Eigen

a c++ linear algebra library

Gaël Guennebaud

[http://eigen.tuxfamily.org]

Inria

Outline

- Eigen as a library
  - Why such a library?
  - What's so special?
  - How does it work?
- Eigen as an opensource community?

Why?

Context

- Matrix computation 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, smartphone, embedded systems, etc.
    - real-time performance

Matrix computation?

MatLab

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

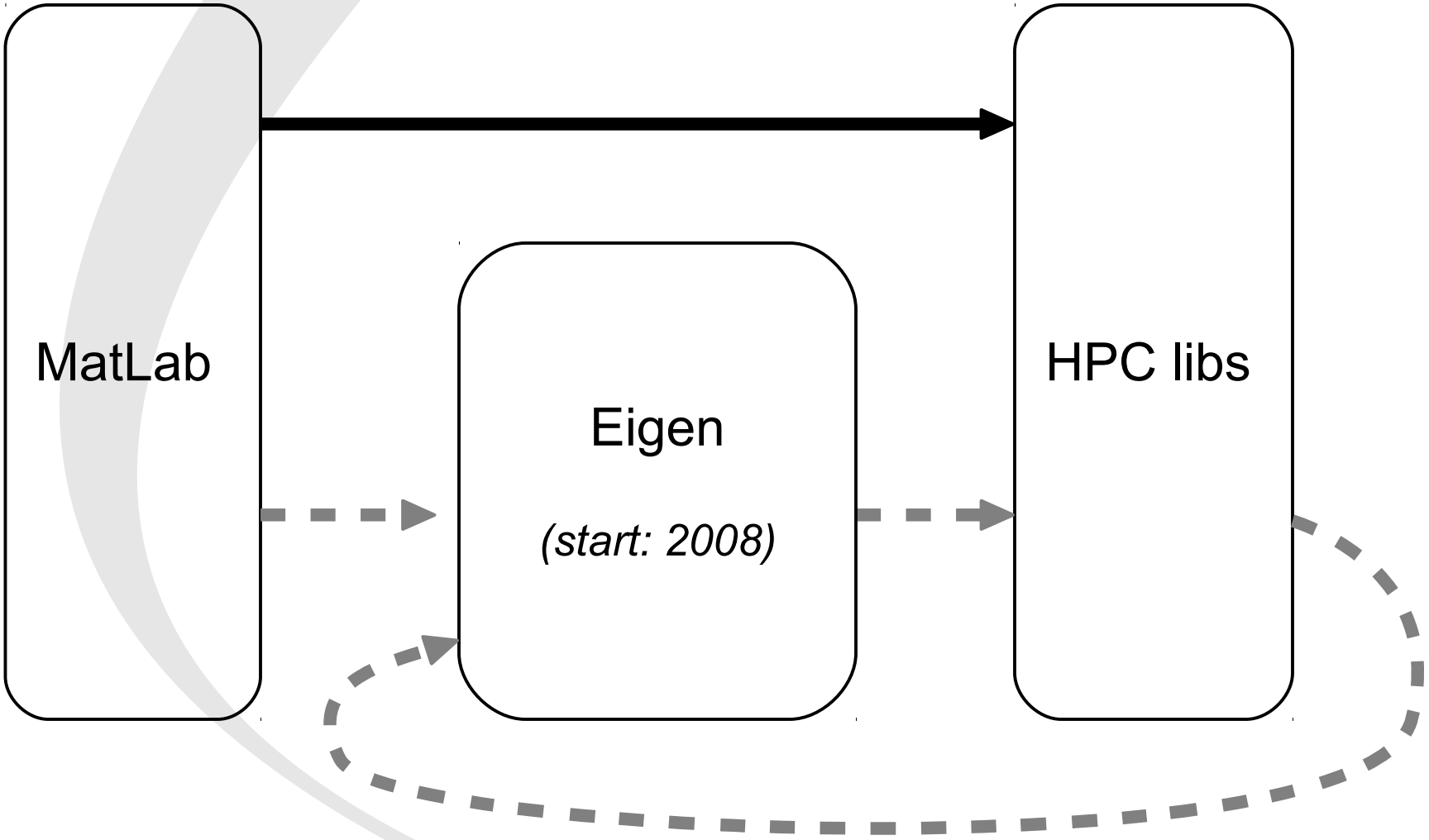
→ **Prototyping**

HPC libs

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

→ **Advanced usages**

Context



What?

