AAD: Breaking the Primal Barrier

Dmitri Goloubentsev[*], Evgeny Lakshtanov[†]

Here, we present a new approach for Automatic Adjoint Differentiation with a special focus on computations where derivatives $\frac{\partial F(X)}{\partial X}$ are required for multiple instances of vectors X. In practice, the presented approach is able to calculate all the differentials faster than the primal (original) C++ program for F. Major application areas are

- Gradient methods for optimisation problems, including global model calibration, speech recognition, deblurring of images and machine learning in general
- Derivatives of mathematical expectation
- Path-wise sensitivities of SDE

Code Transformation vs Operator Overloading Currently, two main approaches are used for the Automatic AD tools:

Code Transformation (CT): Analises the computer program which implements function F to produce a code of the AD method.

Operator Overloading (OO): All mathematical operations are overloaded in such a way that the information about a computational graph of F is saved in the data structure called Tape[1]. Tape is used afterwards to process the backward pass of the AD method.

There are rather succesful CT AAD tools, however, they limit the available language features and make the build system more complex. The need for such a tool is further reflected in the fact that there is currently no CT AAD available for C++. The OO approach usually demonstrates weaker speed performance due to a runtime overhead in each iteration. Let us enter into more details at this point.

Consider a schematic processing of multiple samples X_i using a standard OO AAD library:

```
Loop  i=0..N-1
  BeginTapeRecording();
  Y[i] = F(X[i]);
  StopTapeRecording();
  dX[i] = Reverse(dY[i], tape);
Next  i;
```

Each overloaded operator collects information on valuations, after which the second backwards pass updates the adjoint variables. This approach comes with a number of disadvantages:

- OO or/and tape interpretation runtime overhead for each iteration
- Multithreading is only possible if the original function F(X[i]) is thread-safe
- Relying on compiler for CPU vectorization of a scalar primal function means that vectorization is used very sparsely
- $F()$ may perform unnecessary operations that don't depend on X, and yet are executed at each iteration (e.g. mathematical operations, virtual function calls, dictionary lookups etc)
- Additional memory is required to store the Tape data-structure. Its size is proportional to the number of operations of the primal function F, which may be prohibitive in some cases.

[*]MathLogic LTD, London, UK., dmitri@matlogica.com

[†]CIDMA, Department of Mathematics, University of Aveiro, Aveiro 3810-193, Portugal and MathLogic LTD, London, UK., lakshtanov@matlogica.com

[1]its choice depends on a concrete AAD tool

OO×CT Our innovative idea is to cross[2] both approaches, namely to use the Overloaded Operators to auto-generate AD-version of the primal function at run-time. The created AD-functions can be used for different X_i instead of performing classic OO AAD approach on each $F(X_i)$.

During the first run of the original function, every overloaded function - or operator - will generate instructions for forward and reverse AD-pass. For instance, consider the function $f(a, b, c) = a * b + c$. The first column of the following table lists the consecutive calls of atomic valuations during the execution of the function $f(a, b, c)$.

Valuation	→ Forward()	→ Reverse()
Initialization	v0=a; v1=b;	
Initialization	v2=c;	
operator *	v3=v0*v1;	d1+ = d3 * v0; d0 += d3 * v1;
operator +	v4=v3+v2;	d2 += d4; d3 += d4;
Initialization	f=v4;	
Initialization		d0=0; d1=0;
Initialization		d2=0; d3=0;

Concatenated entries of the second column form the body of the Forward pass, while the body of the Reverse pass is formed by the entries in the third column in reverse order:

```
void AD_Function
   (double a,b,c,&f,&d0,&d1,&d2,d4) {
double v3,d3(0);
v0=a; v1=b; v2=c;
v3=v0*v1;
v4=v3+v2;
f=v4;
// Reverse
d2 += d4;
d3 += d4;
d0 += d3 * v1;
d1 += d3 * v0;
}
```

Now all differentials of the function f and its value at the point (a, b, c) can be computed by calling

```
double d0(0),d1(0),d2(0),d4(1),f;
AD_function(a,b,c,f,d0,d1,d2,d4)
```

Within the proposed framework, the OO is used for only one sample X, after which the resulting program can process multiple input data. Let us list the immediate benefits of this approach:

- Unlike the classic OO AAD, the AD-function does not change from one iteration to the next one. Hence there is no any OO or tape interpretation run-time overhead per X_i sample.
- The AD-function is completely segregated from the user program. All user data are encapsulated within the AD-function, and its memory state is limited to vectors v and d. Thus, multiple samples of X can be processed safely in the multithreaded mode.
- The AD-function can be generated to consistently utilize native CPU vectorization to process 4(8)-double chunks of user data (AVX2 \ AVX512 speed up x4-x8).
- That can lead to a final acceleration of order $8 \times$ #Cores compared to the modern AAD tools.
- Highly efficient, i.e. operations that don't depend on X are not included in the constructed AD-function.
- Although additional memory is also required to store the AD-function, its code remains static and can be shared between CPU cores.

