IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 28, 2018, accepted February 28, 2018, date of publication March 6, 2018, date of current version March 16, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2812173

Fast Automatic Differentiation for Large Scale Bundle Adjustment

YAN SHEN[1] AND YUXING DAI[1,2]
[1]College of Electrical and Information Engineering, Hunan University, Changsha 410082, China
[2]College of Mathematics, Physics and Electronic Information Engineering, Wenzhou University, Wenzhou 325035, China

Corresponding author: Yuxing Dai (daiyx@hnu.edu.cn)

This work was supported by the Natural Science Foundation of Zhejiang Province under Grant LZ16E050002.

ABSTRACT A parallelized implementation of automatic differentiation that derives from the problem of bundle adjustment is proposed in this paper. Reverse mode of automatic differentiation is more efficient to compute the derivatives of functions with n real-value parameters, which is the case of computing the Jacobian matrix in bundle adjustment problem. By reason of a large number of small derivative computing tasks being needed in bundle adjustment problem, we implement an automatic differentiation library based on operator overloading and OpenCL parallel framework. In order to parallelize the computation in the framework of OpenCL, we generate forward and reverse computational sequences from computational graph by topological sorting. This library enables us to write down the function formula elegantly and then evaluate the derivatives rapidly and automatically. Finally, large scale bundle adjustment data sets is used to evaluate our proposed implementation. The result shows that our implementation runs about 3.6 times faster than Ceres Solver, which utilizes OpenMP parallel programming model.

INDEX TERMS Automatic differentiation, bundle adjustment, stereo vision, parallel computing.

I. INTRODUCTION

Many numerical optimization methods involve computations of derivatives, Jacobians and Hessians. Traditionally, these computations can be figured out manually. It is affordable for many small problems. For optimization problems of finding n real-valued parameters $\mathbf{x}$ that minimize m real-valued functions, writing down the derivatives by hand can be a formidable task. There are three categories of computations of derivatives in the computer world. The first one is numerical differentiation by using finite difference approximations. But it can cause large round-off error and truncation error problem. Another one is symbolic differentiation by applying symbolic computation. The main drawback of symbolic differentiation is that it produces large symbolic expressions as the complexity of formula increases. This problem is known as an expression swell. So, the final method which is the topic of this paper is automatic differentiation that can eliminate these problems.

The primary motivation of developing a parallelized automatic differentiation be derived from solving large-scale bundle adjustment (BA) [1], [2] problems in stereo vision. BA problem minimizes the error of reconstruction by adjusting parameters which include poses of camera and positions of 3D points. Numerical optimization methods such as Levenberg-Marquardt (LM) [3] and Powell's Dog-leg [4] are introduced to solve this kind of problem. Both of these methods should compute the derivatives of the projection function $\mathbf{x} = \mathbf{F}(\mathbf{X})$, where $\mathbf{X}$ is a 3D point and $\mathbf{x}$ is the projection on the image plane of this point. Also, this point projection formula can be replaced by a line projection formula in line-based reconstruction. There are two aspects to expound that it is necessary to do further research on fast automatic differentiation for bundle adjustment problem. First, for very large BA problems, tens of millions of image points exist, which means the derivatives of huge amount of projection functions need to be computed. Efficient and parallel implementation of such an automatic differentiation can speed up this process during multiple iteration procedures of BA. Second, in the field of robotics, visual simultaneous localization and mapping (SLAM) is very important [5]. There are three most popular algorithms used in SLAM to estimate camera pose and 3D point locations. They are Kalman filtering (KF), particle filtering (PF), and BA. BA can be superior to KF and PF in terms of accuracy [6], [7], but with much more computation. Fast automatic differentiation makes BA more real-time.

Automatic differentiation consists of two modes [8]: forward mode and reverse mode. For functions with n real-valued parameters, reverse mode is preferred for its computational efficiency. There are a few softwares of automatic differentiation designed for general purpose. ADOL-C is the earliest implementation using C programing language, and is still under development [9]. Hogan [10] proposed a C++ version library which is called "Adept" by utilizing expression template to provide compile time-time representation of mathematical expressions. A Java implementation called ADiJac uses source transformation rather than operator overloading to generate derivatives for functions expressed in Java bytecode [11]. These implementations do not give enough thought to nowadays' techniques of parallelization which can get better performance for large scale problem. For bundle adjustment, Ceres Solver [12] implements an OpenMP supported automatic differentiation which enables parallel acceleration by exploiting multiple cores in CPU. In this paper, we proposed an even faster OpenCL-based automatic differentiation(CLAD) which also uses C++ template and operator overloading.[1] Forward and reverse computational sequences are generated from computational graph by using C++ style function formulation. Then, these sequences are executed on parallel computing devices very efficiently. The rest of this paper is split into following several sections. Section II introduces the automatic differentiation in a brief way. In Section III, we demonstrate the problem of bundle adjustment. Section IV shows how is the CLAD implemented. In Section V, it shows that CLAD can be used for bundle adjustment problem efficiently. Finally, we draw conclusions in Section VI.

II. AUTOMATIC DIFFERENTIATION

Truncation error-free numerical values of the derivatives rather than closed form expressions or approximate numerical derivatives are obtained by automatic differentiation. This largely benefits from the combination of chain rule and computer programming. A function can be decomposed into elementary operations which include binary arithmetic operations and unary functions. Binary arithmetic operations typically are additions, subtractions, multiplications and divisions. Unary functions usually are transcendental functions such as trigonometric functions, exponential and logarithmic functions. As an example, Fig. 1 shows the computational graph of two real functions which are defined in Eq. (1). A computational graph is a directed acyclic graph(DAG).

$$\begin{aligned} f_1(x_1, x_2) &= sin(x_1) + ln(x_2) \\ f_2(x_1, x_2) &= ln(x_1) + sin(x_1) * x_2 \end{aligned} \tag{1}$$

These two functions can be regarded as one vector function $f : \mathbb{R}^2 \mapsto \mathbb{R}^2$. Derivatives related to x_i can be computed in two ways: forward mode and reverse mode.

