Computer Physics Communications 200 (2016) 300–311

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

GPU-accelerated adjoint algorithmic differentiation☆

Felix Gremse [a,b,*], Andreas Höfter [b], Lukas Razik [a,b], Fabian Kiessling [a], Uwe Naumann [b]

[a] Experimental Molecular Imaging, RWTH Aachen University, Germany

[b] Software and Tools for Computational Engineering, RWTH Aachen University, Germany

ARTICLE INFO

Article history:
Received 3 September 2014
Received in revised form
7 August 2015
Accepted 29 October 2015
Available online 12 November 2015

Keywords:
Adjoint algorithmic differentiation
GPU programming

ABSTRACT

Many scientific problems such as classifier training or medical image reconstruction can be expressed as minimization of differentiable real-valued cost functions and solved with iterative gradient-based methods. Adjoint algorithmic differentiation (AAD) enables automated computation of gradients of such cost functions implemented as computer programs. To backpropagate adjoint derivatives, excessive memory is potentially required to store the intermediate partial derivatives on a dedicated data structure, referred to as the "tape". Parallelization is difficult because threads need to synchronize their accesses during taping and backpropagation. This situation is aggravated for many-core architectures, such as Graphics Processing Units (GPUs), because of the large number of light-weight threads and the limited memory size in general as well as per thread. We show how these limitations can be mediated if the cost function is expressed using GPU-accelerated vector and matrix operations which are recognized as intrinsic functions by our AAD software. We compare this approach with naive and vectorized implementations for CPUs. We use four increasingly complex cost functions to evaluate the performance with respect to memory consumption and gradient computation times. Using vectorization, CPU and GPU memory consumption could be substantially reduced compared to the naive reference implementation, in some cases even by an order of complexity. The vectorization allowed usage of optimized parallel libraries during forward and reverse passes which resulted in high speedups for the vectorized CPU version compared to the naive reference implementation. The GPU version achieved an additional speedup of 7.5 ± 4.4, showing that the processing power of GPUs can be utilized for AAD using this concept. Furthermore, we show how this software can be systematically extended for more complex problems such as nonlinear absorption reconstruction for fluorescence-mediated tomography.

Program summary

Program title: AD-GPU

Catalogue identifier: AEYX_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AEYX_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: Standard CPC licence, http://cpc.cs.qub.ac.uk/licence/licence.html

No. of lines in distributed program, including test data, etc.: 16715

No. of bytes in distributed program, including test data, etc.: 143683

Distribution format: tar.gz

Programming language: C++ and CUDA.

Computer: Any computer with a compatible C++ compiler and a GPU with CUDA capability 3.0 or higher.

Operating system: Windows 7 or Linux.

RAM: 16 Gbyte

Classification: 4.9, 4.12, 6.1, 6.5.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author at: Experimental Molecular Imaging, RWTH Aachen University, Germany.
E-mail address: fgremse@ukaachen.de (F. Gremse).

http://dx.doi.org/10.1016/j.cpc.2015.10.027

External routines: CUDA 6.5, Intel MKL (optional) and routines from BLAS, LAPACK and CUBLAS

Nature of problem: Gradients are required for many optimization problems, e.g. classifier training or nonlinear image reconstruction. Often, the function, of which the gradient is required, can be implemented as a computer program. Then, algorithmic differentiation methods can be used to compute the gradient. Depending on the approach this may result in excessive requirements of computational resources, i.e. memory and arithmetic computations. GPUs provide massive computational resources but require special considerations to distribute the workload onto many light-weight threads.

Solution method: Adjoint algorithmic differentiation allows efficient computation of gradients of cost functions given as computer programs. The gradient can be theoretically computed using a similar amount of arithmetic operations as one function evaluation. Optimal usage of parallel processors and limited memory is a major challenge which can be mediated by the use of vectorization.

Restrictions: To use the GPU-accelerated adjoint algorithmic differentiation method, the cost function must be implemented using the provided AD-GPU intrinsics for matrix and vector operations. Unusual features:

GPU-acceleration.

Additional comments: The code uses some features of C++11, e.g. std::shared ptr. Alternatively, the boost library can be used.

Running time: The time to run the example program is a few minutes or up to a few hours to reproduce the performance measurements.

1. Introduction

1.1. Algorithmic Differentiation

Algorithmic Differentiation (AD) is a technique for augmenting numerical simulation programs with the ability to compute first and higher mathematical derivatives. In sharp contrast with classical numerical differentiation by finite differences, AD delivers gradients, Jacobians, and Hessians with machine accuracy by avoiding truncation through analytic differentiation of individual statements within an arbitrarily complex code implemented, for example, in C/C++ or Fortran. The adjoint (also: reverse) mode of AD is of particular interest in the context of large-scale sensitivity analysis and nonlinear optimization. Gradients of arbitrary size n can be computed at a constant (typically between 1 and 100) relative computational cost, i.e., with respect to the original simulation. A finite difference approximation of the same gradient would have to perform at least $n + 1$ original simulations. Assuming a run time of the original simulation of 1 min, the computation of a gradient of size $n = 10^6$ would take between 1 and 100 min in adjoint AD mode compared to at least $10^6 + 1$ min (almost two years) when using finite difference approximation or tangent (also: forward) mode AD. This run time will most likely turn out to be prohibitive, thus, rendering the use of gradient-based optimization techniques infeasible unless adjoint mode AD is available. Gradients of this size are very common, for example, in computational fluid dynamics simulations run frequently in the atmospheric sciences and in automotive or aircraft design.

AD has been applied successfully to a constantly growing number of practically relevant problems in Computational Science, Engineering, Medicine, and Finance as illustrated by numerous publications, for example, in the proceedings of the international AD conference series [1–6]. Both mathematical and algorithmic foundations of AD as well as various advanced subtopics are covered by two text books [7,8]. The AD community runs a web site (www.autodiff.org) with links to research groups, software tools, and an extensive bibliography.

When a computer program evaluates a cost function, the process evaluates differentiable elemental functions and operations, such as sine and multiplication, respectively, and creates many intermediate variables. The computation induces a directed acyclic graph (DAG) which contains the intermediate variables as nodes. The local partial derivatives of the elemental functions with respect to their arguments attached to the edges of the DAG yield the linearized DAG (lDAG) as shown in Fig. 1(a). The gradient can be computed by one reverse sweep through the lDAG propagating adjoints backwards from the dependent outputs (here y) to the independent inputs (here x_1 and x_2) until the gradient is accumulated. The adjoint $\bar{v}$ of a variable v is defined as the derivative of y with respect to v [7,8]. The adjoint of a variable is computed as a weighted sum of the adjoints of its successors in the lDAG, where weights are the local derivatives. Propagation of adjoints can be visualized as an adjoint DAG such as shown in Fig. 1(b). To perform this backpropagation, AD tools need to create and store some representation of the lDAG which is often referred to as the tape.

AD software tools take two alternative approaches: source code transformation (e.g. TAPENADE) or operator and function overloading (e.g. dco). Some AD tools implement a combination of both (NAG AD compiler). While source code transformation promises better performing derivative code, the tools typically fail to cover the latest language standards of Fortran and C/C++. Our target language is C++. Hence we use overloading to implement the ideas to be presented. See also [9–12].

