CHAPTER 1
Effective C++

It is often said that quantitative analysts and developers should focus on algorithms and produce a readable, modular code and leave optimization to the compiler. It is a fact that substantial progress was made recently in the domain of compiler optimization, as demonstrated by the massive difference in speed for code compiled in release mode with optimizations turned on, compared to debug mode without the optimizations. It is also obviously true that within a constantly changing financial and regulatory environment, quantitative libraries must be written with clear, generic, loosely coupled, reusable code that is easy to read, debug, extend, and maintain. Finally, better code may produce a *linear* performance improvement while better algorithms increase speed by orders of magnitude. It is a classic result that 1D finite differences converge in ΔT^2 and ΔX^2 while Monte Carlo simulations converge in $\sqrt{N}$; hence FDM is preferable whenever possible. We will also demonstrate in Part III that AAD can produce thousands of derivative sensitivities for a given computation *in constant time*. No amount of code magic will ever match such performance. Even in Python, which is quite literally *hundreds of times* slower than C++, a good algorithm would beat a bad algorithm written in C++.

However, speed is so critical in finance that we cannot afford to overlook the low-level phenomena that affect the execution time of our al-

gorithms. Those low-level details, including memory cache, vector lanes, and multiple cores, do not affect algorithmic complexity or theoretical speed, but their impact on real-world performance may be very substantial.

A typical example is memory allocation. It is well known that allocations are expensive. We will repeatedly recall that as we progress through the publication and strive to preallocate at high level the memory required by the lower level algorithms in our code. This is not an optimization the compiler can conduct on our behalf. We must do that ourselves, and it is not always easy to do so and maintain a clean code. We will demonstrate some techniques when we deal with AAD in Part III. AAD records every mathematical operation, so with naive code, every single addition, subtraction, multiplication, or division would require an allocation. We will use custom *memory pools* to eliminate that overhead while preserving the clarity of the code.

Another expensive operation is *locking*. We lock *unsafe* parts of the code so they cannot be executed concurrently on different threads. We call *thread safe* such code that may be executed concurrently without trouble. All code is not always thread safe. The unsafe pieces are called *critical regions* and they may be *locked* (using primitives that we explore later in this part) so that only one thread can execute them at a time. But locking is expensive. Code should be thread safe *by design* and locks should be encapsulated in such a way that they don't produce unnecessary overhead. It is not only explicit locks we must worry about, but also hidden locks. For example, memory allocations involve locks. Therefore, all allocations, including the construction and copy of containers, must be banned from code meant for concurrent execution.[1] We will show some examples in Chapters 7 and 12 when we multi-thread our simulation library and preallocate all necessary memory beforehand.

Another important example is memory caches. The limited amount of memory located in CPU caches is orders of magnitude faster than

RAM. Interestingly perhaps, this limitation is not technical, but economical. We *could* produce RAM as fast as cache memory, but it would be too expensive for the PC and workstation markets. We may envision a future where this ultra-fast memory may be produced for a reasonable cost, and CPU caches would no longer be necessary. In the meantime, we must remember caches when we code. CPU caches are a hardware optimization based on a locality assumption, whereby when data is accessed in memory, the same data, or some data stored nearby in memory, is likely to be accessed next. So, every access in memory causes a duplication of the nearby memory in the cache for faster subsequent access. For this reason, code that operates on data stored nearby in memory – or *coalescent* – runs substantially faster. In the context of AAD, this translates into a better performance with a large number of small tapes than a small number of large tapes,[2] despite "administrative" costs per tape. This realization leads us to differentiate simulations path-wise, and, more generally, systematically rely on *checkpointing*, a technique that differentiates algorithms one small piece at a time over short tapes. This is all explained in detail in Part III.