[1]Source code is available from https://github.com/arczee/CLAD/

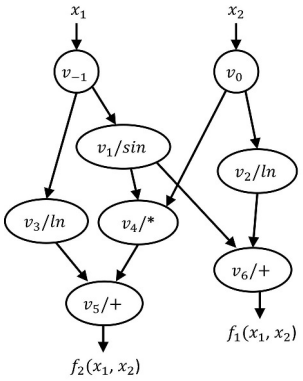


FIGURE 1. Computational graph of Eq.(1)

A. FORWARD MODE

Let $v_{i-n} = x_i, i \in [1, n]$ be the inputs, $v_i, i \in [1, l]$ be the intermediate variables, and $y_{m-i}, i \in [m-1, 0]$ be the outputs of functions $f_k, k = m - i$. A three-part notation is used to formalize the forward computation of derivatives.

In Table 1, relation $j \prec i$ means that v_i depends directly on v_j. In computational graph, this also means that node v_i is a direct successor of node v_j. φ_i is the function describes an elementary operation. $u_i = (v_j)_{j \prec i}$ is the set of all predecessors of v_i. Let the input be a vector $\mathbf{x} = [x_1, \ldots, x_i, \ldots, x_n]$. We set $\dot{\mathbf{x}} = [0, \ldots, 1_i, \ldots, 0]$, then $\dot{y}_{m-i}$ must be the derivative $\partial f_k / \partial x_i$ we wanted to calculate by carrying out the procedure depicted in Table 1. Forward mode is quite straightforward and easy to understand. It is efficient for function $f : \mathbb{R} \mapsto \mathbb{R}^m$. But for function $f : \mathbb{R}^n \mapsto \mathbb{R}$ or $f : \mathbb{R}^n \mapsto \mathbb{R}^m$, the procedure in Table 1 must run n or $n \times m$ times and it is very inefficient. A reverse mode computation for the derivatives is preferred for such case.

TABLE 1. Forward mode computation.

v_{i-n}	$\equiv$	x_i	$i \in [1, n]$
$\dot{v}_{i-n}$	$\equiv$	$\dot{x}_i$	
v_i	$\equiv$	$\varphi_i(v_j)_{j \prec i}$	$i \in [1, l]$
$\dot{v}_i$	$\equiv$	$\sum_{j \prec i} \frac{\partial}{\partial v_j} \varphi_i(u_i) \dot{v}_j$	
y_{m-i}	$\equiv$	v_{l-i}	$i \in [m-1, 0]$
$\dot{y}_{m-i}$	$\equiv$	$\dot{v}_{l-i}$	

B. REVERSE MODE

In reverse mode, we think in an opposite way. The chain rule which is $\dot{y} = f'(x)\dot{x}$ in forward mode is changed to a dual form $\bar{x} = \bar{y}f'(x)$ in reverse mode. We explain it in a graphic way: it means that the derivative of function to node v_i can be expressed as multiplication of adjoint of its successor and derivative of its successor to itself. Adjoint means $\bar{v}_i = \partial y / \partial v_i$. Therefore, the derivative evaluation procedure is listed in Table 2.

This procedure consists of two parts. The first part evaluates the value of the function in forward mode while the second part evaluates the derivatives of functions in reverse mode. If function $f : \mathbb{R}^n \mapsto \mathbb{R}^m$ is scalar-valued, then the

TABLE 2. Reverse mode computation.

v_i	$\equiv$	0	$i \in [n-1, l]$
v_{i-n}	$\equiv$	x_i	$i \in [1, n]$
v_i	$\equiv$	$\varphi_i(v_j)_{j \prec i}$	$i \in [1, l]$
y_{m-i}	$\equiv$	v_{l-i}	$i \in [0, m-1]$
$\bar{v}_{l-i}$	$\equiv$	$\bar{y}_{m-i}$	$i \in [0, m-1]$
$\bar{v}_j$	$+\equiv$	$[\bar{v}_i \frac{\partial \phi_i(u_i)}{\partial v_j}]_{j \prec i}$	$i \in [0, l]$
$\dot{x}_i$	$\equiv$	$\bar{v}_{i-n}$	$i \in [1, n]$

derivatives can be figured out just one time. Otherwise, this procedure should be run m times by providing $\bar{y}$ with $\mathbf{e}_i$ at one time. The final Jacobian matrix is obtained.

III. BUNDLE ADJUSTMENT

A. BASIC PRINCIPLE

Before applying automatic differentiation to bundle adjustment, we first demonstrate the basis of bundle adjustment. In computer vision, 3D points in space are projected onto the 2D image plane of cameras(see Fig. 2). Usually, if we want to build a 3D model of a scene, a series of camera imaging are needed to make the triangulation available [13]. Let $\mathbf{P}_j$ be the projection matrix of the jth camera in the imaging sequence and $\mathbf{X}_i$ be the ith point of the scene. The projection matrix is an arbitrary homogeneous 3×4 matrix having rank 3.

