ViennaCL 1.4.2

User Manual

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

Copyright © 2010–2013, Institute for Microelectronics, Institute for Analysis and Scientific Computing, TU Wien. Portions of this software are copyright by UChicago Argonne, LLC.

Project Head:

Karl Rupp

Code Contributors:

Evan Bollig
Alex Christensen (BYU)
Philipp Grabenweger
Volodymyr Kysenko
Nikolay Lukash
Günther Mader
Vittorio Patriarca
Florian Rudolf
Astrid Rupp
Philippe Tillet
Markus Wagner
Josef Weinbub
Michael Wild

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 either a host-based computing backend, an OpenCL computing backend, or CUDA. 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 1.4.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)

A number of preconditioners is provided with ViennaCL 1.4.2 in order to improve convergence of these solvers, cf. Chap. 5.

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 a unified layer to access CUDA [5], OpenCL [6], and/or OpenMP [7] 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 all current CPUs and modern GPUs from NVIDIA and AMD (see Tab. 1), CPUs using either the AMD Accelerated Parallel Processing (APP) SDK (formerly ATI Stream SDK) or the Intel OpenCL SDK, and Intels MIC platform (Xeon Phi).

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 may not be fully OpenCL-certified. Also, we have observed bugs in AMD APP SDKs 2.7 which effects some algorithms in ViennaCL (e.g. BiCGStab).

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 4XX	ok	ok
NVIDIA GTX 5XX	ok	ok
NVIDIA GTX 6XX	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 4XXX	ok	-
ATI Radeon HD 48XX	ok	essentially ok
ATI Radeon HD 5XXX	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 Radeon HD 77XX	ok	-
ATI Radeon HD 78XX	ok	-
ATI Radeon HD 79XX	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 1.4.2`, the Stream SDK (APP SDK) from AMD/ATI may not comply to the `OpenCL` standard for double precision extensions, and we have observed problems with the APP SDK 2.7 on Linux in both single- and double-precision.

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.

To use ViennaCL, only the following minimal prerequisite has to be fulfilled:

- A fairly recent C++ compiler (e.g. GCC version 4.2.x or above and Visual C++ 2005 and 2010 are known to work)

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

- CMake [8] as build system (optional, but highly recommended for building examples)
- OpenCL [6, 9] for accessing compute devices (GPUs); see Section 1.3 for details.
- CUDA [5] for using CUDA-accelerated operations.
- OpenMP [7] for directive-based parallelism on CPUs.
- 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 12), 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. For using the CUDA backend, simply make sure that the CUDA SDK is installed properly. If you wish to use the OpenCL backend, the include and library directories of your OpenCL SDK should also be added there.

If multiple OpenCL libraries are available on the host system, ViennaCL uses the first platform returned by the system. Consult Chap. 8 for configuring the use of other platforms.

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 1.5.2.

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 [6] are also shipped with ViennaCL in the folder CL/.

1.3.2 AMD Accelerated Parallel Processing SDK (formerly Stream SDK)

AMD has provided the OpenCL library with the Accelerated Parallel Processing (APP) SDK [10] previously, now the OpenCL library is also included in the GPU driver. At the release of ViennaCL 1.4.2, the latest version of the SDK is 2.7. 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 [11].

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 may not provide `OpenCL` certified double precision support [12] on some CPUs and GPUs.

Unfortunately, some versions of the AMD APP SDK are known to have bugs. For example, APP SDK 2.7 on Linux causes BiCGStab to fail on some devices.

1.3.3 INTEL OpenCL SDK

`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 Enabling OpenMP, OpenCL, or CUDA Backends

The new default behavior in `ViennaCL` 1.4.0 is to use the CPU backend. `OpenCL` and `CUDA` backends need to be enabled by appropriate preprocessor **`#define`**s.

By default, `ViennaCL` now uses the single-threaded/OpenMP-enabled CPU backend. The `OpenCL` and the `CUDA`-backend need to be enabled explicitly by using preprocessor constants as follows:

Preprocessor `#define`	**Default computing backend**
none	CPU, single-threaded
`VIENNACL_WITH_OPENMP`	CPU with OpenMP (compiler flags required)
`VIENNACL_WITH_OPENCL`	OpenCL
`VIENNACL_WITH_CUDA`	CUDA

The preprocessor constants can be either defined at the beginning of the source file (prior to any ViennaCL-includes), or passed to the compiler as command line argument. For example, on `g++` the respective command line option for enabling the OpenCL backend is `-DVIENNACL_WITH_OPENCL`. Note that CUDA requires the `nvcc` compiler. Furthermore, the use of `OpenMP` usually requires additional compiler flags (on `g++` this is for example `-fopenmp`).

The CUDA backend requires a compilation using `nvcc`.

Multiple backends can be used simultaneously. In such case, `CUDA` has higher priority than

Example/Tutorial	Dependencies
tutorial/amg.cpp	OpenCL, uBLAS
tutorial/bandwidth-reduction.cpp	-
tutorial/blas1.cpp/cu	-
tutorial/blas2.cpp/cu	uBLAS
tutorial/blas3.cpp/cu	uBLAS
tutorial/custom-kernels.cpp	OpenCL
tutorial/custom-context.cpp	OpenCL
tutorial/eigen-with-viennacl.cpp	Eigen
tutorial/fft.cpp	OpenCL
tutorial/iterative.cpp/cu	uBLAS
tutorial/iterative-ublas.cpp	uBLAS
tutorial/iterative-eigen.cpp	Eigen
tutorial/iterative-mtl4.cpp	MTL 4
tutorial/lanczos.cpp/cu	uBLAS
tutorial/least-squares.cpp/cu	uBLAS
tutorial/matrix-range.cpp/cu	uBLAS
tutorial/mtl4-with-viennacl.cpp	MTL 4
tutorial/power-iter.cpp/cu	uBLAS
tutorial/qr.cpp/cu	uBLAS
tutorial/spai.cpp	OpenCL, uBLAS
tutorial/vector-range.cpp/cu	uBLAS
tutorial/viennacl-info.cpp	OpenCL
benchmarks/blas3.cpp/cu	-
benchmarks/opencl.cpp	OpenCL
benchmarks/solver.cpp/cu	uBLAS
benchmarks/sparse.cpp/cu	uBLAS
benchmarks/vector.cpp/cu	-

Table 1.1: Dependencies for the examples in the examples/ folder. Examples using the CUDA-backend use the .cu file extension. Note that all examples can be run using either of the CPU, OpenCL, and CUDA backend unless an explicit OpenCL-dependency is stated.

OpenCL, which has higher priority over the CPU backend when it comes to selecting the default backend.

1.5 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. A brief overview of the most important flags is as follows:

CMake Flag	Purpose
ENABLE_CUDA	Builds examples with the CUDA backend enabled
ENABLE_OPENCL	Builds examples with the OpenCL backend enabled
ENABLE_OPENMP	Builds examples with OpenMP for the CPU backend enabled
ENABLE_EIGEN	Builds examples depending on Eigen
ENABLE_MTL4	Builds examples depending on MTL 4
ENABLE_UBLAS	Builds examples depending on uBLAS

1.5.1 Linux

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

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

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> 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`.

Execute the examples from the `build/` folder as follows:

```
$> examples/tutorial/blas1
$> examples/benchmarks/vectorbench
```

Note that all benchmark executables carry the suffix `bench`.

Use the `CMake`-GUI via `cmake-gui ..` within the `build/` folder in order to enable or disable optional libraries conveniently.

1.5.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the `GCC` compiler the Xcode [13] package has to be installed. To install `CMake` and `Boost` external portation tools have to be used, for example, Fink [14], DarwinPorts [15] or MacPorts [16]. 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.

1.5.3 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).
- If you set ENABLE_CUDA, ENABLE_CUDA, ENABLE_MTL4, or ENABLE_OPENCL and the paths cannot be found, please select the advanced view and provide the required paths manually.
- If you set ENABLE_UBLAS and the paths cannot be found, please select the advanced view and provide the required paths manually. You may have to specify the linker path for Boost manually within your Visual Studio IDE.
- Click again on 'Configure'. You should not receive an error at this point.
- 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.

