文章编号:1671-4598(2019)05-0155-05 DOI:10.16526/j.cnki.11—4762/tp.2019.05.035 中图分类号:O246;TP311.1 文献标识码:A

基于 OpenCL 的自动微分并行实现及其应用

叶爱芬[1], 王 环[2], 沈 雁[3]
(1. 浙江东方职业技术学院 电气自动化研究室, 浙江 温州 325035;
2. 温州大学 电气数字化设计技术国家地方联合工程实验室, 浙江 温州 325035;
3. 湖南大学 电气与信息工程学院, 湖南 长沙 410082)

摘要: 针对如光束平差这样的大规模优化问题, 实现基于 OpenCL 的并行化自动微分; 采用更有效的反向计算模式, 实现对多参数函数的导数计算; 在 OpenCL 框架下, 主机端完成 C/C++形式的函数构建以及基于拓扑排序的计算序列生成, 设备端按照计算序列完成函数值以及导数的并行计算; 测试结果表明, 将实现的自动微分应用于光束平差的雅可比矩阵计算后, 相比于采用 OpenMP 的 Ceres Solver, 运行速度提高了约 3.6 倍。

关键词: 自动微分; 并行计算; OpenCL

Automatic Differentiation Based on OpenCL Parallel Computing and Its Application

Ye Aifen[1], Wang Huan[2], Shen Yan[3]
(1. Zhejiang Dongfang Polytechnic, Wenzhou 325000, China;
2. National—Local Joint Engineering Laboratory of Electrical Digital Design Technology, Wenzhou University, Wenzhou 325035, China;
3. College of Electrical and Information Engineering, Hunan University, Changsha 410082, China)

Abstract: A parallelized implementation of automatic differentiation that derives from the problem of bundle adjustment is proposed, which is based on OpenCL parallel computing framework. Reverse mode of automatic differentiation is more efficient to compute the derivatives of functions with multiple parameters, which is the case of computing the Jacobian matrix in bundle adjustment problem. Under the framework of OpenCL, C/C++ style function construction and topological sorting based computational sequence generation are implemented on the host side. On the device side, function values and derivatives are computed in parallel according to computational sequence. Large scale bundle adjustment datasets are used to evaluate the proposed implementation. The result shows that our implementation runs about 3.6 times faster than Ceres Solver which utilizes OpenMP parallel programming model.

Keywords: automatic differentiation; parallel computing; OpenCL; bundle adjustment

0 引言

在很多的数值优化[1]方法中都会涉及到对函数导数, 雅可比矩阵的计算。在传统上, 这些计算都是通过手工完成的。手工的计算方式对于很多小型的问题来说是比较简单可行的。但是对于有众多参数的复杂函数来说, 手工计算其导数将是一个庞杂且容易出错的过程。随着现代计算机的迅速发展, 对导数的计算都转而采用计算机来实现。当前有三种通过计算机计算导数的方式。第一种为使用有限差分近似的数值微分方式。该方式简单实用, 但存在着较大的舍入和截取误差。第二种方式为是使用符号计算的符号微分, 该方式可以得到精确的封闭形式的求导结果。但存在一个严重的问题, 那就是随着函数的复杂度的增加, 符号微分产生的符号表达式会呈指数级的增长。这就是所谓的表示式膨胀问题。而第三种方式则是本文所讨论的自动微分[5-6]方式。自动微分克服了前两种方式的缺点, 会是数值计算方面主要的导数计算方式。

自动微分可以很有效地应用于计算机视觉中的大规模光束平差[2-3] (Bundle Adjustment, BA) 优化问题中。光束平差通过不断地优化 2D 点的误差来优化三维重建中相机和 3D 点的位置。采用的方法通常有 Levenberg—Marquardt (LM)、Dog—Leg (DL) 等[4]。这些方法都需要计算投影函数 $x=f(X)$ 的导数来形成雅可比矩阵, 其中 x 为 2D 点, X 为其对应的 3D 点。由于 3D 点的数量众多, 可达到上百万级。因此, 对这一过程进行自动微分的并行化计算可以大大提高计算的速度。

