

Eigen

a c++ linear algebra library

Gaël Guennebaud

[<http://eigen.tuxfamily.org>]

Outline

- 9:00 → 9:30 – Discovering Eigen
 - motivations
 - API preview
 - 9:30 → 10:30 – Eigen's internals
 - design choices
 - meta-programming
 - expression templates
- coffee break –

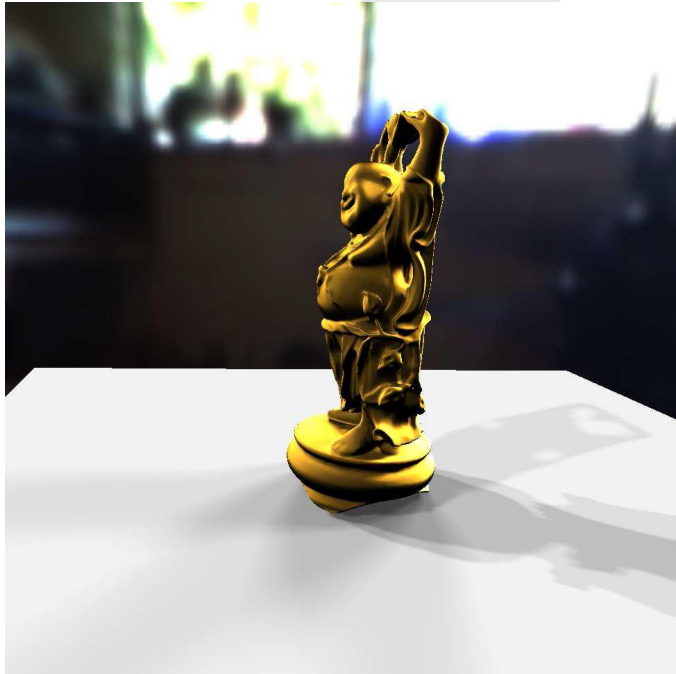
Outline

- 11:00 → 12:30 – Use cases
 - Geometry & 3D algebra
 - RBFs (dense solvers)
 - (bi-)Harmonic interpolation (sparse solvers)
- 12:30 → 13:00
 - Roadmap
 - Open-source community
 - Open discussions

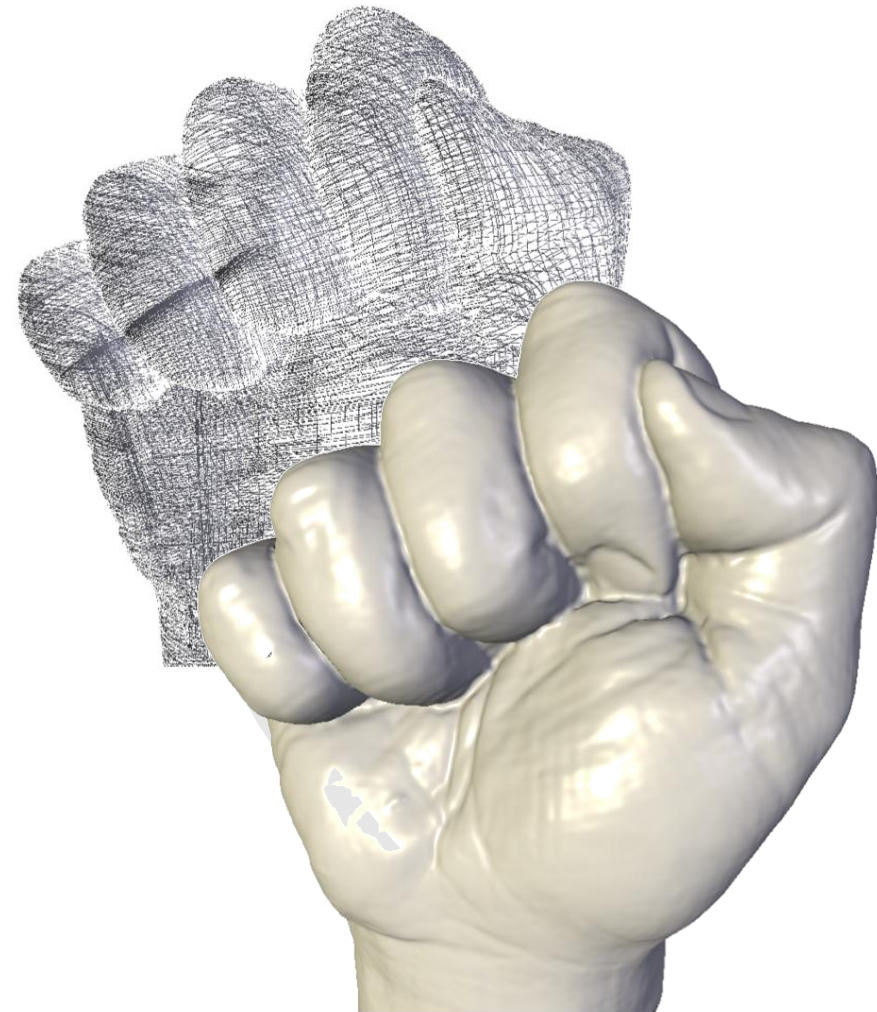


Presentations

Real-time Soft Shadows

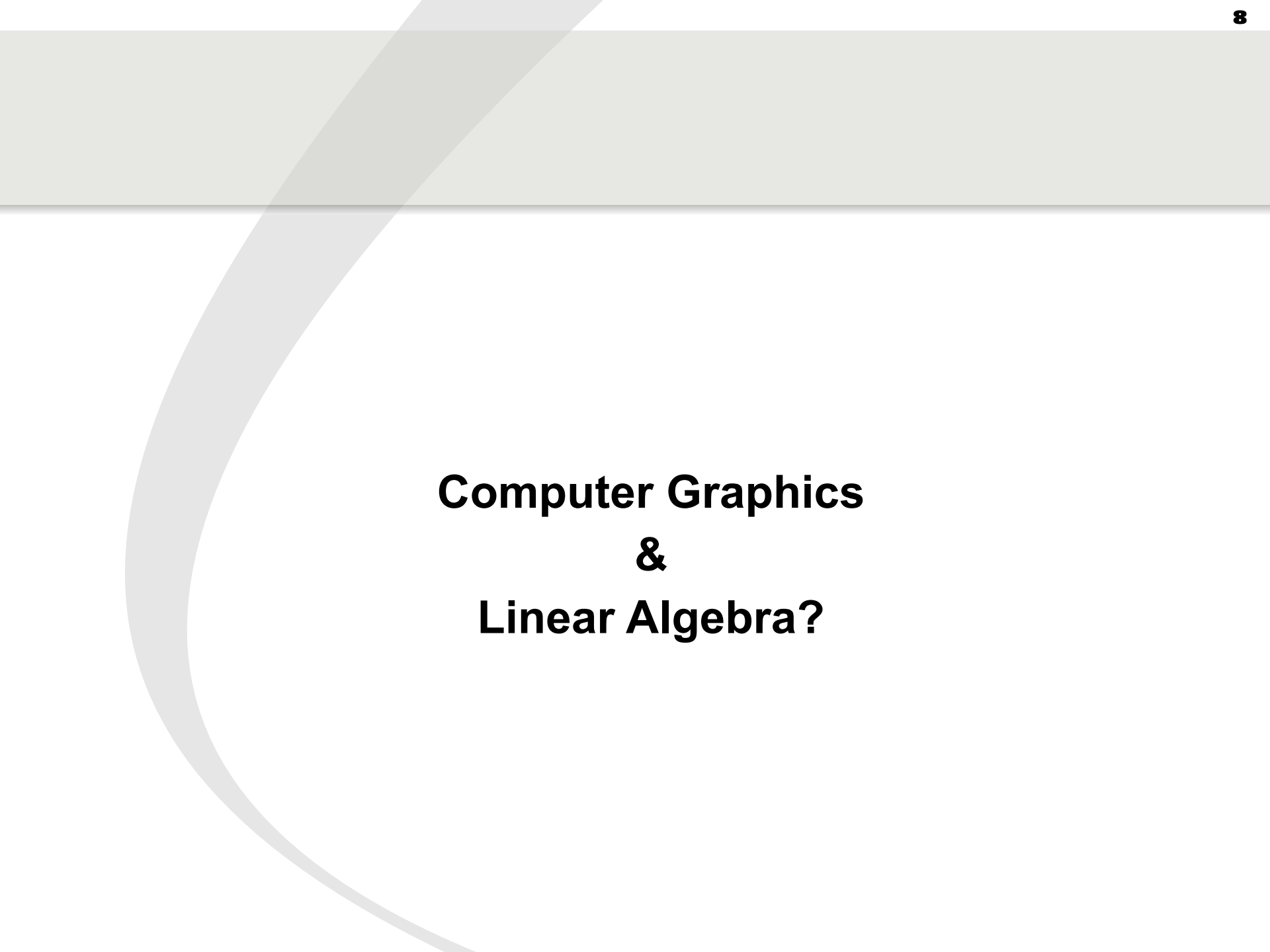


Surface Reconstruction



Skinning & Vector Graphics





Computer Graphics & Linear Algebra?

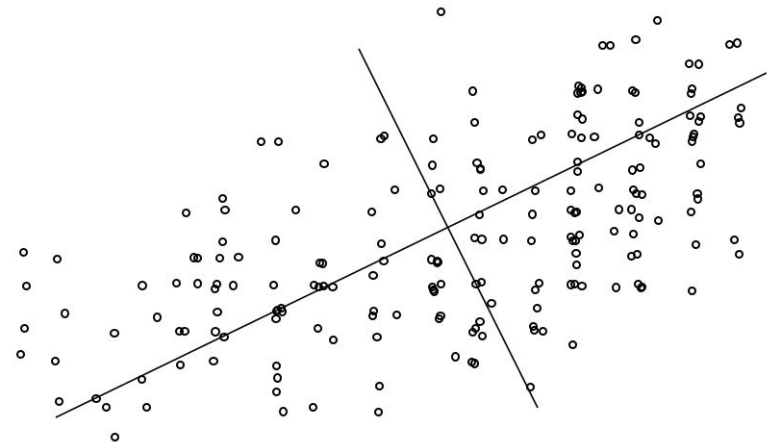
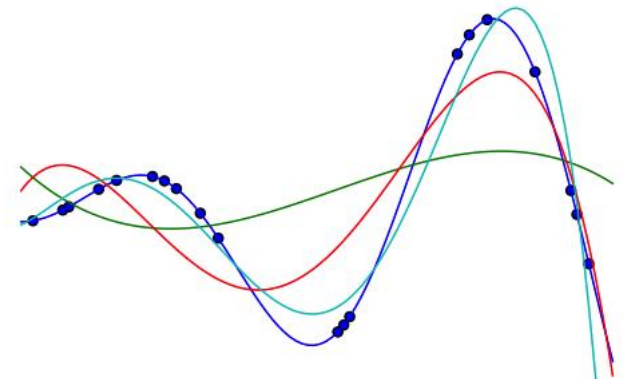
Computer Graphics & Linear Algebra

- points & vectors
 - 2D to 4D vectors (+, -, *, dot, cross, etc.)
- space transformations
 - 2x2 to 4x4 matrices (including non squared matrices as 3x4)
 - inverse (Cramer's rule, Div. & Conq.)
 - polar decomposition
 - Singular Value Decomposition (SVD)
- point sets: normals, oriented bounding boxes
 - Eigen-value decomposition (EVD)

small fixed size linear algebra

Computer Graphics & Linear Algebra

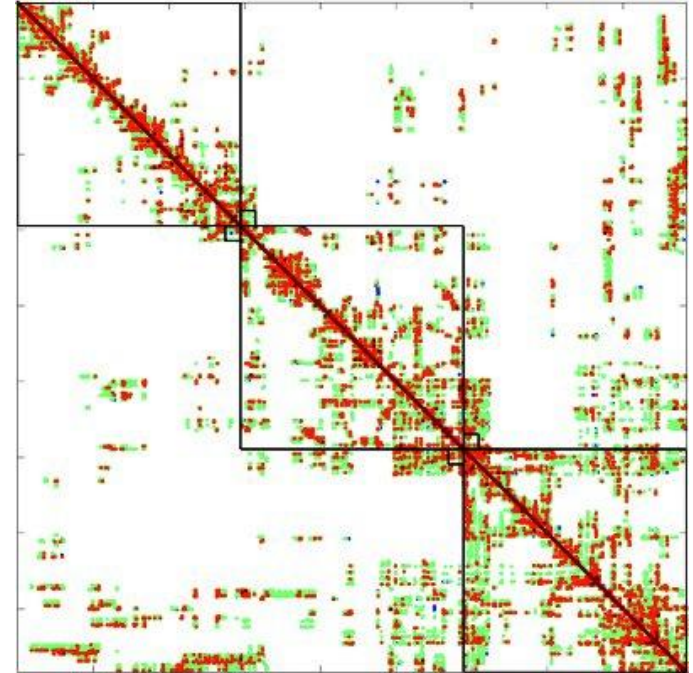
- Linear Least-square
 - Polynomial fit, Curvature estimation, MLS
 - Radial Basis Functions
- Spectral analysis



from small to large dense linear algebra

Computer Graphics & Linear Algebra

- Differential equations, FEM
 - physical simulation
 - mesh processing
 - surface reconstruction
 - interpolation



large sparse linear algebra

Context summary

- Matrix computation are everywhere
 - Various applications:
 - simulators/simulations, video games, audio/image processing, design, robotic, computer vision, augmented reality, etc.
 - Need various tools:
 - numerical data manipulation, space transformations
 - inverse problems, PDE, spectral analysis
 - Need performance:
 - on standard PC, smartpone, embedded systems, etc.
 - real-time performance

Matrix computation?

