Clad — Automatic Differentiation Using Clang and LLVM

2015 J. Phys.: Conf. Ser. 608 012055

(http://iopscience.iop.org/1742-6596/608/1/012055)

Clad – Automatic Differentiation Using Clang and LLVM

V Vassilev[1,2], M Vassilev[2], A Penev[2], L Moneta[1], and V Ilieva[3]

[1] CERN, PH-SFT, Geneva, Switzerland
[2] FMI, University of Plovdiv Paisii Hilendarski, Plovdiv, Bulgaria
[3] Princeton University, Princeton, NJ, USA

E-mail: vvasilev@cern.ch

Abstract. Differentiation is ubiquitous in high energy physics, for instance in minimization algorithms and statistical analysis, in detector alignment and calibration, and in theory. Automatic differentiation (AD) avoids well-known limitations in round-offs and speed, which symbolic and numerical differentiation suffer from, by transforming the source code of functions. We will present how AD can be used to compute the gradient of multi-variate functions and functor objects. We will explain approaches to implement an AD tool. We will show how LLVM, Clang and Cling (ROOT's C++11 interpreter) simplifies creation of such a tool. We describe how the tool could be integrated within any framework. We will demonstrate a simple proof-of-concept prototype, called Clad, which is able to generate n-th order derivatives of C++ functions and other language constructs. We also demonstrate how Clad can offload laborious computations from the CPU using OpenCL.

1. Introduction

Both industry and science often use the mathematical apparatus of differential calculus. Modeling financial markets, climatic changes or searching for the Higgs boson use function optimization and thus derivatives. The numerical calculation of the derivative values yields precision losses. They come from machine's floating point representation and the stability of the used numerical method. The computation fragility becomes even worse when computing higher order derivatives. In practice the user (a programmer) must consider very carefully the input values and the stepping delta, which sometimes can be far from trivial. Moreover, the derivative is hard-coded and becomes a maintenance issue. The developer has to differentiate the function mentally or using an external tool and translate it to the implementation language.

An alternative approach is the so called symbolic differentiation which overcomes the above-described issues, but sometimes it is slow [1]. It does not offer straight forward framework integration. For example, the function to be differentiated is hardcoded the programming language (e.g. C++) and it has to be translated to the symbolic language, differentiated and the result needs to be translated back to the framework's programming language. Moreover, both numerical and symbolic differentiation methods suffer when computing gradients and higher-order derivatives.

Despite sometimes overlooked, there is a hybrid approach which lays in the middle of the both extremes, resolving the mentioned issues. It is the automatic/algorithmic differentiation (AD). A widely accepted definition of AD is "a set of techniques to numerically evaluate the

derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.".

This paper is divided into sections as follows: Section 2, Related Concepts, discusses in brief some of the existing tools in the field and demonstrates their advantages and disadvantages. Section 3, Concepts of Clad, lays down the key concepts of our prototype – Clad. Section 4, Implementation, uncovers some of the important technical details of the concrete realization. Section 5, Applications, shows a few use cases in the context of the computer graphics and high-energy physics. Section 6, Computation Offload, describes a conceptual implementation employing parallel derivative computations. Section 7, Conclusion and Plans.

2. Related Concepts

Informally, the variety of incarnations of the AD can be classified in three major classes:

- Implemented via operator overloading – quick to implement at the cost of excessive amount of memory; most compilers evaluate expressions containing overloaded operators exactly as they are written, without performing any compile-time optimizations [2].
- Implemented via source-to-source transformations – results in faster derivative generation than the operator overloading. The concept of the idea is to rewrite an expression in a particular computer language, such as C++ or Fortran [3]. In order for the previously mentioned operations to be executed, the AD tool have to perform compiler-like processes like code parsing, code analyses, intermediate representations. The source transform approach is considered harder to implement but on the other hand it has advantages such as: better compiler optimizations and relatively lower amount of used memory [4].
- Implemented via compiler modules – an ideal realization of the source-to-source transformations is implementing the AD compiler modules. Integrating automatic differentiation capabilities in a compiler combines the advantages of both the operator overloading and source transform approaches [5].

