C++ techniques for high performance financial modelling

Q. Liu
School of Management,
University of Electronic Science and Technology of China,
Chengdu, Sichuan, People's Republic of China

Abstract

In this paper, several C++ techniques, such as eliminating temporary objects, swapping vectors, utilizing the Matrix Template Library (MTL), and computing at compile-time, are shown to be highly effective when applied to the design of high performance financial models. Primarily, the idea emphasized is to achieve high performance numerical computations by delaying certain evaluations and eliminating many compiler-generated temporary objects. In addition, the unique features of the C++ language, namely function and class templates, are applied to move certain run-time testing into the compiling phase and to decrease the memory usage and speed up performance. As an example, those techniques are used in implementing finite difference methods for pricing convertible bonds; the resulted code turns out to be really efficient.
Keywords: C++, high performance, financial modelling, C++ template, Matrix Template Library, vector swapping, compile-time computation, convertible bond.

1 Introduction

What do Adobe Systems, Amazon.com, Bloomberg, Google, Microsoft Windows OS and Office applications, SAP's database, and Sun's HotSpot Java Virtual Machine have in common? They are all written in C++ (Stroustrup [1]). Still, when people talk about high performance numerical computations, Fortran seems to be the de facto standard language of choice.

To the author's knowledge, C++ is actually widely used by Wall Street financial houses; as an example Morgan Stanley is mentioned by Stroustrup [1] on his website. Techniques developed in the past few years, such as expression template (Furnish [2]) and compile-time computation or meta-arithmetic (Alexandrescu [3]), has made C++ a strong candidate for high performance numerical computations.

In this article I discuss four aspects of C++, namely trying to get rid of unnecessary temporary objects, swapping vectors for objects re-use, taking advantage of the performance gain provided by the Matrix Template Library [4], and doing compile-time computations, which are used in combination to achieve high performance numerical computation for financial modelling. Sample codes throughout this paper are taken directly from the library of a real-world convertible bond pricing model implementing finite difference methods.

2 Watching for temporary objects

C++ programs use quite a few temporary objects, many of which are not explicitly created by programmers (Stroustrup [5], Meyers [6], and Sutter [7]). Those temporary objects will drag down the performance tremendously if not eliminated. A few examples will make this point clear.

A typical step in the pricing process, or commonly known as diffusion on Wall Street, takes a list of stock prices and a list of bond prices, which are probably represented as vectors of doubles in C++ (or vector<double>) as in the following code (with some parameters omitted for simplicity), and returns a list of new bond prices:

```
typedef vector<double> VecDbl;

VecDbl diffuse(VecDbl stocks_in, VecDbl bonds_in) {
        VecDbl bonds_out;
        …
        return bonds_out;
}
```

What is wrong with this simple, innocent piece of code? Use too many unnecessary temporary objects!

Let’s analyze this carefully. First of all, the list of stock and bond prices are passed into the function by-value, as is commonly known in C++. When a function is called, a temporary copy of the object that passes by-value is created by the compiler. In the above code, two temporary objects, one for the list of stock prices and the other for the bond prices, are created (and then destroyed when the function returns). In a typical situation, the list of stock prices may have a length of a few hundred, so it is expensive (in terms of computing time) to create and destroy such a temporary object.

Further, inside the function, a local object of type vector<double> is used to store the values of the new bond prices temporarily. Finally, for the function to return a vector<double> object, one more object may have to be created by the copy constructor, if the function is used as in the following code:

```
        VecDbl my_vd;
        …
        my_vd = diffuse(stocks_in, bonds_in);
```

Note that the additional object created here could be eliminated by doing the function call and the object instantiation in a single step:

```
        VecDbl my_vd( diffuse(stocks_in, bonds_in) );
```

Therefore, depending on how the function is used, one may force C++ to construct yet another object! As a result, this simple function creates at least three unnecessary yet expensive objects, which can hardly be efficient.

To fix the problems in the code, we pass function arguments by-reference or by-pointer. Note that normally the list of stock prices is not changed through out the whole diffusion, but the prices of the bond are modified at every step (so the list of bond prices is used as both input and output as in the following):

```
void diffuse(const VecDbl & stocks_in, VecDbl & bonds_io) {
        VecDbl bonds_local; // for implicit finite difference method
        …
}
```

Because no temporary object needs to be created when function arguments are passed by-reference, no temporary object is created in the modified code above. Let's say that a typical diffusion takes about a thousand steps, so a total of about two thousand objects of vector<double> is eliminated by this simple modification!

For the explicit finite difference method (Hull [8]), even the local object inside the function can be eliminated by the following trick:

```
      while (iter != last) {                    // last == end() -1
              val_plus = *iter_next++;          // value of next element

              *iter++ = Up * val_plus + Mid * val + Down * val_minus;

              val_minus = val;                       // value of previous element
              val = val_plus;                        // value of current element
      }
```

Note that two iterators, one points to the current and the other to the next element, are used to keep track of the elements in the vector. Therefore, a further savings of a thousand objects is achieved.

