Google Summer of Code 2018 - Faster Matrix Algebra for ATLAS

Evaluation Test

David A. Tellenbach

February 27, 2018

1 Overview

2 Class Design

The class SymmetricMatrix is actually a class template with two template paramters `typename Scalar` and `int Dimension`. The first describes the type of the values of the matrix, the second one its dimension. Since any symmetric matrix is a square matrix specifying one dimension is enough.

2.1 Storage

One of the tasks of this evaluation tests was to store only those elements of a symmetric matrix that determines it completely. A $(n \times n)$ matrix contains of n^2 elements. In the symmetric case the ij-th element is equal to the ji-th one, therefore it is sufficient to store

$$\frac{n(n+1)}{2}$$

of its elements, e.g., the upper triangular part of it. Even though matrices are usually considered to be two-dimensional objects hardware memory is linear. The current implementation of the SymmetricMatrix class stores the matrix elements in either an `std::vector` or `std::array`. When storing these elements two storage orders can be considered: Row and column major storing.

Row Major The matrix elements are stored packed row by row as illustrated in Figure 1. Currently the SymmetricMatrix class stores matrix elements row major.

Column Major The other storage order that can be considered is to store matrix elements packed column by column. Such a storage order is called column major and is illustrated in Figure 2.

Storage order can dramatically determine the performance of matrix operations since it determines who matrix elements are loaded into cache.

2.2 Compile- and runtime

Following the design of Eigen the implementation of this evaluation test contains both fixed and dynamically sized symmetric matrices. The generic class template `SymmetricMatrix<typename Scalar, int Dimension>` contains the implementation of fixed sized matrices. The elements are stored in an `std::array`.

$$\begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} \\ & \alpha_{11} & \alpha_{12} \\ & & \alpha_{22} \end{bmatrix} \Rightarrow [\alpha_{00}, \alpha_{01}, \alpha_{02}, \alpha_{11}, \alpha_{12}, \alpha_{22}]$$

Figure 1: Packed storage of a 3×3 matrix in row major order.

$$\begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} \\ & \alpha_{11} & \alpha_{12} \\ & & \alpha_{22} \end{bmatrix} \Rightarrow [\alpha_{00}, \alpha_{01}, \alpha_{11}, \alpha_{02}, \alpha_{12}, \alpha_{22}]$$

Figure 2: Packed storage of a 3×3 matrix in column major order.

Dimension	t1 (Fixed)	t2 (Dynamic)
10	215 ns	731 ns
20	656 ns	2306 ns
30	1413 ns	4573 ns
40	2530 ns	7844 ns
50	3914 ns	11896 ns
60	5812 ns	17107 ns
70	7723 ns	23577 ns
80	10185 ns	30322 ns
90	13629 ns	39718 ns
100	16239 ns	47402 ns

Table 1: Time consumption of adding fixed sized and dynamically sized matrices.

The size of this underlying container is calculated at compile-time.
If the special dimension `Eigen::Dimension` that is just a typedef for -1 is passed, the partial template specialization `SymmetricMatrix<typename Scalar, Eigen::Dynamic>` is used. This class is more flexible than the fixed sized case but is less performant as Table 1 shows.

Since the memory allocation for fixed sized matrices is done during compilation it is completly located on the stack. Since stack size is limited matrices of this type are limited to relativly small dimensions. Large matrices containing tousands of elements should (and often must) be allocated on the heap. In this case dynamically sized matrices are the right choice.

2.3 Addition and Subtraction

Addition and subtraction are important component wise operations of matrices its complexity is always bounded from above by n^2 for a matrix of dimension n. Since we are not storing all n^2 elements of a symmetric matrix, we can perform even better.
The actual implementation works just by adding up the elements of the underlying containers, i.e., instances of `std::array` in the fixed sized case and `std::vector` in the dynamically sized case. E.g., Listing 1 show the implementation of the addition of two dynamically sized matrices.

2.4 Multiplication

Multiplication is a far more complex task than simple component wise operations like addition or subtraction. The simple implementation contains of three nested for-loops and the constant switch between matrix rows and matrix columns is pure cache horror. Big projects like BLAS have optimized matrix multiplication and its implementations like the Intel Math Kernel Library are complex and not easy to mimic. Trying to create an implementation beating or even being competitive with Eigens internal multiplication implementation is no realistic task for this evaluation test.
However the product of two symmetric matrices is in general not symmetric (it is and only is if the matrix product is commutative). Therefore this implementation uses the following trick: We construct instances of Eigen::Matrix from the symmetric matrices and use Eigens internal mechanism to multiply these. Since the result of the multiplication will be Eigen::Matrix anyway, we won't have to issue about memory usage. One could now argue that the temporary constructed instances of Eigen::Matrix will consume memory. This is right in general but by using an optimization technique that can be avoided (at least when the compiler runs with optimizer flags). See Listing 2 to see a concrete implementation of the multiplication of the symmetric matrices of dynamic size.

```cpp
SymmetricMatrix<Scalar>
operator+(const SymmetricMatrix<Scalar>& other) {
    // Check if both dynamic dimensions match
    if (dimension != other.dim()) {
        throw std::invalid_argument("Operation + cannot be performed "
                                    "for instances of SymmetricMatrix "
                                    "with not matching dimension");
        }

    // Construct new matrix and set underlying std::vector
    // by passing the underlying std::vector of this
    SymmetricMatrix<Scalar> ret(elements);

    // Just add up both underlying std::vector
    for (int i = 0; i < elements.size(); ++i) {
        ret.elements[i] += other.elements[i];
    }
    return ret;
}
```

Listing 1: Overloaded operator + for the addition of two dynamically sized matrices.

```cpp
Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>
operator*(SymmetricMatrix<Scalar>& other) {
    // The instance of Eigen::Matrix is constructed in return statement
    // This allow the compiler to optimize temporary instances away
    return Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>(
            constructEigenMatrix() * other.constructEigenMatrix()
        );
}
```

Listing 2: Overloaded operator ∗ for the multiplication of two dynamically sized matrices.

2.5 Runtime Exceptions and Compiletime Errors

Another task of the evaluation test was to throw exceptions whenever an operations is performed where the dimensions of the operands don't match. In the particular case one has to consider the both cases of dynamically and fixed sized matrices differently:

Fixed size operates with fixed size The easiest case. The compiler performs statically typechecks and the implementation guarantees that a compile time error will raise if the operands have non-matching dimensions. This holds for operations with fixed sized instances of SymmetricMatrix and fixed sized instances of Eigen::Matrix. See Listing 3 for an example that will not compile.

Fixed size operates with dynamic size In these cases the implementation checks if the dimensions match during runtime. If they dont, an excpetion is thrown.

Dynamic size operates with dynamic size In these cases the implementation checks if the dimensions match during runtime. If they dont, an excpetion is thrown. See Listing 4 for an try-catch example.

```cpp
// Symmetric matrix of ints with fixed dimension 3 filled with random values
SymmetricMatrix<int, 3> mat1 = SymmetricMatrix<int, 3>::Random();

// Symmetric matrix of ints with fixed dimension 5 filled with random values
SymmetricMatrix<int, 5> mat2 = SymmetricMatrix<int, 5>::Random();

mat1 + mat2;    // Compiler error
```

Listing 3: Addition of matrices with different fixed size. This example will not compile.

```cpp
// Symmetric matrix of ints with dynamic dimension 3 filled with random values
SymmetricMatrix<int> mat1 = SymmetricMatrix<int>::Random(3);

// Symmetric matrix of ints with dynamic dimension 5 filled with random values
SymmetricMatrix<int, 5> mat2 = SymmetricMatrix<int>::Random(5);

try {
    mat1 + mat2;
} catch (std::exception& ex) {
    std::cout << ex.what() << "\n";
}
/* Output: Operation + cannot be performed for instances of SymmetricMatrix with not
   matching dimension */
```

Listing 4: Addition of matrices with different dynamic size.

3 Benchmarks

4 Futur Work

This section explains

4.1 Existing matrix classes in Eigen

4.2 Implementation ideas