Massively Parallel Jacobian Computation

by

Wanqi Li

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Thomas F. Coleman

Waterloo, Ontario, Canada, 2013

I hereby declare that I am the sole author of this report. This is a true copy of the report, including any required final revisions, as accepted by my examiners.

I understand that my report may be made electronically available to the public.

Abstract

The Jacobian evaluation problem is ubiquitous throughout scientific computing. In this article, the possibility of massively parallel computing of Jacobian matrix is discussed. It is shown that the computation of the Jacobian matrix shares the same parallelism with the computation being differentiated, which suggests that once we know how to parallelize a computation, its Jacobian computation can be parallelized in a similar manner. A simple problem is used to demonstrate the potential of massively parallel Jacobian computation with GPU. Significant speedup is observed.

Acknowledgements

I would like to thank all the people who made this possible.

Contents

Chapter 1

Introduction

The Jacobian evaluation problem is ubiquitous throughout scientific computing. This article discusses the topic of accelerating the evaluation of Jacobian matrix by exploiting parallelism. Specifically, we are interested in a special type of parallelism called data parallelism.

Consider a computation with n input variables, p intermediate variables and m output variables:

$$\left.\begin{aligned} v_1 &= F_1\left(v_i\right)_{v_i \prec v_1} \\ v_2 &= F_2\left(v_i\right)_{v_i \prec v_2} \\ &\vdots \\ v_{p+m} &= F_{p+m}\left(v_i\right)_{v_i \prec v_{p+m}} \end{aligned}\right\}. \tag{1.1}$$

Each line in (1.1) will be referred to as an **operation**, which consists of the evaluation of function $F_j\ \left(v_i\right)_{v_i \prec v_j}$ and the assignment of the result to variable v_j. The tuple of variables required for evaluating F_j is written as $\left(v_i\right)_{v_i \prec v_j}$, where the symbol $\prec$ denotes a binary relation $R \subset V \times V$ on the set of variables $V = \{v_i : i = 1-n, \ldots, p+m\}$. In this article, it is always assumed that

$$v_i \prec v_j \Rightarrow i < j, \tag{1.2}$$

i.e., the value of variable v_j only depends on values obtained prior to it. The binary relation R is called **data dependency** [1].

Most of today's computer programs can be viewed as a sequence of operations like (1.1).

Denote the input variables as $x_i = v_{i-n}$, $i = 1, \ldots n$, the intermediate variables as $y_i = v_i$, $i = 1, \ldots, p$ and the output variables as $z_i = v_{p+i}$, $i = 1, \ldots, m$.

Computation (1.1) can be equivalently written as an extended system:

$$\left.\begin{array}{ll} \text{Solve for } y_1 & F_1(x) - y_1 = 0 \\ \text{Solve for } y_2 & F_2(x, y_1) - y_2 = 0 \\ & \vdots \\ \text{Solve for } y_p & F_p(x, y_1, \ldots, y_{p-1}) - y_p = 0 \\ \text{Solve for } z & \bar{F}(x, y_1, \ldots, y_p) - z = 0 \end{array}\right\}. \tag{1.3}$$

In this article, it is always assumed that each function in (1.3) is continuously differentiable. According to the Implicit Function Theorem, in a neighborhood of any point (x, y, z) yielded by computation (1.1), the extended system (1.3) uniquely determines a continuously differentiable function $z = F(x)$.

The Jacobian of the extended system (1.3) w.r.t. (x, y) is as follows:

$$J^E = \left[\begin{array}{c|cccc} J_x^1 & -I & & & \\ J_x^2 & J_{y_1}^2 & -I & & \\ \vdots & \vdots & \vdots & \ddots & \\ J_x^p & J_{y_1}^p & \vdots & J_{y_{p-1}}^p & -I \\ \hline \bar{J}_x & \bar{J}_{y_1} & \cdots & \cdots & \bar{J}_{y_p} \end{array}\right] = \left[\begin{array}{cc} A & L \\ B & M \end{array}\right], \tag{1.4}$$

where $A \in \mathbb{R}^{p \times n}$, $B \in \mathbb{R}^{m \times n}$, $L \in \mathbb{R}^{p \times p}$ and $M \in \mathbb{R}^{m \times p}$. The matrices in J^E are usually very sparse. The Jacobian of function F satisfies the Schur complement form:

$$J = B - ML^{-1}A. \tag{1.5}$$

Forward mode AD (Automatic Differentiation) evaluates Jacobian by computing (1.5) as $J = B - M\left(L^{-1}A\right)$ while reverse mode AD computes $J = B - \left(ML^{-1}\right)A$. Which one is more efficient depends on the size of input n and the size of output m.

For readers unfamiliar with the Schur complement form of AD, a simple derivation of (1.5) is given in Appendix A.

In any computation with significant complexity, the number of intermediate variables p is many orders of magnitude larger than m and n. For example, in a computation with complexity $\Theta(n^3)$ where n is the input size, the number of intermediate variables will be $p = \Theta(n^3)$. Computation of (1.5) is almost always dominated by solving the sparse triangular system L^{-1} which is the focus of this article.

Different strategies have been proposed to evaluate (1.1). In [2][3][4], the structure of the underlying computation is exploited to enable the application of sparse matrix techniques even when the Jacobian itself is dense. If the Jacobian is sparse, then it is possible to compute by AD the product of J with a relatively slim and

dense rectangular matrix and then recover the full sparse Jacobian without too much effort [5][6].

The major topic of this article is to locate structure conductive to parallelism in AD. More specifically, a special type of parallelism called Data Parallelism or SIMD (Single Instruction - Multiple Data) type parallelism [7] is our top interest. An example problem is used to demonstrate the potential of massively parallel Jacobian computation.

The speed of serial computing is dictated by the performance of the single processor in use. Constrained by the laws of physics, improving the speed of single processor is increasingly more expensive and challenging [8]. Meanwhile, our desire for solving ever increasingly large problems continues. Instead of asking for faster single processor, multiple processors are combined together to solve problems in parallel. Today, parallel computing seems to be our only practical way to overcome the performance limit of single processor. Quantum computer and quantum algorithms [9] might affect the way we compute things in the future, but it will not be the concern of this article.

Extra effort is required to enable parallel computing. The first challenge is to locate the parallelizable parts of the computation, either manually or automatically. Once that is achieved, appropriate parallel processor and parallel programming model can be used to exploit the located parallelism accordingly. Specifically, in this article, we will locate data parallelism in AD and apply the massively parallel processor GPU and NVIDIA's CUDA programming model [10] to demonstrate its potential.

Chapter 2 gives the definitions of the basic concepts required to discuss parallel computing. Chapter 3 is a brief review of AD, where parallelism in Jacobian computation is also discussed. The observations made in the first two chapters are applied to a simple example problem in chapter 4 to demonstrate the potential of massively parallel Jacobian computation. Finally, a summary is given in chapter 5.