ADOL-C is an AD tool, based on operator overloading. It can produce first and higher order derivatives of vector functions written in C/C++. One of its advantages is that the derived functions are valid C/C++ routines. ADOL-C can handle codes based on classes, templates and other C++ features. The tool supports the computation of standard objects required for optimization purposes such as gradients, Jacobians and Hessians [6]. ADOL-C uses the concept of active variables for denoting the possible differentiation independent variables. All of those variables have to be declared with a special variable type. The process of derivation is based on an ADOL-C specific internal representation and its start and finish are denoted with calls to special service routines. Every calculation incorporating active variables within the derivation process are recorded in a special data type. Once this is done, ADOL-C proceeds with executing its internal algorithms for computing the derivatives.

ADIC2 is a project following the source-to-source transform for Fortran, C and C++ based programs. It is built on top of the OpenAD project, which incorporates multiple, independent software components [7]. ADIC2 uses the ROSE compiler framework for parsing the input source code programs and for generating corresponding abstract syntax trees (ASTs). Unfortunately ADIC2 is closely coupled with ROSE thus changes in ROSE may result in failures in ADIC2. The tool uses configuration files specifying parameters and settings related to the differentiation process. The AST is passed to a dedicated analyzer which is reducing the amount of code passed to the differentiation algorithms. A XML based data structure is used in order to denote

which parts of the source code should be differentiated and which parts should be marked as statements and thus ignored by the differentiation algorithms. This XML based representation is used by another tool producing the differentiation and a differentiated AST. ADIC2 uses again the ROSE compiler for creating the output source code from the resulted AST.

The Fortran 95 compiler AD incorporates both the operator overloading and source transform techniques thus it is considered as a hybrid compiler. The mathematics support is handled by a compiler integrated module which provides overloaded versions of the arithmetic operators. The user is allowed to select the dependent and independent variables. The compiler uses a special active data type that is used to hold the function value as well as a vector for the direction derivatives. The overloaded operators as well as the active data types are contained in a compiler specific module. The independent variables as well as sections of the code that needs to be differentiated need to be marked using directive-like statements, in order for the compiler to recognize them. Unfortunately, no static data flow analyses of the code is conducted therefore the achieved efficiency is not optimal [8].

Source-to-source transformation technique is considered as the ideal approach for building large scale and run time crucial applications [9]. Furthermore, if the source-to-source transformers are implemented as a part of an existing compiler, it makes the implementation extremely efficient and easy to maintain at the cost of limited portability.

3. Concepts of Clad

An automatic differentiation algorithm takes a function (F) written in a programming language (L), translates it and yields another function (F') written in an another programming language (R), where the translation between $(F) \rightarrow (F')$ follows the rules of the differential calculus and turning F' into a derivative. For many tools (L) matches (R), i.e. the implementation language of the input function is the same as the programming language of the differentiated function. Usually it is so, because of design or technical limitations of the implementation. Most of the uses of (F') tend to be in the same framework and programming environment. An interesting domain of research is when (L) is different from (R) but (L) compatible with (R). For example, (F) is written in C/C++ and (F') is written in OpenCL [10] or CUDA [11].

A derivative is produced by transforming the body of F operation by operation. Every operation is transformed following the well-known differentiation rules. Every statement in a C++ function is treated as a standalone transformation entity. The translation employs the chain rule of differentiation.

The chain rule in differential calculus provide mathematically proved simplification of the translation process. It reduces the implementation complexity of the algorithm responsible for the differentiation. There are two general flavors of implementing the differentiation: top-down or bottom-up. In the top-down approach (also called forward mode or tangent mode) the computations are done on every step and the byproducts are thrown away. In the bottom-up approach (also called reverse mode or adjoint mode) the computations of the byproducts are stored and reused. This is particularly useful when computing derivatives of the same function with respect to different independent variables, for example when calculating function's gradient.

$$\frac{\partial z}{\partial t} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial t}, \tag{1}$$

where $z = f(x, y), x = g(t), y = h(t)$.