1.2. High performance computing

Ideally it should be possible to compute the gradient in roughly twice the function evaluation time. Unfortunately, there are several reasons why this is rarely achieved in practice. Besides the arithmetic operations, many memory storage operations are required during creation and interpretation of the tape. The tape quickly becomes excessively large, because modern processors evaluate gigabytes

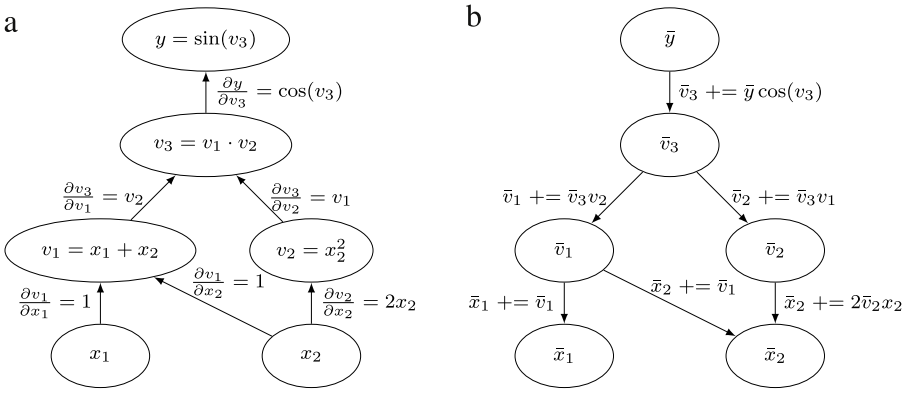


Fig. 1. Linearized adjoint DAGs. (a) Linearized DAG of $y = \sin((x_1 + x_2) \cdot x_2^2)$. (b) Adjoint DAG with the appropriate incremental adjoint calculations on the edges. Execution of these operations from top to bottom results in the adjoints of the input variables $(\bar{x}_1, \bar{x}_2)^T$, i.e., the gradient. Adjoints are initialized with zero.

Table 1
Vector and matrix operations and their adjoint operations (mostly taken from [13,14]). Bold lower case letters are vectors (in $\mathbb{R}^n$), where x_i is the ith component of a vector $\mathbf{x}$, and $\bar{\mathbf{x}}$ its adjoint vector. Bold capitals are matrices (in $\mathbb{R}^{n\times n}$, unless specified differently), $\bar{\mathbf{X}}$ is the adjoint matrix of $\mathbf{X}$, and v is a scalar. Matrices may be symmetric positive definite (indicated by s.p.d.). The element-wise application of a scalar function $\mathbf{y} := EW(\mathbf{x}, op)$ is defined by $y_i := op(x_i)$, for all $1 \le i \le n$, and the element-wise multiplication $\mathbf{z} := EW(\mathbf{x}, \mathbf{y}, \cdot)$ is defined by $z_i := x_i \cdot y_i$, for all $1 \le i \le n$. And ∂_op is the derivative of the unary operator op.

Expression	Adjoint expressions	Notes
$\mathbf{z} := \mathbf{x} + \mathbf{y}$	$\bar{x} \mathrel{+}= \bar{z}; \bar{y} \mathrel{+}= \bar{z}$	Addition
$\mathbf{y} := v \cdot \mathbf{x}$	$\bar{x} \mathrel{+}= v \cdot \bar{y}$	Scale
$v := \sum_{i=1}^{n} x_i$	$\bar{x}_i \mathrel{+}= \bar{v}, \forall i \in [1, 2, \ldots, n]$	Sum
$v := \mathbf{x}^T\mathbf{x}$	$\bar{x} \mathrel{+}= 2 \cdot \bar{v} \cdot \mathbf{x}$	Squared sum
$v := \mathbf{x}^T\mathbf{y}$	$\bar{x} \mathrel{+}= \bar{v} \cdot \mathbf{y}; \bar{y} \mathrel{+}= \bar{v} \cdot \mathbf{x}$	Dot product
$\mathbf{y} := \mathbf{A}\mathbf{x}$	$\bar{A} \mathrel{+}= \bar{y}^T\mathbf{x}; \bar{x} \mathrel{+}= \mathbf{A}^T\bar{y}$	Matrix–vector product
$\mathbf{y} := \mathbf{A}^{-1}\mathbf{x}$	$\bar{A} \mathrel{-}= (\mathbf{A}^{-1}\bar{y})\mathbf{y}^T; \bar{x} \mathrel{+}= \mathbf{A}^{-1}\bar{y}$	Solve $\mathbf{A}\,\mathbf{x} = \mathbf{y}$, with $\mathbf{A}$ s.p.d.
$\mathbf{Y} := \mathbf{A}^{-1}\mathbf{X}$	$\bar{\mathbf{A}} \mathrel{-}= (\mathbf{A}^{-1}\bar{\mathbf{Y}})\mathbf{Y}^T$ $\bar{\mathbf{X}} \mathrel{+}= \mathbf{A}^{-1}\bar{\mathbf{Y}}$	solve $\mathbf{A}\mathbf{X} = \mathbf{Y}$, with $\mathbf{A}$ s.p.d. and $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n\times k}$
$\mathbf{y} := EW(\mathbf{x}, op)$	$\bar{x} \mathrel{+}= EW(\bar{y}, EW(\mathbf{x}, \partial\_op), \cdot)$	Element-wise operation
$\mathbf{A} \mathrel{+}= \mathbf{x}\mathbf{y}^T$	$\bar{x} \mathrel{+}= \mathbf{x}\bar{A}; \bar{y} \mathrel{+}= \bar{A}\mathbf{y}$	Rank-1 update of $\mathbf{A}$

of intermediate variables per second. This issue can be addressed by several techniques. C++ expression templates can be used to preaccumulate local Jacobians [11,12]. Moreover, the lDAG can be pruned by elimination of edges or vertices while creating the tape [15].

Parallelization is another challenge for AD. While it may be relatively easy to speed up the cost function using OpenMP, overloading AAD tools struggle to do so due to the need for synchronization of tape accesses during the forward and backward phases [16]. The use of OpenMP has been shown to potentially slow down the gradient computations which can be improved by the use of expression templates [12].

Well-designed cost functions are likely to make use of highly optimized libraries such as BLAS or LAPACK to perform vector and matrix operations. Since these functions are typically linked in, and not available as source code, they are not easily automatically differentiable. Differentiable versions of these operations need to be implemented which typically cannot compete with the performance of highly optimized library functions. Alternatively, AD tools can treat these functions as intrinsics, by providing symbolic derivatives. A corresponding vector operation needs to be performed during the reverse pass in this case, which is often similar to the forward operation and can also be computed using a highly optimized library [13] (Table 1). This vectorization of the adjoints can lead to substantially smaller tapes because the memory overhead is reduced. Furthermore, it allows integration of iterative linear solvers, which may not be automatically differentiable otherwise [14,17].

GPUs provide massive computational power by running thousands of light-weight threads in parallel [18]. For this to be effective, special programming techniques are required, because the number of concurrent threads is limited by resources, such as registers and shared memory [19]. GPU memory is very fast but limited in size. Since the bandwidth between CPU and GPU memory is much lower, data transfers between them should be minimized as much as possible [20]. C or C++ code can be compiled for GPUs using frameworks such as CUDA [21] or OpenCL [22]. Fortunately, many optimized GPU-accelerated operations for vectors and matrices are available [23–27]. While features such as dynamic memory allocation, recursion, and function pointers are possible with the latest generation of GPUs [21], avoiding these features often results in better performance. For these reasons, implementation of general GPU-accelerated AAD tools is difficult and previous attempts focused on using source transformations inside shaders for special applications [28] or on batch computation of many small gradients which fit into shared memory [29].

1.3. Fluorescence-mediated tomography

This software was originally developed and applied for fluorescence-mediated tomography (FMT) [17,30–32]. FMT is a noninvasive imaging technology to assess the three dimensional distribution of fluorescence in human fingers, breasts or, most commonly, laboratory mice [33]. The fluorescence can be provided by contrast agents or genetically transfected cells expressing fluorescent proteins [31]. Commercially available FMT devices operate with continuous wave illumination, e.g., with a servo-mounted laser, and acquire multiple diffuse transillumination images of the incident and reemitted light [34,35]. In the last years, FMT has become a broadly applicable tool for biomedical and pharmaceutical research, particularly in combination with an anatomical modality such as μCT which provides valuable information for reconstruction and image analysis [30,32,33,36]. Due to the high scattering in the near-infrared wavelengths, fluorescence reconstruction is a mathematically and computationally challenging problem. One important aspect is an accurate optical model containing information about the shape and heterogeneous scattering and absorption maps. In our previous study we derived a scattering map by performing automated segmentation of tissue classes (bones, lungs, fat, muscle, skin) based on the μCT data and assigned known scattering coefficients [17]. Subsequently, an absorption map was reconstructed by iterative nonlinear minimization of a cost function, expressing the difference between measured and predicted surface measurements. The absorption map turned out to be important for quantitative fluorescence reconstruction in several organs such as heart, liver, and kidneys [17]. The required gradients were computed using the proposed software for GPU-accelerated AD and we found the vectorization beneficial for computational speed but also necessary to integrate a GPU-accelerated sparse linear solver into the AD system [14,25]. We found the cost function for this problem to be too complicated to introduce the software, however, and therefore resort to more simple cost functions in the following.