Chapter 2

Commutativity and Parallelism

2.1 Data Dependency Graph

First we introduce a very helpful and widely used concept for discussing the structure of computation [1].

Definition 1 *Define the* ***data dependency graph*** *of computation* (1.1) *as the ordered pair* $G = (V, R)$*, where* V *is the set of nodes and every ordered pair* $(u, v) \in R$ *corresponds to an arc from* u *to* v*.*

The data dependency graph is a DAG (Directed Acyclic Graph), because by assumption (1.2) the transitive closure of R forms a strict partial order. Each operation in (1.1) corresponds to a node in G. Operations in (1.1) can be sequentially executed following any topological ordering on G, because it guarantees that operation only refers to values that are obtained prior to it.

It is well known that topological ordering always exists for any DAG. Furthermore, the topological ordering is unique if and only if the DAG is Hamiltonian[1].

Fig. 2.1(a) illustrates the data dependency graph of a 2D rotation with angle x_1:

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} \cos x_1 & \sin x_1 \\ -\sin x_1 & \cos x_1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \end{bmatrix}. \tag{2.1}$$

Its operations are listed in Fig. 2.1(b). The number assigned to each operation follows a topological ordering on the DAG.

[1] A DAG is Hamiltonian if it possesses a Hamiltonian path. A Hamiltonian path is a path that visits each node exactly once.

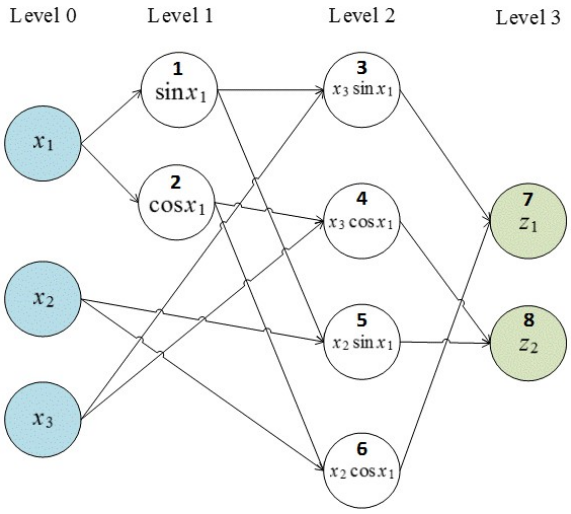

Fig. 2.1(a) DAG of 2D rotation.

1: $y_1 = \sin x_1$
2: $y_2 = \cos x_1$
3: $y_3 = x_3 * y_1$
4: $y_4 = x_3 * y_2$
5: $y_5 = x_2 * y_1$
6: $y_6 = x_2 * y_2$
7: $z_1 = y_3 - y_6$
8: $z_2 = y_4 + y_5$

(b) Operations.

2.2 Commutativity and Parallelism

If operations in (1.1) can be carried out following different execution orders, then at least some operations are commutative in the sense it does not matter which one gets performed first. In that case, it is possible to perform these commutative operations in parallel. On the other hand, if there is only one unique execution order in which these operations can be performed, then there is no commutativity among these operations, hence no parallelism. We can locate parallelism via commutativity.

In [11], a partitioning method is proposed to find parallelizable parts of a computation that involves only linear operators by locating commutativity among these linear operators. In the following, the exactly same method is used to find parallelizable part but for general operations.

Let $G = (V, R)$ be the data dependency graph of computation (1.1). Define $Level\,(v_i)$ as follows:

1. If indegree of v_i is 0, then $Level\,(v_i) = 0$.

2. Otherwise, $Level\,(v_i) = 1 + Level\,(v_i')$, where v_i' is the node corresponding to v_i in the subgraph of G obtained by deleting all nodes at level 0 and their outgoing arcs.

Fig. 2.1(a) illustrates the concept of level. Note the one-on-one correspondence

between non-input nodes and operations. Operations at the same level can be executed in parallel.

This method computes the levels of each variable/operation in linear time, given the condition that G is sparse. The resulting level sets $P_i = \{v \in V : Level\,(v) = i\}$ yield a partitioning on V.

Different topological orderings can be generated by switching the indexes of nodes within the same partition P_i. Hence operations at the same level are commutative. Also since $v_i \prec v_j \Rightarrow Level\,(v_i) < Level\,(v_j)$, operations at the same level are independent with each other. Therefore, operations in the same partition can be executed in parallel. Partition P_i, $i > 0$ will be called a **parallelizable part** of the computation.

Note that the above argument is only based on the data dependency graph. If two computations share the same data dependency graph, then they also share the same partitioning, same parallelism.

$Level\,(v_i)$ is the length of the longest path that ends at v_i. G is Hamiltonian if and only if $\max\limits_{v \in V} Level\,(v) = |V| - 1$, where $|V|$ is the number of variables. If G is non-Hamiltonian, then some part of the computation might be parallelizable. Next we try to quantify the portion of parallelism.

Let T be the execution time of the most time consuming operation in (1.1). Assume that there are enough processors to allow the simultaneous execution of operations in every parallelizable part P_i. Then the parallel execution time for P_i is upper bounded by T. The total parallel execution time is upper bounded by $\max\limits_{v \in V} Level\,(v) * T$. In the spirit of Amdahl's law [12], define the **ratio of parallelism** as

$$r = 1 - \frac{\max\limits_{v \in V} Level\,(v) * T}{|p + m| * T} = \frac{|p + m| - \max\limits_{v \in V} Level\,(v)}{|p + m|}. \tag{2.2}$$

The ratio of parallelism is a rough estimation of the potential benefit of parallel computing. If F is a pure iteration upon a scalar value, which trivially indicates a Hamiltonian data dependency graph, then $r = 0$. If F is an iteration on a 2D vector, and each component is a variable, then $r = 0.5$. The operations for the two components can be performed in parallel and parallel computing has the potential to half the execution time.

2.3 Data Parallelism

Now we review a special type of parallelism called data parallelism. Computing that exploits data parallelism is called **massively parallel** computing. Let P_i, $i > 0$ be a parallelizable part of computation (1.1). A subset $P_i^f \subset P_i$ is called a **massively parallelizable part** if all its corresponding operations are defined

with the same function f, i.e. $P_i^f = \{v_j \in V : F_j = f,\ Level\,(v_j) = i\}$. Operations in a massively parallelizable part apply the same function to different data.

Again, consider the data dependency graph in Fig. 2.1(a). Operations at Level 1 does not form a parallelizable part because they use different functions (one is sine and the other is cosine). In contrast, operations at level 2 are all the product of two variables. So P_2 is a massively parallelizable part.

Using Flynn's classification [7] of computer architectures, there are three types of parallelism:

1. **SIMD (Single Instruction - Multiple Data Streams or Data Parallelism)** Computation applying the same function to different inputs.

