Sharing Storage Using Dirty Vectors

Bruce Christianson* Laurence Dixon* Steven Brown*

Abstract

Consider a computation F with n inputs (independent variables) and m outputs (dependent variables), and suppose that we wish to evaluate the Jacobian of F. Automatic differentiation commonly performs this evaluation by associating vector storage either with the program variables (in the case of forward-mode automatic differentiation) or with the adjoint variables (in the case of reverse). Each vector component contains a partial derivative with respect to an independent variable, or a partial derivative of a dependent variable, respectively. The vectors may be full vectors, or they may be dynamically managed sparse data structures. In either case, many of these vectors will be scalar multiples of one another. For example, any intermediate variable produced by a unary operation in the forward mode will have a derivative vector that is a multiple of the derivative for the argument. Any computational graph node that is read just once during its lifetime will have an adjoint vector that is a multiple of the adjoint of the node that reads it.

It is frequently wasteful to perform component multiplications explicitly. A scalar multiple of another vector can be replaced by a single multiplicative "scale factor" together with a pointer to the other vector. Automated use of this "dirty vector" technique can save considerable memory management overhead and dramatically reduce the number of floating-point operations required. In particular, dirty vectors often allow shared threads of computation to be reverse-accumulated cheaply. The mechanism permits a number of generalizations, some of which give efficient techniques for preaccumulation.

keywords: computational graph, copy-on-write, Jacobian, preaccumulation, reducing FLOP count, sparse vectors.

1 Introducing Dirty Vectors

Consider a computation F with n inputs (independent variables) u_i and m outputs (dependent variables) z_j, and suppose that we wish to evaluate the Jacobian of F. Automatic differentiation (AD) [13] commonly performs this evaluation by associating vector storage either with the program variables (in the case of forward-mode AD) or with the adjoint variables (in the case of reverse) [8, 11]. Each vector component contains a partial derivative with respect to an independent variable, or a partial derivative of a dependent variable, respectively. The vectors may be full vectors, or they may be dynamically managed sparse data structures [1, 9].

Many of these vectors will be scalar multiples of one another. For example, if the program contains the line

$$y := \sin(x), \tag{1}$$

*{B.Christianson, L.C.Dixon, S.Brown}@herts.ac.uk ; Numerical Optimisation Centre, Faculty of Information Sciences, University of Hertfordshire : Hatfield, AL10 9AB, England, Europe

then in the forward node we will have $y' = \cos(x)x'$, where x' and y' are respectively the vectors with components $\partial x/\partial u_i$ and $\partial y/\partial u_i$. In fact, any intermediate variable produced by a unary operation in the forward mode will have a derivative vector that is a scalar multiple of the derivative vector of the argument.

Dually, in reverse mode, any graph node (Wengert variable) [13, §4] that is read just once during its lifetime will have an adjoint vector that is a scalar multiple of the adjoint of the node that reads it. For example, if the program contains the line

$$x := v * w, \tag{2}$$

and this is the only use made of w before it is overwritten, then we will have $\bar{w} = v\bar{x}$, where $\bar{w}$ and $\bar{x}$ are the vectors with components $\partial z_j/\partial w$ and $\partial z_j/\partial x$.

It is frequently wasteful of both space and time to perform such component-by-component multiplications explicitly. A scalar multiple of another vector can be replaced by a "dirty vector" consisting of a single multiplicative "scale factor" together with a pointer to the other vector. A Boolean flag indicates whether scaling multiplications need to be carried out when the dirty vector contents are subsequently accessed.

Here is an appropriate data structure

```
type dirty_vector is
  normalized :  boolean
  factor :  real
  pv :  pointer to vector
end type
```