Facts

- Pure C++ template library
  - header only, no binary to compile/install...
  - no dependency (optional only)
  - opensource: MPL2

→ **easy to install & distribute**

- Multi-platforms
  - GCC, MSVC, Intel ICC, Clang/LLVM
  - SSE(x86), NEON (ARM), Altivec (PowerPC)

Increasing feature set

- Core
  - Matrix and array manipulation (~MatLab, 1D & 2D)
  - Basic linear algebra (~BLAS)
- 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>
    ```

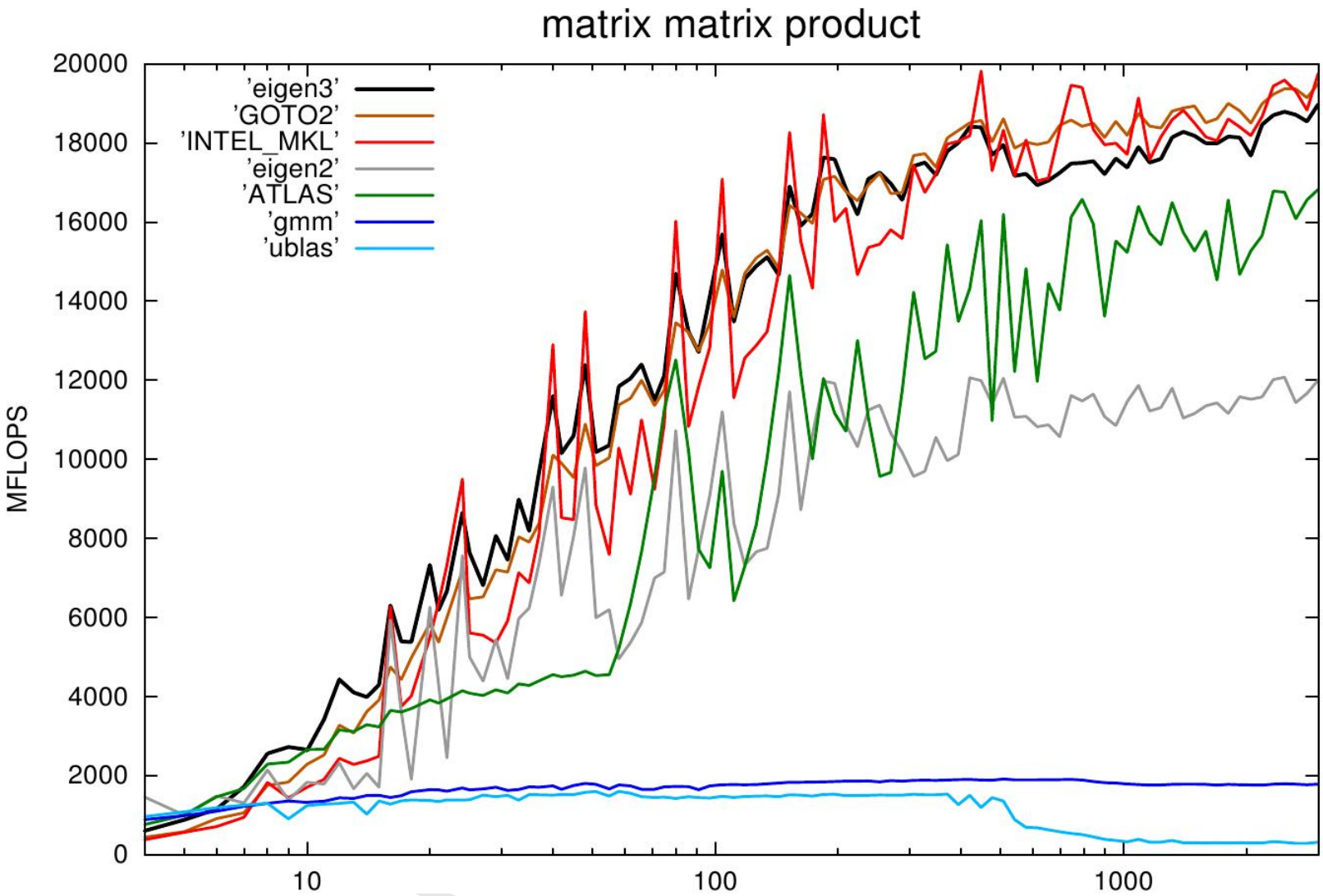
  - malloc-free
  - meta unrolling
- Large objects
  - means dynamic sizes