2. **MISD (Multiple Instructions - Single Data Stream)** Computation applying a set of different functions to the same input.

3. **MIMD (Multiple Instructions - Multiple Data Streams)** Computation conducting different operations that are independent with each other. MIMD is the most general type of parallelism. Both MISD and SIMD are special cases of MIMD.

Processors of different architectures are designed to exploit different types of parallelism. Most notably, multi-core CPU is designed to exploit MIMD type parallelism (parallelizable parts), while GPU is designed to exploit data parallelism (massively parallelizable parts). Since data parallelism is a subset of MIMD type parallelism, parallelism exploitable to GPU is also exploitable to CPU. Though in practice CPU can hardly beat GPU in exploiting data parallelism because of GPU's more specialized design [13]. In real world problems, computations are usually a mixture of serial part, data parallelism and MIMD type parallelism. CPU and GPU can be simultaneously applied to different parts of a computation. This strategy is being studied under the field called heterogeneous computing [14]. But it is not going to be the concern of this article.

The good thing about data parallelism is that we can create a large number of threads sharing the same program code (source code and machine code) for a massively parallelizable part, so that many light-weight cores, like GPU cores [13], can run these threads almost synchronously with minimum amount of inter-thread communication. While for MIMD type parallelism, threads run different program code asynchronously. Much less cores can fit into one processor and each core has much more sophisticated control unit.

Chapter 3

Jacobian Computation with Automatic Differentiation

AD (Automatic Differentiation or Algorithmic Differentiation) is the study of computing derivatives using only the source code of the mathematical function, which is essentially a list of operations in the form of (1.1). AD is built upon the assumption that all the operations are defined with a finite set of differentiable "elementary functions", whose Jacobian can be easily obtained. Computation that meets this assumption can be viewed as a composite function of these elementary functions. The Jacobian matrix of this composite function can be obtained by applying the chain rule. The choice of the set of elementary functions is subjective. It could be as low level as the set of mathematical instruction set of CPU/GPU, or it could be as complicated as functions from mathematical libraries.

Since AD computes derivatives using the chain rule, there is no truncation error for the result. Also, because AD can flexibly exploit the structure and sparsity in a computation [15][2][3][5][6][4], it tends to be the most accurate and efficient scheme of computing derivatives [16][17][18]. AD has been widely deployed in different applications, especially those requiring nonlinear optimization [19][20].

3.1 Jacobian Computation

The Jacobian matrix $J \in R^{m \times n}$ as in (1.5) is defined by the chain rule:

$$J_{i,j} = \frac{dz_i}{dx_j} = \frac{\partial z_i}{\partial x_j} + \sum_{y_k \prec z_i} \frac{\partial z_i}{\partial y_k} \frac{dy_k}{dx_j}, \quad i = 1, \ldots, m \tag{3.1}$$

$$\frac{dy_i}{dx_j} = \frac{\partial y_i}{\partial x_j} + \sum_{y_k \prec y_i} \frac{\partial y_i}{\partial y_k} \frac{dy_k}{dx_j}, \quad i = 1, \ldots, p \tag{3.2}$$

for $j = 1, \ldots, n$. The Schur complement form (1.5) is simply the linear algebra description of the chain rule (3.1) (3.2). For readers unfamiliar with the Schur complement form of AD, a simple derivation of (1.5) from the chain rule is given in Appendix A.

The assumption of AD made at the beginning of this chapter basically says that all the partial derivatives, $\frac{\partial y_i}{\partial x_j}$, $\frac{\partial y_i}{\partial y_k}$, $\frac{\partial z_i}{\partial x_j}$ and $\frac{\partial z_i}{\partial y_k}$ on the right hand side of the chain rule (3.1) (3.2) can be easily obtained. From now on, we will assume these partial derivatives as known constants, since the cost of evaluating these partial derivatives is usually negligible. Detailed discussion about this assumption can be found in standard textbooks of AD [16][17]. In this article, we focus on obtaining the left hand side of (3.1) (3.2).

Formulae (3.1) (3.2) can be equivalently written in total differential form:

$$\frac{dz_i}{dx_j} = \sum_{x_k \prec z_i} \frac{\partial z_i}{\partial x_k}\frac{dx_k}{dx_j} + \sum_{y_k \prec z_i} \frac{\partial z_i}{\partial y_k}\frac{dy_k}{dx_j} \tag{3.3}$$

$$\frac{dy_i}{dx_j} = \sum_{x_k \prec y_i} \frac{\partial y_i}{\partial x_k}\frac{dx_k}{dx_j} + \sum_{y_k \prec y_i} \frac{\partial y_i}{\partial y_k}\frac{dy_k}{dx_j}, \tag{3.4}$$

where $\frac{dx_k}{dx_j} = 1$ if $k = j$ and $\frac{dx_k}{dx_j} = 0$ if $k \neq j$. The computation defined by (3.3) (3.4) will hereafter be referred to as **Jacobian computation**.

Lemma 2 *Let V be the set of variables of computation* (1.1). *Let* $W = \left\{\frac{dx_i}{dx_j}\right\} \cup \left\{\frac{dy_i}{dx_j}\right\} \cup \left\{\frac{dz_i}{dx_j}\right\}$ *be the set of variables of the corresponding Jacobian computation. W is isomorphic to V under their data dependencies.*

Proof. The isomorphism is trivially implied by the indices in (3.3) (3.4):

$$\frac{dx_k}{dx_j} \prec \frac{dz_i}{dx_j} \Leftrightarrow x_k \prec z_i, \quad \frac{dy_k}{dx_j} \prec \frac{dz_i}{dx_j} \Leftrightarrow y_k \prec z_i$$
$$\frac{dx_k}{dx_j} \prec \frac{dy_i}{dx_j} \Leftrightarrow x_k \prec y_i, \quad \frac{dy_k}{dx_j} \prec \frac{dy_i}{dx_j} \Leftrightarrow y_k \prec y_i.$$

■

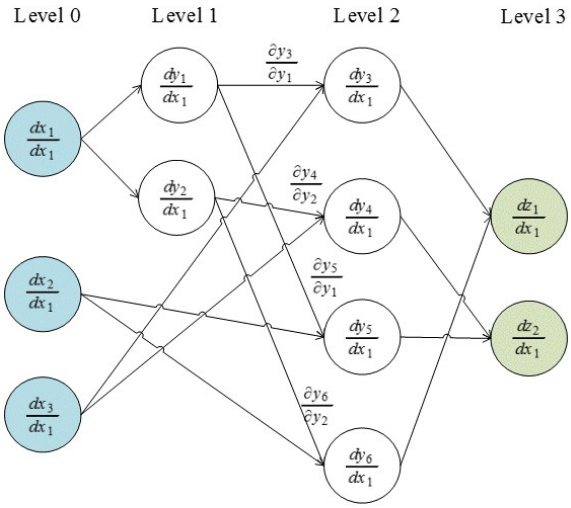

Fig. 3.1 DAG of the Jacobian computation.

