ViennaCL 1.1.2

User Manual

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 by using `OpenCL`. The primary focus is on common linear algebra operations (BLAS levels 1, 2 and 3) and the solution of large sparse systems of equations by means of iterative methods. In `ViennaCL 1.1.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 1.1.2` precomputed and applied on a single CPU core and may thus not lead to overall performance gains over a purely CPU based implementation. Moreover, a Jacobi and a row-scaling preconditioner are available, which can be executed directly in parallel on the `OpenCL` device.

The solvers and preconditioners can also be used with different libraries due to their generic implementation. At present, it is possible to use the solvers and preconditioners directly with types from the `ublas` library, which is part of `Boost` [2]. The iterative solvers can directly be used with Eigen [3] and MTL 4 [4].

Under the hood, `ViennaCL` uses `OpenCL` [5] 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 AMD (see Tab. 1) as well as on CPUs using either the AMD Accelerated Parallel Processing SDK (formerly ATI Stream SDK) or the Intel OpenCL SDK.

> 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 or AMD APP SDK is not yet fully OpenCL-certified. 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 GTX 560	ok	ok
NVIDIA GTX 570	ok	ok
NVIDIA GTX 580	ok	ok
NVIDIA GTX 590	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	essentially ok
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	essentially ok
ATI Radeon HD 59XX	ok	essentially ok
ATI Radeon HD 68XX	ok	-
ATI Radeon HD 69XX	ok	essentially ok
ATI FireStream V92XX	ok	essentially ok
ATI FirePro V78XX	ok	essentially ok
ATI FirePro V87XX	ok	essentially ok
ATI FirePro V88XX	ok	essentially ok

Table 1: Available arithmetics in `ViennaCL` provided by selected GPUs. At the release of `ViennaCL 1.1.2`, the Stream SDK from AMD/ATI does not comply to the `OpenCL` standard for double precision extensions, but we have not observed problems with the latest version of Stream SDK. Support for AMD devices is now per default enabled 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

<div align="center">

`viennacl-support@lists.sourceforge.net`

</div>

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.

To use `ViennaCL`, the following prerequisites have to be fulfilled:

- A recent C++ compiler (e.g. `GCC` version 4.2.x or above and Visual C++ 2008 are known to work)

- `OpenCL` [5, 6] for accessing compute devices (GPUs); see Section 1.3 for details. (optional, since iterative solvers can also be used standalone with other libraries (`ublas`, Eigen, MTL4))

The full potential of `ViennaCL` is only available with the following optional libraries:

- `CMake` [7] as build system (optional, but highly recommended for building examples)

- `ublas` (shipped with `Boost` [2]) provides the same interface as `ViennaCL` and allows to switch between CPU and GPU seamlessly, see the tutorials.

- Eigen [3] can be used to fill `ViennaCL` types directly. Moreover, the iterative solvers in `ViennaCL` can directly be used with Eigen objects.

- MTL 4 [4] can be used to fill `ViennaCL` types directly. Even though MTL 4 provides its own iterative solvers, the `ViennaCL` solvers can also be used with MTL 4 objects.

1.2 Generic Installation of ViennaCL

Since `ViennaCL` is essentially a header-only library (the only exception is described in Chapter 8), 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/`. If the OpenCL headers are not installed on your system, you should repeat the above procedure with the folder `CL/`.

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

In order to compile and run `OpenCL` applications, a corresponding library (e.g. `libOpenCL.so` under Unix based systems) and is required. If `OpenCL` is to be used with GPUs, suitable drivers have to be installed. 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 260.19.21. Note that the latest NVIDIA drivers do not include the `OpenCL` headers anymore. Therefore, the official `OpenCL` headers from the Khronos group [5] are also shipped with `ViennaCL` in the folder `CL/`.

1.3.2 AMD Accelerated Parallel Processing SDK (formerly Stream SDK)

AMD provides the `OpenCL` library with the Accelerated Parallel Processing (APP) SDK [8]. At the release of `ViennaCL 1.1.2`, the latest version of the SDK is 2.4. If used with AMD GPUs, recent AMD GPU drivers are typically required. If `ViennaCL` is to be run on multi-core CPUs, no additional GPU driver is required. The installation notes of the APP SDK provides guidance throughout the installation process [9].

