ViennaCL

GPU-accelerated Linear Algebra at the Convenience of the C++ Boost Libraries

Karl Rupp

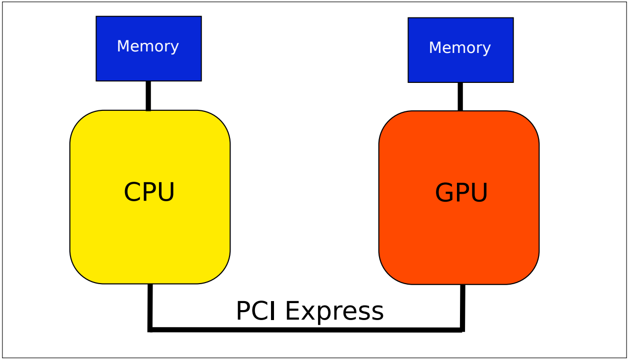
Mathematics and Computer Science Division
Argonne National Laboratory

based on previous work at

Technische Universität Wien, Austria

SIAM CSE, February 26th, 2012

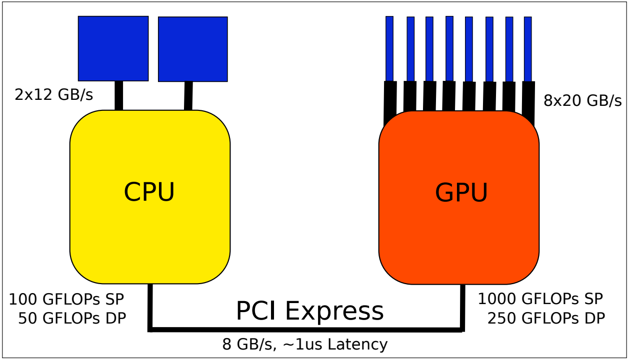
Computing Architecture Schematic



ViennaCL

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

ViennaCL

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

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

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?

GPUs: Library Aspects

Challenge: Hardware

- Portable performance
- Auto-tuning
- Testing requires many different machines

Challenge: Memory

- Allocation failures?
- Multi-GPU?
- PCI-Express bottleneck

Challenge: Programming

- Kernel language?
- Which low-level parameters to expose?

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/`

ViennaCL

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;
}
```

About ViennaCL

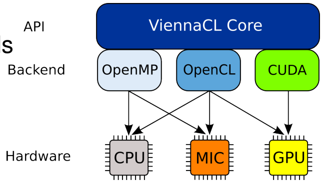
About

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

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
}
```

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
```

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());
}
```

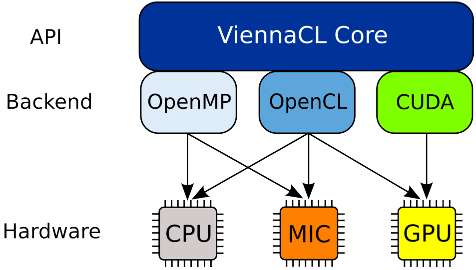
Vector Addition

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

}
```

Generalizing compute kernels

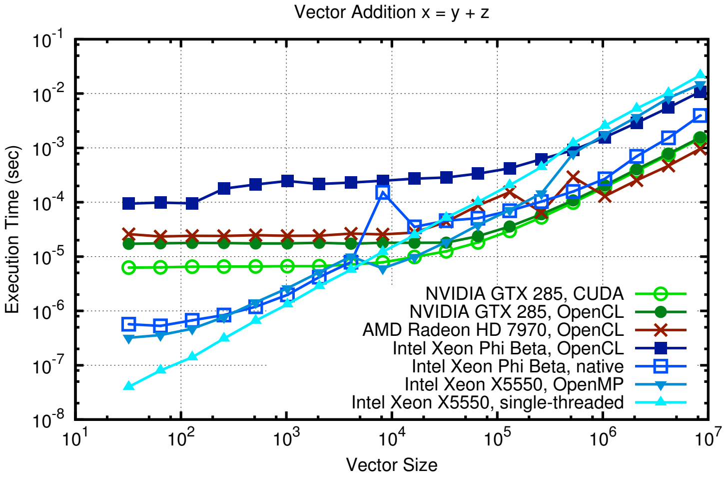
```
  // x[4:8] = a * y[2:6] + b * z[3:7]
  __kernel void avbv(
    double * x, uint off_x,
    double a,
    double * y, uint off_y,
    double b,
    double * z, uint off_z, uint size)
{
  i = get_global_id(0);
  for (size_t i=0; i<size; i += get_global_size())
    x[off_x + i] = a * y[off_y + i] +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

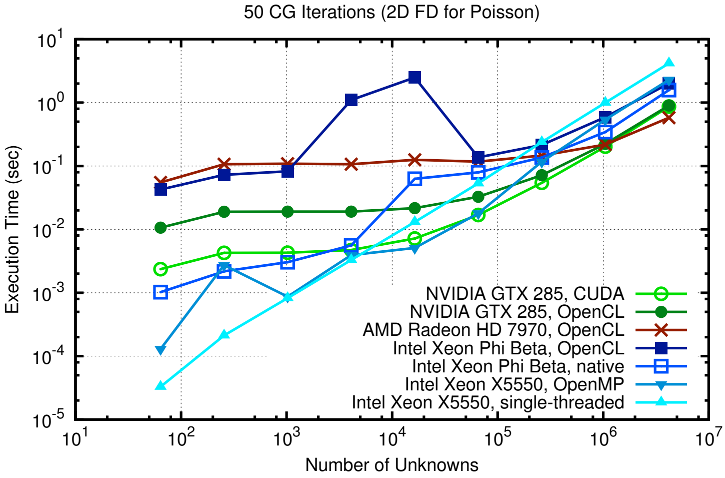
No penalty on GPUs because FLOPs are for free

Benchmarks



ViennaCL

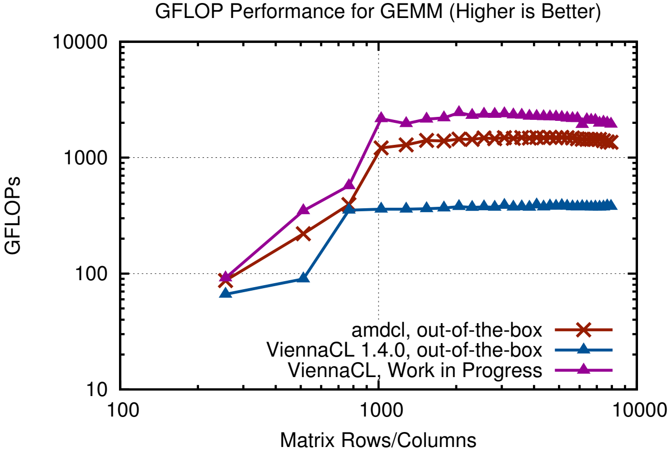
Benchmarks



ViennaCL

Matrix-Matrix Multiplication

Autotuning environment



(AMD Radeon HD 7970, single precision)

ViennaCL

Acknowledgements

Contributors

Thomas Bertani
Evan Bollig
Philipp Grabenweger
Volodymyr Kysenko
Nikolay Lukash
Günther Mader
Vittorio Patriarca
Florian Rudolf
Astrid 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