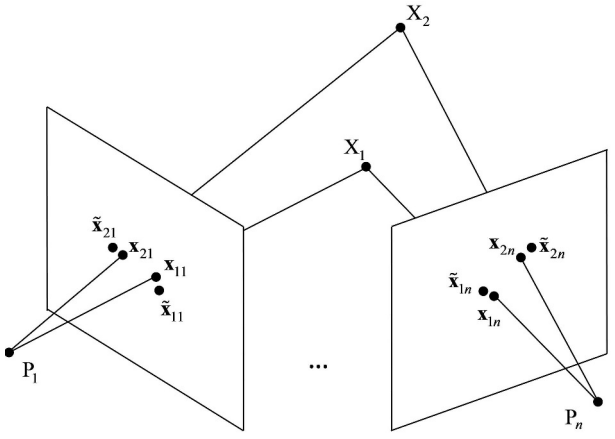


FIGURE 2. Perspective projection of 3D point X on image point x. Note that x̃ is the actual measured image point.

It can be further decomposed as $\mathbf{K}[\mathbf{R}|\mathbf{t}]$, where $\mathbf{K}$ is 3×3 upper triangular intrinsic matrix of the camera, $\mathbf{R}$ and t are 3×3 rotation matrix and 3×1 translation vector which are referred to as the camera's extrinsic orientation parameters. The total numbers of cameras and 3D points are m and n respectively. The 2D projection of point $\mathbf{X}_i$ onto $\mathbf{P}_j$ is

$$\lambda \mathbf{x}_{ij} = \mathbf{P}_j \mathbf{X}_i. \tag{2}$$

Let projection error be $\|\mathbf{x}_{ij} - \tilde{\mathbf{x}}_{ij}\|$. For BA, the intrinsic matrix $\mathbf{K}$ can be known and fixed, but $\mathbf{R}$ and $\mathbf{t}$ should be tuned to get better projection errors, and so are the 3D point coordinates. Reconstruction problem also referred to as structure from motion(SfM) estimation problem which is used to estimate extrinsic parameters of cameras and coordinates of 3D points under the situation that image points of multiple views are known. Due to the noise generated by feature point extraction and reconstruction algorithms, the estimated parameters are not optimal. So bundle adjustment algorithm can come in handy to reduce the errors. BA problem is usually solved by trust region approaches such as Levenberg-Marquardt, Dogleg and two dimensional subspace minimization. Both these methods need to compute the Jacobian matrix to obtain steepest-descent step and Newton step. We let estimated projection be $\tilde{\mathbf{x}} = f(\mathbf{p})$, where $\mathbf{p}$ is the combination of camera parameters and 3D point coordinates. In each iteration of the non-linear optimization method-LM, a series of vectors $\{\mathbf{p}_1, \mathbf{p}_2, \dots\}$, can converge towards a local minimum that minimizes the quantity

$$\| \mathbf{x} - f(\mathbf{p} + \delta_p) \| \approx \| \mathbf{x} - f(\mathbf{p} - \mathbf{J}\delta_p) \| = \| \epsilon - \mathbf{J}\delta_p \| \tag{3}$$

where $\mathbf{J}$ is the Jacobian matrix $\partial f(\mathbf{p})/\partial \mathbf{p}$. The solution searching step δ_p can be solved by so-called augmented normal equation:

$$(\mathbf{J}^T\mathbf{J} + \mu\mathbf{I})\delta_p = \mathbf{J}^{\mathbf{T}}\epsilon \tag{4}$$

where $\mathbf{I}$ is an identity matrix. Let $\mathbf{B} = \mathbf{J}^T\mathbf{J} + \mu\mathbf{I}$, the damping term μ enables $\mathbf{B}$ to be positive definite. Solving Eqn. 4 then becomes solving a positive definite system.

B. ABOUT THE JACOBIAN MATRIX

In a large scale optimization problem, such as BA, millions of parameters including cameras and 3D points may exist. We split these parameters into two or three groups. The first group is a set of 3D points, denoted as $\{\mathbf{a}_1, \cdots, \mathbf{a}_i, \cdots, \mathbf{a}_L\}$. The second group is a set of external parameters of cameras, denoted as $\{\mathbf{b}_1, \cdots, \mathbf{b}_i, \cdots, \mathbf{b}_M\}$. There may exist the third group containing the intrinsic parameters of cameras, and denoted as $\{\mathbf{c}_1, \cdots, \mathbf{c}_i, \cdots, \mathbf{c}_N\}$. If the third group is fixed, then automatic differentiation is not needed for this group. If all images are taken from the same camera, then $N = 1$. For any image point, we have $\mathbf{x}_{ijk}^T = \mathbf{f}([\mathbf{a}_i^T, \mathbf{a}_j^T, \mathbf{a}_k^T]^T)$, which means the projection of the ith 3D point on the jth camera position, and corresponding to the kth camera intrinsic parameter. In BA problem, a uniform function $\mathbf{F}$ can be used to demonstrate the relations between cameras, 3D points and image points. First, put all parameters together and form a parameter vector as

$$\mathbf{p} = [\mathbf{a}_1^T, \cdots, \mathbf{a}_i^T, \cdots, \mathbf{a}_L^T, \mathbf{b}_1^T, \cdots, \mathbf{b}_j^T, \cdots, \mathbf{b}_M^T, \mathbf{c}_1^T, \cdots, \mathbf{c}_k^T, \cdots, \mathbf{c}_N^T]^T \tag{5}$$

and let all image points to be the result vector as

$$\mathbf{x} = [\mathbf{x}_{111}^T, \mathbf{x}_{121}^T, \cdots, \mathbf{x}_{ijk}^T]^T \tag{6}$$

which is caused by function $\mathbf{F}$. We assume that camera intrinsic parameters are known and fixed. So $\mathbf{x}_{ijk}$ can be simplely denoted as $\mathbf{x}_{ij}$. Note that the number of image points in $\mathbf{x}$ is not $L \times M \times N$. Because not all 3D points have their

corresponding image points on all cameras. Then, we form the function as $\mathbf{x} = \mathbf{F}(\mathbf{p})$. This is an extremely complicated non-linear function. But we can obtain large sparsity when differentiate it with respect to parameter vector $\mathbf{p}$ and get large scale sparse Jacobian matrix J. This is due to the fact that there are a large number of zero blocks:

$$\begin{cases} \dfrac{\partial \mathbf{x}_{ij}^T}{\partial \mathbf{a}_u} = 0 \quad u \neq i \\ \dfrac{\partial \mathbf{x}_{ij}^T}{\partial \mathbf{b}_v} = 0 \quad v \neq j \end{cases} \tag{7}$$