> If the SDK is installed in a non-system wide location on UNIX-based systems, 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 AMD APP SDK does not provide `OpenCL`certified double precision support [10] on some CPUs and GPUs. In `ViennaCL`1.0.x, double precision was only experimentally available in `ViennaCL` by defining one of the preprocessor constants

```
// 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. With `ViennaCL 1.1.x`, this is not necessary anymore and double precision support is enabled by default – provided that it is available on the device.

> The functions `norm_1`, `norm_2`, `norm_inf` and `index_norm_inf` are known to cause problems on GPUs in double precision using ATI Stream SDK v2.1.

1.3.3 INTEL OpenCL SDK

At the time of this release, a beta-version of an `OpenCL` SDK by INTEL is available.

Even though the SDK is still in beta-state, `ViennaCL` works fine with the INTEL OpenCL SDK on Windows and Linux. The correct linker path is set automatically in `CMakeLists.txt` when using the `CMake` build system, cf. Sec. 1.2.

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.

Before building the examples, customize `CMakeLists.txt` in the `ViennaCL` root folder for your needs. Per default, all examples using `ublas`, Eigen and MTL4 are turned off. Please enable the respective examples based on the libraries available on your machine. Directions on how to accomplish this are given directly within the `CMakeLists.txt` file.

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 and type

Tutorial No.	Dependencies
`tutorial/blas1.cpp`	`OpenCL`
`tutorial/blas2.cpp`	`OpenCL`, `ublas`
`tutorial/blas3.cpp`	`OpenCL`, `ublas`
`tutorial/iterative.cpp`	`OpenCL`, `ublas`
`tutorial/iterative-ublas.cpp`	`ublas`
`tutorial/iterative-eigen.cpp`	`Eigen`
`tutorial/iterative-mtl4.cpp`	`MTL 4`
`tutorial/custom-kernel.cpp`	`OpenCL`
`tutorial/custom-context.cpp`	`OpenCL`
`tutorial/eigen-with-viennacl.cpp`	`OpenCL`, `Eigen`
`tutorial/mtl4-with-viennacl.cpp`	`OpenCL`, `MTL 4`
`tutorial/viennacl-info.cpp`	`OpenCL`
`benchmarks/vector.cpp`	`OpenCL`
`benchmarks/sparse.cpp`	`OpenCL`, `ublas`
`benchmarks/solver.cpp`	`OpenCL`, `ublas`
`benchmarks/opencl.cpp`	`OpenCL`
`benchmarks/blas3.cpp`	`OpenCL`

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

```
$> make
```

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

```
$> 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`. 💡

1.4.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the `GCC` compiler the Xcode [11] package has to be installed. To install `CMake` and `Boost` external portation tools have to be used, for example, Fink [12], DarwinPorts [13] or MacPorts [14]. 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` on Mac OS 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 Chapter 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:

```cpp
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 1.1.2`) 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 similar to `std::copy()` from the C++ Standard Template Library (STL), should be used for writing 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. 6)

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 (refer to Chapter 3 for more details):

```
// 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 pure CPU implementations is that the bracket operator as well as the parenthesis operator are very slow, because for each access an `OpenCL` data transfer has to be initiated. The overhead of this transfer is orders of magnitude. For example:

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

// fill a ViennaCL vector - VERY SLOW!!
for (size_t i=0; i<gpu_vector.size(); ++i)
    vcl_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 - faster versions:
copy(cpu_vector, vcl_vector);                                        //option 1
copy(cpu_vector.begin(), cpu_vector.end(), vcl_vector.begin()); //option 2
```

In this way, setup costs for the CPU vector and the `ViennaCL` 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`. Since `ViennaCL` 1.1.0, also `column_major` memory layout can be used. The third template argument `alignment` denotes an alignment for the rows and columns for row-major and column-major memory layout (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>  vcl_matrix(4, 5);

//fill it up:
vcl_matrix(0,2) = 1.0;
vcl_matrix(1,2) = -1.5;
vcl_matrix(2,0) = 4.2;
vcl_matrix(3,4) = 3.1415;
```

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:

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. 6)

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 CPU matrix to GPU matrix
copy(cpu_matrix, gpu_matrix);

//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. Please refer to Chap. 9 for an overview of other libraries for which an overload of `copy()` is provided.

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` 1.1.2, 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

Interface	Comment
`CTOR(nrows, ncols)`	Constructor with number of rows and columns
`mat.set()`	Initialize mat with the data provided as arguments
`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. 6)
`mat.handle2()`	Returns the GPU handle holding the column indices (needed for custom kernels, see Chap. 6)
`mat.handle()`	Returns the GPU handle holding the entries (needed for custom kernels, see Chap. 6)

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

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 dynamic memory management of sparse matrices is not provided on `OpenCL` compute devices such as GPUs.

