Meta Adjoint Programming in C++

Klaus Leppkes[a]
Johannes Lotz[a]
Uwe Naumann[a]
Jacques du Toit[b]

[a] Software and Tools for Computational Engineering, RWTH Aachen
[b] Numerical Algorithms Group Ltd., Oxford, UK

Department of Computer Science
Technical Report

Aachener Informatik-Berichte (AIB) | ISSN 0935-3232 | AIB-2017-07
RWTH Aachen University | Department of Computer Science | September 2017

The publications of the Department of Computer Science of *RWTH Aachen University* are in general accessible through the World Wide Web.

`http://aib.informatik.rwth-aachen.de/`

Meta Adjoint Programming in C++

Klaus Leppkes[a]
Johannes Lotz[a]
Uwe Naumann[a]
Jacques du Toit[b]

[a] Software and Tools for Computational Engineering
RWTH Aachen, Germany
lotz@stce.rwth-aachen.de

[b] Numerical Algorithms Group Ltd.
Oxford, UK
jacques@nag.co.uk

Abstract. Adjoint code development for many-core architectures like GPUs is a major challenge. To address issues arising from the parallelization, currently hand-writing the adjoint code is the most promising approach. Unfortunatly, hand-writing has the fundamental drawback of maintainability, i.e. diverging primal and adjoint codes. To overcome this, supporting tools can and should be used. In this report, we present a new approach for generating adjoint C++ codes by algorithmic differentiation. The approach is called meta adjoint programming and based on the template metaprogramming mechanism. We show that with new developments in the C++ language standard, operator and function overloading techniques can be used to achieve efficient and maintainable adjoint codes. Results are shown for a GPU parallel CVA code. An implementation of meta adjoint programming called `dco/map` is presented and comparisons with hand-written adjoint code are carried out.

1 Overview

This report introduces a new approach to compute adjoints of C++ code using algorithmic differentiation (AD): Meta adjoint programming (MAP). AD [GW08,Nau12] is a semantic program transformation technique that generates for a given code (an implemented mathematical function; the *primal*) a corresponding *adjoint* code. The adjoint code computes the algorithmic adjoint projection of the primal, i.e. for a primal that computes

$$y = F(x),\ \text{where } F : \mathbb{R}^n \rightarrow \mathbb{R}^m \tag{1}$$

with inputs $x \in \mathbb{R}^n$ and outputs $y \in \mathbb{R}^m$, its adjoint computes

$$\bar{x} = [\nabla_x F]^T \cdot \bar{y} \tag{2}$$

with output adjoints $\bar{y} \in I\!R^m$, inputs adjoints $\bar{x} \in I\!R^n$, and $\nabla_x F$ the Jacobian matrix. When using AD, above given projection is calculated implicitly, i.e. without accumulating the full Jacobian, transposing and multiplying it. This would result in high computational costs. Instead, when using AD, the cost of one adjoint projection is a constant multiple of the cost of the primal, i.e.

$$\text{cost(adjoint)} = \mathcal{R} \cdot \text{cost(primal)} \ . \tag{3}$$

This property is especially helpful in case of many inputs ($n \gg 1$) and very few outputs (e.g. $m = 1$), e.g. in an optimization context where gradients of a cost function are required. Adjoints are used in a wide range of different applications like, e.g. computational fluid dynamics [GPGP97,GB02], optimal control [HT12], or computational finance [GG06]. In this report we are targeting an application from computational finance, a basic credit value adjustment (CVA) computation which has production code scale.

Many of those applications already are computationally very expensive and therefore not uncommonly run on massively parallel architectures. To achieve a reasonable adjoint run time, it is therefore highly desirable to minimize the overhead of an adjoint, i.e. the run time factor $\mathcal{R}$. AD tools try to achieve that for quite a while for sequential codes. But codes using multi-core CPUs and in particular many-core architectures like GPUs introduce additional synchronization problems in the adjoint code [För14]. Multi-core architectures (shared or distributed memory systems, i.e. OpenMP [1] or MPI [2]) are also targeted by a some tools more recently, e.g. [UHH+09], but many-core architectures are usually even more challenging [GPGK08,GHR+16]. In addition, for large codes, a clean software development process is crucial. How MAP resolves those issues is shown in this report.

In Section 2 we first want to present state-of-the-art AD tool handling of parallel architectures followed by a brief introduction to AD by operator overloading on the level of computational graphs. After a description of the application we are targeting in Section 3, MAP is introduced in the form of a short userguide of dco/map in Section 4. In Section 5 we show run time and applicability results.