To most C++ programmers, the above is probably obvious. C++ does have more subtle surprises for us in term of temporary objects. Look at the following standard piece of code seeing in many textbooks:

```
    for (int k = 0; k < N; k++) {…}
```

Could one see any problem?

Temporary object, of course! It may not be obvious, but the postfix increment operator actually creates an unused temporary object. Thus, the prefix increment operator shall be used here instead, which does not create a temporary object. The savings in this peculiar case is probably negligible, but any performance conscious coder should take home the point.

As a rule of thumb, prefix increment is preferred over postfix increment; unary operator, such as +=, is preferred over its binary counterpart, +, whenever possible. Those may not seem to be any big deal, but in order to achieve high performance numerical computation, one has to pay special attentions to those numerical operators. This point will become even more prominent in the following sections.

3 Re-using vector objects by swapping

Typically, a two-dimensional array of size 200x1000 (roughly the number of price points times the number of steps) for derivatives prices is used in finite difference methods (Clewlow and Strickland [9]). In another word, there is equivalently one individual vector<double> object for each step of diffusion. Normally we are only interested, however, in the final price slide at the valuation date. Therefore, is the two-dimensional array necessary?

Not at all. Since each step of diffusion involves only two neighbouring states, two vector<double> objects are actually enough:

```
    for (int step = 1; step <= 1000; ++step) {
            std::swap(bonds, prev_bonds);
            diffuseOneStep(…, prev_bonds, bonds);
    }
```

Note that by swapping and re-using the two objects, a two-dimensional array is no longer necessary. Swapping of two vector<double> objects can be very efficiently implemented (Stroustrup [5]). Not only the construction of almost a thousand more objects is avoided, but also the resource required for the code is much lighter (run-time resource for two objects instead of for a thousand objects).

4 Using the Matrix Template Library (MTL)

The Matrix Template Library is a free, high performance numerical C++ library maintained currently by the Open Systems Laboratory at Indiana University (MTL website [4]). MTL is based extensively on the modern idea of generic programming ([4] and Stroustrup [5]) and designed using the same approaches as the well-known Standard Template Library (STL). It is interesting to know that as MTL has demonstrated that “C++ can provide performance on par with

Fortran" [4], but it may still be surprising to some that "There are even some applications where the presence of higher-level abstractions can allow significantly higher performance than Fortran" [4].

As a library for linear algebra operations, MTL offers extensive algorithms and utility functions. Only one example of using MTL for financial modelling will be shown here to make the point, however. The following line of code is taken almost directly from the convertible bond model mentioned in the Introduction (with slight modifications to simplify the presentation):

```
mtl::add(mtl::scaled(mtl::scaled(stocks, cr), df), bonds);     //y += x
```

where cr and df are scalar variables, and the variables stocks and bonds are of type mtl::dense1D<double> (similar to vector<double>) as provided by MTL.

What the single line does is this: multiply every stock price in the vector by cr, then multiply the results by df, and finally add the results to bonds. Without using MTL algorithms, at least three loops would be necessary if the operators for addition, multiplication, and assignment were defined conventionally. This would be expensive, for it is well-known that it is optimal to perform more operations in one loop iteration (Dowd and Severance [10]). Further, more loops also mean many more temporary objects needed to be created to store the intermediate results of the arithmetic operations, which will slow down the computation even more (Furnish [2]). One could of course hand-code the one loop that does all the operations in one shot, but that misses the point here, since in so doing, which is ugly and error-prone, we lose the beauty of writing simple, arithmetic-like code.

MTL, however, does all the operations in one loop. Let's now see how MTL achieves this incredible feat. The function mtl::scaled prepares a multiplication of a vector by a scalar, but does not actually execute the multiplication. Then the result is scaled once more by another mtl::scaled. Again the multiplication is not executed. Finally mtl::add does two multiplications and one addition in one loop (for each element in the vector). Further note that the mtl::add here utilizes the unary operator += instead of the conventional binary operator + and then assignment operator; as a result, the temporary object needed by operator + is avoided.

5 Compile-time computation

Loosely speaking, compile-time computation is also known as static polymorphism, meta-programming, or meta-arithmetic, made possible by the C++ template mechanism. High performance is achieved by moving certain computation from run-time to compile-time, delaying certain computation or eliminating unnecessary temporary objects (Furnish [2] and Alexandrescu [3]). Further performance enhancement can be gained by coupling meta-programming with the C++ inline facility and the so-called lightweight object optimization.

What one can do with meta-programming is only limited by one's imagination, as Alexandrescu has aptly demonstrated in his excellent book [3]. Again one very simple example will be shown here just to make the point.