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:

```cpp
//set up a sparse 3 by 5 matrix on the CPU:
std::vector< std::map< unsigned int, float> > cpu_sparse_matrix(4);

//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 ViennaCL matrix:
viennacl::compressed_matrix<float>  vcl_sparse_matrix(4, 5);

//copy to OpenCL device:
copy(cpu_sparse_matrix, vcl_sparse_matrix);

//copy back to CPU:
copy(vcl_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. Please refer to Chap. 9 for an overview of other libraries for which an overload of `copy()` is provided.

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 dynamic memory management of sparse matrices is not provided on `OpenCL` 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.

> In `ViennaCL 1.1.2` the use of `compressed_matrix` over `coordinate_matrix` is encouraged due to better performance!

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. 6)
`mat.handle()`	Returns the GPU handle holding the entries (needed for custom kernels, see Chap. 6)

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 we move on with the description of the basic BLAS operations.

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/`

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 iterative solvers.

3.3 Matrix-Matrix Operations (BLAS Level 3)

Full BLAS level 3 support is since `ViennaCL` 1.1.0, cf. Tab. 3.3. While BLAS levels 1 and 2 are mostly memory-bandwidth-limited, BLAS level 3 is mostly limited by the available computational power of the respective device. Hence, matrix-matrix products regularly

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`

show impressive performance gains on mid- to high-end GPUs when compared to a single CPU core.

Again, the `ViennaCL` API is identical to that of `ublas` and comparisons can be carried out immediately, as is shown in the tutorial located in `examples/tutorial/blas3.cpp`.

As for performance, `ViennaCL` yields decent performance gains at BLAS level 3 on mid- to high-end GPUs compared to CPU implementations using a single core only. Nevertheless, one must not expect to obtain the reported peak performance of hundreds of GFLOPs for the multiplication of arbitrary matrices. These rates can typically only obtained when tailoring the compute kernel(s) to a particular device and certain matrix dimensions, while `ViennaCL` provides kernels that represent a good compromise between efficiency and portability among a large number of different devices and device types.

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`

Verbal	Mathematics	ViennaCL
matrix-matrix product	$C \leftarrow A \times B$	`C = prod(A, B);`
matrix-matrix product	$C \leftarrow A \times B^{\mathrm{T}}$	`C = prod(A, trans(B));`
matrix-matrix product	$C \leftarrow A^{\mathrm{T}} \times B$	`C = prod(trans(A), B);`
matrix-matrix product	$C \leftarrow A^{\mathrm{T}} \times B^{\mathrm{T}}$	`C = prod(trans(A), trans(B));`
tri. matrix solve	$C \leftarrow A^{-1}B$	`C = solve(A, B, tag);`
tri. matrix solve	$C \leftarrow A^{\mathrm{T}^{-1}}B$	`C = solve(trans(A), B, tag);`
tri. matrix solve	$C \leftarrow A^{-1}B^{\mathrm{T}}$	`C = solve(A, trans(B), tag);`
tri. matrix solve	$C \leftarrow A^{\mathrm{T}^{-1}}B^{\mathrm{T}}$	`C = solve(trans(A), trans(B), tag);`
inplace solve	$B \leftarrow A^{-1}B$	`inplace_solve(A, trans(B), tag);`
inplace solve	$B \leftarrow A^{\mathrm{T}^{-1}}B$	`inplace_solve(trans(A), x, tag);`
inplace solve	$B \leftarrow A^{-1}B^{\mathrm{T}}$	`inplace_solve(A, trans(B), tag);`
inplace solve	$B \leftarrow A^{\mathrm{T}^{-1}}B^{\mathrm{T}}$	`inplace_solve(trans(A), x, tag);`

Table 3.3: BLAS level 3 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 (either using `ublas`, Eigen, MTL4 or `OpenCL`) and on the GPU (using `OpenCL`).

4.1   Direct Solvers

`ViennaCL 1.1.2` provides triangular solvers and LU factorization without pivoting for the solution of dense linear systems. The interface is similar to that of `ublas`

```
using namespace viennacl::linalg;  //to keep solver calls short
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 = solve(vcl_matrix, vcl_rhs, upper_tag());
//solution of a lower triangular system:
vcl_result = solve(vcl_matrix, vcl_rhs, lower_tag());

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

In `ViennaCL 1.1.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.

It is also possible to solve for multiple right hand sides:

```
using namespace viennacl::linalg;  //to keep solver calls short
viennacl::matrix<float>  vcl_matrix;
viennacl::matrix<float>  vcl_rhs_matrix;
viennacl::matrix<float>  vcl_result;

/* Set up matrices here */
```

```
  //solution of an upper triangular system:
  vcl_result = solve(vcl_matrix, vcl_rhs_matrix, upper_tag());

  //solution of a lower triangular system:
  vcl_result = solve(vcl_matrix, vcl_rhs_matrix, lower_tag());
```

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 prop-
erties 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`, Eigen and MTL4 objects!
> Please have a look at Chap. 9 and the respective tutorials in the examples/tutori-
> als/ folder.

> In `ViennaCL 1.1.2`, GMRES using ATI GPUs yields wrong results due to a bug
> in Stream SDK v2.1. Consider using newer versions of the 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());
```

Customized error tolerances can be set in the solver tags. The convention is that solver tags
take the relative error tolerance as first argument and the maximum number of iteration
steps as second argument. Furthermore, after the solver run the number of iterations and

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 estimated error can be obtained from the solver tags as follows:

```
// conjugate gradient solver with tolerance 1e10
// and at most 100 iterations:
viennacl::linalg::cg_tag custom_cg(1e-10, 100);
vcl_result = viennacl::linalg::solve(vcl_matrix, vcl_rhs, custom_cg);
//print number of iterations taken and estimated error:
std::cout << "No. of iters: " << custom_cg.iters() << std::endl;
std::cout << "Est. error: " << custom_cg.error() << std::endl;
```

The BiCGStab solver tag can be customized in exactly the same way. The GMRES solver tag takes as third argument the dimension of the Krylov space. Thus, a tag for GMRES(30) with tolerance $1E-10$ and at most 100 total iterations (hence, up to three restarts) can be set up by

```
viennacl::linalg::gmres_tag custom_gmres(1e-10, 100, 30);
```

4.3 Preconditioners

`ViennaCL` ships with a generic implementation of an incomplete LU factorization preconditioner with threshold (ILUT), a Jacobi preconditioner and a row-scaling preconditioner. The incomplete factorization for ILUT 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. More preconditioners are in preparation and any contributions are very welcome.

The preconditioner also works for `ublas` types!

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


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

/* Set up matrix and vectors here */
```

