A Discussion of Selected Vienna-Libraries for Computational Science

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

[1]MCS Division, Argonne National Laboratory, USA
[2]Institut für Mikroelektronik, TU Wien, Austria

Pre-Conference Handout Slides (Preview)

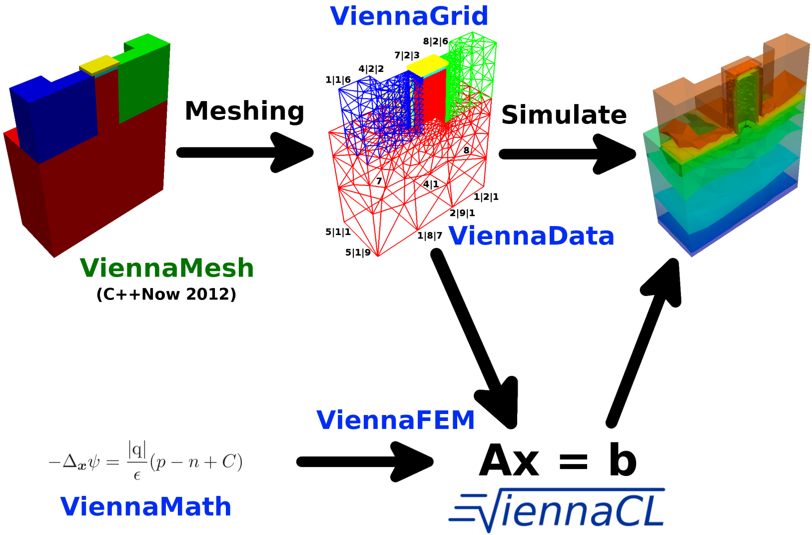
C++Now, May 15th, 2013

Contents

Part 0: What is this all about?

Simulation Flow

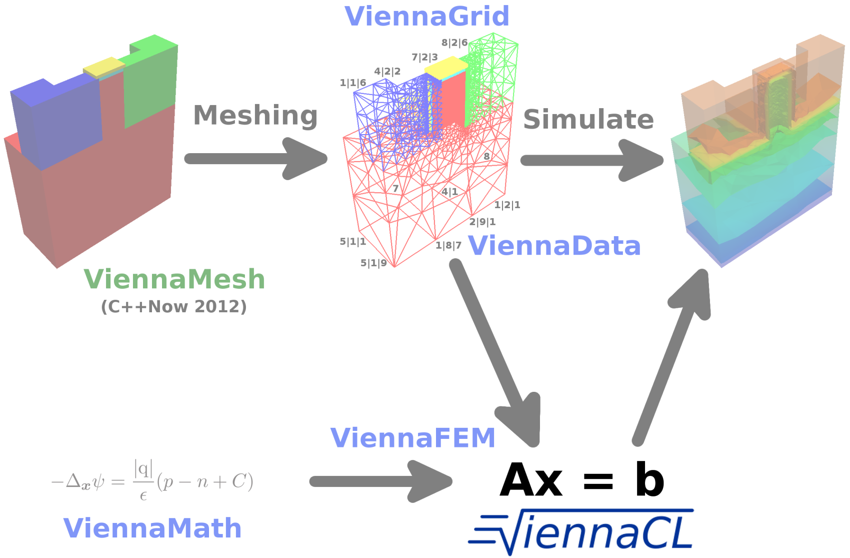
The Many Steps in Simulating a FinFET



Contents

Part 1: ViennaCL

Simulation Flow



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

From Boost.uBLAS to ViennaCL

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

From 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), 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/`

From Boost.uBLAS to ViennaCL

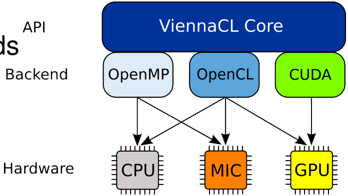
Generic CG Implementation (Sketch)

```
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

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
```

Internals

Vector Addition

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

Temporaries are costly (particularly on GPUs)

Expression Templates

Limited expansion

Map to a set of predefined kernels

```
vector_expressuib<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());
}
```