[1] `www.openmp.org`
[2] e.g. `www.open-mpi.org`

2 Adjoint AD Tools and Parallelism

AD tools are usually categorized by two main approaches, *AD by overloading* and *AD by source transformation*. Whatever approach is taken, an adjoint code requires a data flow reversal of the primal. This is done in two phases, the forward run and the reverse run. During the forward run, data is stored that is used for the reversal realized in the reverse run. Source transformation tools usually generate forward code that stores data on a stack which is used by a generated reverse code. Overloading tools on the other hand usually store the full data flow at run time on a so-called *tape*. The tape is a data structure representing the computational graph. Once recorded, the tape can be used for the reversal (called *interpretation*) independently of the original code. Main advantage of operator overloading tools is a straightforward applicability especially in C++ as well as the property of always having the up-to-date adjoint for a given primal (i.e. primal and adjoint are not running out-of-sync). Unfortunately, parallelism does not fit well into the record-interpret-strategy since the tape is by default a sequential data structure. In addition, the tape consumes a huge amount of memory, which is a scarce resource on many-core architectures. Source transformed adjoint codes are expected to generate more efficient adjoint code, run time-wise as well as memory-wise. This is achieved by the reverse run being baked into the code segment, where compiler-based optimization can better be applied. Recapturing the primal parallelism is usually easier in source-transformed code by hand in a post-processing step. The following approaches can be used for getting a parallel adjoint code:

1. *Using finite differences (bumping) instead of adjoints.*
   A practical approach is to just not use adjoints and avoid the sequential nature of it by calculating the gradient with finite differences. Finite differences, of course, do have major shortcoming (e.g. accuracy), but also important advantages. Computing individual gradient entries is embarrassingly parallel, i.e. parallelization is trivial. In addition, discontinuities are smoothed implicitly. Nonetheless, the computational complexity grows with the number of input parameters, which results in high financial costs for hardware or compute-time.
2. *Hand-writing adjoints.*
   Writing an adjoint by hand is a time consuming and thus expensive task. The code can of course exploit all possible bits and pieces to get an efficient, parallel adjoint. In addition to the high personnel expenses, from a software development perspective, hand-writing the

adjoint is tedious, since it needs to be adapted to every change in the primal. Otherwise primal and adjoint diverge resulting in wrong values.

3. *Parallel taping with overloading.*
   Tape-based AD by overloading tools can be used to compute adjoints in parallel with various approaches. The potentially most promising one is based on parallel taping, i.e. in case of shared memory (OpenMP) per-thread, and in case of distributed memory (MPI) per-process. This way the recording step inherits the parallelism from the primal. Interpretation still requires manual effort to tackle the data races. In addition, the amount of memory required for the tape is expected to be still a major problem.
4. *Source transformation.*
   Source transformation tools can be used to support the hand-writing of parallel adjoint codes. Since no tool exists that automatically generates parallel adjoint code, the synchronization needs to be done by hand. As already mentioned, source transformation tools generate more efficient adjoint code; on the other hand are diverging primal and adjoint codes again problematic from the software development standpoint.

MAP differs from the above given approaches in the following essential points. MAP works with overloading, but

1. it doesn't record a tape
   we follow the source transformation approach of recomputation instead
2. it creates adjoint code at compile time
   using template metaprogramming, the adjoint code is generated by the C++ compiler
3. it supports data types that implicitly handle synchronization
   since overloading is used, such functionality can be hidden from the user

In the next section, we introduce the showcase application.

3 Application: CVA Computation

The sub-prime crisis of 2007/2008 and subsequent liquidity crisis revealed that financial practitioners had at best a somewhat naive notion of credit exposure. It was difficult to say how an institution's balance sheet would change if one of its counterparties defaulted.

One of the ways regulators have sought to address this problem is by requiring banks to calculate and report new exposure figures. One of these is the Credit Value Adjustment (CVA). Roughly speaking, this is the expected amount that an institution will lose due to potential defaults of a given counterparty. If we ignore discounting, then the formula for CVA is

$$\mathrm{CVA} = (1 - R) \int_0^T \mathbb{E}[E(\tau) \,|\, \tau = t\,] \, P_{dt} \tag{4}$$

where τ is the (random) default time of the counterparty, P_t is the probability that the counterparty defaults before time t, E(t) is the exposure to the counterparty at time t, $\mathbb{E}$ is conditional expectation under the so-called risk neutral measure, and R is the recovery rate.

Computing CVA is a time-consuming and compute-intensive task. For two large institutions, the exposure $E(t)$ may depend on hundreds of thousands or millions of individual contracts, the so-called netting set, each of which have to be valued separately so that the final exposure can be computed. In addition, many of these contracts will be for derivative instruments, the valuation of which requires simulation or numerical solution of PDEs.