Let $\mathbf{A}_{ij} = \partial \mathbf{x}_{ij}/\partial \mathbf{a}_i$, $\mathbf{B}_{ij} = \partial \mathbf{x}_{ij}/\partial \mathbf{b}_j$. They are matrix blocks with size 2×6 and 2×3 for example. The exact form of Jacobian matrix $\mathbf{J}$ consists of a large number of $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$, which is shown in Eq. (8).

$$\mathbf{J} = \begin{bmatrix} \mathbf{A}_{11} & 0 & 0 & \mathbf{B}_{11} & 0 & 0 \\ 0 & \dots & 0 & \dots & 0 & 0 \\ 0 & 0 & \mathbf{A}_{1m} & \mathbf{B}_{1m} & 0 & 0 \\ \dots & 0 & 0 & 0 & \dots & 0 \\ 0 & \dots & 0 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 & \dots & 0 \\ \mathbf{A}_{n1} & 0 & 0 & 0 & 0 & \mathbf{B}_{11} \\ 0 & \dots & 0 & 0 & 0 & \dots \\ 0 & 0 & \mathbf{A}_{nm} & 0 & 0 & \mathbf{B}_{1m} \end{bmatrix} \tag{8}$$

Sparse matrix $\mathbf{J}$ is viewed in block form. The size of its block rows is the number of image points. The block row that corresponds to $\mathbf{x}_{ij}^T$ contains the derivatives of $\mathbf{x}_{ij}^T$ with respect to parameter $\mathbf{p}$, but only $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ are non-zero. This form of Jacobian matrix $\mathbf{J}$ enables us only to compute and store $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$.

IV. CLAD: AN IMPLEMENTATION OF AUTOMATIC DIFFERENTIATION

In this section, we implement a library of parallelized automatic differentiation called CLAD for "large-small" problems with OpenCL [14], [15] and hardware acceleration support. "large-small" means that one single computational graph is small, but there are a large number of such computations. Just like in bundle adjustment, the computation of derivatives of a single image point is relatively small, while there are millions of image points need to be addressed. The CLAD library is well parallelized and consists of two parts. The first part generates computational graph and sequence on the host side in the framework of OpenCL. Template and function overloading features [16] in C++ are utilized for the construction of computational graph. Another part executes the generated computational sequences in parallel by feeding proper data to the device side.

A. HOST SIDE PROGRAMMING

In order to simplify the usage of this library from user's viewpoint, the construction of functions should be in an intuitive and convenient way just like writing it with the original C/C++ style language. An example of the function mentioned in Section II can be written as follows.

```
DOUBLE x1,x2;
DOUBLE f1=ln(x1)+x1*x2-sin(x1)
DOUBLE f2=x1*x2
```

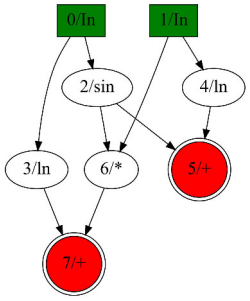
To obtain such simplicity, some crucial tasks should be done. First, structure *ADV_data* is introduced and defined as

```
template<typename T>
struct ADV_Data {
  OpType op;
  shared_ptr<ADV_Data<T>> arg[2];
  T val;
  T dval;
  int id;
}
```

This structure represents a node in the computational graph and a variable in programming. *OpType* is an enumeration type that indicates the operation of the node. In addition, we add *CONST* in it and regard a const value as a node, and also add *PLACEHOLDER* to indicate which node is an input variable. For each node, we assigned a unique *id* to it starting from zero. Wrapper class *ADV* is designed as the final variable representation for writing a mathematical formula and is defined as follows.

```
template<typename T>
class ADV {
public:
  shared_ptr<ADV_Data<T>> ADVptr;
  ADV();
  ADV(shared_ptr<ADV_Data<T>> ptr);
  ADV(const ADV &adv);
  ADV(const T val);
  ADV<T>& operator=(const ADV<T> &rhs);
  ADV<T>& operator=(const T &val);
}
```

The purpose of using shared_ptr to create node *ADV_Data* dynamically is to make sure that *ADV* variables can across over functions in C/C++ to implement the structure of functions in mathematics. As an example, *dot3(ADV<T> *a,*




```
Node list : 5 7 2 4 3 6 0 1
Reverse   : 5 7 4 3 6 1 2 0
Forword   : 0 1 2 3 4 5 6 7
```

FIGURE 3. Computational graph, forward and reverse computational sequences generated by CLAD for Eq. (1).

*ADV <T> *b)* implement inner product for 3D vectors and returns an *ADV* variable can capture all the structure of operations.

All of the constructors and assignment operators in this class insure correct variable generation. All of the operations between variables are implemented out of the class by function or operator overloading. Each overloading creates a new variable that describes the operation of its predecessors. The overloaded operator of the addition operation is written as follows.

```
template<typename T>
ADV<T> operator+(const ADV<T> &x,
const ADV<T> &y)
{
  ADV<T> adv;
  adv()->val=y()->val+x()->val;
  adv()->op=ADD;
  adv()->var[0]=x.ADVptr;
  adv()->var[1]=y.ADVptr;
  return adv;
}
```

For double-float precision, we define *ADV<double>* as *DOUBLE* for short. This is used in most situations. By creating variables with different operations, an underlying computational graph of the object function is then constructed.