Part I

Core Functionality

The ViennaCL core consists of operations and algorithms which are available on all three computing backends (CUDA, host-based, OpenCL). These features are considered stable and full support is provided. However, note that performance-characteristics may differ considerably on the different computing backends. In particular, the use of GPUs will not pay off if the data is too small, hence PCI-Express latency is dominant.

Chapter 2

Memory Model

With the support of multiple compute backends, memory buffers need to be managed differently depending on whether CUDA, OpenCL or a plain host-based buffer is in use. These different *memory domains* are abstracted in a class viennacl::backend::mem_handle, which is able to refer to a buffer in all three backends, possibly at the same time. Objects of type mem_handle are the building blocks of scalars, vectors and matrices in ViennaCL, cf. Chap. 3.

The raw handles for each memory domain can be obtained via the member functions cuda_handle(), opencl_handle() and ram_handle(). Note that the former two may not be available if no support for the respective backend is activated using the preprocessor constants VIENNACL_WITH_CUDA and VIENNACL_WITH_OPENCL, cf. Sec. 1.4.

2.1 Memory Handle Operations

Each supported backend is required to support the following functions (arguments omitted for brevity, see reference documentation in doc/doxygen for details):

- memory_create(): Create a memory buffer
- memory_copy(): Copy the (partial) contents of one buffer to another
- memory_write(): Write from a memory location in CPU RAM to the buffer
- memory_read(): Read from the buffer to a memory location in CPU RAM

A common interface layer in viennacl::backend dispatches into the respective routines in the backend for the currently active memory domain of the handle.

2.2 Querying and Switching Active Memory Domains

A mem_handle object creates its buffer according to the following prioritized list, whichever is available: CUDA, OpenCL, host runtime (CPU RAM). The current memory domain can be queried using the member function memory_domain() and returns one of the values MEMORY_NOT_INITIALIZED, MAIN_MEMORY, OPENCL_MEMORY, or CUDA_MEMORY defined in the struct viennacl::memory_types.

The currently active memory handle can be switched from outside using the member function `switch_memory_domain()`. For example, to indicate that the memory referenced by a handle `h`, the line

```
h.switch_active_handle_id(viennacl::MAIN_MEMORY);
```

is sufficient. However, no memory is created, copied, or manipulated when switching the currently active handle, because a `mem_handle` object does not know what the buffer content is referring to and is thus not able to convert data between different memory domains if required.

In order to copy the contents of a memory buffer in one memory domain to a memory buffer in another memory domain within the same `mem_handle`-object, the data type must be supplied. This is accomplished using the function `viennacl::backend::switch_memory_domain (mem_handle, viennacl::memory_types)`, which takes the data type as template argument. Thus, in order to make current data of type **`float`** availabe in CPU RAM for a handle `h`, the function

```
viennacl::backend::switch_memory_domain<float>(h, viennacl::MAIN_MEMORY);
```

is sufficient.

If data should be transferred from one memory handle `h1` to another memory handle `h2`, the function `viennacl::backend::typesafe_memory_copy(h1, h2)` is provided. It takes the data type as template argument and ensures a data conversion between different memory domains if required (e.g. `cl_uint` to **`unsigned int`**).

Chapter 3

Basic Types

This chapter provides a brief overview of the basic interfaces and usage of the provided data types. Operations on the various types are explained in Chapter 4. For full details, refer to the reference pages in the folder `doc/doxygen`.

3.1 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 computing device. `scalar<T>` is designed to behave much like a scalar type on conventional host-based CPU processing, but library users have to keep in mind that every operation on `scalar<T>` may require the launch of an appropriate compute kernel on the GPU, thus making the operation much slower then the conventional CPU equivalent. Even if the host-based computing backend of `ViennaCL` is used, some (small) overheads occur.

Be aware that operations between objects of type `scalar<T>` (e.g. additions. comparisons) have large overhead on GPU backends. A separate compute kernel launch is required for every operation in such case.

3.1.1 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 memory handle (CPU, `CUDA`, or `OpenCL`)

Table 3.1: 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.

3.1.2 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. 3.1.

3.2 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 1.4.2`) 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.

3.2.1 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());
```

```
//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 4 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;
```

3.2.2 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. 3.2 for other member functions.

Accessing single elements of a vector using operator() or operator[] is very slow for GPUs due to PCI-Express latency! 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 with GPU backends!!
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.

Interface	Comment
`CTOR(n)`	Constructor with number of entries
`v(i)`	Access to the i-th element of v (slow for GPUs!)
`v[i]`	Access to the i-th element of v (slow for GPUs!)
`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 memory handle (needed for custom kernels, see Chap. 9)

Table 3.2: Interface of `vector<T>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

3.3 Dense Matrix Type

`matrix<T, F, alignment>` represents a dense matrix with interface listed in Tab. 3.3. The second optional template argument `F` specifies the storage layout and defaults to `row_major`. As an alternative, a `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).

3.3.1 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 on GPU backends! 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 memory handle (needed for custom kernels, see Chap. 9)

Table 3.3: 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. Please refer to Chap. 6 for an overview of other libraries for which an overload of `copy()` is provided.

3.3.2 Members

The members are listed in Tab. 3.3. The usual operator overloads are not listed explicitly

3.4 Sparse Matrix Types

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

In `ViennaCL 1.4.2`, the use of `compressed_matrix` is encouraged over `coordinate_matrix`

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

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 memory handle holding the row indices (needed for custom kernels, see Chap. 9)
`mat.handle2()`	Returns the memory handle holding the column indices (needed for custom kernels, see Chap. 9)
`mat.handle()`	Returns the memory handle holding the entries (needed for custom kernels, see Chap. 9)

Table 3.4: Interface of the sparse matrix type `compressed_matrix<T, F>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

3.4.1.1 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(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. 3.4)

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. 6 for an overview of other libraries for which an overload of `copy()` is provided.

3.4.1.2 Members

The interface is described in Tab. 3.4.

3.4.2 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.

3.4.2.1 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. 3.4.1.1

3.4.2.2 Members

The interface is described in Tab. 3.5.

Note that only a few preconditioners work with `coordinate_matrix` so far, cf. Sec. 5.3.

Interface	Comment
`CTOR(nrows, ncols)`	Constructor with number of rows and columns
`mat.reserve(num)`	Reserve memory for `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 memory handle holding the row and column indices (needed for custom kernels, see Chap. 9)
`mat.handle()`	Returns the memory handle holding the entries (needed for custom kernels, see Chap. 9)

Table 3.5: Interface of the sparse matrix type `coordinate_matrix<T, A>` in `ViennaCL`. Destructors and operator overloads for BLAS are not listed.

3.4.3 ELL Matrix

A sparse matrix in ELL format of type `ell_matrix` is stored in a block of memory of size $N \times n_{\max}$, where N is the number of rows of the matrix and $n_{\max}$ is the maximum number of nonzeros per row. Rows with less than $n_{\max}$ entries are padded with zeros. In a second memory block, the respective column indices are stored.

The ELL format is well suited for matrices where most rows have approximately the same number of nonzeros. This is often the case for matrices arising from the discretization of partial differential equations using e.g. the finite element method. On the other hand, the ELL format introduces substantial overhead if the number of nonzeros per row varies a lot.

For an example use of an `ell_matrix`, have a look at `examples/benchmarks/sparse.cpp`.

Note that preconditioners in Sec. 5.3 do not work with `ell_matrix` yet.

3.4.4 Hybrid Matrix

The higher performance of the ELL format for matrices with approximately the same number of entries per row and the higher flexibility of the CSR format is combined in the `hyb_matrix` type, where the main part of the system matrix is stored in ELL format and excess entries are stored in CSR format.

For an example use of an `hyb_matrix`, have a look at `examples/benchmarks/sparse.cpp`.

Note that preconditioners in Sec. 5.3 do not work with `hyb_matrix` yet.

3.5 Proxies

Similar to `uBLAS`, `ViennaCL` provides `range` and `slice` objects in order to conveniently manipulate dense submatrices and vectors. The functionality is provided in the headers `viennacl/vector_proxy.hpp` and `viennacl/matrix_proxy.hpp` respectively. A range refers to a contiguous integer interval and is set up as