Convertible bonds are complicated financial contracts with many parameters. To pass all those parameters to the pricing code, a map with keys and values as strings are used. The values could actually be int, double, string, or some other types. The code has to convert all the values stored in strings to their proper type efficiently. How could this be done?

One could of course use a series of if-test's to determine the various types at run-time. That is not efficient, however. Or one could handle each value individually, but that is not elegant and error-prone. C++ meta-programming in fact enables us to do better and do something as the following:

```
        ReturnType val_lv; // ReturnType can be int, double, etc.
        findParam(key_in, params_in, val_lv);
```

Where given a return type, the program will choose the right function to use at compile-time. The findParam functions are explicitly defined for each possible return type as in the following fashion:

```
typedef map<string,string> StrPair;

template<class OutType>         // template function
void findParam(const string & key_, const StrPair & map_, OutType &
val_out ) {}

template<> void findParam(const string & key_, const StrPair & map_, int
& val_out ) {                   // specialize int type
        ParamFinderImpl<int>::findParam(stoi, key_,map_, val_out);
}

template<> void findParam(const string & key_, const StrPair & map_,
double & val_out ) {                // specialize double type
        ParamFinderImpl<double>::findParam(stof, key_,map_, val_out);
}
```

Note stoi converts a string to an int, while stof to a double. Here the template function specialization, or template<> (Stroustrup [5]), is utilized. Further, since the template parameter in findParam<int>, for example, can be deduced from the type of the relevant function argument, the <int> does not have to be specified when to be either defined or called.

As a result, the client code for using findParam is very simple and uniform. More importantly, since choosing the right version of findParam's is done at compile-time according to the return types specified by the client, the program is more efficient. Furthermore, some of the functions could be inlined to improve the performance additionally.

For completeness, the definition of the class ParamFinderImpl is shown below:

```
template<class OutType>
struct ParamFinderImpl {
        typedef bool (*p2f)( const string & s, OutType & val_out );

        static void findParam(p2f func_, const string & key_, const StrPair
& map_, OutType & val_out ) {
                StrPair::const_iterator pmi;

                if ((pmi = map_.find( key_ )) != map_.end() )
                        func_( pmi->second, val_out );
        }
};
```

6 Performance estimation

The convertible bond from Ayache et al.. [11] (see Table 1 below for details) is used in the performance test. The AFV model (Ayache et al. [11]) is implemented with the Crank-Nicolson method. The diffusion is done daily; in another word, there are 1826 time-steps in the diffusion. The state variable (stock or bond price) is divided into 281 points.

Table 1: Convertible bond data used in performance estimation.

Valuation date	01/01/2005 (mm/dd/yyyy)
Maturity	01/01/2010
Conversion ratio	1
Convertible	01/01/2005 to 01/01/2010
Call price	110
Callable	01/01/2007 to 01/01/2010
Call notice period	0
Put price	105
Putable	On 01/01/2008 (one day only)
Coupon rate	8%
Coupon frequency	Semi-annual
First coupon date	07/01/2005
Par	100
Hazard rate, p	0.02
Volatility	0.2
Recovery rate, R	0.0
Partial default	η=0.0
Risk-free interest, r	0.05

The C++ code is compiled using Microsoft Visual Studio .NET 2003 with optimization flag /O2. The program is executed on a Lenovo Laptop (240 MB memory and 1500 MHz Pentium Processor).

For ten runs, the main diffusion loop takes an average of 0.244 seconds to finish. Roughly speaking, four bonds could be priced in about one second, or two hundred bonds done in less than one minute. With such high speed, traders would be able to do portfolio-based optimization in real-time. This is believed to be quite efficient.

Acknowledgement

The work is supported in part by a National Natural Science Foundation of China grant (No. 70571012).

References

[1] Stroustrup, B., C++ Applications. public.research.att.com/~bs/applications.html
[2] Furnish, G., Disambiguated glommable expression templates. *Computers in Physics*, **11(3)**, pp. 263-269, 1997.
[3] Alexandrescu, A., *Modern C++ Design*, Addison-Wesley: Boston, 2001.
[4] MTL, The Matrix Template Library. www.osl.iu.edu/research/mtl/
[5] Stroustrup, B., *The C++ Programming Language*, Special ed., Addison-Wesley, 2000.
[6] Meyers, S., *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.
[7] Sutter, H., *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 2000.
[8] Hull, J. C., *Options, Futures, and Other Derivatives*, 5th ed., Prentice Hall: Upper Saddle River, New Jersey, 2003.
[9] Clewlow, L. & Strickland, C., *Implementing Derivatives Models*, John Wiley & Sons: New York, 1998.
[10] Dowd, K. & Severance, C. R., *High Performance Computing*, 2nd ed., O'Reilly & Associates: Cambridge, 1998.
[11] Ayache, E., P., Forsyth, A. & Vetzal, K. R., The valuation of convertible bonds with credit risk. *Journal of Derivatives* **11**, pp. 9-29, 2003.