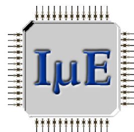
Lessons Learned in Developing the Linear Algebra Library ViennaCL

Florian Rudolf[1], Karl Rupp[1,2], Josef Weinbub[1]

http://karlrupp.net/



[1] Institute for Microelectronics
[2] Institute for Analysis and Scientific Computing
Technische Universität Wien, Austria

Workshop Programming of Heterogeneous Systems in Physics

July 15th, 2014

Outline

ViennaCL Overview and Internals

Tuning Potpourri: Iterative Solvers

Community Building

Development Infrastructure

Miscellaneous

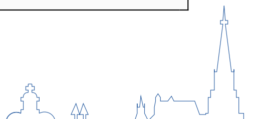
Summary

From Boost.uBLAS to ViennaCL

Consider Existing CPU Code (Boost.uBLAS)

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


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

/* Fill A, x, y here */

double val = inner_prod(x, y);
y += 2.0 * x;
A += val * outer_prod(x, y);

x = solve(A, y, upper_tag()); // Upper tri. solver

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

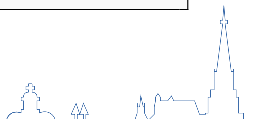
High-level code with syntactic sugar



Previous Code Snippet Rewritten with ViennaCL

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

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

/* Fill A, x, y here */

double val = inner_prod(x, y);
y += 2.0 * x;
A += val * outer_prod(x, y);

x = solve(A, y, upper_tag()); // Upper tri. solver

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

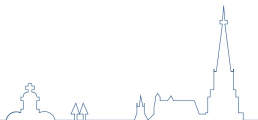
High-level code with syntactic sugar



ViennaCL in Addition Provides Iterative Solvers

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

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

/* Fill A, x, y here */

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

No Iterative Solvers Available in Boost.uBLAS...



From Boost.uBLAS to ViennaCL

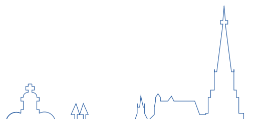
Thanks to Interface Compatibility

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

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

/* Fill A, x, y here */

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

Code Reuse Beyond GPU Borders

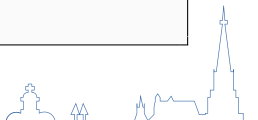
Eigen `http://eigen.tuxfamily.org/`

MTL 4 `http://www.mtl4.org/`



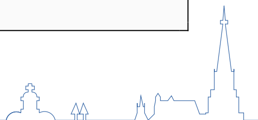
Generic CG Implementation (Sketch)

```
for (unsigned int i = 0; i < tag.max_iterations(); ++i)
{
  tmp = viennacl::linalg::prod(matrix, p);

  alpha     = ip_rr / inner_prod(tmp, p);
  result   += alpha * p;
  residual -= alpha * tmp;

  new_ip_rr = inner_prod(residual, residual);
  if (new_ip_rr / norm_rhs_squared < tag.tolerance())
    break;

  beta  = new_ip_rr / ip_rr;
  ip_rr = new_ip_rr;

  p = residual + beta * p;
}
```



Yesterday's Vector Addition Tutorial Revisited

```
#include <iostream>
#include "viennacl/vector.hpp"

typedef float   NumericT;
using namespace viennacl;

int main()
{
  std::size_t N = 128*1024;
  vector<NumericT> x = scalar_vector<NumericT>(N, 1.0);
  vector<NumericT> y = scalar_vector<NumericT>(N, 2.0);

  x += y;

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



About ViennaCL

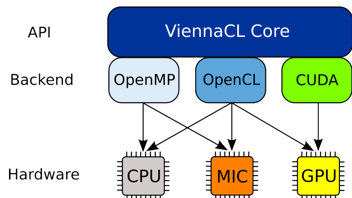
About

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

About ViennaCL

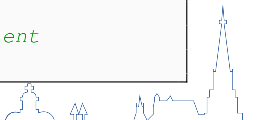
Basic Types

- scalar
- vector
- matrix, compressed_matrix, coordinate_matrix, ell_matrix, hyb_matrix

Data Initialization

- Using viennacl::copy()

