is described in Tab. 2.4.

2.4.2 Coordinate Matrix

In the second sparse matrix type, `coordinate_matrix<T, alignment>`, entries are stored as triplets `(i,j,val)`, where `i` is the row index, `j` is the column index and `val` is the entry. Again, `T` is the floating point type. The optional `alignment` defaults to `1` at present. In general, sparse matrices should be set up on the CPU and then be pushed to the compute device using `copy()`, because the management of map based sparse matrices is inefficient on most compute devices such as GPUs.

2.4.2.1 Example Usage

The use of `coordinate_matrix<T, alignment>` is similar to that of the first sparse matrix type `compressed_matrix<T, alignment>`, thus we refer to Sec. 2.4.1.1

2.4.2.2 Members

The interface is described in Tab. 2.5.

Interface	Comment
`CTOR(nrows, ncols)`	Constructor with number of rows and columns
`mat.reserve(num)`	Reserve memory for up to `num` nonzero entries
`mat.size1()`	Number of rows in mat
`mat.size2()`	Number of columns in mat
`mat.nnz()`	Number of nonzeroes in mat
`mat.resize(m, n, bool preserve)`	Resize mat to m rows and n columns. Currently, the boolean flag is ignored and entries always discarded.
`mat.resize(m, n)`	Resize mat to m rows and n columns. Does not preserve old values.
`mat.handle12()`	Returns the GPU handle holding the row and column indices (needed for custom kernels, see Chap. 5)
`mat.handle()`	Returns the GPU handle holding the entries (needed for custom kernels, see Chap. 5)

Table 2.5: Interface of the sparse matrix type `coordinate_matrix<T, A>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

Chapter 3

Basic Operations

The basic types have been introduced in the previous chapter, so the basic operations can now be described.

3.1 Vector-Vector Operations (BLAS Level 1)

ViennaCL provides all vector-vector operations defined at level 1 of BLAS. Tab. 3.1 shows how these operations can be carried out in ViennaCL. The function interface is compatible with ublas, thus allowing quick code migration for ublas users.

For full details on level 1 functions, refer to the reference documentation located in doc/doxygen/

The plane rotation function may not work as expected on some GPUs. If this is the case, library users are advised to use separate result vectors and write the operations explicitly using the other BLAS level 1 functions.

3.2 Matrix-Vector Operations (BLAS Level 2)

The interface for level 2 BLAS functions in ViennaCL is similar to that of ublas and shown in Tab. 3.2.

For full details on level 2 functions, refer to the reference documentation located in doc/doxygen/

ViennaCL is not only able to solve triangular matrices, as requested by BLAS, it provides several iterative solvers for the solution of large systems of equations. See Section 4.2 for more details on linear solvers.

Verbal	Mathematics	ViennaCL
swap	$x \leftrightarrow y$	`swap(x,y);`
stretch	$x \leftarrow \alpha x$	`x *= alpha;`
assignment	$y \leftarrow x$	`y = x;`
multiply add	$y \leftarrow \alpha x + y$	`y += alpha * x;`
multiply subtract	$y \leftarrow \alpha x - y$	`y -= alpha * x;`
inner dot product	$\alpha \leftarrow x^{\mathrm{T}} y$	`inner_prod(x,y);`
L^1 norm	$\alpha \leftarrow \|x\|_1$	`alpha = norm_1(x);`
L^2 norm	$\alpha \leftarrow \|x\|_2$	`alpha = norm_2(x);`
L^∞ norm	$\alpha \leftarrow \|x\|_\infty$	`alpha = norm_inf(x);`
L^∞ norm index	$i \leftarrow \max_i \|x_i\|$	`i = index_norm_inf(x);`
plane rotation	$(x, y) \leftarrow (\alpha x + \beta y, -\beta x + \alpha y)$	`plane_rotation(alpha, beta, x, y);`

Table 3.1: BLAS level 1 routines mapped to `ViennaCL`. Note that the free functions reside in namespace `viennacl::linalg`

3.3 Matrix-Matrix Operations (BLAS Level 3)

`ViennaCL 1.0.x` does not provide matrix-matrix operations as defined at level 3 of BLAS. This is subject to change in future releases.