1.4. Aim of the study

The aim of this study is to provide a software for GPU-accelerated AAD and show its value using performance measurements. Therefore, we introduce four increasingly complex cost functions in Section 2. In Section 3 we report performance measurements with respect to gradient computation times and memory consumption. To separate the improvements of vectorization and GPU acceleration, we first distinguish between naive and vectorized AAD for CPUs. Then, vectorized CPU and GPU versions are compared to assess the additional gains due to using GPUs. Section 4 provides a user guide for the software, which is available as supplemental material, and describes how to systematically extend the functionality for more complex cost functions. Finally, in Section 5, we discuss our results and future directions of research.

2. Cost functions

We implemented four different cost functions, which are described in the following. They all have a variable input size, which allows us to analyze the performance for increasing dimensions of the input domain.

2.1. Sum of sigmoids

The `SumSigmoid` cost function is defined as $y = \sum_{i=1}^{n} \text{sigmoid}(x_i)$, i.e., it applies the scalar sigmoid function to each element of the input vector followed by computing the sum of the results (see Fig. 2). We use this function as a simple introductory example. The scalar sigmoid function $y = 1/(1 + \exp(-x))$, of which the derivative is $\partial\_\,\text{sigmoid}(x) = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$, is used, for example, in the context of gradient-based methods for training neural networks [37]. We use the prefix "$\partial\_$" to denote the first derivative. The computational complexity of evaluating the value and the gradient of `SumSigmoid` is linear in the input size, i.e., $O(n)$. For our experiments we use a default input size of $n = 2 \cdot 10^7$.

2.2. Robust linear least squares

Sparse least squares problems need to be solved for many applications. They occur when measurements $\mathbf{b}$ are acquired about an unknown state $\mathbf{x}$, which is observed through a linear system, which is represented as a sparse matrix $\mathbf{A}$. The least squares term $\|\mathbf{Ax} - \mathbf{b}\|^2$ is minimized to reconstruct $\mathbf{x}$. Typically, iterative sparse least squares methods are used [38]. Due to the squared residual, outliers or high-valued measurements may dominate the solution. The problem can be overcome by replacing the squaring operator with the robust Huber function or another robust loss function [39]. The Huber function is defined as

$$Huber(x) = \begin{cases} \frac{1}{2}x^2, & \text{if } |x| \leq \delta \\ \delta\left(|x| - \frac{1}{2}\delta\right), & \text{otherwise.} \end{cases}$$

We arbitrarily set $\delta = 1.28$. Its parabolic shape near the origin and a linear shape for inputs with higher magnitude yields a cost function (Fig. 3) which is not quadratic anymore, however. This cost function can be minimized using iterative nonlinear methods, such as the nonlinear conjugate gradient method, which requires the gradient at each iteration [40]. For our experiments we assume that the sparse matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ contains 6 random nonzero elements per row and that $m = 5 \cdot n$. The constant $\mathbf{A}$ is stored inside the C++ functor that implements the cost function. The computational complexity is linear in terms of the input size, because there is a constant number of nonzeros per row. We use a default input size of $n = 10^6$.

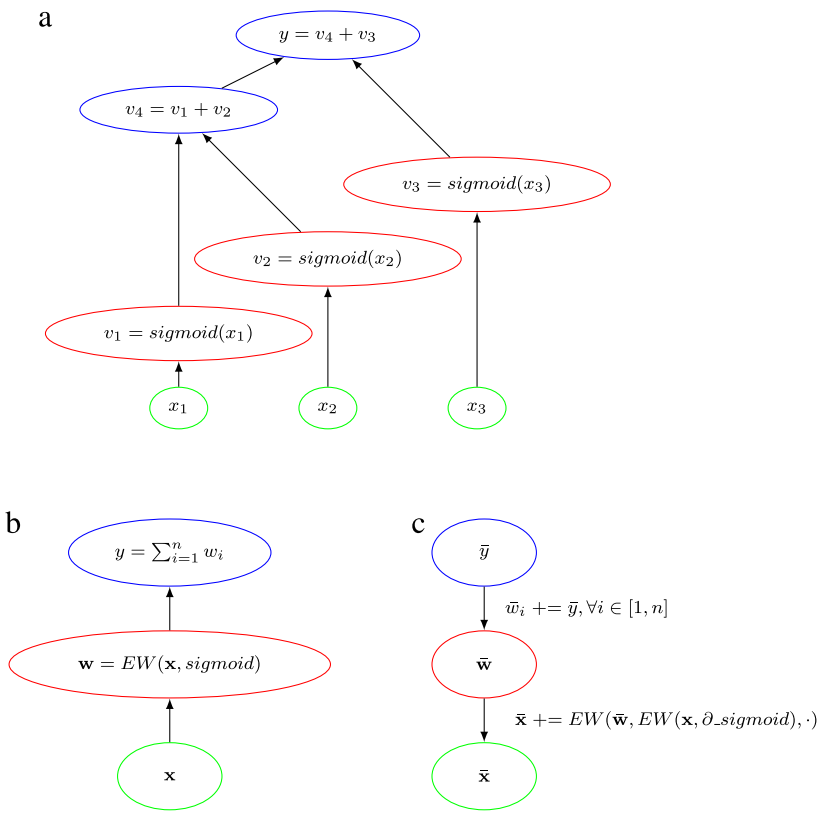


Fig. 2. `SumSigmoid` function. (a) DAG of the naive mode for $n = 3$. (b) Vectorized DAG, where $EW(\ldots)$ is the element-wise operation as defined in Table 1. (c) Adjoint DAG. Colors are used to highlight corresponding parts of naive and vectorized DAGs. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

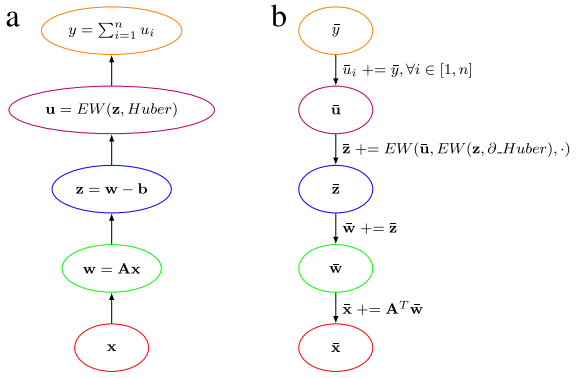


Fig. 3. Robust Least Squares. (a) Vectorized DAG. (b) Corresponding adjoint operations, where $EW(\ldots)$ is the element-wise operation as defined for Table 1. **A** and **b** are constant data that do not depend on the input **x**.

2.3. Maximum entropy models

Maximum entropy models, also called log-linear models, can be used as binary classifiers. The log-likelihood l is computed as the linear combination of a feature vector $\mathbf{x}$ with the parameter $\mathbf{f}$ of the log-linear model: $l = \mathbf{x}^T\mathbf{f}$. Training of the classifier requires a large number of feature vectors. The cost term $\sum_{i=1}^{n}(-\text{logit}((\mathbf{A}\mathbf{x})_i))$ needs to be minimized, where $\mathbf{A} \in \mathbb{R}^{m\times n}$ contains the features as rows [41] and the scalar logit() function is defined as $\text{logit}(x) := \log(x/(1-x))$.