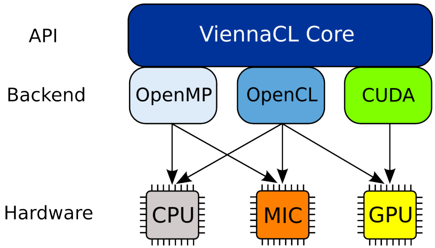
Vector Addition

```
// 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);

/* work on x in main memory here */

switch_memory_domain(x, src_memory_loc);
```

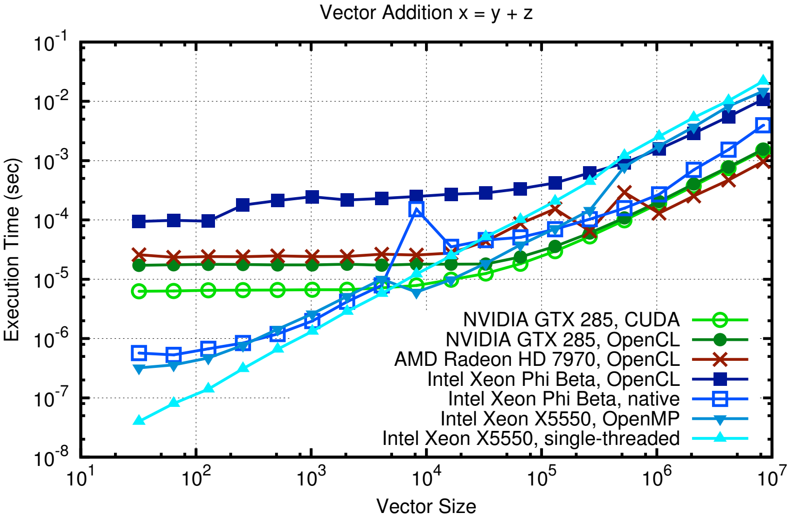


Generalizing compute kernels

```
  // 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)
{
  size_t i = get_global_id(0);
  for (; 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];
}
```

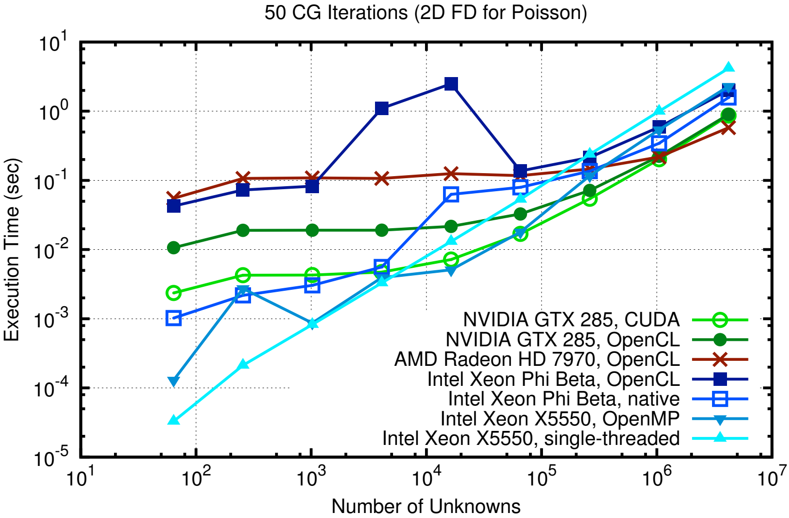
No penalty on GPUs because FLOPs are for free

Benchmarks



Vector Addition x = y + z
Execution Time (sec)
Vector Size
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

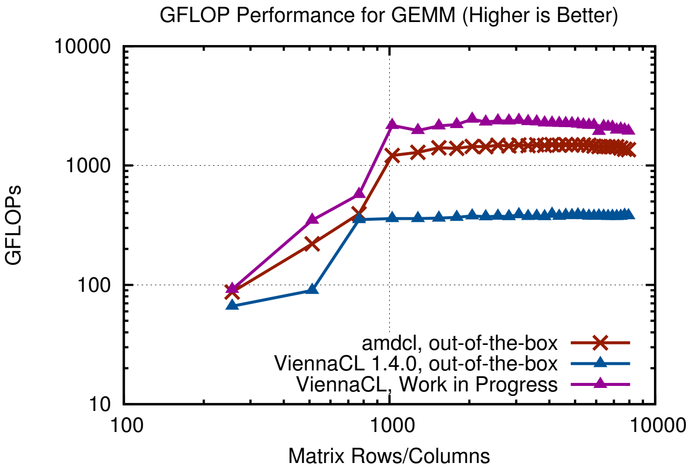
Benchmarks



50 CG Iterations (2D FD for Poisson)
Execution Time (sec)
Number of Unknowns
10^{1}
10^{0}
10^{-1}
10^{-2}
10^{-3}
10^{-4}
10^{-5}
10^{1}
10^{2}
10^{3}
10^{4}
10^{5}
10^{6}
10^{7}
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