```
     std::vector<double>      std_x(100);
   ublas::vector<double>    ublas_x(100);
viennacl::vector<double>      vcl_x(100);


for (size_t i=0; i<100; ++i){
    std_x[i] = rand();
  ublas_x[i] = rand();
    vcl_x[i] = rand();  //possible, inefficient
}
```



About ViennaCL

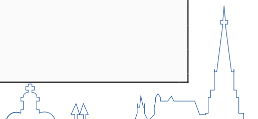
Basic Types

- scalar
- vector
- matrix, compressed_matrix, coordinate_matrix, ell_matrix, hyb_matrix

Data Initialization

- Using viennacl::copy()

```
     std::vector<double>       std_x(100);
   ublas::vector<double>     ublas_x(100);
viennacl::vector<double>       vcl_x(100);

/* setup of std_x and ublas_x omitted */

viennacl::copy(std_x.begin(), std_x.end(),
               vcl_x.begin());   //to GPU
viennacl::copy(vcl_x.begin(), vcl_x.end(),
               ublas_x.begin()); //to CPU
```



About ViennaCL

Basic Types

scalar

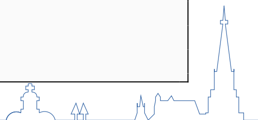
vector

matrix, compressed_matrix, coordinate_matrix, ell_matrix, hyb_matrix

Data Initialization

Using viennacl::copy()

```
     std::vector<std::vector<double> >    std_A;
   ublas::matrix<double>                ublas_A;
viennacl::matrix<double>                  vcl_A;

/* setup of std_A and ublas_A omitted */

viennacl::copy(std_A,
               vcl_A);    // CPU to GPU
viennacl::copy(vcl_A,
               ublas_A);  // GPU to CPU
```



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!

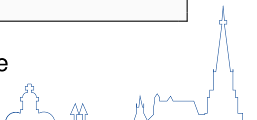
Expression Templates

```
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());
}
```

Vector Addition

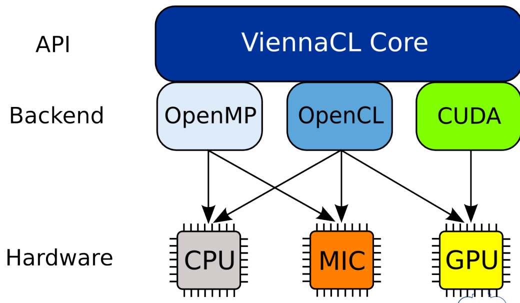
```
// x = y + z
void avbv(...) {
  switch (active_handle_id(x))
  {
    case MAIN_MEMORY:
      host_based::avbv(...);
      break;
    case OPENCL_MEMORY:
      opencl::avbv(...);
      break;
    case CUDA_MEMORY:
      cuda::avbv(...);
      break;
    default:
      raise_error();
  }
}
```

Memory buffers can switch memory domain at runtime



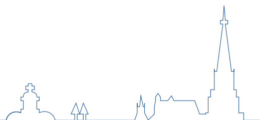
Memory Buffer Migration

```
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);
```



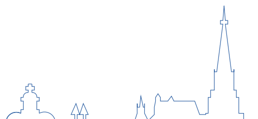
Generalizing compute kernels

```
  // x = y + z
  __kernel void avbv(
    double * x,

    double * y,

    double * z, uint size)
{
  for (size_t i  = get_global_id(0);
               i  < size;
               i += get_global_size())
    x[i] = y[i] + z[i];

}
```



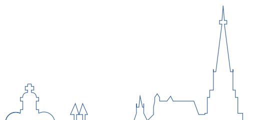
Generalizing compute kernels

```
  // x = a * y + b * z
  __kernel void avbv(
    double * x,
    double a,
    double * y,
    double b,
    double * z, uint size)
{
  for (size_t i  = get_global_id(0);
              i  < size;
              i += get_global_size())
    x[i] = a * y[i] + b * z[i];

