Eigen: A C++ Linear Algebra Template Library

Md Ashiqur Rahman

Outline

- Introduction & Motivation
- How it works
- Implementation of Eigen
  - Expression templates, Lazy evaluation, Vectorization
- Aliasing problems
- Platforms
- Eigen vs BLAS/Lapack
- Benchmark
- Conclusion

Introduction

- A C++ template library for linear algebra
- Header only, nothing to install or compile
- Provide good speed, simple interface and use
- Opensource

Why Another Library

- Multiplatform and Good compiler support
- A single unified library
- Most libraries specialized in one of the features or module
- Eigen satisfy all these criteria

-free, fast, versatile, reliable, decent API, support for both sparse and dense matrices, vectors and array, linear algebra algorithms (LU, QR, ...), geometric transformations.

How it works

- Takes 3 compulsory and 3 optional arguments

```
Matrix<typename Scalar,
       int RowsAtCompileTime,
       int ColsAtCompileTime,
       int Options = 0,
       int MaxRowsAtCompileTime = RowsAtCompileTime,
       int MaxColsAtCompileTime = ColsAtCompileTime>
```

- Could be different types

```
typedef Matrix<float, 4, 4> Matrix4f;
typedef Matrix<double, Dynamic, Dynamic> MatrixXd;
typedef Matrix<float, 3, 1> Vector3f;

typedef Matrix<int, 1, 2> RowVector2i;
```

Eigen Implementation: 1D array

- Simple matrix addition example

```
int size = 50;
Eigen::VectorXf u(size), v(size), w(size);
u = v + w;
```

- Use one dimensional array, one loop to traverse the array

```
for(int i = 0; i < 50; ++i)
u[i] = v[i] + w[i];
```

Eigen Implementation: use expression template

- Addition should be done using temporary object

```cpp
VectorXf tmp = v + w;
VectorXf u = tmp;
for(int i = 0; i < size; i++) tmp[i] = v[i] + w[i];
for(int i = 0; i < size; i++) u[i] = tmp[i];
```

- Eigen uses expression template to prevent unnecessary use of temporary objects.

```cpp
for(int i = 0; i < size; i++) u[i] = v[i] + w[i];
```

```cpp
int size = 50;
Eigen::VectorXf u(size), v(size), w(size);
u = v + w;
```

Eigen Implementation: lazy evaluation

- Intelligent lazy evaluation of expressions.

- Exceptions:
    - Matrix product
    - Nested expressions

```
matrix1 = matrix2 + matrix3 * matrix4;
```

    - If cost model results to choose immediate evaluation

```
matrix1 = matrix2 * (matrix3 + matrix4);
```

Eigen Implementation: lazy or immediate evaluation

- Assignment operator implementation (=)

```
template<typename Derived>
template<typename OtherDerived>
inline Derived& MatrixBase<Derived>
::operator=(const MatrixBase<OtherDerived>& other)
{
return internal::assign_selector<Derived,OtherDerived>::run(derived(), other.derived());
}
```

- Internal::assign_selector

```
template<typename Derived, typename OtherDerived,
bool EvalBeforeAssigning = int(OtherDerived::Flags) & EvalBeforeAssigningBit,
bool NeedToTranspose = Derived::IsVectorAtCompileTime
        && OtherDerived::IsVectorAtCompileTime
        && int(Derived::RowsAtCompileTime) == int(OtherDerived::ColsAtCompileTime)
        && int(Derived::ColsAtCompileTime) == int(OtherDerived::RowsAtCompileTime)
        && int(Derived::SizeAtCompileTime) != 1>
struct internal::assign_selector;
```

Eigen Implementation: Automatic vectorization

- Does automatic vectorization by itself, not compiler dependent.

- Different vectorization for different architecture

- SIMD instruction sets SSE2, AltiVect, ARM NEON

Eigen Implementation: Automatic vectorization

- SSE, NEON works with 16 bytes packets.

- 4 floats or ints or 2 doubles per packets.