Matrix-Matrix Multiplication

Autotuning environment

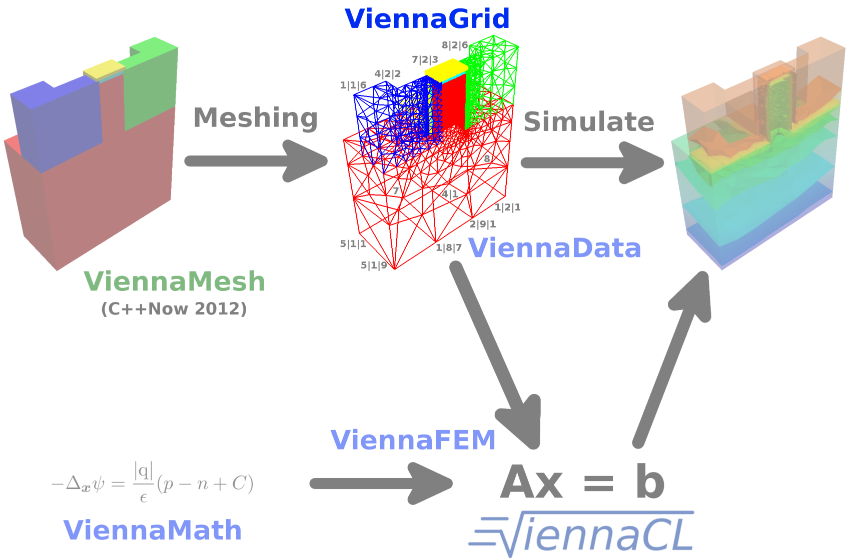


(AMD Radeon HD 7970, single precision)

Contents

Part 2: ViennaGrid

Simulation Flow



What about Boost.Geometry?

A Quick Look at Boost.Geometry

Concepts: Point, Segment, Linestring, Box, etc.

Algorithms: `distance()`, `intersects()`, `convex_hull()`, etc.

Central entity: Point

```
int a[3] = {1, 2, 3};
int b[3] = {2, 3, 4};

double d = boost::geometry::distance(a, b);
```

Why Boost.Geometry is Not Enough

What about 3D objects? Tetrahedra? Hexahedra?

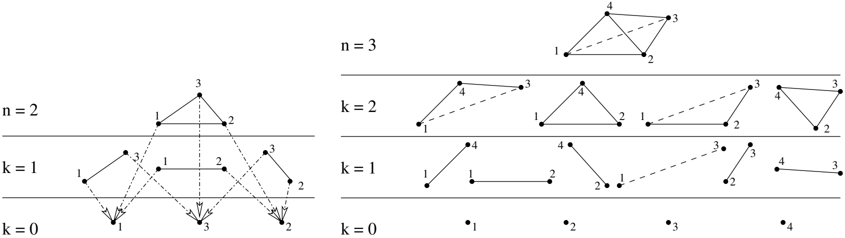
Traversal of boundary objects missing

No storage facilities for *many* objects

n-cell Concept

Concept of n-*cell*

Sub-k-cells of an n-cell



Separation of Geometry and Topology

Geometry: Euclidian space, coordinate system

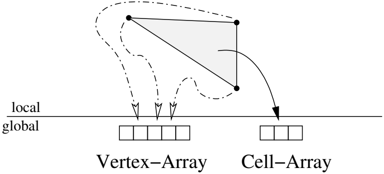
Topology: Connection between points (lines, triangles, etc.)