Verbal	Mathematics	ViennaCL
matrix vector product	$y \leftarrow Ax$	`y = prod(A, x);`
matrix vector product	$y \leftarrow A^{\mathrm{T}}x$	`y = prod(trans(A), x);`
inplace mv product	$x \leftarrow Ax$	`x = prod(A, x);`
inplace mv product	$x \leftarrow A^{\mathrm{T}}x$	`x = prod(trans(A), x);`
scaled product add	$y \leftarrow \alpha Ax + \beta y$	`y = alpha * prod(A, x) + beta * y`
scaled product add	$y \leftarrow \alpha A^{\mathrm{T}}x + \beta y$	`y = alpha * prod(trans(A), x) + beta * y`
tri. matrix solve	$y \leftarrow A^{-1}x$	`y = solve(A, x, tag);`
tri. matrix solve	$y \leftarrow A^{\mathrm{T}^{-1}}x$	`y = solve(trans(A), x, tag);`
inplace solve	$x \leftarrow A^{-1}x$	`inplace_solve(A, x, tag);`
inplace solve	$x \leftarrow A^{\mathrm{T}^{-1}}x$	`inplace_solve(trans(A), x, tag);`
rank 1 update	$A \leftarrow \alpha xy^{\mathrm{T}} + A$	`A += alpha * outer_prod(x,y);`
symm. rank 1 update	$A \leftarrow \alpha xx^{\mathrm{T}} + A$	`A += alpha * outer_prod(x,x);`
rank 2 update	$A \leftarrow \alpha(xy^{\mathrm{T}} + yx^{\mathrm{T}}) + A$	`A += alpha * outer_prod(x,y);` `A += alpha * outer_prod(y,x);`

Table 3.2: BLAS level 2 routines mapped to `ViennaCL`. Note that the free functions reside in namespace `viennacl::linalg`

Chapter 4

Algorithms

This chapter gives an overview over the available algorithms in ViennaCL. The focus of ViennaCL is on iterative solvers, for which ViennaCL provides a generic implementation that allows the use of the same code on the CPU and on the GPU.

4.1 Direct Solvers

ViennaCL 1.0.4 provides triangular solvers and LU factorization without pivoting for the solution of dense linear systems. The interface is similar to that of ublas

```
viennacl::matrix<float>  vcl_matrix;
viennacl::vector<float>  vcl_rhs;
viennacl::vector<float>  vcl_result;

/* Set up matrix and vectors here */

//solution of an upper triangular system:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::upper_tag());
//solution of an lower triangular system:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::lower_tag());

//solution of a full system right into the load vector vcl_rhs:
viennacl::linalg::lu_factorize(vcl_matrix);
viennacl::linalg::lu_substitute(vcl_matrix, vcl_rhs);
```

In ViennaCL 1.0.x there is no pivoting included in the LU factorization process, hence the computation may break down or yield results with poor accuracy. However, for certain classes of matrices (like diagonal dominant matrices) good results can be obtained without pivoting.

4.2 Iterative Solvers

ViennaCL provides different iterative solvers for various classes of matrices, listed in Tab. 4.1. Unlike direct solvers, the convergence of iterative solvers relies on certain properties of the system matrix. Keep in mind that an iterative solver may fail to converge, especially if the matrix is ill conditioned or a wrong solver is chosen.

For full details on linear solver calls, refer to the reference documentation located in doc/doxygen/ and to the tutorials

The iterative solvers can directly be used for ublas objects!

In ViennaCL 1.0.4, GMRES using ATI GPUs is known to yield wrong results! However, GMRES works smoothly if using the ATI Stream SDK on CPUs. At present it is not clear whether these problems on ATI GPUs are caused by ViennaCL or the ATI Stream SDK.

```
viennacl::compressed_matrix<float>  vcl_matrix;
viennacl::vector<float>  vcl_rhs;
viennacl::vector<float>  vcl_result;

/* Set up matrix and vectors here */

//solution using conjugate gradient solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::cg_tag());

//solution using BiCGStab solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::bicgstab_tag());

//solution using GMRES solver:
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::gmres_tag());
```

4.3 Preconditioners

ViennaCL ships with a generic implementation of an incomplete LU factorization preconditioner with threshold (ILUT). The incomplete factorization is computed on a single CPU core due to its sequential nature, so one must not expect large performance gains if most time is spent on preconditioning. Other preconditioners more suitable for GPUs are in preparation.