AD works with the assumption that a derivative exists, i.e. it does not check whether the differentiated function is continuous at any interval. If the logic of the original function is mathematically incorrect, there is no way how AD can produce a correct derivative.

4. Implementation

Clad is an AD tool, implemented on top of the LLVM compiler infrastructure [12]. The underlying compiler platform (Clang) builds an internal representation of the source code in the form of an AST. Parts of AST are used to communicate with third-party libraries, which can further specialize the compilation process. Because of the internal design and the communication through well-defined representations, Clad can be packed in different ways. It can be easily transformed from a plugin into a standalone tool or as an extension to third-party tools such as the interactive C++ interpreter Cling [13].

Once Clad receives the necessary information, it can decide whether a derivative is requested and produce it. It transforms the body of the candidate for differentiation and clones its AST nodes while applying the differentiation rules. If the body is not present it tries to find user provided directions how to proceed.

4.1. Usage

Currently, Clad is shipped as a plugin library for the Clang compiler. One can attach the library to the Clang compiler and it will produce derivatives at compilation time as a part of the current object file. However, this approach has some limitations: it is bound to a particular compiler and compiler version. It is also required to compile the project, using derivatives with the same compiler. For this reason, Clad can operate in three conceptually different modes (Figure 1), generating derivatives in different representations. Derivatives can be a part of, either:

- an object file – if a derivative is requested Clad would put it as if it was present in the source file;
- a source file – Clad can write out valid source code of the derivative into a source file. This is handy when the user wants to produce the list of the requested derivatives and compile them with another compiler;
- a shared library – Clad can write out the derivatives into a shared object (dynamic-link library). This is useful when the user prefers another compiler, which is binary compatible with Clang, For instance, the user wants to compile the application with GCC/ICC but wants the derivatives to be still used.

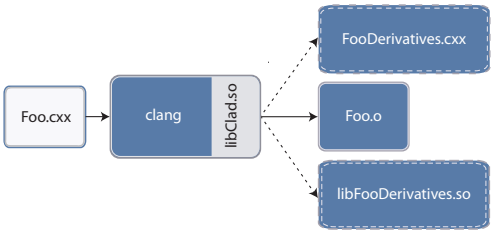

Figure 1: Clad usage scenarios.

Listing 1 shows how simple is to use Clad. The example demonstrates how to produce the first derivative of power of two (*pow2*). One needs to include a small header file, introducing Clad's runtime and use *clad::differentiate* function to specify which function needs to be differentiated. It takes two arguments: function to differentiate and the position of the independent variable. In the example below, Clad will fill in the body of *pow2_darg0* statement-by-statement following the derivation rules.

Listing 2 and 3 illustrate the textual mode in Clad. Clad can use the produced derivative straight away or to write its source code into a file. In the Listing 2 there is the user code and in the Listing 3 there are the automatically generated derivatives.

```
#include "clad/Differentiator/Differentiator.h"
double pow2(double x) { return x * x; }
double pow2_darg0(double); // Body will be filled by Clad.
int main() {
  clad::differentiate(pow2, 0);
  printf("Result is %f\n", pow2_darg0(4.2)); // Prints out 8.4
  return 0;
}
```

Listing 1: Clad will produce the body of the forward declared function.

```
#include "clad/Differentiator/Differentiator.h"

float example1(float x, float y) {
  return x * x * y * y;
}
void diffExamples() {
  clad::differentiate(example1, 0);
  clad::differentiate(example1, 1);
}
```

Listing 2: Differentiation of function *example1*.

```
float example1_darg0(float x, float y) {
  return (((x + x) * y) * y);
}

float example1_darg1(float x, float y) {
  return ((x * x) * y + x * x * y);
}
```

Listing 3: Derivatives of the function *example1*.