The vector `v` to which `pv` points may be a full or sparse vector. Every element of the vector `v` is to be regarded as scaled (multiplied) by the scale `factor`. Several different dirty vectors (possibly with different scale factors) may point to the same vector `v`. The header of vector `v` includes a usecount, which indicates how many dirty vectors point to `v`. Whenever a dirty vector is initialized to a multiple of `v` then the usecount of `v` is incremented. Conversely, the call `release(v)` will decrement the usecount of `v` and, if the count reaches zero, will garbage-collect the storage for `v`. We adopt the convention that a dirty vector has the value zero if the pointer `pv` is null. If the actual values of a dirty vector are required then we can explicitly renormalize

```
procedure renormalize (d :  dirty_vector) is
  if not d.normalized then
    declare t :  vector = allocate (size_of (d.pv.target))
    t.usecount := 1
    d.normalized := true
    for each component i of d.pv.target do
      t[i] := d.factor * d.pv.target[i]
    enddo
    release (d.pv.target)
    d.factor := 1.0
    d.pv := pointer_to (t)
  endif
end procedure
```

The forward accumulation step $y := \sin(x)$ described earlier is implemented as follows

```
y.dash.normalized := false
if x.dash.normalized then
  y.dash.factor := cos (x.value)
else
  y.dash.factor := x.dash.factor * cos(x.value)
endif
y.dash.pv := x.dash.pv
y.dash.pv.target.usecount := y.dash.pv.target.usecount + 1
y.value := sin (x.value)
```

This code contains an explicit check of `normalized` to avoid an unnecessary multiplication by one. In practice, the check will be done by the hardware rather than explicitly in code. Indeed, checking in code may lead to a slowdown because of failed branch prediction in the instruction prefetch unit. From now on, we do not write such checks explicitly in our code, but assume them in our operation count analysis.

The reverse accumulation code for $x := v * w$ is

```
if w.bar.pv = null then
  w.bar.normalized := false
  w.bar.factor := x.bar.factor * v.value
  w.bar.pv := x.bar.pv
  w.bar.pv.taget.usecount := w.bar.pv.taget.usecount + 1
else
  renormalize (w.bar)
  for each component i of x.bar.pv.target do
    w.bar.pv.target[i] := w.bar.pv.target[i]
        + x.bar.pv.target[i]*v.value
  enddo
endif
```

followed by similar code for `v.bar`.

2 Performance Analysis

The deployment of dirty vectors can save a large amount of memory management overhead and storage and dramatically reduce in the number of floating-point arithmetic operations required for derivative calculation.

In this section we examine the conditions under which a saving is produced. For this analysis, we assume that the following floating-point operations are available:

$a + x$		unary plus
$a - x$		unary minus
$a * x$	(d)	unary multiply
$f(x)$	(d)	unary non-linear function (includes $\sin x, 1/x, \ln x$ etc.)
$x + y$		binary plus
$x - y$		binary minus
$x * y$	(d)	binary multiply

Here x and y are program variables, and a is any constant. Operations marked with (d) are referred to as *dilations*.

Consider a computational graph (or Wengert list) for F. Suppose that it contains N nodes (elements), indexed by integers $i \in \{1, \ldots, N\}$. Each node has an associated value v_i. Apart from the first n nodes, which correspond to independent variables, each node also has associated with it an operation (drawn from the above list) and either one or two arguments. Assume that the vector $v_i{}'$ associated by the forward mode with v_i has p_i elements and that the vector $\bar{v}_i$ associated by the reverse mode with v_i has q_i elements. If full vectors are used then $p_i = n$ and $q_i = m$ for all i.

2.1 Examples

Example. Suppose in forward mode that the program contains the line

$$y := \log(\sin(x)), \tag{3}$$

corresponding to the Wengert list fragment:

$$v_{102} = \sin(v_{101}), \quad v_{103} = \log(v_{102}) \tag{4}$$

and no other use is made of v_{102}.

Then the use of dirty vectors saves at least $p_{101} - 2$ multiply operations, since we need never evaluate v'_{102} explicitly. A naive vector forward AD implementation would require p_{101} multiplications by $\cos(v_{101})$ to form v'_{102} and a further p_{101} multiplications by $1/v_{102}$ to form v'_{103}. If dirty vectors are used, then `v102.dash.pv` points to the same values as `v101.dash.pv` but has `v102.dash.factor = v102.dash.factor*cos (v101.value)` evaluated at a cost of one multiply. The vector `v103.dash.pv` initially also points to the same values as `v101.dash.pv` but has `v103.dash.factor = v102.dash.factor/v102.value` requiring one further multiply. Finally, before we can use the components of `v103`, they must be renormalized, at a cost of p_{101} multiply operations. We defer this renormalization until we are sure that it is required. Only at this point must actual vector storage for `v103.dash` be allocated. We never allocate or garbage-collect vector storage for `v102.dash`, nor do we traverse the elements of `v101.dash.pv` to form `v102.dash`.