```
std::size_t lower_bound = 1;
std::size_t upper_bound = 7;
viennacl::range r(lower_bound, upper_bound);
```

A slice is similar to a range and allows in addition for arbitrary increments (*stride*). For example, to create a slice consisting of the indices $2, 5, 8, 11, 14$, the code

```
std::size_t start  = 2;
std::size_t stride = 3;
std::size_t size   = 5
viennacl::slice s(start, stride, size);
```

In order to address a subvector of a vector `v` and a submatrix of a matrix `M`, the proxy objects `v_sub` and `M_sub` are created as follows:

```
typedef viennacl::vector<ScalarType>                      VectorType;
typedef viennacl::matrix<ScalarType, viennacl::row_major> MatrixType;

viennacl::vector_range<VCLVectorType> v_sub(v, r);
viennacl::matrix_range<VCLMatrixType> M_sub(M, r, r);
```

As a shortcut, one may use the free function `project()` in order to avoid having to write the type explicitly:

```
project(v, r);    //returns a vector_range as above
project(M, r, r); //returns a matrix_range as above
```

In the same way `vector_slice`s and `matrix_slice`s are set up.

The proxy objects can now be manipulated in the same way as vectors and dense matrices. In particular, operations such as vector proxy additions and matrix additions work as usual, e.g.

```
vcl_sub += vcl_sub; //or project(v, r) += project(v, r);
M_sub += M_sub;     //or project(M, r, r) += project(M, r, r);
```

Submatrix-Submatrix products are computed in the same manner and are handy for many block-based linear algebra algorithms.

Example code can be found in `examples/tutorial/vector-range.cpp` and `examples/tutorial/matrix-range.cpp`

Chapter 4

Basic Operations

The basic types have been introduced in the previous chapter, so we move on with the description of the basic BLAS operations. Almost all operations supported by uBLAS are available, including element-wise operations on vectors. Thus, consider the ublas-documentation as a reference as well.

4.1 Vector-Vector Operations (BLAS Level 1)

ViennaCL provides all vector-vector operations defined at level 1 of BLAS. Tab. 4.1 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/

4.2 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. 4.2.

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

4.3 Matrix-Matrix Operations (BLAS Level 3)

Full BLAS level 3 support is since ViennaCL 1.1.0, cf. Tab. 4.3. 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 show impressive performance gains on mid- to high-end GPUs when compared to a single CPU core.

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;`
elementwise product	$y_i \leftarrow x_i \cdot z_i$	`y = element_prod(x,z);`
elementwise division	$y_i \leftarrow x_i \cdot z_i$	`y = element_div(x,z);`
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 4.1: BLAS level 1 routines mapped to ViennaCL. Note that the free functions reside in namespace `viennacl::linalg`

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. However, highest performance is usually obtained only with careful tuning to the respective target device. Generally, ViennaCL provides kernels that represent a good compromise between efficiency and portability among a large number of different devices and device types.

For certain matrix dimensions, typically multiples of 64 or 128, ViennaCL also provides tuned kernels reaching over 1 TFLOP in single precision (AMD HD 7970).

4.4 Initializer Types

Initializer types in ViennaCL 1.4.2 can only be used for initializing vectors and matrices, not for computations!

In order to initialize vectors, the following initializer types are provided, again similar to uBLAS:

`unit_vector<T>(s, i)`	Unit vector of size s with entry 1 at index i, zero elsewhere.
`zero_vector<T>(s)`	Vector of size s with all entries being zero.
`scalar_vector<T>(s, v)`	Vector of size s with all entries equal to v.

For example, to initialize a vector `v1` with all 42 entries being 42.0, use

```
viennacl::vector<float> v1 = viennacl::scalar_vector<float>(42, 42.0f);
```

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 4.2: BLAS level 2 routines mapped to `ViennaCL`. Note that the free functions reside in namespace `viennacl::linalg`. `tag` is one out of `lower_tag`, `unit_lower_tag`, `upper_tag`, and `unit_upper_tag`.

Similarly the following initializer types are available for matrices:

`identity_matrix<T>(s, i)`	Identity matrix of dimension $s \times s$.
`zero_matrix<T>(s1, s2)`	Matrix of size $s_1 \times s_2$ with all entries being zero.
`scalar_matrix<T>(s1, s2, v)`	Matrix of size $s_1 \times s_2$ with all entries equal to v.

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 4.3: BLAS level 3 routines mapped to `ViennaCL`. Note that the free functions reside in namespace `viennacl::linalg`

Chapter 5

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).

5.1 Direct Solvers

ViennaCL 1.4.2 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 1.4.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());
```

5.2 Iterative Solvers

ViennaCL provides different iterative solvers for various classes of matrices, listed in Tab. 5.1. 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, Eigen and MTL4 objects! Please have a look at Chap. 6 and the respective tutorials in the examples/tutorials/ folder.

In ViennaCL 1.4.2, 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 5.1: 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 $1\mathrm{E}{-10}$ 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);
```

5.3 Preconditioners

`ViennaCL` ships with a generic implementation of several preconditioners. The preconditioner setup is expect for simple diagonal preconditioners always carried out on the CPU host due to the need for dynamically allocating memory. Thus, one may not obtain an overall performance benefit if too much time is spent on the preconditioner setup.

The preconditioner also works for `uBLAS` types!

An overview of preconditioners available for the various sparse matrix types is as follows:

Matrix Type	ICHOL	(Block-)ILU[0/T]	Jacobi	Row-scaling	AMG	SPAI
`compressed_matrix`	yes	yes	yes	yes	yes	yes
`coordinate_matrix`	no	no	yes	yes	no	no
`ell_matrix`	no	no	no	no	no	no
`hyb_matrix`	no	no	no	no	no	no

Broader support of preconditioners particularly for `ell_matrix` and `hyb_matrix` is scheduled for future releases. AMG and SPAI preconditioners are described in Chap. 7.

In the following it is assumed that the sparse linear system of equations is given as follows:

```
typedef viennacl::compressed_matrix<float>   SparseMatrix;

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

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

5.3.1 Incomplete LU Factorization with Threshold (ILUT)

The incomplete LU factorization preconditioner aims at computing sparse matrices lower and upper triangular matrices L and U such that the sparse system matrix is approximately given by $A \approx LU$. In order to control the sparsity pattern of L and U, a threshold strategy is used (ILUT) [1]. Due to the serial nature of the preconditioner, the setup of ILUT is always computed on the CPU using the respective ViennaCL backend.

```
//compute ILUT preconditioner:
viennacl::linalg::ilut_tag ilut_config;
viennacl::linalg::ilut_precond< SparseMatrix > vcl_ilut(vcl_matrix,
                                                        ilut_config);

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

The triangular substitutions may be applied in parallel on GPUs by enabling *level-scheduling* [1] via the member function call `use_level_scheduling(true)` in the `ilut_config` object.

Three parameters can be passed to the constructor of `ilut_tag`: The first specifies the maximum number of entries per row in L and U, while the second parameter specifies the drop tolerance. The third parameter is the boolean specifying whether level scheduling should be used.

The performance of level scheduling depends strongly on the matrix pattern and is thus disabled by default.

5.3.2 Incomplete LU Factorization with Static Pattern (ILU0)

Similar to ILUT, ILU0 computes an approximate LU factorization with sparse factors L and U. While ILUT determines the location of nonzero entries on the fly, ILU0 uses the sparsity pattern of A for the sparsity pattern of L and U [1]. Due to the serial nature of the preconditioner, the setup of ILU0 is computed on the CPU.