Valid C++ functions are generated with a system-defined name inferred from original function name, the independent variable of differentiation, and the derivative order. By construction, the signature of the newly-generated function is the same as the derivation template function.

Differentiation may be applied not only on simple functions but also on more complex language constructs such as:

- Templated C++ constructs;
- Classes and structs (eg. functors);
- Virtual functions.

4.1.1. Builtin Derivatives Some functions don't have bodies, because they are only forward declared and their implementation is in a library. The differentiation of some functions could be steered for performance improvements. Clad has a mechanism allowing to override the default differentiation policy and provide user-directed substitutions. Listing 4 specializes the default behavior of the differentiation, by replacing the default cosine (*cos*) with a user-specified one.

The implementation of the user-specific substitutions relies on a namespace with overloaded semantics. All substitutions need to be inside a special namespace called *custom_derivatives* and to follow specific naming rules. Before Clad builds a derivative, it checks if there is already a predefined derivative candidate. If this is the case it simply uses the one available.

All built-in derivatives rely on this mechanism. Differentiation of trigonometric functions and the derivatives of other special functions is done using user substitutions.

4.1.2. Higher Order Derivatives and Mixed Partial Derivatives Clad provides a convenient way of obtaining n-th order derivative out of a specific function. This is possible by invoking the templated version of the *clad::differentiate* function for instance see Listing 4.

Mixed derivatives are produced on Listing 5.

```
//...
namespace custom_derivatives { double sin_darg0(double x) { return my_better_impl::cos(x); } }
void secondDerivative() {
  clad::differentiate<2>(sin, 0);
}
```

Listing 4: Generation of the second derivative of sin, using substitutions.

```
float example2(float x, float y) {
  return x * x * y;
}
float example2_darg0(float x, float y);
auto example_darg0 = clad::differentiate(example2, 0);
auto example_darg0_darg1 = clad::differentiate(example2_darg0, 1);
```

Listing 5: Generation of mixed partial derivatives.

4.2. Performance

Clad works in synergy with the Clang compiler. Table 1 shows the times to compile the test cases with and without Clad. We test the overhead in two extreme cases – one, when it differentiates a function body with many statements (Listing 6), and another, where the body contains a very large expression (Listing 7). In more realistic scenarios Clad's overhead is negligible. It is so because the functions to be differentiated are much shorter, and the volume of these functions compared to the rest of the code is much smaller.

Table 1: Compilation times of Listing 6 and Listing 7.

Test	With Clad	No Clad	Clad overhead
Large body	1.007s	0.993s	0.014s (1.39%)
Large expression	3.258s	2.432s	0.826s (25.35%)

```
double f1(double x) {
  x = x + 2; x = x * x; x = x + x;
  // ... repeated 1020 times.
  return x;
}
```

Listing 6: Performance – Large function body.

```
double f2(double x) {
  return
    x+2 + x*x + x+x +
    // ... 1020 repetitions.
}
```

Listing 7: Performance – Large expression.

5. Applications

We explored two major application domains for the automatic differentiator Clad – computer graphics and high-energy physics.

5.1. Computer Graphics
We embedded Clad in a demonstration path tracer, called SmallPT [14]. We investigated how difficult and laborious would the integration be. SmallPT shoots rays of light from the viewer towards the scene and computes lighting properties at the intersection points. In order to do that, it computes three partial derivatives of the function describing the surface. They form the normal vector at the intersection point between a ray and a surface. We replaced the hand-written implementation with invocations of Clad. The performance is comparable to the performance of the hand-written calculations. Calculating the normal vectors using numerical approximation is about three times slower. The timing was collected from 10^{10} calculations of the gradient at random points on a sphere and a hyperbolic solid. Another advantage of embedding Clad allows even to improve the flexibility of our modification of SmallPT implementation, because it allows to find derivatives of arbitrary implicit surfaces (not only spheres as in the original code). The addition of a new surface is now easier, because the hand-written derivative will not be needed.