自动微分有前向模式和反向模式两种。对于含有多个参数的函数, 从 计算效率上来说更倾向于采用后向计算模式。目前存在这些通用的用于计算自动微分的软件, 如 ADOL—C[7]、CPPAD 以及 Ceres Solver。对于用于光束平差的 Ceres Solver, 其实现了一个基于 OpenMP 的自动微分。在本文中, 则基于开放的 OpenCL 构架, 提供了一个更高效的并行化自动微分实现。

1 自动微分的原理

相对于封闭形式求导结果或者近似的数值求导结果,

收稿日期:2018-10-17; **修回日期**:2018-11-28。
基金项目:浙江省自然科学基金重点项目(LZ16E050002)。
作者简介:叶爱芬(1982-),女,浙江永嘉人,硕士,讲师,主要从事机器学习,自动控制方向的研究。

自动微分可以获得无截断误差的数值结果。这很大程度上得益于链式法则和计算机编程模式的相结合。每一个函数可以被分解为基本运算符的组合，其中基本运算符包括加、减、乘、除等二元算术运算符，以及像三角函数和指数函数等在内的超越函数。因此函数可以采用计算图的形式来表示。例如考虑如下的函数：

$$\begin{aligned} f_1(x_1,x_2) &= ln(x_1)+\sin(x_2)-x_2^2 \\ f_2(x_1,x_2) &= ln(x_1)*\sin(x_2) \end{aligned} \tag{1}$$

式中，可以视为一个向量函数 $f:R^2\mapsto R^2$。其分解得到的计算图见 0。图中每个运算都由一个带编号的节点表示。可以通过前向和反向两个模式来计算 f 关于 x 的导数。

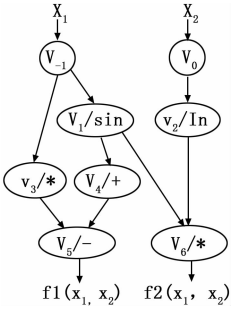

图 1 公式（1）的计算流程图

1.1 前向计算模式

令 $v_{i-n}=x_i, i\in[1,n]$ 为输入，$v_i, i\in[1,l]$ 为中间变量，$y_{m-i}, i\in[m-1,0]$ 为函数 $f_k, k=m-i$ 的输出。一个三阶段的记法可以用来对导数的前向计算进行形式化，自动微分前向计算模式的计算过程如下所示：

$$\begin{aligned} v_{i-1} &\equiv x_i \\ \dot{v}_{i-n} &\equiv \dot{x}_i \end{aligned}, \quad i\in[1,n] \tag{2}$$

$$\begin{aligned} v_{i-1} &\equiv \varphi_i(v_j)_{j<i} \\ \dot{v}_{i-n} &\equiv \sum_{j<i}\frac{\partial}{\partial v_j}\varphi_i(u_i)\dot{v}_j \end{aligned}, i\in[1,n] \tag{3}$$

$$\begin{aligned} y_{m-i} &\equiv v_{l-i} \\ y_{m-i} &\equiv \dot{v}_{l-i} \end{aligned}, \quad i\in[m-1,0] \tag{4}$$

式中，关系 $j<i$ 表示 v_i 与 v_j 是直接相关的。在计算图也表示节点 v_i 是 v_j 的一个直接后继。φ_i 是描述基本运算的函数。$u_i=(v_j)_{j<i}$ 表示 v_i 的所有前驱的集合。令所有输入为一个向量 $x=[x_1,\cdots x_i,\cdots,x_n]$。若将输入的导数设定为 $\dot{x}=[0,\cdots,1_i,\cdots,0]$，则 $\dot{y}_{m-i}$ 就为希望计算得到的导数 $\partial f_k/\partial x_i$。前向计算模式比较简洁易懂，并且对于 $f:R\mapsto R^m$ 这样的标量函数来说，其计算是非常有效的。因为输入参数只有一个，那么只需要令 $\dot{x}=[1]$ 便可以计算出所有的导数值。但是对于像 $f:R^n\mapsto R$ 或 $f:R^n\mapsto R^m$ 这样的向量函数，双述的计算过程必须运行 n 次或 $n\times m$ 次。而反向的计算模式则可以避免这个问题，提高计算的效率。