```
//compute ILU0 preconditioner:
viennacl::linalg::ilu0_tag ilu0_config;
viennacl::linalg::ilu0_precond< SparseMatrix > vcl_ilut(vcl_matrix,
                                                        ilu0_config);

//solve (e.g. using conjugate gradient solver)
```

```
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::bicgstab_tag(),
                                     vcl_ilut);   //preconditioner here
```

The triangular substitutions may be applied in parallel on GPUs by enabling *level-scheduling* [1] via the member function call `use_level_scheduling(true)` in the `ilu0_config` object.

One parameter can be passed to the constructor of `ilu0_tag`, being the boolean specifying whether level scheduling should be used.

The performance of level scheduling depends strongly on the matrix pattern and is thus disabled by default.

5.3.3 Block-ILU

To overcome the serial nature of ILUT and ILU0 applied to the full system matrix, a parallel variant is to apply ILU to diagonal blocks of the system matrix. This is accomplished by the `block_ilu` preconditioner, which takes the system matrix type as first template argument and the respective ILU-tag type as second template argument (either `ilut_tag` or `ilu0_tag`). Support for accelerators using `CUDA` or `OpenCL` is provided.

```
//compute block-ILU preconditioner using ILU0 for each block:
block_ilu_precond<SparseMatrix,
                  ilu0_tag> vcl_block_ilu0(vcl_matrix,
                                           ilu0_tag());

//solve
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::bicgstab_tag(),
                                     vcl_block_ilu0);
```

A third argument can be passed to the constructor of `block_ilu_precond`: Either the number of blocks to be used (defaults to 8), or an index vector with fine-grained control over the blocks. Refer to the Doxygen pages in doc/doxygen for details.

The number of blocks is a design parameter for your sparse linear system at hand. Higher number of blocks leads to better memory bandwidth utilization on GPUs, but may increase the number of solver iterations.

5.3.4 Jacobi Preconditioner

A Jacobi preconditioner is a simple diagonal preconditioner given by the reciprocals of the diagonal entries of the system matrix A. Use the preconditioner as follows:

```
//compute Jacobi preconditioner:
jacobi_precond< SparseMatrix > vcl_jacobi(vcl_matrix,
                                          viennacl::linalg::jacobi_tag());
```

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

5.3.5 Row Scaling Preconditioner

A row scaling preconditioner is a simple diagonal preconditioner given by the reciprocals of the norms of the rows of the system matrix A. Use the preconditioner as follows:

```
//compute row scaling 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_row_scaling);
```

The tag `viennacl::linalg::row_scaling_tag()` can be supplied with a parameter denoting the norm to be used. A value of `1` specifies the l^1-norm, while a value of `2` selects the l^2-norm (default).

5.4 Eigenvalue Computations

Two algorithms for the computations of the eigenvalues of a matrix A are implemented in `ViennaCL`:

- The Power Iteration [17]
- The Lanczos Algorithm [18]

Depending on the parameter `tag` either one of them is called. Both algorithms can be used for either `uBLAS` or `ViennaCL` compressed matrices.
In order to get the eigenvalue with the greatest absolut value the power iteration should be called.
The Lanczos algorithm returns a vector of the largest eigenvalues with the same type as the entries of the matrix.

The algorithms are called for a matrix object `A` by

```
std::vector<double> largest_eigenvalues = viennacl::linalg::eig(A, ltag);
double largest_eigenvalue = viennacl::linalg::eig(A, ptag);
```

5.4.1 Power Iteration

The Power iteration aims at computing the eigenvalues of a matrix by calculating the product of the matrix and a vector for several times, where the resulting vector is used for the

next product of the matrix and so on. The computation stops as soon as the norm of the vector converges.
The final vector is the eigenvector to the eigenvalue with the greatest absolut value.
To call this algorithm, `piter_tag` must be used. This tag has only one parameter: `terminationfactor` defines the accuracy of the computation, i.e. if the new norm of the eigenvector changes less than this parameter the computation stops and returns the corresponding eigenvalue (default: $1e-10$).
The call of the constructor may look like the following:

```
viennacl::linalg::piter_tag ptag(1e-8);
```

Example code can be found in `examples/tutorial/power-iter.cpp`.

5.4.2 The Lanczos Algorithm

In order to compute the eigenvalues of a sparse high-dimensional matrix the Lanczos algorithm can be used to find these. This algorithm reformulates the given high-dimensional matrix in a way such that the matrix can be rewritten in a tridiagonal matrix at much lower dimension. The eigenvalues of this tridiagonal matrix are equal to the largest eigenvalues of the original matrix.
The eigenvalues of the tridiagonal matrix are calculated by using the bisection method [17].
To call this Lanczos algorithm, `lanczos_tag` must be used. This tag has several parameters that can be passed to the constructor:

- The exponent of epsilon for the tolerance of the reorthogonalization, defined by the parameter `factor` (default: 0.75)
- The method of the Lanczos algorithm: 0 uses partial reorthogonalization, 1 full reothogonalization and 2 does not use reorthogonalization (default: 0)
- The number of eigenvalues that are returned is specified by `num_eigenvalues` (default: 10)
- The size of the krylov space used for the computations can be set by the parameter `krylov_size` (default: 100). The maximum number of iterations can be equal or less this parameter

The call of the constructor may look like the following:

```
viennacl::linalg::lanczos_tag ltag(0.85, 15, 0, 200);
```

Example code can be found in `examples/tutorial/lanczos.cpp`.

5.5 QR Factorization

The current QR factorization implementation depends on `uBLAS`.

A matrix $A \in \mathbb{R}^{n \times m}$ can be factored into $A = QR$, where $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix and $R \in \mathbb{R}^{n \times m}$ is upper triangular. This so-called QR-factorization is important for eigenvalue computations as well as for the solution of least-squares problems [17]. `ViennaCL` provides a generic implementation of the QR-factorization using Householder reflections in file `viennacl/linalg/qr.hpp`. An example application can be found in `examples/tutorial/qr.hpp`.

The Householder reflectors v_i defining the Householder reflection $I - \beta_i v_i v_i^{\mathrm{T}}$ are stored in the columns below the diagonal of the input matrix A [17]. The normalization coefficients β_i are returned by the worker function `inplace_qr`. The upper triangular matrix R is directly written to the upper triangular part of A.

```
std::vector<ScalarType> betas = viennacl::linalg::inplace_qr(A, 12);
```

If A is a dense matrix from `uBLAS`, the calculation is carried out on the CPU using a single thread. If A is a `viennacl::matrix`, a hybrid implementation is used: The panel factorization is carried out using `uBLAS`, while expensive BLAS level 3 operations are computed on the OpenCL device using multiple threads.

Typically, the orthogonal matrix Q is kept in inplicit form because of computational efficiency However, if Q and R have to be computed explicitly, the function `recoverQ` can be used:

```
viennacl::linalg::recoverQ(A, betas, Q, R);
```

Here, `A` is the inplace QR-factored matrix, `betas` are the coefficients of the Householder reflectors as returned by `inplace_qr`, while `Q` and `R` are the destination matrices. However, the explicit formation of Q is expensive and is usually avoided. For a number of applications of the QR factorization it is required to apply Q^T to a vector b. This is accomplished by

```
viennacl::linalg::inplace_qr_apply_trans_Q(A, betas, b);
```

without setting up Q (or Q^T) explicitly.

Have a look at `examples/tutorial/least-squares.cpp` for a least-squares computation using QR factorizations.

Chapter 6

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_WITH_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.

6.1 Boost.uBLAS

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

```
//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_WITH_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.