Method	Matrix class	ViennaCL
Conjugate Gradient (CG)	symmetric positive definite	`y = solve(A, x, cg_tag());`
Stabilized Bi-CG (BiCGStab)	non-symmetric	`y = solve(A, x, bicgstab_tag());`
Generalized Minimum Residual (GMRES)	general	`y = solve(A, x, gmres_tag());`

Table 4.1: Linear solver routines in `ViennaCL` for the computation of y in the expression $Ay = x$ with given A, x.

The preconditioner also works for `ublas` types!

```
using viennacl::linalg::ilut_precond;
using viennacl::compressed_matrix;

compressed_matrix<float>  vcl_matrix;
viennacl::vector<float>  vcl_rhs;
viennacl::vector<float>  vcl_result;

/* Set up matrix and vectors here */

//compute preconditioner:
ilut_precond< compressed_matrix<float> > vcl_ilut(vcl_matrix,
                                                  viennacl::linalg::
                                                     ilut_tag());

//solve (e.g. using conjugate gradient solver)
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::cg_tag(),
                                     vcl_ilut);   //preconditioner provided
                                         here
```

Chapter 5

Custom Compute Kernels

For custom algorithms the built-in functionality of ViennaCL may not be sufficient or not fast enough. In such cases it can be desireable to write a custom OpenCL compute kernel, which is explained in this chapter. The following steps are necessary and explained one after another:

- Write the OpenCL source code
- Compile the compute kernel
- Launch the kernel

A tutorial on this topic can be found at examples/tutorial/tut5.cpp.

5.1 Setting up the Source Code

The OpenCL source code has to be provided as a string. You can either write the source code directly into a string in you C++ files, or you can read the OpenCL source from a file. For demonstration purposes, we write the source directly as a string constant:

```
const char * my_compute_kernel =
"__kernel void elementwise_prod(\n"
"          __global const float * vec1,\n"
"          __global const float * vec2, \n"
"          __global float * result,\n"
"          unsigned int size) \n"
"{ \n"
"  for (unsigned int i = get_global_id(0); i < size; i += get_global_size
    (0))\n"
"    result[i] = vec1[i] * vec2[i];\n"
"};\n";
```

The kernel takes three vector arguments vec1, vec2 and result and the vector length variable size. The compute kernel computes the entry-wise product of the vectors vec1 and vec2 and writes the result to the vector vec3. For more detailed explanation of the OpenCL source code, please refer to the specification available at the Khronos group webpage [3].

5.2 Compilation of the Source Code

The source code in the string constant `my_compute_kernel` has to be compiled to an `OpenCL` program. An `OpenCL` program is a compilation unit and may contain several different compute kernels, so one could also include another kernel function `inplace_elementwise_prod` which writes the result directly to one of the two operands `vec1` or `vec2` in the same program.

```
viennacl::ocl::program my_prog;         //create the object
my_prog.create(my_compute_kernel);      //compile the source
```

The next step is to extract the kernel `my_compute_kernel` from the compiled program:

```
viennacl::ocl::kernel my_kernel;                  //create kernel object
my_kernel.prepareInit("elementwise_prod", my_prog);  //extract kernel
```

Now, the kernel is set up to use the function `elementwise\_prod` compiled into the program `my\_prog`.

The kernel extraction actually occurs at the time the first kernel argument is set. This allows to avoid any kernel extraction overhead for unused kernels.

Due to the just-in-time extraction, kernel objects cannot be copied or assigned to other kernel objects.

5.3 Launch the Kernel

To launch the kernel, the kernel arguments have to be set. We assume that three `ViennaCL` vectors `vec1`, `vec2` and `result` have already been set up, the vector length is available in the variable `vector_size`:

```
unsigned int pos = 0;
my_kernel.setArgument(pos++, vec1.handle());
my_kernel.setArgument(pos++, vec2.handle());
my_kernel.setArgument(pos++, result.handle());
my_kernel.setArgument(pos++, vector_size);
```

The `OpenCL` handles for the vectors are obtained by the member function `handle()`, the vector length argument can be passed directly. The kernel is now ready for launch:

```
my_kernel.start1D(vector_size,    //number of global threads
                  vector_size);   //number of threads per work group
```

The first argument specifies the number of global threads. Assuming small vectors (less than 100 entries), we can assign one vector per entry. The second argument specifies the number of threads per work group and can be omitted, in which case the `OpenCL` driver tries to find suitable work group sizes automatically. Please consult the `OpenCL` specification [3] for more details on the execution model.