For our experiments we assume $m = 5 \cdot n$ to avoid overfitting. The DAG is shown in Fig. 4. The computational complexity and the memory complexity are $O(n^2)$. We use a default input size of $n = 5000$.

2.4. Cholesky solver

Symmetric positive definite (s.p.d.) matrices occur frequently, for example in the context of finite element or finite difference methods for partial differential equations [17,42]. Often, such a matrix is constructed from input parameters and then a linear system is solved during a forward function evaluation. AAD tools need to backpropagate the adjoints through the linear solver which may be problematic with iterative solvers [14]. Dense s.p.d. systems can be solved using Cholesky decomposition at computational complexity $O(n^3)$. While

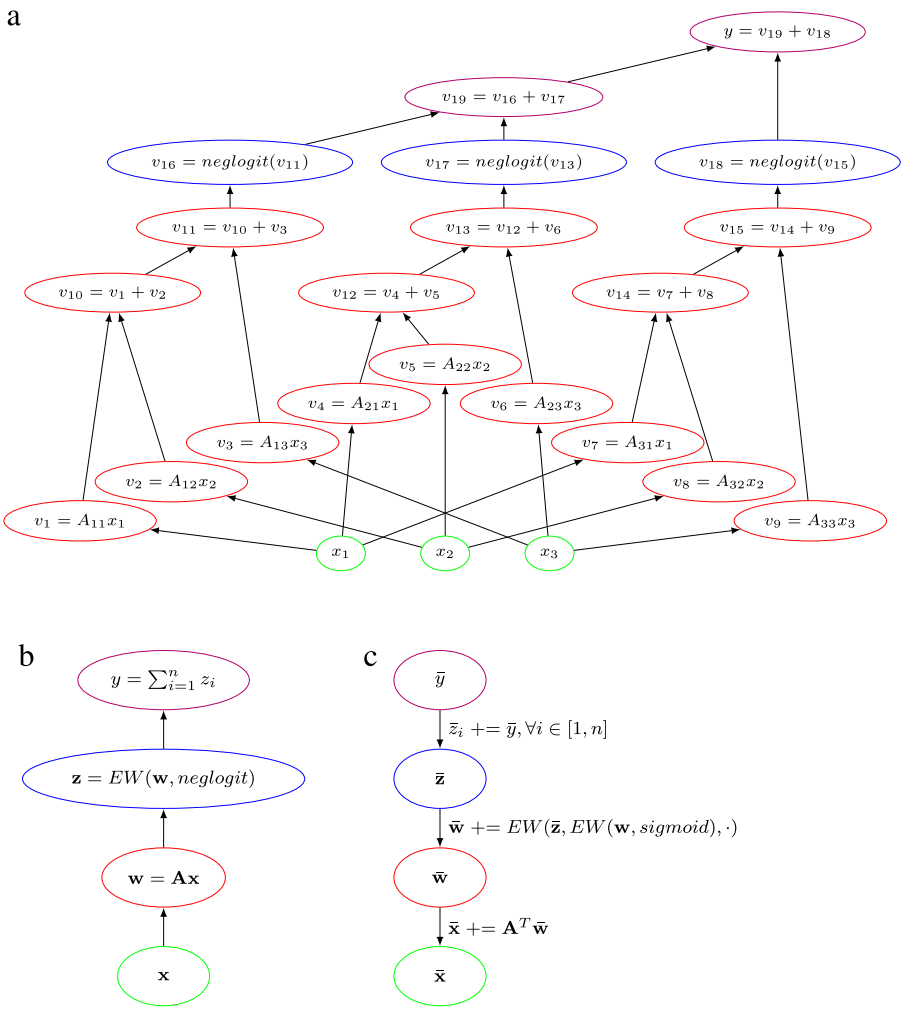


Fig. 4. Maximum Entropy cost function. (a) DAG for $n = 3$ with intrinsic function neglogit$(x) = -\log(x/(1-x))$. (b) Vectorized DAG. (c) Corresponding reverse vector operations are written on the edges of the graph. $EW(\ldots)$ is the element-wise operation as defined in Table 1. **A** is constant data that does not depend on the input **x**.

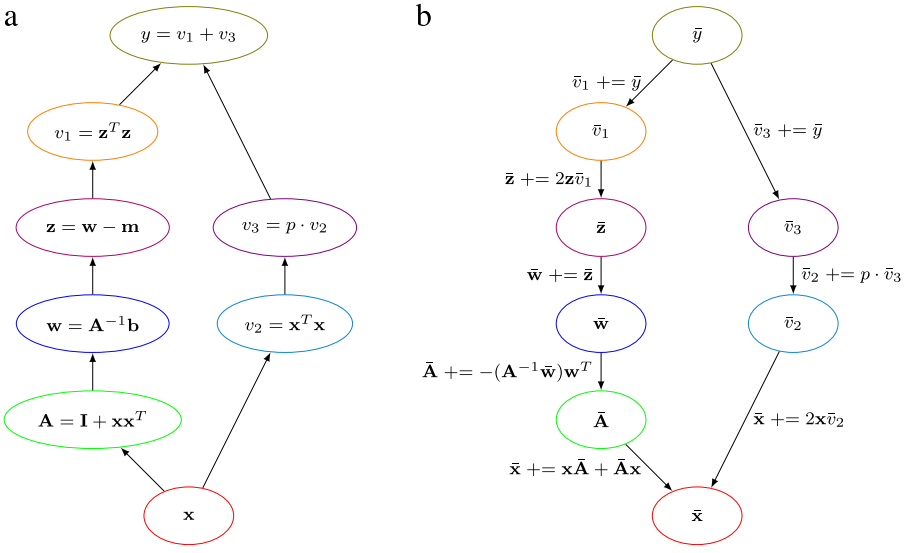


Fig. 5. `SolveCholesky` cost function: (a) The DAG involves construction of a dense matrix **A** by applying a rank-one update with **x** to the identity matrix **I**. Then a linear system $\mathbf{A}\mathbf{w} = \mathbf{b}$ is solved. The cost term consists of the squared $\mathscr{L}_2$-distance of the solution **w** to a constant vector **m**. Furthermore, a magnitude penalty (with weight p) is applied on **x** for regularization. **A**, **b**, **m**, and p are constant, i.e., do not depend on **x**. (b) The adjoint DAG requires solving a linear system with the same matrix as in the forward evaluation.

realistic problems often use sparse iterative solvers, we here use a dense Cholesky solver for the purposes of simplicity and illustration and because it is compatible with naive AAD methods.

Table 2
Function evaluation times (in ms) using CPU and GPU computing, evaluated for four reference cost functions. Three CPU versions were evaluated, single-threaded, OpenMP-parallelized and linked against highly optimized parallel BLAS and LAPACK functions from the MKL. Furthermore, the GPU version was measured. To show the incremental improvement, the speedup relative to the previous column is shown in parentheses. OpenMP-parallelization provided a systematic speed-up for all four cost functions. MKL provided a notable further speedup for the `SolveCholesky` cost function only. The GPU version achieved a speed-up for all four cost functions compared to the CPU-MKL version.

Cost function	CPU serial	CPU OMP	CPU MKL	GPU
SumSigmoid	310	88.4 (3.50)	102 (0.87)	10.0 (10.1)
RobustLeastSquares	277	91.5 (3.03)	123 (0.74)	18.5 (6.66)
MaximumEntropy	114	32.4 (3.52)	24.6 (1.32)	5.63 (4.36)
SolveCholesky	126	54.6 (2.31)	11.9 (4.59)	6.06 (1.96)

Table 3
Tape memory (in MB) required for gradient computations (for each column, the ratio to the previous column is shown in parentheses). Memory consumption is much higher for the naive implementation and similar for CPU- and GPU-based vectorized versions.