6.2 Eigen

To copy data from Eigen [3] objects (version 3.x.y) to ViennaCL, the copy()-functions are used just as for uBLAS and STL types:

```
//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

```
//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:

```
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_WITH_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.

6.3 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_WITH_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.

Part II

Addon Functionality

With the introduction of host-based, CUDA- and OpenCL-enabled computing backends in ViennaCL 1.4.0, certain functionality is not available for all three backends and listed in the following. For example, the OpenCL kernel generator makes sense in the OpenCL computing backend, thus this functionality is moved out of the set of core functionality.

Also, certain functionality is still in experimental stage and might experience interface changes. Although all functionality flagged as experimental and listed in this section passes a respective set of tests, library users are advised to use them with extra care and be prepared for interface changes when upgrading to a newer version of ViennaCL.

Chapter 7

Additional Algorithms

The following algorithms are still not yet mature enough to be considered core-functionality, and/or are available with the `OpenCL` backend only.

7.1 Additional Iterative Solvers

The following iterative solvers are only available on selected computing backends.

7.1.1 Mixed-Precision Conjugate Gradients

A two-stage mixed-precision CG algorithm is available as follows:

```
viennacl::linalg::mixed_precision_cg_tag   mixed_prec_cg_config;
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     mixed_prec_cg_config);
```

As usual, the first parameter to the constructor of `mixed_precision_cg_tag` is the relative tolerance for the residual, while the second parameter denotes the maximum number of solver iterations. The third parameter denotes the relative tolerance for the inner low-precision CG iterations and defaults to 0.01.

Have a look at `examples/banchmarks/solver.cpp` for an example.

A mixed-precision solver makes sense only if the matrix and right-hand-side vector are supplied in **`double`** precision.

The mixed-precision solver is currently available with the `OpenCL` compute backend only.

Description	ViennaCL option constant
Classical Ruge-Stüben (RS)	VIENNACL_AMG_COARSE_RS
One-Pass	VIENNACL_AMG_COARSE_ONEPASS
RS0	VIENNACL_AMG_COARSE_RS0
RS3	VIENNACL_AMG_COARSE_RS3
Aggregation	VIENNACL_AMG_COARSE_AG
Smoothed aggregation	VIENNACL_AMG_COARSE_SA

Table 7.1: AMG coarsening methods available in ViennaCL. Per default, classical RS coarsening is used.

Description	ViennaCL option constant
Direct	VIENNACL_AMG_INTERPOL_DIRECT
Classic	VIENNACL_AMG_INTERPOL_ONEPASS
RS0 coarsening	VIENNACL_AMG_INTERPOL_RS0
RS3 coarsening	VIENNACL_AMG_INTERPOL_RS3

Table 7.2: AMG interpolation methods available in ViennaCL. Per default, direct interpolation is used.

7.2 Additional Preconditioners

In addition to the preconditioners discussed in Sec. 5.3, two more preconditioners are available with the OpenCL backend and are described in the following.

7.2.1 Algebraic Multigrid

Algebraic Multigrid preconditioners are only available with the OpenCL backend and are experimental in ViennaCL 1.4.2. Interface changes as well as considerable performance improvements may be included in future releases!

Algebraic Multigrid preconditioners depend on uBLAS.

Algebraic multigrid mimics the behavior of geometric multigrid on the algebraic level and is thus suited for black-box purposes, where only the system matrix and the right hand side vector are available [19]. Many different flavors of the individual multigrid ingredients exists [20], of which the most common ones are implemented in ViennaCL.

The two main ingredients of algebraic multigrid are a coarsening algorithm and an interpolation algorithm. The available coarsening methods are listed in Tab. 7.1. The available interpolation methods are given in Tab. 7.2. In addition, the following parameters can be controlled in the amg_tag and can be passed to the constructor:

- Strength of dependence threshold (default: 0.25)
- Interpolation weight (default: 1)

- Jacobi smoother weight (default: 1)
- Number of pre-smoothing steps (default: 1)
- Number of post-smoothing steps (default: 1)
- Number of coarse levels

Note that the efficiency of the various AMG flavors are typically highly problem-specific. Therefore, failure of one method for a particular problem does NOT imply that other coarsening or interpolation strategies will fail as well.

7.2.2 Sparse Approximate Inverses

Sparse Approximate Inverse preconditioners are only available with the `OpenCL` backend and are experimental in `ViennaCL 1.4.2`. Interface changes as well as considerable performance improvements may be included in future releases!

Sparse Approximate Inverse preconditioners depend on `uBLAS`.

An alternative construction of a preconditioner for a sparse system matrix A is to compute a matrix M with a prescribed sparsity pattern such that

$$\|AM - I\|_F \rightarrow \min\,, \tag{7.1}$$

where $\|\cdot\|_F$ denotes the Frobenius norm. This is the basic idea of sparse approximate inverse (SPAI) preconditioner. It becomes increasingly attractive because of their inherent high degree of parallelism, since the minimization problem can be solved independently for each column of M. `ViennaCL` provides two preconditioners of this family: The first is the classical SPAI algorithm as described by Grote and Huckle [21], the second is the factored SPAI (FSPAI) for symmetric matrices as proposed by Huckle [22].

SPAI can be employed for a CPU matrix `M` of type `MatrixType` as follows:

```
// setup SPAI preconditioner, purely CPU-based
viennacl::linalg::spai_precond<MatrixType>
  spai_cpu(M, viennacl::linalg::spai_tag(1e-3, 3, 5e-2));

//solve (e.g. using stab. Bi-conjugate gradient solver)
vcl_result = viennacl::linalg::solve(M,
                                     rhs,
                                     viennacl::linalg::bicgstab_tag(),
                                     spai_cpu);
```

The first parameter denotes the residual norm threshold for the full matrix, the second parameter the maximum number of pattern updates, and the third parameter is the threshold for the residual of each minimization problem.

For GPU-matrices, only parts of the setup phase are computed on the CPU, because compute-intensive tasks can be carried out on the GPU:

```
// setup SPAI preconditioner, GPU-assisted
viennacl::linalg::spai_precond<GPUMatrixType>
  spai_gpu(vcl_matrix, viennacl::linalg::spai_tag(1e-3, 3, 5e-2));

//solve (e.g. using conjugate gradient solver)
vcl_result = viennacl::linalg::solve(vcl_matrix,
                                     vcl_rhs,
                                     viennacl::linalg::bicgstab_tag(),
                                     spai_gpu);
```

The `GPUMatrixType` is typically a `viennacl::compressed_matrix` type.

For symmetric matrices, FSPAI can be used with the conjugate gradient solver:

```
viennacl::linalg::fspai_precond<MatrixType> fspai_cpu(M, viennacl::linalg::
    fspai_tag());

//solve (e.g. using stab. Bi-conjugate gradient solver)
vcl_result = viennacl::linalg::solve(M,
                                     rhs,
                                     viennacl::linalg::cg_tag(),
                                     fspai_cpu);
```

Our experience is that FSPAI is typically more efficient than SPAI when applied to the same matrix, both in computational effort and in terms of convergence acceleration of the iterative solvers.

At present, there is no GPU-accelerated FSPAI included in `ViennaCL`.

Note that FSPAI depends on the ordering of the unknowns, thus bandwidth reduction algorithms may be employed first, cf. Sec. 7.4.

7.3 Fast Fourier Transform

The fast Fourier transform is experimental in `ViennaCL 1.4.2` and available with the `OpenCL` backend only. Interface changes as well as considerable performance improvements may be included in future releases!

Since there is no standardized complex type in `OpenCL` at the time of the release of `ViennaCL 1.4.2`, vectors need to be set up with real- and imaginary part before computing a fast Fourier tranform (FFT). In order to store complex numbers z_0, z_1, etc. in a `viennacl::vector`, say `v`, the real and imaginary parts are mapped to even and odd entries of `v` respectively: `v[0] = Real(z_0), v[1] = Imag(z_0), v[2] = Real(z_1), v[3] = Imag(z_1)`, etc.

The FFT of `v` can then be computed either by writing to a second vector `output` or by directly writing the result to `v`

```
viennacl::fft(v, output);
viennacl::inplace_fft(v);
```

Conversely, the inverse FFT is computed as