1.2 反向计算模式

在反向计算模式中，采用的是一种相反的计算思路。用于前向计算中的链式法则 $\dot{y}=f'(x)\dot{x}$ 在反向计算中被转变为对偶形式 $\bar{x}=\bar{y}f'(x)$。可以从计算图的角度来理解它：该对偶形式意味着函数对节点 v_j 的导数可以表示成函数对其后继的导数和其后继对该节点自身的导数的乘积。函数对节点 v_j 各个后继的导数记为 $\bar{v}_i$。反向计算模式下的导数计算过程如表 1 所示：

表 1 自动微分反向计算模式的计算过程

计算过程	
$v_i\equiv 0$	$i\in[n-1,l]$
$v_{i-n}\equiv x_i$	$i\in[1,n]$
$v_i\equiv\varphi_i(v_j)_{j<i}$	$i\in[1,l]$
$y_{m-i}\equiv v_{l-i}$	$i\in[0,m-1]$
$\bar{v}_{l-i}\equiv\bar{y}_{m-i}$	$i\in[0,m-1]$
$y_{m-i}+\equiv\left[\bar{v}_i\frac{\partial\varphi_i(u_i)}{\partial v_j}\right]_{j<i}$	$i\in[0,l]$
$\bar{x}_i\equiv\bar{v}_{i-n}$	$i\in[0,n]$

该计算过程分为两部分。第一部分像正向计算模式一样计算出函数值。第二部分则反向地估计函数对所有输入参数的导数。对于函数 $f:R^n\mapsto R^m$，所示的计算过程只需要运行 m 次便可以获得所有的求导结果，从而得到最终的雅克布矩阵。

2 并行实现

本节中实现的基于 OpenCL[8-9] 的自动微分采用反向计算模式，可以用于“large－small”问题。所谓的“large－small”问题，就是单一函数的计算图并不复杂，但是却存在着大量的重复的计算。光束平差问题便是一个例子。由于大量的 3D 点和相机的存在，就需要计算大量的投影函数对 3D 点坐标和相机参数的导数，来形成最终的稀疏雅克布矩阵。基于 OpenCL 的自动微分的实现分为主机端和设备端两部分。在主机端运用了 C＋＋的函数重载和模板特性[10]来有效地生成计算图。而设备端则根据计算图并行地计算出求导结果。

2.1 主机端编程

为了简化使用的方式，对待求导函数的构建应该尽量的趋近于原生的 C/C＋＋风格的代码编写。式的函数可以写成如下的简洁形式：

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

要实现如此的函数构建的简单化，需要完成一些关键的任务。首先，定义一个用于描述节点的结构体：

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

在该结构体中，OpType 是一个用于指示运算符的枚举类型，指示该节点为其前驱进行某种运算的结果。除了常

规的运算之外，还引入“CONST”和“INPUT”来分别表示该节点是否为常量或者函数的输入参数。每个节点在生成时还被赋予一个唯一的 id 值。arg 指向该节点的一个或两个前驱节点，指示该节点运算符的操作数。dval 用于在求导过程中存放函数对该节点的导数值。