B. COMPUTATION SEQUENCE GENERATION

So far as computational graph is obtained, values and derivatives of functions can be evaluated according to it. For the purpose of parallelly computing values and derivatives of functions which are projections and Jacobian matrix blocks in BA problem, a computational sequence that consists of nodes in the correct order must be produced.

At first, a breadth-first search(BFS) is applied to generate a list of nodes. By virtue of *id* of every node, getting the forward sequence for function value computation is very easy. Since each node with a larger *id* number must be successor of nodes with lower *id* numbers. Then we simply get the forward sequence by sorting nodes with respect to their ids in ascending order. But for reverse mode sequence, we must execute a topological sorting [17] to solve the dependency problem(shown in Fig. 3). This algorithm is summarized in Algorithm 1. As long as all the computational sequences are generated, devices such as Multicore CPUs and GPUs can then be utilized for parallel computing by providing them computational sequences and input data.

C. DATA STRUCTURE

The organization of dataset for our CLAD system should be taken into acount. For bundle adjustment, parameters for projection including camera parameters and 3D points become input variables for CLAD. But one image point only corresponds to one camera and one 3D point. Most of the time, intrinsic parameters of a camera are out of consideration for bundle adjustment, while they still serve as input variables. In order to execute large scale automatic differentiation through parallel computing on CPU/GPU, The large number

Algorithm 1 Topological Sorting for Generating Reverse Computational Sequence

Require: Computational graph $G = (V, E)$
Ensure: An reverse computational sequence L

```
Initialize counter array C recording in-degrees of each node
for v in V do
    Let s, t be predecessors of v
    Increase the in-degrees of s, t in C
end for
while L is not full do
    Find node v with zero in-degree
    Add it to L in the back
    Let s, t be predecessors of v
    Decrease the in-degrees of s, t
end while
```

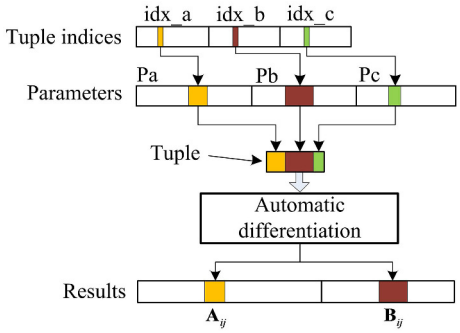


FIGURE 4. A BA example of data structure used in CLAD. There are three parameter groups. The third group is fixed, which means its corresponding derivatives are not needed.

of parameters must match the computational sequences well, and shoule be processed on the device side efficiently. As we discussed before, derivatives that computed in BA are split into two parts:$\mathbf{A}_{ij}$, $\mathbf{B}_{ij}$, which are derivatives of image points with respect to cameras and 3D points. For generalization, all parameters can be split into K groups. The number of parameters in each group can be N_i, $i \in [0, K)$, and the dimension of a parameter in each group is D_i, $i \in [0, K)$. Then *tuples* can be formed by extracting parameters from each parameter group. Tuples are elementary units that the automatic differentiation machine should process, and have constraint that the sum of total dimensions must be in accord with the number of input nodes. Taking BA as an example, there are three parameter groups which are intrinsic parameters of cameras,external parameters of cameras and 3D point coordinates respectively. Extracting parameters from them forms a *tuple*. One thing that must be noted is that not all parameters need derivatives and can be treated as fixed. In OpenCL, one work item can handle a *tuple*, multiple tuple can be processed parallelly. *Tuple indices* are used to indicate that which parameters should be taken from each parameter group for each *tuple*. We can see the corresponding data structure in Fig. 4.

D. PARALLEL COMPUTING

OpenCL programing [18] consists of code of two sides. On the host side, some preparatory work should be done, including initialization, providing data and starting the kernels. On the device side, *kernels* are executed in parallel on OpenCL devices such as CPUs, GPUs and FPGAs. OpenCL devices include many compute units(CU) that correspond to workgroups when running kernels. Host side code specifies how many threads(which are also called work items) to run for a kernel code. Work items in the same workgroup run in Single Instruction Multiple Thread(SIMT) mode and share local memory. On-chip local memory is much faster than off-chip global memory, but its size is limited. When processing each *tuple*, there must be some memory space to store values and derivatives of each node in the graph. For tens of millions of *tuples*, the storage memory must be huge and unnecessary. Because only the values of output nodes and the derivatives of input nodes are needed to form the final results. Thus, we can exploit the private registers in each work item to store them temporarily. The size of private registers per each work item is much device-dependent. For AMD GPU, there may be 16384 128-bits registers per CU that can be used by work items within a CU. Taking into consideration that there are only tens of nodes within a graph, the values and derivatives of each node can be stored in private registers. By utilizing high-speed local memory and private register, we can avoid slow global memory access and boost the computing process. The data storage framework is shown in Fig. 5

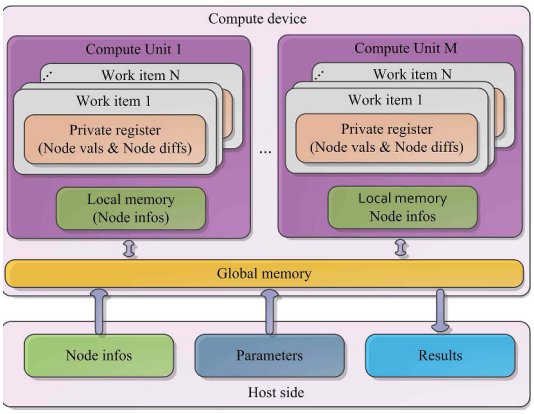


FIGURE 5. The data storage framework of CLAD under OpenCL. Node infomations and parameters are provided by Host. Node informations are cached in local memory, which include IDs, operator, operands, and corresponding computational sequences. Very fast private registers are used to cache value and derivative of each node.