The simplest case which is often considered in the literature is that of a portfolio of interest rate swaps. These are simple derivatives where payments are made depending on the value of an observed market rate such as LIBOR or EURIBOR. In this case computing CVA (by Monte Carlo) requires choosing a model for the joint evolution of LIBOR and P_t (so that one can capture so-called wrong way risk), generating many independent evolutions of the LIBOR rate and P_t over $[0, T]$, then at each time t valuing the portfolio of swaps to compute the exposure $E(t)$, and finally computing the integral above and averaging over all evolutions. For a large swap portfolio this calculation can become expensive, especially since swaps can be very long dated, meaning that T is on the order of 30 to 40 years.

A key regulatory requirement is that banks hedge their CVA exposure. Hedging CVA requires computing at least the gradient of the CVA with respect to the inputs of the model. For a swap portfolio the inputs will typically be various interest and credit curves observed in the market. Since these curves are quite long dated and are made up of many discrete instruments, the number of inputs to the CVA model can be large. Hedging CVA in theory requires knowing the full gradient. This makes adjoint

AD an attractive option, and banks have started to adopt it into their production CVA systems.

In order to test our new approach in this setting we created a ”prototype” CVA code. The code is based on [Pfa15]. The netting set consists of a single swap and the interest rate market is described by the two factor G2++ model [HW94]. A third factor is added for the default process, and the correlation between the three factors gives rise to wrong way risk. In the finance literature such a three factor model would be one of the more sophisticated approaches to CVA. For the sake of simplicity the code does not actually compute the full CVA, but it comes very close (adding two loops and some book keeping is all that is needed). From a code point of view the key features of this model are the lengthy and expensive closed form formulae for computing $E(t)$. The formulae contain lots of exponentials and some double exponentials, combined into various rational expressions. There are 10 ”read only” input arrays which form 5 linear interpolating splines. These splines are evaluated several times on each call to $E(t)$.

4 Meta Adjoint Programming

As already mentioned earlier, meta adjoint programming (MAP) resolves a couple of hard problems faced by a progammer when implementing an adjoint GPU code in C++. After a description of the main benefits of MAP, this section shows the interface of our implementation (`dco/map`) and a couple of examples.

The following problems can be tackled by using MAP:

1. *Primal and adjoint code diverge.*
   When hand-writing adjoint code or generating it with source transformation tools, the underlying problem of primal and adjoint code do not naturally coincide. The primal development cycle may be faster and the adjoint code lags behind – either because of missing manpower (hand-writing) or because of missing features in the source transformation tool. MAP resolves this issue since it uses operator overloading techniques. The same code is instantiated (in terms of C++ templates) with passive data type (e.g. double) and active data type. When instantiated with active data type, adjoints can be computed. MAP should of course feature *perfect decay*, i.e. when instantiating the code with passive data types, the code should be as efficient as an unaltered original code. This is the case with `dco/map`.

2. *AD by overloading requires too much memory.*
   Classical overloading approaches write a so-called *tape*, which stores the full computational graph of the executed code. The tape therefore grows as fast as floating point operations are executed. This usually fills the available memory withit seconds of program execution, because of which various techniques need to be applied to resolve this (e.g. checkpointing or external adjoints). Instead of such a store-all strategy, MAP follows a recompute-all strategy (similar to source transformation). The memory footprint is therefore much smaller. In addition, the adjoint memory is directly linked in a one-to-one relationship to primal memory. This decreases the required memory for the overall adjoint computation roughly to twice the original primal memory.
3. *Hand-written adjoints are hard to write but more efficient.*
   When hand-writing adjoint code, a well-versed programmer can optimize the code in many ways, e.g. the data layout, exploit common subexpressions in primal and adjoint or use the implicit function theorem to avoid backward iterative processes. Through the use of modern C++ and expression template mechanism, at least for straight-line code MAP can approach similar performance as hand-writing or source transformation (see results in Section 5). In addition, interfaces are possible to support the user in incorporating more efficient computations in the default MAP behavior. This can be done under conservation of the prefect decay property.
4. *Parallelism is problematic with AD by overloading.*
   A general problem with adjoint code is the property that primal parallel reads become adjoint parallel writes, i.e. additional data races are introduced. In addition memory access is a big issue. To get efficient code, coalesced memory access is crucial, since otherwise waiting on data is unnecessarily wasting clock cycles. This is especially true when using GPUs, where thousands of cores access memory simultaneously. Storing and restoring the tape under consideration of those two issues is hard, if not impossible to achieve. Since MAP is in many respects more similar to source transformation than to overloading (i.e. no tape, adjoint memory next to primal memory), those issues can be addressed with different synchronization and data layout configurations.

4.1 Parallelism

The two main problems that need to be addressed are:

1. Shared-memory parallelism reduces the available amount of memory per thread. A memory efficient adjoint is therefore crucial. In addition, especially on GPUs, the memory hierarchy and associated access times make custom memory management potentially very slow. Best is therefore to keep as much data in registers (very fast non-addressable memory) as possible. This can be achieved by giving the compiler as many hints as possible at compile time.
2. The adjoint computation introduces additional data races in case of parallelism. Wherever the primal computation has non-exclusive read access (i.e. multiple threads read shared data), the adjoint will have parallel write access. Different synchronization strategies using different data layouts are possible.