5.2. High-Energy Physics
The ROOT Framework [15] is widely adopted in high-energy physics for data analysis. It offers a wide range of mathematical tools for fitting and minimization. These tools use extensively derivatives and some of them are hand-written while others are numerically calculated. We plan on adopting Clad in ROOT6 through its C++ interpreter – Cling [13]. We expect performance improvement and Clad is expected to become a gateway for derivative computation on General-Purpose Computing on Graphics Processing Units (GPGPU). This work is still to be done soon.

6. Computation Offload
Derivative calculation is very computing-intense and time-consuming process. Clad's immediate goal is to increase the computational performance and to make use of all computing power of the environment. The natural evolution towards execution on GPGPUs is prominent. Computation can be accelerated by using the offloading of calculations from CPU towards the available GPGPUs. There are mainly two conceptually-different approaches:

- Using built-in approaches in the compiler to guide compilation (usually by 'pragma' directives) to turn on the parallel execution, the automatic offload of specific computation-intense parts of the code to selected accelerators. This is the approach in OpenMP 4.0 [16]. The advantage is that the code does not change significantly. If the compiler does not support these options, it ignores the parallelisation directives gracefully. The disadvantages are: very hard to make full use of the underlying computing architectures, because the pragma directives cannot specialize the algorithm for all architectures; requires some effort and re-engineering to get some parallelism.
- Using a specific programming language (such as OpenCL and CUDA) to talk to the hardware. This approach supposes rewriting the algorithms, or parts of them. The main advantage of this approach is that the algorithm can fully comply with the architecture of the target hardware. The disadvantage is that it requires a lot of effort and expertise to port it for every architecture and hardware.

We target scalability by changing semantics of the language syntax constructs and based on them we enhance the compiler actions. A key goal for this transformations is to be as transparent as possible. Most actions are performed automatically – at compile time. The goal is to use most of the advantages of the above-mentioned approaches, without forcing major changes in the user code.

Listing 8 presents an example of CPU-calculated gradient. In the implementation we find a sum of the partial derivatives of the Rosenbrock function.

```
float rosenbrock(float x[], int size) {
  auto rosenbrockX = clad::differentiate(rosenbrock_func, 0);
  auto rosenbrockY = clad::differentiate(rosenbrock_func, 1);
  float sum = 0;
  for (int i = 0; i < size-1; i++) {
    float one = rosenbrockX.execute(x[i], x[i + 1]);
    float two = rosenbrockY.execute(x[i], x[i + 1]);
    sum += one + two;
  }
  return sum;
}
```

Listing 8: Rosenbrock function implementation using Clad.

Listing 9 shows how the user should specify that he/she wants the compilation to be offloaded. The original implementation needs to be transformed into a lambda function, which is very straight-forward and almost transparent conversion. It is not much harder than putting a pragma directive in the source code. The code is not trivial to offload, because it has a for-loop, which should be made parallel to be able to exploit the properties of the GPGPU architecture. Furthermore, we need to calculate the sum of the results of the calculations, that requires reduction in the highly parallel hardware.

```
float rosenbrock_offloaded(float x[], int size) {
  return clad::experimental_offload([=] {
    auto rosenbrockX = clad::differentiate(rosenbrock_func, 0);
    auto rosenbrockY = clad::differentiate(rosenbrock_func, 1);
    float sum = 0;
    for (int i = 0; i < size-1; i++) {
      float one = rosenbrockX.execute(x[i], x[i + 1]);
      float two = rosenbrockY.execute(x[i], x[i + 1]);
      sum += one + two;
    }
    return sum;
  });
}
```

Listing 9: Computation offloading – Conceptual implementation using C++11 lambda function.

