ViennaCL

The Vienna Computing Library

Florian Rudolf

Institute for Microelectronics
Institute for Analysis and Scientific Computing

Technische Universität Wien, Austria

CG Libs - Smart Libraries for Computer Graphics 2013

Talk Overview

Session 1

- Introduction to ViennaCL
- How-To: ViennaCL Basics

Session 2

- How-To: Advanced ViennaCL
- ViennaCL: Behind the curtain

Introduction to `ViennaCL`

What to expect

- What is ViennaCL
- OpenCL
- History of ViennaCL
- Goals of ViennaCL
- Installation of ViennaCL

What Is `ViennaCL`?

What is it about the Name?

- The beautiful city of **Vienna**
- Open**CL**

History of OpenCL

Prior to 2008

OpenCL developed by Apple Inc.

2008

OpenCL working group formed at Khronos Group

OpenCL specification 1.0 released

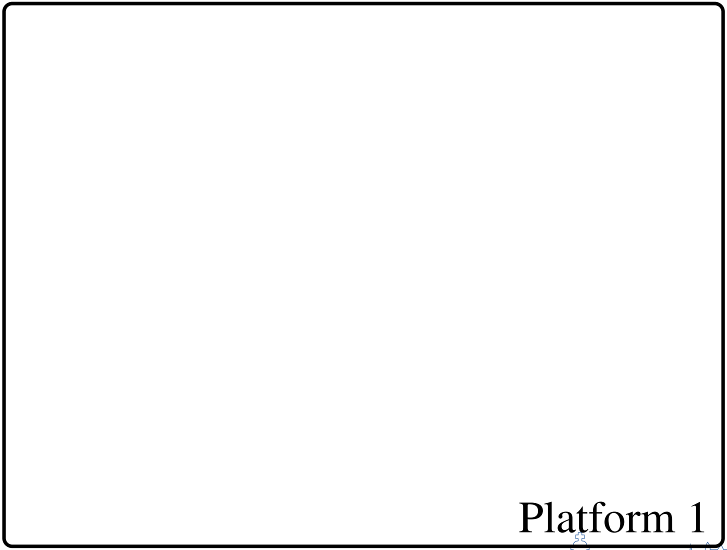
2010

OpenCL 1.1 (multi-device, subbuffer manipulation)

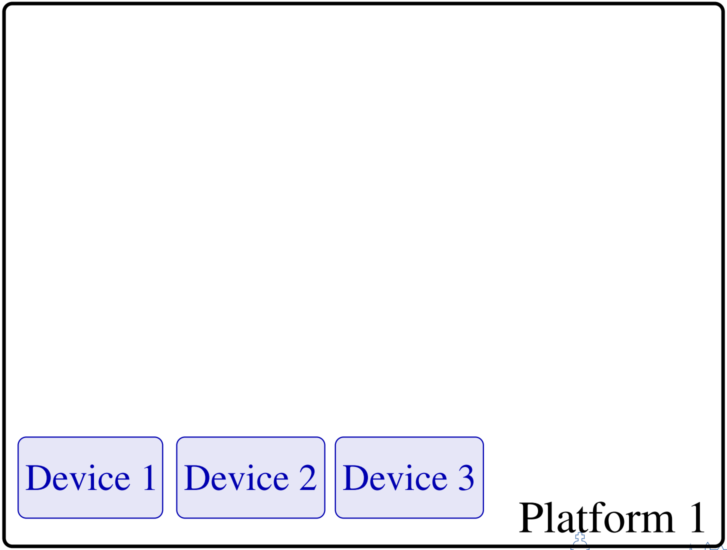
2011

OpenCL 1.2 (device partitioning)