The first problem is addressed by the data layout of the base data type as well as the template expression mechanism. As long as the primal code is written in such a way that the compiler can efficiently place the variables in registers, that should be possible for the adjoint as well. Reasons are, first, that adjoint memory is located directly next to its primal memory, and second, that the adjoint code is fully known at compile time due to template expression use. This is underpinned by the measurements in Section 5.

The second problem requires additional data types that take care of those data races in the background. In particular, those data types need to decay to usual passive data (e.g. `double`) in case of a passive computation. This is obvious, since additional synchronization is only required in active computations. The remaining section describes solution approaches to the second problem. After discussing scalar variables and their synchronization, different approaches to vector-valued variables (arrays) are addressed.

Let's assume the following code is executed in parallel with different threads. The input `x` is shared among all threads, while each thread gets a separate memory location for `yi`.

```
template <typename T>
void f(const T& x, T& yi) {
  yi = sin(x); // if T=double
               //   passive computation only
               // if T=type<double>
               //   in addition adjoint propagation at assignment,
               //   i.e. x.adjoint += cos(x.value)*yi.adjoint
}
```

For primal computation only (e.g. instantiated with `T=double`), the code is free of data races. The active computation on the other hand has shared write access to the adjoint of `x`. To get correct results, synchronization is required. For synchronization we introduce the *connector type* (`connector_t`). This type holds local adjoint memory as well as a reference to the original data. At destruction, it performs the increment on the original data in a thread safe way, i.e. an atomic increment (e.g. in OpenMP using the `#pragma omp atomic` statement). The resulting code then looks like:

```
template <typename T>
void f(const T& x, T& yi) {
  connector_t<T> x_loc(x); // if T=double => connector_t<T>=double
                           // if T=type<double>: save reference
                           //                    to x in x_loc
  yi = sin(x_loc); // perform adjoint propagation
                   //   x_loc.adjoint += cos(x_loc.value)*yi.adjoint
} // ~x_loc:
  //   atomic increment: x.adjoint += x_loc.adjoint
```

This approach works correctly but is not the most efficient way, since atomic increments may occur very frequently. An alternative would be to first accumulate in per-thread memory followed by a reduction at the very end. This is especially required for GPU computations, since atomic operations are slow compared to reductions.

To show the approach using reductions we need to introduce arrays. Though it is possible to use `std::vector<type<double> >`, it is not well suited for reductions, since value and adjoint are interleaved in memory. A data layout is required where value and adjoint memory is separated from each other. In addition, the adjoint memory is to be extended to hold separate memory for each thread, see Fig. 1 for a graphical representation of those three layouts:

- `std::vector<type<double> >`
  Value and adjoint component are interleaved. Meta data is defined by the standard, e.g. the array length. When accessing an element with the bracket operator (e.g. `x[i]`), a reference to `type<double>` is returned. Synchronization when updating adjoints need to be done by wrapping the returned type into a connector type.
- `array_t<type<double>, atomic_push>`
  In this array value and adjoint memory is separated. Meta data is the array length. When accessing an element with the bracket operator, a connector type is returned. On destruction the corresponding adjoint is incremented atomically.
- `array_t<type<double>, reduction_push>`
  This array can be created from the previous one. Therefore value and adjoint memory is not only separated (it holds pointer to the

remote value and adjoint memory). The adjoint memory is also extended to have additionally one memory location for each element and each thread. When accessing an element with the bracket operator, a different connector type is returned with references to thread-local memory. Here no atomic update is required. In the array's destructor, a reduction takes place and it atomically increments the remote memory. Under the assumption that enough memory is available, this is the most efficient approach (see Section 5). The meta data section in addition holds the number of threads.

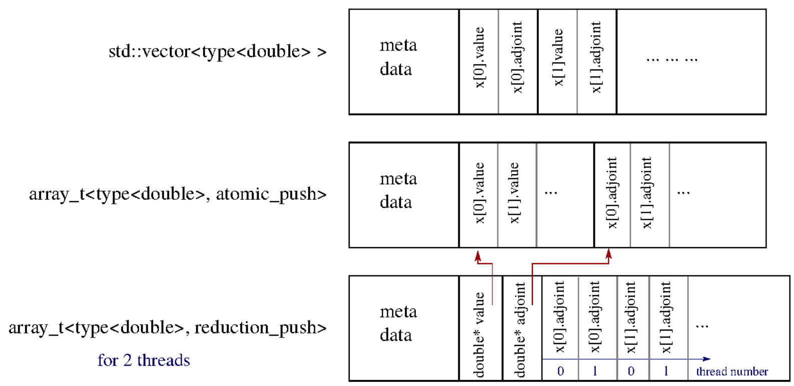

Fig. 1. Different data layouts for the array data structure called `x`. Meta data is e.g. the number of threads or the length of the array.

