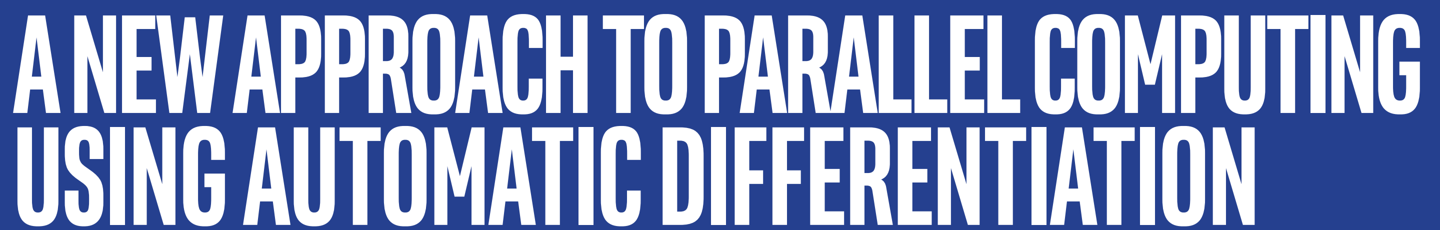


A NEW APPROACH TO PARALLEL COMPUTING USING AUTOMATIC DIFFERENTIATION

Getting Top Performance on Modern Multicore Systems

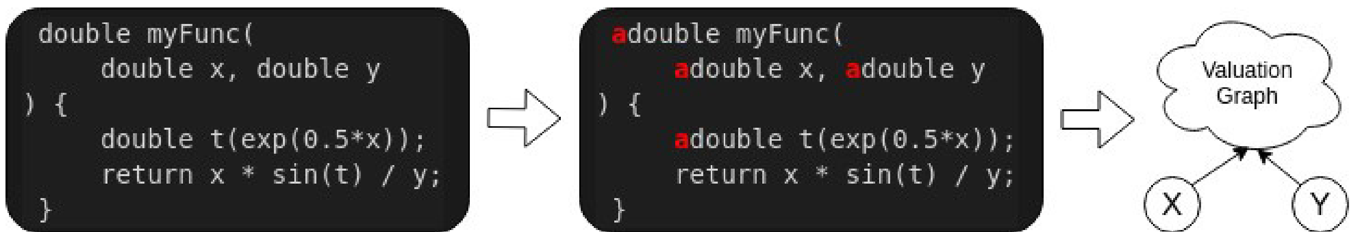
Dmitri Goloubentsev, Head of Automatic Adjoint Differentiation, Matlogica, and Evgeny Lakshtanov, Principal Researcher, Department of Mathematics, University of Aveiro, Portugal and Matlogica LTD

If you're interested in high-performance computing, high-level, object-oriented languages aren't the first things that come to mind. Object abstractions come with a runtime penalty and are often difficult for compilers to vectorize. Adapting your code for multithreading execution is a huge challenge, and the resulting code is often a headache to maintain.

You're in luck if performance-critical parts of your code are localized and can be flattened and safely parallelized. However, many performance-critical problems can benefit from object-oriented programming abstractions. We're proposing a different programming model that lets you achieve top performance on single instruction multiple data (SIMD), non-uniform memory access (NUMA) multicore systems.

Operator Overloading for Valuation Graph Extraction

We'll focus on problems where the same function, `F(1)`, needs to be executed on a data set `X[i]`. For example, let's look at Monte Carlo simulations in the finance world where `X[i]` is a random sample and `F(.)` is a pricing function (**Figure 1**). We use an **operator overloading pattern** to extract all primitive operations performed by `F(.)`.



1 **Example operator overloading pattern**

This pattern is very common in **automatic adjoint differentiation (AAD)** libraries. Unlike traditional AAD libraries, we don't build a data structure to represent the valuation graph. Instead, we compile binary machine code instructions to replicate valuations as defined in the graph, which can be seen as a just-in-time (JIT) compilation. However, we don't work with the source code directly. Instead, we compile a valuation graph produced by the user's algorithm. Since we want to apply `F(.)` to a large set of data points, we can compile this code to expand all scalar operations to full SIMD vector operations and process four (AVX2) or eight (AVX-512) data samples in parallel.

Learn by Example

Let's look at a simple option pricing framework where we use various abstract business objects. In this example, we simulate asset values as a random process:

```
template<class vtype>
vtype simulateAssetOneStep(
   const vtype current_value
   , Time current_t
   , Time next_t
   , const BankRate<vtype>& rate
   , const AssetVolatility<vtype>& vol_obj
   , const vtype& random_sample
) {
   double dt = (next_t-current_t);
   vtype vol=vol_obj(current_value, current_t);
   vtype next_value = current_value * (
       1 + (-vol*vol / 2+ rate(current_t))*dt
       +  vol * std::sqrt(dt) * random_sample
   );
   return next_value;
}
```