OpenCL Platform Model

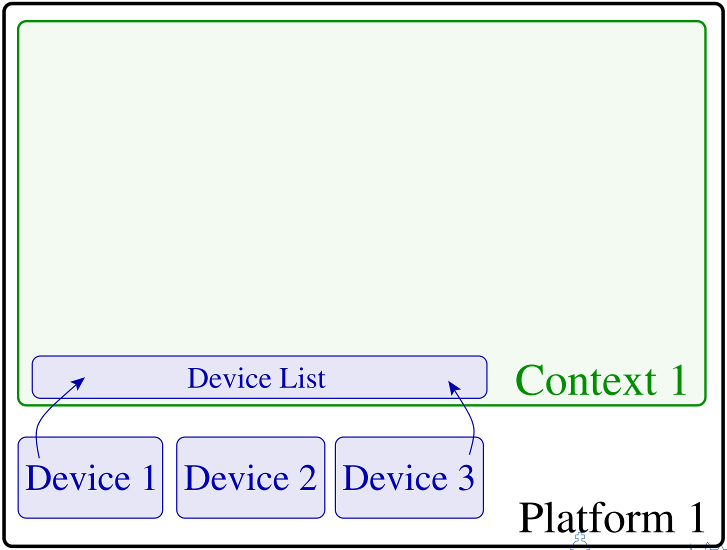


OpenCL Platform Model

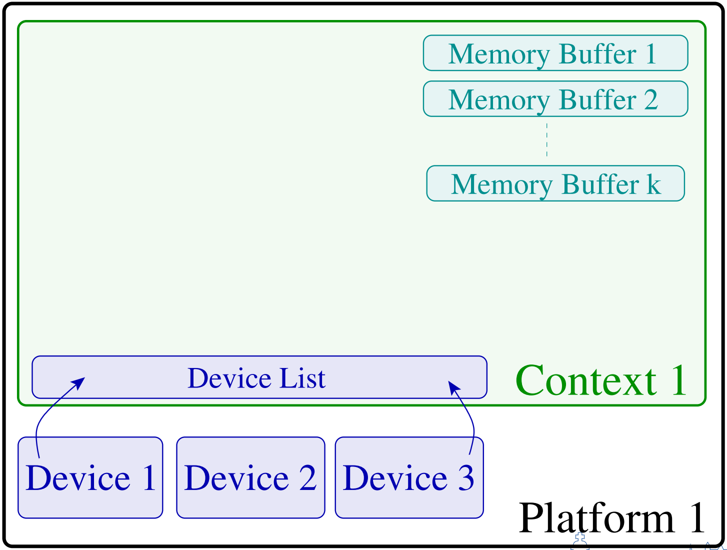


OpenCL Platform Model

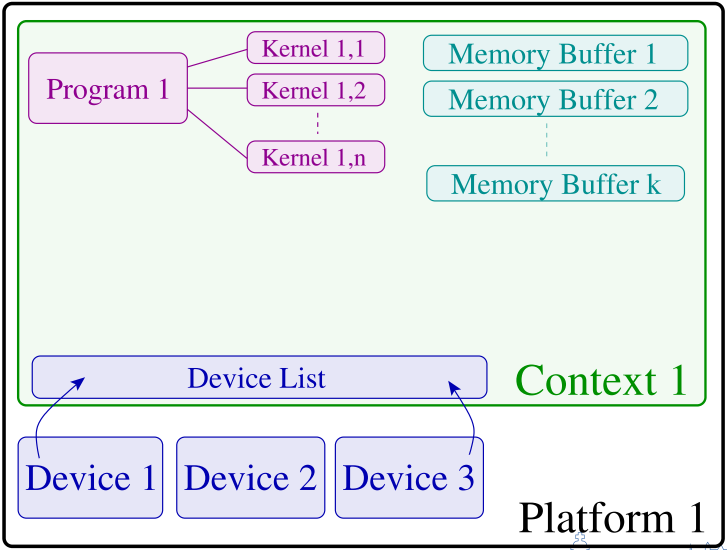
OpenCL Platform Model



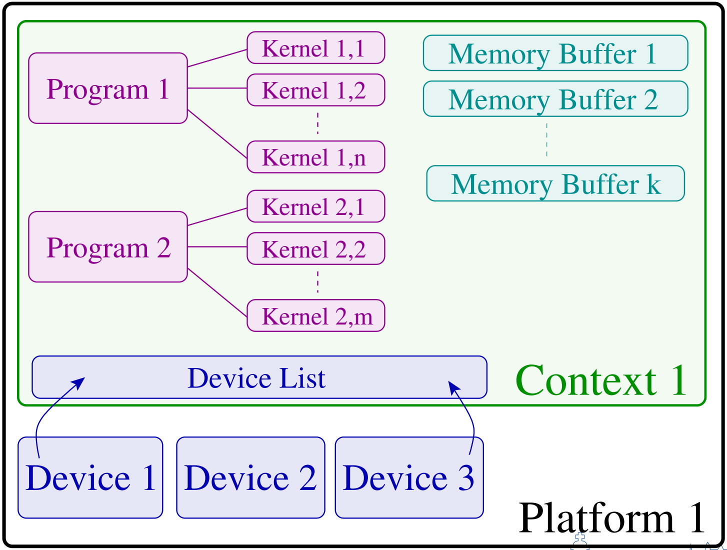
OpenCL Platform Model



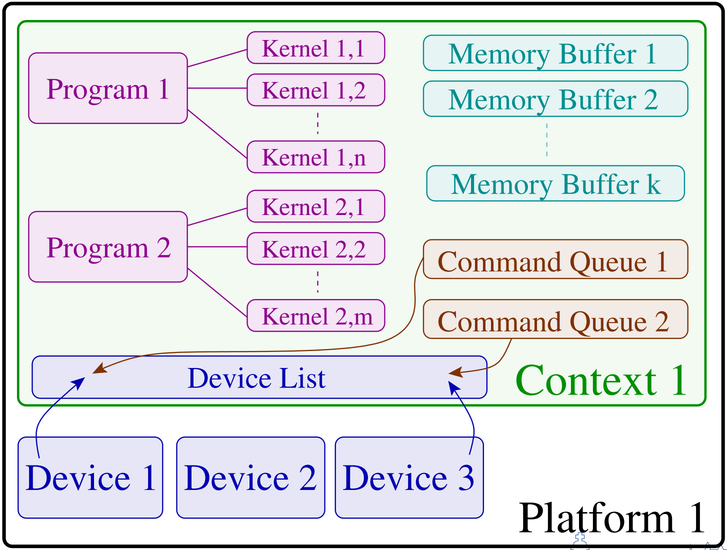
OpenCL Platform Model



OpenCL Platform Model



OpenCL Platform Model



OpenCL Host API

```c
context = clCreateContextFromType(NULL, CL_DEVICE_TYPE_GPU, NULL, NULL,
     NULL);
queue = clCreateCommandQueue(context, NULL, 0, NULL);
memobjs[0] = clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*2*
     num_entries, NULL, NULL);
memobjs[1] = clCreateBuffer(context, CL_MEM_READ_ONLY |
     CL_MEM_COPY_HOST_PTR, sizeof(float)*2*num_entries, srcA, NULL);

program = clCreateProgramWithSource(context, 1, &kernel_src, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);

kernel = clCreateKernel(program, "my_kernel", NULL);
clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&memobjs[0]);
clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&memobjs[1]);
clSetKernelArg(kernel, 2, sizeof(float)*(local_work_size[0]+1)*16, NULL);

global_work_size[0] = 128;
 local_work_size[0] = 64;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_work_size,
     local_work_size, 0, NULL, NULL);
```

Issues

“Where is the error?”

Manage OpenCL handles

OpenCL Kernel Language

Sample Operation: Inplace Vector Addition

$$\begin{pmatrix} v_1^1 \\ v_1^2 \\ \vdots \\ v_1^n \end{pmatrix} += \begin{pmatrix} v_2^1 \\ v_2^2 \\ \vdots \\ v_2^n \end{pmatrix}$$

OpenCL Kernel

```
__kernel void inplace_add(
          __global const float * vec1,
          __global const float * vec2,
          unsigned int size)
{
  for (unsigned int i  = get_global_id(0);
                    i  < size;
                    i += get_global_size(0))
    vec1[i] += vec2[i];
}
```

History of `ViennaCL`

2010

April: Roots in the Master's Thesis of Florian Rudolf

May 28th: ViennaCL 1.0.0 released

November: 1000th download

December: ViennaCL 1.1.0
(BLAS level 3, refurbished OpenCL backend)

History of `ViennaCL`

2010

April: Roots in the Master's Thesis of Florian Rudolf

May 28th: ViennaCL 1.0.0 released

November: 1000th download

December: ViennaCL 1.1.0
(BLAS level 3, refurbished OpenCL backend)

2011

March: Accepted for Google Summer of Code

December: ViennaCL 1.2.0
(AMG, SPAI, FFT, QR, graph algorithms, structured matrices)

History of `ViennaCL`

2012

March: Accepted for Google Summer of Code

May: Tutorial at NVIDIA GTC 2012

May: ViennaCL 1.3.0
(ranges and slices, Automated OpenCL kernels, eigen values, ILU0, SVD)

December: ViennaCL 1.4.0
(CUDA and host backend, initializer types)

History of ViennaCL

2012

March: Accepted for Google Summer of Code

May: Tutorial at NVIDIA GTC 2012

May: ViennaCL 1.3.0
(ranges and slices, Automated OpenCL kernels, eigen values, ILU0, SVD)

December: ViennaCL 1.4.0
(CUDA and host backend, initializer types)

2013

May: Accepted for Google Summer of Code

June: Tutorial at CGLibs

Goals of ViennaCL

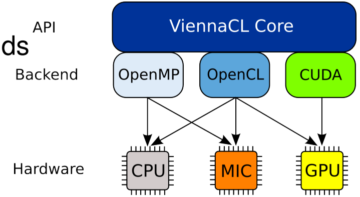
About

High-level linear algebra C++ library

OpenMP, OpenCL, and CUDA backends

Header-only

Multi-platform



Dissemination

Free Open-Source MIT (X11) License

http://viennacl.sourceforge.net/

50-100 downloads per week

Design Rules

Reasonable default values

Compatible to Boost.uBLAS whenever possible

In doubt: clean design over performance

Goals of ViennaCL

Core features

- Linear algebra, BLAS
- Solver (direct and iterative)
- Preconditioners

Additional features

- Fast Fourier Transform
- Eigenvalue computations
- QR factorization
- Bandwidth reduction
- Nonnegative matrix factorization

Goals of ViennaCL

Interfaces to other libraries

- Boost.uBLAS
- Eigen
- MTL4

Backends

- CPU
- OpenCL
- CUDA

C++ library

- Generic free functions
- Expression templates

Installation of `ViennaCL`

Three Steps

- Download from http://viennacl.sourceforge.net/
- Unzip
- Copy source folder

Dynamic Library?

- ViennaCL is header-only
- Linking depends on used backend (OpenMP, OpenCL, CUDA)

Sample Applications

- 22 tutorials
- 7 benchmarks
- about 35 tests

How-To: `ViennaCL` Basics

How-To: `ViennaCL` Basics

What to expect

- From BLAS to Boost.uBLAS to ViennaCL
- Basic Types
- OpenCL Kernels
- Basic Usage: Data Management
- Basic Usage: Algebra
- Basic Usage: Solver
- Summary

From BLAS to Boost.uBLAS to ViennaCL

BLAS

- **B**asic **L**inear **A**lgebra **S**ubprograms
- De facto API standard
- Low level interface

Boost.uBLAS

- C++ API for BLAS
- High level interface
- Part of Boost libraries

From BLAS to Boost.uBLAS to ViennaCL

Consider Existing CPU Code (Boost.uBLAS)

```
using namespace boost::numeric::ublas;

matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << "  2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

High-Level Code with Syntactic Sugar

From BLAS to Boost.uBLAS to ViennaCL

Porting the previous code to GPU

“How much time will you need?“

- 1 minute?
- 1 hour?
- 1 day?

Quality Criteria

- Working hours spent
- Performance
- Correctness
- High-level description (Maintainability)

Consider Existing CPU Code (Boost.uBLAS)

```
using namespace boost::numeric::ublas;


matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << "  2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

Previous Code Snippet Rewritten with ViennaCL

```
using namespace viennacl;
using namespace viennacl::linalg;

matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

rhs += 2.0 * x;
double val = inner_prod(x, rhs);
matrix += val * outer_prod(x, rhs);

x = solve(A, rhs, upper_tag()); // Upper triangular solver

std::cout << "  2-norm: " << norm_2(x) << std::endl;
std::cout << "sup-norm: " << norm_inf(x) << std::endl;
```

From BLAS to Boost.uBLAS to ViennaCL

ViennaCL in addition provides iterative solvers

```
using namespace viennacl;
using namespace viennacl::linalg;

compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solver
x = solve(A, rhs, gmres_tag());    // GMRES solver
```

No iterative solvers available in uBLAS...

From BLAS to Boost.uBLAS to ViennaCL

Thanks to interface compatibility

```
using namespace boost::numeric::ublas;
using namespace viennacl::linalg;

compressed_matrix<double> A(1000, 1000);
vector<double> x(1000), rhs(1000);

/* Fill A, x, rhs here */

x = solve(A, rhs, cg_tag()); // Conjugate Gradient solver
x = solve(A, rhs, bicgstab_tag()); // BiCGStab solver
x = solve(A, rhs, gmres_tag());    // GMRES solver
```

OpenCL kernels

OpenCL kernels are needed for actual computation

- Provided by ViennaCL
- Support for expression templates
- Automatic kernel generation

Each of the following commands launches a separate OpenCL kernel

```
v1 = v2;
v1 += v2;
v1 -= v2;
v1 = alpha * v2;
v1 += alpha * v2;
v1 -= alpha * v2;
v1 *= alpha;
v1 /= alpha;
```

OpenCL kernels

OpenCL kernels have to be compiled at run time

- OpenCL JIT compiler
- Kernels can be grouped in programs

Compilation strategies

- Each kernel individually: several milliseconds per kernel
- All at once: Takes seconds
- In groups: Compile groups of kernels whenever potentially needed

OpenCL kernels

Consider the following code

```
int main(){
  vector<double> x(100), y(100);
  matrix<double> A(100, 100), B(100, 100);
  matrix<double, column_major> C(100, 100);

  x += 3.1415 * y;
  C = prod(trans(A), B);
  y = prod(C, x);

  std::cout << y << std::endl;
}
```

OpenCL Kernels

Which kernels are compiled?

When are they compiled?

Hint: Program compiles and executes normally

OpenCL kernels

Consider the following code

```
int main(){
  vector<double> x(100), y(100);
  matrix<double> A(100, 100), B(100, 100);
  matrix<double, column_major> C(100, 100);

  x += 3.1415 * y;
  C = prod(trans(A), B);
  y = prod(C, x);

  std::cout << y << std::endl;
}
```

OpenCL Kernels

Which kernels are compiled?

When are they compiled?

Hint: Program compiles and executes normally

OpenCL kernels

Consider the following code

```
int main(){
  vector<double> x(100), y(100);
  matrix<double> A(100, 100), B(100, 100);
  matrix<double, column_major> C(100, 100);

  x += 3.1415 * y;
  C = prod(trans(A), B);
  y = prod(C, x);

  std::cout << y << std::endl;
}
```

OpenCL Kernels

Which kernels are compiled?

When are they compiled?