4.2 Implementation: `dco/map`

`dco/map` is an implementation of meta adjoint programming. In addition to the basic data type and its functionality required for computing adjoints of straight-line code, `dco/map` supports a set of control flow structs, as well as a user interface as simple as possible; though a fundamental code refactoring is still required for incorporating `dco/map` into an existing code base. The interface for accessing internal data (i.e. set value or derivative component) as well as data type names are as close as possible to `dco/c++` [LLN16]. The interplay of both packages is also possible but not covered in this report. In the following, we show small examples to see the use of `dco/map`. The control flow functionality is wrapped in C macros, thus hidden. As you will see, all macros take the currently used floating point data type as first argument.

All examples in the following do include `dco_map.hpp` and can use the typedef `using type = dco_map::ga1s<double>::type`.

- *Straight-line code*

  The following example shows use of the basic data type

```
dco_map::ga1s<double>::type
```

  The template parameter (here: `double`) is the type of value and adjoint component. A temporary variable `t` is declared as `const auto` and has an expression type.

```
  type x(2.0), y; // active data
  dco_map::derivative(y) = 1.0; // set output adjoint to 1.0

  {
    const auto t = x*x;
    y = sin(t);
  }

  printf("adjoint␣of␣x␣=␣%f\n", dco_map::derivative(x));
```

  Output is "`adjoint of x = -2.614574`".

- *Branches*

  If/else-branches can be implemented using macro definitions `MAP_IF`, `MAP_ELSE`, and `MAP_IF_END` as follows.

```
  type x(2.0), y;
  dco_map::derivative(y) = 1.0;
  {
    type t;
    MAP_IF(type, x <= 2.0) { // MAP_IF takes in addition
                             // the condition
      t = x*x;
    } MAP_ELSE {
      t = sin(x);
    } MAP_IF_END;
    y = sin(t);
  }
  printf("adjoint␣of␣x␣=␣%f\n", dco_map::derivative(x));
```

  Output is again (since the condition evaluates to `true`)

  `adjoint of x = -2.614574`

- *Loops*

  Loops can be implemented using the macro definitions `MAP_FOR` and `MAP_FOR_END`. The following example computes a sum over a simple function of the input and the loop index. In `dco/map`, overwriting variables needs special attention, because values potentially need to be restored or recomputed in later use. Since in the following example, `sum` is not used nonlinearly before overwriting it, no special treatment is required.

```
int n = 2;
type x(2.0), y;
dco_map::derivative(y) = 1.0;
{
  type sum = 0.0;
  MAP_FOR(type, i, 0, n-1, 1) { // MAP_FOR takes in addition
                                // the loop index variable
                                // name, the start index, the
                                // end index, and the increment
    const auto t = x*x;
    sum += sin((i+1)*t);
  } MAP_FOR_END;
  y = sin(sum);
}
printf("adjoint␣of␣x␣=␣%f\n", dco_map::derivative(x));
```

Output is "`adjoint of x = -3.676858`".

– *Function Calls*

Function calls can be implemented using the macro definition `MAP_CALL`. Given a function definition

```
template <typename T> void f(const T& x, T& y) { y = x*x; }
```

The caller can be implemented as follows.

```
type x(2.0), y;
dco_map::derivative(y) = 1.0;
{
  type t;
  MAP_CALL(type, f(x, t)); // MAP_CALL just takes the
                           // function call
  y = sin(t);
}
printf("adjoint␣of␣x␣=␣%f\n", dco_map::derivative(x));
```

Output is again "`adjoint of x = -2.614574`".

5 Results

As results, we want to show first that the meta adjoint programming technique and its corresponding implementation we propose is efficient. This is done by measuring and comparing run times required for computing algorithmic adjoints of a test code using different approaches. But second, potentially even more important, we want to show applicability of MAP to the application described in Section 3 under conservation of the efficiency. This is in particular important for the GPU.

Since the programming techniques used for `dco/map` are part of C++11 and still quite recent, run times may vary even more over compiler manufacturer, its version, and hardware architecture (CPU/GPU) as usual. We show a couple of measurements for different combinations of the mentioned parameters on a Linux system to present our findings. In the benchmarks, for comparisons, we use `dco/c++` as plain overloading tool and TAPENADE as source transformation tool.