The *kernel* code for automaticly processing each *tuple* is very trivial. First, evaluate the value of every node according to the forward sequence. Then evaluate the derivative of each node according to the reverse sequence. The values of output nodes(function values) can also be used to evaluate residual errors for almost all of the optimization algorithms. The primary pseudocode of the *kernel* is listed in Algorithm 2. Function *compute_vals* calculate the value of the current node, and *compute_diffs* calculate the derivatives of the current node with respect to its one or two predecessors. Index array fwIdx indicates the position of each node in the node list and is also a forward sequence.

TABLE 3. Test datasets.

Dataset	Cameras	3D points	2D points
Ladybug	1723	156502	14992
Trafalgar	257	65132	225911
Dubrovnik	356	226730	1255268
Venice	1778	993923	5001946
Rome	4585	1324582	9125125

Algorithm 2 Kernel Function for Automatic Differentiation

Require: Node list $\mathcal{NL}$, parameters $\mathcal{P}$, tuple indices $\mathcal{TI}$ forward sequence $\mathcal{FS}$, reverse sequence $\mathcal{RS}$, input nodes $\mathcal{IN}$, output nodes $\mathcal{ON}$.

Ensure: function values $\mathcal{V}$ and derivatives $\mathcal{D}$.

```
Initialize private register PV and PD to store values and derivatives of each node.
Copy NL, FS and RS to LNL, LFS and LRS ;
Copy part of P to LV according to TI
for id in FS do
    compute_vals(id, LNL, PV)
end for
Copy values of ON from PV to V
for id1 in ON do
    Clear PD to 0, except that PD[id1] = 1
    for id2 in RS do
        compute_diffs(id2, LNL, PV, PD)
    end for
    Copy derivatives of IN from PD to D
end for
```

V. APPLICATION OF CLAD IN BUNDLE ADJUSTMENT

A. FORMULATION OF PROJECTIONS

Applying CLAD to bundle adjustment can be straightforward. Before writing the projection formula using CLAD, the form of rotation should be specified. There are many ways to represent rotation in 3D space, such as rotation matrix, eular angle, quarternions and rodrigues parameters. The most used representations in bundle adjustment are quarternions and rodrigues parameters [19].

Quarternions are represented in the form: $\mathbf{q} = \langle s, \mathbf{v} \rangle$, and rotation transformation of a 3D point $\mathbf{X}_i$ under the coordinate frame of jth camera is

$$\mathrm{Rot}(\mathbf{X}_i) = \tilde{\mathbf{q}}_j \cdot q(\mathbf{X}_i) = \tilde{\mathbf{q}}_j \oplus q(\mathbf{X}_i) \oplus \tilde{\mathbf{q}}_j^{-1} \tag{9}$$

where $q(\mathbf{X}_j)$ turns $\mathbf{X}_i$ into a quaternion form $\langle 0, \mathbf{X}_i \rangle$, '$\cdot$' represents rotation transformation, and '$\oplus$' represents a Hamilton product.

For rodriguez form, rotation is represented by an unitary rotation axis vector $\mathbf{k}$ and a rotation angle θ , a point $\mathbf{X}_i$ can

be rotated by the following rodriguez formula

$$\mathrm{Rot}(\mathbf{X}_i) = \mathbf{X}_i cos\theta + (\mathbf{k} \times \mathbf{X}_i) sin\theta + \mathbf{k}(\mathbf{k} * \mathbf{v}(1 - cos\theta)) \tag{10}$$

While we can combine rotation axis and rotation angle together and express rotation by a non-unitized vector. The whole projection formula using rodrigues representation is written as

$$\begin{aligned} \mathbf{X}' &= \mathrm{Rot}(\mathbf{X}) + \mathbf{t} \\ \mathbf{x} &= f \cdot \left[\frac{\mathbf{X}'(1)}{\mathbf{X}'(3)}, \frac{\mathbf{X}'(2)}{\mathbf{X}'(3)} \right]^T \end{aligned} \tag{11}$$

where f is the focal length, $\mathbf{t}$ is the cartesian translation vector between the origins of the camera coordinates frame and the world coordinate frame. This equation can be translated into C++ style language as follows.

```
vector<DOUBLE> proj_func()
{
  DOUBLE theta2=dot(angle_axis, angle_axis);
  DOUBLE theta=sqrt(theta2);
  DOUBLE costheta=cos(theta);
  DOUBLE sintheta=sin(theta);
  DOUBLE theta_inverse=1.0/theta;
  DOUBLE w[3]={angle_axis[0]*theta_inverse,
  angle_axis[1]*theta_inverse,
  angle_axis[2]*theta_inverse};
  DOUBLE w_cross_pt[3];
  cross(w,pt3D,w_cross_pt);
  DOUBLE tmp=dot(w,pt3D)*(1.0-costheta);
  DOUBLE tmp3D[3];
  tmp3D[0]=pt3D[0]*costheta+
  w_cross_pt[0]*sintheta+w[0]*tmp;
  tmp3D[1]=pt3D[1]*costheta +
  w_cross_pt[1]*sintheta+w[1]*tmp;
  tmp3D[2]=pt3D[2]*costheta +
  w_cross_pt[2]*sintheta+w[2]*tmp;
  tmp3D[0]=tmp3D[0]+transl[0];
  tmp3D[1]=tmp3D[1]+transl[1];
  tmp3D[2]=tmp3D[2]+transl[2];
  x=focal*tmp3D[0]/tmp3D[2];
  y=focal*tmp3D[1]/tmp3D[2];
  vector<DOUBLE> outputs;
  outputs.push_back(x);
  outputs.push_back(y)
  return outputs
}
```

The constructed intrinsic computational graph is shown in Fig. 6, which then is converted to computational sequences for parallel computing of the Jacobian matrix blocks $\mathbf{A}_{ij}$ and $\mathbf{B}_{ij}$ under OpenCL programming framework.