Fig. 3.1 shows the data dependency graph of the Jacobian computation for the 2D rotation (2.1). It is isomorphic to the DAG in Fig. 2.1. These two figures share exactly the same parallelizable parts, because parallelism is nothing but a property of their shared topological structure.

The fact that V and W are isomorphic to each other indicates that computation (1.1) and its Jacobian computation can be executed following the same set of topological orderings. Therefore they share the same parallelizable parts. The major implication of this is that Jacobian computation of (1.1) is equally parallelizable with computation (1.1). Formulating AD in matrix form as in (1.5) does not provide us any extra parallelism. If computation (1.1) is not parallelizable, then the linear algebra computation in (1.5) will not be parallelizable either. Specifically, solving the sparse triangular system L^{-1} will contain no parallelizable steps. As we will explain further in the following section.

3.2 Parallelism in AD

In any computation with significant complexity, the number of intermediate variables p is many orders of magnitude larger than m and n. For example, in a computation with complexity $\Theta(n^3)$ where n is the input size, the number of intermediate variables will be $p = \Theta(n^3)$. Thus computation of (1.5) is usually dominated by solving the sparse lower triangular system $L^{-1}A$ for forward mode

AD or the sparse upper triangular system ML^{-1} for reverse mode AD. The other part of Jacobian computation is almost negligible in terms of time elapse. Since upper triangular system can be treated as lower triangular system after being transposed, in this article we will only consider solving $L^{-1}A$.

L_j denotes the matrix obtained by setting all the entries of L to 0 except the off-diagonal entries in the j-th column. Since L is a lower-triangular matrix with negative unit diagonal, we have

$$\begin{aligned}(-L)^{-1} &= [(I-L_1)(I-L_2)\cdots(I-L_{p-1})]^{-1} \\ &= (I-L_{p-1})^{-1}\cdots(I-L_2)^{-1}(I-L_1)^{-1} \\ &= (I+L_{p-1})\cdots(I+L_2)(I+L_1). \end{aligned} \tag{3.5}$$

Factor $(I+L_j)$ in (3.5) is the linear operator performing the j-th step of column-oriented forward substitution for solving $L^{-1}A$. Some of these factors (forward substitution steps) are commutative. The partitioning method described in section 2.2 also applies here. The following result is from [11].

Let $\tau = \max_{i\in\{1,\ldots,p\}} Level(y_i) - 1$. If variables in V are indexed by a topological ordering satisfying

$$Level(y_j) < Level(y_i) \Rightarrow j < i,$$

then the partitioning method in section 2.2 gives a partitioning on the factors of (3.5)

$$p-1 = q_\tau < q_{\tau-1} < \cdots < q_0 = 0 \tag{3.6}$$

such that

$$(-L)^{-1} = P_\tau P_{\tau-1}\cdots P_1 \tag{3.7}$$

where each factor $P_k = \left(I + L_{q_k} + L_{q_k-1} + \cdots + L_{q_{k-1}}\right)$ is a summation of the linear operators in the k-th parallelizable part and the identity matrix. Each linear operator defines one forward substitution step in solving $L^{-1}A$. The linear operators (forward substitution steps) in the same parallelizable part are commutative and thus can be combined together as P_k and performed in parallel. Note:

1. The partitioning (3.6) is from the analysis of the data dependency graph of (1.1) which is already available before the Jacobian computation if computation (1.1) is carried out in parallel. So the partitioning is there for free.

2. Nonzero entries in the partitioned factor P_k are the same with their corresponding entries in L. So factor P_k is available for free.

Again using 2D rotation as an example. Consider the L matrix for the Jacobian computation shown in Fig. 3.1:

$$(-L) = \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ -\frac{\partial y_3}{\partial y_1} & & 1 & & & \\ & -\frac{\partial y_4}{\partial y_2} & & 1 & & \\ -\frac{\partial y_5}{\partial y_1} & & & & 1 & \\ & -\frac{\partial y_6}{\partial y_2} & & & & 1 \end{bmatrix}.$$

For this very special case $\tau = \max_{i \in \{1,\ldots,p\}} Level\,(y_i) - 1 = 1$. There is only one partition

$$\begin{aligned} (-L)^{-1} &= P_1 = (I + L_1 + \cdots + L_6) \\ &= \begin{bmatrix} 1 & & & & & \\ & 1 & & & & \\ \frac{\partial y_3}{\partial y_1} & & 1 & & & \\ & \frac{\partial y_4}{\partial y_2} & & 1 & & \\ \frac{\partial y_5}{\partial y_1} & & & & 1 & \\ & \frac{\partial y_6}{\partial y_2} & & & & 1 \end{bmatrix}. \end{aligned} \tag{3.8}$$

Standard forward substitution requires 4 sequential steps to solve $L^{-1}A$. With the partition P_1, only one parallel step is required to solve $L^{-1}A$, because P_1 combines commutative forward substitution steps together so that they can be performed in parallel.

In general, solving system $L^{-1}A$ using (3.5) requires $\Theta\,(p-1)$ sequential steps, while using (3.7) only requires $\tau = \max_{i \in \{1,\ldots,p\}} Level\,(y_i) - 1$ parallel steps. The potential benefit of parallel computing for Jacobian computation is quantified by the ratio of parallelism (2.2).

The partition (3.7) suggests that matrix L is accessed block-wise column by column from left to right. This pattern can be taken advantage of to reduce memory usage without affecting any parallelism, as proposed by Coleman et al. in [21][22]:

1. Do not store all the non-zero elements in L. Instead, evaluate P_k with AD only when it is needed.

2. Do not form P_k explicitly. Instead, compute the matrix-vector product $P_k v$ directly with AD.

In practice, the storage of L is often the bottleneck of performance. As p increases, matrix L can easily exhaust the main memory and force the system to use paging files, which is tremendously slower. Using the strategy proposed in [21][22], the memory usage can be significantly reduced to avoid paging files, or even further, to keep all the variables in high-speed cache and avoid the slower main memory access.

3.3 Data Parallelism in AD

Consider the general matrix multiplication $C = AB$, where each entry of C is given by $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$. All c_{ij}'s are independent from each other, and thus are parallelizable. But for them to be massively parallelizable, as defined in section 2.3, it is required that all c_{ij}'s are defined with the same function.

If A is dense, then all c_{ij}'s are the inner product of one row of A and one column of B. In that case, c_{ij}'s are massively parallelizable. However, if A is sparse, the different sparsity patterns of rows of A will make c_{ij}'s computationally different from each, in which case, only c_{ij}'s correspond to rows sharing the same sparsity pattern are massively parallelizable.

After obtaining factors $\{P_k\}$ as in the last section, computation (1.5) is mainly a sequence of sparse matrix-matrix multiplications, each of which consists of many parallelizable sparse row-column products. Two sparse row-column products are massively parallelizable only when their rows contain the same number of nonzeros. This requires that their corresponding operations in (1.1) depend on the same number of variables.

