ViennaCL and PETSc Tutorial

Karl Rupp
rupp@mcs.anl.gov

Mathematics and Computer Science Division
Argonne National Laboratory

FEMTEC 2013
May 23th, 2013
Vienna Computing Library

http://viennacl.sourceforge.net/
Consider Existing CPU Code (Boost.uBLAS)

```cpp
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
From Boost.uBLAS to ViennaCL

Previous Code Snippet Rewritten with ViennaCL

```cpp
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

```cpp
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...
Thanks to Interface Compatibility

```cpp
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/](http://eigen.tuxfamily.org/)
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
Basic Types

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

Data Initialization

```cpp
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(); // (1)
// ublas_x[i] = rand(); // (2)
    vcl_x[i] = rand(); // (3)
```

(3) is fastest, right?
Basic Types

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

Data Initialization

Using viennacl::copy()

```cpp
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()

```cpp
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
```

Iterator concept doesn’t quite work on accelerators
Vector Addition

\[ x = y + z; \]

Temporaries are costly (particularly on GPUs)

Expression Templates

Limited expansion

Map to a set of predefined kernels

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

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

50 CG Iterations (2D FD for Poisson)

- NVIDIA GTX 285, CUDA
- NVIDIA GTX 285, OpenCL
- AMD Radeon HD 7970, OpenCL
- Intel Xeon Phi Beta, OpenCL
- Intel Xeon Phi Beta, native
- Intel Xeon X5550, OpenMP
- Intel Xeon X5550, single-threaded

Execution Time (sec) vs. Number of Unknowns
Matrix-Matrix Multiplication

Autotuning environment

GFLOP Performance for GEMM (Higher is Better)

GFLOPs vs. Matrix Rows/Columns

- amdcl, out-of-the-box
- ViennaCL 1.4.0, out-of-the-box
- ViennaCL, Work in Progress

(AMD Radeon HD 7970, single precision)
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
PETSc

Portable Extensible Toolkit for Scientific Computing
Obtaining PETSc

Linux Package Managers
Web: http://mcs.anl.gov/petsc, download tarball
Git: https://bitbucket.org/petsc/petsc
Mercurial: https://bitbucket.org/petsc/petsc-hg

Installing PETSc

```bash
$> cd /path/to/petsc/workdir
$> git clone \
  https://bitbucket.org/petsc/petsc.git \
  --branch master --depth 1
$> cd petsc