    ```
    Matrix<float,Dynamic,1>
    ```

  - cache friendly kernels *(perf ~ MKL)*
  - multi-threading *(OpenMP)*

- Vectorization (SIMD)
- Unified API → write generic code
- Mixed fixed/dynamic dimensions

Performance



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> MatrixX;
MatrixX A, B, X;
// init A and B
// solve for A.X=B using LU decomposition
X = A.lu().solve(B);
```

How?

Expression templates

- Example:

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

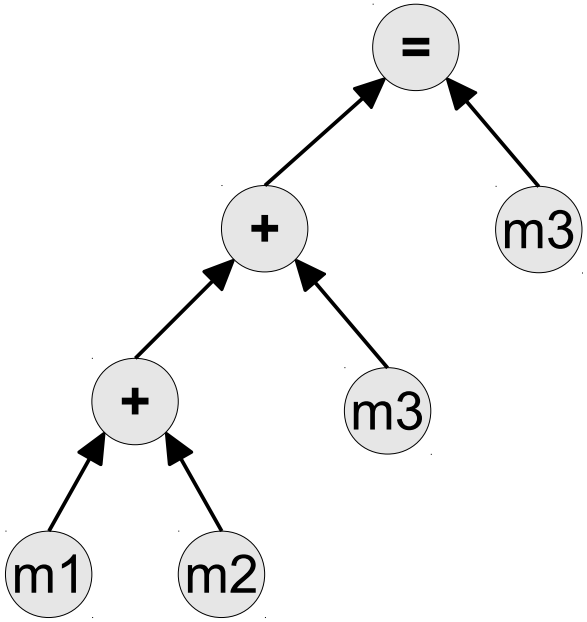
- Standard C++ way:

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

Expression templates

- Example:

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



- Expression templates:
  - "+" returns an expression
    => expression tree
  - e.g.: A+B returns:

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

  - complete example:

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

Expression templates

- Example:

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

- Evaluation:
  - Top-down creation of an evaluator
    - e.g.:

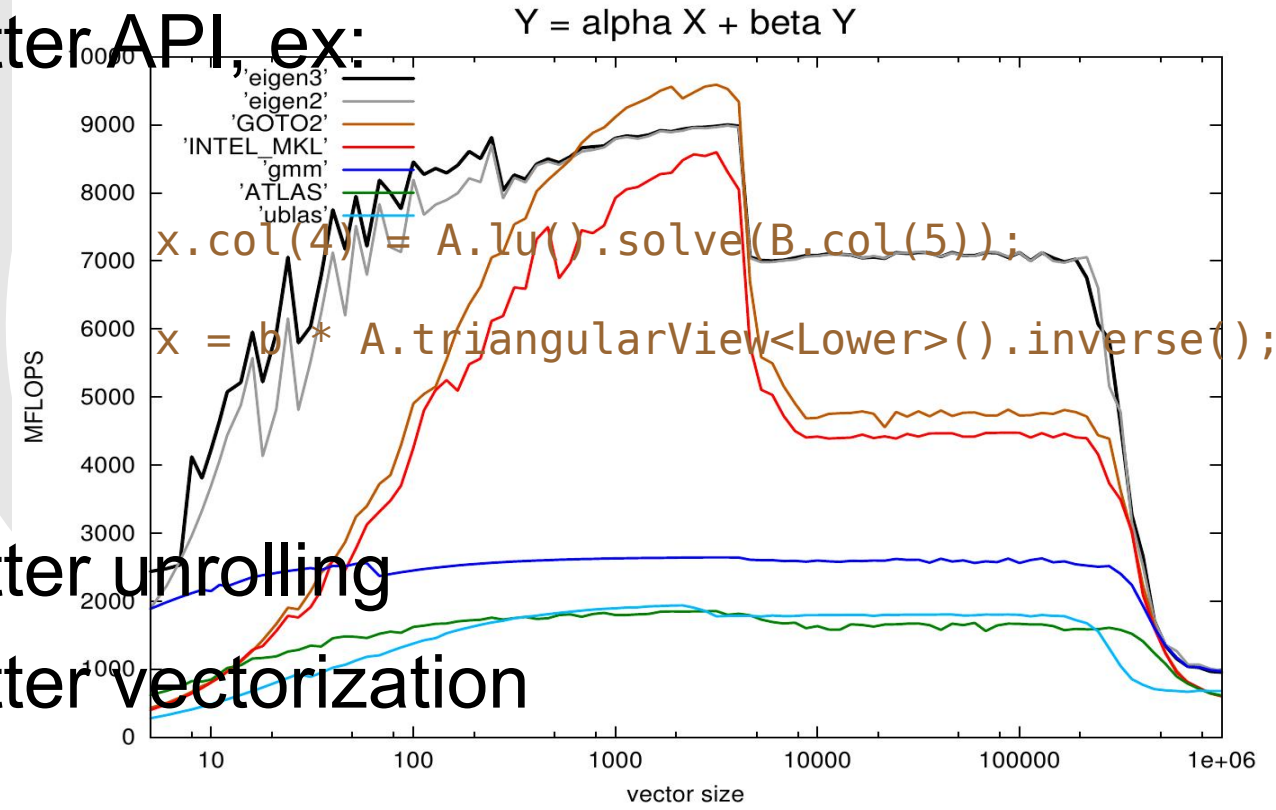
```
Evaluator<Sum<type_of_A,type_of_B> > {
  Evaluator<type_of_A> evalA(A);
  Evaluator<type_of_B> evalB(B);
  Scalar coeff(i,j) {
    return evalA.coeff(i,j) + evalB.coeff(i,j);
  }
};
```

  - Assignment produces:

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

ET: Immediate benefits

- Fused operations
  - Temporary removal
  - Reduce memory accesses, cache misses
- Better API, ex:




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

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

- Better unrolling
- Better vectorization

Cost model

- Cost Model