Hint: Program compiles and executes normally

OpenCL kernel management

Special case: Matrix-Matrix product

- Result matrix: Row/Column major memory layout
- First factor: Row/Column major, possibly transposed
- Second factor: Row/Column major, possibly transposed

Leads to 32 different kernels for matrix-matrix multiplication

- Compiled separately on request

Basic Types

Supported types

- Scalar
- Vector
- Dense matrix
- Sparse matrix
- Structured matrix

Numeric Types

- float
- double

Basic Types

Scalars

- Represents a single scalar value on the computing device
- Behave like underlying type
- Implicit cast to underlying type
- Potentially expensive (Overhead!)

```
viennacl::scalar<NumericType> gpu_scalar;
viennacl::scalar<float> gpu_float;
viennacl::scalar<double> gpu_double;
```

Basic Types

Scalars

```
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;
// automatic transfer and conversion
gpu_float = cpu_double;

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

Basic Types

Vectors

- Represents a vector on the computing device
- Operator overloading
- Alignment support

```
viennacl::vector<NumericType> gpu_vector;

viennacl::vector<float> gpu_float_vector;
viennacl::vector<double> gpu_double_vector;
```

Vectors

```
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
vcl_vec *= 4.2;

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

Basic Types

Vectors

```
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, vcl_vec);

//manipulate GPU vector here
vcl_vec *= 4.2;

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

Basic Types

Alignment

- Default = 1
- Template parameter
- In ViennaCL 1.5.0 deprecated (will be runtime parameter)

```
viennacl::vector<NumericType, Alignment>
    gpu_vector_with_alignment;

viennacl::vector<float, 4>
    gpu_float_vector_with_alignment;

viennacl::vector<double, 8>
    gpu_double_vector_with_alignment;
```

Vector initializer

$$\text{unit\_vector} \Rightarrow \mathbf{u}_{N,i} = \begin{cases} 1 & \text{if } i = N; \\ 0 & \text{else} \end{cases}$$

$$\text{zero\_vector} \Rightarrow \mathbf{z}_i = 0$$

$$\text{scalar\_vector} \Rightarrow \mathbf{s}_i = s$$

```
viennacl::unit_vector<NumericType>(size, index);

viennacl::zero_vector<NumericType>(size);

viennacl::scalar_vector<NumericType>(size, scalar);
```

Vector initializer

```
viennacl::vector<ScalarType> vcl_vec =
    viennacl::unit_vector<ScalarType>(10, 5);
// Creates the vector (0, 0, 0, 0, 1, 0, 0, 0, 0, 0)

viennacl::vector<ScalarType> vcl_vec =
    viennacl::zero_vector<ScalarType>(10);
// Creates the vector (0, 0, 0, 0, 0, 0, 0, 0, 0, 0)

viennacl::vector<ScalarType> vcl_vec =
    viennacl::scalar_vector<ScalarType>(6, 1.5);
// Creates the vector (1.5, 1.5, 1.5, 1.5, 1.5, 1.5)
```

Basic Types

Dense matrix

- Represents a dense matrix on the computing device
- Dense matrix ⇒ zero elements are rare
- Alignment support (same as vectors)
- Row major or column major

```
viennacl::matrix<NumericType> gpu_matrix;
viennacl::matrix<NumericType, Scheme, Alignment>
    gpu_matrix_with_scheme_and_alignment;

viennacl::matrix<float> gpu_float_matrix;
viennacl::matrix<float, row_major, 4>
    row_major_gpu_float_matrix_with_alignment;

viennacl::matrix<double> gpu_double_matrix;
viennacl::matrix<double, column_major, 8>
    column_major_gpu_double_matrix_with_alignment;
```

Dense matrix

```
//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;
```

Matrix initializer

$$\text{identity\_matrix} \Rightarrow \mathbf{I}_{i,j} = \begin{cases} d & \text{if } i = j; \\ 0 & \text{else} \end{cases}$$

$$\text{zero\_matrix} \Rightarrow \mathbf{Z}_{i,j} = 0$$

$$\text{scalar\_matrix} \Rightarrow \mathbf{S}_{i,j} = s$$

```
viennacl::identity_matrix<NumericType>(size, diagonal);

viennacl::zero_matrix<NumericType>(row_number,
    column_number);

viennacl::scalar_matrix<NumericType>(row_number,
    column_number, scalar);
```

Matrix initializer

```
viennacl::matrix<ScalarType> vcl_mat =
    viennacl::identity_matrix<ScalarType>(4, 1);
```

Creates the following matrix: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Basic Types

Matrix initializer

```
viennacl::matrix<ScalarType> vcl_mat =
    viennacl::zero_matrix<ScalarType>(3, 5);
```

Creates the following matrix: $\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$

Matrix initializer

```
viennacl::matrix<ScalarType> vcl_mat =
    viennacl::scalar_matrix<ScalarType>(4, 3, 4.2);
```

Creates the following matrix: $\begin{pmatrix} 4.2 & 4.2 & 4.2 \\ 4.2 & 4.2 & 4.2 \\ 4.2 & 4.2 & 4.2 \\ 4.2 & 4.2 & 4.2 \end{pmatrix}$

Basic Types

Sparse matrix

- Represents a sparse matrix on the computing device
- Sparse matrix $\Rightarrow$ zero elements are frequent
- Alignment support (same as vectors and dense matrices)

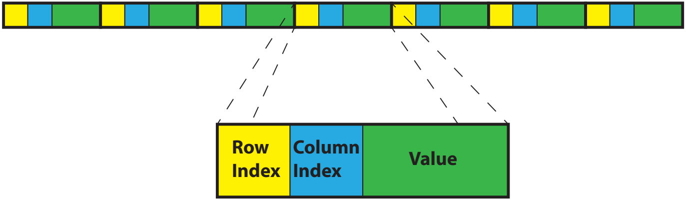
Sparse matrix types

- Coordinate matrix
- Compressed matrix
- ELL matrix
- Hybrid matrix

Basic Types

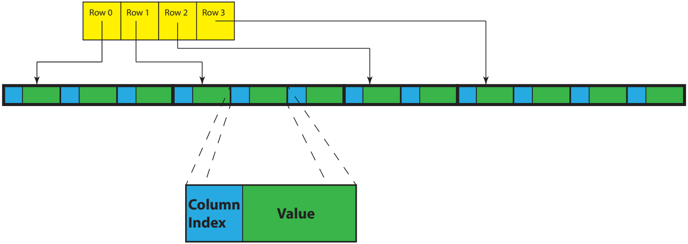
Coordinate matrix

Easy to setup/extend



Compressed matrix

- Less memory required
- Fast matrix-vector multiplication



Basic Types

ELL matrix

- Similar to compressed matrix
- Fixed number of non-zero values per row
- No row jumper array required

Hybrid matrix

- Combination of compressed matrix and ELL matrix

Compressed matrix

```
//set up a sparse 4 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> sparse_matrix(4, 5);

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

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

Basic Types

Structured matrix

- Dense matrices but with special structure
- Access to one element might change other elements

Supported types

- Circulant matrix
- Hankel matrix
- Toeplitz matrix
- Vandermonde matrix

Structured matrix

Circulant matrix

Hankel matrix

Toeplitz matrix

Vandermonde matrix

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

Structured matrix

- Circulant matrix
- **Hankel matrix**
- Toeplitz matrix
- Vandermonde matrix

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

Structured matrix

- Circulant matrix
- Hankel matrix
- **Toeplitz matrix**
- Vandermonde matrix

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

Basic Types

Structured matrix

- Circulant matrix
- Hankel matrix
- Toeplitz matrix
- **Vandermonde matrix**

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

Basic Usage: Data Management

How to access/transfer ViennaCL vectors/matrices elements?

Direct element access

```
vector<ScalarType> vcl(10);

for (int i = 0; i < 10; ++i)
    vcl(i) = i;

for (int i = 0; i < 10; ++i)
    std::cout << vcl(i) << std::endl;
```

How to access/transfer ViennaCL vectors/matrices elements?

- Direct element access
- Iterator

```
vector<ScalarType> vcl(10);

ScalarType tmp = 0;
for (vector<ScalarType>::iterator it = vcl.begin();
     it != vcl.end(); ++it, tmp += 42.0)
     *it = tmp;

for (vector<ScalarType>::iterator it = vcl.begin();
     it != vcl.end(); ++it)
     std::cout << *it < std::endl;
```

Basic Usage: Data Management

How to access/transfer ViennaCL vectors/matrices elements?

- Direct element access
- Iterator
- Copy functions

```
std::vector<ScalarType> cpu(10);
viennacl::vector<ScalarType> vcl(10);

for (int i = 0; i < 10; ++i)
    cpu[i] = i;

viennacl::copy( cpu.begin(), cpu.end(), vcl.begin() );

viennacl::copy( vcl.begin(), vcl.end(), cpu.begin() );
```

Basic Usage: Data Management

How to access/transfer ViennaCL vectors/matrices elements?

- Direct element access
- Iterator
- Copy functions

```
std::vector<ScalarType> cpu(10);
viennacl::vector<ScalarType> vcl(10);

for (int i = 0; i < 10; ++i)
    cpu[i] = i;

viennacl::copy( cpu, vcl );

viennacl::copy( vcl, cpu );
```

Granularity of Operations

Filling a vector with data

```
viennacl::vector<double> v(10000);

for (size_t i=0; i<v.size(); ++i)
  v(i) = i;
```

Basic Usage: Data Management

Granularity of Operations

Filling a vector with data

```
viennacl::vector<double> v(10000);

for (size_t i=0; i<v.size(); ++i)
  v(i) = i;
```

GPU Computing Is Fast, Right?