Chapter 6

Benchmark Results

We have compared the performance gain of ViennaCL with standard CPU implementations using a single core. The code used for the benchmarks can be found in the folder examples/benchmark/ within the source-release of ViennaCL. Results are grouped by computational complexity and can be found in the subsequent sections.

CPU RAM OS	AMD Phenom II X4-965 8 GB Funtoo Linux 64 bit
Kernel for AMD cards: AMD driver version:	2.6.33 10.4
kernel for Nvidia cards: Nvidia driver version:	2.6.34 195.36.24
ViennaCL version	1.0.0

Compute kernels are not fully optimized yet, results are likely to improve considerably in future releases of ViennaCL

Due to only partial support of double precision by GPUs from ATI, the benchmarks do not include double precision arithmetics there, cf. Tab. 1.

6.1 Vector Operations

Benchmarks for the addition of two vectors and the computation of inner products are shown in Tab. 6.1.

6.2 Matrix-Vector Multiplication

We have compared execution times of the operation

$$\mathbf{y} = \mathbf{A}\mathbf{x}\,, \tag{6.1}$$

Compute Device	add, float	add, double	prod, float	prod, double
CPU	0.174	0.347	0.408	0.430
Nvidia GTX 260	0.087	0.089	0.044	0.072
Nvidia GTX 470	0.042	0.133	0.050	0.053
ATI Radeon 5850	0.026	-	0.105	-

Table 6.1: Execution times (seconds) for vector addition and inner products.

Compute Device	float	double
CPU	0.0333	0.0352
Nvidia GTX 260	0.0028	0.0043
Nvidia GTX 470	0.0024	0.0041
ATI Radeon 5850	0.0032	-

Table 6.2: Execution times (seconds) for sparse matrix-vector multiplication using `compressed_matrix`.

where A is a sparse matrix (ten entries per column on average). The results in Tab. 6.2 shows that by the use of `ViennaCL` and a mid-range GPU, performance gains of up to one order of magnitude can be obtained.

6.3 Iterative Solver Performance

The solution of a system of linear equations is encountered in many simulators. It is often seen as a black-box: System matrix and right hand side vector in, solution out. Thus, this black-box process allows to easily exchange existing solvers on the CPU with a GPU variant provided by `ViennaCL`. Tab. 6.3 shows that the performance gain of GPU implementations can be significant. For applications where most time is spent on the solution of the linear systems, the use of `ViennaCL` can reduce the total execution time by about a factor of five.

Compute Device	CG, float	CG, double	GMRES, float	GMRES, double
CPU	0.407	0.450	4.84	7.58
Nvidia GTX 260	0.067	0.092	4.27	5.08
Nvidia GTX 470	0.063	0.087	3.63	4.68
ATI Radeon 5850	0.233	-	22.7	-

Table 6.3: Execution times (seconds) for ten iterations of CG and GMRES without preconditioner. Results for BiCGStab are similar to that of CG.

Chapter 7

Design Decisions

During the implementation of ViennaCL, several design decisions have been necessary, which are often a trade-off among various advantages and disadvantages. In the following, we discuss several design decisions and their alternatives.

7.1 Transfer CPU-GPU-CPU for Scalars

The ViennaCL scalar type scalar<> essentially behaves like a CPU scalar in order to make any access to GPU ressources as simple as possible, for example

```
float cpu_float = 1.0f;
viennacl::linalg::scalar<float> gpu_float = cpu_float;

gpu_float = gpu_float * gpu_float;
gpu_float -= cpu_float;
cpu_float = gpu_float;
```

As an alternative, the user could have been required to use copy as for the vector and matrix classes, but this would unnecessarily complicate many commonly used operations like

```
if (norm_2(gpu_vector) < 1e-10) { ... }
```

or

```
gpu_vector[0] = 2.0f;
```

where one of the operands resides on the CPU and the other on the GPU. Initialization of a separate type followed by a call to copy is certainly not desired for the above examples.

However, one should use scalar<> with care, because the overhead for transfers from CPU to GPU and vice versa is very large for the simple scalar<> type.

Use scalar<> with care, it is much slower than built-in types on the CPU!