B. EVALUATION AND RESULTS

We evaluate our proposed CLAD on several large-scale bundle adjustment datasets which are listed in Table 4 and are known as bundle adjustment in the large(BAL) problem [20]. For comparison, Google's Ceres Solver is introduced. Ceres Solver is an open source library for solving large optimization problems including bundle adjustment. The automatic differentiation used in Ceres Solver can be accelerated by using OpenMP which is a multi-threading framework that allows the compiler to generate code for task and data parallelism.

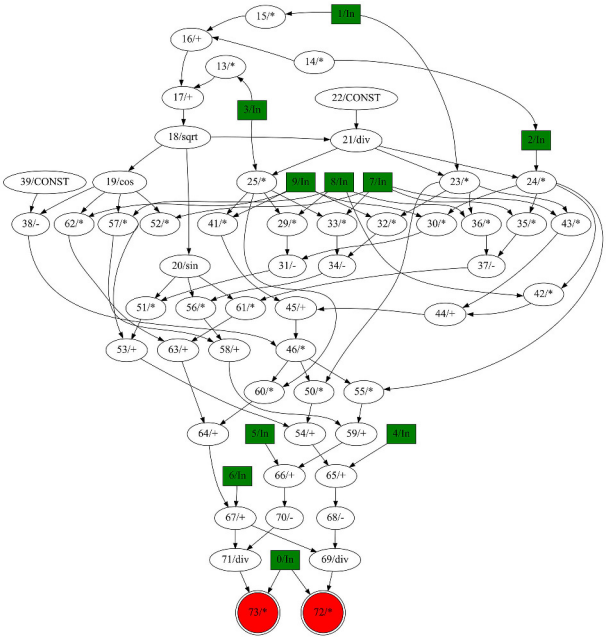


FIGURE 6. The final computational graph of projection formula generated by CLAD. The green nodes are input parameters including camera's paramters and coordinates of 3D point. the red nodes are coordinates of image point as outputs.

TABLE 4. Runing times(s) of CLAD applied to bundle adjustment datasets.

Dataset	CLAD(i7)	Ceres(i7)	CLAD(E5)	Ceres(E5)	CLAD(GPU)
Ladybug	0.074	0.277	0.052	0.190	0.114
Trafalgar	0.026	0.107	0.018	0.064	0.039
Dubrovnik	0.136	0.465	0.094	0.341	0.210
Venice	0.546	1.936	0.383	1.400	0.864
Rome	1.056	3.377	0.687	2.623	1.626

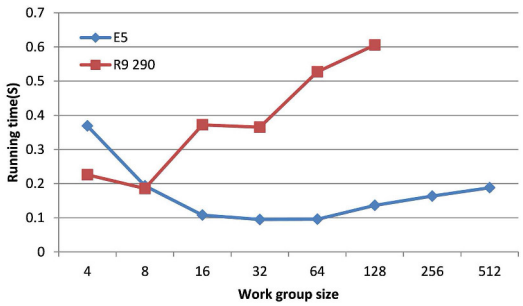


FIGURE 7. Performance with different size of work group.

At first, due to the reason that there are many branching in the kernel, work group size should be considered. By testing Dubrovnik dataset on Intel XEON E5 CPU and AMD R9 290 GPU with different work group size, the result is shown in Fig. 7. A better performance of the CPU is achieved with work group size 32. While in the case of GPU, work group size 8 is much better. The choice of work group size is uncertain. Because there are all kinds of devices with all kinds of hardware architectures. We usually leave this decision to the system.

We run testing on two hardware platforms. One is compatible desktop PC. The desktop PC uses an Intel XEON E5 2643 v2 CPU with 8G DDR3 RAM. An AMD R9 290

graphics card is installed on the desktop PC to evaluate the performance on GPU. Another one is a mobile workstation with i7 4700MQ CPU and 8G DDR3 RAM. OpenCL 2.0 is supported by both the CPUs and GPU. Compared to previous versions of OpenCL, OpenCL 2.0 introduced SVM(Shared Virtual Memory) which enables direct data sharing between host and devices and avoids redundant data transfer operation. For OpenMP used in Ceres Solver, we set the number of threads to the number of cores of each CPU to achieve maximum performance.

The evaluation results are shown in Table 4. Regardless which CPU is used, our proposed CLAD achieves a better performance. CLAD runs about 3.6 times faster than Ceres Solver. We get unfavorable results when test on GPU. Even though AMD R9 290 has 40 CUs, but it works at 1Ghz which is lower than 2.7Ghz and 3.2Ghz of i7 and E5 respectively. Furthermore, the code of computing the forward and reverse sequence of automatic differentiation involves many branches. GPUs allow branching, but usually with much performance penalty. However, CPUs have more complex ALU and better branch prediction which are necessary for the proposed implementation of automatic differentiation.

VI. CONCLUSION

In this article, We have first demonstrated how an automatic differentiation system works by introducing forward mode and reverse mode computation. For functions with n real-value parameters, the reverse mode for derivatives is more efficient than forward mode, which is more suitable for the computation of large scale sparse Jacobian matrix in bundle adjustment problems. We implement an automatic differentiation library called CLAD based on template and overloading futures in C++. In order to parallelize the computation in the framework of OpenCL, we generate forward and reverse computational sequences from computational graph. This library enables us to write down the function formula elegantly and then evaluate these derivatives rapidly even when the provided input parameters are very large. We evaluate this library on large scale bundle adjustment datasets. The result shows that our implementation runs 3.6 times faster than Ceres Solver which utilizes OpenMP parallel programming model on the same hardware platform.