}
```



Generalizing compute kernels

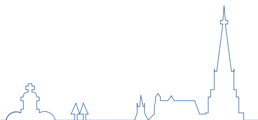
```
  // x[4:8] = a * y[2:6] + b * z[3:7]
  __kernel void avbv(
    double * x, uint off_x,
    double a,
    double * y, uint off_y,
    double b,
    double * z, uint off_z, uint size)
{
  for (size_t i  = get_global_id(0);
              i  < size;
              i += get_global_size())
    x[off_x + i] = a * y[off_y + i] + b * z[off_z + i];

}
```



Generalizing compute kernels

```
  // x[4:2:8] = a * y[2:2:6] + b * z[3:2:7]
  __kernel void avbv(
    double * x, uint off_x, uint inc_x,
    double a,
    double * y, uint off_y, uint inc_y,
    double b,
    double * z, uint off_z, uint inc_z, uint size)
{
  for (size_t i  = get_global_id(0);
              i  < size;
              i += get_global_size())
    x[off_x + i * inc_x] =  a * y[off_y + i * inc_y]
                          + b * z[off_z + i * inc_z];
}
```

No penalty on GPUs because FLOPs are for free



Outline

ViennaCL Overview and Internals

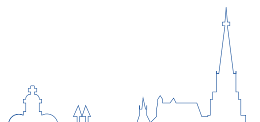
Tuning Potpourri: Iterative Solvers

Community Building

Development Infrastructure

Miscellaneous

Summary



Conjugate Gradients

Pseudocode

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i A p_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

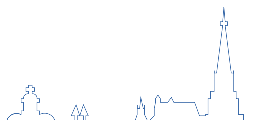
EndFor

BLAS-based Implementation

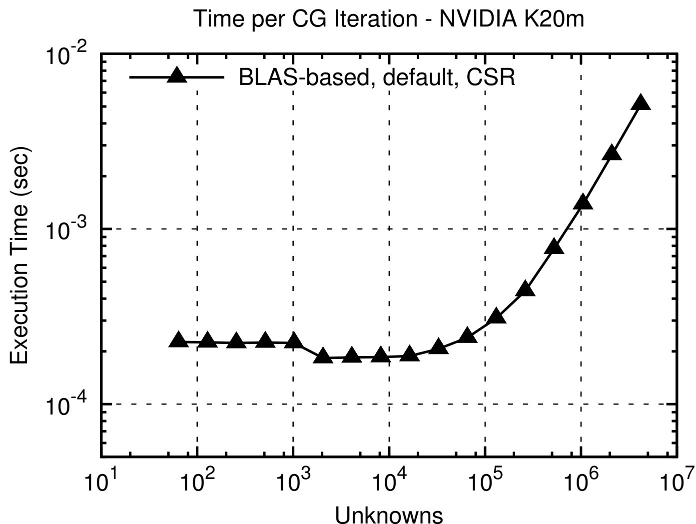
-