Combining all the stated benefits one can achieve a fantastic performance of 0.4 at one core, comparing to the original program implemented using a standard *double* arithmetics. For anybody who wishes to play around we prepared a "prototype" C++ implementation available at `www.matlogica.com` free of charge. This prototype implementation is simplified and cannot work effectively with large computations. In addition to a two-pass compilation, the relative compilation

[2]Not to be confused with AAD of the *mixed* type, where the tool does an analysis first and then takes the decision on what approach should be used in each particular situation

time and the volume of memory involved are unacceptable for practical use[3]. Part of the reason for this is that the size of AAD functions is proportional to the tape i.e. to the linearized (unfolded) version of the primal algorithm. Below we discuss how we can address these drawbacks.

A prototype C++ implementation The code of the prototype library is short and self explanatory. Nevertheless, we found it reasonable to provide some comments in case it needs further clarificaiton. According to the canonical AD, all variables can be distinguished by their roles as inputs, intermediates, or outputs. The header file NaiveAADLib.h defines the active class *dagdouble* which transforms each *double* variable into a node of a future calculation Directed Acyclic Graph.

```
class dagdouble {
public:
    dagdouble() {}
    dagdouble(const double& val,
              bool isInput = false)
        : val(val) {
        indx = getNextVarCounter();
        if (!isInput) {
            aadAssignConst(indx, val);
        } else {
            inputIndex.insert(indx);
        }
    }
    dagdouble(const dagdouble& other) :
            val(other.val) {
        indx = getNextVarCounter();
        aadAssign(indx, other.indx);
    }
     dagdouble& operator =
           (const dagdouble& other) {
        val = other.val;
        indx = getNextVarCounter();

        aadAssign(indx, other.indx);
        return *this;
    }
    void markAsOutput() {
        outputIndex.insert(indx);
    }

    double val;
    int indx;
};
```

As an example, we supplied an overloaded version of the multiplication operator. Here, we provide only the scalar version of the code. For the vector version, the reader is encouraged to consult the Prototype library source code.

```
dagdouble operator *(const dagdouble& a,
                     const dagdouble& b) {
    dagdouble res;
    res.val = a.val * b.val;
    res.indx = getNextVarCounter();
    aadMult(res.indx, a.indx, b.indx);
    return res;
}

void aadMult(int res_indx, int a_indx,
             int b_indx) {
    stringstream fstr, rstr;
    if (codeVersion == ScalarCode) {
        fstr << "v" << res_indx << "=" << "v"
           << a_indx << "*v" << b_indx << ";";
        rstr << "d" << a_indx << "+=" << "d"
           << res_indx << "*v" << b_indx << ";"
           << "d" << b_indx << "+=" << "d"
           << res_indx << "*v" << a_indx << ";";
    }
    aad_func_fwd.push_back(fstr.str());
    aad_func_rev.push_back(rstr.str());
}
```

It creates a new variable (getNextVarCounter()) and writes the corresponding code of the Forward and Reverse passes (applying aadMult()).

The distribution contains a basic example main.cpp where the library's functioning is carried out by defining a preprocessor variable #define USE_GENRATED_AAD_FUNCTION. Depending on its value, the user program either generates a file which contains a newly created AD function, or uses the earlier generated one.

The included synthetic benchmark test (example.cpp) produce the following results:

clang	Absolute time	Relative factor
AVX2 Primal	83ms	1
AVX2 Adjoint	64ms	0.73
AVX512 Primal	84ms	1
AVX512 Adjoint	39ms	0.46

All of the aforementioned disadvantages of the prototype approach can be addressed by generating a binary code directly from OO. This idea was implemented in AAD-Compiler by MathLogic LTD.