Execution time: 1 sec (approx)

`std::vector`: $<$ 1 ms

Basic Usage: Data Management

Granularity of Operations

Transfer is done for each element separately

High overhead, similar to scalar operations

How to avoid the pitfall

Use temporary vector

Use copy functions

```
viennacl::vector<double> v(10000);
std::vector<double> cpu_v( v.size() );

for (size_t i=0; i<cpu_v.size(); ++i)
  cpu_v(i) = i;

viennacl::copy(cpu_v, v);
```

Basic Usage: Data Management

How to access/transfer ViennaCL vectors/matrices elements?

Direct element access $\Rightarrow$ Bad idea!

Iterator $\Rightarrow$ Bad idea!

Copy functions $\Rightarrow$ **Good idea!**

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

for (int i = 0; i < 10; ++i)
    cpu_vec[i] = i;

viennacl::copy( cpu_vec, vcl_vec );

viennacl::copy( vcl_vec, cpu_vec );
```

Fast copy

copy does not require linear arrays ⇒ temporary required

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

for (int i = 0; i < 10; ++i)
    cpu_vec[i] = i;

viennacl::copy( cpu_vec, vcl_vec );

viennacl::copy( vcl_vec, cpu_vec );
```

Basic Usage: Data Management

Fast copy

copy does not require linear arrays ⇒ temporary required

If container is linear memory ⇒ use fast_copy instead

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

for (int i = 0; i < 10; ++i)
    cpu_vec[i] = i;

viennacl::fast_copy( cpu_vec, vcl_vec );

viennacl::fast_copy( vcl_vec, cpu_vec );
```

Basic Usage: Algebra

Algebra operations

- Trivial operations done by **operator overloading**
- Scalar multiplication support for vector and matrices
- Inner product and norm support for vectors
- Matrix transpose using function **trans**
- Matrix-vector and matrix-matrix multiplication using function **prod**

Basic Usage: Algebra

Scalar operations

```
NumericType s1, s2;
viennacl::scalar<NumericType> vcl_s1, vcl_s2, vcl_s3;

vcl_s1  = 5;
vcl_s2  = vcl_s1 * 3;
vcl_s3 -= vcl_s1 + (vcl_s2 / 4);

s1      = vcl_s3;
s2      = vcl_s1 - vcl_s2 * 2;
```

Basic Usage: Algebra

Vector operations

```
viennacl::scalar<NumericType> vcl_s1, vcl_s2;
viennacl::vector<NumericType> v1(10), v2(10), v3(10);

v2  = vcl_s1 * v1 + v2
v3 += vcl_s1 * v2;

v3  = 0.5 * v2 - v3;
```

Basic Usage: Algebra

Vector operations

```
NumericType s1, s2;
viennacl::scalar<NumericType> vcl_s1, vcl_s2;
viennacl::vector<NumericType> v10), v2(10), v3(10);

vcl_s1 = viennacl::linalg::inner_prod(v1, v2);
s1     = viennacl::linalg::inner_prod(v1, v2);

s1     = viennacl::linalg::norm_1(v1);
vcl_s2 = viennacl::linalg::norm_2(v2);
s2     = viennacl::linalg::norm_inf(v3);
```

Basic Usage: Algebra

Matrix operations

```
viennacl::scalar<NumericType> vcl_s1, vcl_s2;
viennacl::matrix<NumericType> M1(10, 10),
    M2(10, 10), M3(10,10);

M2  = vcl_s1 * M1 + M2
M3 += vcl_s2 * M2;

M3  = 0.5 * M2 -  M3;

M3  = viennacl::trans(M2); // Transposed matrix

// Matrix-matrix product
M1  = viennacl::linalg::prod( M2, M3 );
M1  = viennacl::linalg::prod( M2, viennacl::trans(M3) );
```

Basic Usage: Algebra

GEMM: ViennaCL vs. CUBLAS

```
// ViennaCL
M1 = vcl_s1 * prod( M2, trans(M3) ) + vcl_s2 * M3;

// CUBLAS
cublasStatus_t cublasDgemm(handle,
    transa, transb,
    m, n, k,
    alpha,
    A, lda,
    B, ldb,
    beta,
    C, ldc);
```

Matrix-vector operations

```
viennacl::vector<NumericType> v1(10), v210), v3(20);
viennacl::matrix<NumericType> M1(10, 10), M2(20, 10);

v1 = viennacl::linalg::prod( M1, v2 );
v1 = viennacl::linalg::prod( viennagrid::trans(M1), v2 );
v3 = viennacl::linalg::prod( M2, v2 );
v1 = viennacl::linalg::prod( M1, v3 );
    // ERROR! dimension missmatch
```

Basic Usage: Solver

Solving a system of linear equations

$Ax = b$

A and b given, x is unknown

Common problem in mathematics

Types of solver

Direct solver

Iterative solver

Basic Usage: Solver

Direct solver

Solving the system directly

e.g. Gaussian elimination with pivoting

Iterative solver

Solving using an iterative process

Convergence relies on matrix properties

Recommended for large and sparse systems

No write operation needed on matrix
(most only use matrix-vector multiplication)

Direct solver

- LU factorization
- No pivoting (work in progress)

```
using namespace viennacl::linalg;

viennacl::matrix<float> vcl_matrix;
viennacl::vector<float> vcl_rhs, 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 vector vcl_rhs:
lu_factorize(vcl_matrix);
lu_substitute(vcl_matrix, vcl_rhs);
```

Basic Usage: Solver

Iterative solver

Conjugate Gradient (CG)

Stabilized Bi-CG (BiCGStab)

Generalized Minimum Residual (GMRES)

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());`

Basic Usage: Solver

Iterative solver

Solver configuration via tag

```
using namespace viennacl::linalg;

// conjugate gradient solver with tolerance 1e10
// and at most 100 iterations:
viennacl::linalg::cg_tag custom_cg(1e-10, 100);

vcl_result = solve(vcl_matrix, vcl_rhs, custom_cg);

//print number of iterations taken and estimated error:
cout << "No. of iters: " << custom_cg.iters() << endl;
cout << "Est. error: " << custom_cg.error() << endl;
```

Summary

What have we learned?

- ViennaCL provides interface compatibility with Boost.uBLAS
- Basic types of ViennaCL
- How OpenCL kernels are used
- How to transfer data to and from ViennaCL
- How to work with ViennaCL types
- Simple algebraic operations
- Solver for systems of linear equations

How-To: Advanced `ViennaCL`

What to expect

- Subvectors and Submatrices
- Escaping the Curse of Temporaries
- Interface: Eigen
- Performance
- Summary

Subvectors and Submatrices

Important for Many Algorithms

- LU, Cholesky
- QR, SVD

Two sub-types available

- Range $[a, b)$
- Slice a:inc:size

Ranges and slices are proxies

- Read- and writeable

Subvectors and Submatrices

Range example

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

typedef viennacl::vector<ScalarType> VectorType;
typedef viennacl::matrix<ScalarType> MatrixType;

// v[1:6]
viennacl::vector_range<VCLVectorType> v_sub(v, r);
// M[1:6,1:6]
viennacl::matrix_range<VCLMatrixType> M_sub(M, r, r);
```

Subvectors and Submatrices

Slice example

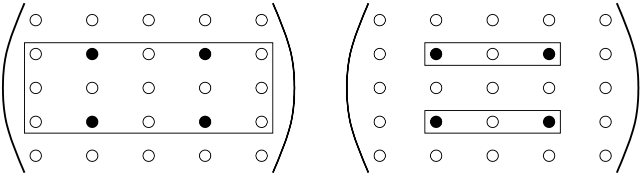
```
std::size_t start = 2;
std::size_t stride = 3;
std::size_t size = 5
viennacl::slice s(start, stride, size);

typedef viennacl::vector<ScalarType> VectorType;
typedef viennacl::matrix<ScalarType> MatrixType;

// v[2, 5, 8, 11, 14]
viennacl::vector_slice<VCLVectorType> v_sub(v, r);
// M[2,2], ..., M[2,14], ..., M[14,2], ..., M[14,14]
viennacl::matrix_slice<VCLMatrixType> M_sub(M, r, r);
```

Convenience Functions

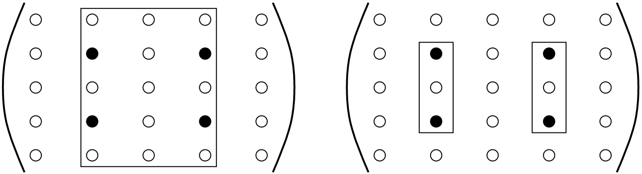
```
viennacl::vector<ScalarType> v1(4), v2(2);
viennacl::matrix<ScalarType> M1(4,4), M2(2,2);

range r(0, 2);
slice s(0, 2, 2);

v2 = project(v1, r);
project(v1, s) = v2;

M2 = project(M1, r, r);
viennacl::copy(M2, project(M1, s, s) );
```

Subvectors and Submatrices

Copy Headaches: Row-Major



Copy Headaches: Column-Major



Escaping the Curse of Temporaries

Simple BLAS Level 1 Operation

Consider

```
vec1 = vec2 + alpha * vec3 - beta * vec4;
```

With naive C++, this is equivalent to

```
tmp1 <- alpha * vec3
tmp2 <- beta * vec4;
tmp3 <- tmp1 - tmp2;
tmp4 <- vec2 + tmp3;
vec1 <- tmp4;
```

Temporaries Lead to Poor Performance

Costly on CPUs, extremely expensive on GPUs

Use expression templates for avoiding temporaries

Escaping the Curse of Temporaries

Vector Addition

```
x = y + z;
```

Naive Operator Overloading

```
vector<T> operator+(vector<T> & v, vector<T> & w);
```

$t \leftarrow y + z, x \leftarrow t$

Temporaries are extremely expensive!