- 4 Addition per packets

- Our vector size 50,

```
for(int i = 0; i < 4*(size/4); i+=4) u.packet(i) = v.packet(i) + w.packet(i);
for(int i = 4*(size/4); i < size; i++) u[i] = v[i] + w[i];
```

```
int size = 50;
Eigen::VectorXf u(size), v(size), w(size);
u = v + w;
```

Eigen Implementation: which vectorization to use

- Implemented in an helper class **internal::assign_traits**

```
enum {
        StorageOrdersAgree = (int(Derived::IsRowMajor) == int(OtherDerived::IsRowMajor)),
        MightVectorize = StorageOrdersAgree && (int(Derived::Flags) & int(OtherDerived::Flags) & ActualPacketAccessBit),
        MayInnerVectorize  = MightVectorize && int(InnerSize)!=Dynamic && int(InnerSize)%int(PacketSize)==0
                                                                         && int(DstIsAligned) && int(SrcIsAligned),
        MayLinearize = StorageOrdersAgree && (int(Derived::Flags) & int(OtherDerived::Flags) & LinearAccessBit),
        MayLinearVectorize = MightVectorize && MayLinearize && DstHasDirectAccess && (DstIsAligned || MaxSizeAtCompileTime==Dynamic),
        MaySliceVectorize  = MightVectorize && DstHasDirectAccess && (int(InnerMaxSize)==Dynamic || int(InnerMaxSize)>=3*PacketSize)
 };
```

Eigen Implementation: Linear Vectorization implementation

- Need to skip first few coefficients to group coefficients by packets of 4.
- First, determine architecture specific packet size

```
const int packetSize = internal::packet_traits<typename Derived1::Scalar>::size;
```

- Start of first coefficient

```
const int alignedStart = internal::assign_traits<Derived1,Derived2>::DstIsAligned ? 0 :
     internal::first_aligned(&dst.coeffRef(0), size);
```

- Skipping coefficients

```
for(int index = 0; index < alignedStart; index++)
dst.copyCoeff(index, src);
```

Eigen Implementation: Linear Vectorization implementation

- Vector size 50 is not multiple of packet size 4 floats, 48 is the maximum number.

```
const int alignedEnd = alignedStart + ((size-alignedStart)/packetSize)*packetSize;
```

- Vectorization part

```
for(int index = alignedStart; index < alignedEnd; index += packetSize)
{
dst.template copyPacket<Derived2, Aligned, internal::assign_traits<Derived1,Derived2>::
SrcAlignment>(index, src);
}
```

- Last two coefficients

```
for(int index = alignedEnd; index < size; index++)
dst.copyCoeff(index, src);
```

Aliasing Problem

- Occurs when a matrix operation applied on a matrix and saved in the same matrix.

```
mat = mat.transpose();
```

- Produce wrong results.

- Solution is to use temporary variable

```
tmp = mat.transpose();
mat = tmp;
```

Platforms

- Supported compilers:

  – GCC (from 3.4 to 4.6) , MSVC (2005,2008,2010) , Intel ICC, Clang/LLVM

- Supported systems:

  – x86/x86_64 (Linux,Windows)

  – ARM (Linux), PowerPC

- Supported SIMD vectorization engines:

  – SSE2, SSE3, SSSE3, SSE4

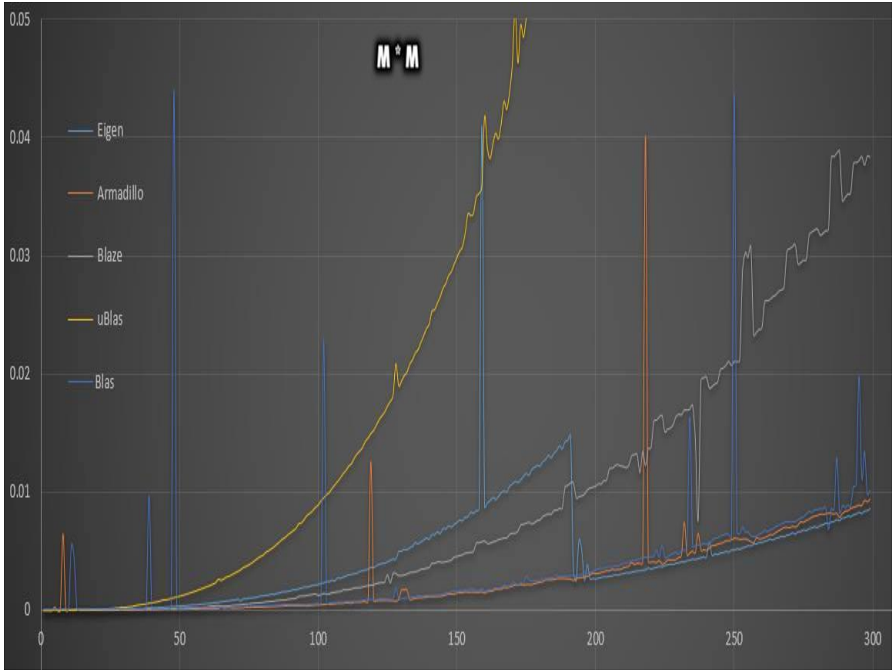
  – NEON (ARM)