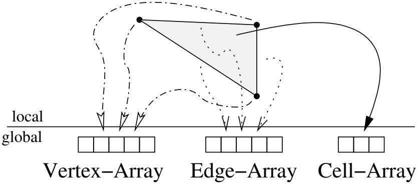
ViennaGrid Datastructure

Datastructure Requirements

Don't store boundary k-cells unnecessarily

Fast local iteration on k-cells





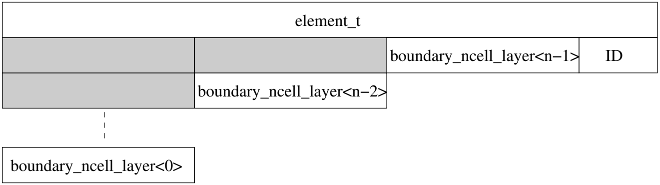
	Amount	Mem/Obj.	Total Mem.
Vertices	4913	24 B	115 KB
Edges	31024	16 B	485 KB
Facets	50688	48 B	2376 KB
Cells	24576	112 B	2688 KB
Total			**5664 KB**

	Amount	Mem/Obj.	Total Mem.
Vertices	4913	24 B	115 KB
Edges	0	-	0 KB
Facets	0	-	0 KB
Cells	24576	32 B	768 KB
Total			**883 KB**

n-cell Implementation

Implementation of `element_t`

- Recursive inheritance from boundary layer of dimension $n-1$
- Tag dispatching to enable/disable topological layers




```
template <typename ConfigType, typename ElementTag>
class element_t :
  public boundary_ncell_layer<ConfigType, ElementTag, ElementTag::dim-1>,
  public result_of::element_id_handler<ConfigType, ElementTag>::type
```

User API Design Goals

- STL-style, reuse conventions
- Allow index-based traversal
- Avoid common C++ pitfalls (e.g. template member functions)

```
//iteration over all vertices in the domain:
typedef result_of::ncell_range<DomainType, 0>::type  VertexRange;
typedef result_of::iterator<VertexRange>::type       VertexIterator;

VertexRange vertices = ncells<0>(domain);
for (VertexIterator it  = vertices.begin();
                    it != vertices.end();
                  ++it)
{
  // do something with each vertex here
}
```

- *Ranges* provide iterators over n-cells
- Extendible

User API Design Goals

Boundary iteration: $k < n$

Coboundary iteration: $k > n$

```
//iteration over all triangles of a vertex
typedef result_of::ncell_range<VertexType, 2>::type  TriangleRange;
typedef result_of::iterator<TriangleRange>::type     TriangleIterator;

TriangleRange triangles = ncells<0>(vertex, domain);
for (TriangleIterator it  = triangles.begin();
                      it != triangles.end();
                    ++it)
{
  // do something with each triangle here
}
```

Coboundary information not a-priori available in datastructure

Built and cached at first request

Domain Configuration

Top Level Configuration of Domain

Mostly a collection of tags

Predefined configuration classes for common cases

ViennaGrid 1.0.x: Supplemented by global customizations

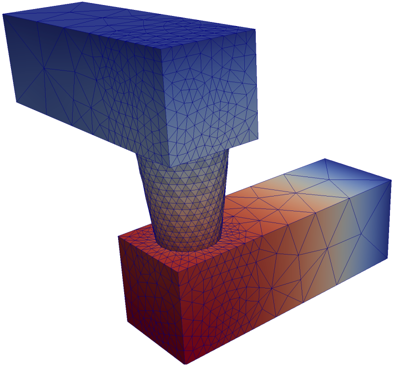
```
struct triangular_3d
{
  typedef double               numeric_type;
  typedef cartesian_cs<3>      coordinate_system_tag;
  typedef triangle_tag         cell_tag;
};

result_of::domain<triangular_3d>::type     Domain;
```