然后，实现了一个 Wrapper 类 ADV 来描述最终的数学形式上的变量，并用此来完成数学表达式的构建。ADV 类的定义如下：

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

采用智能指针来为每个 ADV 变量创建节点可以使得 ADV 变量能够不受作用域的限制，像数学形式上的函数那样完成代码中的对应函数。例如要实现一个 3 元素向量的点积，可以写为“dot3（ADV<T> * a，ADV<T> * b)”。该函数能返回一个新建的 ADV 变量来记录所有的运行结构。ADV 类中所有的构造函数和赋值运算符重载都用来确保变量的正确生成。所有的关于 ADV 变量间的运算都用过类定义外的友元的函数重载来实现。每个函数重载创建新 ADV 变量来记录这一关于其前驱的运算。例如加法的函数重载实现为：

```
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;
}
```

程序上的“$c=a+b$;”表达式便可以直接地表示数学上的 $c=a+b$。通过使用不同运算的函数重载来构造函数，函数所对应的计算图也同时被构建出来。另外对于大多数情况下使用的双精度浮点型来说，可以将 ADV<double>定义为 DVAR。

2.2　计算序列的生成

在构建函数以及对应的计算图之后，便可以据此运用反向计算模式来进行导数的计算。为了正确地计算各个节点的导数值，同时也是为了便于设备端的并行计算。需要将计算图转化为计算序列。

在反向计算模式下，计算序列又分为正向计算序列和反向计算序列，正向计算序列用于计算函数值，而反向计算序列则用于计算函数的导数值。由于每个节点的 id 在生成时满足后继节点的 id 值一定大于其前驱节点的 id 值。所以用于计算函数值的正向计算序列可以简单地取为各节点的升序排列。然而在确定反向计算序列时，必须考虑节点之间的依赖关系。如 0 所示，如果先从节点 7 开始处理，在处理节点 2 时，其所依赖的节点 5 的导数值还并没有被计算出来。

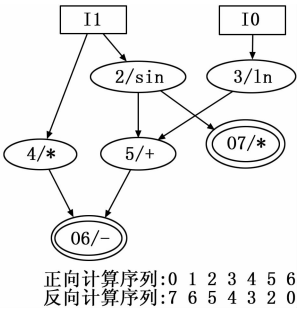

图 2　编程模式下的计算图及正向和反向计算序列

因此，采用拓扑排序来完成反向计算序列生成。所生成的反向计算序列中各节点满足相互间的依赖关系。拓扑排序的伪代码见算法 1：

算法 1：用于生成反向计算序列拓扑排序
输入：计算图 $G=(V,E)$
输出：反向计算序列 L
初始化计数数值 C，用于记录各节点的入度
for $v$in V do
　令 s,t 为 v 的前驱节点
　增加 s,t 在 C 中的入度
end for
while L 非满 do
　找到具有零入度的节点 v
　添加到 L 尾部
　令 s,t 为 v 的前驱节点
　减少 s,t 在 C 中的入度
end while

2.3　设备端并行计算

当在主机端完成计算序列的生成之后，便可以将需计算的函数的参数传入设备端，在设备端按照计算序列计算出函数值以及函数的导数值。

在 OpenCL 下，多个 kernel 函数并行地在如 CPU、GPU 或者 FPGA 设备上运行，其每个运行的实例称为一个工作项。其并行的数量取决于设备上计算单元的数量，以及计算单元上局部存储器大小等因素。主机端需要指定全局工作项的数量，其对应到需要进行求导的函数的数量。同时也可以指定局部工作项的数量，形成工作组。每个工作组中的工作项以 SIMT（Single Instruction Multiple Thread）的模式，并且共享一组数量有限的高速局部存储器。在像光束平差的应用中，所有的导数计算使用相同的计算图和序列。如果计算图的尺寸比较小，那么可以将生成的计算序列传输到局部存储器以访问提高速度。

用于执行每个实际求导过程的 kernel 函数相对比较简单。首先按照给定的正向计算序列和函数参数值计算出函数值。然后再按照反向计算序列依次计算每个节点的导数值。最后将作为输入节点的导数返回给主机端。该计算过程的伪代码见算法 2：