For now, we make our point more concrete with the extended example of an elementary matrix product. We need a simplistic matrix class, which we develop as a wrapper over an STL vector. Matrices are most often implemented this way, for example, in Numerical Recipes [20].

```
template <class T>
class matrix
{
        //  Dimensions
        size_t      myRows;
        size_t      myCols;

        //  Data
        vector<T>   myVector;

public:

        using value_type = T;

        //  Constructors
        matrix() : myRows(0), myCols(0) {}
        matrix(const size_t rows, const size_t cols)
            : myRows(rows), myCols(cols), myVector(rows*cols) {}

        //  Access
        size_t rows() const { return myRows; }
        size_t cols() const { return myCols; }
        //  So we can call matrix [i][j]
        T* operator[] (const size_t row)
            { return &myVector[row*myCols]; }
        const T* operator[] (const size_t row) const
            { return &myVector[row*myCols]; }
};
```

We test a naive matrix product code that sequentially computes the result cells as the dot product of each row vector on the left matrix with the corresponding column vector on the right matrix. Such code is a direct translation of matrix algebra and we saw it implemented in a vast number of financial libraries.

```
1  inline void matrixProductNaive(
2  const matrix<double>& a,
3  const matrix<double>& b,
4  matrix<double>& c)
5  {
6      const size_t rows = a.rows(), cols = b.cols(), n = a.cols();
7
8      //  Outermost loop on result rows
9      for (size_t i = 0; i < rows; ++i)
10     {
11         const auto ai = a[i];
12         auto ci = c[i];
13
14         //  Loop on result columns
15         for (size_t j = 0; j < cols; ++j)
16         {
17             //  Innermost loop for dot product
18             double res = 0.0;
19             for (size_t k = 0; k < n; ++k)
20             {
21                 res += ai[k] * b[k][j];
22             }
23             //  Set result
24             c[i][j] = res;
25         }
26     }
27 }
```

This code is compiled on Visual Studio 2017 in release 64 bits mode, with all optimizations on. Note that the following settings must be set on the project's properties page, tab “C/C++”:

- “Code Generation / Enable Enhanced Instruction Set” must be set to “Advanced Vector Extensions 2” to produce AVX2 code.
- “Language / OpenMP Support” must be set to “yes” so we can use OpenMP pragmas.

For two random 1,000 × 1,000 matrices, it takes around 1.25 seconds to complete the computation on our iMac Pro. Looking into the inner-most loop, we locate the code on line 21, executed 1 billion times:

```
            res += ai[k] * b[k][j];
```

One apparent bottleneck is that $b[k][j]$ resolves into $(\&b.myVector[k * b.myCols])[j]$. The multiplication $k * b.myCols$, conducted a billion times, is unnecessary and may be

replaced by an order of magnitude faster addition, at the cost of a somewhat ugly code, replacing the lines 17–22 by:

```
            //  Dot product
            double res = 0.0;
            const double* bkj = &b[0][j];
            size_t r = b.rows();
            for (size_t k = 0; k < n; ++k)
            {
                res += ai[k] * *bkj;
                bkj += r;
            }
```

And the result is *still* 1.25 second! The compiler was already making that optimization and we polluted the code unnecessarily. So far, the theory that optimization is best left to compilers holds. We revert the unnecessary modification. But let's see what is going on with memory in that innermost loop.

The loop iterates on k and each iteration reads $ai[k]$ and $b[k][j]$ (for a fixed j). Data storage on the $matrix$ class is row major, so successive $ai[k]$ are localized in memory next to each other. But the successive $b[k][j]$ are distant by 1,000 doubles (8,000 bytes). As mentioned earlier, CPU caches are based on locality: every time memory is accessed that is not already duplicated in the cache, that memory *and the cache line around it, generally 64 bytes, or 8 doubles*, are transferred into the cache. Therefore, the access to a is cache efficient, but the access to b is not. For every $b[k][j]$ read in memory, the line around it is unnecessarily transferred into the cache. On the next iteration, $b[k+1][j]$, localized 8,000 bytes away, is read. It is obviously not in the cache; hence, it is transferred along with its line again. Such unnecessary transfer may even erase from the cache some data needed for forthcoming calculations, like parts of a. So the code is not efficient, not because the number of mathematical operations is too large, but because it uses the cache inefficiently.