Escaping the Curse of Temporaries

Expression Templates

```
vector_expr<vector<T>, op_plus, vector<T> >
operator+(vector<T> & v, vector<T> & w) { ... }

vector::operator=(vector_expr<...> const & e) {
  viennacl::linalg::avbv(*this, 1,e.lhs(), 1,e.rhs());
}
```

Allows to Avoid a Significant Amount of Temporaries

- Covers most frequent cases
- Influence on compilation times moderate

Escaping the Curse of Temporaries

Expression templates have their limitations

```
viennacl::vector<NumericType> v;
viennacl::matrix<NumericType> M;

v = viennacl::linalg::prod(M, v);
```

Temporary object is required here!

ViennaCL detects such cases and takes care of it

Interface: Eigen

Data transfer

- Like transfer from and to std container
- Using provided copy functions

Interface compatibility

- ViennaCL algorithms work with Eigen
- e.g.: iterative solver

Interface: Eigen

Data transfer: vectors

```
#define VIENNACL_HAVE_EIGEN

Eigen::VectorXd eigen_vector(dim);

// fill Eigen objects in a very sophisticated way with
    numbers here

viennacl::vector<double> viennacl_vector(dim);

// copy data from Eigen objects to ViennaCL objects
viennacl::copy(eigen_vector, viennacl_vector);

// do some heavy linear algebra with ViennaCL here

// copy back to Eigen:
viennacl::copy(viennacl_vector, eigen_vector);
```

Data transfer: dense matrix

```
#define VIENNACL_HAVE_EIGEN

Eigen::MatrixXd eigen_densematrix(dim, dim);

// fill Eigen objects in a very sophisticated way with
    numbers here

viennacl::matrix<double> viennacl_densematrix(dim, dim);

// copy data from Eigen objects to ViennaCL objects
viennacl::copy(eigen_densematrix,viennacl_densematrix);

// do some heavy linear algebra with ViennaCL here

// copy back to Eigen:
viennacl::copy(viennacl_densematrix, eigen_densematrix);
```

Interface: Eigen

Data transfer: sparse matrix

```
#define VIENNACL_HAVE_EIGEN

Eigen::SparseMatrix<double, Eigen::RowMajor>
    eigen_sparsematrix(dim, dim);

// fill Eigen objects in a very sophisticated way with
    numbers here

viennacl::compressed_matrix<double> viennacl_sparsematrix(
    dim, dim);

// copy data from Eigen objects to ViennaCL objects
viennacl::copy(eigen_sparsematrix, viennacl_sparsematrix);

// do some heavy linear algebra with ViennaCL here

// copy back to Eigen:
viennacl::copy(viennacl_sparsematrix, eigen_sparsematrix);
```

Interface: Eigen

Interface Compatibility: iterative solver

```
#define VIENNACL_HAVE_EIGEN
using namespace viennacl::linalg;

Eigen::SparseMatrix<double, Eigen::RowMajor>
    matrix(dim, dim);
Eigen::VectorXd rhs(dim);
Eigen::VectorXd result(dim);
// fill eigen_matrix and eigen_rhs here

// Solve system using CG from ViennaCL
result = solve(matrix, rhs, cg_tag());

// Solve system using BiCGStab from ViennaCL
result = solve(matrix, rhs, bicgstab_tag());

// Solve system using GMRES from ViennaCL
result = solve(matrix, rhs, gmres_tag());
```

Granularity of Operations

Solving linear systems

```
viennacl::matrix<double> mat(N, N);
viennacl::vector<double> rhs(N);

for (size_t i=0; i<1000; ++i)
{
   viennacl::vector<double> result
     = solve(mat, rhs, bicgstab_tag());
   /* process result */
}
```

Why Is There No Speed-Up?

Granularity of Operations

Solving linear systems

```
viennacl::matrix<double> mat(N, N);
viennacl::vector<double> rhs(N);

for (size_t i=0; i<1000; ++i)
{
   viennacl::vector<double> result
     = solve(mat, rhs, bicgstab_tag());
   /* process result */
}
```

Why Is There No Speed-Up?

$N = 3$

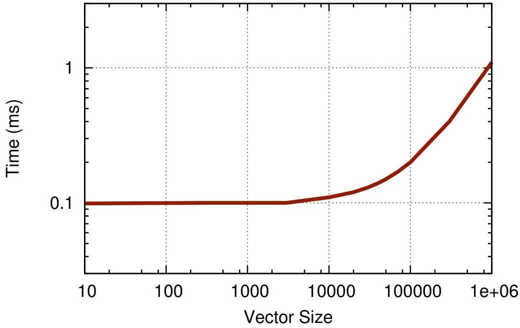
Performance

Lets take a look

Performance

Sample Operation

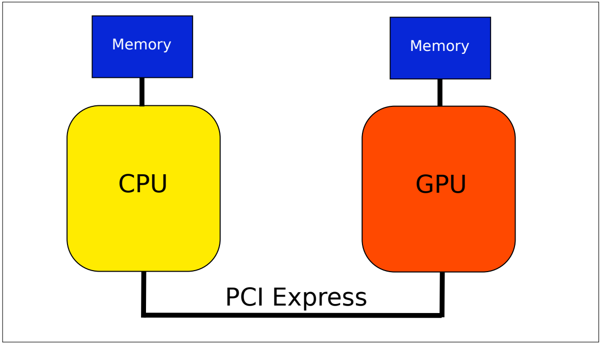
$$v_1 \leftarrow v_2$$



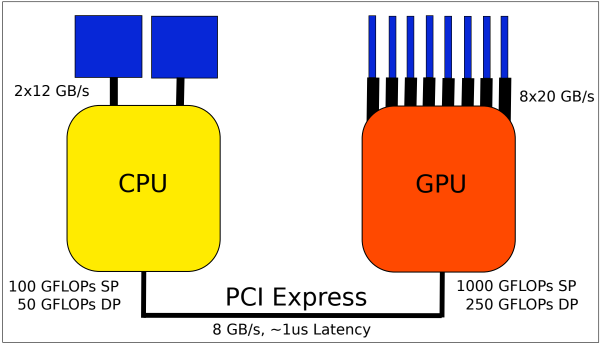
OpenCL Kernel Launch Overhead

$10 - 100\ \mu$s

GPUs: Disillusion - Computing Architecture Schematic



GPUs: Disillusion - Computing Architecture Schematic

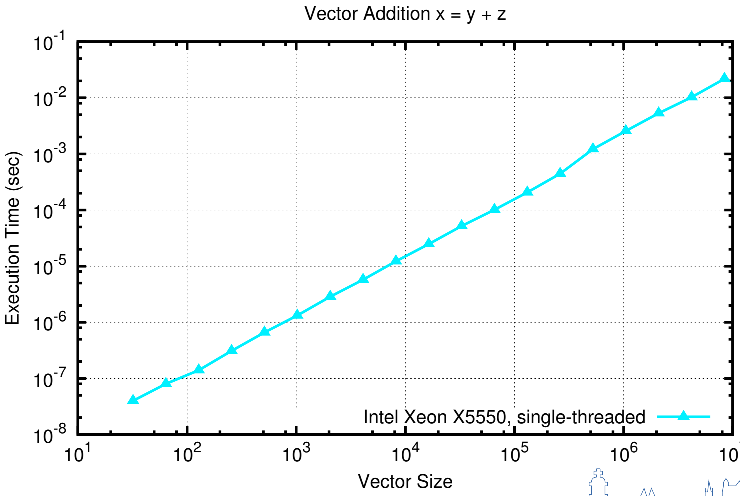


Good for large FLOP-intensive tasks, high memory bandwidth

PCI-Express can be a bottleneck

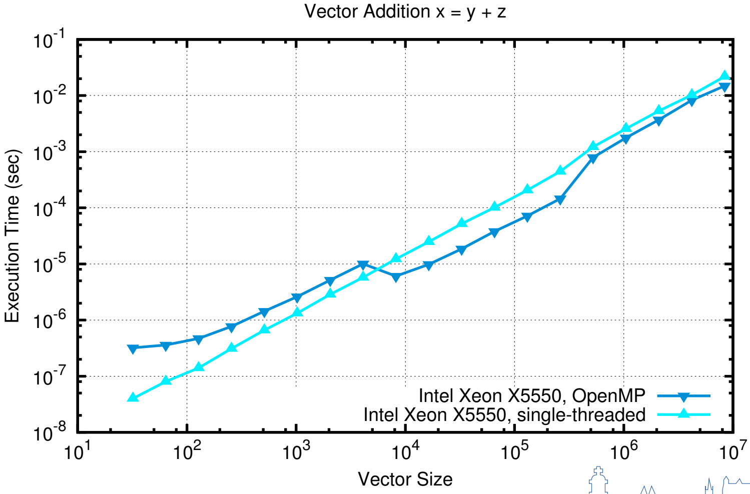
$\gg 10$-fold speedups (usually) not backed by hardware