```
viennacl::ifft(v, output);
viennacl::inplace_ifft(v);
```

In ViennaCL 1.4.2 the FFT with complexity $N \log N$ is computed for vectors with a size of a power of two only. For other vector sizes, a standard discrete Fourier transform with complexity N^2 is employed. This is subject to change in future versions.

7.4 Bandwidth Reduction

Bandwidth reduction algorithms are experimental in ViennaCL 1.4.2. Interface changes as well as considerable performance improvements may be included in future releases!

The bandwidth of a sparse matrix is defined as the maximum difference of the indices of nonzero entries in a row, taken over all rows. A low bandwidth typically allows for the use of efficient banded matrix solvers instead of iterative solvers. Moreover, better cache utilization as well as lower fill-in in LU-factorization based algorithms can be expected.

For a given sparse matrix with large bandwidth, ViennaCL provides routines for renumbering the unknowns such that the reordered system matrix shows much smaller bandwidth. Typical applications stem from the discretization of partial differential equations by means of the finite element or the finite difference method. The algorithms employed are as follows:

- Classical Cuthill-McKee algorithm [23]
- Modified Cuthill-McKee algorithm [23]
- Gibbs-Poole-Stockmeyer algorithm, cf. [24]

The modified Cuthill-McKee algorithm also takes nodes with small, but not necessarily minimal degree as root node into account and may lead to better results than the classical Cuthill-McKee algorithm. A parameter $a \in [0, 1]$ controls the number of nodes considered: All nodes with degree d fulfilling

$$d_{\min} \leq d \leq d_{\min} + a(d_{\max} - d_{\min})$$

are considered, where $d_{\min}$ and $d_{\max}$ are the miminum and maximum nodal degrees in the graph. A second parameter gmax specifies the number of additional root nodes considered.

The algorithms are called for a matrix of a type compatible with std::vector< std::map <int, double> > by

```
r = viennacl::reorder(matrix, viennacl::cuthill_mckee_tag());
r = viennacl::reorder(matrix,
                      viennacl::advanced_cuthill_mckee_tag(a, gmax));
r = viennacl::reorder(matrix, viennacl::gibbs_poole_stockmeyer_tag());
```

and return the permutation array. In `ViennaCL 1.4.2`, the user then needs to manually reorder the sparse matrix based on the permutation array. Example code can be found in `examples/tutorial/bandwidth-reduction.cpp`.

7.5 Nonnegative Matrix Factorization

Nonnegative Matrix Factorization is experimental in `ViennaCL 1.4.2` and available with the `OpenCL` backend only. Interface changes as well as considerable performance improvements may be included in future releases!

In various fields such as text mining, a matrix V needs to be factored into factors W and H such that the function

$$f(W, H) = \|V - WH\|_{\mathrm{F}}^2$$

is minimized. The algorithm proposed by Lee and Seoung [25] is available in ViennaCL in the header file `viennacl/linalg/nmf.hpp` as

```
viennacl::matrix<ScalarType> V(size1, size2);
viennacl::matrix<ScalarType> W(size1, k);
viennacl::matrix<ScalarType> H(k, size2);

viennacl::linalg::nmf_config conf;
viennacl::linalg::nmf(v_ref, w_nmf, h_nmf, conf);
```

For an overview of the parameters (tolerances) of the configuration object `conf`, please refer to the Doxygen documentation in `doc/doxygen/`.

Chapter 8

Configuring OpenCL 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 OpenCL devices and contexts was introduced in ViennaCL with version 1.1.0. In the following we give a description of the provided functionality.

In ViennaCL 1.4.2 there is no native support for automatically executing operations over multiple GPUs. Partition of data is left to the user.

8.1 Context Setup

Unless specified otherwise (see Chap. 10), 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`). ViennaCL uses the first platform returned by the OpenCL backend for the context. If a different platform should be used on a machine with multiple platforms available, this can be achieved with

```
viennacl::ocl::set_context_platform_index(id, platform_index);
```

where the context ID is `id` and `platform_index` refers to the array index of the platform as returned by `clGetPlatformIDs()`.

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 OpenCL 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.

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 10.

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.

8.2 Switching Contexts and Devices

ViennaCL always uses the currently active OpenCL 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.

8.3 Setting OpenCL Compiler Flags

Each `OpenCL` context provides a member function `.build_options()`, which can be used to pass OpenCL compiler flags prior to compilation. Note that flags need to be passed to the context prior to the compilation of the respective kernels, i.e. prior the first instantiation of the respective matrix or vector types.

To pass the `-cl-mad-enable` flag to the current context, the line

```
viennacl::ocl::current_context().build_options("-cl-mad-enable");
```

is sufficient. Confer to the `OpenCL` standard for a full list of flags.

Chapter 9

Custom OpenCL 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.

9.1 Setting up the OpenCL 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_program =
"__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 abd 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 [6].

9.2 Compilation of the OpenCL 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 =
  viennacl::ocl::current_context().add_program(my_compute_program,
                                               "my_compute_program");
```

The next step is to register the kernel `elementwise_prod` included in the compiled program:

```
my_prog.add_kernel("elementwise_prod");
```

Similarly, one proceeds with other kernels in the compiled program. The next step is to extract the kernel object `my_kernel` from the compiled program:

```
viennacl::ocl::kernel & my_kernel = my_prog.get_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. Therefore, first allocate the required `ViennaCL` objects and compile/add all custom kernels, before you start taking references to custom programs or kernels.

Instead of extracting references to programs and kernels directly at program 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");
```

This simplifies application development considerably, since no program and kernel objects need to be passed around.

9.3 Launching the OpenCL 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 [6]). 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 10

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

10.1 Passing Contexts to ViennaCL

ViennaCL 1.4.2 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 8.

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.

`ViennaCL` does not destroy the provided context automatically upon exit. The user should thus call `clReleaseContext()` as usual for destroying the context.

10.2 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 11

Automated OpenCL User-Kernel Generation

While ViennaCL provides a convenient means of including custom OpenCL compute kernels, cf. Chap. 9, it can be rather tedious to come up with a good compute kernel, or to come up with many similar kernels differing in small details only. For the case of BLAS level 1 and level 2 operations, ViennaCL now provides an automated kernel generator, which takes a high-level specification of the operations and create one or more suitable OpenCL kernels. This allows for high-performance implementations of algorithms which may otherwise lead to spurious temporary objects.

Consider the operation

$$\mathbf{x} = \mathbf{A} \times \left[(\mathbf{y} \cdot (\mathbf{y} + \mathbf{z}))\mathbf{y} + \mathbf{z} \right] ,$$

where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ denote vectors, $\mathbf{A}$ is a dense matrix, and the dot denotes the vector dot product. With the generator it is sufficient to write the following C++ code in order to obtain an OpenCL kernel:

```
// Instantiation of the symbolic variables
symbolic_vector<NumericT, 0> sX;
symbolic_matrix<NumericT, 1> sA;
symbolic_vector<NumericT, 2> sY;
symbolic_vector<NumericT, 3> sZ;

//Creation of the custom operation
custom_operation my_op( sX = prod(sA, inner_prod(sY, sY+sZ) * sY + sZ),
                        "operation_name" );
```

where NumericT is either **float** or **double**. The string provided as second parameter is required and can be used to identify, manage and retrieve different kernels. No two custom_operations are allowed to be identified using the same string.

The custom operation object my_op can be enqueued like any other kernel:

```
//Execution of the custom operation
viennacl::ocl::enqueue(my_op(x,A,y,z));
```

Here, x, y, z are of type viennacl::vector<NumericT> and A is of type viennacl::matrix <NumericT>.

Sample code can be found in `tests/src/generator_*.cpp`

The kernel generator is still experimental, yet already able to generate rather complex compute kernels.

Chapter 12

OpenCL 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 number of work groups and work items per work group (128) for each compute kernel. To obtain highest performance, optimal work sizes have to be determined for each kernel in dependence of the underlying device.

12.1 Start Tuning Runs

ViennaCL 1.4.2 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.