To remedy that, we modify the order of the loops so that the innermost loop iterates over coalescent memory for both matrices:

```
1  inline void matrixProductSmartNoVec(
2  const matrix<double>& a,
3  const matrix<double>& b,
4  matrix<double>& c)
5  {
6      const size_t rows = a.rows(), cols = b.cols(), n = a.cols();
7
8      //  zero result first
9      for (size_t i = 0; i < rows; ++i)
10     {
11         auto ci = c[i];
12         for (size_t j = 0; j < cols; ++j)
13         {
14             ci[j] = 0;
15         }
16     }
17
18     //  Loop on result rows as before
19     for (size_t i = 0; i < rows; ++i)
20     {
21         const auto ai = a[i];
22         auto ci = c[i];
23
24         //  Then loop not on result columns but on dot product
25         for (size_t k = 0; k < n; ++k)
26         {
27             const auto bk = b[k];
28             //  We still jump when reading memory,
29             //      but not in the innermost loop
30             const auto aik = ai[k];
31
32             //  And finally loop over columns in innermost loop
33             //      without vectorization to isolate impact of cache alone
34             #pragma loop(no_vector)
35             for (size_t j = 0; j < cols; ++j)
36             {
37                 //  No more jumping through memory
38                 ci[j] += aik * bk[j];
39             }
40         }
41     }
42 }
```

The pragma on line 34 will be explained ahead.

This code produces the exact same result as before, in 550 milliseconds, more than twice as fast! And we conducted just the same amount of operations. The only difference is cache efficiency. To modify the order of the loops is an operation too complex for the compiler to make for us. It is something we must do ourselves.

It is remarkable and maybe surprising how much cache efficiency matters. We increased the speed more than twice just changing the order of the loops. Modern CPUs operate a lot faster than RAM so our

software is *memory bound*, meaning CPUs spend most of their time waiting on memory, unless the useful memory is cached in the limited amount of ultra-fast memory that sits on the CPU. When we understand this and structure our code accordingly, our calculations complete substantially faster.

And we are not quite done there yet.

What does this "#pragma loop(no_vector)" on line 34 stand for? We introduced SIMD (Single Instruction Multiple Data) in the Introduction. SIMD only works when the exact same instructions are applied to multiple data stored side by side in memory. The naive matrix product code could not apply SIMD because the data for b was not coalescent. This was corrected in the smart code, so the innermost loop may now be vectorized. We wanted to measure the impact of cache efficiency alone, so we disabled SIMD with the pragma "#pragma loop(no_vector)" over the innermost loop.

Visual Studio, like other modern compilers, *auto-vectorizes* (innermost[3]) loops whenever it believes it may do so safely and efficiently. If we remove the pragma but leave the code otherwise unchanged, the compiler should auto-vectorize the innermost loop[4]. Effectively, removing the pragma further accelerates calculation by 60%, down to 350 milliseconds. The SIMD improvement is very significant, if somewhat short of the theoretical acceleration. We note that the innermost loop is a *reduction*, having explained in the Introduction how parallel reductions work and why they struggle to achieve parallel efficiency. In addition, data must be *aligned* in memory in a special way to fully benefit from AVX2 vectorization, something that we did not implement, this being specialized code, outside of the scope of this text.

What this teaches us is that we must be SIMD aware. SIMD is applied in Visual Studio outside of our control, but we can check which loops the compiler effectively vectorized by adding "/Qvec-report:2" (with-

out the quotes) in the “Configuration Properties/ C/C++ / Command Line/ Additional Options” box of the project's properties. We should strive to code innermost loops in such a way as to encourage the compiler to vectorize them and then check that it is effectively the case at compile time.[5] To fail to do so may produce code that runs at half of its potential speed or less.

Altogether, to change the order of the loops accelerated the computation by a factor 3.5, from 1250 to 350 milliseconds. Over half is due to cache efficiency, and the rest is due to vectorization.