Example. In forward mode

$$v_{102} = \log(v_{101}), \quad v_{103} = \sin(v_{102}), \quad v_{104} = \cos(v_{102}) \tag{5}$$

and no other use is made of v_{102}. This saves at least $p_{101} - 3$ multiply operations, since we never evaluate v'_{102} explicitly, but require an extra multiply operation to evaluate `v104.dash.factor`.

Example. In forward mode

$$v_{102} = \log(v_{101}), \quad v_{103} = \sin(v_{102}), \quad v_{104} = v_1 + v_{102} \tag{6}$$

produces no saving in operation count, although it does reduce the memory overhead.

Example. In forward mode

$$v_{102} = \log(v_{101}), \quad v_{103} = \sin(v_{102}), \quad v_{104} = v_1 * v_{102} \tag{7}$$

produces a saving of $p_{103} - 3$ multiply operations, provided no other use is made of v_{102}.

Example. In forward mode

$$v_{102} = v_{101} + 2.0, \quad v_{103} = \sin(v_{102}) \tag{8}$$

produces no saving in operation count, although it does save some vector storage.

Example. Suppose in reverse mode the program contains the line

$$y := \log(v * w) \tag{9}$$

corresponding to the Wengert list fragment

$$v_{103} = v_{101} * v_{102}, \quad v_{104} = \log(v_{103}) \tag{10}$$

and no other use is made of v_{103}.

Then the use of dirty vectors saves at least $q_{104} - 3$ multiply operations, since we need never evaluate v'_{103} explicitly.

Example. In reverse mode

$$v_{103} = \log(v_{101}), \quad v_{104} = v_{102} * v_{103} \tag{11}$$

and no other use is made of v_{103}.

Then the use of dirty vectors saves at least $q_{104} - 2$ multiplies.

2.2 Benefits

In forward mode, dirty vectors save operations whenever a node produced by a unary dilation operation is read only by dilation operations. In the case of reverse mode, the dual condition is that a node produced by a dilation operation be read just once, and only by a dilation operation.

DEFINITION 2.1. *Write $i \prec j$ if node j has node i as an argument. Let a_i be the number of nodes that are arguments to node i. Let r_i be the number of nodes that have node i as an argument. Call node i* read-once *if $r_i = 1$. Let $U(i)$ be the assertion that operation i is unary, $D(i)$ that operation i is a dilation, and $R(i)$ that node i is read-once.*

Let M_p be the subset of $\{1, \ldots, N\}$ consisting of those i for which

$$U(i) \ \& \ D(i) \ \& \ (i \prec j \Longrightarrow D(j)).$$

Dually, let M_q consist of those i for which

$$R(i) \ \& \ D(i) \ \& \ (i \prec j \Longrightarrow D(j)).$$

PROPOSITION 2.1. *The use of dirty vectors in forward mode saves at least*

$$\sum_{i \in M_p} (p_i - r_i - 1) \text{ multiply operations.}$$

The use of dirty vectors in reverse mode saves at least

$$\sum_{i \in M_q} (q_i - a_i - 1) \text{ multiply operations.}$$

Note that the savings given in Proposition 2.1 are conservative.

The saving in storage management is more difficult to quantify. In principle, the cost of one vector allocate and one release, together with the overhead of accessing and traversing the vector elements, can be saved for each element of M_p or M_q. Although the sharing of vectors may result in additional page faults when the shared nodes are not nearby, it is always possible to *copy* the dirty vector instead of sharing. The bottom line is that the careful introduction of dirty vectors cannot make memory management overhead worse than it already is, and may make it considerably better.

Obtaining the full theoretical benefit available would require a number of implementation subtleties, such as a shadow-copy mechanism and deferred updates. These refinements are unnecessary in practice. Experiment shows that naive implementations of the type outlined in the preceding section deliver equivalent performance in all but carefully crafted pathological cases.

