MATLOGICA

Open Source Benchmark demonstrating a leap in performance for valuation and AAD risk using AADC on Intel Scalable Xeon CPUs

Introduction

In this paper we will present a benchmark that has been developed to demonstrate how real world problems can be solved using AADC library and top performance can be achieved using Intel Scalable Xeon CPU Cascade Lake. The benchmark is purposely implemented in plain C++ for ease of understanding and minimization of run-time overheads due to object abstractions. The code that represents client's analytics is designed and implemented without any upfront intentions for multi-threading or vectorization. The result of this work demonstrates that applying Matlogica's technique to this code base allows achieving top performance of modern Xeon CPUs. Previously to get comparable performance users would need to do extensive rework to port existing analytics to GPU. Moreover, we demonstrate that computing sensitivities using Automatic Differentiation method is also very efficient and problem size isn't constrained by available memory.

Benchmark description and configuration

We have used simple and efficient analytics implemented as scalar single-threaded C++ which computes so-called Valuation Adjustments(xVA) that are computationally difficult and required by the financial industry for daily operation. We chose to focus on financial applications, however this technique is not limited to any specific domain and can be applied across multitude of problems.

This benchmark is configured to simulate one interest rate curve using Hull-White model, three projection curves and two credit curves: one for company and one for counterparty.

For portfolio we randomly generate 100 50y interest rate swaps with random future start dates.

Using underlying pricing models and portfolio level CSA rules we calculate CVA and DVA scalar outputs. Interest rate curves are built using 250 interpolation points and credit curves are buld using 140 interpolation points, for which we produce sensitivity analysis risks totaling 2564 risk outputs.

Hardware configuration

Cascade lake CLX-8280M with total 56 cores (112 threads)
Memory 192GB
MatLogica's AADC library (5/15 release)
Compiler ICC 19.1.1

Valuation acceleration

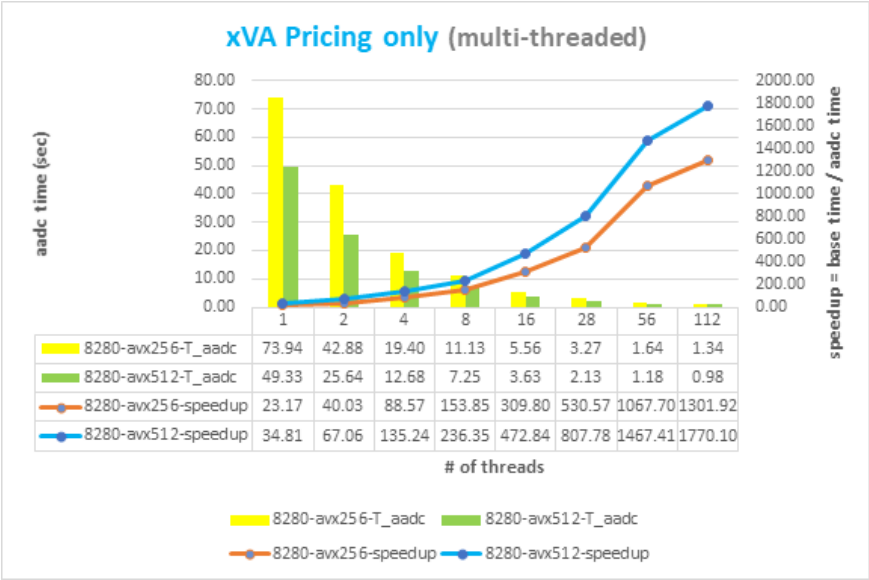
In provided source code, the user's analytics is assumed to be implemented by xVAProblem class(*) in single-thread mode. It uses a template type for all real values and can be instantiated using native double type as well as active type idouble(*). Computations which use native double instance set performance baseline of the benchmark with the baseline execution time measurements taken using a single CPU core and relying on the compiler (Intel c++) to do all vectorization work.

Using operator overloading technique with the "idouble" active type on xVAProblem class allows AADC to extract valuation graph and form binary instructions that replicate user analytics at run-time.

Applying AADC to this analytics is a straightforward process. We want to record a AADC function for one MonteCarlo path. This includes evolution of all market objects, pricing of all trades at all future time points and computation of CVA/DVA integrals. Input variables for this function should be the normal random variables needed to generate one path.

Since we need to process a large set of MonteCarlo samples, we can automatically create a AADC function to simultaneously handle 8(or 4) samples in vector form using native avx512(or avx2) CPU registers and vector op-code instructions. The function memory use is tightly controlled and can be allocated per thread,hence multiple threads can process subsets of MonteCarlo paths safely in parallel. As a result, we have safely parallelized analytics that was't originally designed for multi-thread execution.

The graphs below present the speed up of Monte-Carlo valuation loop achieved by applying AADC and using 1-56 threads on Intel Xeon using avx512 and avx2 instruction sets.