  - Track an approximation of the cost to evaluate one coefficient
- Control of:
  - loop unrolling (partial)
  - evaluation of sub expressions, e.g.:
    - *(**a+b**) * c* → *(a+b)* is evaluated into a temporary
    - enable vectorization of sub expressions

Top-down expression analysis

- Products
  - detect BLAS-like sub expressions
    - e.g.: `m4 -= 2 * m2.adjoint() * m3;`

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

```
m4.block(...) +=   ((2+4i) * m2).adjoint()
                 * m3.block(...).transpose();
```

```
Evaluator<Product<type_of_A,type_of_B> > {
  EvaluatorForProduct<type_of_A> evalA(A);
  EvaluatorForProduct<type_of_B> evalB(B);
};
```

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> > { … };
```

Tree optimizer

- Last example:

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

  - Tree optimizer

→

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

  - Rule:

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

Community?

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

      → high quality API
- Today
  - most development @ Inria (Gaël + full-time engineer)
- Future

  → consortium... ??

User community

- Active project with many users
  - Website: ~30k unique visitors / month (+10% / month)
- Robotics, computer vision, graphics
  - Google street view
  - Willow garage (ROS, PCL)
  - CEA

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

Last slide

- What next?
  - AVX, CUDA
  - More parallelization
  - Sparse block matrices
  - Non linear optimization
  - Utility modules
    - Polynomial manipulation/solver
    - Autodiff
- Main contributors
  - Benoit Jacob, Jitse Niesen, Hauke Heibel,
    Désiré Nuentsa, Christoph Hertzberg, Thomas Capricelli