Some benchmarks - vector addition

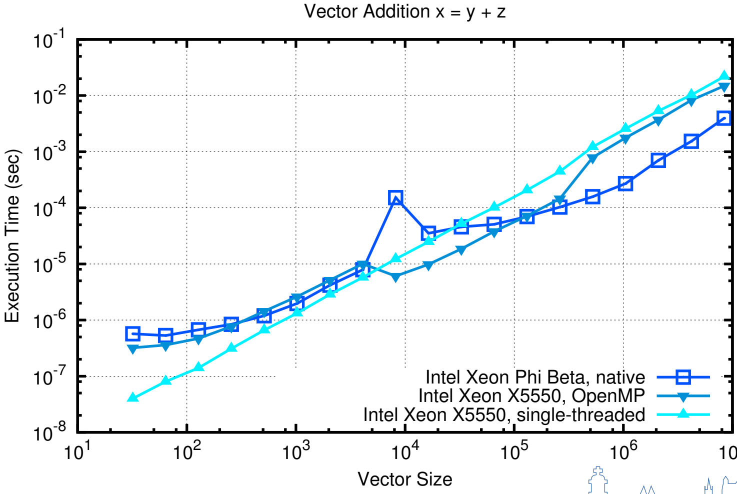


Performance

Some benchmarks - vector addition

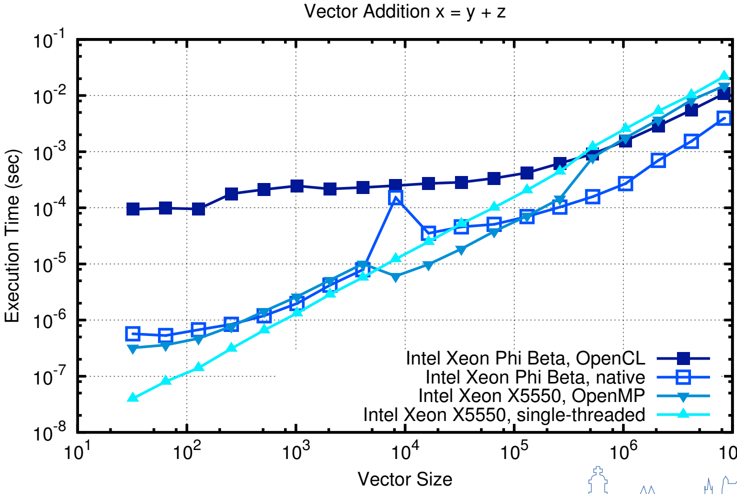


Some benchmarks - vector addition



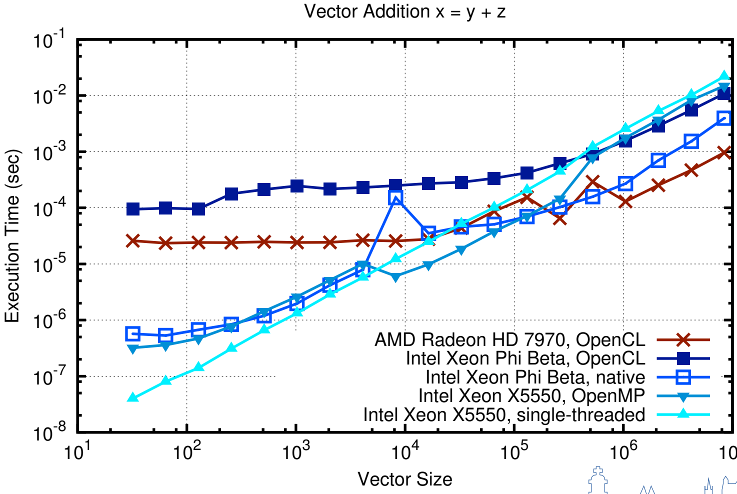
Performance

Some benchmarks - vector addition



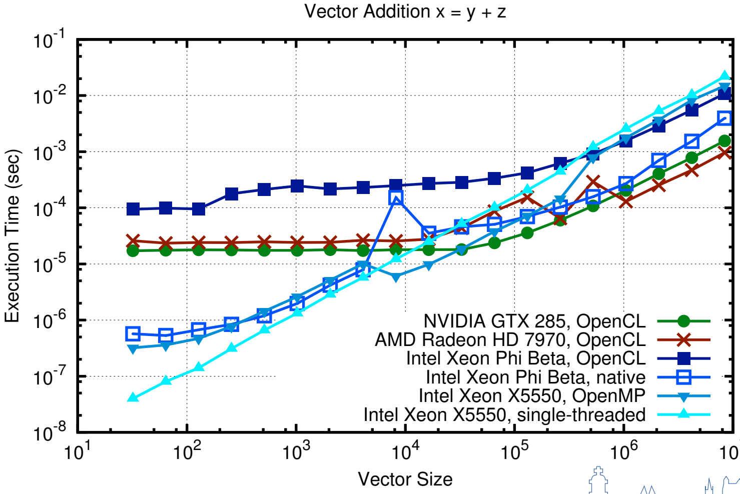
Performance

Some benchmarks - vector addition

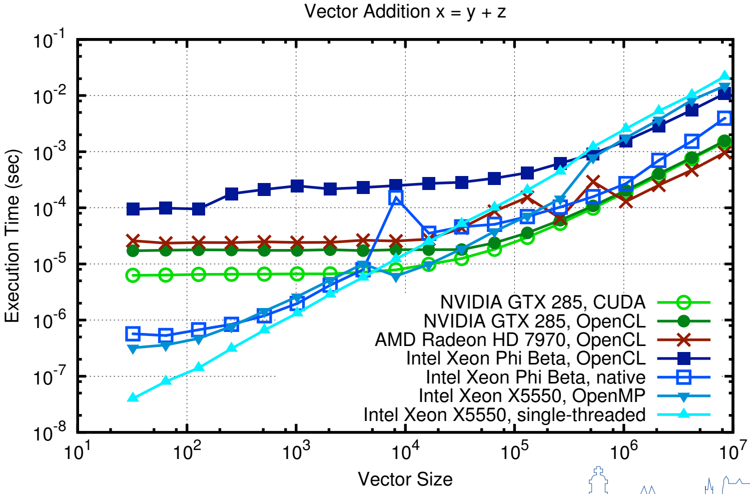


Performance

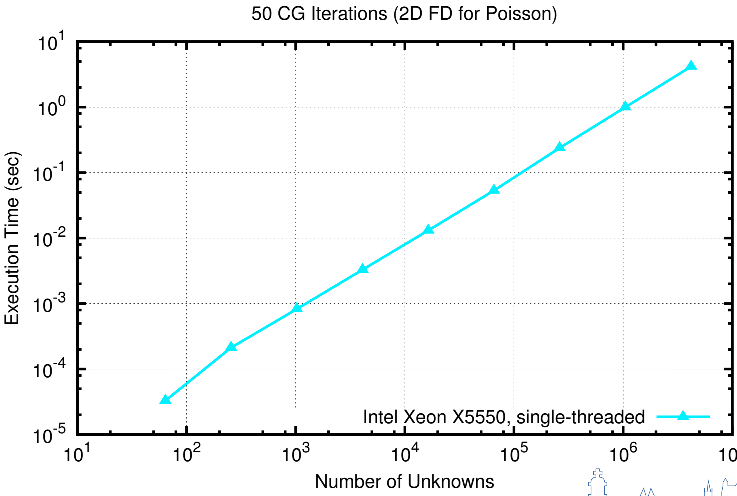
Some benchmarks - vector addition



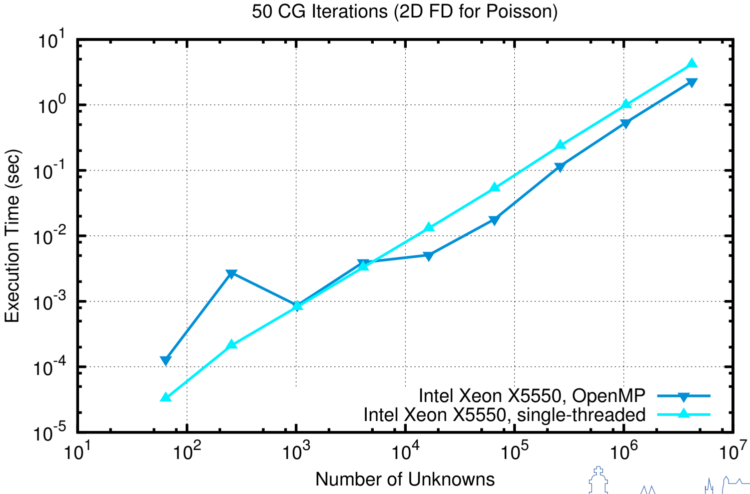
Some benchmarks - vector addition



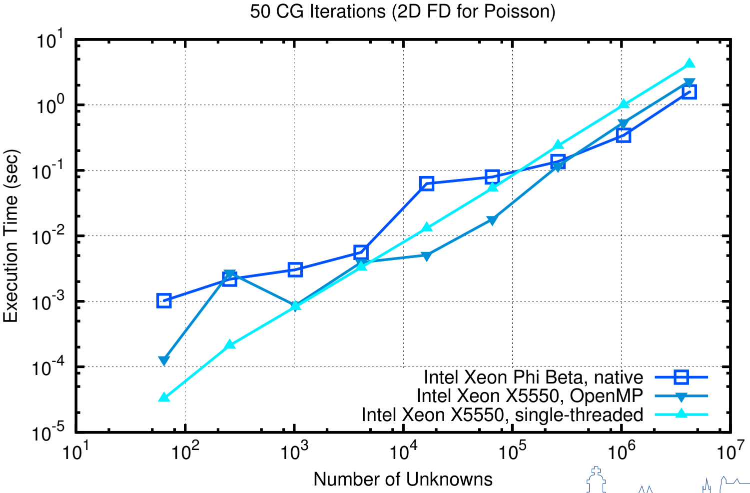
Some benchmarks - CG solver



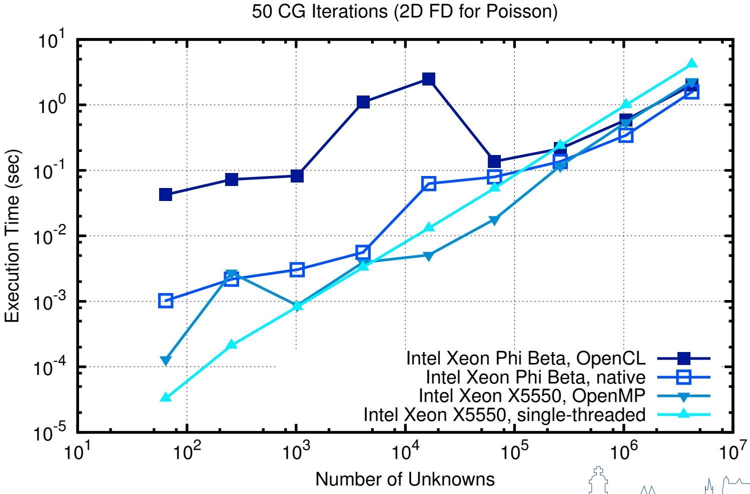
Some benchmarks - CG solver



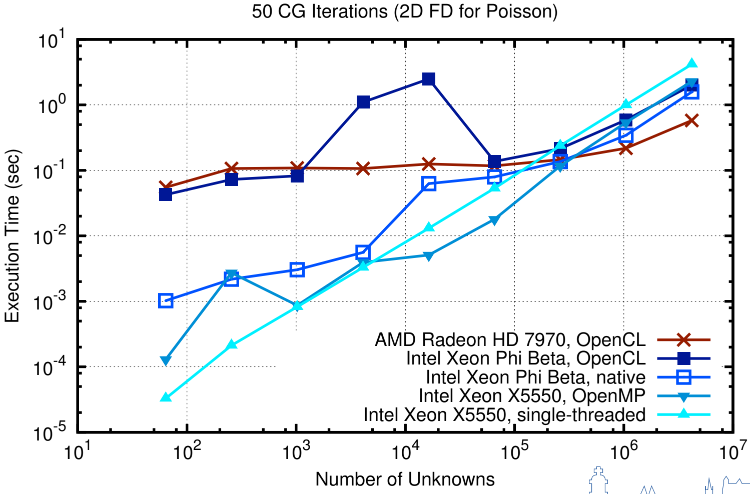
Some benchmarks - CG solver



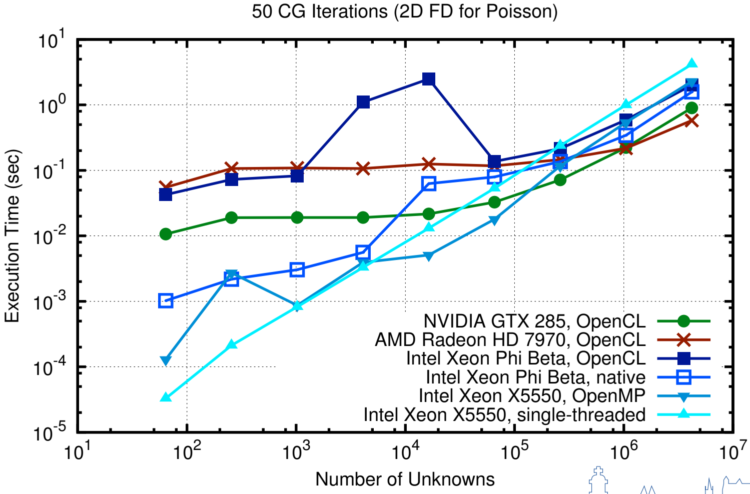
Some benchmarks - CG solver



Some benchmarks - CG solver

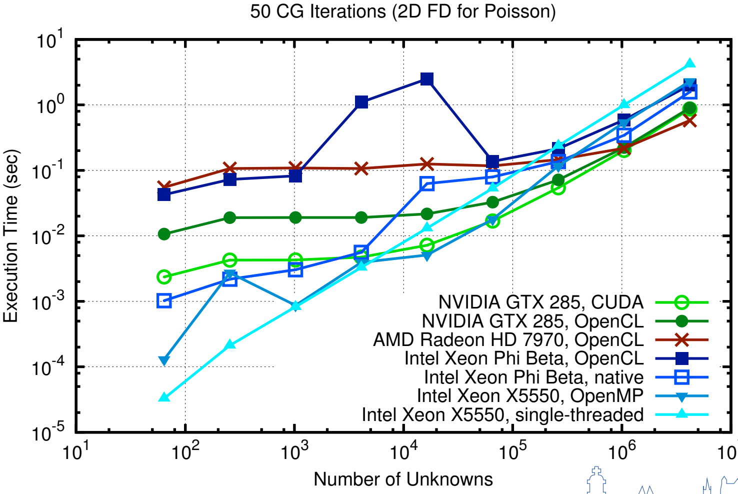


Some benchmarks - CG solver



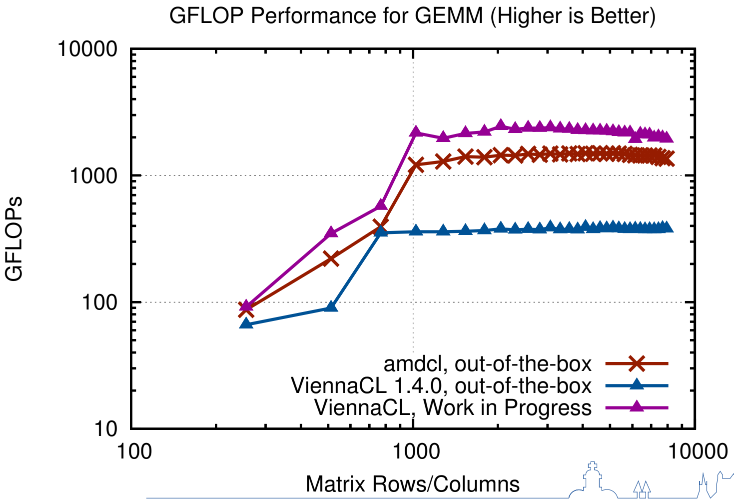
Performance

Some benchmarks - CG solver



Some benchmarks - Matrix-Matrix Multiplication

Auto-tuning environment (AMD Radeon HD 7970, single precision)



Summary

What have we learned?

- What are subvectors/submatrices and how to use them
- How to eliminate temporaries
- Expression templates and when they help us
- Interface to Eigen
- ViennaCL isn't optimized for small vectors/matrices
- Performance bottleneck
- Overview of ViennaCL performance

`ViennaCL`: Behind the curtain

ViennaCL: Behind the curtain

What to expect

- Backends
- OpenCL kernel management
- Extending ViennaCL
- ViennaCL and OpenGL
- Summary

ViennaCL: Behind the curtain

Backends

- There is more than OpenCL
- CUDA from NVIDIA
- OpenACC
- Each framework has advantages and disadvantages

Backends

OpenCL

```
const char *kernel_string =
"__kernel void mykernel(__global double *buffer) {