SpMV, AXPY
For $i = 0$ until convergence

1. SpMV $\leftarrow$ No caching of Ap_i
2. DOT $\leftarrow$ Global sync!
3. -
4. AXPY
5. AXPY $\leftarrow$ No caching of r_{i+1}
6. DOT $\leftarrow$ Global sync!
7. -
8. AXPY

EndFor



Conjugate Gradients



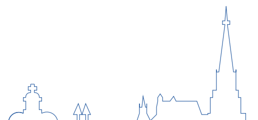
(2D Finite Difference Discretization)

Conjugate Gradient Optimizations

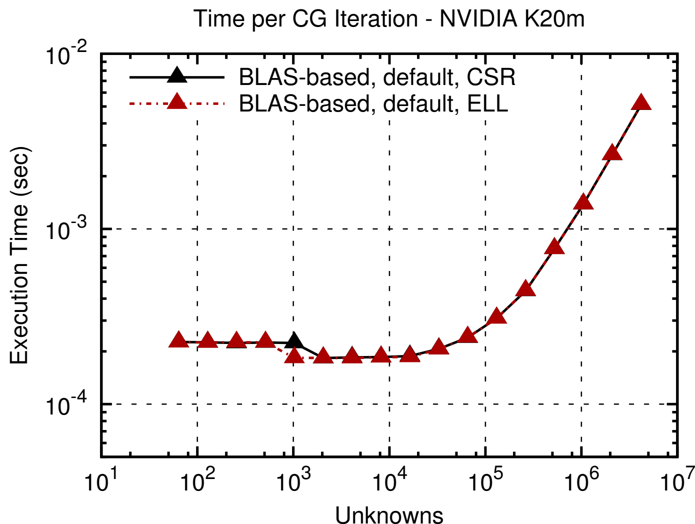
Optimization 1

- Get best performance out of SpMV
- Compare different sparse matrix types

Cf.: N. Bell: Implementing sparse matrix-vector multiplication on throughput-oriented processors. *Proc. SC '09*

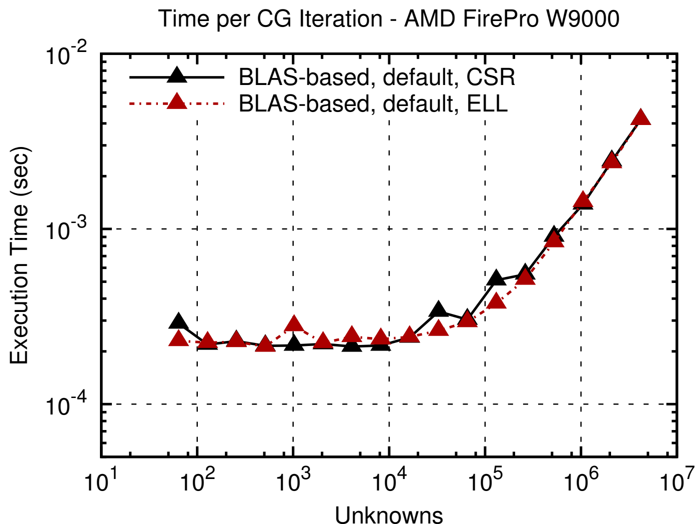


Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients

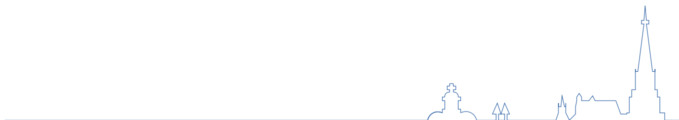


(2D Finite Difference Discretization)

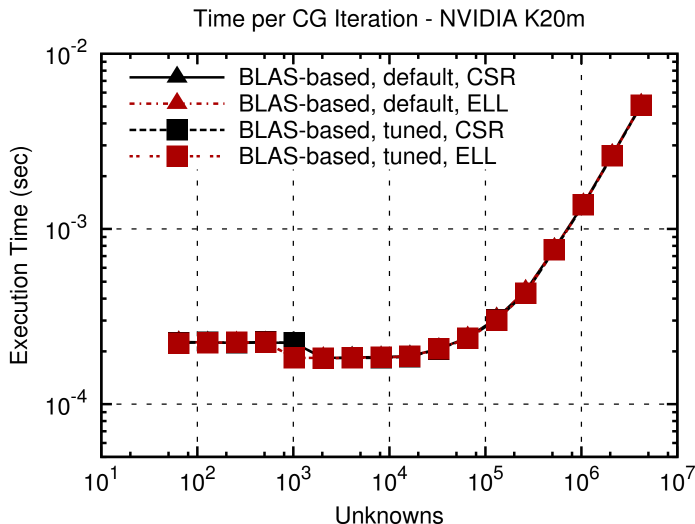
Conjugate Gradient Optimizations

Optimization 2

Optimize kernel parameters for each operation

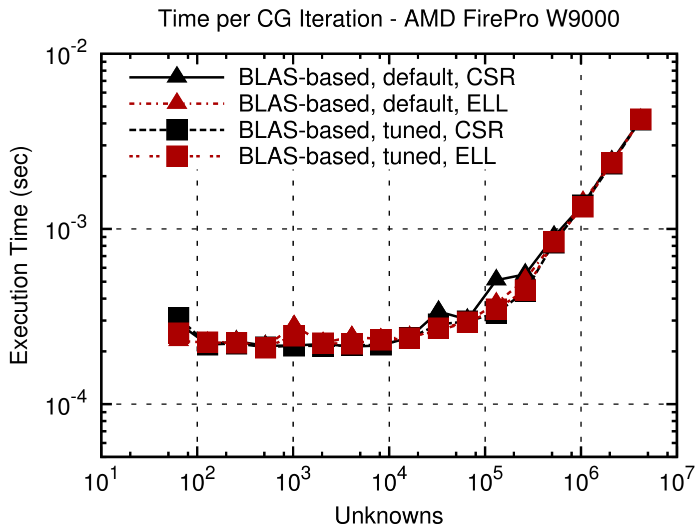


Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradient Optimizations

Optimization 3: Rearrange the algorithm

- Remove unnecessary reads
- Remove unnecessary synchronizations
- Use custom kernels instead of standard BLAS

Conjugate Gradients

Standard CG

Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 0$ until convergence

1. Compute and store Ap_i
2. Compute $\langle p_i, Ap_i \rangle$
3. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