5.1 Benchmarking Straight-Line Code

The test code we use as benchmark was kindly provided by D. Bommes from a test case of the software package CoMISo[3] [BZK12]. It is released as a benchmark package, which can be downloaded via github[4]. The code is coming from a method for constructing a 3D cross-frame field, a 3D extension of the 2D cross-frame field as applied to surfaces in applications such as quadrangulation and texture synthesis [HTWB11]. The code itself is a straight-line code with approximately 80 lines. It is generated by Maple$^{\text{TM}}$ [MGH$^+$05] with common subexpression optimization switched on. A special characteristic is that the expressions on the right-hand side of assignments are quite long, on average ~ 20 operations.

The following hardware / software configuration is used to perform the measurements. As CPU, we use the `Intel i7-6700K @ 4.00GHz` and as GPU the `NVIDIA Quadro M4000`. For the compilation of GPU device code, we use `CUDA 8.0` with `g++` as host compiler. To have similar primal run times, the problem sizes are chosen to be $4 \cdot 10^6$ on the CPU and $7.5 \cdot 10^3$ on the GPU (runs are single threaded in this benchmark). Results are given in Fig. 2.

The chart shows run times for one primal evaluation and for one adjoint evaluation using different approaches (`dco/c++`, `dco/map`, and TAPENADE). Apart from the outlier `g++4.7`, all compiler manufacturer, its versions, as well as the architecture seems to have a quite constant behaviour for the various versions of the code. We want to emphasize, that this is an hard-earned achievement of performance testing with different compilers. This effort has to continue to retrieve similar performance on Windows as well, where we currently see worse run times. Unfortunately, under Windows, the compilation time is a problem for this test code (especially with Visual Studio and Intel compilers we see $\sim$ 25 minutes; clang for Windows seems to perform better, i.e. $\sim$ 30 seconds).

[3] `https://www.graphics.rwth-aachen.de/software/comiso`

[4] `https://gitlab.stce.rwth-aachen.de/stce/MAP_benchmark.git`

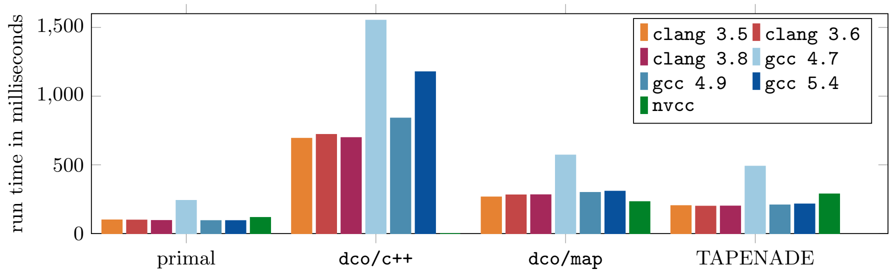

Fig. 2. Run times in milliseconds for the benchmark for a single primal evaluation as well as one adjoint evaluation using different approaches. `dco/c++` refers to a plain overloading approach using `dco/c++`, `dco/map` implements meta adjoint programming, and TAPENADE is a source transformation tool. Run times are shown for different compilers, versions, as well for the GPU (nvcc).

To be more specific, Fig. 2 shows the run times given in the following table.

		primal	`dco/c++`	`dco/map`	TAPENADE
CPU	run times	$\sim$ 95ms	693 $\sim$ 1552ms	$\sim$ 266ms	$\sim$ 199ms
	factor	1.0	7.3 $\sim$ 16.3	2.8	2.1
GPU	run times	117ms	—	232ms	288ms
	factor	1.0	—	2.0	2.5

The *factor* is the run time ratio of one adjoint evaluation to one primal evaluation, i.e. $\mathcal{R}$ from Eq. 3. A factor of 2.8 on the CPU for an overloading tool can be considered very good. But especially with the agressive compiler omptimizations on the GPU a factor of 2.0 can be achieved.

5.2 Application to CVA Code

In this section, we want to demonstrate applicability and performance when applying `dco/map` to a larger code base, which could appear as production code. Timings for parallel execution on the GPU are shown. The following five versions are evaluated, each using different data structures and synchronization techniques from Section 4.1.

1. *primal*
   The basic primal evaluation on the GPU running 32 threads with 200 thread blocks. The computation is carried out for 1000000 paths and 35 euler steps per path. The number of parameters (interest and credit

curves observed in the market) is 62. This is identical to the size of the calculated gradient.

2. `dco/map` *(a): atomic increments.*
   `dco/map` is used for the adjoint computation. Whenever shared inputs are read, the adjoint performs atomic increments.
3. `dco/map` *(b): reductions.*
   `dco/map` is used for the adjoint computation. Shared input arrays are enlarged as described in Section 4.1. Increments are first performed on thread local memory followed by a reduction and atomic increments at the end.
4. `dco/map` *(c): reductions and passive recomputations.*
   `dco/map` is used for the adjoint computation. Shared input arrays are handled with reductions. In addition, some parts of the code are omitted in the active forward run. The values that are required for the subsequent adjoint computation are computed passively. I.e. active forward computation is exchanged with passive forward recomputation.
5. `dco/map` *(d): reductions and checkpointing.* `dco/map` is used for the adjoint computation. Shared input arrays are handled with reductions. In addition, the passive recomputations are also omitted and required values are checkpointed.