	1	2	4	8	16	28	56	112
8280-avx256-T_aadc	73.94	42.88	19.40	11.13	5.56	3.27	1.64	1.34
8280-avx512-T_aadc	49.33	25.64	12.68	7.25	3.63	2.13	1.18	0.98
8280-avx256-speedup	23.17	40.03	88.57	153.85	309.80	530.57	1067.70	1301.92
8280-avx512-speedup	34.81	67.06	135.24	236.35	472.84	807.78	1467.41	1770.10

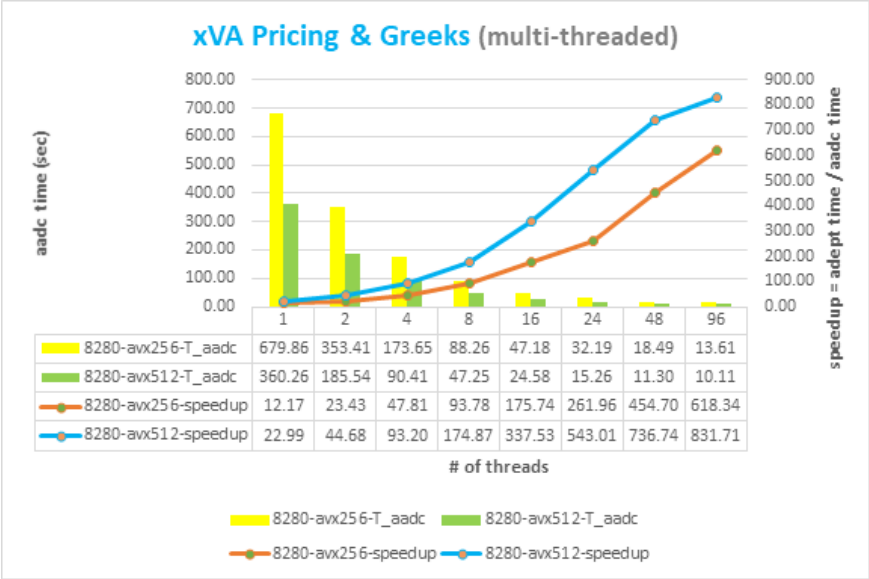
Valuation and AAD Risk acceleration

In this section we will demonstrate how AADC can be used to accelerate valuations and computation of sensitivities using Automatic Adjoint Differentiation. We will showcase the performance of the AADC library against state-of-art AAD libraries and not the AAD method itself. For this reason, we define our baseline as the time that takes for the open source AAD Adept library to calculate all AAD sensitivities using the original user analytics and a single CPU core. Again running the original analytics and adept in multi-thread mode would require careful design and programming, so for the baseline we resort to executing it in a single-threaded mode only.

The same operator overloading pattern with active "idouble" instance of the xVAProblem class can be used to form machine code instructions for the adjoint function(*) that calculates sensitivities to the specified input variables. We mark all model parameters and points on the market curves as variables we want to calculate sensitivities for using the ".markAsDiff()" method. This allows AADC to only form instructions to variables where sensitivities are actually needed even though these variables are considered to be "constant" from the MonteCarlo path valuation point of view. This is a very powerful optimization and can be configured at runtime for a specific set of required risks. As before, only stochastic normal input variables driving the MonteCarlo simulation actually change from one path to the next.

In order to produce full jacobian for CVA and DVA output variables and all model inputs, we need to execute 2 passes of the adjoint function. So each MonteCarlo path contains one valuation and two adjoint valuations. Averaging over all simulations give us the xVA values and their path-wise sensitivities(*).

The sensitivity calculation can also take advantage of vectorization and multi-threading in the same manner as in valuation only case. This allows for very efficient distribution of MonteCarlo valuations and AAD risks for multi-core, multi-socket systems and full vectorization.



	1	2	4	8	16	24	48	96
8280-avx256-T_aadc	679.86	353.41	173.65	88.26	47.18	32.19	18.49	13.61
8280-avx512-T_aadc	360.26	185.54	90.41	47.25	24.58	15.26	11.30	10.11
8280-avx256-speedup	12.17	23.43	47.81	93.78	175.74	261.96	454.70	618.34
8280-avx512-speedup	22.99	44.68	93.20	174.87	337.53	543.01	736.74	831.71

Conclusion

In this paper we demonstrated how MatLogica AADC library can be used to achieve breakthrough performance improvement for a computationally expensive XVA valuation and risk valuation model on Intel AVX-512. We have used an approachable example in the finance sector, as well as a relatively simple C++ code to showcase ease of integration of MatLogica AADC with an existing

code base, introducing a novel way not only to improve performance but to simplify development and support of computationally extensive analytics.

We showed that by using MatLogica AADC, the benefits of Intel AVX-512 can be realised fully and top performance results can be achieved due to our patent pending proprietary way to drive CPU execution flow.

For this benchmark, we focused on a simple C++ example, however this technique has been proved to work with large projects such as the popular open source QuantLib library in quantitative finance. This solution can be also extended to other industries with a need for high computational performance on problems that are usually associated with GPU

Please contact MatLogica if you're interested in our proposition or would like to see a live demonstration.

Glossary

XVA https://en.wikipedia.org/wiki/XVA

CSA https://en.wikipedia.org/wiki/Credit_Support_Annex

AVX512 https://en.wikipedia.org/wiki/AVX-512

AAD https://en.wikipedia.org/wiki/Automatic_differentiation#Reverse_accumulation

Operator Overloading https://en.wikipedia.org/wiki/Automatic_differentiation#Operator_overloading_(OO)

Adept https://en.wikipedia.org/wiki/Adept_(C%2B%2B_library)

Further reading on AAD

Community Portal for Automatic Differentiation http://www.autodiff.org

Capriotti, L. (2011). Fast Greeks by algorithmic differentiation. The Journal of Computational Finance, 14(3), 3.

Savine, A., (2020). Computation graphs for aad and machine learning part iii: application to derivatives risk sensitivities Wilmott, vol. 2020, iss. 106, p. 24–39, 2020.

Matlogica, (2019). AAD: Breaking the Primal Barrier. Wilmott, 2019(103), 8-11.

sales@matlogica.com