7.2 Transfer CPU-GPU-CPU for Vectors

The present way of data transfer for vectors and matrices from CPU to GPU to CPU is to use the provided `copy` function, which is similar to its counterpart in the Standard Template Library (STL):

```
std::vector<float> cpu_vector(10);
ViennaCL::LinAlg::vector<float> gpu_vector(10);

/* fill cpu_vector here */

//transfer values to gpu:
copy(cpu_vector.begin(), cpu_vector.end(), gpu_vector.begin());

/* compute something on GPU here */

//transfer back to cpu:
copy(gpu_vector.begin(), gpu_vector.end(), cpu_vector.begin());
```

A first alternative approach would have been to to overload the assignment operator like this:

```
//transfer values to gpu:
gpu_vector = cpu_vector;

/* compute something on GPU here */

//transfer back to cpu:
cpu_vector = gpu_vector;
```

The first overload can be directly applied to the `vector`-class provided by `ViennaCL`. However, the question of accessing data in the `cpu_vector` object arises. For `std::vector` and C arrays, the bracket operator can be used, but the parenthesis operator cannot. However, other vector types may not provide a bracket operator. Using STL iterators is thus the more reliable variant.

The transfer from GPU to CPU would require to overload the assignment operator for the CPU class, which cannot be done by `ViennaCL`. Thus, the only possibility within `ViennaCL` is to provide conversion operators. Since many different libraries could be used in principle, the only possibility is to provide conversion of the form

```
template <typename T>
operator T() {/* implementation here */}
```

for the types in `ViennaCL`. However, this would allow even totally meaningless conversions, e.g. from a GPU vector to a CPU boolean and may result in obscure unexpected behavior.

Moreover, with the use of `copy` functions it is much clearer, at which point in the source code large amounts of data are transferred between CPU and GPU.

7.3 Solver Interface

We decided to provide an interface compatible to ublas for dense matrix operations. The only possible generalization for iterative solvers was to use the tagging facility for the specification of the desired iterative solver.

7.4 Iterators

Since we use the iterator-driven copy function for transfer from CPU to GPU to CPU, iterators have to be provided anyway. However, it has to be repeated that they are usually VERY slow, because each data access (i.e. dereferentiation) implies a new transfer between CPU and GPU. Nevertheless, CPU-cached vector and matrix classes could be introduced in future releases of ViennaCL.

A remedy for quick iteration over the entries of e.g. a vector is the following:

```
std::vector<double> temp(gpu_vector.size());
copy(gpu_vector.begin(), gpu_vector.end(), temp.begin());
for (std::vector<double>::iterator it = temp.begin();
     it != temp.end();
    ++it)
{
 //do something with the data here
}
copy(temp.begin(), temp.end(), gpu_vector.begin());
```

The three extra code lines can be wrapped into a separate iterator class by the library user, who also has to ensure data consistency during the loop.

7.5 Initialization of Compute Kernels

Since OpenCL relies on passing the OpenCL source code to a built-in just-in-time compiler at run time, the necessary kernels have to be generated every time an application using ViennaCL is started.

One possibility was to require a mandatory

```
viennacl::init();
```

before using any other objects provided by ViennaCL, but this approach was discarded for the following two reasons:

- If viennacl::init(); is accidentally forgotten by the user, the program will most likely terminate in a rather uncontrolled way.
- It requires the user to remember and write one extra line of code, even if the default settings are fine.

Initialization is instead done in the constructors of ViennaCL objects. This allows a fine-grained control over which source code to compile where and when. For example, there is no reason to compile the sparse matrix compute kernels at program startup if there are no sparse matrices used at all.

Moreover, the just-in-time compilation of all available compute kernels in ViennaCL takes several seconds. Therefore, a request-based compilation is used to minimize any overhead due to just-in-time compilation.

The request-based compilation is a two-step process: At the first instantiation of an object of a particular type from ViennaCL, the full source code for all objects of the same type is compiled into a OpenCL program for that type. Each program contains plenty of compute kernels, which are not yet initialized. Only if an argument for a compute kernel is set, the kernel actually cares about its own initialization. Any subsequent calls of that kernel reuse the already compiled and initialized compute kernel.

When benchmarking ViennaCL, first a dummy call to the functionality of interest should be issued prior to taking timings. Otherwise, benchmark results include the just-in-time compilation, which is a constant independent of the data size.

Versioning

Each release of ViennaCL carries a three-fold version number, given by

ViennaCL X.Y.Z .