REFERENCES

[1] B. Triggs, P. F. Mclauchlan, R. I. Hartley, and A. W. Fitzgibbon, "Bundle adjustment—A modern synthesis," in *Proc. Int. Workshop Vis. Algorithms, Theory Pract.*, 1999, pp. 298–372.

[2] Y. Gong, D. Meng, and E. J. Seibel, "Bound constrained bundle adjustment for reliable 3D reconstruction," *Opt. Express*, vol. 23, no. 8, pp. 10771–10785, 2015.

[3] M. I. A. Lourakis and A. A. Argyros, "SBA: A software package for generic sparse bundle adjustment," *ACM Trans. Math. Softw.*, vol. 36, no. 1, p. 2, 2009.

[4] M. I. A. Lourakis and A. A. Argyros, "Is Levenberg–Marquardt the most efficient optimization algorithm for implementing bundle adjustment?" in *Proc. 10th IEEE Int. Conf. Comput. Vis.*, vol. 2. Oct. 2005, pp. 1526–1531.

[5] C. Cadena *et al.*, "Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age," *IEEE Trans. Robot.*, vol. 32, no. 6, pp. 1309–1332, Dec. 2016.

[6] H. Strasdat, J. M. M. Montiel, and A. J. Davison, "Real-time monocular SLAM: Why filter?" in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2010, pp. 2657–2664.

[7] H. Strasdat, J. Montiel, and A. J. Davison, "Visual SLAM: Why filter?" *Image Vis. Comput.*, vol. 30, no. 2, pp. 65–77, 2012.

[8] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. Philadelphia, PA, USA: SIAM, 2008.

[9] A. Griewank, D. Juedes, and J. Utke, "Algorithm 755: ADOL-C: A package for the automatic differentiation of algorithms written in C/C++," *ACM Trans. Math. Softw.*, vol. 22, no. 2, pp. 131–167, 1996.

[10] R. J. Hogan, "Fast reverse-mode automatic differentiation using expression templates in C++," *ACM Trans. Math. Softw.*, vol. 40, no. 4, pp. 1–16, 2014.

[11] E. I. Sluşanschi and V. Dumitrel, "ADiJaC—Automatic differentiation of Java classfiles," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 1–33, 2016.

[12] S. Agarwal and K. Mierle. *Ceres Solver*. Accessed: Feb. 3, 2018. [Online]. Available: http://ceres-solver.org

[13] N. Snavely, S. M. Seitz, and R. Szeliski, "Modeling the world from Internet photo collections," *Int. J. Comput. Vis.*, vol. 80, no. 2, pp. 189–210, 2007.

[14] D. R. Kaeli, P. Mistry, D. Schaa, and D. P. Zhang, *Heterogeneous Computing With OpenCL 2.0*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann, 2015.

[15] A. Munshi, B. Gaster, T. G. Mattson, J. Fung, and D. Ginsburg, *OpenCL Programming Guide*, 1st ed. Reading, MA, USA: Addison-Wesley, 2011.

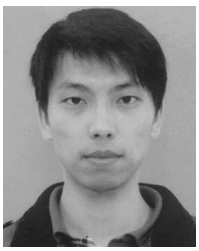
[16] E. Phipps and R. Pawlowski, "Efficient expression templates for operator overloading-based automatic differentiation," in *Recent Advances in Algorithmic Differentiation*. Berlin, Germany: Springer, 2012, pp. 309–319.

[17] K. Kristensen, A. Nielsen, C. W. Berg, H. Skaug, and B. M. Bell, "TMB: Automatic differentiation and Laplace approximation," *J. Statist. Softw.*, vol. 70, no. 5, pp. 1–21, 2016.

[18] T. Shimobaba, T. Ito, N. Masuda, Y. Ichihashi, and N. Takada, "Fast calculation of computer-generated-hologram on AMD HD5000 series GPU and OpenCL," *Opt. Express*, vol. 18, no. 10, pp. 9955–9960, 2010.

[19] J. S. Dai, "Euler–Rodrigues formula variations, quaternion conjugation and intrinsic connections," *Mech. Mach. Theory*, vol. 92, pp. 144–152, Oct. 2015.

[20] S. Agarwal, N. Snavely, S. M. Seitz, and R. Szeliski, "Bundle adjustment in the large," in *Proc. Eur. Conf. Comput. Vis.*, 2010, pp. 29–42.



YAN SHEN was born in Changde, China, in 1985. He received the B.Sc. degree (Engineering) from Hunan Normal University, Changsha, China, in 2008. He is currently pursuing the Ph.D. degree at the College of Electrical and Information Engineering, Hunan University. His research interests include robotics, computer vision, and image processing.

YUXING DAI was born in Changsha, China, in 1965. He received the Ph.D. degree in control theory and control engineering from Central South University, Changsha, in 2003. From 1995 to 2001, he was a Professor with the Department of Electronic Engineering, Hunan Normal University. From 2001 to 2011, he was a Professor and the Director with the Department of Electronic Science and Technology, College of Electrical and Information Engineering, Hunan University, Changsha. Since 2011, he has been a Professor and the Dean with the College of Physics and Electronic Information Engineering, Wenzhou University, Wenzhou, China. He is also the Director of the National-Local Joint Engineering Laboratory of Digitalize Electrical Design Technology, Wenzhou University. He has headed over 20 national, provincial, and industrial projects. He has authored or co-authored 5 books, over 100 journal and conference articles, and over 10 inventions. He holds 12 patents. His research interests include modeling, control and optimization of power electronic systems, microgrids and computer numerical control machine tools, computational intelligence, and engineering practice. He was a recipient of 10 ministerial and provincial science and technology progress awards.