算法 2：计算导数的 Kernel 函数 for i=0 to size (forward _ seq) do

```
    compute _ val (node _ op, arg1 _ val, arg2 _ val)
  end for
  for i=0 to size (input _ args) do
    val _ out [base+i] =node _ val [i]
  end for
  for i=0 to size (funcs) do
    for j=0 to size (reverse _ seq)
    compute _ diff (node _ op, node _ val,
    arg1 _ diff, arg2 _ diff)
    end for
  for j=0 to size (input _ args) do
    diff _ out [base+i * N+j] =node _ diff [j]
    end for
  end for
```

3 应用实例

本节中实现的基于 OpenCL[8-9] 的自动微分采用反向计算模式，可以用本文实现的自动微分可以很容易地应用到光束平差问题中。首先对光束平差中所需的投影方程进行分析。设 3D 点齐次坐标和相机投影矩阵分别为 $\widetilde{X}$ 和 P。3D 点在相机上的投影则为 $\widetilde{x} = P\widetilde{X}$。然而 $P = K[R \mid t]$ 是一个 4×3 的矩阵，共有 9 个值，其中 K 为相机的内参，R 为 3×3 的旋转矩阵，t 为相机的位置。对于大量的相机来说，直接应用旋转矩阵是比较低效的。通常会采用四元数和 Rodrigues 参数[11]，它们与 RX 是基本等效的。X 为 $\widetilde{X}$ 的非齐次坐标。

四元数可以表示为 $q = < s, v >$。在相机坐标系下对 3D 点的旋转可以表示为：

$$Rot(X) = q \cdot q(X) = q \oplus q(X) \oplus q^{-1} \tag{5}$$

其中：$\oplus$ 为 Hamilton 积，$q(X)$ 则用于将 X 变为四元数形式 $< 0, X >$。在 Rodrigues 参数形式中，旋转表示为绕单位向量 k 的旋转，旋转角度为 θ。因此，X 的旋转在 Rodrigues 参数形式下表示为：

$$Rot(X) = X\cos\theta + (k \times X)\sin\theta + (1 - \cos\theta)(k * v)k \tag{6}$$

通过将旋转与位移的结合，可以得到 3D 点在相机上的投影。在光束平差中，视 3D 点坐标 X 和相机矩阵 $[R \mid t]$ 为投影 x 的参数。采用 Rodrigues 参数形式，编写如下的代码完成 x 相对于 X 和 $[R \mid t]$ 的计算图的构建：

```
DVAR x, y;
DVAR theta2=dot (angle _ axis, angle _ axis);
DVAR theta=sqrt (theta2);
DVAR costheta=cos (theta);
DVAR sintheta=sin (theta);
DVAR theta _ inverse=1. 0/theta;
DVAR w [3] = {  angle _ axis [0] * theta _ inverse,
      angle _ axis [1] * theta _ inverse,
   angle _ axis [2] * theta _ inverse
   };
DVAR w _ cross _ pt [3];
cross (w, pt3D, w _ cross _ pt);
DVAR tmp=dot (w, pt3D) * (1. 0-costheta);
DVAR 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];
```

其对应的计算图见图 3。该计算图随后被转化为计算序列，并伴随所有 3D 点坐标和相机参数送入设备的并行计算。计算出投影点坐标，以及投影点作为函数对 3D 点坐标和相机参数的导数。最终获得雅可比矩阵的每个块 A_{ij} 和 B_{ij}。A_{ij} 为投影点坐标 (x, y) 对 3D 点坐标 (X, Y, Z) 的导数。B_{ij} 为投影点坐标对向量化的相机参数 p 的导数。