MatLab

- + friendly API
- + large set of features
- math only
- extremely slow for small objects

→ **Prototyping**

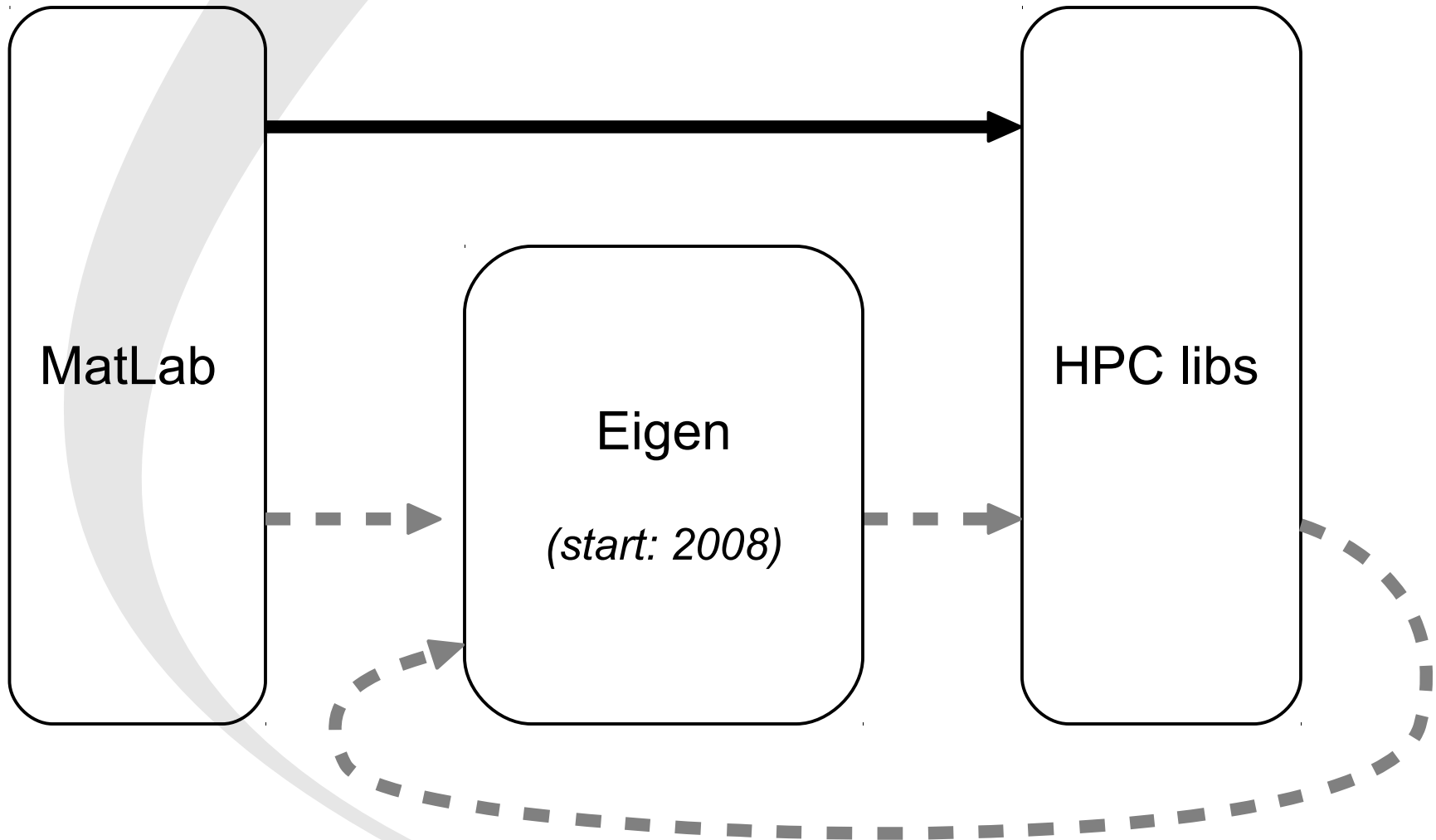


Zoo of libs

- + highly optimized
- 1 feature = 1 lib
- +/- tailored for advanced user / clusters
- slow for small objects

→ **Advanced usages**

Matrix computation?



Facts

- Pure C++ template library
 - header only
 - no binary to compile/install
 - no configuration step
 - no dependency (*optional only*)

```
#include <Eigen/Eigen>
```

```
using namespace Eigen;
```

```
int main() {  
    Matrix4f A = Matrix4f::Random();  
    std::cout << A << std::endl;  
}
```

```
$ g++ -O2 example.cpp -o example
```

Facts

- Pure C++ template library
 - header only
 - no binary to compile/install
 - no configuration step
 - no dependency (*optional only*)
- Packaged by all Linux distributions (*incl. macport*)
- Opensource: MPL2
 - **easy to install & distribute**

Multi-platforms

- Supported compilers:
 - GCC (≥ 4.2), MSVC (≥ 2008), Intel ICC, Clang/LLVM, ~~old apple's compilers~~
- Supported systems:
 - x86/x86_64, ARM, PowerPC
 - Linux, Windows, OSX, IOS
- Supported SIMD vectorization engines:
 - SSE{2,3,4}
 - NEON (*ARM*)
 - AltiVec (*PowerPC*)

Large feature set

- Core
 - Matrix and array manipulation (~MatLab, 1D & 2D)
 - Basic linear algebra (~BLAS)
 - incl. triangular & self-adjoint matrix
- LU, Cholesky, QR, SVD, Eigenvalues
 - Matrix decompositions and linear solvers (~Lapack)
- Geometry (transformations, ...)
- Sparse
 - Manipulation
 - Solvers (LLT, LU, QR & CG, BiCGSTAB, GMRES)
- WIP modules (autodiff, non-linear opt., FFT, etc.)

→ **“unified API” - “all-in-one”**

Optimized for both small and large objects

- Small objects
 - means fixed sizes:
`Matrix<float,4,4>`
 - malloc-free
 - meta unrolling
 - specialized algo
 - Large objects
 - means dynamic sizes
`Matrix<float,Dynamic,1>`
 - cache friendly kernels
 - multi-threading (*OpenMP*)
- Vectorization (SIMD)
 - Unified API → write generic code
 - Mixed fixed/dynamic dimensions

Generic code (1/2)

- Non-generic code:

```
class Sphere {  
    float[3] center;  
    float radius;  
    /* ... */  
};
```

Generic code (1/2)

- Write generic code:

```
template <                                int AmbientDim=Eigen::Dynamic>
class HyperSphere {
    Eigen::Matrix<float ,AmbientDim,1> center;
    float                                radius;
    /* ... */
};

typedef HyperSphere<2>           HyperSphere2;
typedef HyperSphere<3>           HyperSphere3;
typedef HyperSphere<3>           Sphere;
typedef HyperSphere<>            HyperSphereX;
```

- Eigen takes care of the low level optimizations

Generic code (2/2)

- Write fully generic code:

```
template <typename Scalar, int AmbientDim=Eigen::Dynamic>
class HyperSphere {
    Eigen::Matrix<Scalar,AmbientDim,1> center;
    Scalar radius;
    /* ... */
};

typedef HyperSphere<float, 2> HyperSphere2f;
typedef HyperSphere<double,3> HyperSphere3d;
```

Custom scalar types

- Can use custom types everywhere
 - Exact arithmetic (rational numbers)
 - Multi-precision numbers (e.g., via mpfr++)
 - Auto-diff scalar types
 - Interval
 - Symbolic

- Example:

```
typedef Matrix<mpreal,Dynamic,Dynamic> MatrixMP;  
MatrixMP A, B, X;  
// init A and B  
// solve for A.X=B using LU decomposition  
X = A.lu().solve(B);
```

Communication with the world

→ standard matrix representations

- to Eigen

```
float* raw_data = malloc(...);  
Map<MatrixXd> M(raw_data, rows, cols);  
// use M as a MatrixXd  
M = M.inverse();
```

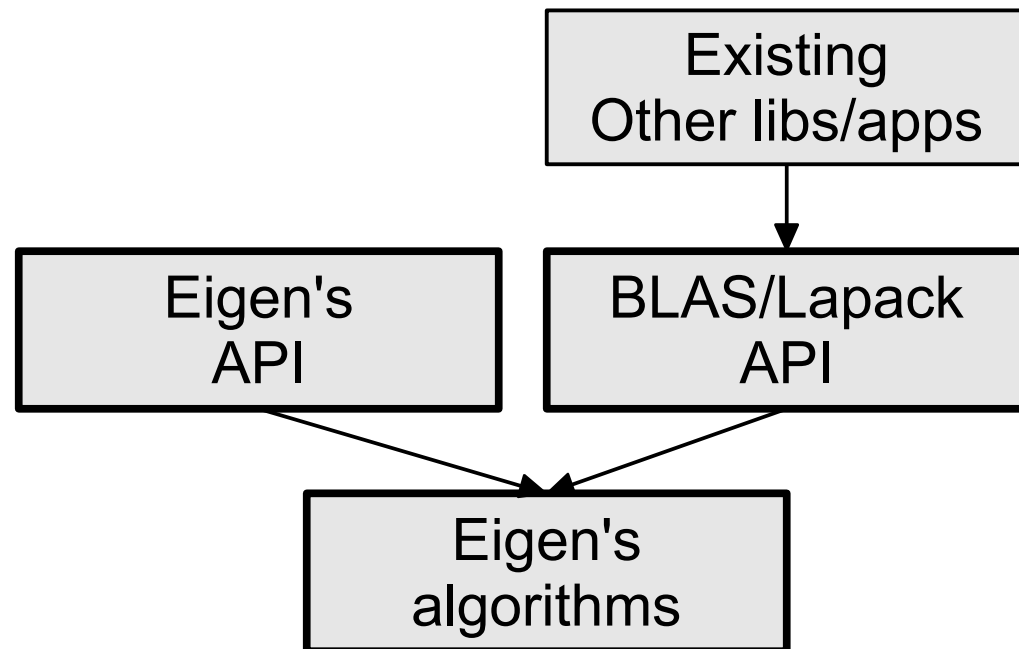
- from Eigen

```
MatrixXd M;  
float* raw_data = M.data();  
int stride = M.outerStride();  
raw_data[i+j*stride]
```

→ same for sparse matrices

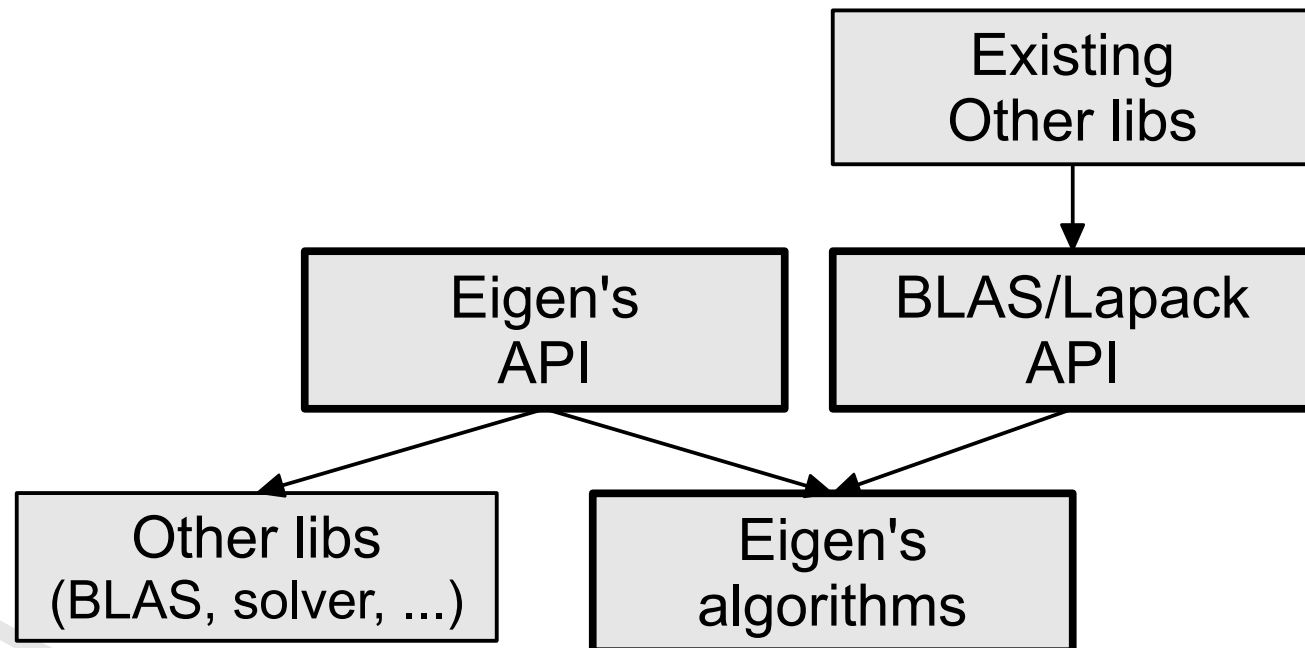
Eigen & BLAS

- Call Eigen's algorithms through a BLAS/Lapack API
 - Alternative to ATLAS, OpenBlas, Intel MKL
 - e.g., sparse solvers, Octave, Plasma, etc.
 - Run the Lapack test suite on Eigen



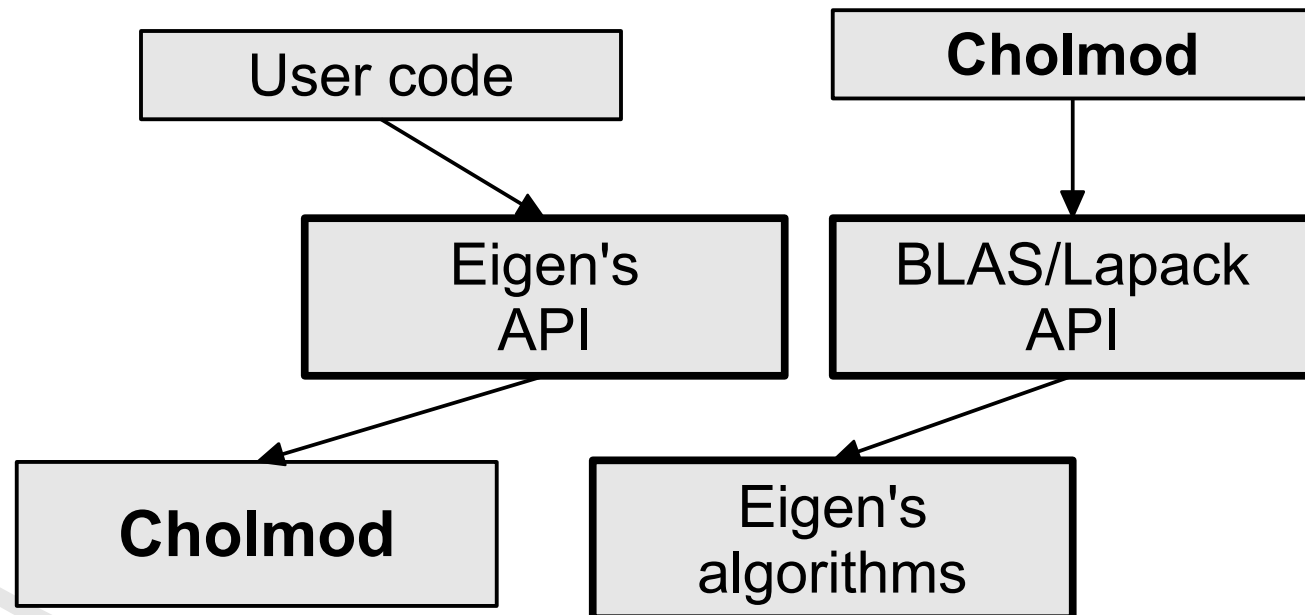
External backends

- External backends
 - Fallback to existing BLAS/Lapack/etc. (done by Intel)
 - Unified interface to many sparse solvers:
 - UmfPack, Cholmod, PaSTiX, Pardiso