4. $x_{i+1} = x_i + \alpha_i p_i$
5. $r_{i+1} = r_i - \alpha_i Ap_i$
6. Compute $\langle r_{i+1}, r_{i+1} \rangle$
7. $\beta_i = \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
8. $p_{i+1} = r_{i+1} + \beta_i p_i$

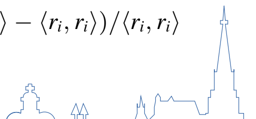
EndFor

Pipelined CG

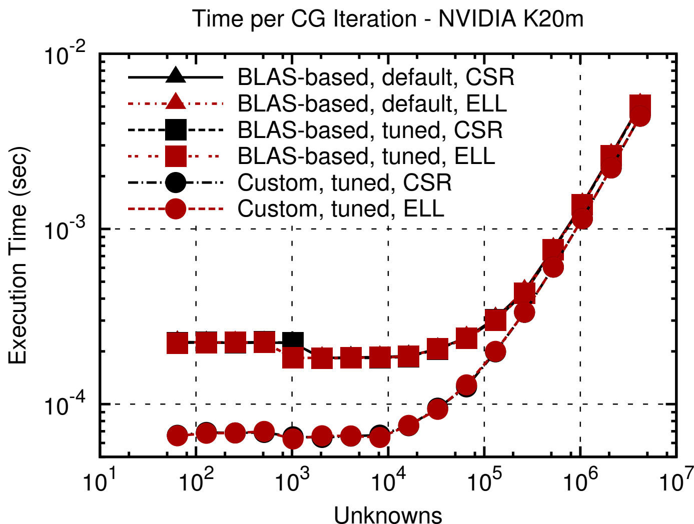
Choose x_0
$p_0 = r_0 = b - Ax_0$
For $i = 1$ until convergence

1. $i = 1$: Compute α_0, β_0, Ap_0
2. $x_i = x_{i-1} + \alpha_{i-1} p_{i-1}$
3. $r_i = r_{i-1} - \alpha_{i-1} Ap_i$
4. $p_i = r_i + \beta_{i-1} p_{i-1}$
5. Compute and store Ap_i
6. Compute $\langle Ap_i, Ap_i \rangle$, $\langle p_i, Ap_i \rangle$, $\langle r_i, r_i \rangle$
7. $\alpha_i = \langle r_i, r_i \rangle / \langle p_i, Ap_i \rangle$
8. $\beta_i = (\alpha_i^2 \langle Ap_i, Ap_i \rangle - \langle r_i, r_i \rangle) / \langle r_i, r_i \rangle$

EndFor

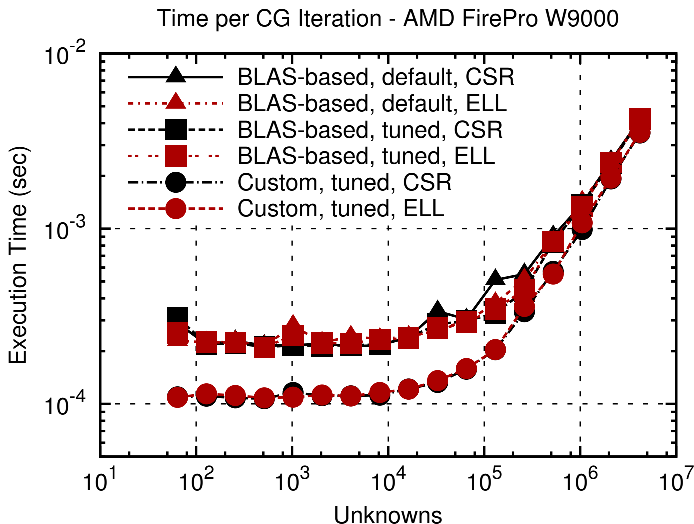


Conjugate Gradients



(2D Finite Difference Discretization)

Conjugate Gradients



(2D Finite Difference Discretization)

Outline

ViennaCL Overview and Internals

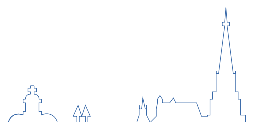
Tuning Potpourri: Iterative Solvers

Community Building

Development Infrastructure

Miscellaneous

Summary



Community Building

Recruiting Students

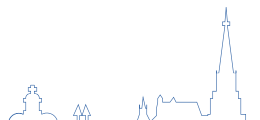
- Projects for Bachelor’s and Master’s theses
- Summer internships (Google Summer of Code)

Working with Students