Work in progress: Everything in a single config class

ViennaGrid Features

Other Features

- Segments
- I/O: VTK, various mesh-formats
- Voronoi information
- Refinement



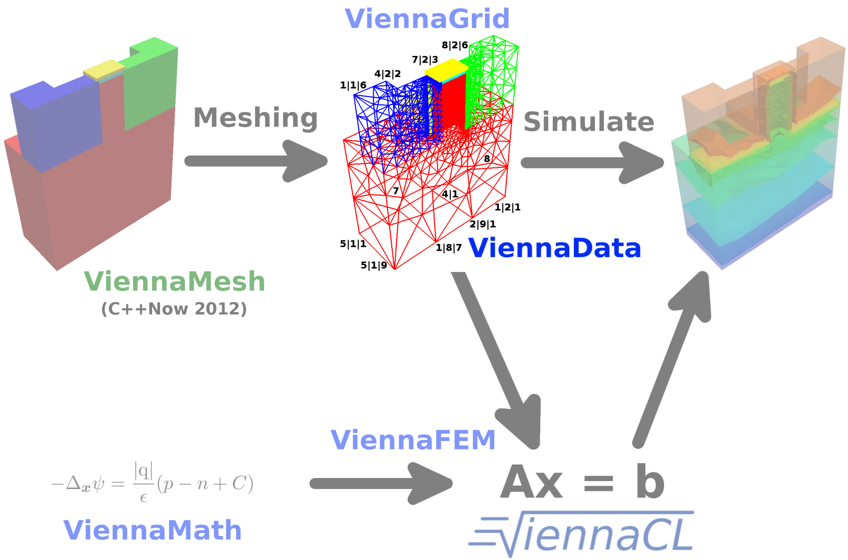
Work in Progress

- PLC, hybrid meshes, multigrid

Contents

Part 3: ViennaData

Simulation Flow



Data Storage

Plain Object-Oriented Approach?

```
struct Triangle {
 PointType a, b, c;
 bool is_on_boudary;
 double rho; };   //e.g. specific mass
```

Pros and Cons

Data is directly stored with the object

Each (!) triangle carries a boolean flag

Reusability reduced ⇒ better rename to `TriangleWithMaterial`

Monolithic, don't do this!

Data Storage

Store Data with Mesh:

```
class Mesh {
 vector<Triangle> triangles;
 std::map<size_t, bool>  boudary_map; //no memory wasted
 vector<double> rho_for_triangles;   };
```

Pros and Cons

- `Triangle` is still reusable
- `Mesh` class has to handle data storage complexity
- Each additional data requires a change of `Mesh` class
- `Mesh` object has to be passed to all modules

Monolithic, don't do this!

Approach by ViennaData

- Introduction of a hidden data container
- Data is stored in a map-like manner using keys

```
//generic interface:
viennadata::access<KeyType, ValueType>(key)(obj);

//boundary flag and specific mass retrieval for triangle:
bool on_boundary = access<BoundaryKeyType, bool>(boundary_key)(triangle);
double       rho = access<MassKeyType,   double>(    mass_key)(triangle);
```

Pros and Cons

- `Triangle` and `Mesh` are still reusable
- Can be used with arbitrary objects (third-party libraries)
- Unified interface for data access

Generic Interface

```
viennadata::access<KeyType, ValueType>(key)(obj);
```

Default Storage Scheme

If nothing is known about the object:

```
std::map<ObjectType *,
         std::map<KeyType, ValueType> >
```

The compiler creates such a map for each of the following:

```
access<long, double>(42)(triangle);
access<char, double>('b')(triangle);
access<std::string, double>("mass")(triangle);
access<std::string, double>("mass")(vertex);
```

Performance Considerations

- $\mathcal{O}(\log N) + \mathcal{O}(\log K)$ access time
- N ... objects of same type (e.g. triangles)
- K ... keys of same type
- Usually too slow in a high-performance setting

Can We Do Better?

- In general: No
- In certain situations: Yes

Type-based Key Dispatch

- Only one key per type
- $\mathcal{O}(\log N)$ access time

```
access<mass_key, double>(mass_key())(triangle);

// or:
access<mass_key, double>()(triangle);
```

Internal Datastructure

```
std::map<ObjectType *, ValueType>
```

- Transparent to user
- Simple preprocessor macro for activation

Numeric IDs for Objects

- Only one key per type
- $\mathcal{O}(K)$ access time
- $\mathcal{O}(1)$ access time with type-based key dispatch

```
access<std::string, double>("mass")(triangle);

// with type-based key dispatch:
access<mass_key, double>()(triangle);
```

Internal Datastructure

```
std::vector<ValueType>
```

- Transparent to user
- Simple preprocessor macro for activation
- Overload generic `id()` accessor

Benchmarking ViennaData

ID-based access to data via ViennaData (class `LightWeight`)