Method	Brief description	Parameters
ILUT	incomplete LU factorization	First parameter: Maximum number of entries per row. Second parameter: Drop tolerance.
Jacobi	Divide each row in A by its diagonal entry	none
Row Scaling	Divide each row in A by its norm	First parameter specifies the norm (1: l^1-norm, 2: l^2-norm)

Table 4.2: Preconditioners for iterative solvers in `ViennaCL`.

```cpp
//compute ILUT preconditioner:
ilut_precond< SparseMatrix > vcl_ilut(vcl_matrix,
                                      viennacl::linalg::ilut_tag());
//compute Jacobi preconditioner:
jacobi_precond< SparseMatrix > vcl_jacobi(vcl_matrix,
                                      viennacl::linalg::jacobi_tag());
//compute ILUT preconditioner:
row_scaling< SparseMatrix > vcl_row_scaling(vcl_matrix,
                                      viennacl::linalg::row_scaling_tag());

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

Chapter 5

Configuring Contexts and Devices

Support for multiple devices was officially added in `OpenCL` 1.1. Among other things, this allows e.g. to use all CPUs in a multi-socket CPU mainboard as a single `OpenCL` compute device. Nevertheless, the efficient use of multiple `OpenCL` devices is far from trivial, because algorithms have to be designed such that they take distributed memory and synchronization issues into account.

Support for multiple devices and contexts was introduced in `ViennaCL` with version 1.1.0. In the following we give a description of the provided functionality.

5.1   Context Setup

Unless specified otherwise (see Chap. 7), `ViennaCL` silently creates its own context and adds all available default devices with a single queue per device to it. All operations are then carried out on this context, which can be obtained with the call

```
viennacl::ocl::current_context();
```

This default context is identified by the ID 0 (of type **long**). By default, only the first device in the context is used for all operations. This device can be obtained via

```
viennacl::ocl::current_context().current_device();
viennacl::ocl::current_device(); //equivalent to above
```

A user may wish to use multiple contexts, where each context consists of a subset of the available devices. To setup a context with ID `id` with a particular device type only, the user has to specify this prior to any other `ViennaCL` related statements:

```
//use only GPUs:
viennacl::ocl::set_context_device_type(id, viennacl::ocl::gpu_tag());
//use only CPUs:
viennacl::ocl::set_context_device_type(id, viennacl::ocl::cpu_tag());
//use only the default device type
viennacl::ocl::set_context_device_type(id, viennacl::ocl::default_tag());
//use only accelerators:
viennacl::ocl::set_context_device_type(id, viennacl::ocl::accelerator_tag()
    );
```

Instead of using the tag classes, the respective `OpenCL` constants `CL_DEVICE_TYPE_GPU` etc. can be supplied as second argument.

23

Another possibility is to query all devices from the current platform:

```
std::vector< viennacl::ocl::device > devices =
    viennacl::ocl::platform().devices();
```

and create a custom subset of devices, which is then passed to the context setup routine:

```
//take the first and the third available device from 'devices'
std::vector< viennacl::ocl::device > my_devices;
my_devices.push_back(devices[0]);
my_devices.push_back(devices[2]);

//Initialize the context with ID 'id' with these devices:
viennacl::ocl::setup_context(id, my_devices);
```

Similarly, contexts with other IDs can be set up.

> For details on how to initialize `ViennaCL` with already existing contexts, see Chapter 7.

The library user is reminded that memory objects within a context are allocated for all devices within a context. Thus, setting up contexts with one device each is optimal in terms of memory usage, because each memory object is then bound to a single device only. However, memory transfer between contexts (and thus devices) has to be done manually by the library user then. Moreover, the user has to keep track in which context the individual `ViennaCL` objects have been created, because all operands are assumed to be in the currently active context.

5.2 Switching Contexts and Devices

`ViennaCL` always uses the currently active context with the currently active device to enqueue compute kernels. The default context is identified by ID '0'. The context with ID `id` can be set as active context with the line.

```
viennacl::ocl::switch_context(id);
```

Subsequent kernels are then enqueued on the active device for that particular context.

Similar to setting contexts active, the active device can be set for each context. For example, setting the second device in the context to be the active device, the lines

```
viennacl::ocl::current_context().switch_device(1);
```

are required. In some circumstances one may want to pass the device object directly, e.g. to set the second device of the platform active:

```
std::vector<viennacl::ocl::device> const & devices =
    viennacl::ocl::platform().devices();
viennacl::ocl::current_context().switch_device(devices[1]);
```

If the supplied device is not part of the context, an error message is printed and the active device remains unchanged.

Chapter 6

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 desirable 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

- Launching the kernel

A tutorial on this topic can be found at `examples/tutorial/custom-kernels.cpp`.

The interface for custom kernels was simplified considerably in `ViennaCL` 1.1.0.

6.1   Setting up the Source Code

The `OpenCL` source code has to be provided as a string. One can either write the source code directly into a string within C++ files, or one 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 `result`. For more detailed explanation of

the `OpenCL` source code, please refer to the specification available at the Khronos group webpage [5].

6.2 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 & prog =
  viennacl::ocl::current_context().add_program(my_compute_program,
                                          "my_compute_program");
```

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

```
viennacl::ocl::kernel & my_kernel = my_prog.add_kernel("elementwise_prod");
```

Now, the kernel is set up to use the function `elementwise_prod` compiled into the program `my_prog`.