- Benchmark for the project’s documentation
- Add-on components rather than ’critical’ features
- Continuous documentation

Project Organization

- Version control!
- Modularize early



Community Building

Public Communication

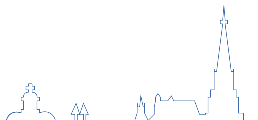
- Private communication kills the community
- Mailing lists (e.g. sourceforge), IRC, ...
- Public developer meetings

Announce Releases

- Social Media: Twitter, LinkedIn, Google+, ...
- Mailinglists: NA-Digest, etc.
- Provide a forum
- Changelog
- Information vs. Annoyance

Other

- Roadmap
- Provide repository write access early
- Be responsive (fast replies)



Outline

ViennaCL Overview and Internals

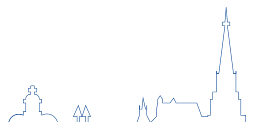
Tuning Potpourri: Iterative Solvers

Community Building

Development Infrastructure

Miscellaneous

Summary



Development Infrastructure

Technical Must-Haves

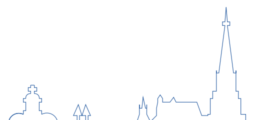
- Version control
- Build system (e.g. CMake)
- Nightly tests
- Compile cleanly at high warning levels

Recommended

- Continuous integration
- Doxygen
- Coding style guide

Tools

- Debugger
- Profiler



Outline

ViennaCL Overview and Internals

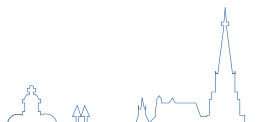
Tuning Potpourri: Iterative Solvers

Community Building

Development Infrastructure

Miscellaneous

Summary

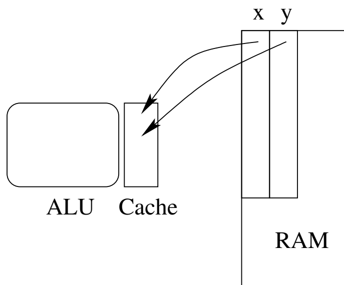


Expression Templates are Not Enough

Consider