Again, consider the 2D rotation example. In Fig. 2.1, $P_1 = \{y_1, y_2\}$ is not a massively parallelizable part because the operations for y_1 and y_2 are defined with sine and cosine functions respectively. Meanwhile, in Fig. 3.1, $P_1 = \left\{ \frac{dy_1}{dx_1}, \frac{dy_2}{dx_1} \right\}$ is a massively parallelizable part, because $\frac{dy_1}{dx_1}$ and $\frac{dy_2}{dx_1}$ both correspond to row-column inner product with two nonzero entries in the row.

Jacobian computation is equally parallelizable with its underlying computation as a consequence of Lemma 2. Jacobian computation is at least as massively parallelizable as its underlying computation. This is because Jacobian computation only involves sparse linear operators. Operations at the same level are considered the same if their corresponding operations in (1.1) depend on the same number of variables.

Chapter 4

Massively Parallel Automatic Differentiation with GPU

GPUs are one of the most widely installed devices designed to exploit data parallelism. Acceleration has been observed from many computations using GPU [23][13][24]. In this chapter, GPUs are applied to exploit the data parallelism in the Jacobian computation of an example problem in order to demonstrate the potential of massively parallel Jacobian computation. Significant speedup is observed.

4.1 Exploit Data Parallelism with GPU

Detailed introduction to general-purpose computing with a GPU can be found in textbook [13] and NVIDIA manual [10]. In this section, only the CUDA [10] programming model will be briefly reviewed.

Operations in a massively parallelizable part share the same function. In CUDA C, this homogeneity is abstracted by a CUDA **kernel** function, which is basically a function in the C language. The kernel function defines the common behavior of the massively parallelizable part.

After launching the kernel, a set of **threads** will be created and assigned with unique thread indices. These threads will apply the same kernel function to different data accessed through their unique thread indices. Depending on the data accessing pattern of the computation, the thread index could be up to 3-dimensional.

Threads are organized into independent **blocks** with unique block indices. Each block is assigned to a SM (Streaming Multiprocessor), from which each thread gets computing resources. All the blocks have the same number of thread and are assigned with the same proportion of resources. Depending on the data access-

ing pattern of the computation, the block index could be up to 3-dimensional. 32 threads[1] of the same block form a **warp**. Warp is the basic scheduling unit of GPU. An active warp executes one instruction at a time. Parallel execution is achieved by letting the 32 threads in a warp execute the same instruction synchronously.

Threads of the same block run on the same SM. They share the high speed local cache allocated to the block and synchronize with other threads in the same block. Meanwhile, threads of different blocks might be running on different SMs. Local cache sharing among blocks are not allowed. Threads from different blocks can not synchronize with each other. A kernel is finished once all its blocks are done.

Streaming Multiprocessor is the basic processing unit on a GPU. Every SM has its own registers, cache, controller and an array of ALUs (Arithmetic Logic Unit). Multiple ALUs allow SM to apply the same instruction to different data streams in parallel, which is the basic idea behind massively parallel computing. In [10], three principles are suggested to optimize GPU-accelerated computation:

1. Make sure there are enough number of blocks and threads to saturate all the SMs and all their ALUs.

2. Maximize memory throughput by using the appropriate type of memory.

3. Avoid branching of the control flow. Minimizing instructions with low throughput. Minimize the number of instructions.

4.2 Heat Conductivity Inverse Problem

To demonstrate the potential of massively parallel Jacobian computation, we consider a very simple inverse problem which is also used in [15].

1-D heat conduction is mathematically modeled by the following partial differential equation:

$$\begin{cases} \dfrac{\partial u(x,t)}{\partial t} = \dfrac{\partial}{\partial x}\left(s(x)\dfrac{\partial u(x,t)}{\partial x}\right), \quad x \in (0,1),\, t \in (0,T] \\ u(x,0) = u^0(x), \quad x \in (0,1) \\ u(0,t) = f(t), \quad u(1,t) = g(t), \quad t \in (0,T] \end{cases}. \tag{4.1}$$

Function $u(x,t)$ represents the temperature of a rod at position x and time t. Function $s(x)$ is the unknown conductivity of the rod. Our goal is to determine $s(x)$ given the rod's initial temperature $u^0(x)$ at time $t = 0$, boundary temperature $f(t)$ and $g(t)$, and temperature $u(x,T)$ measured at time $t = T$.

[1] Warp size is implementation specific. 32 is the warp size of current CUDA implementation [10].

To obtain the numerical result, discretization of spatial and time domains is required to employ a finite difference method:

$$\begin{aligned} x_j &= j\Delta x, \quad \Delta x = \frac{1}{M+1} \\ t_n &= n\Delta t, \quad \Delta t = \frac{T}{N} \\ u_j^n &= u\left(x_j, t_n\right), \ \ s_j = s\left(x_j\right), \end{aligned}$$

where M is the number of interior nodes in the discretized grid and N is the number of time steps. Denote $\lambda = \Delta t/\Delta x^2$. The finite difference operators that will be used to approximate equation (4.1) are

$$\begin{aligned} &\text{Forward Time} && D_t^+ u_j^n = \frac{u_j^{n+1} - u_j^n}{\Delta t} \\ &\text{Forward Space} && D_x^+ u_j^n = \frac{u_{j+1}^n - u_j^n}{\Delta x} \\ &\text{Backward Space} && D_x^- u_j^n = \frac{u_j^n - u_{j-1}^n}{\Delta x} \\ &\text{Centered Space} && D_x^+ D_x^- u_j^n = D_x^- D_x^+ u_j^n = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}. \end{aligned}$$

The heat equation (4.1) is approximated as

$$D_t^+ u_j^n = \frac{1}{2}\left(D_x^+\left(s_j D_x^- u_j^n\right) + D_x^-\left(s_j D_x^+ u_j^n\right)\right). \tag{4.2}$$

Expanding (4.2) to obtain

$$\begin{aligned} u_j^{n+1} &= c_{j-1}u_{j-1}^n + c_j u_j^n + c_{j+1}u_{j+1}^n \\ &= \begin{bmatrix} c_{j-1} & c_j & c_{j+1} \end{bmatrix} \begin{bmatrix} u_{j-1}^n \\ u_j^n \\ u_{j+1}^n \end{bmatrix} \end{aligned} \tag{4.3}$$

where for $j \in \{2, \ldots, M-1\}$

$$c_{j-1} = \frac{\lambda}{2}\left(s_{j-1} + s_j\right),\ c_j = 1 - \frac{\lambda}{2}\left(s_{j-1} + 2s_j + s_{j+1}\right),\ c_{j+1} = \frac{\lambda}{2}\left(s_j + s_{j+1}\right), \tag{4.4}$$

for $j = 1$

$$c_0 = \frac{\lambda}{2}\left(3s_1 - s_2\right),\ c_1 = 1 - 2\lambda s_1,\ c_2 = \frac{\lambda}{2}\left(s_1 + s_2\right), \tag{4.5}$$

and for $j = M$

$$c_{M-1} = \frac{\lambda}{2}\left(s_{M-1} + s_M\right),\ c_M = 1 - 2\lambda s_M,\ c_{M+1} = \frac{\lambda}{2}\left(s_M - s_{M-1}\right). \tag{4.6}$$