The classes `BankRate` and `AssetVolatility` can define different ways of computing model parameters, and implementation can be done deep in the derived classes. This function can be used with the native `double` type. When applied along timepoints, `t[i]` can be used to simulate asset value at the option expiry:

```
template<class vtype>
vtype onePathPricing (
   vtype asset
   , double strike
   , const std::vector<Time>& t
   , const BankRate<vtype>& rate_obj
   , const AssetVolatility<vtype>& vol_obj
   , const std::vector<vtype>& random_samples
) {
   for (int t_i = 0; t_i < t.size()-1; ++t_i) {
       asset = simulateAssetOneStep(asset, t[t_i], t[t_i+1], rate_obj,
vol_obj, random_samples[t_i]);
   }
   return std::max(asset - strike, 0.);
}
```

However, this leads to bad performance because the compiler can't effectively vectorize the code and business objects may contain virtual function calls. Using the AAD runtime compiler, we can execute the function, record one random path of asset evolution, and compute option intrinsic value at the expiry:

```
typedef __m256d mmType;  // __mm256d and __mm512d are supported

   int AVXsize = sizeof(mmType) / sizeof(double);

   aadc::AADCFunctions<mmType> aad_funcs;

   std::vector<idouble> random_samples(num_time_steps, 0.);
   idouble rate(0.03), vol(0.15), asset(100.0);
   aadc::VectorArg random_arg;

   aad_funcs.startRecording();
       // Mark vector of random variables as input only
       markVectorAsInput(random_arg, random_samples, false);

       // Mark rate, initial asset value and volatility as inputs
       aadc::AADCArgument rate_arg(rate.markAsInput());
       aadc::AADCArgument asset_arg(asset.markAsInput());
       aadc::AADCArgument vol_arg(vol.markAsInput());

       BankRate<idouble> rate_obj(rate);
       AssetVolatility<idouble> vol_obj(vol);

       idouble payoff= onePathPricing(asset, strike, t, rate_obj, vol_obj, random_samples);
       aadc::AADCResult payoff_arg(payoff.markAsOutput());
   aad_funcs.stopRecording();
```

At this stage, the `func` object contains compiled, vectorized machine code that replicates valuations to produce the final payoff output value given the arbitrary `random_samples` vector as input. The function object remains constant after recording and requires memory context for execution:

```
 // allocate memory needed for vectorized execution
shared_ptr<WorkSpace<mmType> > ws = func.createWorkSpace();

mmType mm_total_price(mmSetConst<mmType>(0.0));

// initialize function inputs
ws->val(rate_arg) = mmSetConst<mmType>(init_rate);
ws->val(asset_arg) = mmSetConst<mmType>(init_asset);
ws->val(vol_arg) = mmSetConst<mmType>(init_vol);

// run Monte Carlo loop
for (int mc_i = 0; mc_i < (num_mc_paths / AVXsize); ++mc_i) {
    vector<mmType> avx_random_samples(generateRandomAvxVector());
    // Set function arguments
    ws->setVector(random_samples_arg, avx_random_samples);

    // call the recorded function
    func.forward(ws);
    // get result
   mm_total_price=mmAdd(ws->val(payoff_arg), mm_total_price);
}
// calc avx vector element wise sum
cout << mmSum(mm_total_price) << endl;
```

Free Multithreading

Making efficient and safe multithreaded code can be difficult. Notice that the recording happens only for one input sample and can be executed in the controlled, stable, single-threaded environment. The resulting recorded function, however, is threadsafe and only needs separate workspace memory allocated for each thread. This is a very attractive property, since it lets us turn non-multi-thread-safe code into something that can be safely executed on multicore systems. Even optimal NUMA memory allocation becomes a trivial task. (You can view the full code listing for the multithreaded example **here**.)

Automatic Differentiation

This technique not only accelerates your function, it can also create an adjoint function to compute derivatives of all inputs with respect to all outputs. This is similar to the back-propagation algorithm used for deep neural network (DNN) training. Unlike DNN training libraries, this approach works for almost any arbitrary C++ code. To record an adjoint function, simply mark which input variables are required for differentiation:

```
// compute derivative w.r.t. initial value
AADC::Argument asset_arg(asset.markAsDiff());
```

Finally, to execute the adjoint function, initialize the gradient values of outputs and call the `reverse()` method on the function object:

```
ws->diff(payoff_arg) = mmSetConst<mmType>(1.0);

func.reverse(ws);

cout << "Derivative of dPrice/dSpot = " << ws->diff(asset_arg)[0] << endl;
```

Getting Top Performance

Hardware is evolving toward increasing parallelism with a lot more cores, wider vector registers, and accelerators. For object-oriented programmers, it's hard to adapt single-threaded code to existing parallel methods like OpenMP and CUDA. Using the AADC tool from Matlogica, programmers can turn their object-oriented, single-threaded, scalar code into AVX2/AVX512 vectorized, multithreaded, and threadsafe lambda functions. Crucially, the AADC tool can also generate a lambda function for the Adjoint method of computing, with all required derivatives using the same interface. Visit **Matlogica** for more details and a demo version of AAD-C.

Acknowledgements

Evgeny Lakshtanov is partially supported by Portuguese funds through the Center for Research and Development in Mathematics and Applications (CIDMA) and the Portuguese Foundation for Science and Technology (FCT, Fundação para a Ciência e a Tecnologia), within project UIDP/04106/2020.