After the minimalistic transformation is done, Clad should take over and perform the rest automatically. Every call to *clad::experimental_offload* is replaced by a call to a function generated by the Clad plugin. This is done by an AST transform of the lambda function, allowing to transmit the parameters and to invoke one or more kernels.

In order to achieve optimal performance, Clad has to keep the computing units busy, which is far from a trivial task. Computation offload to GPGPU has to be very well planned and should happen when processing large data sets. One of the reasons is that all data needs to be copied through the system bus to a peripheral device, which introduces big overheads. Clad-generated code which passes data and executes kernels, should provide optimal parallel load as well as minimum data transfer between different types of memory (as it is possible to get too

much latency). The use of read only, write only parameters and other similar techniques are recommended in order to achieve better performance.

The given example was tested for performance with 100 calls to Rosenbrock's function over 1024*1024*48 float numbers. The resulting benchmarks are shown in Table 2.

Table 2: OpenCL parallel execution results.

Test	Device	Clock	Compute Units	Work Group	Global Memory	Local Memory	Time
Original	Intel i7-2635QM	2GHz	1	–	–	–	12.466s
Multicore	Intel i7-2635QM	2GHz	4	1024	8192MB	32KB	11.128s
ATI	Radeon 6490M	150MHz	2	256	256MB	32KB	18.183s
AMD	Devastator	844MHz	6	256	2047MB	32KB	15.479s
NVIDIA	Tesla K20m	705MHz	13	1024	4800MB	47KB	10.615s

The benchmarking showed that the CPU multicore is slightly faster than the original single core computation. This is because of the copying of large amounts of data. Overall, the GPGPU offload is very sensitive to the particular hardware. This is mainly due to the copy of large data sets sequentially between the host and the device. There is also latency because of the synchronous manner of execution. The next data transfer waits for the previous computation to be executed. In the case of NVIDIA Tesla, the CPU offload reached 50%, which is very good result for a prototype implementation.

7. Conclusion and Plans

Derivative production is important not only in high-energy physics but in many other domains. The automatic differentiation is often an overlooked approach to compute derivatives. It eliminates the precision losses in the numerical differentiation and it is faster than symbolic differentiation. It can also simplify the complexity of gradient computations. The AD tool mainly focuses on C and partially C++, because of the complexity of the language. We presented an innovative proof-of-concept prototype facilitating automatic differentiation, called Clad. It is based on the industrial-strength compiler technologies Clang and LLVM. Clad can differentiate non-trivial C++ routines and it is getting closer to production grade quality. Adding C support is trivial, the only work that needs to be done is writing the runtime environment to be C compliant.

We explained how it can be used to produce derivatives of various orders; how to produce mixed derivatives; how to perform user-based substitutions, steering the differentiation process; and how to offload computations in heterogeneous environments. There is still a lot of room for improvements. We have a conceptual implementation in OpenCL, providing a way how to offload computations in heterogeneous environments (making extensive use of GPGPUs). However, this work is still experimental and it requires a lot of efforts for Clad to be made robust.

We plan to generalize the computation of gradient and Jacobian of functions. We plan to reduce the computational complexity of these computations by making use of the reverse automatic differentiation mode in cases of many seeds.

Another immediate plan is to integrate Clad into Cling – the C++ interpreter of ROOT6, which would make Clad available to the entire high-energy physics community. Then, we can proceed using Clad in ROOT's minimization and fitting algorithms of ROOT.

Acknowledgments
The work was partially facilitated by Google Summer of Code Program 2013 and 2014, through CERN-SFT mentoring organization. The heterogeneous environment benchmarking are supported by the University of Plovdiv "Paisii Hilendarski" through Fund "Research" under contract NIS14-FMIIT-002/26.03.2014.