External backends

- External backends
 - Fallback to existing BLAS/Lapack/etc. (done by Intel)
 - Unified interface to many sparse solvers:
 - UmfPack, **Cholmod**, PaSTiX, Pardiso



Documentation

- Documentation
- Support
 - Forum, IRC, Mailing-List
 - Bugzilla

API demo

Internals

Technical aspects

~~Matrix factorizations?~~

~~Matrix products?~~

~~Sparse algebra?~~

Expression templates

Meta-programming

Vectorization

Preliminaries 1/3 - C++

- Template programming & Inheritance:

```
template<typename Scalar, int N>
class Vector : public SomeBaseClass {
    Scalar m_data[N];
    /* ... */
};
```

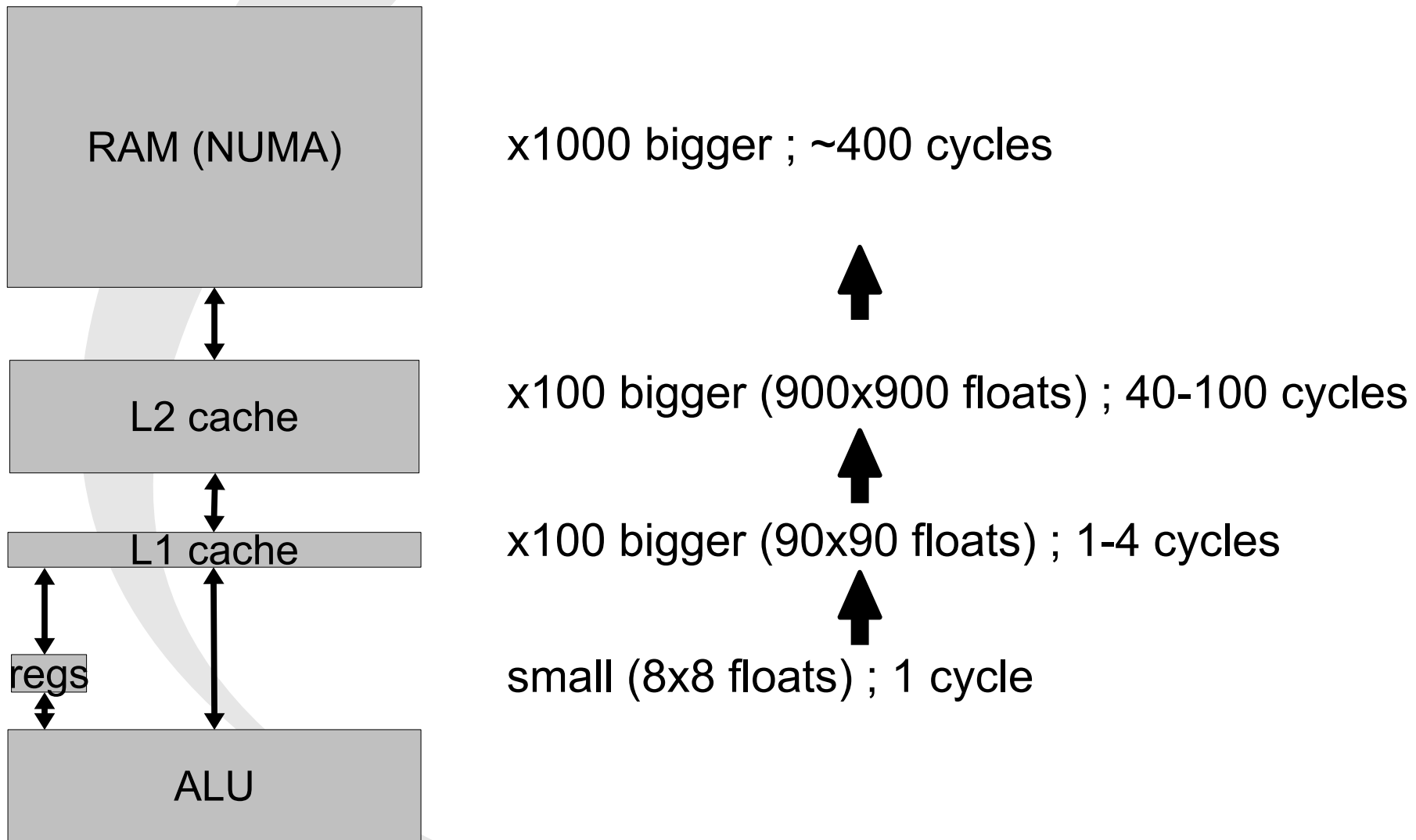
- Partial template specialization:

```
template<typename Real, int N>
class Vector<complex<Real>,N> : public AnotherBaseClass {
    Real m_real[N];
    Real m_imag[N];
    /* ... */
};
```

→ pattern matching

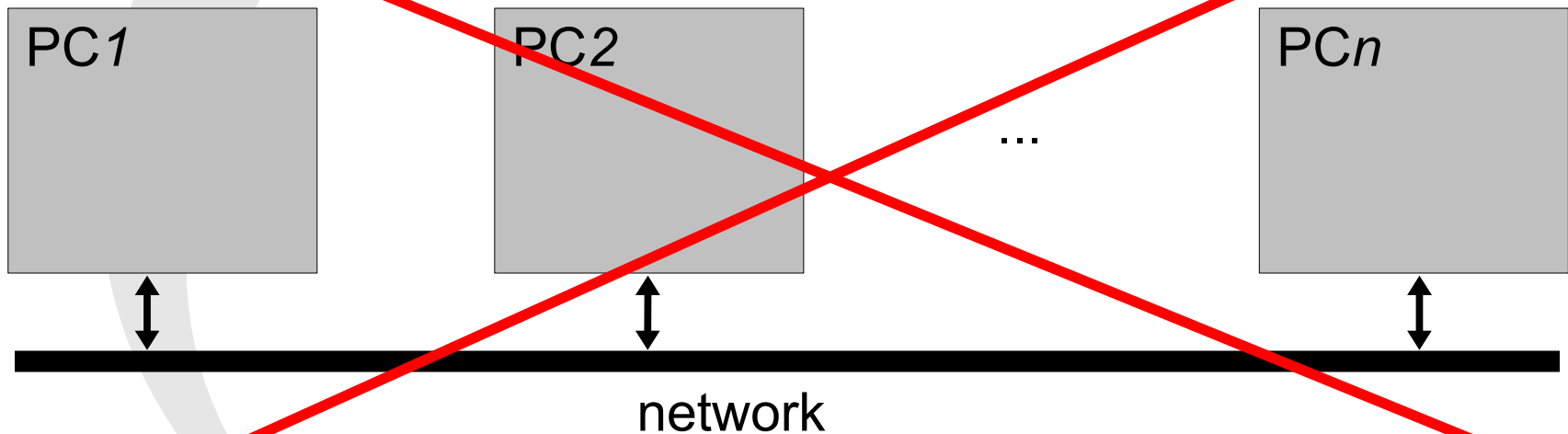
```
Vector<double,3>          v1;
Vector<complex<float>,3> v2;
```


Preliminaries 2/3 - Memory Hierarchy



Preliminaries 3/3 - Parallelism

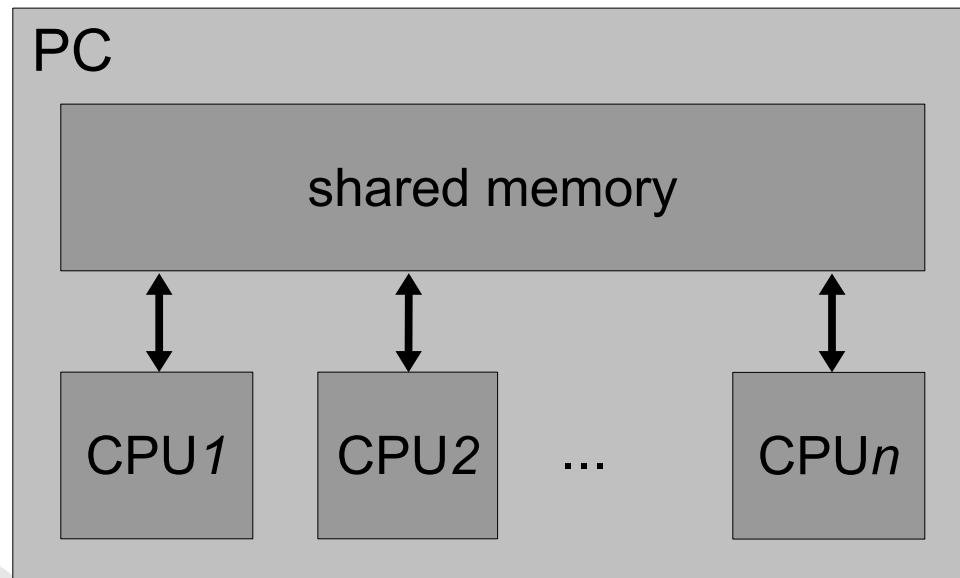
- 4 levels of parallelism:
 - cluster of PCs → MPI



out of the scope of Eigen

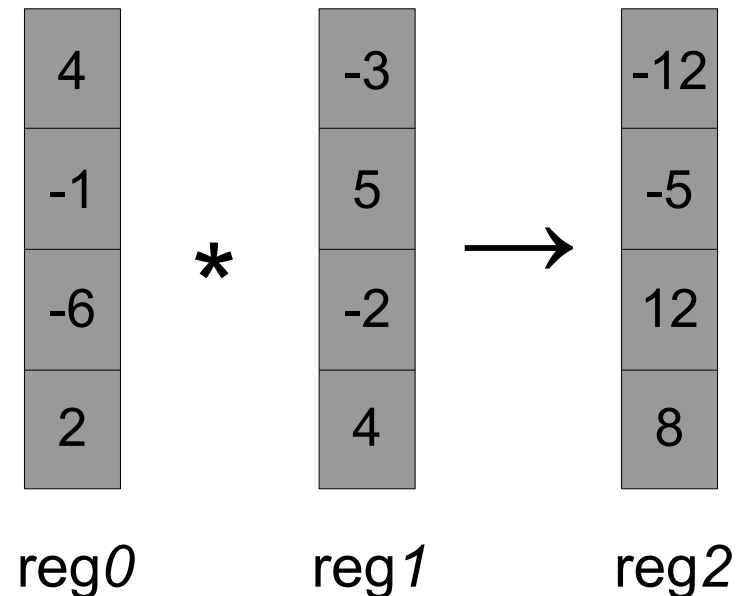
Preliminaries 3/3 - Parallelism

- 4 levels of parallelism:
 - ~~cluster of PCs~~ → MPI
 - multi/many-cores → OpenMP



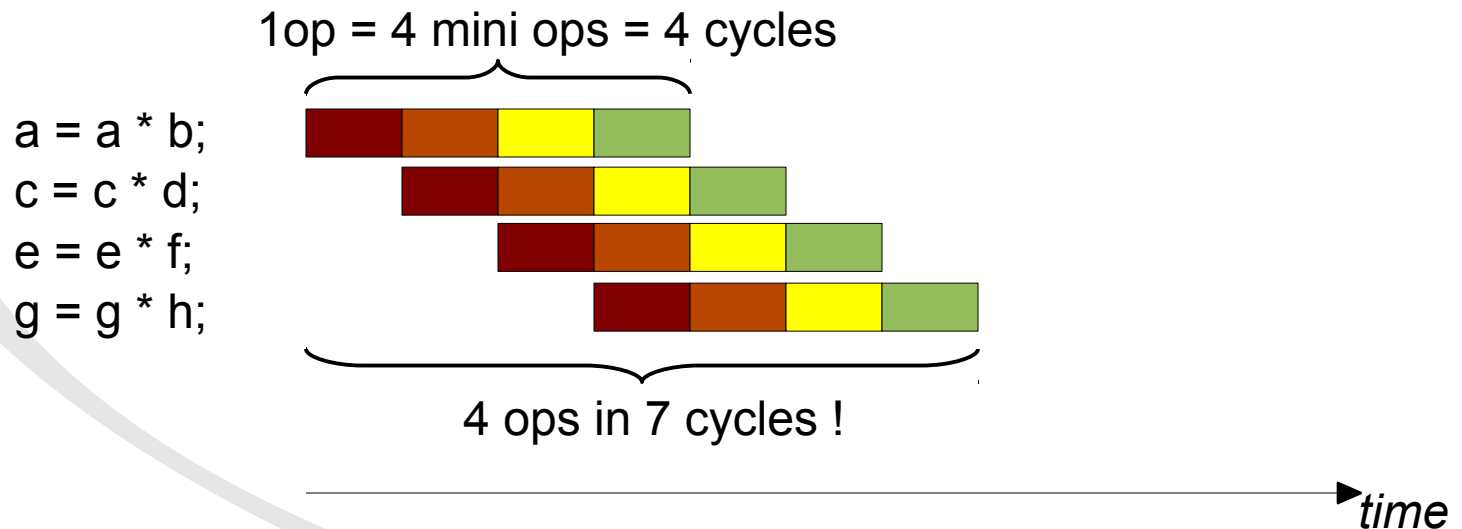
Preliminaries 3/3 - Parallelism

- 4 levels of parallelism:
 - ~~cluster of PCs~~ → MPI
 - multi/many-cores → OpenMP
 - SIMD → intrinsics for vector instructions (SSE, AVX, ...)