Run time results in milliseconds are given in the following table.

	primal	`dco/map` (a)	`dco/map` (b)	`dco/map` (c)	`dco/map` (d)
run times	14.7	290	85	83	52
factor	1.0	19.7	5.8	5.6	3.7

As shown, plain application of `dco/map` with atomic increments results in a factor of 19.7. With some effort it is possible to achieve a factor of 3.7 for this code.

6 Summary and Outlook

As presented in this article, meta adjoint programming (MAP) is a promising approach to achieve usability as well as efficiency for adjoint code development on many-core architectures. The overloading approach ensures usability and template metaprogramming is the basis for efficiency. We've presented the interface as well as run time results for an implementation called `dco/map` demonstrating applicability to a real-world C++ code.

Using this approach in conjunction with other tools (e.g. `dco/c++`) as well as developing additional features (especially handling more control flow structs) is future work. Efficiency-wise, a profiling and thorough analyis on Windows (incl. respective compilers as Visual C++ or Intel) has to take place.

References

BZK12. D. Bommes, H. Zimmer, and L. Kobbelt. Practical mixed-integer optimization for geometry processing. In *Curves and Surfaces*, Lecture Notes in Computer Science, pages 193–206. Springer, 2012.

För14. Michael Förster. *Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP*. Springer, 2014.

GB02. N. R. Gauger and J. Brezillon. Aerodynamic shape optimization using adjoint method. *Journal of Aero. Soc. of India*, 54(3):247–254, 2002.

GG06. M. Giles and P. Glasserman. Smoking adjoints: Fast monte carlo greeks. *Risk*, 19(1):88–92, 2006.

GHR+16. Felix Gremse, Andreas Höfter, Lukas Razik, Fabian Kiessling, and Uwe Naumann. Gpu-accelerated adjoint algorithmic differentiation. *Computer physics communications*, 200:300–311, 2016.

GPGK08. Markus Grabner, Thomas Pock, Tobias Gross, and Bernhard Kainz. Automatic differentiation for gpu-accelerated 2d/3d registration. In *Advances in Automatic Differentiation*, pages 259–269. Springer, 2008.

GPGP97. MBMB Giles, N Pierce, M Giles, and N Pierce. Adjoint equations in cfd-duality, boundary conditions and solution behaviour. In *13th Computational Fluid Dynamics Conference*, page 1850, 1997.

GW08. A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 105 in Other Titles in Applied Mathematics. SIAM, 2nd edition, 2008.

HT12. R. Hannemann-Tamás. *Adjoint Sensitivity Analysis for Optimal Control of Non-Smooth Differential-Algebraic Equations*. PhD thesis, 2012.

HTWB11. J. Huang, Y. Tong, H. Wei, and H. Bao. Boundary aligned smooth 3d cross-frame field. *ACM Trans. Graph.*, 30(6):143:1–143:8, 2011.

HW94. John C Hull and Alan D White. Numerical procedures for implementing term structure models ii: Two-factor models. *The Journal of Derivatives*, 2(2):37–48, 1994.

LLN16. Klaus Leppkes, Johannes Lotz, and Uwe Naumann. Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features. Technical Report AIB-2016-08, RWTH Aachen University, September 2016.

MGH+05. M. B. Monagan, K. O. Geddes, K. M. Heal, G. Labahn, S. M. Vorkoetter, J. McCarron, and P. DeMarco. *Maple 10 Programming Guide*. Maplesoft, 2005.

Nau12. U. Naumann. *The Art of Differentiating Computer Programs. An Introduction to Algorithmic differentiation*. Number 24 in Software, Environments, and Tools. SIAM, 2012.

Pfa15. A. Pfadler. Computing sensitivities of cva using adjoint algorithmic differentiation. Master's thesis, d-fine GmbH, Kellog College, University of Oxford, 2015.

UHH+09. Jean Utke, Laurent Hascoet, Patrick Heimbach, Chris Hill, Paul Hovland, and Uwe Naumann. Toward adjoinable mpi. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

Aachener Informatik-Berichte

This list contains all technical reports published during the past three years. A complete list of reports dating back to 1987 is available from:

`http://aib.informatik.rwth-aachen.de/`

To obtain copies please consult the above URL or send your request to:

Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

2014-01 *	Fachgruppe Informatik: Annual Report 2014
2014-02	Daniel Merschen: Integration und Analyse von Artefakten in der modellbasierten Entwicklung eingebetteter Software
2014-03	Uwe Naumann, Klaus Leppkes, and Johannes Lotz: dco/c++ User Guide
2014-04	Namit Chaturvedi: Languages of Infinite Traces and Deterministic Asynchronous Automata
2014-05	Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, and Peter Schneider-Kamp: Automated Termination Analysis for Programs with Pointer Arithmetic
2014-06	Esther Horbert, Germán Martín García, Simone Frintrop, and Bastian Leibe: Sequence Level Salient Object Proposals for Generic Object Detection in Video
2014-07	Niloofar Safiran, Johannes Lotz, and Uwe Naumann: Algorithmic Differentiation of Numerical Methods: Second-Order Tangent and Adjoint Solvers for Systems of Parametrized Nonlinear Equations
2014-08	Christina Jansen, Florian Göbe, and Thomas Noll: Generating Inductive Predicates for Symbolic Execution of Pointer-Manipulating Programs
2014-09	Thomas Ströder and Terrance Swift (Editors): Proceedings of the International Joint Workshop on Implementation of Constraint and Logic Programming Systems and Logic-based Methods in Programming Environments 2014
2014-14	Florian Schmidt, Matteo Ceriotti, Niklas Hauser, and Klaus Wehrle: HotBox: Testing Temperature Effects in Sensor Networks
2014-15	Dominique Gückel: Synthesis of State Space Generators for Model Checking Microcontroller Code
2014-16	Hongfei Fu: Verifying Probabilistic Systems: New Algorithms and Complexity Results
2015-01 *	Fachgruppe Informatik: Annual Report 2015

2015-02 Dominik Franke: Testing Life Cycle-related Properties of Mobile Applications

2015-05 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder: Inferring Lower Bounds for Runtime Complexity

2015-06 Thomas Ströder and Wolfgang Thomas (Editors): Proceedings of the Young Researchers' Conference "Frontiers of Formal Methods"

2015-07 Hilal Diab: Experimental Validation and Mathematical Analysis of Cooperative Vehicles in a Platoon

2015-08 Mathias Pelka, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 1st KuVS Expert Talk on Localization

2015-09 Xin Chen: Reachability Analysis of Non-Linear Hybrid Systems Using Taylor Models

2015-11 Stefan Wüller, Marián Kühnel, and Ulrike Meyer: Information Hiding in the Public RSA Modulus

2015-12 Christoph Matheja, Christina Jansen, and Thomas Noll: Tree-like Grammars and Separation Logic

2015-13 Andreas Polzer: Ansatz zur variantenreichen und modellbasierten Entwicklung von eingebetteten Systemen unter Berücksichtigung regelungs- und softwaretechnischer Anforderungen

2015-14 Niloofar Safiran and Uwe Naumann: Symbolic vs. Algorithmic Differentiation of GSL Integration Routines

2016-01 * Fachgruppe Informatik: Annual Report 2016

2016-02 Ibtissem Ben Makhlouf: Comparative Evaluation and Improvement of Computational Approaches to Reachability Analysis of Linear Hybrid Systems

2016-03 Florian Frohn, Matthias Naaf, Jera Hensel, Marc Brockschmidt, and Jrgen Giesl: Lower Runtime Bounds for Integer Programs

2016-04 Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder: Proving Termination of Programs with Bitvector Arithmetic by Symbolic Execution

2016-05 Mathias Pelka, Grigori Goronzy, Jó Agila Bitsch, Horst Hellbrück, and Klaus Wehrle (Editors): Proceedings of the 2nd KuVS Expert Talk on Localization

2016-06 Martin Henze, René Hummen, Roman Matzutt, Klaus Wehrle: The SensorCloud Protocol: Securely Outsourcing Sensor Data to the Cloud

2016-07 Sebastian Biallas : Verification of Programmable Logic Controller Code using Model Checking and Static Analysis

2016-08 Klaus Leppkes, Johannes Lotz, and Uwe Naumann: Derivative Code by Overloading in C++ (dco/c++): Introduction and Summary of Features

2016-09 Thomas Ströder, Jürgen Giesl, Marc Brockschmidt, Florian Frohn, Carsten Fuhs, Jera Hensel, Peter Schneider-Kamp, and Cornelius Aschermann: Automatically Proving Termination and Memory Safety for Programs with Pointer Arithmetic

2016-10 Stefan Wüller, Ulrike Meyer, and Susanne Wetzel: Towards Privacy-Preserving Multi-Party Bartering

2017-01 * Fachgruppe Informatik: Annual Report 2017

2017-02 Florian Frohn and Jürgen Giesl: Analyzing Runtime Complexity via Innermost Runtime Complexity

2017-04 Florian Frohn and Jürgen Giesl: Complexity Analysis for Java with AProVE

2017-05 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl: Complexity Analysis for Term Rewriting by Integer Transition Systems

2017-06 Oliver Kautz, Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe: CD2Alloy: A Translation of Class Diagrams to Alloy

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.