OOP-style storage in classes with payload

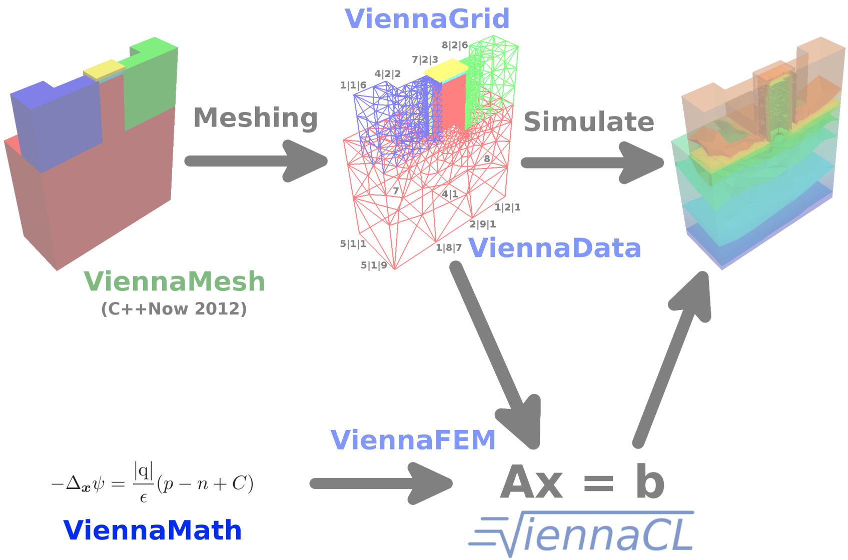
```
template <size_t N>
struct FatClass {
  double data;
  char payload[N];
};
```

	10^3 Objects (us)	10^6 Objects (ms)
`LightWeight`	**4**	**5**
`FatClass<10>`	**1.3**	**4**
`FatClass<100>`	2.1	11
`FatClass<1000>`	2.5	11

Contents

Part 4: ViennaMath

Simulation Flow



A Symbolic Math Library in C++

- Symbolic evaluation and manipulation of math expressions
- Unified run time and compile time interface
- Differentiation and integration provided
- LaTeX output

Example Usage

```
variable x(0);
variable y(1);
variable z(2);
expr f = x + y - z;
expr g = f * f;
eval(f, make_vector(1.0, 2.0, 4.0)); // returns -1.0
eval(g, make_vector(1.0, 2.0, 4.0)); // returns  4.0
```

Run Time Evaluation

ViennaMath

A Symbolic Math Library in C++

- Symbolic evaluation and manipulation of math expressions
- Unified run time and compile time interface
- Differentiation and integration provided
- LaTeX output

Example Usage

```
ct_variable<0> x;
ct_variable<1> y;
ct_variable<z> z;
ct_constant<1> c1;
ct_constant<2> c2;
ct_constant<4> c4;
eval(x + y - z, make_vector(c1, c2, c4)); // returns -1
```

- Compile Time Evaluation

Substitution and Differentiation (Run Time)

```
variable x(0);
variable y(1);
variable z(2);
substitute(x, y, x*y + z); // returns y*y+z
diff(x*y + z, x);          // returns y
```

Substitution and Differentiation (Compile Time)

```
ct_variable<0> x;
ct_variable<1> y;
ct_variable<2> z;
substitute(x, y, x*y + z); // returns y*y+z
diff(x*y + z, x);          // returns y
```

Numerical Integration (Run Time): $\int_0^1 x^2 \, \mathrm{d}x$

```
expr f = integral( make_interval(0, 1), x*x, x );

numerical_quadrature integrator(new gauss_quad_1());
integrator(f);                                 // method 1
integrator(make_interval(0, 1), x*x, x);       // method 2
```

Analytical Integration (Compile Time): $\int_0^1 x^2 \, \mathrm{d}x$, $\int_0^1 \int_0^{1-x} xy \, \mathrm{d}x\mathrm{d}y$

```
integrate(make_interval(c0, c1),
          x*y,
          x );    //returns y/2.0
integrate(make_interval(c0, c1),
          integrate( make_interval(c0, c1 - x), x*y, y),
          x);    //returns 1.0/24.0
```

Function Symbols

Represent a function (not evaluable)