A JIT AAD-Compiler is a professional version of the prototype library. It represents a completely enabled OO AAD library and offers the following features:

- Optimized machine binary code generation optimized for both run-time performance and quick code generation (patent pending)
- Streaming Compilation (patent pending)

[3]See the relative compilation time benchmarks for prototype and professional AAD-Compilers

- Incremental Checkpointing
- Proprietary AD-Code-Folding Compression
- Support for multiple platforms and C++ compilers
- Support for AVX2 and AVX512 vectorization (patent pending)
- Enables multithreaded valuations even when underlying user program isn't multithread safe (patent pending)

The AAD-Compiler has been extensively tested and documented. Licensing terms distribution can be found at `www.matlogica.com`.

AAD-Compiler speed-benchmark results The benchmark results for the AAD-Compiler are based on tests of different nature, including random synthetic tests, previously published tests such as LW, Toon or GMM, as well as the standard Financial models like LMM or Stochastic Volatility Calibration model [1],[2],[4].

The first test we consider is based on the open source benchmark from the article F.Srajer et al. (2018). Authors test various AAD-tools for the Gaussian Mixture Model with 2.5M iterations and get the following results (absolute time of adjoints in seconds):

# variables		5.36e+4	4.29e+5
Manual		3.89e+2	6.16e+3
Finite diff.			
Adept	C++	4.09e+3	3.99e+4
ADOLC	C++	1.04e+4	
Ceres	C++		
Tapenade	C	1.32e+3	1.59e+4
DiffSharp	F#		
MuPAD	Matlab	•	•
Julia-F	Julia		
Julia-F (vect)	Julia	•	•
Autograd	Python		
Theano	Python	•	•
Theano (vect)	Python	•	•

Authors note that "The bullet symbolizes that a tool crashed and no entry means that a tool did not finish in the time limit. Only tools that could compute at least one problem instance are shown."

We execute the Adept's and AAD-Compiler's versions of the GMM with 429 thousands arguments and 2.5M iterations and get the following absolute time of adjoints (in seconds). The AAD-Compiler has optionally activated Code compressor, so both cases are presented here.

Tool	Compressor	AVX2	AVX512
Adept	-	2.89e+4	N/A
AADC	On	1.3e+3	1.17e+3
AADC	Off	2.86e+3	2.86e+3

Note that the huge number of required differentials (0.5M) makes the GMM a memory-bound problem and memory bandwidth is almost maxed out at AVX2. For smaller sized problems performance scales well with the AVX vector size.

AAD-Compiler Relative time of adjoints for real world models The following table demonstrates relative time of the full gradient to the execution time of a primal algorithm.

Performance coeff:	AVX2	AVX512
Toon	0.25	0.2
Heston	0.37	0.22
LMM	0.35	0.21

Toon benchmark is assumed to run over multiple input data.

The behaviour of the performance relative time on the number of CPU cores can be found in the following table:

LMM	1 Core	4 Cores	8 Cores
AVX2	0.35	0.095	0.05
AVX512	0.21	0.06	0.035

AAD-Compiler Relative compilation time for various real world and toy models From the following table one can conclude that compilation time is equivalent to around 400 executions of the primal algorithm.

Model	Relative Compilation time
Heston	970
LMM	650
GMM	700
Toon	240

AAD-Compiler memory-benchmark re-

sults

For memory benchmarks we also used the conventional Lax&Wendroff and Toon tests from [1].

Tool	Comp.	LW	Toon	GMM
Adept		3mb	24.5mb	23mb
AADC	ON	42kb	11.5mb	7mb
AADC	OFF	16mb	59.5mb	15mb

AADC's values are normalized to a single sample. To obtain the real memory consumption, one needs to multiply the values by an AVX vector length.

Forward function. The Forward function is useful on its own and constitutes a vectorized replication of the primal algorithm. Similarly to the complete AD function, it is MT safe even if the primal algorithm is not. This replication proves extremely useful if one wishes to use Finite Differences, or just accelerate any complex computations. Given the current trend in using multiple CPU cores to accelerate computations, this feature is useful in its own right.

References

[1] Hogan, R. J. 2014. Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4), 26.

[2] Srajer, F., Kukelova, Z. and Fitzgibbon, A. 2018. A benchmark of selected algorithmic differentiation tools on some problems in computer vision and machine learning. *Optimization Methods and Software*, 33(4-6), 889-906.

[3] AAD-Compiler prototype library. `www.github.com`

[4] Goloubentcev, D. and Lakshtanov, E. 2019. Remarks on stochastic automatic adjoint differentiation and financial models calibration. arXiv preprint arXiv:1901.04200.

[5] AAD-Compiler by MathLogics LTD. `www.matlogica.com`