12.2 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

```
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` [26], which is shipped with `ViennaCL` in `external/`

Chapter 13

Structured Matrix Types

Structured matrix types are experimental in ViennaCL 1.4.2. Interface changes as well as considerable performance improvements may be included in future releases!

There are a number of structured dense matrices for which some algorithms such as matrix-vector products can be computed with much lower computational effort than for the general dense matrix case. In the following, four structured dense matrix types included in ViennaCL are discussed. Example code can be found in examples/tutorial/structured-matrices.cpp.

13.1 Circulant Matrix

A circulant matrix is a matrix of the form

$$\begin{pmatrix} c_0 & c_{n-1} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ c_{n-2} & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \end{pmatrix}$$

and available in ViennaCL via

```
#include "viennacl/circulant_matrix.hpp"

std::size_t s = 42;
viennacl::circulant_matrix circ_mat(s, s);
```

The circulant_matrix type can be manipulated in the same way as the dense matrix type matrix. Note that writing to a single element of the matrix is structure-preserving, e.g. changing circ_mat(1,2) will automatically update circ_mat(0,1), circ_mat(2,3) and so on.

13.2 Hankel Matrix

A Hankel matrix is a matrix of the form

$$\begin{pmatrix} a & b & c & d \\ b & c & d & e \\ c & d & e & f \\ d & e & f & g \end{pmatrix}$$

and available in ViennaCL via

```
#include "viennacl/hankel_matrix.hpp"

std::size_t s = 42;
viennacl::hankel_matrix hank_mat(s, s);
```

The `hankel_matrix` type can be manipulated in the same way as the dense matrix type `matrix`. Note that writing to a single element of the matrix is structure-preserving, e.g. changing `hank_mat(1,2)` in the example above will also update `hank_mat(0,3)`, `hank_mat(2,1)` and `hank_mat(3,0)`.

13.3 Toeplitz Matrix

A Toeplitz matrix is a matrix of the form

$$\begin{pmatrix} a & b & c & d \\ e & a & b & c \\ f & e & a & b \\ g & f & e & a \end{pmatrix}$$

and available in ViennaCL via

```
#include "viennacl/toeplitz_matrix.hpp"

std::size_t s = 42;
viennacl::toeplitz_matrix toep_mat(s, s);
```

The `toeplitz_matrix` type can be manipulated in the same way as the dense matrix type `matrix`. Note that writing to a single element of the matrix is structure-preserving, e.g. changing `toep_mat(1,2)` in the example above will also update `toep_mat(0,1)` and `toep_mat(2,3)`.

13.4 Vandermonde Matrix

A Vandermonde matrix is a matrix of the form

$$\begin{pmatrix} 1 & \alpha_1 & \alpha_1^2 & \dots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \alpha_2^2 & \dots & \alpha_2^{n-1} \\ 1 & \vdots & \vdots & \vdots & \\ 1 & \alpha_m & \alpha_m^2 & \dots & \alpha_m^{n-1} \end{pmatrix}$$

and available in ViennaCL via

```
#include "viennacl/vandermonde_matrix.hpp"