Differential Operators

Gradient, Divergence

Symbolic Integration Domain

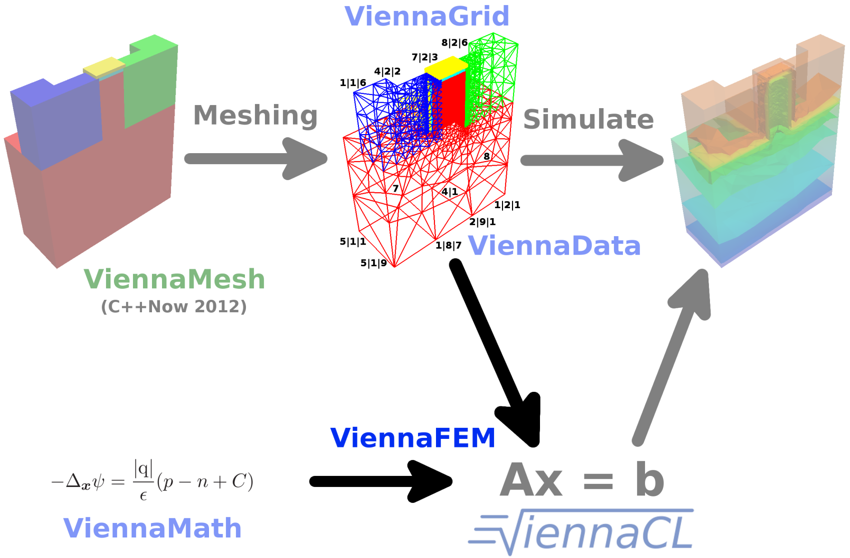
Specify actual integration domain and integration variables *later*

```
function_symbol u;
equation eq( laplace(u), 1.0 );  // Poisson equation

function_symbol v;
expr w = integral(symbolic_interval(),
                  grad(u) * grad(v));
```

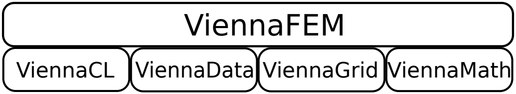
Contents

Part 5: ViennaFEM

Simulation Flow



Simulation Flow



Library-Centric Design

- ViennaCL for linear solver
- ViennaData for data storage
- ViennaGrid for mesh handling
- ViennaMath for symbolic math

Addresses Come&Go in Academia

- Focus on one package
- No in-depth knowledge of all packages required
- Additional emphasis on good interfaces

ViennaFEM

ViennaGrid deals with the Mesh

```
DomainType my_domain;
viennagrid::io::netgen_reader my_reader;
my_reader(my_domain, "mystructure.mesh");
```

Equation specification via ViennaMath: $\Delta u = -1$

```
equation poisson_eq = viennamath::make_equation(laplace(u), -1);
```

Assembly via ViennaFEM

```
viennafem::pde_assembler fem_assembler;
fem_assembler(viennafem::make_linear_pde_system(poisson_eq, u),
              my_domain,
              system_matrix, load_vector);
```

Linear solver provided by ViennaCL

```
VectorType pde_result
  = viennacl::linalg::solve(system_matrix, load_vector, cg_tag());
```

Lame equation for Linear Elasticity

$$\int_\Omega \varepsilon(u) : \sigma(v) \, \mathrm{d}\Omega = \int_\Omega F \cdot v \, \mathrm{d}\Omega \quad \forall v \in \mathcal{V}$$

With $F = (0, 0, 1)^{\mathrm{T}}$:

```
std::vector< Expression > strain = strain_tensor(u);
std::vector< Expression > stress = stress_tensor(v);

Equation weak_form_lame = make_equation(
            integral(symbolic_interval(),
                     tensor_reduce( strain, stress )),
            //=
            integral(symbolic_interval(),
                     constant(1.0) * v[2])
                                         );
```

Summary

Software Packages

- ViennaCL: GPU-accelerated Linear Algebra
- ViennaData: Generic Data Storage
- ViennaFEM: Modular High-Level Finite Element Package
- ViennaGrid: Generic Mesh Datastructure
- ViennaMath: Symbolic Math Kernel

`http://vienna{cl,data,fem,grid,math}@sourceforge.net`

Design Philosophy

- Orthogonal Software Design
- Convenient High-Level User API
- Avoid Dependencies for Fast Migration to other Machines
- Free Open Source