Preliminaries 3/3 - Parallelism

- 4 levels of parallelism:
 - ~~cluster of PCs → MPI~~
 - multi/many-cores → OpenMP
 - SIMD → intrinsics for vector instructions (SSE, AVX, ...)
 - pipelining → needs non dependent instructions



Peak performance

- Example
 - Intel Core2 Quad CPU Q9400 @ 2.66GHz (x86_64)
 - pipelining → 1 mul + 1 add / cycle (**ideal case**)
 - SSE → x **4 single precision** ops at once
 - frequency → x **2.66G**
 - **peak performance: 21,790 Mflops** (for 1 core)

that's our goal!



Problem statement

- Example:

```
m3 = m1 + m2 + m3;
```

- Standard C++ way:

```
class Matrix {  
    float m_data[M*N];  
    float& operator()(int i, int j) { return m_data[i+j*M]; }  
};
```

```
Matrix operator+(const Matrix& A, const Matrix& B) {  
    Matrix res;  
  
    for(int j=0; j<N; ++j)  
        for(int i=0; i<M; ++i)  
            res(i,j) = A(i,j) + B(i,j);  
  
    return res;  
}
```

Problem statement

- Example:

```
m3 = m1 + m2 + m3;
```

- Standard C++ way, result:

```
tmp1 = m1 + m2;  
tmp2 = tmp1 + m3;  
m3    = tmp2;
```

- 3 loops :(
- 2 temporaries :(
- **8***M*N memory accesses :(

Expression templates

- Example:

```
m3 = m1 + m2 + m3;
```

- Expression templates:

- “+” returns an expression:

```
Sum<Matrix,Matrix>  
operator+(const Matrix& A, const Matrix& B) {  
    return Sum<Matrix,Matrix>(A,B);  
}
```

```
template<typename type_of_A, typename type_of_B>  
class Sum {  
    const type_of_A &A;  
    const type_of_B &B;  
};
```

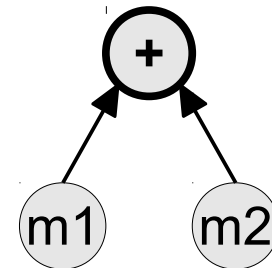
Expression templates

- Example:

$$m3 = m1 + m2 + m3$$

→ “expression tree”

Sum<Matrix,Matrix>



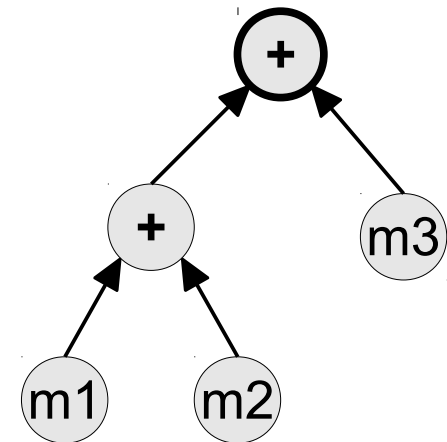
Expression templates

- Example:

$$m3 = m1 + m2 + m3$$

→ “expression tree”

Sum< Sum<Matrix,Matrix>,
Matrix >



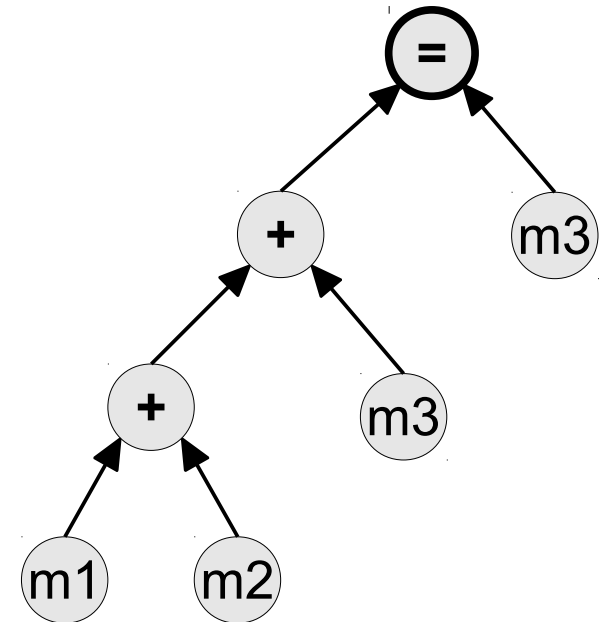
Expression templates

- Example:

$$m3 = m1 + m2 + m3$$

→ “expression tree”

```
Assign<Matrix,  
    Sum< Sum<Matrix,Matrix>,  
        Matrix > >
```



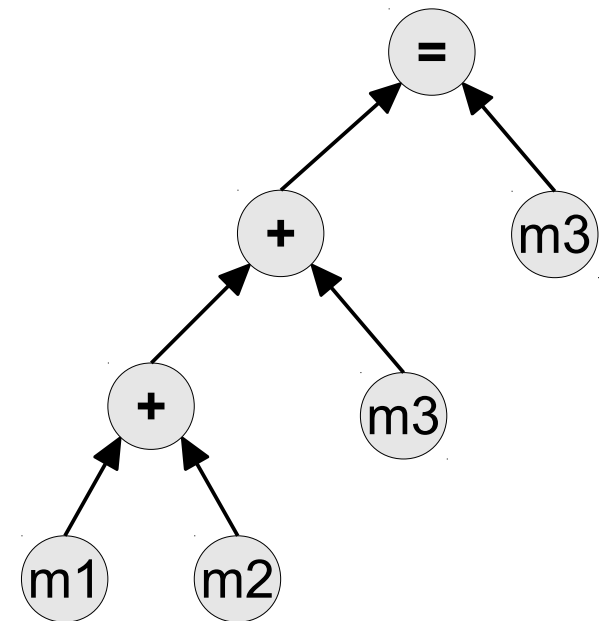
Expression templates

- Example:

`m3 = m1 + m2 + m3;`

→ “expression tree”

```
Assign<Matrix,  
    Sum< Sum<Matrix,Matrix>,  
        Matrix > >
```



- Immediate question:
 - How to evaluate this?

Evaluation of expressions

- Bottom-up approach:

```
template<type_of_A, type_of_B>
class Sum {
    const type_of_A &A;
    const type_of_B &B;

    Scalar coeff(i, j) {
        return A.coeff(i,j) + B.coeff(i,j);
    }
};
```

→ simple, can specialize the implementation based on the operand types...

... but not on the **whole expression** :(

Evaluation of expressions

- Solution: **top-down** creation of an evaluator
 - evaluator:

```
template<ExprType> class Evaluator;
```

- partial specialization for each operation, e.g.:

```
template<type_of_A, type_of_B>
class Evaluator< Sum<type_of_A, type_of_B> > {
    Evaluator<type_of_A> evalA(A);
    Evaluator<type_of_B> evalB(B);
    Scalar coeff(i, j) {
        return evalA.coeff(i, j) + evalB.coeff(i, j);
    }
};
```

Evaluation of expressions

- Matrix evaluator:

```
class Evaluator<Matrix> {  
    const Matrix &mat;  
    Scalar coeff(i) { return mat.data[i]; }  
};
```


Evaluation of expressions

- Solution: **top-down** creation of an evaluator
 - assignment evaluator (dest ← source):

```
template<Dest, Source>
class Evaluator< Assign<Dest,Source> > {
    Evaluator<Dest>    evalDst(dest);
    Evaluator<Source> evalSrc(source);
    void run() {
        for(int i=0; i<evalDst.size(); ++i)
            evalDst.coeff(i) = evalSrc.coeff(i);
    }
};
```

- Example: $m_3 = m_1 + m_2 + m_3$;
 - compiles to:

```
for(i=0; i<m3.size(); ++i)
    m3[i] = "Evaluator(m1+m2+m3)".coeff(i);
```

Template Instantiations

```
for(i=0; i<m3.size(); ++i)
    m3[i] = "Evaluator(m1+m2+m3)".coeff(i);
```

```
class Evaluator< Sum< Sum<Matrix,Matrix>, Matrix > > {
    Evaluator<Sum<Matrix,Matrix> > evalA("m1+m2");
    Evaluator<Matrix> evalB("m3");
    Scalar coeff(i) {
        return evalA.coeff(i) + evalB.coeff(i);
    }
};
```

m3[i]

```
class Evaluator< Sum<Matrix,Matrix> > {
    Evaluator<Matrix> evalA("m1");
    Evaluator<Matrix> evalB("m2");
    Scalar coeff(i) {
        return evalA.coeff(i) + evalB.coeff(i);
    }
};
```

m1[i] m2[i]

generated
by the
compiler!

Template Instantiations

```
m3 = m1 + m2 + m3;
```



- After inlining:

```
for(i=0; i<m3.size(); ++i)  
    m3[i] = m1[i] + m2[i] + m3[i];
```

- 1 loop
- no temporaries
- /2 memory accesses

Expression templates

- Generalize to any coefficient-wise operations

- example:

```
m3.block(1,2,rows,cols) = 2*m1 - m2.transpose();
```

- expression type:

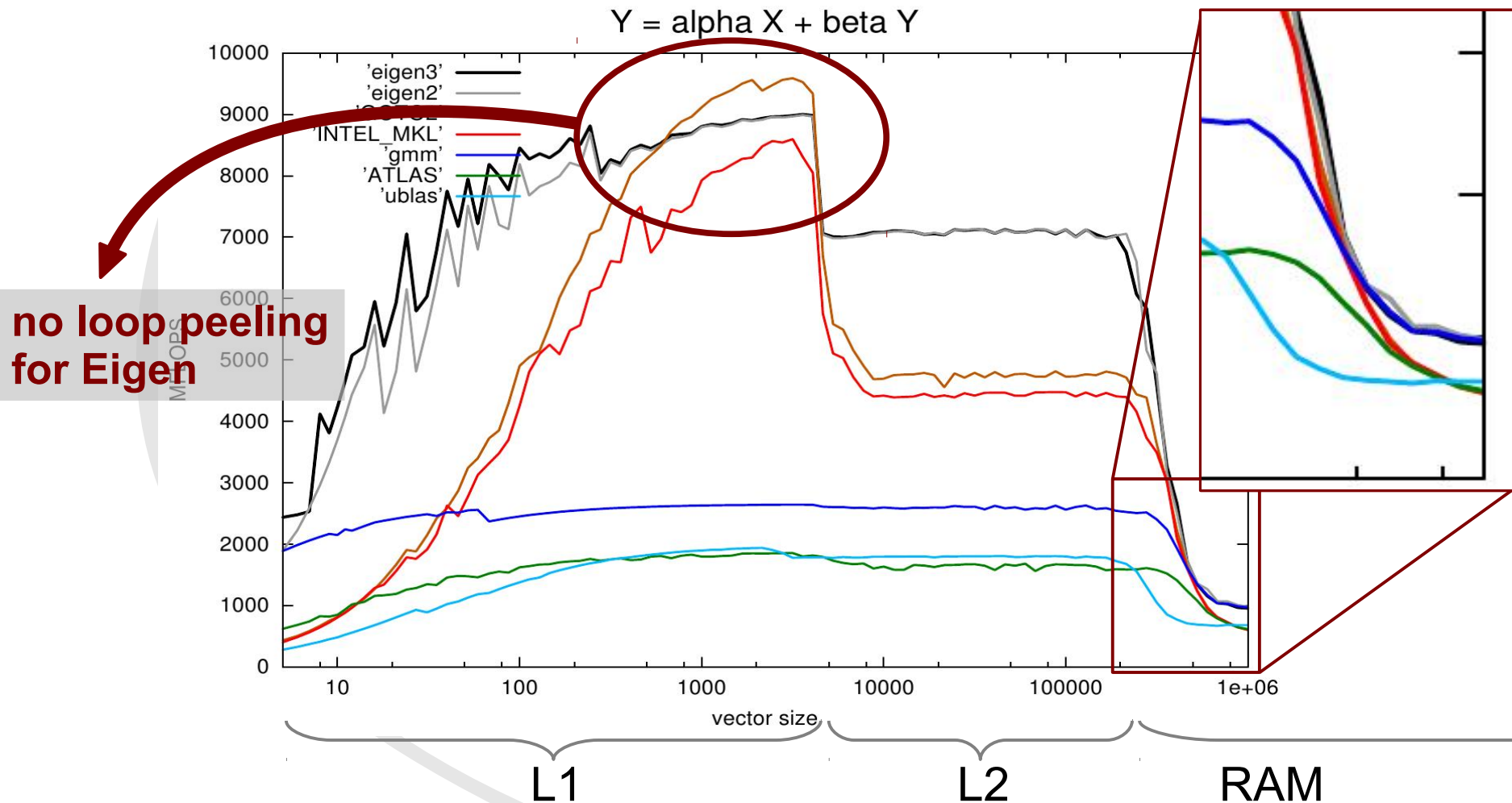
```
Assign<Block<Matrix>,  
      Difference< ScalarMultiple<Matrix>,  
                  Transpose<Matrix> > >
```

- compiles to:

```
for(int j=0; j<cols; ++j)  
  for(int i=0; i<rows; ++i)  
    m3(i+1,j+2) = 2*m1(i,j) - m2(j,i);
```

Expr. templates: Fused operations

- reduce temporaries, memory accesses, cache misses



Expr. templates: Better API

- Better API
 - more examples:

```
x.col(4) = A.lu().solve(B.col(5));
```

```
x = b * A.triangularView<Lower>().inverse();
```

Combinatorial complexity

- Explosion of types and possible combinations

```
Sum<.,.> operator+(const Matrix& A, const Matrix& B);  
Sum<.,.> operator+(const Sum<.,.>& A, const Matrix& B);  
Sum<.,.> operator+(const Sum<.,.>& A, const Sum<.,.>& B);  
Sum<.,.> operator+(const Sum<.,.>& A, const Transpose<.>& B);  
Sum<.,.> operator+(const Transpose<.>& A, const Matrix& B);  
...
```

→ need a common base class
+ polymorphism

Combinatorial complexity

- Common base class:

```
class Matrix : MatrixBase {...};
class Sum<A,B> : MatrixBase {...};
```

```
class Evaluator<Sum<A,B> >
    : EvaluatorBase {
    /* ... */
    virtual Scalar coeff(i,j);
};
```

```
class MatrixBase {
```

```
Sum<MatrixBase,MatrixBase>
operator+(const MatrixBase& other) {
    return Sum<MatrixBase,MatrixBase>(*this, other);
}
```

cannot work this way!

```
};
```

→ need **compile-time** polymorphism

→ CRTP (Curiously Recurring Template Pattern)

CRTP

- base class:

```
class Matrix    : MatrixBase {...};  
class Sum<A,B> : MatrixBase {...};
```

```
class MatrixBase {
```

```
    Sum<MatrixBase,MatrixBase>  
    operator+(const MatrixBase& other) {  
        return Sum<MatrixBase,MatrixBase>(*this, other);  
    }
```

```
};
```

CRTP

- base class + static polymorphism:

```
class Matrix    : MatrixBase< Matrix > {...};
class Sum<A,B>  : MatrixBase< Sum<A,A> > {...};

template<typename Derived>
class MatrixBase {

    template<typename OtherDerived>
    Sum<Derived,OtherDerived>
    operator+(const MatrixBase<OtherDerived>& other) {
        return Sum<Derived,OtherDerived>(derived(), other.derived());
    }

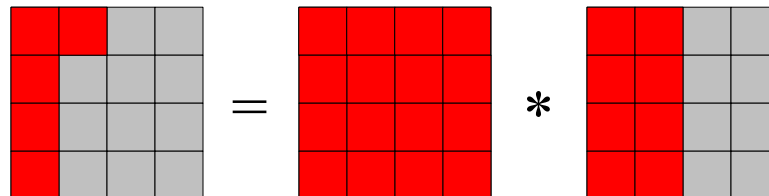
    Derived& derived() { return static_cast<Derived&>(*this); }
};
```

Product-like operations?