  buffer[get_global_id(0)] = 42.0;
};";

int main() {
  ...
  cl_program my_prog = clCreateProgramWithSource(
         my_context,1,&kernel_string,&source_len,&err);
  clBuildProgram(my_prog,0,NULL,NULL,NULL,NULL);
  cl_kernel my_kernel = clCreateKernel(my_prog,
                         "mykernel",&err);
  clSetKernelArg(my_kernel,0,sizeof(cl_mem),&buffer);
  clEnqueueNDRangeKernel(queue,my_kernel,1,NULL,
                &global_size,&local_size,0,NULL,NULL);
}
```

Additional boilerplate code required (low-level API)

Broad hardware support (separate SDKs)

No more development effort from NVIDIA

NVIDIA CUDA

```
// GPU kernel:
__global__ void kernel(double *buffer)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  buffer[idx] = 42.0;
}

// host code:
int main()
{
  ...
  cudaMalloc((void**)&buffer,size);
  kernel<<<blocknum, blockdim>>>(buffer);
  ...
}
```

Almost no additional code required

Vendor-lock

Relies on `nvcc` being available

Backends

OpenACC

```
void func(...) {
  #pragma acc data pcopyin(A[0:size][0:size])
  {
    #pragma acc kernels loop
    for(int i=0; i< size; i++)
      for(int j=0; j < size; j++)
        A[i][j] = 42;
  }
}

int main()
{
  double A[1337][1337];
  func(A);
}
```

Simple OpenMP-type pragma annotations

Compiler support?

Insufficient control over memory transfers?

Backends

What to use?

Why choose one when we can support all?

ViennaCL has a backend layer

Backend is responsible for hardware interaction

Not only OpenCL anymore

Since ViennaCL 1.4.0

Different backends supported

OpenCL

OpenMP

CUDA

Backends

Backend support has to be enabled explicitly

```
viennacl::vector<float> v1, v2;
v1 += v2;
```

CPU used!

Backends

Backend support has to be enabled explicitly

```
#define VIENNACL_WITH_OPENCL

viennacl::vector<float> v1, v2;
v1 += v2;
```

Now we are using OpenCL

Backends

Lets take a look!