std::size_t s = 42;
viennacl::vandermonde_matrix vand_mat(s, s);
```

The `vandermonde_matrix` type can be manipulated in the same way as the dense matrix type `matrix`, but restrictions apply. For example, the addition or subtraction of two Vandermonde matrices does not yield another Vandermonde matrix. Note that writing to a single element of the matrix is structure-preserving, e.g. changing `vand_mat(1,2)` in the example above will automatically update `vand_mat(1,3)`, `vand_mat(1,4)`, etc.

Part III

Miscellaneous

Chapter 14

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.

14.1 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!

14.2 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.

14.3 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.

14.4 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.

14.5 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 a lazy manner when requesting OpenCL kernels. Kernels with similar functionality are grouped together in a common compilation units. 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.

Appendices

Appendix A

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!

Appendix B

Change Logs

Version 1.4.x

Version 1.4.2

This is a maintenance release, particularly resolving compilation problems with Visual Studio 2012.

- Largely refactored the internal code base, unifying code for `vector`, `vector_range`, and `vector_slice`. Similar code refactoring was applied to `matrix`, `matrix_range`, and `matrix_slice`. This not only resolves the problems in VS 2012, but also leads to shorter compilation times and a smaller code base.
- Improved performance of matrix-vector products of `compressed_matrix` on CPUs using OpenCL.
- Resolved a bug which shows up if certain rows and columns of a `compressed_matrix` are empty and the matrix is copied back to host.
- Fixed a bug and improved performance of GMRES. Thanks to Ivan Komarov for reporting via sourceforge.
- Added additional Doxygen documentation.

Version 1.4.1

This release focuses on improved stability and performance on AMD devices rather than introducing new features:

- Included fast matrix-matrix multiplication kernel for AMD's Tahiti GPUs if matrix dimensions are a multiple of 128. Our sample HD7970 reaches over 1.3 TFLOPs in single precision and 200 GFLOPs in double precision (counting multiplications and additions as separate operations).
- All benchmark FLOPs are now using the common convention of counting multiplications and additions separately (ignoring fused multiply-add).
- Fixed a bug for matrix-matrix multiplication with `matrix_slice<>` when slice dimensions are multiples of 64.

- Improved detection logic for Intel OpenCL SDK.
- Fixed issues when resizing an empty `compressed_matrix`.
- Fixes and improved support for BLAS-1-type operations on dense matrices and vectors.
- Vector expressions can now be passed to `inner_prod()` and `norm_1()`, `norm_2()` and `norm_inf()` directly.
- Improved performance when using OpenMP.
- Better support for Intel Xeon Phi (MIC).
- Resolved problems when using OpenCL for CPUs if the number of cores is not a power of 2.
- Fixed a flaw when using AMG in debug mode. Thanks to Jakub Pola for reporting.
- Removed accidental external linkage (invalidating header-only model) of SPAI-related functions. Thanks again to Jakub Pola.
- Fixed issues with copy back to host when OpenCL handles are passed to CTORs of vector, matrix, or `compressed_matrix`. Thanks again to Jakub Pola.
- Added fix for segfaults on program exit when providing custom OpenCL queues. Thanks to Denis Demidov for reporting.
- Fixed bug in `copy()` to `hyb_matrix` as reported by Denis Demidov (thanks!).
- Added an overload for `result_of::alignment` for `vector_expression`. Thanks again to Denis Demidov.
- Added SSE-enabled code contributed by Alex Christensen.

Version 1.4.0

The transition from 1.3.x to 1.4.x features the largest number of additions, improvements, and cleanups since the initial release. In particular, host-, OpenCL-, and CUDA-based execution is now supported. OpenCL now needs to be enabled explicitly! New features and feature improvements are as follows:

- Added host-based and CUDA-enabled operations on ViennaCL objects. The default is now a host-based execution for reasons of compatibility. Enable OpenCL- or CUDA-based execution by defining the preprocessor constant `VIENNACL_WITH_OPENCL` and `VIENNACL_WITH_CUDA` respectively. Note that CUDA-based execution requires the use of nvcc.
- Added mixed-precision CG solver (OpenCL-based).
- Greatly improved performance of ILU0 and ILUT preconditioners (up to 10-fold). Also fixed a bug in ILUT.
- Added initializer types from Boost.uBLAS (`unit_vector`, `zero_vector`, `scalar_vector`, `identity_matrix`, `zero_matrix`, `scalar_matrix`). Thanks to Karsten Ahnert for suggesting the feature.

- Added incomplete Cholesky factorization preconditioner.
- Added element-wise operations for vectors as available in Boost.uBLAS (`element_prod` , `element_div`).
- Added restart-after-N-cycles option to BiCGStab.
- Added level-scheduling for ILU-preconditioners. Performance strongly depends on matrix pattern.
- Added least-squares example including a function `inplace_qr_apply_trans_Q()` to compute the right hand side vector $Q^T b$ without rebuilding Q.
- Improved performance of LU-factorization of dense matrices.
- Improved dense matrix-vector multiplication performance (thanks to Philippe Tillet).
- Reduced overhead when copying to/from `ublas::compressed_matrix`.
- ViennaCL objects (scalar, vector, etc.) can now be used as global variables (thanks to an anonymous user on the support-mailinglist).
- Refurbished OpenCL vector kernels backend. All operations of the type v1 = a v2 @ b v3 with vectors v1, v2, v3 and scalars a and b including += and -= instead of = are now temporary-free. Similarly for matrices.
- `matrix_range` and `matrix_slice` as well as `vector_range` and `vector_slice` can now be used and mixed completely seamlessly with all standard operations except `lu_factorize()`.
- Fixed a bug when using copy() with iterators on vector proxy objects.
- Final reduction step in `inner_prod()` and norms is now computed on CPU if the result is a CPU scalar.
- Reduced kernel launch overhead of simple vector kernels by packing multiple kernel arguments together.
- Updated SVD code and added routines for the computation of symmetric eigenvalues using OpenCL.
- `custom_operation`'s constructor now support multiple arguments, allowing multiple expression to be packed in the same kernel for improved performances. However, all the datastructures in the multiple operations must have the same size.
- Further improvements to the OpenCL kernel generator: Added a repeat feature for generating loops inside a kernel, added element-wise products and division, added support for every one-argument OpenCL function.
- The name of the operation is now a mandatory argument of the constructor of `custom_operation` .
- Improved performances of the generated matrix-vector product code.
- Updated interfacing code for the Eigen library, now working with Eigen 3.x.y.
- Converter in source-release now depends on Boost.filesystem3 instead of Boost.filesystem2, thus requiring Boost 1.44 or above.

Version 1.3.x

Version 1.3.1

The following bugfixes and enhancements have been applied:

- Fixed a compilation problem with GCC 4.7 caused by the wrong order of function declarations. Also removed unnecessary indirections and unused variables.
- Improved out-of-source build in the src-version (for packagers).
- Added virtual destructor in the `runtime_wrapper`-class in the kernel generator.
- Extended flexibility of submatrix and subvector proxies (ranges, slices).
- Block-ILU for `compressed_matrix` is now applied on the GPU during the solver cycle phase. However, for the moment the implementation file in `viennacl/linalg/detail/ilu/opencl_block_ilu.hpp` needs to be included separately in order to avoid an OpenCL dependency for all ILU implementations.
- SVD now supports double precision.
- Slighly adjusted the interface for NMF. The approximation rank is now specified by the supplied matrices W and H.
- Fixed a problem with matrix-matrix products if the result matrix is not initialized properly (thanks to Laszlo Marak for finding the issue and a fix).
- The operations $C+ = prod(A, B)$ and $C- = prod(A, B)$ for matrices A, B, and C no longer introduce temporaries if the three matrices are distinct.

Version 1.3.0

Several new features enter this new minor version release. Some of the experimental features introduced in 1.2.0 keep their experimental state in 1.3.x due to the short time since 1.2.0, with exceptions listed below along with the new features:

- Full support for ranges and slices for dense matrices and vectors (no longer experimental)
- QR factorization now possible for arbitrary matrix sizes (no longer experimental)
- Further improved matrix-matrix multiplication performance for matrix dimensions which are a multiple of 64 (particularly improves performance for NVIDIA GPUs)
- Added Lanczos and power iteration method for eigenvalue computations of dense and sparse matrices (experimental, contributed by Günther Mader and Astrid Rupp)
- Added singular value decomposition in single precision (experimental, contributed by Volodymyr Kysenko)
- Two new ILU-preconditioners added: ILU0 (contributed by Evan Bollig) and a block-diagonal ILU preconditioner using either ILUT or ILU0 for each block. Both preconditioners are computed entirely on the CPU.

- Automated OpenCL kernel generator based on high-level operation specifications added (many thanks to Philippe Tillet who had a lot of *fun fun fun* working on this)
- Two new sparse matrix types (by Volodymyr Kysenko): `ell_matrix` for the ELL format and `hyb_matrix` for a hybrid format (contributed by Volodymyr Kysenko).
- Added possibility to specify the OpenCL platform used by a context
- Build options for the OpenCL compiler can now be supplied to a context (thanks to Krzysztof Bzowski for the suggestion)
- Added nonnegative matrix factorization by Lee and Seoung (contributed by Volodymyr Kysenko).

Version 1.2.x

Version 1.2.1

The current release mostly provides a few bug fixes for experimental features introduced in 1.2.0. In addition, performance improvements for matrix-matrix multiplications are applied. The main changes (in addition to some internal adjustments) are as follows:

- Fixed problems with double precision on AMD GPUs supporting `cl_amd_fp64` instead of `cl_khr_fp64` (thanks to Sylvain R.)
- Considerable improvements in the handling of `matrix_range`. Added project() function for convenience (cf. Boost.uBLAS)
- Further improvements of matrix-matrix multiplication performance (contributed by Volodymyr Kysenko)
- Improved performance of QR factorization
- Added direct element access to elements of `compressed_matrix` using `operator()` (thanks to sourceforge.net user Sulif for the hint)
- Fixed incorrect matrix dimensions obtained with the transfer of non-square sparse Eigen and MTL matrices to ViennaCL objects (thanks to sourceforge.net user ggrocca for pointing at this)

Version 1.2.0

Many new features from the Google Summer of Code and the IμE Summer of Code enter this release. Due to their complexity, they are for the moment still in *experimental* state (see the respective chapters for details) and are expected to reach maturity with the 1.3.0 release. Shorter release cycles are planned for the near future.

- Added a bunch of algebraic multigrid preconditioner variants (contributed by Markus Wagner)
- Added (factored) sparse approximate inverse preconditioner (SPAI, contributed by Nikolay Lukash)

- Added fast Fourier transform (FFT) for vector sizes with a power of two, standard Fourier transform for other sizes (contributed by Volodymyr Kysenko)
- Additional structured matrix classes for circulant matrices, Hankel matrices, Toeplitz matrices and Vandermonde matrices (contributed by Volodymyr Kysenko)
- Added reordering algorithms (Cuthill-McKee and Gibbs-Poole-Stockmeyer, contributed by Philipp Grabenweger)
- Refurbished CMake build system (thanks to Michael Wild)
- Added matrix and vector proxy objects for submatrix and subvector manipulation
- Added (possibly GPU-assisted) QR factorization
- Per default, a `viennacl::ocl::context` now consists of one device only. The rationale is to provide better out-of-the-box support for machines with hybrid graphics (two GPUs), where one GPU may not be capable of double precision support.
- Fixed problems with `viennacl::compressed_matrix` which occurred if the number of rows and columns differed
- Improved documentation for the case of multiple custom kernels within a program
- Improved matrix-matrix multiplication kernels (may lead to up to 20 percent performance gains)
- Fixed problems in GMRES for small matrices (dimensions smaller than the maximum number of Krylov vectors)

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 new 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

Appendix C

License

Copyright (c) 2010-2013 Institute for Microelectronics, Institute for Analysis and Scientific Computing, TU Wien. Portions of this software are copyright by UChicago Argonne, LLC. Argonne National Laboratory, with facilities in the state of Illinois, is owned by The United States Government, and operated by UChicago Argonne, LLC under provision of a contract with the Department of Energy.

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

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

[5] "NVIDIA CUDA." [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

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

[7] "OpenMP." [Online]. Available: http://openmp.org

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

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

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

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

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

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

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

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

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

[17] G. H. Golub and C. F. Van Loan, *Matrix Computations*. John Hopkins University Press, 1996.

[18] H. D. Simon, "The lanczos algorithm with partial reorthogonalization," *Mathematics of Computation*, vol. 42, pp. 115–142, 1984.

[19] U. Trottenberg, C. Oosterlee, and A. Schüller, *Multigrid*. Academic Press, 2001.

[20] U. M. Yang, *Numerical Solutions of Partial Differential Equations on Parallel Computers*, ser. Lecture Notes in Computational Science and Engineering. Springer, 2006, ch. Parallel Algebraic Multigrid Methods - High Performance Preconditioners, pp. 209–236.

[21] M. J. Grote and T. Huckle, "Parallel Preconditioning with Sparse Approximate Inverses," *SIAM J. Sci. Comp.*, pp. 838–853, 1997.

[22] T. Huckle, "Factorized Sparse Approximate Inverses for Preconditioning," *J. Supercomput.*, pp. 109–117, 2003.

[23] E. Cuthill and J. McKee, "Reducing the bandwidth of sparse symmetric matrices," in *Proceedings of the 1969 24th National Conference*, ser. ACM '69. ACM, 1969, pp. 157–172.

[24] J. G. Lewis, "Algorithm 582: The gibbs-poole-stockmeyer and gibbs-king algorithms for reordering sparse matrices," *ACM Trans. Math. Softw.*, vol. 8, pp. 190–194, 1982.

[25] D. D. Lee and S. H. Seung, "Algorithms for Non-negative Matrix Factorization," in *Advances in Neural Information Processing Systems 13*, 2000, p. 556562.

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