- Expression templates
 - very good for any coefficient-wise operations
 - what about matrix products?

```
class Evaluator< Product<type_of_A,type_of_B> > {
    Scalar coeff(i,j) {
        return (A.row(i).cwiseProduct(B.col(i).transpose())).sum();
    }
};
```

- what's wrong?



→ OK for very very small matrices only

How to make products efficient?

→ cache-aware product algorithm

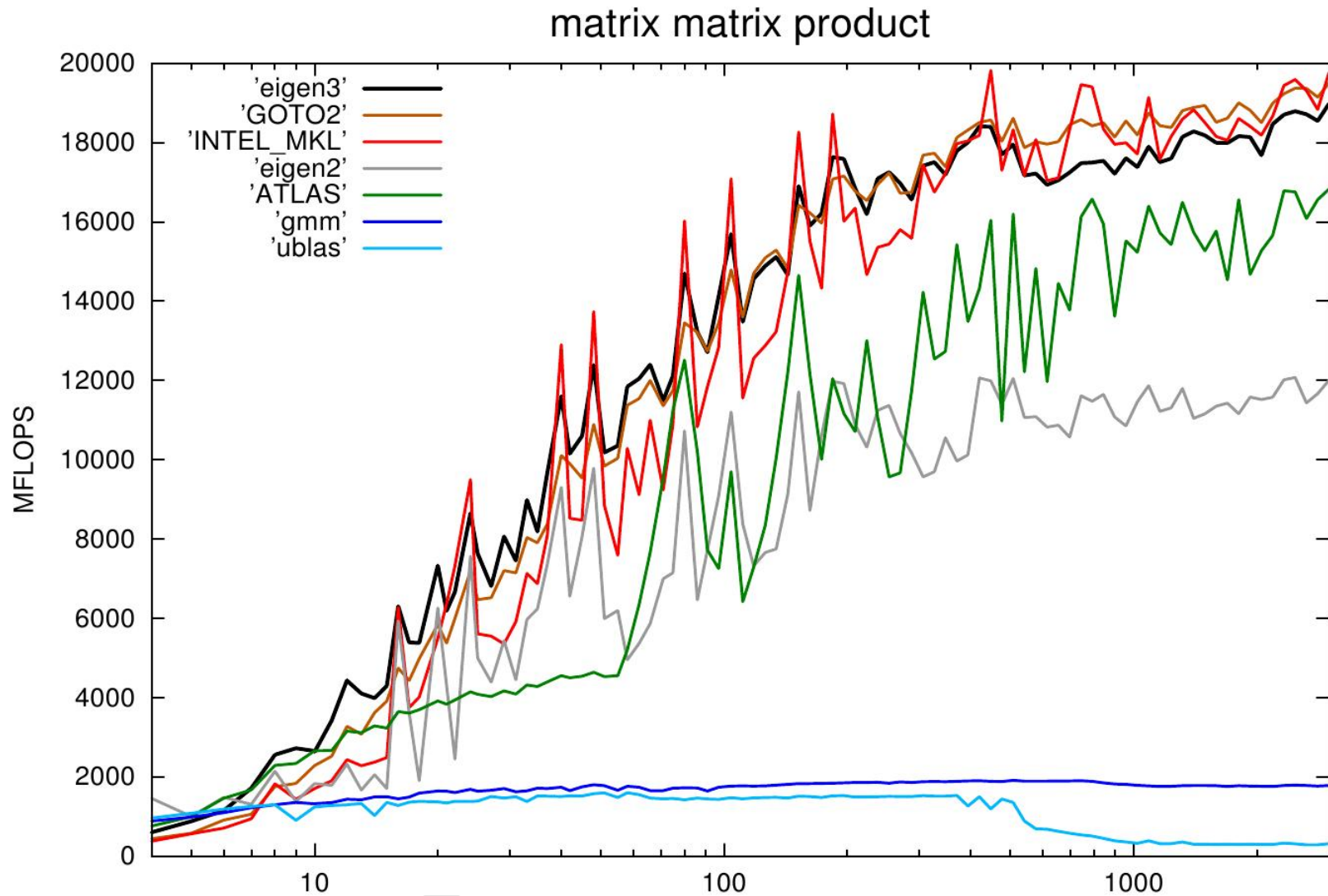
- optimize L2, L1 and register reuses
 - needs access the data of the result

```
D = C + A * B ;
```

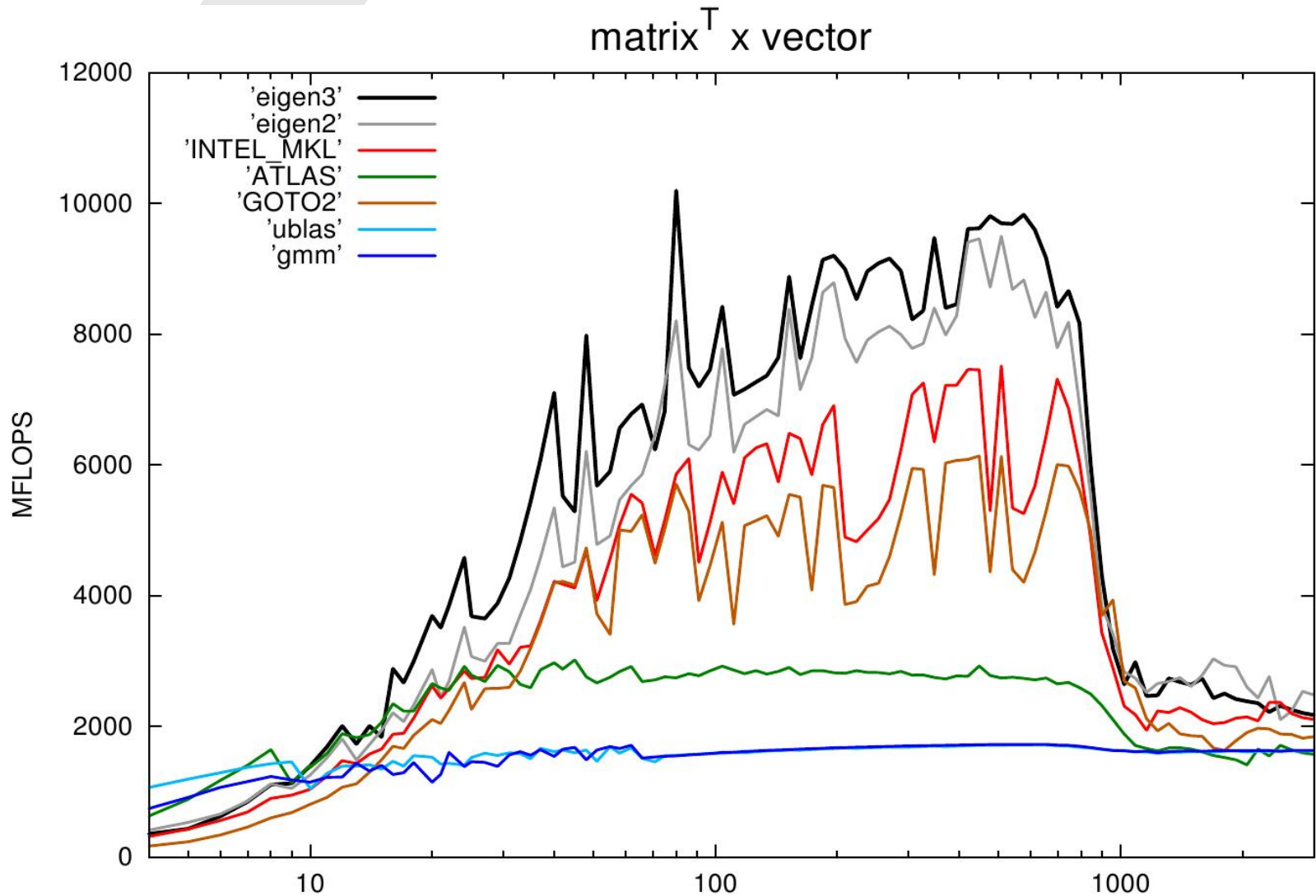
needs to be evaluated
into a temporary:

```
Matrix tmp;  
gemm(A, B, tmp);  
D = C + tmp;
```

Performance



Performance



How to make products efficient?

- Combinatorial complexity

```
A * B;          2*A * B.adjoint();
```

```
A.col(j).transpose() * B;
```

```
A.transpose() * B;
```

```
-A.block(i,j,r,c) * (2*B).transpose();
```

- one product version is very complex (lot of instructions)
→ handle only one generic version:

```
gemm<op1,op2>(A,B,s,C);  
→ C += s * op1(A) * op2(B);
```

- op_ = nop, conjugate, transpose, adjoint
- A, B, C → reference to block of memory with strides

Top-down expression analysis

- Products

- avoid temporary when possible

- e.g.: `m4 -= (2 * m2).adjoint() * m3;`
 → `gemm<Adj,Nop>(m2, m3, -2, m4);`

```

Evaluator<Assign<type_of_C,Product<type_of_A,type_of_B> > > {
  Evaluator(C,P) {

    EvaluatorForProduct<type_of_A> evalA(P.A());
    EvaluatorForProduct<type_of_B> evalB(P.B());

    gemm<evalA.op,evalB.op>(  evalA.data, evalB.data,
                             evalA.scale*evalB.scale,
                             C);
  }
};

```

Top-down expression analysis (cont.)

- More complex example:

```
m4 -= m1 + m2 * m3;
```

– so far:

```
tmp = m2 * m3;  
m4 -= m1 + tmp;
```

– better:

```
m4 -= m1;  
m4 -= m2 * m3;
```

```
// catch R = A + B * C  
Evaluator<Assign<R, Sum<A, Product<B, C> > > { ... };
```

Tree optimizer

- Even more complex example:

```
res -= m1 + m2 + m3*m4 + 2*m5 + m6*m7;
```

- Tree optimizer

```
→ res -= ((m1 + m2 + 2*m5) + m3*m4) + m6*m7;
```

- yields:


```
res -= m1 + m2 + 2*m5;
res -= m3*m4;
res -= m6*m7;
```

- Need only two rules:

```
// catch A * B + Y and builds Y' + A' * B'
TreeOpt<Sum<Product<A,B>,Y> > { ... };
```

```
// catch X + A * B + Y and builds (X' + Y') + (A' * B')
TreeOpt<Sum<Sum<X,Product<A,B> >,Y> > { ... };
```

→ demo

Tree optimizer

- Last example:

```
res += m1 * m2 * v;
```

- Tree optimizer

```
→ res += m1 * (m2 * v);
```

- Rule:

```
TreeOpt<Product<Product<A,B>,C> > { ... };
```

Vectorization

- How to exploit SIMD instructions?
 - Evaluator:

```
class Evaluator< Sum<type_of_A,type_of_B> > {  
    Scalar coeff(i,j) {  
        return evalA.coeff(i,j) + evalB.coeff(i,j);  
    }  
  
    Packet packet(i,j) {  
        return padd(evalA.packet(i,j), evalB.packet(i,j));  
    }  
};
```

 unified wrapper to intrinsics
(SSE, NEON, AVX)

Vectorization

- How to exploit SIMD instructions?
 - Assignment:

```
class Evaluator< Assign<Dest,Source> > {
    void run() {
        for(int i=0; i<evalDst.size(); ++i)
            evalDst.coeff(i,j) = evalSrc.coeff(i,j);
    }

    void run_simd() {
        for(int i=0; i<evalDst.size(); i+=PacketSize)
            evalDst.writePacket(i,j, evalSrc.packet(i,j));
    }
};
```

Vectorization: result

```
#include<Eigen/Core>
using namespace Eigen;

void foo(Matrix2f& u,
         float a, const Matrix2f& v,
         float b, const Matrix2f& w)
{
    u = a*v + b*w - u;
}
```

```
movl 8(%ebp), %edx
movss 20(%ebp), %xmm0
movl 24(%ebp), %eax
movaps %xmm0, %xmm2
shufps $0, %xmm2, %xmm2
movss 12(%ebp), %xmm0
movaps %xmm2, %xmm1
mulps (%eax), %xmm1
shufps $0, %xmm0, %xmm0
movl 16(%ebp), %eax
mulps (%eax), %xmm0
addps %xmm1, %xmm0
subps (%edx), %xmm0
movaps %xmm0, (%edx)
```

Unrolling

- Small sizes
 - cost dominated by loop logic
 - remove the loop... yourself!