Cost function	CPU Naive	CPU Vector	GPU Vector
SumSigmoid	2 726	641 (4.25)	611 (1.05)
RobustLeastSquares	4 425	922 (4.80)	879 (1.05)
MaximumEntropy	12 026	1002 (12.0)	955 (1.05)
SolveCholesky	10 635	13.0 (816)	13.5 (0.97)

In the `SolveCholesky` cost function (see Fig. 5), a matrix $\mathbf{A} = \mathbf{I} + \mathbf{x}\mathbf{x}^T \in \mathbb{R}^{n\times n}$ is constructed based on the input $\mathbf{x}$ followed by solving system $\mathbf{A}\mathbf{w} = \mathbf{b}$ for $\mathbf{w}$. Then the squared $\mathscr{L}_2$-distance of $\mathbf{w}$ and a constant measurement vector is used as cost term in addition to a magnitude penalty applied on $\mathbf{x}$ for regularization. Similar cost functions occur for example in diffuse optical tomography [17,14,43]. The computation of the gradient *∂_SolveCholesky* results in excessive use of tape memory due to the primal computational cost of $O(n^3)$. Fortunately, this can be improved using the concept of vectorization, because the adjoint operation requires solving a linear system using the same matrix (Table 1). Linear s.p.d. systems are self-adjoint; yielding an adjoint system with the same system matrix. The previously factorized matrix can be stored on the tape and reused during backpropagation. This reduces the memory usage for taping from $O(n^3)$ to $O(n^2)$. For our experiments we used a default input size of $n = 900$.

3. Performance measurements

For the performance measurements, we used four different cost functions, which are described in the previous section. First, the time for the function evaluations was measured to assess the speedup achieved by parallel CPU and GPU processing. Then we measured the memory required for taping during gradient computations. We compared the three mentioned versions, i.e., the naive and vectorized CPU versions and the vectorized GPU version. To achieve this, we queried the amount of memory assigned to the process before and after generation of the complete DAG. For the CPU-based versions the CPU memory was measured. For the GPU version, only the GPU memory was measured because this constitutes the main bottleneck and the used CPU memory is negligible in most cases. Furthermore, the time to compute the gradients was measured. We performed these measurements for certain default input sizes of the cost functions to allow reporting in tables. These input sizes were selected so that they require less than 16 GB memory for taping when using the naive version. Additionally, we performed all measurements for a range of input sizes, to assess the effect of the input size on the computation time. All function evaluations and gradient computations were performed using double precision floating-point arithmetic.

3.1. Function evaluation speed

First, we measured the effect of CPU parallelization using OpenMP. This resulted in a speedup of 3.1 ± 0.6 (mean and standard deviation) compared to a single-threaded version (Table 2). Usage of the Math Kernel Library (MKL), which is also internally parallelized, resulted in a notable further speedup only for the `SolveCholesky` function. Compared to the CPU-MKL version, the GPU-accelerated code resulted in an additional speedup of 5.8 ± 3.5 (Table 2).

3.2. Tape size

Measurements of the memory consumption showed that vectorization yields substantial reductions of the tape size (Table 3). For the first three functions, tape size reductions by factors between 4.2 and 12.0 were achieved. For the `SolveCholesky` cost function, the required tape memory was reduced by an order of complexity, i.e., from $O(n^3)$ to $O(n^2)$, which resulted in a dramatic reduction by the factor 815.7. The memory consumption was similar for the CPU-based and GPU-based vectorized versions. The memory reduction is particularly important for the GPU mode because GPUs are much more limited in the amount of available memory.

Table 4
Gradient computation times (in ms). The different versions are listed with increasing performance from left to right. The relative speedup is shown in parentheses. OpenMP improves the vectorized mode but slows down the naive mode.

Cost function	CPU Naive OMP	CPU Naive serial	CPU Vector serial	CPU Vector OMP	CPU Vector MKL	GPU Vector
SumSigmoid	4 251	3025 (1.41)	869(3.48)	341.1 (2.55)	351.1 (0.97)	26.8 (13.1)
RobustLeastSquares	7 333	6661 (1.10)	718 (9.28)	222.7 (3.22)	307.7 (0.72)	42.5 (7.24)
MaximumEntropy	18 194	9889 (1.84)	814 (12.1)	144.9 (5.62)	74.5 (1.94)	10.3 (7.23)
SolveCholesky	17 723	9904 (1.79)	132 (74.8)	56.4 (2.35)	19.4 (2.91)	8.11 (2.39)

Table 5
Ratios of gradient computation times to function evaluation times. CPU gradient computation times are compared to the CPU-MKL function evaluation version. The GPU gradient times are compared to the GPU function evaluation times.

Cost function	Naive CPU OMP	Naive CPU serial	Vector CPU serial	Vector CPU OMP	Vector CPU MKL	Vector GPU
SumSigmoid	41.9	29.8	8.56	3.36	3.46	2.68
RobustLeastSquares	59.6	54.2	5.83	1.81	2.50	2.30
MaximumEntropy	741.0	402.8	33.2	5.90	3.04	1.83
SolveCholesky	1489.7	832.5	11.1	4.74	1.63	1.34

3.3. Gradient computation speed

During performance measurements, we found that OpenMP resulted in a performance loss for the naive CPU version (Table 4), because multiple threads have to synchronize when accessing the tape during generation of the compute graph. Turning off OpenMP resulted in a speedup of 1.5 ± 0.3. Vectorization resulted in a cumulative speedup of 24.9 ± 33.5 compared to the single-threaded naive version. Usage of OpenMP provided an additional speedup of 3.6 ± 1.8 for the vectorized version. This shows that vectorization is a suitable method to achieve a benefit from OpenMP for AAD. By using the highly optimized MKL library for the vector operations, another speedup of 1.6 ± 1.0 could be achieved. Finally, the GPU version resulted in a speedup of 7.5 ± 4.4 compared to the CPU-MKL version (Table 4).

However, for small input sizes, the CPU version performs better than the GPU version, because GPU operations require a certain amount of work to sufficiently utilize the GPU processors (Fig. 6).

We also compared the gradient computation times to the function evaluation times (Table 5). This shows that using the naive mode results in gradient computation times that are much higher, i.e., more than 100 times, than the function evaluation times. Therefore, this would become a major bottleneck for gradient descent methods where one gradient computation and a few function evaluations are required for the line search at each iteration. Using CPU and GPU vector modes, the ratios are substantially lower, showing that the gradient computation time would not dominate the total time during iterative gradient descent methods. For `SolveCholesky`, it is even below 2, because the expensive Cholesky decomposition can be reused during backpropagation.