> Note that C++ references to kernels and programs may become invalid as other kernels or programs are added. This is the case at the first instantiation of an `ViennaCL` object of a particular type. Therefore, first allocate the required `ViennaCL` objects and compile/add all custom kernels, before you start passing references to programs or kernels around.

Instead of holding references to programs and kernels directly at compilation, one can obtain them at other places within the application source code by

```
viennacl::ocl::program & prog =
  viennacl::ocl::current_context().get_program("my_compute_program");
viennacl::ocl::kernel & my_kernel = my_prog.get_kernel("elementwise_prod");
```

6.3 Launching the Kernel

Before launching the kernel, one may adjust the global and local work sizes (readers not familiar with that are encouraged to read the `OpenCL` standard [5]). The following code specifies a one-dimensional execution model with 16 local workers and 128 global workers:

```
my_kernel.local_work_size(0, 16);
my_kernel.global_work_size(0, 128);
```

In order to use a two-dimensional execution, additionally parameters for the second dimension are set by

```
my_kernel.local_work_size(1, 16);
my_kernel.global_work_size(1, 128);
```

However, for the simple kernel in this example it is not necessary to specify any work sizes at all. The default work sizes (which can be found in `viennacl/ocl/kernel.hpp`) suffice for most cases.

To launch the kernel, the kernel arguments are set in the same way as for ordinary functions. We assume that three `ViennaCL` vectors `vec1`, `vec2` and `result` have already been set up:

```
viennacl::ocl::enqueue(my_kernel(vec1, vec2, result, vec1.size()));
```

Per default, the kernel is enqueued in the first queue of the currently active device. A custom queue can be specified as optional second argument, cf. the reference documentation located in `doc/doxygen/`.

Chapter 7

Using ViennaCL in User Provided OpenCL Contexts

Many projects need similar basic linear algebra operations, but essentially operate in their own `OpenCL` context. To provide the functionality and convenience of `ViennaCL` to such existing projects, existing contexts can be passed to `ViennaCL` and memory objects can be wrapped into the basic linear algebra types `vector`, `matrix` and `compressed_matrix`. This chapter is devoted to the description of the necessary steps to use `ViennaCL` on contexts provided by the library user.

> An example of providing a custom context to `ViennaCL` can be found in `examples/tutorial/custom-contexts.cpp`

7.1   Passing Contexts to ViennaCL

`ViennaCL 1.1.2` is able to handle an arbitrary number of contexts, which are identified by a key value of type **long**. By default, `ViennaCL` operates on the context identified by 0, unless the user switches the context, cf. Chapter 5.

According to the `OpenCL` standard, a context contains devices and queues for each device. Thus, it is assumed in the following that the user has successfully created a context with one or more devices and one or more queues per device.

In the case that the context contains only one device `my_device` and one queue `my_queue`, the context can be passed to `ViennaCL` with the code

```
cl_context my_context = ...;       //a context
cl_device_id my_device = ...;      //a device in my_context
cl_command_queue my_queue = ...;   //a queue for my_device

//supply existing context 'my_context'
// with one device and one queue to ViennaCL using id '0':
viennacl::ocl::setup_context(0, my_context, my_device, my_queue);
```

If a context ID other than 0, say, `id` is used, the user-defined context has to be selected using

```
viennacl::ocl::switch_context(id);
```

It is also possible to provide a context with several devices and multiple queues per device. To do so, the device IDs have to be stored in a STL vector and the queues in a STL map:

```
cl_context my_context = ...;    //a context

cl_device_id my_device1 = ...; //a device in my_context
cl_device_id my_device2 = ...; //another device in my_context
...

cl_command_queue my_queue1 = ...; //a queue for my_device1
cl_command_queue my_queue2 = ...; //another queue for my_device1
cl_command_queue my_queue3 = ...; //a queue for my_device2
...

//setup existing devices for ViennaCL:
std::vector<cl_device_id> my_devices;
my_devices.push_back(my_device1);
my_devices.push_back(my_device2);
...

//setup existing queues for ViennaCL:
std::map<cl_device_id,
        std::vector<cl_command_queue> > my_queues;
my_queues[my_device1].push_back(my_queue1);
my_queues[my_device1].push_back(my_queue2);
my_queues[my_device2].push_back(my_queue3);
...

//supply existing context with multiple devices
//and queues to ViennaCL using id '0':
viennacl::ocl::setup_context(0, my_context, my_devices, my_queues);
```

It is not necessary to pass all devices and queues created within a particular context to `ViennaCL`, only those which `ViennaCL` should use have to be passed. `ViennaCL` will by default use the first queue on each device. The user has to care for appropriate synchronization between different queues.