Denote the solution of equation (4.2) at time $t = T$ as U^N , and the given temperature measurement at time $t = T$ as u^N. Both U^N and u^N are M-dimensional vectors. U^N can be viewed as a function of the conductivity s. s is also an M-dimensional vector after discretization. The inverse problem is to find conductivity s that matches the measured data u^N. That is, we want to solve the M-dimensional nonlinear system

$$F\left(s\right) = u^N - U^N = 0. \tag{4.7}$$

4.2.1 Numerical Methods

Many methods for solving (4.7) involve computation of the Newton step [2]

$$w_{\text{Newton}} = -J\left(s\right)^{-1} F\left(s\right). \tag{4.8}$$

Equations (4.3)-(4.6) defines all the operations required to compute U^N. It is easy to see that these operations are defined with only 5 elementary functions. The operation defined by (4.3) is an inner product of two 3-dimensional vectors. Operations defined by (4.4)-(4.6) contains 4 different forms of weighted summations. Computation using only these 5 elementary functions evidently satisfies the assumption of AD as stated in the beginning of chapter 3. The Jacobians of these 5 elementary functions can be obtained in a straight forward manner. Therefore AD is applicable for obtaining the Jacobian $J\left(s\right)$ of $F\left(s\right)$ in (4.8).

Though methods have been proposed to obtain the Newton step w_{Newton} without forming the full Jacobian matrix $J\left(s\right)$ [2], they are beyond the scope of this article. In this article, we will use AD to form full Jacobian matrix $J\left(s\right)$, in order to demonstrate the potential of massively parallel Jacobian computation.

4.2.2 Data Parallelism

Apply the partitioning method in section 2.2 to computation F defined by operations (4.3)-(4.6). The levels of the non-input variables are as follows

$$\begin{array}{ll} Level\left(c_j\right) = 1 & j = 1, 2, \ldots M \\ Level\left(U_j^n\right) = n + 1 & j = 1, 2, \ldots M,\ \ n = 1, \ldots, N \end{array}. \tag{4.9}$$

Starting from level 2, every operation is an inner product as defined in (4.3). Hence each level starting from level 2 is massively parallelizable. Another convenient

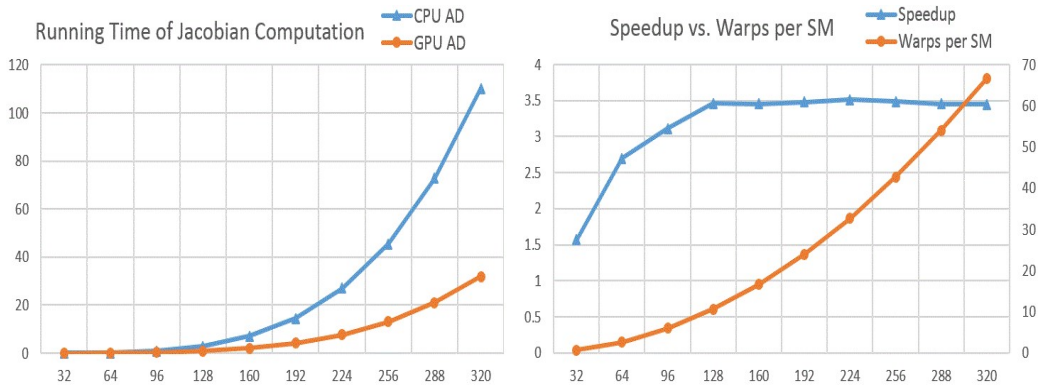

Figure 4.1: Fig. 4.1 Performance comparison between CPU-based AD vs. GPU-based AD.

property of F is that its ratio of parallelism,

$$r = \frac{(N+1) * M^2 - (N+1)}{(N+1) * M^2} = 1 - \frac{1}{M^2}, \tag{4.10}$$

increases as the size of the problem increases. This property will provide us enough data parallelism to observe GPU acceleration.

As a consequence of Lemma 2, (4.9) and (4.10) also apply to the Jacobian computation of F.

4.3 Experimental Results

For our experiments, a simple AD C/C++ library was implemented for illustrative purposes. This library is called SAD (Simple Automatic Differentiation). The source code is openly available at [25]. SAD only supports the 5 elementary functions required for the heat conductivity inverse problem stated earlier. The derivatives of these 5 elementary functions are computed while solving equation (4.2). After that, the Jacobian matrix $J(s)$ is formed via forward mode AD. Two versions of forward mode AD is implemented. One uses CPU to carry out the Jacobian computation serially. The other version uses GPU to exploit the data parallelism in the Jacobian computation. In this experiment, Intel Core i7 is the CPU and NVIDIA GeForce GT 650m is the GPU. Because of the large memory usage of this experiment, the program is compiled for Windows 8 operating system using 64-bit compiler. Detailed information about this experiment can be found at [25].

The running time for forming full Jacobian is shown on the left of Fig. 4.1. The horizontal axis is the size of the problem, i.e. the grid size M. The vertical axis is running time in seconds. Thanks to the high ratio of parallelism of Jacobian computation, as shown in (4.10), GPU-based AD is significantly faster than CPU-based AD, starting from $M = 32$.

In the right plot of Fig. 4.1, the number of warps per SM increases as the size of the problem increases. This is consistent with the ratio of parallelism (4.10) of the heat equation - it increases with M.

However, the speedup stops increasing at 3.5 when M is larger than 128, and warps per SM is larger than 11. This is because every SM only has a finite number of cores, or ALUs (Arithmetic Logic Unit). There is a limit for the extent to which SM can exploit parallelism. After warps per SM exceeding 11, extra even parallelism will not bring further speedup because the computing power of SMs has been saturated. The point of saturation is a determined by the resources available on the GPU.

4.4 Conclusion

If a computation is parallelizable, then its Jacobian computation with forward mode AD is equally parallelizable. GPU can significantly accelerate Jacobian computation if its underlying computation possesses a high ratio of parallelism. The speedup increases as the available parallelism increases, till the computing capability of GPU becomes saturated. After saturation, additional parallelism will not bring further speedup.