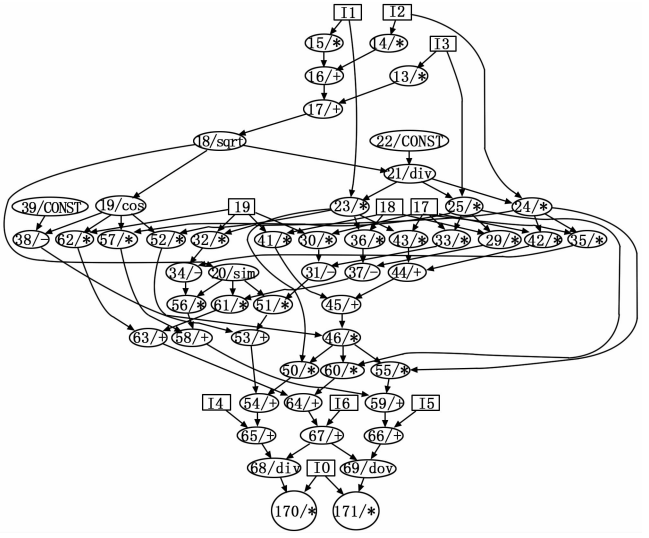

图 3 Rodrigues 参数形式下投影函数的计算图

4 测试与分析

本节中对实现的并行化自动微分进行测试。将其应用到 BAL (Bundle Adjustment in the Large) 问题[12] 中。所选择的数据集见表表 3。作为对比，同时引入了对 Ceres Solver 在该问题上的测试。Ceres Solver 是 Google 公司推出的用于解大型数值优化问题的开源软件。其中的自动微分基于 OpenMP 多线程框架，在 CPU 上实现了线程级别的并行计算。

表 2　各数据集的主要数据

数据集名称	相机数量	3D 点数量	2D 点数量
Ladybug	1723	156502	14992
Trafalgar	257	65132	225911
Dubrovnik	356	226730	1255268
Venice	1778	993923	5001946
Rome	4585	1324582	9125125

测试的平台为一台兼容 PC 机。采用的处理器为 Intel XEON E5 2643 v2，其共有 12 个超线程核心，运行频率为 3.2 GHz，配备的内存为 1866 MHz 的 8GB DDR3 RAM。用于测试 OpenCL 并行计算的 GPU 采用 AMD 的 R9 290，其共有 40 个计算单元，4GB 显存，核心和显存的工作频率分别为 945 MHz 和 1 240 MHz。R9 290 能够提供很强大的并行计算能力，其单精度浮点计算性能可达 4848 GFLOPS，而双精度浮点计算能力也可达到 606 GFLOPS。对于大多数科学计算来说，例如使用光束平差法的优化问题，通常都使用双精度浮点数据。

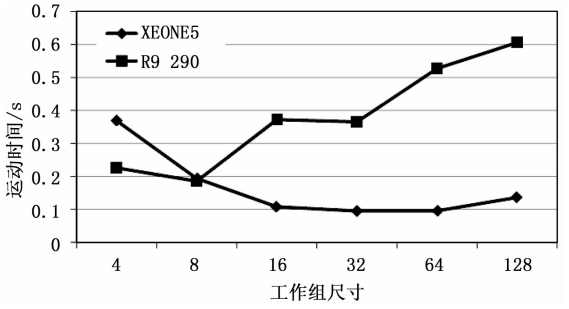

图 4　工作组尺寸对性能的影响

首先测试工作组的大小对计算的性能的影响，采用的数据集为 Dubrovnik。工作组的大小从 4 变化到 128。在 CPU 和 GPU 上各运行 100 次取均值，得到的结果见 0。从结果可以看出，CPU 运行所需时间普遍小于 GPU 所需时间。随着工作组尺寸的增长，GPU 计算的性能会逐渐的下降。其主要原因在于 GPU 上工作组内存储器的访问冲突。GPU 上的存储器按较大地址块（如 1K 字节）的形式访问，同一个块上的地址必须访问。而在 CPU 上该问题得到很大的缓解，因为 CPU 上的地址块大小要小很多。