7.2   Wrapping Existing Memory with ViennaCL Types

Now as the user provided context is supplied to `ViennaCL`, user-created memory objects have to be wrapped into `ViennaCL` data-types in order to use the full functionality. Typically, one of the types `scalar`, `vector`, `matrix` and `compressed_matrix` are used:

```
cl_mem my_memory1 = ...;
cl_mem my_memory2 = ...;
cl_mem my_memory3 = ...;
cl_mem my_memory4 = ...;
cl_mem my_memory5 = ...;

//wrap my_memory1 into a vector of size 10
viennacl::vector<float> my_vec(my_memory1, 10);

//wrap my_memory2 into a row-major matrix of size 10x10
viennacl::matrix<float> my_matrix(my_memory2, 10, 10);
```

```
//wrap my_memory3 into a CSR sparse matrix with 10 rows and 20 nonzeros
viennacl::compressed_matrix<float> my_sparse(my_memory3,
                                             my_memory4,
                                             my_memory5, 10, 10, 20);

//use my_vec, my_matrix, my_sparse as usual
```

The following has to be emphasized:

- Resize operations on `ViennaCL` data types typically results in the object owning a new piece of memory.

- copy() operations from CPU RAM usually allocate new memory, so wrapped memory is "forgotten"

- On construction of the `ViennaCL` object, `clRetainMem()` is called once for the provided memory handle. Similarly, `clReleaseMem()` is called as soon as the memory is not used any longer.

The user has to ensure that the provided memory is larger or equal to the size of the wrapped object.

Be aware the wrapping the same memory object into several different `ViennaCL` objects can have unwanted side-effects. In particular, wrapping the same memory in two `ViennaCL` vectors implies that if the entries of one of the vectors is modified, this is also the case for the second.

Chapter 8

Kernel Parameter Tuning

The choice of the global and local work sizes for `OpenCL` kernels typically has a considerable impact on the obtained device performance. The default setting in `ViennaCL` is – with some exceptions – to use the same global and local work sizes for each compute kernel. To obtain highest performance, optimal work sizes have to be determined for each kernel in dependence of the underlying device.

8.1 Start Tuning Runs

`ViennaCL 1.1.2` ships with a automated tuning environment, which tries to determine the best kernel parameters for the available device. At present, only kernel parameters for the first device are optimized. The tuning programs are located in

- `examples/parameters/vector.cpp`: Tuning for vector kernels

- `examples/parameters/matrix.cpp`: Tuning for matrix kernels

- `examples/parameters/sparse.cpp`: Tuning for sparse matrix kernels

and are built together with other examples when using `CMake`. The executables are

- `vectorparams`,

- `matrixparams`,

- `sparseparams`

respectively and are executed without additional parameters. During execution, these programs create three XML files `vector_parameters.xml`, `matrix_parameters.xml` and `sparse_parameters.xml`, which hold the best parameter set.

At present, only `ViennaCL` types with standard alignment are benchmarked. Higher performance can be obtained when allowing further memory alignments and comparing different implementations. This, however, is not yet available, but may be part of future versions.

8.2 Load Best Parameters at Startup

In order to load the best parameters at each startup, the parameter reader located at `viennacl/io/kernel_parameters.hpp` can be used. The individual kernels for the respective `ViennaCL` types can be loaded with the lines

```cpp
using viennacl::io;
read_kernel_parameters< viennacl::vector<float> >("vector_parameters.xml");
read_kernel_parameters< viennacl::matrix<float> >("matrix_parameters.xml");
read_kernel_parameters< viennacl::compressed_matrix<float> >("
   sparse_parameters.xml");

//similarly for the numeric type double
```

where the filename is as usual relative to current working directory. A simple example doing just that can be found in `examples/parameters/parameter_reader.cpp`. In principle, kernel parameters can all be located in a single XML file, from which the call to `read_kernel_parameters()` will then extract the relevant ones for the respective `ViennaCL` type and the available device.

> Please note that in order to read the parameters, the project has to be linked with pugixml [15], which is shipped with `ViennaCL` in `external/`

Chapter 9

Interfaces to Other Libraries

`ViennaCL` aims at compatibility with as many other libraries as possible. This is on the one hand achieved by using generic implementations of the individual algorithms, and on the other hand by providing the necessary wrappers.

The interfaces to third-party libraries provided with `ViennaCL` are explained in the following subsections. Please feel free to suggest additional libraries for which an interface should be shipped with `ViennaCL`.

Since it is unlikely that all third-party libraries for which `ViennaCL` provides interfaces are installed on the target machine, the wrappers are disabled by default. To selectively enable the wrappers, the appropriate preprocessor constants `VIENNACL_HAVE_XXXX` have to be defined *prior to any `#include` statements for `ViennaCL` headers*. This can for example be assured by passing the preprocessor constant directly when launching the compiler. With `GCC` this is for instance achieved by the `-D` switch.

9.1 `ublas`

Since all types in `ViennaCL` have to same interface as their counterparts in `ublas`, most code written for `ViennaCL` objects remains valid when using `ublas` objects.

```cpp
//Option 1: Using ViennaCL:
using namespace viennacl;
using namespace viennacl::linalg;

//Option 2: Using ublas:
//using namespace boost::numeric::ublas;

matrix<float> dense_matrix(5,5);
vector<float> dense_vector(5,5);
compressed_matrix<float> sparse_matrix(1000, 1000);

//fill with data:
dense_matrix(0,0) = 2.0;
....

//run solvers
vector<float> result1 = solve(dense_matrix, dense_vector, upper_tag());
vector<float> result2 = viennacl::linalg::solve(sparse_matrix,
                                              dense_vector, cg_tag());
```

The above code is valid for either the `ViennaCL` namespace declarations, or the `ublas` namespace. Note that the iterative solvers are not part of `ublas` and therefore the explicit namespace specification is required. More examples for the exchangability of `ublas` and `ViennaCL` can be found in the tutorials in the `examples/tutorials/` folder.

When using the iterative solvers, the preprocessor constant `VIENNACL_HAVE_UBLAS` must be defined prior to any other `ViennaCL` include statements. This is essential for enabling the respective wrappers.

> Refer in particular to `iterative-ublas.cpp` for a complete example on iterative solvers using `ublas` types.

9.2 `Eigen`

To copy data from `Eigen` [3] objects to `ViennaCL`, the `copy()`-functions are used just as for `ublas` and STL types:

```cpp
//from Eigen to ViennaCL
viennacl::copy(eigen_vector, vcl_vector);
viennacl::copy(eigen_densematrix, vcl_densematrix);
viennacl::copy(eigen_sparsematrix, vcl_sparsematrix);
```

In addition, the STL-compliant iterator-version of `viennacl::copy()` taking three arguments can be used for copying vector data. Here, all types prefixed with `eigen` are `Eigen` types, the prefix `vcl` indicates `ViennaCL` objects. Similarly, the transfer from `ViennaCL` back to `Eigen` is accomplished by

```cpp
//from ViennaCL to Eigen
viennacl::copy(vcl_vector, eigen_vector);
viennacl::copy(vcl_densematrix, eigen_densematrix);
viennacl::copy(vcl_sparsematrix, eigen_sparsematrix);
```

The iterative solvers in `ViennaCL` can also be used directly with `Eigen` objects:

```cpp
using namespace viennacl::linalg; //for brevity of the following lines
eigen_result = solve(eigen_matrix, eigen_rhs, cg_tag());
eigen_result = solve(eigen_matrix, eigen_rhs, bicgstab_tag());
eigen_result = solve(eigen_matrix, eigen_rhs, gmres_tag());
```

When using the iterative solvers with `Eigen`, the preprocessor constant `VIENNACL_HAVE_EIGEN` must be defined prior to any other `ViennaCL` include statements. This is essential for enabling the respective wrappers.

> Refer to `iterative-eigen.cpp` and `eigen-with-viennacl.cpp` for complete examples.

9.3 MTL 4

The following lines demonstate how `ViennaCL` types are filled with data from `MTL 4` [4] objects:

```
//from Eigen to ViennaCL
viennacl::copy(mtl4_vector, vcl_vector);
viennacl::copy(mtl4_densematrix, vcl_densematrix);
viennacl::copy(mtl4_sparsematrix, vcl_sparsematrix);
```

In addition, the STL-compliant iterator-version of `viennacl::copy()` taking three arguments can be used for copying vector data. Here, all types prefixed with `mtl4` are `MTL 4` types, the prefix `vcl` indicates `ViennaCL` objects. Similarly, the transfer from `ViennaCL` back to `MTL 4` is accomplished by

```
//from ViennaCL to MTL4
viennacl::copy(vcl_vector, mtl4_vector);
viennacl::copy(vcl_densematrix, mtl4_densematrix);
viennacl::copy(vcl_sparsematrix, mtl4_sparsematrix);
```

Even though `MTL 4` provides its own set of iterative solvers, the iterative solvers in `ViennaCL` can also be used:

```
using namespace viennacl::linalg; //for brevity of the following lines
mtl4_result = solve(mtl4_matrix, mtl4_rhs, cg_tag());
mtl4_result = solve(mtl4_matrix, mtl4_rhs, bicgstab_tag());
mtl4_result = solve(mtl4_matrix, mtl4_rhs, gmres_tag());
```

Our internal tests have shown that the execution time of `MTL 4` solvers is equal to `ViennaCL` solvers when using `MTL 4` types.

When using the iterative solvers with `MTL 4`, the preprocessor constant `VIENNACL_HAVE_MTL4` must be defined prior to any other `ViennaCL` include statements. This is essential for enabling the respective wrappers.

Refer to `iterative-mtl4.cpp` and `mtl4-with-viennacl.cpp` for complete examples.

Chapter 10

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	AMD Phenom II X4-965
RAM	8 GB
OS	Funtoo Linux 64 bit
Kernel for AMD cards:	2.6.33
AMD driver version:	10.4
Kernel for Nvidia cards:	2.6.34
Nvidia driver version:	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 at the time of these benchmarks, double precision arithmetics is not included, cf. Tab. 1.

> ⚠️ 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.

10.1 Vector Operations

Benchmarks for the addition of two vectors and the computation of inner products are shown in Tab. 10.1.

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 10.1: 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 10.2: Execution times (seconds) for sparse matrix-vector multiplication using `compressed_matrix`.

10.2 Matrix-Vector Multiplication

We have compared execution times of the operation

$$\mathbf{y} = \mathbf{A}\mathbf{x} \,, \tag{10.1}$$

where $\mathbf{A}$ is a sparse matrix (ten entries per column on average). The results in Tab. 10.2 shows that by the use of `ViennaCL` and a mid-range GPU, performance gains of up to one order of magnitude can be obtained.