References

[1] Castro M, Vieira R and Biscaia Jr E 2000 Automatic differentiation tools in the dynamic simulation of chemical engineering processes *Brazilian Journal of Chemical Engineering* **17** 373–382 ISSN 0104-6632 URL http://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-66322000000400002&nrm=iso

[2] Bartholomew-Biggs M, Brown S, Christianson B and Dixon L 2000 Automatic differentiation of algorithms *Journal of Computational and Applied Mathematics* **124** 171–190 ISSN 0377-0427 numerical Analysis 2000. Vol. IV: Optimization and Nonlinear Equations URL http://www.sciencedirect.com/science/article/pii/S0377042700004222

[3] Gay D 2006 Semiautomatic differentiation for efficient gradient computations *Automatic Differentiation: Applications, Theory, and Implementations* (*Lecture Notes in Computational Science and Engineering* vol 50) ed Bücker M, Corliss G, Naumann U, Hovland P and Norris B (Springer Berlin Heidelberg) pp 147–158 ISBN 978-3-540-28403-1 URL http://dx.doi.org/10.1007/3-540-28438-9_13

[4] Bischof C H, Hovland P D and Norris B 2002 Implementation of automatic differentiation tools. *PEPM* ed Thiemann P (ACM) pp 98–107 ISBN 1-58113-455-X URL http://dblp.uni-trier.de/db/conf/pepm/pepm2002.html#BischofHN02

[5] Cohen M, Naumann U and Riehme J 2003 Towards differentiation-enabled Fortran 95 compiler technology *Proceedings of the 2003 ACM Symposium on Applied Computing* SAC '03 (New York, NY, USA: ACM) pp 143–147 ISBN 1-58113-624-2 URL http://doi.acm.org/10.1145/952532.952564

[6] Walther A and Griewank A 2012 Getting started with ADOL-C *Combinatorial Scientific Computing* ed Naumann U and Schenk O (Chapman-Hall CRC Computational Science) chap 7, pp 181–202

[7] Narayanan S H K, Norris B and Winnicka B 2010 ADIC2: Development of a component source transformation system for differentiating C and C++ *Procedia Computer Science* **1** 1845–1853 ISSN 1877-0509 iCCS 2010 URL http://www.sciencedirect.com/science/article/pii/S1877050910002073

[8] Naumann U and Riehme J 2005 A differentiation-enabled Fortran 95 compiler *ACM Trans. Math. Softw.* **31** 458–474 ISSN 0098-3500 URL http://doi.acm.org/10.1145/1114268.1114270

[9] Voßbeck M, Giering R and Kaminski T 2008 Development and first applications of TAC++ *Advances in Automatic Differentiation* (*Lecture Notes in Computational Science and Engineering* vol 64) ed Bischof C H, Bücker H M, Hovland P D, Naumann U and Utke J (Springer Berlin Heidelberg) pp 187–197 ISBN 978-3-540-68935-5 URL http://dx.doi.org/10.1007/978-3-540-68942-3_17

[10] Gaster B, Howes L, Kaeli D R, Mistry P and Schaa D 2013 *Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition* 2nd ed (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.) ISBN 9780124055209

[11] Cook S 2013 *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs* 1st ed (San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.) ISBN 9780124159334

[12] Lattner C and Adve V 2004 LLVM: A compilation framework for lifelong program analysis & transformation *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)* (Palo Alto, California)

[13] Vasilev V, Canal P, Naumann A and Russo P 2012 Cling – the new interactive interpreter for ROOT 6 *Journal of Physics: Conference Series* **396** 052071 URL http://stacks.iop.org/1742-6596/396/i=5/a=052071

[14] Beason K 2014 SmallPT: Global illumination in 99 lines of C++ URL http://www.kevinbeason.com/smallpt/

[15] Brun R and Rademakers F 1997 ROOT – an object oriented data analysis framework *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **389** 81–86 ISSN 0168-9002 new Computing Techniques in Physics Research V URL http://www.sciencedirect.com/science/article/pii/S016890029700048X

[16] OpenMP A R B 2013 OpenMP application program interface URL http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf