Scientific Software / Technisch Wetenschappelijke Software Homework Assignment 5

Matrix–Vector Multiplication using Expression Templates

Clémence Duplat r0977801

Assignment

Question 1

For each of the 5 different implementations, derive their asymptotic time complexity theoretically based on the given code.

- **NO-ET**
  Complexity:

  $\mathcal{O}(N^2)$

  Derivation:

  With no expressions templates, the operations are performed explicitly in the straightforward manner.
  - $X * x$: matrix-vector multiplication two nested loops $\mathcal{O}(N^2)$
  - transpose(X): two nested loops $\mathcal{O}(N^2)$
  -$transpose(X) * (X * x)$: matrix-vector multiplication two nested loops $\mathcal{O}(N^2)$
  -$\beta * x$: scalar-vector multiplication one nested loops $\mathcal{O}$(N)
  -$transpose(X) * (X * x) + \beta * x$: addition of two N vectors $\mathcal{O}$(N)
  Thus the total complexity is the most dominant: $\mathcal{O}(N^2)$.

- **OP1**
  Complexity:

  $\mathcal{O}(N^3)$

  Derivation:

  Here, we use the same implementation as for the NO-ET. The difference is that the expression templates are used. The expression templates will instead of calculating the operations directly, return an object of the type of operation to do (postpone the operations). Those objects will be cheap to create and copy in comparison to the complete result. OP1 will use a combination of different expression templates.
  - $X * x$ will use an operator of matrix-vector multiplication template found in vector_expressions.hpp. It creates an operator with only one nested loops ($\mathcal{O}(n)$) with res+=m_(i,j)*v_(j) inside, computing the ith element of the resulting vector. If we want to compute all the elements of the vector, this operator (object) should be called N times ($\mathcal{O}(N^2)$).

- The transpose expression is a matrix template found in matrix_expressions.hpp. The operator contain no nested loops and we just perform an inversion m(i,j). The complexity cost will only be realized when we perform an operation on this transposed matrix.
- $transpose(X) * (X * x)$ is a matrix-vector multiplication using the two last terms and will be the dominant operation here. The operations are nested. We will need to compute the dot product ($O(N)$) of each row of the transpose with the vector. For the ith element the matrix-vector multiplication is of complexity $O(N)$ and called N times ($O(N^2)$). When we call it in OP1 it will thus have a complexity of $\mathcal{O}(N^3)$.
- The $\beta * x$ is a scalar-vector multiplication template and can be found in vector_expressions.hpp. The operator contains doesn't need nested loops to have access to the ith row of the vector(O(1)). If we want to have access to all the elements of the vector, we will have to call this operator N times. When called in OP1 this will thus have time complexity of O(N).
- $transpose(X) * (X * x) + \beta * x$ are vector sum templates and can be found in vector_expressions.hpp. The operator contains no nested loops (O(1)) when we want to have access to the ith element. To have access to all the elements, the operator is called N times. When called in OP1 this will thus have time complexity of O(N).
The total time complexity when calling the assignment operator in OP1 is the most dominant and thus $\mathcal{O}(N^3)$.

- **OP2**
Complexity:

$\mathcal{O}(N^3)$

Derivation:

Here we stored the matrix-vector multiplication $X * x$ in a temporary vector t. There is no additional complexity by doing this, we are just storing the result differently. The complexity of $transpose(X) * (X * x)$ and $transpose(X) * (t)$ will be the same. The time complexity will thus be the same as for OP1.

- **OP3**
Complexity:

$\mathcal{O}(N^2)$

Derivation:

This time, we store the matrix-vector multiplication as $tws :: vector < Scalar > t = X * x$. The computation of $X * x$ is still of complexity $\mathcal{O}(N^2)$ and the storage (only be done once) will not add any complexity. When computing $transpose(X) * (t)$ we are in a matrix-vector operator and thus of complexity $O(N^2)$ (explanation already said above). The difference is that the operations are now sequential (and not nested) with both a complexity of $\mathcal{O}(N^2)$. The other operations will keep the same complexity. The final complexity will be the most dominant operation and thus $\mathcal{O}(N^2)$.