在对所有的数据进行测试时，结合对工作组尺寸的分析，针对 CPU 采用的工作组尺寸为 32，而针对 GPU 采用的工作组尺寸为 8。所获得的测试结果见 0。使用同样的 CPU，本文基于 OpenCL 的方法比 Ceres Solver 基于 OpenMP 的方法在速度上快了大约 3.6 倍。而基于 GPU 的实现比 Ceres Solver 快了 1.6 倍。由此可见，CPU 实现比 GPU 实现的性能要好。尽管 R9 290 的 GPU 有 40 个计算单元，但其只运行在 1 GHz。而 XEON E5 的 CPU 运行在 3.2 GHz。同时，自动微分中存在着许多的转移分支，这并不利于 GPU 发挥其流水线的特性。相反，CPU 是被设计为执行复杂任务的，有着强大的分支预测能力。因此，大规模的自动微分更适合在多核的 CPU 中执行。

表 3　自动微分在各数据集上的测试结果

数据集名称	CPU/ s	GPU/ s	Ceres Solver/s(CPU)
Ladybug	0.052	0.114	0.190
Trafalgar	0.018	0.039	0.064
Dubrovnik	0.094	0.210	0.341
Venice	0.383	0.864	1.400
Rome	0.687	1.626	2.623

5　总结

本文首先展示了自动微分系统的工作原理，包括正向模式和反向模式。对于多参数的函数，揭示了反正模式比正向模式具有更高的效率。以 OpenCL 为并行计算框架，实现了针对大型优化问题的并行化自动微分。该实现采用反向计算模式，以 C/C++的风格构建函数，并生成计算图和计算序列。以光束平差为应用背景，通过测试，该实现比 Ceres Soler 快约 3.6 倍。同时可以发现，自动微分在 CPU 上的实现要优于在 GPU 上的实现。

参考文献：

[1] Nocedal J, Wright S J. Numerical optimization [M]. Springer New York, 2006.

[2] 张　巍，陈清军. 土桩结构非线性相互作用体系行波效应的并行计算分析 [J]. 湖南大学学报（自科版），2012，39 (6)：19－25.

[3] Lourakis M I A, Argyros A A. SBA: a software package for generic sparse bundle adjustment [J]. ACM Transactions on Mathematical Software, 2009, 36 (1): 2.

[4] Lourakis M I A, Argyros A A. Is Levenberg－Marquardt the Most Efficient Optimization Algorithm for Implementing Bundle Adjustment [A]. Tenth IEEE International Conference on Computer Vision [C]. IEEE, 2005 (2): 1526－1531.

[5] Griewank A. Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation [M]. Society for Industrial and Applied Mathematics, 2015.

[6] 张春晖，程　强，曹建文. 针对 C 语言的自动微分系统及其应用 [J]. 计算机应用研究，2009，26 (1)：158.

[7] Griewank A, Juedes D, Utke J. Algorithm 755: ADOL－C: a package for the automatic differentiation of algorithms written in C/C++ [M]. ACM, 1996.

[8] Gaster B, Howes L, Kaeli D R, et al. Heterogeneous Computing with OpenCL: Revised OpenCL 1.2 Edition [M]. Morgan Kaufmann Publishers Inc. 2012.

[9] Munshi A, Gaster B, Mattson T G, et al. OpenCL Programming Guide [C]. Addison－Wesley Professional, 2012.

[10] Phipps E, Pawlowski R. Efficient Expression Templates for Operator Overloading－Based Automatic Differentiation. Recent Advances in Algorithmic Differentiation [M]. Springer Berlin Heidelberg, 2012: 309－319.

[11] Corke P. Robotics, Vision and Control [M]. Springer Berlin Heidelberg, 2011.

[12] Agarwal S, Snavely N, Seitz S M, et al. Bundle Adjustment in the Large [A]. European Conference on Computer Vision [C]. Springer－Verlag, 2010: 29－42.