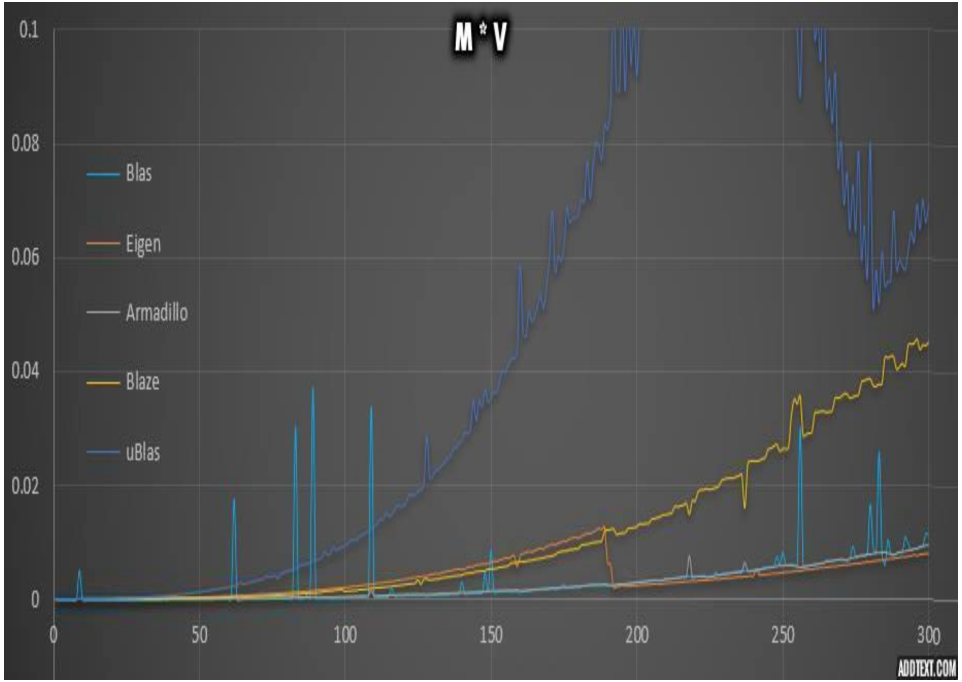
  – Altivec (PowerPC)

Eigen vs BLAS/Lapack

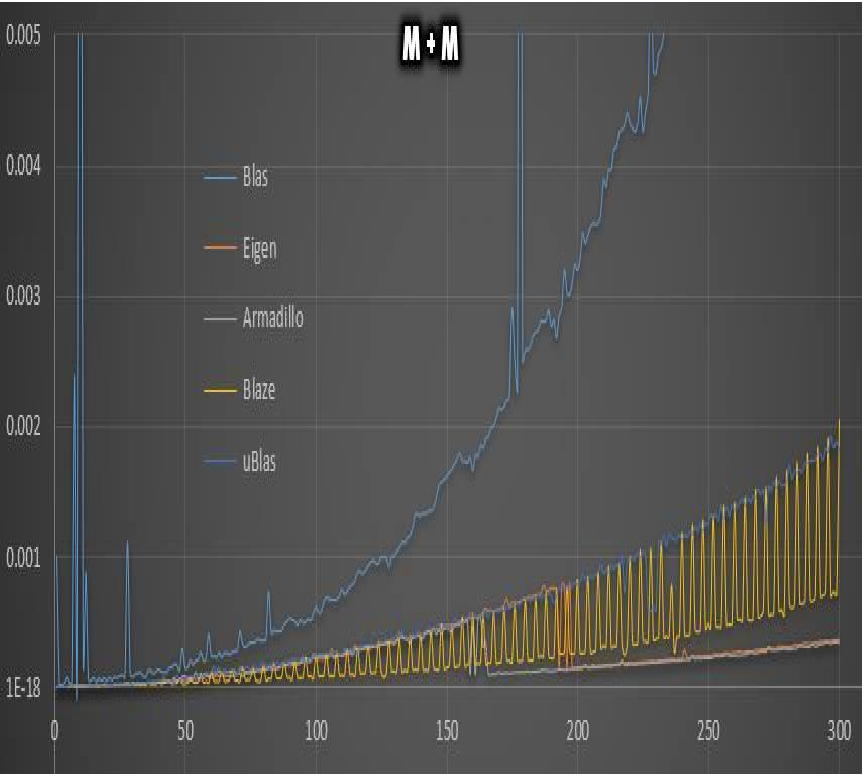
- Fixed size matrices, vectors
- Sparse matrices and vectors
- More features like Geometry module, Array module
- Most operations are faster or comparable with MKL and GOTO
- Better API
- Complex operations are faster