Always try to structure calculation code so that innermost loops sequentially access coalescent data. Do not hesitate to modify the order of the loops to make that happen. This simple manipulation accelerated our matrix product by a factor close to 4, similar to multi-threading over a quad core CPU.

Finally, we may easily distribute the *outermost* loop over the available CPU cores with another simple pragma above line 19:

```
// OpenMp directive: execute loop in parallel
#pragma omp parallel for
for (int i = 0; i < rows; ++i)
{
    const auto ai = a[i];
    auto ci = c[i];

    for (size_t k = 0; k < n; ++k)
    {
        const auto bk = b[k];
        const auto aik = ai[k];

        for (size_t j = 0; j < cols; ++j)
        {
            ci[j] += aik * bk[j];
        }
    }
}
```

This pragma is an OpenMP directive that instructs the compiler to multi-thread the loop below it over the available cores on the machine running the program. Note that we changed the type of the outermost

counter i from $size\_t$ to int so OpenMP would accept to multi-thread the loop. OpenMP's auto-parallelizer is somewhat peculiar this way, not unlike Visual Studio's auto-vectorizer.

This code produces the exact same result in just 40 milliseconds, approximately 8.125 times faster, more than our number (8) of cores! This is due to a so-called “hyper-threading” technology developed by Intel for their recent chips, consisting of two *hardware threads* per core that allow each core to switch between threads at hardware level while waiting for memory access. The OS effectively “sees” 16 hardware threads, as may be checked on the Windows Task Manager, and their scheduling over the 8 physical cores is handled on chip. Depending on the context, hyper-threading may increase calculation speed by up to 20%. In other cases, it may *decrease* performance due to excessive *context switches* on a physical core when execution switches between two threads working with separate regions of memory.

Contrary to SIMD, multi-threading is not automatic; it is controlled by the developer. In very simple cases, it can be done with simple pragmas over loops as demonstrated here. But even in these cases, this is not fully satisfactory. The code is multi-threaded at compile time, which means that it always runs concurrently. But users may want to control that behavior. For instance, when the matrix product is part of a program that is itself multi-threaded at a higher level, it is unnecessary, and indeed decreases performance, to run it concurrently. Concurrency is best controlled at run time. Besides, to multi-thread complex, structured code like Monte-Carlo simulations, we need more control than OpenMP offers. In very simple contexts, however, OpenMP provides a particularly light, easy, and effective solution.

Altogether, with successive modifications of our naive matrix product code, but without any change to the algorithm, its complexity, or the mathematical operations involved, we increased the execution speed by a factor of 30, from 1,250 to 40 milliseconds. Obviously, the results

are unchanged. We achieved this very remarkable speed-up by tweaking our code to take full advantage of our modern hardware, including on-chip cache, SIMD, and multiple cores. It is our responsibility to know these things and to develop code that leverages them to their full potential. The compiler will not do that for us on its own.

NOTES

1 Intel, among others, offers a concurrent lock-free allocator with its freely available Threading Building Blocks (TBB) library. Without recourse to third-party libraries, we must structure our code to avoid concurrent allocations altogether.

2 The *tape* is the data structure that records all mathematical operations.

3 Evidently, given SIMD constraints, innermost loops are the only candidates for vectorization.

4 Provided the setting “C/C++ / Code Generation / Floating Point Model” is manually set to Fast in the project properties page for the release configuration on the relevant platform, presumably x64. When this is not the case, Visual Studio does not vectorize reductions. Whether the reduction was vectorized or not can be checked by writing “/Qvec-report:2” in the “Additional Options” box in the “C/C++ / Command Line” setting.

5 Visual Studio is somewhat parsimonious in its auto-vectorization and frequently declines to vectorize perfectly vectorizable loops (although it would never vectorize a loop that should not be vectorized). Therefore we must check that our loops are effectively vectorized and, if not, rewrite them until such time the compiler finally accepts to apply SIMD. This may be a frustrating process. STL algorithms are generally easier auto-vectorized than hand-crafted loops, yet another reason to prefer these systematically.