$> export PETSC_DIR=$PWD PETSC_ARCH=mpich-gcc-dbg
$> ./configure --with-cc=gcc --with-fc=gfortran \
  --download-fblas-lapack \
  --download-{mpich,ml,hypre}
```
**Portable Extensible Toolkit for Scientific Computing**

**Architecture**
- tightly coupled (e.g. XT5, BG/P, Earth Simulator)
- loosely coupled such as network of workstations
- GPU clusters (many vector and sparse matrix kernels)

**Software Environment**
- Operating systems (Linux, Mac, Windows, BSD, proprietary Unix)
- Any compiler
- Usable from C, C++, Fortran 77/90, Python, and MATLAB
- Real/complex, single/double/quad precision, 32/64-bit int

**System Size**
- 500B unknowns, 75% weak scalability on Jaguar (225k cores) and Jugene (295k cores)
- Same code runs performantly on a laptop

Free to everyone (BSD-style license), open development
Portable **Extensible** Toolkit for Scientific Computing

**Philosophy: Everything has a plugin architecture**

- Vectors, Matrices, Coloring/ordering/partitioning algorithms
- Preconditioners, Krylov accelerators
- Nonlinear solvers, Time integrators
- Spatial discretizations/topology

**Example**

Vendor supplies matrix format and associated preconditioner, distributes compiled shared library.

Application user loads plugin at runtime, no source code in sight.
Portable Extensible Toolkit for Scientific Computing

Toolset

algorithms
(parallel) debugging aids
low-overhead profiling

Composability

try new algorithms by choosing from product space
composing existing algorithms (multilevel, domain decomposition, splitting)

Experimentation

Impossible to pick the solver \textit{a priori}
PETSc’s response: expose an algebra of composition
keep solvers decoupled from physics and discretization
PETSc

Portable Extensible Toolkit for Scientific Computing

Computational Scientists
PyLith (CIG), Underworld (Monash), Magma Dynamics (LDEO, Columbia), PFLOTRAN (DOE), SHARP/UNIC (DOE)

Algorithm Developers (iterative methods and preconditioning)

Package Developers
SLEPc, TAO, Deal.II, Libmesh, FEniCS, PETSc-FEM, MagPar, OOFEM, FreeCFD, OpenFVM

Funding
Department of Energy
SciDAC, ASCR ISICLES, MICS Program, INL Reactor Program
National Science Foundation
CIG, CISE, Multidisciplinary Challenge Program

Documentation and Support
Hundreds of tutorial-style examples
Hyperlinked manual, examples, and manual pages for all routines
Support from petsc-maint@mcs.anl.gov
Flow Control for a PETSc Application

Main Routine

- Timestepping Solvers (TS)
- Nonlinear Solvers (SNES)
- Linear Solvers (KSP)
- Preconditioners (PC)

Application Initialization
Function Evaluation
Jacobian Evaluation
Postprocessing
PETSc Pyramid

PETSc Structure

- PETSc PDE Application Codes
  - ODE Integrators
  - Visualization
  - Nonlinear Solvers
    - Linear Solvers
    - Preconditioners + Krylov Methods
    - Object-Oriented
      - Matrices, Vectors, Indices
    - Grid Management
  - Profiling Interface
    - Computation and Communication Kernels
      - MPI, MPI-IO, BLAS, LAPACK
To evaluate a local function $f(x)$, each process requires

- its local portion of the vector $x$
- its **ghost values**, bordering portions of $x$ owned by neighboring processes
### DMDA Global Numberings

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
</tr>
<tr>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>5  6  7</td>
<td>8  9</td>
</tr>
<tr>
<td>0  1  2</td>
<td>3  4</td>
</tr>
</tbody>
</table>

**Natural numbering**

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td>6  7  8</td>
<td>13 14</td>
</tr>
<tr>
<td>3  4  5</td>
<td>11 12</td>
</tr>
<tr>
<td>0  1  2</td>
<td>9  10</td>
</tr>
</tbody>
</table>

**PETSc numbering**
**Global**: Each vertex has a unique id, belongs on a unique process

**Local**: Numbering includes vertices from neighboring processes

These are called ghost vertices

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>X X X</td>
<td>X X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X X X</td>
<td>X X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12 13 14</td>
<td>15 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 9 10</td>
<td>11 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4 5 6</td>
<td>7 X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 2</td>
<td>3 X</td>
</tr>
</tbody>
</table>

**Global numbering**

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td></td>
<td></td>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6 7 8</td>
<td>13 14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3 4 5</td>
<td>11 12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 1 2</td>
<td>9 10</td>
</tr>
</tbody>
</table>
 Wouldn’t it be nice if we could just write our code for the natural numbering?

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
</tr>
<tr>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>5 6 7</td>
<td>8 9</td>
</tr>
<tr>
<td>0 1 2</td>
<td>3 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
</tr>
<tr>
<td>6 7 8</td>
<td>13 14</td>
</tr>
<tr>
<td>3 4 5</td>
<td>11 12</td>
</tr>
<tr>
<td>0 1 2</td>
<td>9 10</td>
</tr>
</tbody>
</table>

Natural numbering

PETSc numbering
Wouldn’t it be nice if we could just write our code for the natural numbering?

Yes, that’s what `DMDAVecGetArray()` is for.

**DMDA offers local callback functions**

- `FormFunctionLocal()`, set by `DMDASetLocalFunction()`
- `FormJacobianLocal()`, set by `DMDASetLocalJacobian()`

**Evaluating the nonlinear residual \( F(x) \)**

Each process evaluates the local residual

PETSc assembles the global residual automatically

Uses `DMLocalToGlobal()` method
p-Bratu Equation

2-dimensional model problem

\[-\nabla \cdot (|\nabla u|^{p-2}\nabla u) - \lambda e^u - f = 0, \quad 1 \leq p \leq \infty, \quad \lambda < \lambda_{\text{crit}(p)}\]

Singular or degenerate when \(\nabla u = 0\), turning point at \(\lambda_{\text{crit}}\).
The $p$-Bratu Equation

$p$-Bratu Equation

2-dimensional model problem

\[-\nabla \cdot (|\nabla u|^{p-2}\nabla u) - \lambda e^u - f = 0, \quad 1 \leq p \leq \infty, \quad \lambda < \lambda_{\text{crit}}(p)\]

Singular or degenerate when $\nabla u = 0$, turning point at $\lambda_{\text{crit}}$.

Regularized Variant

Remove singularity of $\eta$ using a parameter $\varepsilon$:

\[-\nabla \cdot (\eta \nabla u) - \lambda e^u - f = 0\]

\[\eta(\gamma) = (\varepsilon^2 + \gamma) \frac{p-2}{2} \quad \gamma(u) = \frac{1}{2} |\nabla u|^2\]

Physical interpretation: diffusivity tensor flattened in direction $\nabla u$
Conclusions

PETSc Can Help You

- Solve algebraic and DAE problems in your application area
- Rapidly develop efficient parallel code, can start from examples
- Develop new solution methods and data structures
- Debug and analyze performance
- Advice on software design, solution algorithms, and performance

petsc-{users,dev,maint}@mcs.anl.gov

You Can Help PETSc

- Report bugs and inconsistencies, or if you think there is a better way
tell us if the documentation is inconsistent or unclear
- Consider developing new algebraic methods as plugins, contribute if your idea works