(don't overestimate compiler's abilities)

```
for(int i=n-1; i>=0; --i)
  foo(i,args);
```

```
void foo_impl(int i,args) {
  foo(i,args);
  if(i>0)
    foo_impl(i-1,args);
}
```

```
foo_impl(n-1,args);
```

functional approach

```
template<int I> struct foo_impl {
  static void run(args) {
    foo(I,args);
    foo_impl<I-1>::run(args);
  }
};
```

```
template<> struct foo_impl<-1> {
  static void run(args) {}
};
```

```
foo_impl<N-1>::run(args);
```


Controls

- Still many questions:
 - which loops have to be unrolled?
 - which sub-expressions have to be evaluated?
 - is vectorization worth it?
 - Depend on many parameters:
 - scalar type
 - expression complexity
 - kind of operations
 - architecture
- **need an evaluation cost model**

Cost model

- Cost Model
 - Track an approximation of the cost to evaluate one coefficient
 - each scalar type defines: *ReadCost*, *AddCost*, *MulCost*
 - combined for each expression by the evaluator, e.g.:

```
class Evaluator< Sum<A,B> > {  
    enum {  
        Cost = NumTraits<Scalar>::AddCost  
            + Evaluator<A>::Cost  
            + Evaluator<B>::Cost;  
    };  
    ...  
};
```

Cost model

- Examples:

- loop unrolling (partial)

```
// somewhere in Evaluator<Assign>:
assign_impl<      (Evaluator<Src>::Cost*N > threshold)
                ? NoUnrolling : Unrolling >::run(dst,src);
```

- evaluation of sub expressions

- $(a+b) * c \rightarrow (a+b)$ is evaluated into a temporary
- enable vectorization of sub expressions

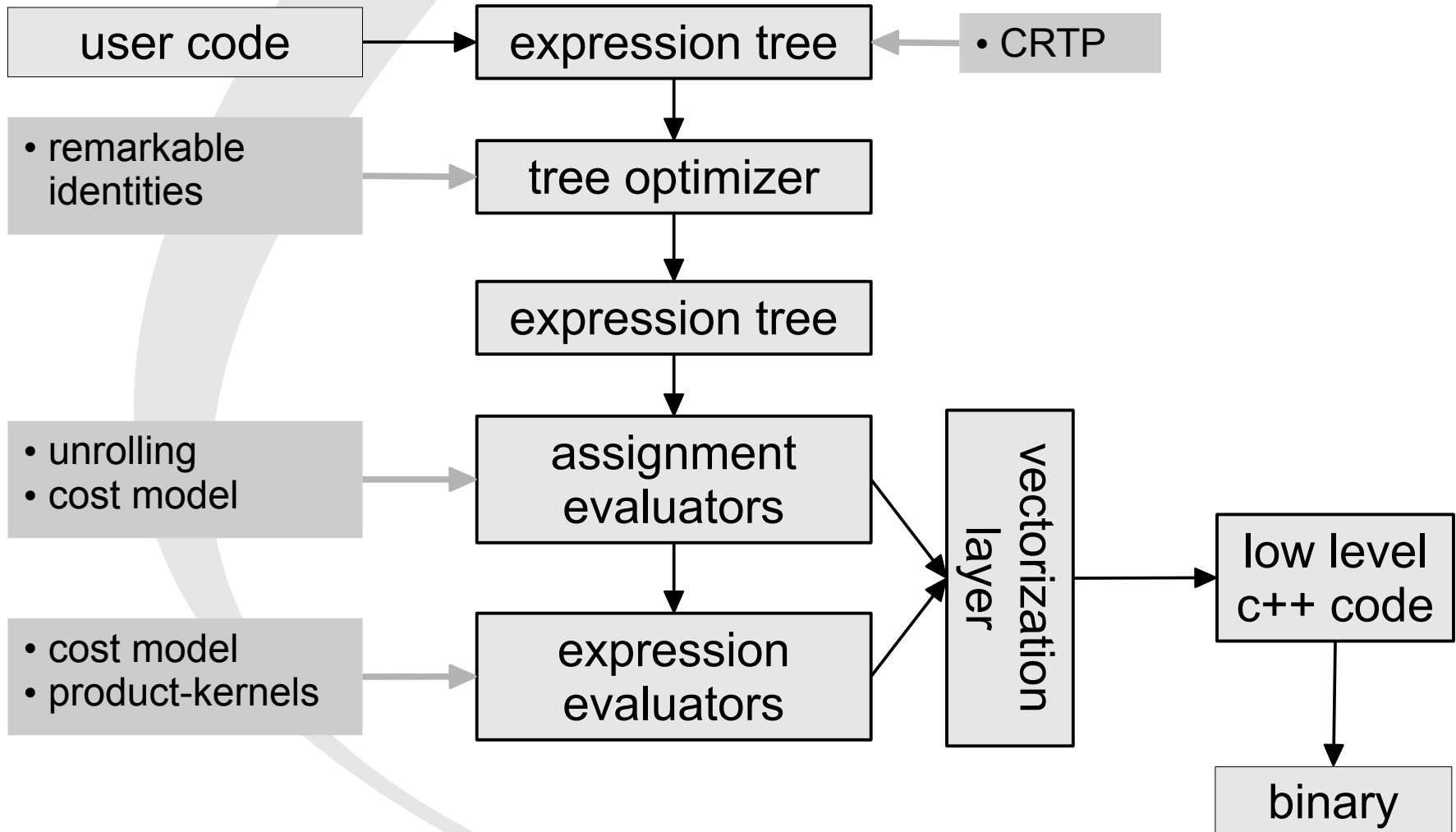
$(2*A+B).log() + C.abs()/4$

1 loop, but
no vectorization

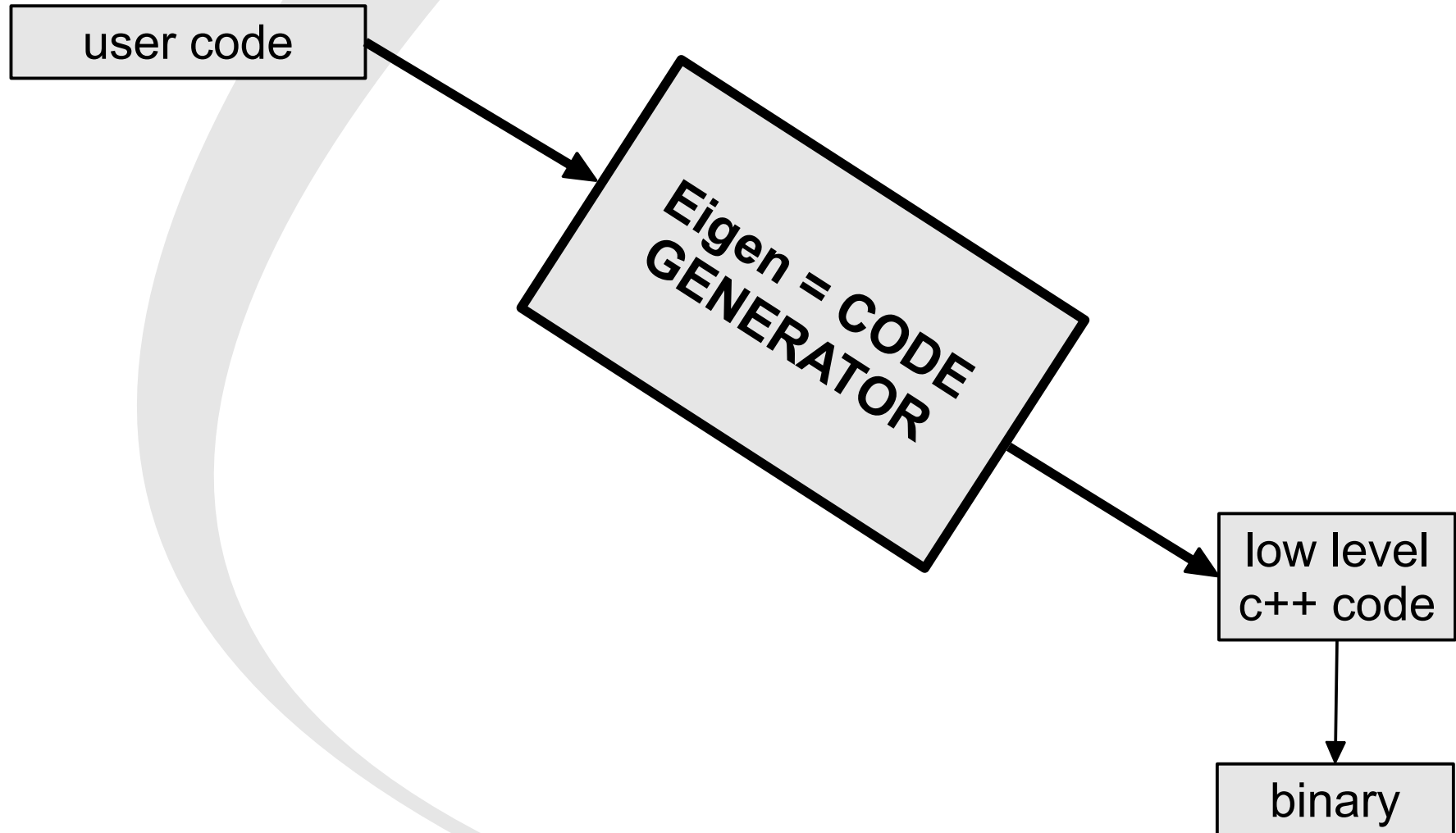
?

```
t1 = 2*A+B;      // vec on
t1 = t1.log();  // no vec
t1 + C.abs()/4; // vec on
```

Putting everything together



Putting everything together



Reductions

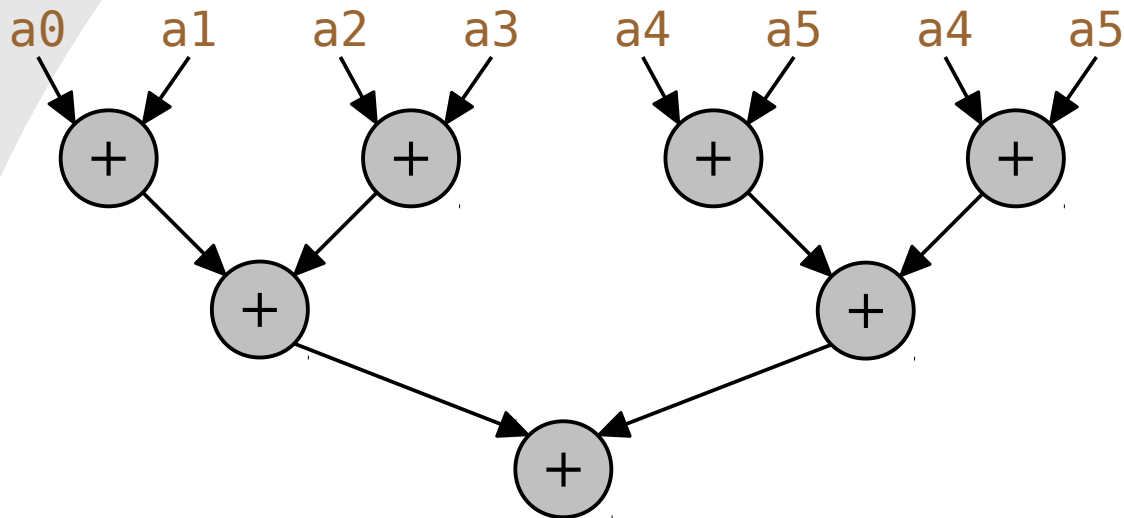
- Example: `dot = (v1.array() * v2.array()).sum();`
- Naive way:

```
class MatrixBase {  
    ...  
    Scalar sum() const {  
        Evaluator<Derived> eval(this->derived());  
        Scalar acc = 0;  
        for(int i=0; i<size(); ++i)  
            acc = acc + eval.coeff(i);  
        return acc;  
    }  
    ...  
};
```

cannot exploit
instruction level parallelism (:

Reductions

→ divide & conquer:



- Exercise: write a generic D&C meta-unroller!
 - solution in *Eigen/src/Core/Redux.h*

→ demo

Eigen tutorial

– coffee break –

Eigen tutorial

Use cases

Space Transformation & OpenGL

Space Transformations

- Needs
 - Translations, Scaling, Rotations,
 - Isometry, Affine/Projective transformation
 - ... in arbitrary dimensions
- Many different approaches

Transformations

Own cooking

- Low level math:

```
Matrix3f rot = ???;
v' = rot * (2*v-p) + p;
```

- Directly manipulate a D+1 matrix:

- form the matrix:

```
Matrix<float,3,4> T;
T.leftCols<3>() = 2*rot;
T.col(3) = p - rot * p;
```

$$v' = 2*rot*v + (p-rot*p)$$

- apply the transformation:

<pre>Vector4f v1; v1 << v, 1; v' = T * v1;</pre>	<pre>v' = T.leftCols<3>() * v + T.col(3);</pre> <hr style="border: 0.5px solid black;"/> <pre>v' = T * v.homogeneous();</pre>
--	---

Transformations

The procedural approach

- OpenGL1 inspired

```
Transform<float,3,Affine> T; // wrap a Matrix4f
```

```
T.setIdentity();  
T.translate(p);  
T.rotate(angle,axis);  
T.translate(-p);  
T.scale(2);
```

```
T.setIdentity();  
T.preScale(2);  
T.preTranslate(-p);  
T.preRotate(angle,axis);  
T.preTranslate(p);
```

```
v' = T * v;
```

- cons:

- hide the concatenation logic
 - how to concatenate on the left?
 - error prone, does help to understand transformations
 - far away to what people write on paper

Transformations

The “natural” approach

- Example:

```
Transform<float,3,Affine> T;
```

```
T = Translation3f(p) * rot * Translation3f(-p) * Scaling(2);
```

```
v' = T * v;
```

- Unified concatenate/apply → “*”

```
Translation3f(p) * v; // compiles to “p+v”
```

```
Isometry3f T1;
```

```
T1 * Scaling(-1,2,2); // returns an Affine3f
```

Transformations

The “natural” approach

- 3D rotations:
 - AngleAxis, Unit quaternion, Unitary matrix

```
... * AngleAxis3f(M_PI/2, Vector3f::UnitZ()) * ...
```

- Unified conversions → “=”

```
Quaternionf q; q = AngleAxis3f(...);
```

- Unified inversion → `.inverse()`

```
Translation3f(p).inverse() // → Translation3f(-p)
```

```
Isometry3f T1;
```

```
T1.inverse() // → T1.linear().transpose()
              * Translation3f(-T1.translation())
```

Transformations

Generic programming

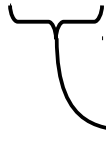
- Write generic optimized functions:

```
template<typename TransformationType>
void foo(const TransformationType& T) {
    T * v
    T.inverse() * v
    Projective3f(...) * T * Translation3f(...) * T.inverse()
    ...
}
```


Eigen & OpenGL

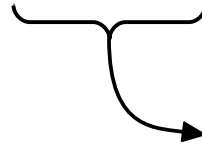
- Expression templates & OpenGL

```
Vector3f p;  
glUniform3fv(p.data());
```



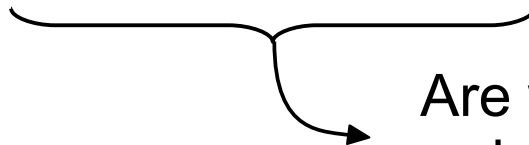
we already know that from “p”!

```
Vector3f p1, p2;  
glUniform3fv((0.5*(p1+p2)).data());
```



not available on expressions!

```
Matrix4f A, B;  
glUniformMatrix4fv((A*B).eval().data());
```



Are we sure the storage
order match?