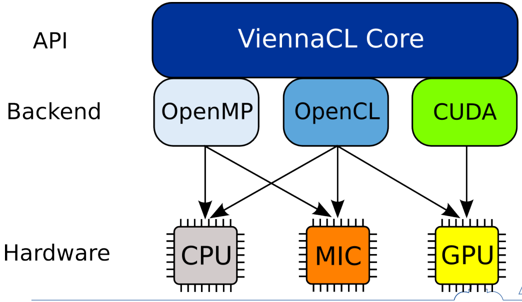
Vector Addition

Memory buffers can switch memory domain at runtime

```
void avbv(...) { // x = y + z
  switch (active_handle_id(x))
  {
    case MAIN_MEMORY:
      host_based::avbv(...);
      break;
    case OPENCL_MEMORY:
      opencl::avbv(...);
      break;
    case CUDA_MEMORY:
      cuda::avbv(...);
      break;
    default:
      raise_error();
  }
}
```

Memory Buffer Migration

```
vector<double> x = zero_vector<double>(42);

memory_types src_memory_loc = memory_domain(x);
switch_memory_domain(x, MAIN_MEMORY);
/* do work on x in main memory here */
switch_memory_domain(x, src_memory_loc);
```



Memory buffer switching at runtime

```
#define VIENNACL_WITH_OPENCL
#define VIENNACL_WITH_OPENMP

viennacl::vector<float> v1, v2;

switch_memory_domain(v1, MAIN_MEMORY);
switch_memory_domain(v2, MAIN_MEMORY);

v1 += v2; \\ working on CPU with OpenMP

switch_memory_domain(v1, OPENCL_MEMORY);
switch_memory_domain(v2, OPENCL_MEMORY);

v1 += v2; \\ working on GPU with OpenCL
```

OpenCL kernel management

Best kernel implementations depend on target hardware

NVIDIA, AMD, Intel

Best work group size depends on target hardware

	NVIDIA				AMD			
	32	64	128	256	32	64	128	256
64	191	151	174	193	324	262	256	249
128	194	177	**195**	214	357	290	**272**	**247**
256	161	164	195	214	307	264	256	248
512	**145**	157	198	211	282	255	258	253

Execution times for sparse matrix-vector product in milliseconds

OpenCL kernel management

Kernel parameter tuning

Default number of work groups = 128

Default number of work items per work group = 128

Automatic tuning environment ⇒ XML file

How to use kernel parameters

```
using namespace viennacl;
using viennacl::io;

read_kernel_parameters< vector<float> >
    ("float_vector_parameters.xml");
read_kernel_parameters< matrix<float> >
    ("float_matrix_parameters.xml");
read_kernel_parameters< compressed_matrix<float> >
    ("float_sparse_parameters.xml");
```

OpenCL kernel management

ViennaCL expression template don't cover all operations

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

Automated kernel generation

Supported since ViennaCL 1.3.0

Experimental support

Symbolic variables

Operation is defined with C++ symbolic variables

Custom kernel object is generated

OpenCL kernel management

Automated kernel generation

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

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

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

OpenCL kernel management

Automated kernel execution

```
viennacl::vector<NumericType> x, y, z;
viennacl::matrix<NumericType> A;

// fill data here

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

Extending ViennaCL

Not Everything Covered by ViennaCL

Complicated vector expressions in a single compute kernel

Direct OpenCL Kernel Handling is a Pain

```
const char * my_kernel_sources =
"__kernel void element_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);  \n"
"                    i < size;  \n"
"                    i += get_global_size(0))\n"
"    result[i] = vec1[i] * vec2[i];\n"
"};\n";
```

The OpenCL Way (error checks and casts omitted)

```
size_t source_len = std::string(my_compute_program).length();
cl_program my_prog = clCreateProgramWithSource(my_context, 1,
                         &my_kernel_sources, &source_len, &err);
err = clBuildProgram(my_prog, 0, NULL, NULL, NULL, NULL);

const char * kernel_name = "element_prod";
cl_kernel my_kernel = clCreateKernel(my_prog, kernel_name, &err);

err = clSetKernelArg(my_kernel, 0, sizeof(cl_mem), &mem_vec1);
err = clSetKernelArg(my_kernel, 1, sizeof(cl_mem), &mem_vec2);
err = clSetKernelArg(my_kernel, 2, sizeof(cl_mem), &mem_result);
err = clSetKernelArg(my_kernel, 3, sizeof(unsigned int), &vsize);
err = clEnqueueNDRangeKernel(queues[0], my_kernel, 1, NULL,
                       &global_size, &local_size, 0, NULL, NULL);
```

Issues

Access `my_kernel` at some other location in an application?

What to do with `my_prog`?

Extending ViennaCL

The ViennaCL Way (namespaces omitted)

```
program & my_prog =
  current_context().add_program(my_kernel_sources,
                                "my_program");
kernel & my_kernel = my_prog.add_kernel("element_prod");
enqueue(my_kernel(vec1, vec2, result, vec1.size()) );
```

At any other Location within the Application

```
kernel & my_kernel = get_kernel(
    "my_program", "element_prod");
viennacl::ocl::enqueue(
    my_kernel(vec1, vec2, result, vec1.size()) );
```

Allows for Adding Missing Functionality Easily

A bit of OpenCL knowledge required

Extending ViennaCL

Integrate ViennaCL into User-Environment

User-provided context, queue and device

```
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);
```

Wrapping Memory Buffers

```
   cl_mem my_memory = ...;
   viennacl::vector<float> my_vec(my_memory, 10);
```

Use ViennaCL operations as usual

ViennaCL and OpenGL

ViennaCL and OpenGL

- Since OpenCL 1.1: OpenGL interoperability
- With own OpenCL context: easy task

Workflow

- Setup OpenGL and OpenCL
- Create OpenGL buffer and OpenCL memory object
- Pass OpenCL memory object to ViennaCL
- Do ViennaCL magic
- Use data in OpenGL

Setup OpenGL context (simple glut-glew magic)

```
glutInit(&argc, argv);

glutInitDisplayMode(...);
glutInitWindowPosition(100,100);
glutInitWindowSize(1600,800);
glutCreateWindow("CL - GL");

glewInit();
```

ViennaCL and OpenGL

Setup OpenCL context with OpenGL interoperability support

```
cl_context_properties properties[] = {
    CL_GL_CONTEXT_KHR, (cl_context_properties)
        glXGetCurrentContext(),
    CL_GLX_DISPLAY_KHR, (cl_context_properties)
        glXGetCurrentDisplay(),
    CL_CONTEXT_PLATFORM, (cl_context_properties)
        viennacl::ocl::get_platforms()[0].id(),
    0};

cl_device_id my_device =
    viennacl::ocl::current_device().id();

cl_context my_context = clCreateContext(properties, 1,
    &my_device, NULL, NULL, &err);
cl_command_queue my_queue = clCreateCommandQueue(
    my_context, my_device, 0, &err );
```

ViennaCL and OpenGL

Setup OpenCL context with OpenGL interoperability support

```
cl_context_properties properties[] = {
    CL_GL_CONTEXT_KHR, (cl_context_properties)
        glXGetCurrentContext(),
    CL_GLX_DISPLAY_KHR, (cl_context_properties)
        glXGetCurrentDisplay(),
    CL_CONTEXT_PLATFORM, (cl_context_properties)
        viennacl::ocl::get_platforms()[0].id(),
    0};

cl_device_id my_device =
    viennacl::ocl::current_device().id();

cl_context my_context = clCreateContext(properties, 1,
    &my_device, NULL, NULL, &err);
cl_command_queue my_queue = clCreateCommandQueue(
    my_context, my_device, 0, &err );
```

Setting up ViennaCL for our context

```
viennacl::ocl::setup_context(
    0,          // the ViennaCL context ID
    my_context, // our context with OpenGL
        interoperability
    my_device,  // the device we are working on
    my_queue ); // the command queue for our context

// tell ViennaCL that we want to use our context
viennacl::ocl::switch_context( 0 );
```

ViennaCL and OpenGL

Create OpenGL buffer and OpenCL memory object

```
glGenBuffers(1, &gl_buffer);
glBindBuffer( GL_PIXEL_UNPACK_BUFFER, gl_buffer );
glBufferData( GL_PIXEL_UNPACK_BUFFER, size, NULL,
    GL_DYNAMIC_COPY );

cl_mem cl_buffer = clCreateFromGLBuffer(my_context,
    CL_MEM_READ_WRITE, gl_buffer, &err);
```

ViennaCL magic

```
// Create viennacl vector from OpenCL memory
viennacl::vector<float> my_vec(cl_buffer, size);

// Acquire memory object for write read/write operation
clEnqueueAcquireGLObjects(my_queue, 1, &cl_buffer,
     0, NULL, NULL);

// copy CPU data to ViennaCL
viennacl::copy( tmp_vec, my_vec );
// doing some stuff
my_vec *= 0.5f;

// Release memory object
clEnqueueReleaseGLObjects(my_queue, 1, &cl_buffer,
     0, NULL, NULL);
```

ViennaCL magic

```
// Create viennacl vector from OpenCL memory
viennacl::vector<float> my_vec(cl_buffer, size);

// Acquire memory object for write read/write operation
clEnqueueAcquireGLObjects(my_queue, 1, &cl_buffer,
     0, NULL, NULL);

// copy CPU data to ViennaCL
viennacl::copy( tmp_vec, my_vec );
// doing some stuff
my_vec *= 0.5f;

// Release memory object
clEnqueueReleaseGLObjects(my_queue, 1, &cl_buffer,
     0, NULL, NULL);
```

ViennaCL and OpenGL

Lets take a look at this example

Summary

What have we learned?

- ViennaCL has different backends
- How to enable and use backends
- OpenCL management in ViennaCL
- How to use an own OpenCL kernel with ViennaCL
- How to provide own OpenCL contexts
- ViennaCL works with OpenGL!
- How to use ViennaCL to work with OpenGL

Acknowledgements

Contributors

Thomas Bertani
Evan Bollig
Philipp Grabenweger
Volodymyr Kysenko
Nikolay Lukash
Günther Mader
Vittorio Patriarca
Florian Rudolf
Astrid Rupp
Karl Rupp
Philippe Tillet
Markus Wagner
Josef Weinbub
Michael Wild

Summary

High-Level C++ Approach of ViennaCL

- Convenience of single-threaded high-level libraries (Boost.uBLAS)
- Header-only library for simple integration into existing code
- MIT (X11) license

http://viennacl.sourceforge.net/

Selected Features

- Backends: OpenMP, OpenCL, CUDA
- Iterative Solvers: CG, BiCGStab, GMRES
- Preconditioners: AMG, SPAI, ILU, Jacobi
- BLAS: Levels 1-3