10.3 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. 10.3 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 10.3: Execution times (seconds) for ten iterations of CG and GMRES without preconditioner. Results for BiCGStab are similar to that of CG.

Chapter 11

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.

11.1 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! ⚠

11.2 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.

11.3 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.

11.4 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.

11.5 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

$$\text{ViennaCL X.Y.Z.}$$

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 1.1.x

Version 1.1.2

This final release of the `ViennaCL` 1.1.x family focuses on refurbishing existing functionality:

- Fixed a bug with partial vector copies from CPU to GPU (thanks to sourceforge.net user kaiwen).

- Corrected error estimations in CG and BiCGStab iterative solvers (thanks to Riccardo Rossi for the hint).

- Improved performance of CG and BiCGStab as well as Jacobi and row-scaling preconditioners considerably (thanks to Farshid Mossaiby and Riccardo Rossi for a lot of input).

- Corrected linker statements in CMakeLists.txt for MacOS (thanks to Eric Christiansen).

- Improved handling of `ViennaCL` types (direct construction, output streaming of matrix- and vector-expressions, etc.).

- Updated old code in the `coordinate_matrix` type and improved performance (thanks to Dongdong Li for finding this).

- Using `size_t` instead of **`unsigned int`** for the size type on the host.

- Updated double precision support detection for AMD hardware.

- Fixed a name clash in direct_solve.hpp and ilu.hpp (thanks to sourceforge.net user random).

- Prevented unsupported assignments and copies of sparse matrix types (thanks to sourceforge.net user kszyh).

Version 1.1.1

This new revision release has a focus on better interaction with other linear algebra libraries. The few known glitches with version 1.1.0 are now removed.

- Fixed compilation problems on MacOS X and `OpenCL` 1.0 header files due to undefined an preprocessor constant (thanks to Vlad-Andrei Lazar and Evan Bollig for reporting this)

- Removed the accidental external linkage for three functions (we appreciate the report by Gordon Stevenson).

- New out-of-the-box support for `Eigen` [3] and `MTL 4` [4] libraries. Iterative solvers from ViennaCL can now directly be used with both libraries.

- Fixed a problem with GMRES when system matrix is smaller than the maximum Krylov space dimension.

- Better default parameter for BLAS3 routines leads to higher performance for matrix-matrix-products.

- Added benchmark for dense matrix-matrix products (BLAS3 routines).

- Added viennacl-info example that displays infos about the `OpenCL` backend used by `ViennaCL`.

- Cleaned up CMakeLists.txt in order to selectively enable builds that rely on external libraries.

- More than one installed `OpenCL` platform is now allowed (thanks to Aditya Patel).

Version 1.1.0

A large number of new features and improvements over the 1.0.5 release are now available:

- The completely rewritten `OpenCL` back-end allows for multiple contexts, multiple devices and even to wrap existing OpenCL resources into ViennaCL objects. A tutorial demonstrates the new functionality. Thanks to Josip Basic for pushing us into that direction.

- The tutorials are now named according to their purpose.

- The dense matrix type now supports both row-major and column-major storage.

- Dense and sparse matrix types now now be filled using STL-emulated types (`std:: vector< std::vector<NumericT> >` and `std::vector< std::map< unsigned int, NumericT> >`)

- BLAS level 3 functionality is now complete. We are very happy with the general out-of-the-box performance of matrix-matrix-products, even though it cannot beat the extremely tuned implementations tailored to certain matrix sizes on a particular device yet.

- An automated performance tuning environment allows an optimization of the kernel parameters for the library user's machine. Best parameters can be obtained from a tuning run and stored in a XML file and read at program startup using pugixml.

- Two now preconditioners are now included: A Jacobi preconditioner and a row-scaling preconditioner. In contrast to ILUT, they are applied on the OpenCL device directly.

- Clean compilation of all examples under Visual Studio 2005 (we recommend newer compilers though...).

- Error handling is now carried out using C++ exceptions.

- Matrix Market now uses index base 1 per default (thanks to Evan Bollig for reporting that)

- Improved performance of norm_X kernels.

- Iterative solver tags now have consistent constructors: First argument is the relative tolerance, second argument is the maximum number of total iterations. Other arguments depend on the respective solver.

- A few minor improvements here and there (thanks go to Riccardo Rossi and anonymous sourceforge.net users for reporting the issues)

Version 1.0.x

Version 1.0.5

This is the last 1.0.x release. The main changes are as follows:

- Added a reader and writer for MatrixMarket files (thanks to Evan Bollig for suggesting that)

- Eliminated a bug that caused the upper triangular direct solver to fail on NVIDIA hardware for large matrices (thanks to Andrew Melfi for finding that)

- The number of iterations and the final estimated error can now be obtained from iterative solver tags.

- Improvements provided by Klaus Schnass are included in the developer converter script (OpenCL kernels to C++ header)

- Disabled the use of reference counting for OpenCL handles on Mac OS X (caused seg faults on program exit)

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 and 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

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] "Eigen Library." [Online]. Available: http://eigen.tuxfamily.org/

[4] "MTL 4 Library." [Online]. Available: http://www.mtl4.org/

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

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

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

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

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

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

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

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

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

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

[15] "pugixml." [Online]. Available: http://code.google.com/p/pugixml/