Eigen & OpenGL

- <Eigen/OpenGLSupport>

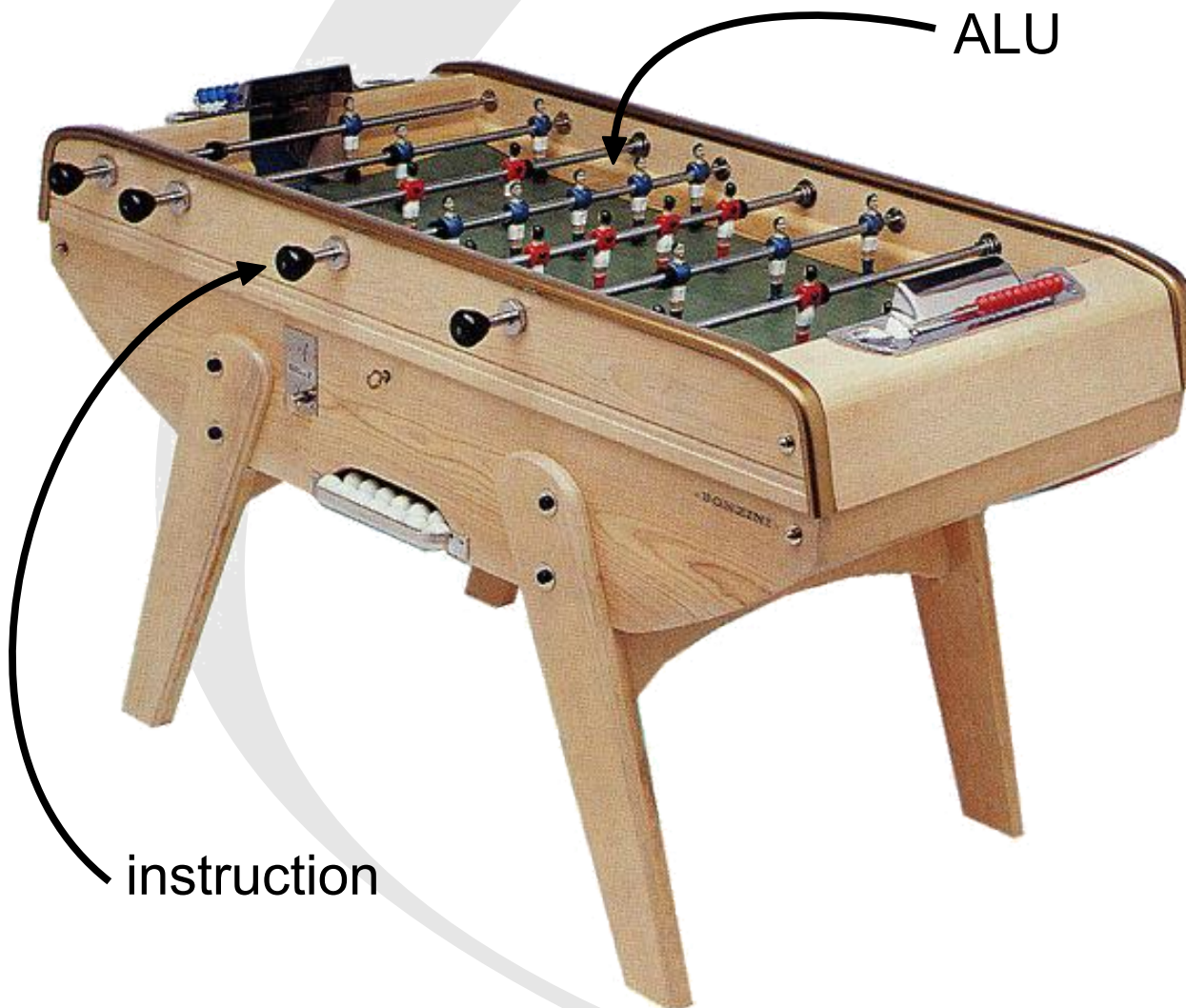
```
#include <Eigen/OpenGLSupport>  
using Eigen::glUniform;
```

```
Vector3f p1, p2;  
Matrix3f A;  
glUniform(p1+p2);  
glUniform(A*p1);  
glUniform(A.topRows<2>().transpose());
```

```
...
```

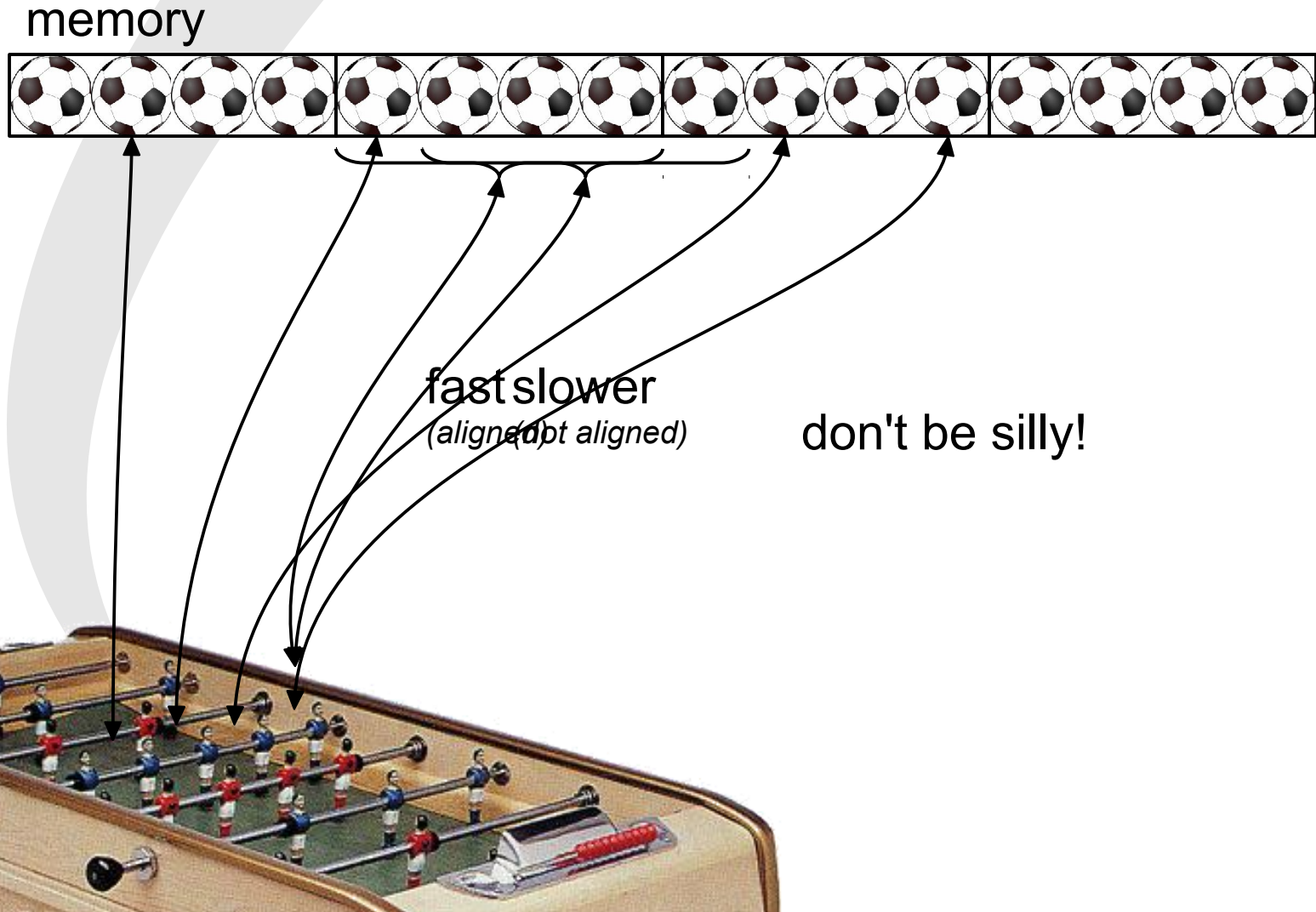
Vectorization of 3D vectors

Vectorization: difficulties?



Challenge:
*put 4 balls in front
of each player!*

Vectorization: difficulties?



AoS versus SoA

- Array of Structure

```
std::vector<Vector3f> points_aos;
```

- Structure of Array

```
struct SoA {  
    VectorXf x, y, z;  
};
```

```
SoA points_soa;
```

- Example: compute the mean

```
for(int i, ...) mean_aos += points_aos[i];
```

```
mean_soa <<  points_soa.x.mean(),  
             points_soa.y.mean(),  
             points_soa.z.mean();
```

AoS versus SoA

- Eigen's way

```
Matrix<float,3,Dynamic,ColMajor> points_aos;  
Matrix<float,3,Dynamic,RowMajor> points_soa;
```

- Highly flexible:

```
points_xxx.col(i) = Vector3f(...);
```

```
Affine3f T;  
point_xxx = T * points_xxx;
```

```
mean = points_xxx.rowwise().mean();
```

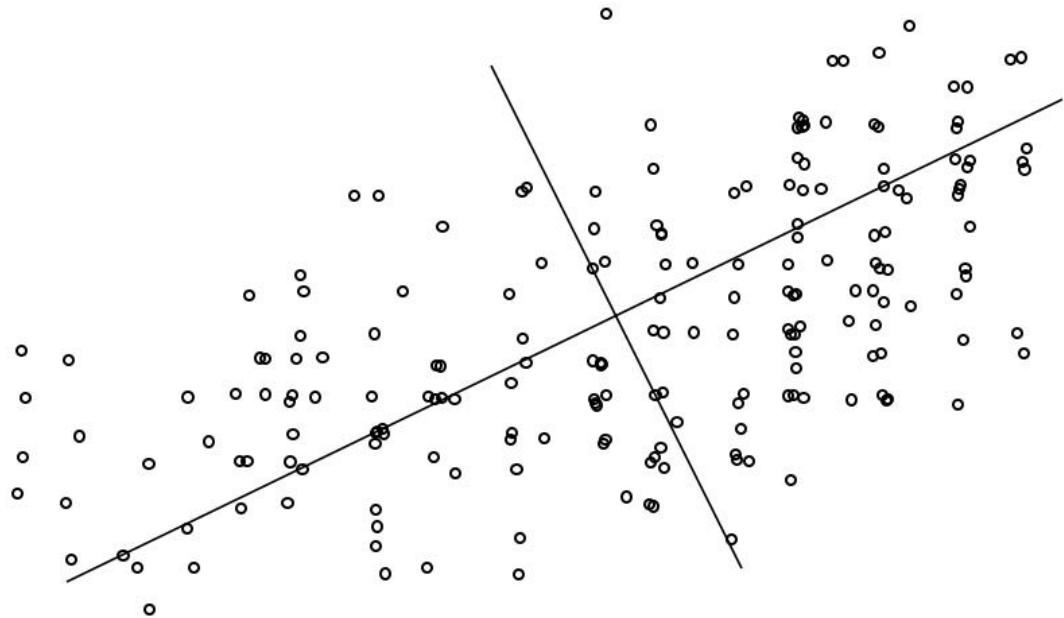
→ *demo*

Covariance Analysis

Covariance Analysis

- Example on 3D point clouds

```
Matrix3Xf points = ...;  
Matrix3Xf c_points = points.colwise() - points.rowwise().mean();  
Matrix3f cov = c_points * c_points.transpose();  
SelfAdjointEigenSolver<Matrix3f> eig(cov);
```



Covariance Analysis

- Example on 3D point clouds

```
Matrix3Xf points = ...;  
Matrix3Xf c_points = points.colwise() - points.rowwise().mean();  
Matrix3f cov = c_points * c_points.transpose();  
SelfAdjointEigenSolver<Matrix3f> eig(cov);
```

- normal estimations

```
Vector3f normal = eig.eigenvectors().col(0);
```

- local planar parameterization

```
Matrix3Xf l_points = eig.eigenvectors().transpose() * points;
```

- compute (cheap) oriented bounding boxes

```
AlignedBox3f bbox;  
for(int i=0; i<points.cols(); ++i)  
    bbox.extend(l_points.col(i));  
Quaternionf q(eig.eigenvectors().transpose());
```

Covariance Analysis

- Tips for 2D and 3D matrices
 - default iterative algorithm:

```
SelfAdjointEigenSolver<Matrix3f> eig;  
eig.compute(cov);
```

- closed form algorithms:
(fast but lack a few bits of precision)

```
SelfAdjointEigenSolver<Matrix3f> eig;  
eig.computeDirect(cov);
```

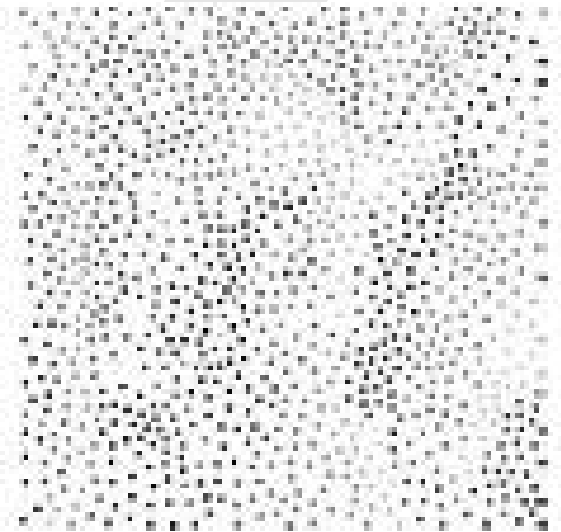


Linear Regression

(Dense Solvers)

Scattered Data Approximation

- Example in the functional settings



input:

- sample positions \mathbf{p}_i
- with associated values f_i

output:

- a smooth scalar field $f : \mathbb{R}^d \rightarrow \mathbb{R}$
s.t., $f(\mathbf{p}_i) \approx f_i$

Basis Functions Decomposition

- Express the solution as:

$$f(\mathbf{x}) = \sum_j \alpha_j \phi_j(\mathbf{x})$$

- Radial Basis Functions

$$f(\mathbf{x}) = \sum_j \alpha_j \phi(\|\mathbf{x} - \mathbf{q}_j\|)$$

– example:

- $\phi(t) = t^3$
- \mathbf{q}_j ? \rightarrow evenly distributed

Matrix formulation

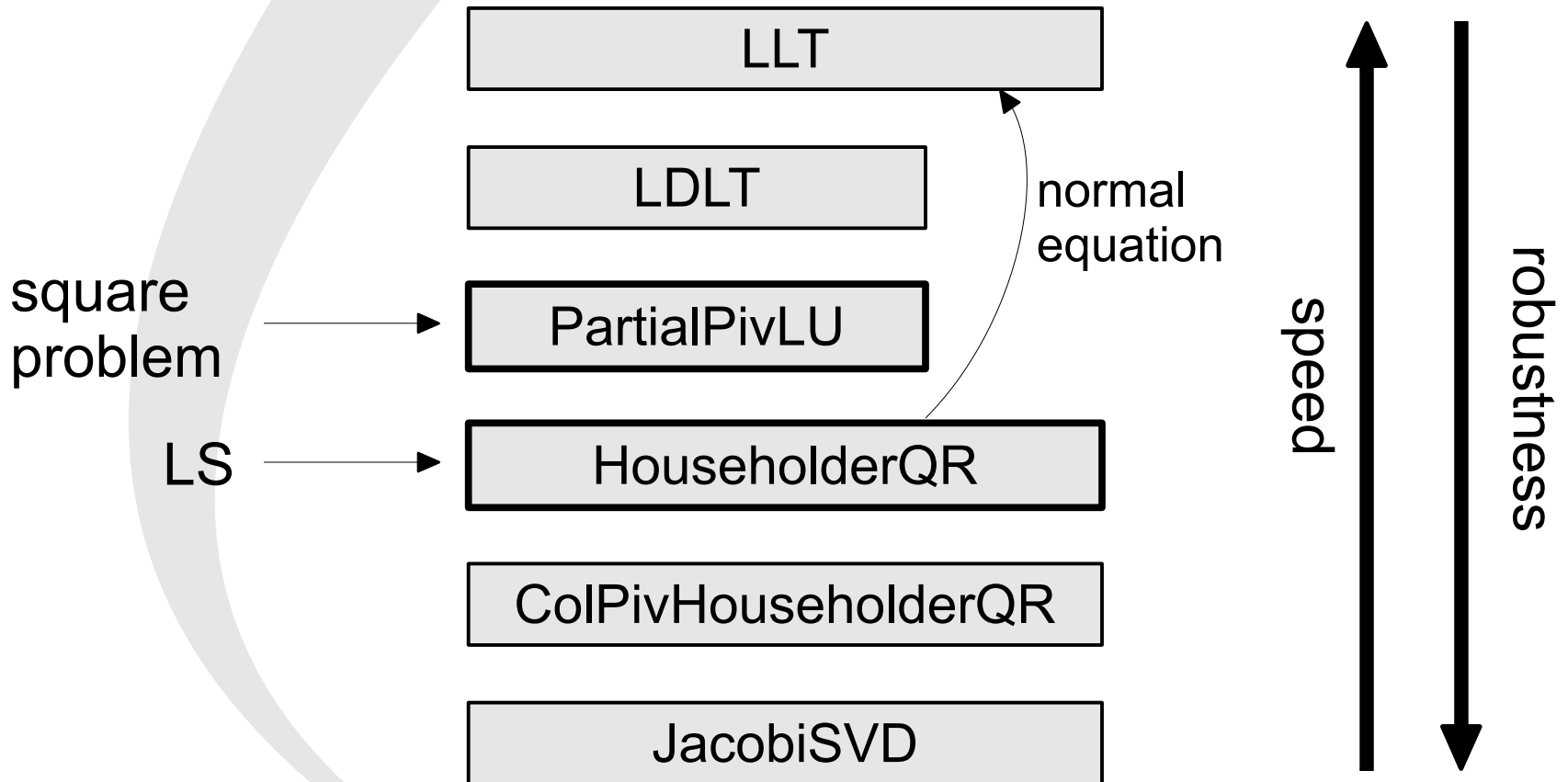
- Least square minimization: $\alpha = \arg \min_{\alpha} \sum_i (f(\mathbf{p}_i) - f_i)^2$
- Matrix form:

$$\begin{bmatrix} \vdots \\ \cdots \phi(\|\mathbf{p}_i - \mathbf{q}_j\|) \cdots \\ \vdots \end{bmatrix} \cdot \boldsymbol{\alpha} = \begin{bmatrix} \vdots \\ f_i \\ \vdots \end{bmatrix} \rightarrow \boldsymbol{\alpha} = \arg \min_{\alpha} \|A \boldsymbol{\alpha} - \mathbf{b}\|^2$$

- as many unknowns as constraints: $\Leftrightarrow A \boldsymbol{\alpha} = \mathbf{b}$
→ interpolation

→ *demo*

Dense Solvers



(bi-)Harmonic interpolation
(Sparse Solvers)

Laplacian equation

$$\Delta f = 0$$
$$\Delta f = \nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} + \dots$$

- Many applications
 - interpolation
 - smoothing
 - regularization
 - deformations
 - parametrization
 - etc.

Laplacian equation

$$\Delta f = 0$$

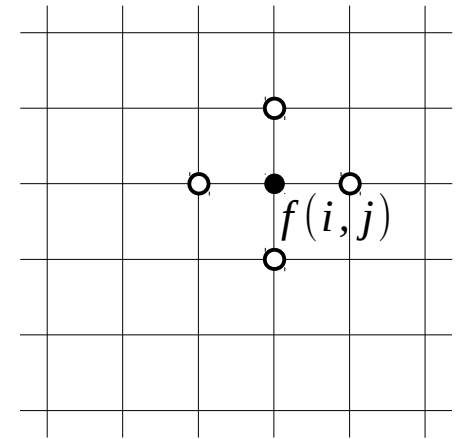


[courtesy of A. Jacobson et al.]

Discretization

- Example on a 2D grid:

$$\Delta \Leftrightarrow \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$



$$\Delta f(i, j) = \frac{(f(i-1, j) + f(i+1, j) + f(i, j-1) + f(i, j+1))}{4} - f(i, j) = 0$$

- Matrix form:

$$\mathbf{L} \mathbf{f} = 0$$

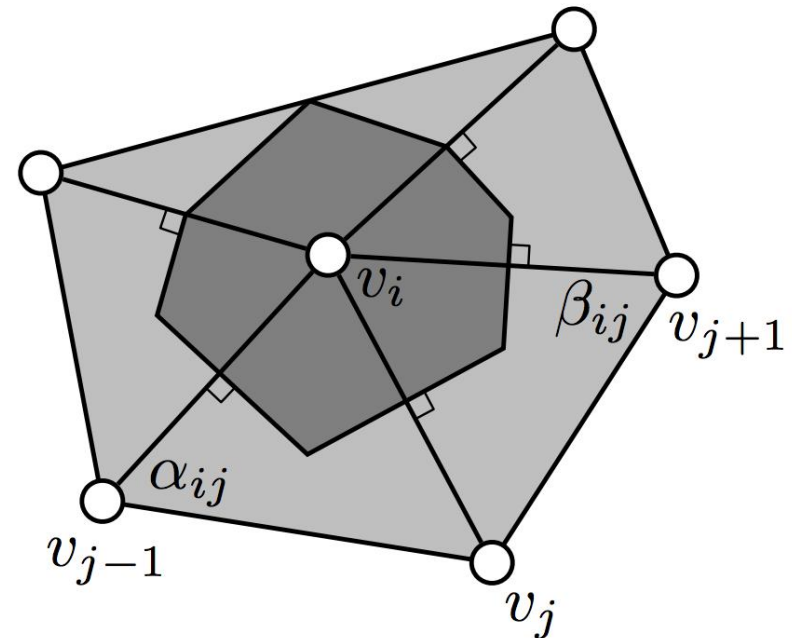
Discretization

- On a 3D mesh:

$$\Delta f(v_i) = \sum_{v_j \in N_1(v_i)} L_{i,j} (f(v_j) - f(v_i))$$

$$L_{i,j} = \frac{\cot \alpha_{ij} + \cot \beta_{ij}}{A_i + A_j}$$

$$L_{i,i} = - \sum_{v_j \in N_1(v_i)} L_{i,j}$$



[courtesy of M. Botsch and O. Sorkine]

→ demo

Sparse Representation

- Naive way:

```
std::map<pair<int,int>, double>
```

- Eigen::SparseMatrix

– Matrix:

0	3	0	0	0
22	0	0	0	17
7	5	0	1	0
0	0	0	0	0
0	0	14	0	8

– Compressed Column-major Storage:

Values:	22	7	3	5	14	1	17	8
InnerIndices:	1	2	0	2	4	2	1	4
OuterStarts:	0	2	4	5	6	8		

Constraints

- Dirichlet boundary conditions
 - fix a few (or many) values: $f(v_i) = \bar{f}_i, v_i \in \Gamma$
 - updated problem:

$$\begin{bmatrix} \mathbf{L}_{00} & \mathbf{L}_{01} \\ \mathbf{L}_{10} & \mathbf{L}_{11} \end{bmatrix} \cdot \begin{bmatrix} \hat{\mathbf{f}} \\ \bar{\mathbf{f}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \theta \end{bmatrix} \Rightarrow \mathbf{L}_{00} \cdot \hat{\mathbf{f}} = -\mathbf{L}_{01} \cdot \bar{\mathbf{f}}$$

→ *demo*

Bi-harmonic interpolation

- Continuous formulation:

$$\Delta \cdot \Delta f = 0$$

- Discrete form:

$$\mathbf{L} \cdot \mathbf{L} \cdot \mathbf{f} = 0$$

→ *demo*

Solver Choice

- Questions:
 - Solve multiple times with the same matrix?
 - yes → direct methods
 - Dimension of the support mesh
 - 2D → direct methods
 - 3D → iterative methods
 - Can I trade the performance? Good initial solution?
 - yes → iterative methods
 - Hill conditioned?
- Still lost? → sparse benchmark

→ *demo*



What next?

Inria

Coming soon: 3.2

- Already in 3.2-beta1
 - SparseLU
 - SparseQR
 - GeneralEigenSolver ($Ax=IBx$)
 - Ref<>
 - write generic but non-template function!

WIP: AVX

- AVX
 - SIMD on 256bits register (8 floats, 4 doubles)
 - ... or 128bits (4 floats, 2 doubles)
- Challenge
 - select the best register-width

WIP: CUDA

- Why only now?
 - CUDA 5 made it possible
- Roadmap
 - call Eigen from CUDA kernel
 - useful for small fixed size algebra
 - add a CudaMatrix class
 - coefficient-wise ops
 - special assignment evaluator
 - products & solvers
 - wrap optimized CUDA libraries (ViennaCL, CuBlas, Magma, etc.)

WIP: SparseMatrixBlock

- SparseMatrixBlock
 - “SparseMatrix<Matrix4f>”
 - useful with high-order elements
 - classic with iterative methods (→ ViennaCL)
 - would be a first with direct methods!
 - huge speed-up expected!

WIP: Non-Linear Optimization

- Non-linear least square
 - Generic Levenberg-Marquart
 - Dense & Sparse
- Quadratic Programming
 - linear least-square + inequalities

WIP: utility modules

- Auto-diff
- Polynomials
 - differentiation & integration



Concluding remarks

License

- Initially:
 - LGPL3+
 - default choice
 - not as liberal as it might look...
- Now:
 - MPL2 (Mozilla Public License 2.0)
 - same spirit but with tons of advantages:
 - accepted by industries
 - do work with header only libraries
 - versatile (apply to anything)
 - a lot simpler
 - good reputation

Developer Community

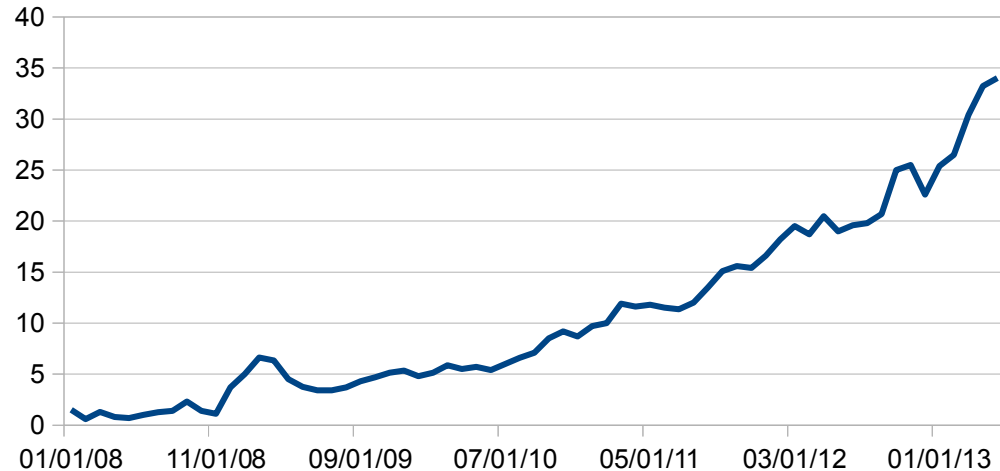
- Jan 2008: start of Eigen2
 - part of KDE
 - packaged by all Linux distributions
 - open repository
 - open discussions on mailing/IRC
 - 300 members, 300 messages/month
 - good quality API
- Today
 - most development @ Inria (Gaël + full-time engineer)
- Future
 - consortium... ??

User community

- Active project with many users

- Website:

- ~30k unique visitors/months



- Major domains

- Geometry processing, Robotics, Computer vision, Graphics

Summary

- **Many unique features:**
 - C++ friendly API
 - Easy to use, install, distribute, etc.
 - Versatile
 - small, large, sparse
 - custom scalar types
 - large set of tools
 - No compromise on performance
 - static allocation, temporary removal, unrolling, auto vectorization, cache-aware algorithms, multi-threading, etc.
 - Multi-platforms

Acknowledgements

- Main contributors
 - Benoit Jacob,
 - Jitse Niesen,
 - Hauke Heibel,
 - Désiré Nuentza,
 - Christoph Hertzberg,
 - Thomas Capricelli

+ 100 others
- You're welcome to join!
 - documentation
 - bug report/patches
 - write unit tests
 - discuss the future on ML
 - ...
 - don't be shy!