3.4. Devices and software

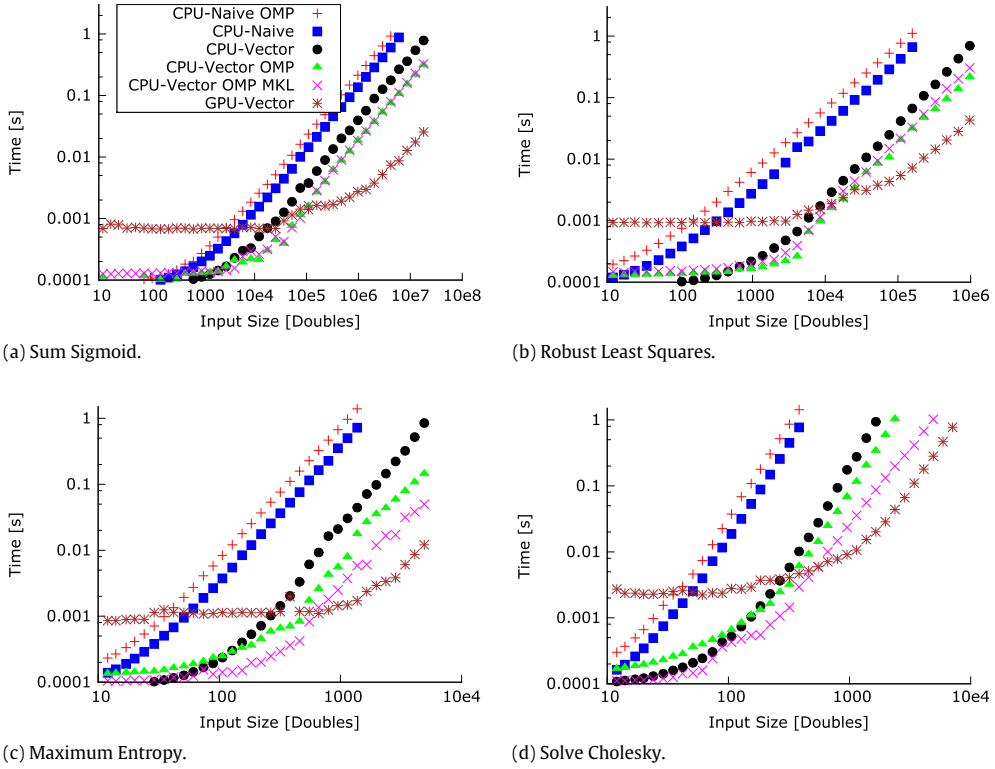
A PC (Fujitsu Celsius M730) equipped with an Intel Xeon E5-1620 v2 (3.7 GHz) quad-core processor, 64 GB of DDR3 RAM (CAS latency 13, DRAM Frequency 933.2 MHz) and an Nvidia GeForce GTX Titan Black (15 Kepler cores, 2880 CUDA cores, 889 MHz, 6 GB memory at 1750 MHz) was used for performance measurements. The used operating system was Windows 7 (64-bit). The Intel MKL library 11.1 Update 3 was used for parallel host operations. The C++ code was compiled with Visual Studio 2012 Ultimate (Update 4). The CUDA Toolkit 7.0 was used for the measurements.

4. User guide

4.1. Gradient computations

To use our code for gradient computations, a cost function needs to be implemented as a C++ class providing a generic overload of the function call operator, taking a `DeviceVector<T>` as input (Listing 1). Then the function `AdjointModelGradient` is called to compute the gradient and function value at a given argument. This approach triggers template instantiation and compilation of the cost function class using the `AdjointDouble` class instead of the regular C++ `double`. An `AdjointDouble` contains a `double` as value and a reference to its adjoint which is stored on the tape because the adjoint is required beyond the lifetime of the `AdjointDouble` instance. The overloaded operators, such as `*`, `+`, and `-`, compute the local derivatives and store these on the tape. Furthermore, overloads are implemented for some scalar functions such as *sin*, *cos*, or *sigmoid*.

The tape consists of two arrays, one for the adjoints and one for the local partial derivatives. These correspond to nodes and edges of the lDAG, respectively. The arrays are generously preallocated, which is a $O(1)$ operation because only the virtual address space is reserved and the actual memory is assigned to the process on demand, i.e., page by page by the operating system. Storage of each edge requires 24 bytes, i.e., one `double` for the local derivative and two pointers to the source and destination adjoints. The scalar operations perform the storage on the statically allocated tape in a thread-safe manner, using an atomic increment of the current position on the node and edge arrays, which is implemented using the `std::atomic` functionality available with C++ 11. After completion of the function evaluation, the entire lDAG is stored on the tape and the adjoints are propagated backwards to compute the gradient.



(a) Sum Sigmoid.

(b) Robust Least Squares.

(c) Maximum Entropy.

(d) Solve Cholesky.

Fig. 6. Gradient computation times over the input size. The performance is measured for the four cost functions. Asymptotically, the naive CPU version with OpenMP is the slowest. It is improved by turning off OpenMP because this avoids synchronization overhead during taping. Vectorization results in a strong performance gain, further benefited by turning on OpenMP. Usage of highly optimized MKL functions further increases the performance. GPU processing achieves the highest performance, asymptotically.

Listing 1: Usage of the software. A C++ class `MaximumEntropyFunctor` is defined which provides a generic function call operator. `MaximumEntropyFunctor` contains constant data (a dense matrix **A**). The C++ main function instantiates an object of `MaximumEntropyFunctor`, allocates argument **x** and gradient **g**, and calls the function `AdjointModelGradient` to compute the gradient and the function value at **x**.

```
1  class MaximumEntropyFunctor{
2    DeviceMatrix<double> A; // constant data
3  public:
4    explicit MaximumEntropyFunctor(DeviceMatrix<double> A):A(A){}
5
6    template<typename T>
7    T operator()(DeviceVector<T> x){
8      DeviceVector<T> tmp=Mul(A,x);
9      DeviceVector<T> y=ElementWise(tmp,ElementFunctors::NegLogit());
10     return Sum(y);
11   }
12 };
13
14 int main(void){
15   DeviceMatrix<double> A=LoadFromFile(...);
16   MaximumEntropyFunctor f(A);
17   int n=100;
18   DeviceVector<double> x(n);
19   ElementWiseInit(x,0.0);
20   DeviceVector<double> g(n);
21   double score=AdjointModelGradient(x,g,f);
22 }
```

4.2. GPU-accelerated operations

To enable vector-based programming we implemented generic C++ classes to represent vectors and matrices (`DeviceVector<>` and `DeviceMatrix<>`). For these classes, only the elements, e.g., `doubles`, are stored on the GPU, while the memory required for the class structure is stored in CPU memory. A `DeviceVector` can represent an original memory block or point to a subvector of another vector or a row or column of a matrix. Similarly, a `DeviceMatrix` can represent an original 2D array or point to a submatrix of another `DeviceMatrix<>`. This is achieved by storing a stride, i.e., gap, between subsequent elements of vectors and between rows of matrices. Fortunately, this is compatible with the interface of the cuBLAS library [26]. Furthermore, these classes manage the memory in an exception-safe manner using C++11 smart pointers (`std::shared_ptr`), i.e., a memory block is automatically released when the last subvector, submatrix, row or column referring to it is destructed. The ability to use subvectors and submatrices is particularly useful

to implement blocking schemes, as we did to provide a simple and self-contained Cholesky solver. We linked vector and matrix operations to the cuBLAS library wherever possible. More general reduction operations than available in cuBLAS were implemented using the Thrust library [24]. Furthermore, we implemented CUDA kernels for element-wise operations and to transpose and multiply sparse matrices in the CRS format [25]. Since the `RobustLeastSquares` cost function performs multiplication of a sparse matrix with a dense vector, we also implemented this operation on the GPU.

To utilize the processing power of GPUs for AAD, we had to extended the naive adjoint AD tool with special functionality, simply because the `AdjointDouble` class and its overloads for scalar operations are not suitable for lightweight threads of GPUs. Modern GPUs run thousands of threads concurrently and the required synchronization when taping scalar operations would be prohibitively expensive. To enable vectorized taping, which avoids these problems, both data structures and operations were adjusted. Using the rich feature set of C++ [44], we implemented template specializations for the templated classes `DeviceVector<>` and `DeviceMatrix<>` for the element type `AdjointDouble`. In contrast to the default template instantiation which would use a memory block to store `AdjointDoubles` continuously, i.e., values and adjoints in alternating succession, the template specialization stores values and adjoints in separate arrays. This structure-of-array pattern allows direct usage of these arrays for high-performance libraries such as cuBLAS [26]. The template specialization `DeviceMatrix<AdjointDouble>` provides basically the same functionality as the regular template instantiations, e.g., `DeviceMatrix<double>`, including the ability to create submatrices and fetch rows and columns.