Chapter 5

Summary

In chapter 2, the observation is made that as long as the data dependency graph is non-Hamiltonian, some part of the computation is parallelizable. During the procedure of checking the existence of Hamiltonian path, the level of each variable is computed (section 2.2). Operations at the same level are parallelizable (commutative) and are called a parallelizable part of the computation. The ratio of parallelism (2.2) is used to quantify the parallelism in a computation. Parallelizable operations defined with the same function are called a massively parallelizable part. Massively parallelizable part can be easily determined once the parallelizable parts are located. Parallelism is a property of the data dependency graph, while data parallelism is related to the content of the operations.

Jacobian computation is defined as the computation of obtaining a Jacobian matrix by applying the chain rule as in (3.1)(3.2). Jacobian computation can be compactly written in Schur complement form (1.5). The dominant part of Jacobian computation is solving the sparse lower triangular system ($L^{-1}A$) for forward mode AD or the sparse upper triangular system (ML^{-1}) for reverse mode AD. It is shown (Lemma 2) that Jacobian computation defined this way is isomorphic to its underlying computation, i.e. the computation being differentiated. This means, the parallelism observed in computation (1.1) also applies to its Jacobian computation. A partitioning method proposed in [26] is used to combine forward substitution steps in solving $L^{-1}A$ into parallelizable factors (section 3.3). Chapter 3 is ended with the remark: Jacobian computation and its underlying computation are equally parallelizable; Jacobian computation is at least as massively parallelizable as its underlying computation.

Chapter 4 uses GPU to exploit the data parallelism of a simple example problem - the heat conductivity inverse problem. This problem possesses the property that the ratio of parallelism increases as the size of the problem increases (4.10). A Simple AD C/C++ library called SAD is implemented for illustrative purpose [25]. SAD supports both CPU-based and GPU-based AD. Significant speedup is

observed for GPU-based AD. Speedup increases along with the increase of available parallelism at start. The increase stops once the computing capability of the GPU is saturated. After that, even additional parallelism will not promote the speedup.

Automatic Differentiation is a rich field with different types of techniques to obtain derivatives efficiently. Special structures (in addition to parallelism) of the computation can significantly accelerate Jacobian computation with proper exploitation [15][2][3][5][6][4]. In many applications, Jacobian-related results can be obtained even without forming the full Jacobian matrix [2]. Therefore, the example problem in chapter 4 is rather simple and naive. But we do believe in this article we have validated some fundamental observations and provided a general guide line for massively parallel Jacobian computation.

Appendix A

Schur Complement Form of AD

AD (Automatic Differentiation or Algorithmic Differentiation) is the study of computing derivatives using only the source code of the mathematical function, which is essentially a list of operations in the form of (1.1). AD is built upon the assumption that all the operations are defined with a finite set of differentiable "elementary functions", whose Jacobian can be easily obtained. Computation that meets this assumption can be viewed as a composite function of these elementary functions. The Jacobian matrix of this composite function can be obtained by applying the chain rule. The choice of the set of elementary functions is subjective. It could be as low level as the set of mathematical instruction set of CPU/GPU, or it could be as complicated as functions from mathematical libraries. Since the chain rule is an exact formula, AD has no truncation error. AD is reported to be superior in computing Jacobian matrix both in terms of operation count and accuracy [16][17]. It has been widely applied in different areas [19][20] throughout scientific computing.

A.1 AD as Forward Propagation

The Jacobian computation defined by the chain rule in section 3.1 can be carried out by the following forward propagation procedure:

1. Loop through $j = 1, 2, \ldots, n$.

   (a) Set $\frac{dx_j}{dx_j} = 1$ and $\frac{dx_i}{dx_j} = 0$ for $i \neq j$. Set variables $\frac{dy_i}{dx_j}$ $\forall i = 1, \ldots p$ and $\frac{dz_i}{dx_j}$ $\forall i = 1, \ldots, m$ to zero.

   (b) (B) Propagate values from $\left\{\frac{dx_i}{dx_j}\right\}$ to $\left\{\frac{dz_i}{dx_j}\right\}$ via arcs weighted by $\frac{\partial z_i}{\partial x_k}$,

i.e. update $\frac{dz_i}{dx_j}$ by

$$\frac{dz_i}{dx_j} := \sum_{x_k \prec z_i} \frac{\partial z_i}{\partial x_k} \frac{dx_k}{dx_j}, \quad i = 1, \ldots, m. \tag{A.1}$$

(c) (A) Propagate values from $\left\{ \frac{dx_i}{dx_j} \right\}$ to $\left\{ \frac{dy_i}{dx_j} \right\}$ via arcs weighted by $\frac{\partial y_i}{\partial x_k}$, i.e. update $\frac{dy_i}{dx_j}$ by

$$\frac{dy_i}{dx_j} := \sum_{x_k \prec y_i} \frac{\partial y_i}{\partial x_k} \frac{dx_k}{dx_j}, \quad i = 1, \ldots p. \tag{A.2}$$

(d) (L) Propagate values within $\left\{ \frac{dy_i}{dx_j} \right\}$ following some topological ordering, via arcs weighted by $\frac{\partial y_i}{\partial y_k}$, i.e. update $\frac{dy_i}{dx_j}$ by

$$\frac{dy_i}{dx_j} := \frac{dy_i}{dx_j} + \sum_{y_k \prec y_i} \frac{\partial y_i}{\partial y_k} \frac{dy_k}{dx_j}. \tag{A.3}$$

(e) (M) Propagate values from $\left\{ \frac{dy_i}{dx_j} \right\}$ to $\left\{ \frac{dz_i}{dx_j} \right\}$ via arcs weighted by $\frac{\partial z_i}{\partial y_k}$, i.e. update $\frac{dz_i}{dx_j}$ by

$$\frac{dz_i}{dx_j} := \frac{dz_i}{dx_j} + \sum_{y_k \prec z_i} \frac{\partial z_i}{\partial y_k} \frac{dy_k}{dx_j}. \tag{A.4}$$

2. Form Jacobian matrix by assigning $J_{ij} := \frac{dz_i}{dx_j}$.

It is straight forward to check that the above procedure indeed carries out the Jacobian computation (3.3) (3.4). Combining (A.1) and (A.4) reproduces (3.3). Combining (A.2) and (A.3) reproduces (3.4).

At step (L), when propagate values within $\left\{ \frac{dy_i}{dx_j} \right\}$, we must make sure the value of variable $\frac{dy_k}{dx_j}$ is used only after it is ready. Therefore (L) must be performed following some topological ordering of the data dependency graph.

Steps (B), (A) and (M) can be viewed as forward propagation on bipartite subgraphs, because variables in these partitions are independent with each other. Thanks to this independence, propagations on these nodes can be performed in parallel.

The forward propagation described above is called forward mode AD in the literature [16][17].

A.2 AD in the Language of Linear Algebra

Forward propagation on graphs can be easily translated into the language of linear algebra [27]. Forward propagation described in section A.1 is not an exception.