- **OP4**
Complexity:

$\mathcal{O}(N^2)$

Derivation:

The OP4 is a lambda function and will exploit the structure of the equation.
- The initialization imply a complexity of $\mathcal{O}(N)$ (size of the vector).
- We have an outer loop that iterates of N elements and contains:
- One inner loop with inner_Xix+=X(i,j)*x(j) which is an addition of products of two numbers ($\mathcal{O}(1)$) iterating of N elements.
- One inner loop with (j)+=X(i,j)*inner_Xix which is an addition of products of two numbers ($\mathcal{O}(1)$) iterating of N elements.
The complexity of both together is thus $\mathcal{O}(N^2)$.
- The final operation y+=beta*x is a scalar-vector product. It has already be defined above for the expression templates and has thus a time complexity of $\mathcal{O}(N)$.

The total asymptotic time complexity is the most dominant part and thus $\mathcal{O}(N^2)$.

Question 2

Compare these complexities with figure 1 from the assignment. Do these observations match your derived complexities? Why (not)? For the methods with equal complexities, can you explain why their timings (do not) differ? What are the performance (dis)advantages of expression templates? (maximum 20 lines)

First of all, we now that a $\mathcal{O}(N^2)$ complexity in log-log scale would appear as a straight line with a slope of 2 (time increases quadratically with the size of N). A $\mathcal{O}(N^3)$ complexity would appear as a straight line with a slope of 3 (time increases cubicly with the size of N). On the plot, we clearly see that OP1 and OP2 ($\mathcal{O}(N^3)$) has a steeper line than OP3, OP4 and NO-NET ($\mathcal{O}(N^2)$). In the big lines, the observations match thus our derived complexities.

However the $\mathcal{O}(N^2)$ methods differ in their timings even if they are supposed to have the same complexity. OP4 has a better timing than the two others, particularly when N becomes large. This is because it don't need to reassign memory for each vector. We see that OP3 and NO-NET begins to overlap when N becomes large. The optimization present in OP3 is only an advantage for a smaller N.
For a smaller N, the $\mathcal{O}(N^3)$ methods differ in their timings even if they are supposed to have the same complexity but when N becomes large, they overlap perfectly. We see that for a small N, they have the same timing as the methods of $O(N^2)$ complexities. However, due to the nested nature of those methods, the complexity will significantly increase when N becomes large.
In general, the differences in the timings when they have the same Big O complexity could be caused by hidden constants.

The principal advantages of expression templates is that they avoid to create temporary operations i.e. matrix-vector multiplications. This will avoid the creation of temporaries that are not necessary and thus can improve the performance (is able to perform more aggressive optimizations). Indeed, there is a difference in the number of allocations between a code using expression templates (less) and a code without (more). Expression templates can also make our code more modular, more readable due to a more mathematical style.
However, the disadvantage of expressions templates is the increased code complexity. It can also lead to a longer compilation time because it has to instantiate the templates and perform optimization.

Question 3

Indicate bellow what modifications you did you make to improve the memory usage of `xtx_op3`.

```
template<typename Xmat,typename Scalar>
class xtx_op3{

    private:
        Xmat X;
        Scalar beta;

    public:
    xtx_op3(Xmat const X_, Scalar beta_):X(X_),beta(beta_){
        assert(X.num_rows()>0);
        assert(X.num_columns()>0);
    }

    template<typename V>
    void operator()(V const & x, V & y) const {
        assert(X.num_columns()==y.size());
        assert(X.num_columns()==x.size());
        std::fill(y.begin(), y.end(), Scalar(0));
       y=X*x
       y=tws::transpose(X)*(y)+beta*x;
}
};
```

Briefly explain these modifications (maximum 5 lines).

As said in the assignment they are two small ways to improve the **memory** usage of OP3 without replacing it with OP4 (in the beginning I had a good implementation of one nested loops considerably reducing the timing). I will, in the place of creating a new temporary vector t, using an in-place operation. We know that y is already a vector, so we can perform the matrix-vector multiplication and directly store it in y. Thanks to that, our method will be more memory-efficient (and without nested loops).

Question 4

If you were writing a linear algebra library, how would you prevent or warn the user for the (potential) increased execution time encountered in implementations OP1 and OP2? (maximum 5 lines)

The problem for OP1 and OP2 is that, when N becomes too large, the nested loops (can be expressed in a tree) becomes too complex. We could thus inform the user, with a compile-time warning, when an expression tree becomes too complex.