To trigger vectorized taping during the forward pass, we implemented overloaded functions for all required vector and matrix operations, e.g., for matrix–vector multiplication in Listing 2. These function overloads apply the required vector operation on the value array. Furthermore, they tape a 'propagator' object which performs the corresponding vector operation to the adjoint arrays during the reverse pass (Table 1). The propagator classes need to derive from the abstract base class `IPropagator`, i.e., need to implement a `Propagate()` method to be called during backpropagation of the adjoints, as well as a virtual destructor. The `IPropagator` objects are stored as special edges on the tape. It should be noted that scalar operations for `AdjointDouble` are still performed on the CPU and not on the GPU. This allows combination of the naive CPU version with the vectorized GPU version which is useful for incremental optimization of the adjoint code, e.g., by porting only the most expensive operations to the GPU.

Listing 2: Extending the functionality for matrix–vector multiplication. A propagator class `DeviceVectorMatrixMulPropagator` which derives from the abstract base class `IPropagator` provides the adjoint operation in the virtual `Propagate` method. A function overload for matrix–vector multiplication adds an instance of the propagator class to the tape.

```
1  class IPropagator{
2  public:
3    virtual void Propagate()=0;
4    virtual ~IPropagator(){}
5  };
6
7  class DeviceVectorMatrixMulPropagator: public IPropagator{
8    DeviceVector<double> x,y;
9    DeviceMatrix<double> A;
10 public:
11   DeviceVectorMatrixMulPropagator(DeviceVector<double> y,
12     DeviceVector<double> x, DeviceMatrix<double> A)
13     :y(y),A(A),x(x){}
14
15   virtual void Propagate(){
16     y+=x*A;
17   }
18 };
19
20 static DeviceVector<AdjointDouble> Mul(
21   DeviceMatrix<double> A,DeviceVector<AdjointDouble> x)
22 {
23   DeviceVector<AdjointDouble> y(A.Height());
24   Mul(y.Values(),A,x.Values());
25   AdjointModelTape::Add(
26     new DeviceVectorMatrixMulPropagator(x.Adjoints(),y.Adjoints(),A));
27   return y;
28 }
```

4.3. Extending the functionality

To apply the software to more complex problems, extension of the functionality may be necessary because we only provide overloads for the operations listed in Table 1. Then the required functions need to be implemented as exemplified in Listing 2. To ensure correct implementation we recommend to test each overload individually. This can be achieved by testing the gradient computation of a simple cost function using the new function. Tangent mode AD is very useful for this purpose because it also allows computation of the gradient. Since it may be prohibitively slow to compute entire gradients for cost functions with large input domains, we only compute a random subset of the gradient elements using the function `TestRandomGradientPositions`. Unfortunately this requires availability of differentiable GPU code which is usually not available for linked-in functions. Alternatively, the GPU code can be tested against the naive CPU AAD code that only requires a differentiable CPU version which may be considerably easier and faster to implement.

5. Discussion

We showed how the concept of vectorization benefits both CPU- and GPU-based AAD. The naive CPU-based version requires excessive amounts of memory for taping which is strongly improved by vectorization. This reduction of the memory consumption is particularly

useful for GPU-accelerated AAD, because GPUs typically have much less memory than the host computers. Cholesky decomposition is an example where the tape size is reduced by an order of complexity, i.e., from $O(n^3)$ to $O(n^2)$, when switching to vector mode.

Gradient computation times should ideally be of the same order as the function evaluation times, because they require a similar amount of arithmetic operations [7]. However, the naive reverse mode results in run times that are orders of magnitudes slower than the function evaluations. This is due to three reasons. First, the cost function needs to be provided as differentiable code, because high-performance libraries that are linked in are not automatically differentiable. This code usually performs worse than the vendor-provided and highly optimized libraries such as MKL. Second, the taping causes many memory operations covering large amounts of memory, which hardly benefits from the caching architecture. Third, the usage of multi-threading, e.g., OpenMP, is counter-productive because the threads need to synchronize when accessing the tape, which is necessary at each arithmetic operation.

Vectorization solves or at least reduces these problems. It allows usage of highly optimized parallel vector operations during forward and backward passes of the gradient computation, which greatly improves the performance. Furthermore, differentiable versions of the vector operations are not required which reduces programming effort unless they are needed for the purpose of testing.

Vectorization simplifies the transition towards GPU processing, because chunky operations are ideal for GPUs and because existing optimized vector and matrix operations can be used. Our approach keeps the main data of the tape on the GPU, i.e., the vector elements. This is important because memory transfers between GPU and CPU memory are avoided. Data about the logical structure of the vectors and matrices is stored on CPU memory. Using this approach, we could show that GPU processing achieves substantial reductions for the gradient computation times compared to the vectorized CPU version, for sufficiently large problem sizes.

While tangent mode AD is hardly suitable to compute large gradients, because one function evaluation is required for each element of the gradient, we found it very useful for testing. To test a newly implemented function overload, we integrate it into a simple cost function and compute a random subset of the gradient entries using the tangent versions and compare it with the gradient computed in adjoint mode. This approach allows systematic and robust extension of functionality by adding more vector operations.

5.1. Limitations

Our naive AAD code is relatively simple and we used it to illustrate difficulties and possible improvements of AAD. There are other approaches that perform on line pruning during creation of the tape [15] or use C++ expression templates to combine a whole statement into one taped operation [11,12]. Furthermore, our method cannot be used directly for existing code. Instead, it requires that the cost function is re-implemented using our vector and matrix classes and operations. While vector-based processing has the advantage of using optimized vector operations, it may be less cache efficient because entire vectors may need to be channeled repeatedly through the caching architecture. While the vector operations reduce the tape size, the limited GPU memory may still not be sufficient for some applications. This could be overcome by taping data on the host memory once the GPU runs out of memory. Currently, we only have one static tape, i.e., multiple threads cannot create their own tapes. This could be improved by instantiating multiple instances of tapes, e.g., one per thread [12].

5.2. Conclusion

We showed how GPUs can be used for efficient gradient computations using adjoint algorithmic differentiation. The concept of vectorization reduces the tape size and simplifies the transition to many-core architectures. We believe that this method is comprehensible, extensible and broadly applicable for many problems in scientific computing that require gradient computations and we successfully applied this method in the context of fluorescence-mediated tomography; see [17,30–32] for an in-depth discussion of this case study.

Acknowledgments

This work was supported by the European Research Council (ERC Starting Grant 309495: NeoNaNo) and the German Ministry for Education and Research (BMBF) (funding programs Virtual Liver (0315743), Photonik Forschung Deutschland (13N13355)).

References

[1] G. Corliss, A. Griewank (Eds.), Automatic Differentiation: Theory, Implementation, and Application, in: Proceedings Series, SIAM, Philadelphia, 1991.
[2] M. Berz, C. Bischof, G. Corliss, A. Griewank (Eds.), Computational Differentiation: Techniques, Applications, and Tools, in: Proceedings Series, SIAM, Philadelphia, 1996.
[3] G. Corliss, C. Faure, A. Griewank, L. Hascoët, U. Naumann (Eds.), Automatic Differentiation of Algorithms—From Simulation to Optimization, Springer, New York, 2002.
[4] H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (Eds.), Automatic Differentiation: Applications, Theory, and Tools, in: Lecture Notes in Computational Science and Engineering, vol. 50, Springer, Berlin, 2005.
[5] C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, in: Lecture Notes in Computational Science and Engineering, vol. 64, Springer, Berlin, 2008.
[6] S. Forth, P. Hovland, E. Phipps, J. Utke, A. Walther (Eds.), Recent Advances in Algorithmic Differentiation, Vol. 87, Springer Science & Business Media, Berlin, 2012.
[7] A. Griewank, A. Walther, Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, second ed., in: Other Titles in Applied Mathematics, vol. 105, SIAM, Philadelphia, PA, 2008.
[8] U. Naumann, The Art of Differentiating Computer Programs. An Introduction to Algorithmic Differentiation, in: Software, Environments, and Tools, vol. 24, SIAM, Philadelphia, PA, 2012.
[9] R.A. Bartlett, D.M. Gay, E.T. Phipps, Automatic differentiation of C++ codes for large-scale scientific computing, in: V. Alexandrov, G. Albada, P. Sloot, J. Dongarra (Eds.), Computational Science ICCS 2006, in: Lecture Notes in Computer Science, vol. 3994, Springer, 2006, pp. 525–532.
[10] D. Gay, Semiautomatic differentiation for efficient gradient computations, in: M. Bücker, G. Corliss, U. Naumann, P. Hovland, B. Norris (Eds.), Automatic Differentiation: Applications, Theory, and Implementations, in: Lecture Notes in Computational Science and Engineering, vol. 50, Springer, 2006, pp. 147–158.
[11] R.J. Hogan, Fast reverse-mode automatic differentiation using expression templates in C++, ACM Trans. Math. Software 40 (4) (2014) 26.
[12] J. Lotz, K. Leppkes, U. Naumann, dco/c++ - Derivative Code by Overloading in C++, Tech. Report, AIB-2011-06, RWTH Aachen, May 2011.
[13] M.B. Giles, Collected matrix derivative results for forward and reverse mode algorithmic differentiation, in: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 35–44.
[14] A.J. Davies, D.B. Christianson, L.C.W. Dixon, R. Roy, P. van der Zee, Reverse differentiation and the inverse diffusion problem, Adv. Eng. Softw. 28 (4) (1997) 217–221.
[15] U. Naumann, Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph, Math. Program. 99 (3) (2004) 399–421.