For users migrating from an older release of ViennaCL to a new one, the following guidelines apply:

- X is the *major version number*, starting with 1. A change in the major version number is not necessarily API-compatible with any versions of ViennaCL carrying a different major version number. In particular, end users of ViennaCL have to expect considerable code changes when changing between different major versions of ViennaCL.
- Y denotes the *minor version number*, restarting with zero whenever the major version number changes. The minor version number is incremented whenever significant functionality is added to ViennaCL. The API of an older release of ViennaCL with smaller minor version number (but same major version number) is *essentially* compatible to the new version, hence end users of ViennaCL usually do not have to alter their application code, unless they have used a certain functionality that was not intended to be used and removed in the new version.
- Z is the *revision number*. If either the major or the minor version number changes, the revision number is reset to zero. Releases of ViennaCL, that only differ in their revision number, are API compatible. Typically, the revision number is increased whenever bugfixes are applied, compute kernels are improved or some extra, not significant functionality is added.

Always try to use the latest version of ViennaCL before submitting bug reports!

Change Logs

Version 1.0.x

Version 1.0.4

The changes in this release are:

- All tutorials now work out-of the box with Visual Studio 2008.
- Eliminated all `ViennaCL` related warnings when compiling with Visual Studio 2008.
- Better (experimental) support for double precision on ATI GPUs, but no `norm_1`, `norm_2`, `norm_inf` and `index_norm_inf` functions using ATI Stream SDK on GPUs in double precision.
- Fixed a bug in GMRES that caused segmentation faults under Windows.
- Fixed a bug in `const_sparse_matrix_adapter` (thanks to Abhinav Golas and Nico Galoppo for almost simultaneous emails on that)
- Corrected incorrect return values in the sparse matrix regression test suite (thanks to Klaus Schnass for the hint)

Version 1.0.3

The main improvements in this release are:

- Support for multi-core CPUs with ATI Stream SDK (thanks to Riccardo Rossi, UPC. BARCELONA TECH, for suggesting this)
- `inner_prod` is now up to a factor of four faster (thanks to Serban Georgescu, ETH, for pointing the poor performance of the old implementation out)
- Fixed a bug with `plane_rotation` that caused system freezes with ATI GPUs.
- Extended the doxygen generated reference documentation

Version 1.0.2

A bug-fix release that resolves some problems with the Visual C++ compiler.

- Fixed some compilation problems under Visual C++ (version 2005 und 2008).

- All tutorials accidentally relied on `ublas`. Now `tut1` and `tut5` can be compiled without `ublas`
- Renamed `aux/` folder to `auxiliary/` (caused some problems on windows machines)

Version 1.0.1

This is a quite large revision of `ViennaCL 1.0.0`, but mainly improves things under the hood.

- Fixed a bug in lu_substitute for dense matrices
- Changed iterative solver behavior to stop if a certain relative residual is reached
- ILU preconditioning is now fully done on the CPU, because this gives best overall performance
- All OpenCL handles of `ViennaCL` types can now be accessed via member function `handle()`
- Improved GPU performance of GMRES by about a factor of two.
- Added generic `norm_2` function in header file `norm_2.hpp`
- Wrapper for `clFlush()` and `clFinish()` added
- Device information can be queried by `device.info()`
- Extended documentation and tutorials

Version 1.0.0

First release

License

Copyright (c) 2010, Institute for Microelectronics, TU Wien

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the ”Software”), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ”AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Bibliography

[1] Y. Saad, *Iterative Methods for Sparse Linear Systems, Second Edition*. Society for Industrial and Applied Mathematics, April 2003.

[2] "Boost C++ Libraries." [Online]. Available: http://www.boost.org/

[3] "Khronos OpenCL." [Online]. Available: http://www.khronos.org/opencl/

[4] "Nvidia OpenCL." [Online]. Available: http://www.nvidia.com/object/cuda_opencl_new.html

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

[6] "ATI Stream SDK." [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/Pages/default.aspx

[7] "ATI Stream SDK - Documentation." [Online]. Available: http://developer.amd.com/gpu/ATIStreamSDK/pages/Documentation.aspx

[8] "ATI Knowledge Base - Double Support." [Online]. Available: http://developer.amd.com/support/KnowledgeBase/Lists/KnowledgeBase/DispForm.aspx?ID=88

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

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

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

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