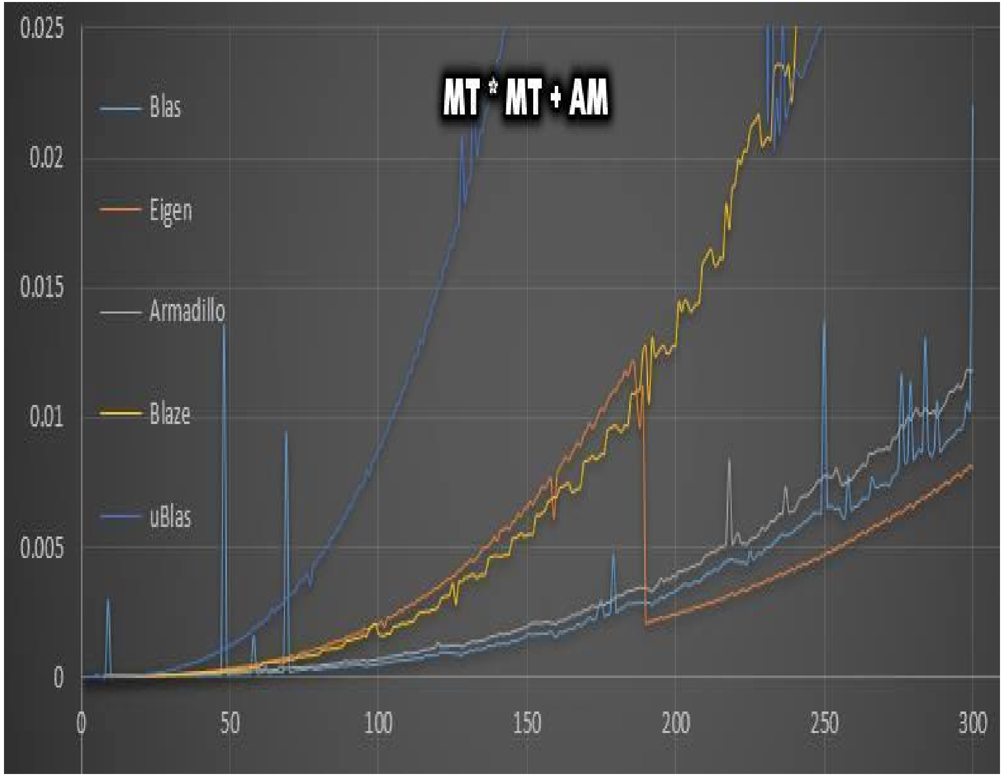
Benchmark


M * M
0.05
0.04
0.03
0.02
0.01
0
Eigen
Armadillo
Blaze
uBlas
Blas
0
50
100
150
200
250
300


M * V
0.1
0.08
0.06
0.04
0.02
0
Blas
Eigen
Armadillo
Blaze
uBlas
0
50
100
150
200
250
300
ADDTEXT.COM

Benchmark


M + M
0.005
0.004
0.003
0.002
0.001
1E-18
0
50
100
150
200
250
300
Blas
Eigen
Armadillo
Blaze
uBlas


MT * MT + AM
0.025
0.02
0.015
0.01
0.005
0
50
100
150
200
250
300
Blas
Eigen
Armadillo
Blaze
uBlas

Conclusion

- From benchmark it shows, eigen is comparable with most linear algebra library available.

- Simple interface make it more attractive

- Low memory overhead

- All features and modules in a single library make it more usable.