[16] C. Bischof, N. Guertler, A. Kowarz, A. Walther, Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C, in: C. Bischof, H. Bücker, P. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, in: Lecture Notes in Computational Science and Engineering, vol. 64, Springer, 2008, pp. 163–173.
[17] F. Gremse, B. Theek, S. Kunjachan, W. Lederle, A. Pardo, S. Barth, T. Lammers, U. Naumann, F. Kiessling, Absorption reconstruction improves biodistribution assessment of fluorescent nanoprobes using hybrid fluorescence-mediated tomography, Theranostics 4 (10) (2014) 960–971.
[18] M. Garland, D.B. Kirk, Understanding throughput-oriented architectures, Commun. ACM 53 (11) (2010) 58–66.
[19] N. Bell, S. Dalton, L.N. Olson, Exposing fine-grained parallelism in algebraic multigrid methods, SIAM J. Sci. Comput. 34 (4) (2012) C123–C152.
[20] C. Gregg, K. Hazelwood, Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, in: IEEE International Symposium on Performance Analysis of Systems and Software, 2011, pp. 134–144.
[21] NVIDIA Corporation, Santa Clara, CA, USA, CUDA C Programming Guide, 2014. http://docs.nvidia.com/cuda/cuda-c-programming-guide.
[22] J.E. Stone, D. Gohara, G. Shi, OpenCL: A parallel programming standard for heterogeneous computing systems, IEEE Des. Test 12 (3) (2010) 66–73.
[23] N. Bell, M. Garland, Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2009. http://code.google.com/p/cusp-library.
[24] J. Hoberock, N. Bell, Thrust: A parallel template library, version 1.7.0, 2013. http://thrust.github.io/.
[25] F. Gremse, A. Höfter, L.O. Schwen, F. Kiessling, U. Naumann, GPU-accelerated sparse matrix-matrix multiplication by iterative row merging, SIAM J. Sci. Comput. 37 (1) (2015) C54–C71.
[26] NVIDIA Corporation, Santa Clara, CA, USA, CUBLAS Documentation, 2014. http://docs.nvidia.com/cuda/cublas.
[27] EM Photonics, Newark, DE, USA, CULA Programmer's Guide, 2014. http://www.culatools.com/cula_dense_programmers_guide.
[28] M. Grabner, T. Pock, T. Gross, B. Kainz, Automatic Differentiation for GPU-Accelerated 2D/3D Registration, in: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (Eds.), Advances in Automatic Differentiation, Springer, 2008, pp. 259–269.
[29] G. Kozikowski, B. Kubica, Parallel approach to Monte Carlo simulation for option price sensitivities using the adjoint and interval analysis, in: R. Wyrzykowski, J. Dongarra, K. Karczewski, J. Waśniewski (Eds.), Parallel Processing and Applied Mathematics, in: Lecture Notes in Computer Science, vol. 8385, Springer, Berlin, Heidelberg, 2014, pp. 600–612.
[30] B. Theek, F. Gremse, S. Kunjachan, S. Fokong, R. Pola, M. Pechar, R. Deckers, G. Storm, J. Ehling, F. Kiessling, T. Lammers, Characterizing EPR-mediated passive drug targeting using contrast-enhanced functional ultrasound imaging, J. Control. Release 182 (2014) 83–89.
[31] S. Kunjachan, R. Pola, F. Gremse, B. Theek, J. Ehling, D. Moeckel, B. Hermanns-Sachweh, M. Pechar, K. Ulbrich, W.E. Hennink, G. Storm, W. Lederle, F. Kiessling, T. Lammers, Passive versus active tumor targeting using RGD- and NGR-Modified polymeric nanomedicines, Nano Lett. 14 (2) (2014) 972–981.
[32] F. Gremse, D. Doleschel, S. Zafarnia, A. Babler, W. Jahnen-Dechent, T. Lammers, W. Lederle, F. Kiessling, Hybrid μCT-FMT imaging and image analysis, J. Vis. Exp. (100) (2015) e52770.
[33] A. Ale, V. Ermolayev, E. Herzog, C. Cohrs, M.H. de Angelis, V. Ntziachristos, FMT-XCT: in vivo animal studies with hybrid fluorescence molecular tomography-X-ray computed tomography, Nature Methods 9 (6) (2012) 615–620.
[34] F. Leblond, S.C. Davis, P.A. Valdés, B.W. Pogue, Pre-clinical whole-body fluorescence imaging: Review of instruments, methods and applications, J. Photochem. Photobiol. B 98 (1) (2010) 77–94.
[35] F. Gremse, F. Kiessling, Hybrid optical imaging, in: A. Brahme (Ed.), Comprehensive Biomedical Physics, in: Optical Molecular Imaging, vol. 4, Elsevier, 2014, pp. 269–280.
[36] S. Kunjachan, F. Gremse, B. Theek, P. Koczera, R. Pola, M. Pechar, T. Etrych, K. Ulbrich, G. Storm, F. Kiessling, T. Lammers, Noninvasive optical imaging of nanomedicine biodistribution, ACS Nano 7 (1) (2013) 252–262.
[37] I.E. Livieris, P. Pintelas, A new conjugate gradient algorithm for training neural networks based on a modified secant equation, Appl. Math. Comput. 221 (2013) 491–502.
[38] C.C. Paige, M.A. Saunders, LSQR: An algorithm for sparse linear equations and sparse least squares, ACM Trans. Math. Software 8 (1) (1982) 43–71.
[39] P.J. Huber, E.M. Ronchetti, Robust Statistics, second ed., Wiley, Hoboken, NJ, 2009.
[40] J. Nocedal, S. Wright, Numerical optimization, in: Springer series in operations research and financial engineering, second ed., Springer, New York, NY, 2006.
[41] R. Malouf, A comparison of algorithms for maximum entropy parameter estimation, in: Proceedings of the 6th Conference on Natural Language Learning, in: COLING-02, vol. 20, Association for Computational Linguistics, Stroudsburg, PA, USA, 2002, pp. 1–7.
[42] J. Lotz, U. Naumann, Algorithmic Differentiation of Numerical Methods: Tangent-Linear and Adjoint Direct Solvers for Systems of Linear Equations, Tech. Report, AIB-2012-10, RWTH Aachen (May 2012).
[43] S.R. Arridge, O. Dorn, V. Kolehmainen, M. Schweiger, A. Zacharopoulos, Parameter and structure reconstruction in optical tomography, J. Phys.: Conf. Ser. 135 (1) (2008) 012001.
[44] S.B. Lippman, J. Lajoie, B.E. Moo, C++ Primer, Auflage: 5th revised edition. Edition, Addison Wesley, Upper Saddle River, NJ, 2012.