```
u = x + y;
v = x - y;
```

Suboptimal performance with almost any library

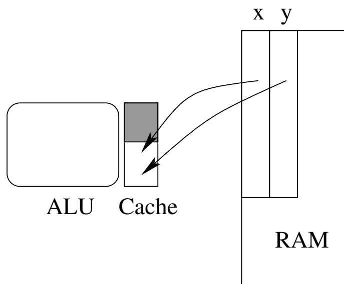


Expression Templates are Not Enough

Consider

```
u = x + y;
v = x - y;
```

Suboptimal performance with almost any library



Expression Template Limitations

Expression Templates are Not Enough

Consider

```
u = x + y;
v = x - y;
```

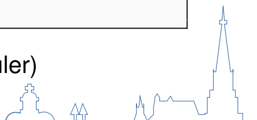
Suboptimal performance with almost any library

OpenCL Kernel Generation

Separate temporary avoidance from operation execution

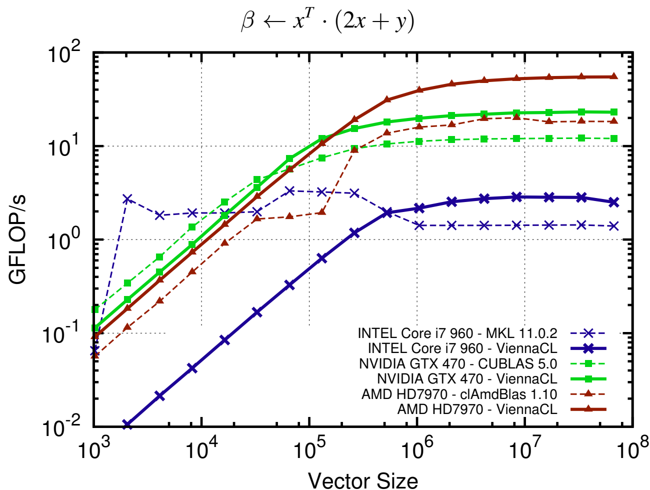
```
viennacl::kernel_fusion(true); //API not final!
u = x + y;
v = x - y;
...
viennacl::kernel_fusion(false);
```

Semi-transparent kernel fusion in preparation (scheduler)



Expression Template Limitations

Benchmark Results



Tillet *et al.*, HotPar '13

OpenCL on the CPU

Cost of a Function Call

Plain C: Tens of nanoseconds

OpenCL: Several microseconds

```
     std::vector<double>      std_x(100);
viennacl::vector<double>      vcl_x(100);

for (size_t i=0; i<100; ++i){
    std_x[i] = rand();
    vcl_x[i] = rand();  //possible, inefficient
}
```

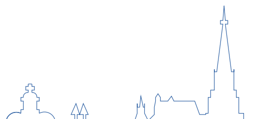
Host-based Execution

Required for serial code

Rich library set

Recommendation

Don’t use OpenCL for CPUs



C++ Loves Iterators

- Fundamental for STL
- Forward Iterator vs. Random Access Iterator

```
std::copy(x.begin(), x.begin() + 5, y.begin() + 4);
```

Iterators

C++ Loves Iterators

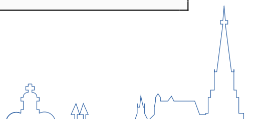
- Fundamental for STL
- Forward Iterator vs. Random Access Iterator

```
std::copy(x.begin(), x.begin() + 5, y.begin() + 4);
```

Massive Parallelism

- Forward Iterator is sequential by nature
- Only Random Access Iterator suitable for parallelism
- Simpler APIs:

```
x[range(0, 5)] = y[range(4, 9)];
```



Summary and Conclusion

ViennaCL

- Convenient high-level linear algebra library
- CUDA-, OpenCL-, and OpenMP-backends

http://viennacl.sourceforge.net/

Using Heterogeneous Systems

- Reuse software libraries
- ’Time to science’ most important

Slides

- Available in folder PHSP2014 at

http://github.com/karlrupp/slides