Forward propagation on a bipartite graph can be written as a matrix-vector multiplication. The matrix here is often sparse. Entry in the i-th row j-th column of the matrix is the weight of the arc from the j-th node of the starting side to the i-th node of the terminal side. The j-th component of the vector is the initial value of the j-th node of the starting side. The matrix-vector product yields another vector, whose i-th component is the value of the i-th node of the terminal side. Steps (B), (A) and (M) are all forward propagations on bipartite subgraphs. We will denote their matrices as B, A and M respectively.

Step (L) is a forward propagation on a general DAG, i.e. the subgraph of $\left\{\dfrac{dy_i}{dx_k}\right\}$. Let L_j be the same matrix defined in (3.5). Then the forward propagation step on node j can be written as $v := (I + L_j)\, v$, where each component of v is the value of its corresponding node in $\left\{\dfrac{dy_i}{dx_k}\right\}$. Assume that nodes in $\left\{\dfrac{dy_i}{dx_k}\right\}$ are indexed following some topological ordering, then forward propagation of step (L) can be written as

$$\begin{aligned} v &= (I + L_{p-1}) \cdots (I + L_j) \cdots (I + L_1)\, v \\ &= \left[(I + L_1)^{-1} \cdots (I + L_j)^{-1} \cdots (I + L_{p-1})^{-1}\right]^{-1} v \\ &= \left[(I - L_1) \cdots (I - L_j) \cdots (I - L_{p-1})\right]^{-1} v \\ &= (-L)^{-1}\, v, \end{aligned} \tag{A.5}$$

where L is the same matrix as in (1.5) and (3.5). Solving $L^{-1}v$ with column-oriented forward substitution is computationally equivalent with forward propagation on the subgraph of $\left\{\dfrac{dy_i}{dx_k}\right\}$.

Finally, put all these four matrices together, the forward propagation described in section A.1 can be written as

$$J = B - M\left(L^{-1}A\right). \tag{A.6}$$

By the commutativity of matrix multiplication, (A.6) can also be computed in reverse order:

$$J = B - \left(ML^{-1}\right) A, \tag{A.7}$$

which corresponds to a reverse propagation on the DAG. In the literature of AD, (A.7) is called reverse mode AD. In practice, the size of matrices M and A could be quite different, in which case, we might prefer forward or reverse mode AD over the other.

All four matrices B, M, L and A are very likely to be sparse. And usually $p \gg n$ and $p \gg m$. Thus the most expensive part of AD is usually solving the sparse lower triangular system $L^{-1}A$ for forward mode AD (A.6) or the sparse upper triangular system ML^{-1} for reverse mode AD (A.7). Solving a sparse lower triangular system is computationally equivalent with forward propagation. Two standard strategies of forward propagation are depth-first search and breadth-first search. In the literature of solving sparse triangular systems, the depth-first search approach is called Sparse Vector Method [28] and the breadth-first search approach is called Sparse Partitioned A^{-1} Method [11][11]. Both of these two methods are equivalent with the forward propagation described in section A.1 in terms of number of operations. However, in terms of massively parallel computing, the latter is preferred.

Bibliography

[1] F. Balmas, "Displaying dependence graphs: a hierarchical approach," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 16, no. 3, pp. 151–185, 2004.

[2] T. F. Coleman and W. Xu, "Fast (structured) newton computations," *SIAM Journal on Scientific Computing*, vol. 31, no. 2, pp. 1175–1191, 2008.

[3] T. F. Coleman and A. Verma, "Structure and efficient hessian calculation," Cornell University, Tech. Rep., 1996.

[4] T. F. Coleman, X. Xiong, and W. Xu, "Using directed edge separators to increase efficiency in the determination of jacobian matrices via automatic differentiation," in *Recent Advances in Algorithmic Differentiation*. Springer, 2012, pp. 209–219.

[5] T. F. Coleman and J. J. Moré, "Estimation of sparse jacobian matrices and graph coloring blems," *SIAM journal on Numerical Analysis*, vol. 20, no. 1, pp. 187–209, 1983.

[6] ——, "Estimation of sparse hessian matrices and graph coloring problems," *Mathematical Programming*, vol. 28, no. 3, pp. 243–270, 1984.

[7] M. J. Flynn, "Some computer organizations and their effectiveness," *Computers, IEEE Transactions on*, vol. 100, no. 9, pp. 948–960, 1972.

[8] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. DobbŠs Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[9] M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*. Cambridge university press, 2010.

[10] Nvidia, "Nvidia cuda programming guide," 2011.

[11] F. L. Alvarado and R. Schreiber, "Optimal parallel solution of sparse triangular systems," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, pp. 446–460, 1993.

[12] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM, 1967, pp. 483–485.

[13] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[14] A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang, "Heterogeneous computing: Challenges and opportunities," *Computer*, vol. 26, no. 6, pp. 18–27, 1993.

[15] T. F. Coleman, F. Santosa, and A. Verma, "Semi-automatic differentiation," in *Computational Methods for Optimal Design and Control*. Springer, 1998, pp. 113–126.

[16] A. Griewank and A. Walther, *Evaluating derivatives: principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics, 2008, vol. 105.

[17] U. Naumann, *The Art of Differentiating Computer Programs*. Society for Industrial and Applied Mathematics, 2012, vol. 24.

[18] T. F. Coleman and W. Xu, "Admat-2.0," 2009.

[19] D. Fournier, H. Skaug, J. Ancheta, J. Ianelli, A. Magnusson, M. Maunder, A. Nielsen, and J. Sibert, "In press. ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models," *Optimization Methods and Software*.

[20] P. J. Werbos, "Backwards differentiation in ad and neural nets: Past links and new opportunities," in *Automatic Differentiation: Applications, Theory, and Implementations*. Springer, 2006, pp. 15–34.

[21] T. F. Coleman, "The efficient evaluation of structured hessians by automatic differentiation," (to appear).

[22] T. F. Coleman and W. Xu, "The efficient evaluation of structured gradients (and underdetermined jacobian matrices) by automatic differentiation," (to appear).

[23] J. Nickolls and W. J. Dally, "The gpu computing era," *Micro, IEEE*, vol. 30, no. 2, pp. 56–69, 2010.

[24] D. C. Cireşan, U. Meier, L. M. Gambardella, and J. Schmidhuber, "Deep, big, simple neural nets for handwritten digit recognition," *Neural computation*, vol. 22, no. 12, pp. 3207–3220, 2010.

[25] "Simple automatic differentiation," 2013. [Online]. Available: https://github.com/vanci/SAD.git

[26] A. P. E. Coffman Jr and R. L. Graham, "Optimal scheduling for two-processor systems," *Acta Informatica*, vol. 1, no. 3, pp. 200–213, 1972.

[27] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. Society for Industrial and Applied Mathematics, 2011, vol. 22.

[28] W. Tinney, V. Brandwajn, and S. Chan, "Sparse vector methods," *Power Apparatus and Systems, IEEE Transactions on*, no. 2, pp. 295–301, 1985.