In the forward case, the dirty vector technique amounts to a run-time variant of the technique of *hoisting* [2]. The dirty vector technique does not require the unary operations to be adjacent in the Wengert list: in graphical terms they may form a tree rather than a linear sequence. In the reverse case, dirty vectors provide at least the same savings in operation count savings and temporary variable storage overhead as the compile-time technique of preaccumulating individual assignment statements used by ADIFOR [3]. The dirty vector technique allows additional automatic savings across sequences of assignment statements, even when these are separated by procedure call boundaries or occur in different loop iterations.

A further benefit is obtained by modifying the procedure for adding together two dirty vectors that point at the same `v`, so that it simply adds the factors. This is useful when reverse accumulating sequences of assignment statements in which some variables enter more than once on the right hand side.

2.3 Limitations

Using dirty vectors does have some performance limitations. In the case of forward mode, the maximum size of M_p is M_u, where M_u is the number of unary operations in the calculation of F. Typically $M_u < M_b$, where M_b is the number of binary operations in the calculation of F. If no suitable sequences of consecutive operations exist, then no saving occurs.

In the case of reverse mode, the situation is more complex. We may, if we are lucky, have an ideal situation: a number of expression trees with no shared intermediate variables, which are combined near the end of the calculation to form the dependent variables. In this case we get the entire Jacobian (which may be full) for a floating-point cost of about three evaluations of F. However, we are unlikely to be so lucky in general. In the (equally unlikely) worst case, where every node (apart from the dependent variables) is read twice, no saving occurs at all. Both of these extremes are illustrated in the next section. Neither case is typical.

We can make a rough theoretical estimate of the typical expected saving from using dirty reverse in the absence of a high level of sparsity. Let M_r be the number of nodes (including independent variables) that are read exactly r times. Then

$$\text{total nodes } = N = n + M_u + M_b = m + M_1 + M_2 + M_3 \cdots$$

$$\text{total arcs } = M_u + 2M_b = M_1 + 2M_2 + 3M_3 + \cdots$$

so

$$M_2 + 2M_3 + 3M_4 + \cdots = M_b + m - n.$$

Experience shows that most computational graphs consist of a large proportion of nodes that are read once, and a small proportion that are read many times. If dirty vectors are used and m is reasonably large, then the reverse accumulations to the read-once nodes have negligible cost relative to the others. The number of $m-$vector accumulation steps required for the read-many nodes is

$$\begin{aligned} A &= 2M_2 + 3M_3 + 4M_4 \cdots \\ &= (M_b + m - n) + (M_2 + M_3 + M_4 + \cdots) \\ &= (M_b - n) + (N - M_1), \end{aligned}$$

so A/M_b is likely to be not much larger than one.

Since the read-many nodes are relatively infrequent, each path joining them is likely to contain at least one dilation, so each arc in A is likely to require an $m-$vector multiply-and-add. Even if it does not, the overhead of accessing and traversing the vector elements typically outweighs the floating-point arithmetic costs. In either case, the cost of reverse is about $M_u + 2M_b$ without dirty vectors, compared with $A \approx M_b$ to first order with dirty vectors.

Of course, this analysis is crude, and takes no precise account of sparsity or of exactly where the dilations fall. However, it is consistent with experiment, as illustrated by the last test problem in the next section.

3 Performance Experience

We have implemented dirty vectors in Fortran-90 at the Numerical Optimisation Centre at Hatfield, in a package that runs on top of a sparse vector implementation. In this section we use dirty vectors to reverse-accumulate Jacobians for four test functions. The first two of these are designed to illustrate cases where dirty vectors produce no savings at all. The third illustrates the case of optimum savings, and the fourth is a realistic problem.

F_1 is defined as follows:

$n = m = 50$

$v_{51} = \sum_{i=1}^{50} u_i$
$v_i = v_{i-1} * v_{i-2} \qquad i = 52 \ldots 10,050$

$z_j = v_{10,000+j} + j \qquad j = 1 \ldots 50$

F_2 is defined as follows:

$n = m = 50$

$z_j = g(u_j) + j \qquad j = 1 \ldots 50$

where $g(x) = \underbrace{\sin(\sin \ldots \sin(x))}_{200 \ times}$

F_3 is defined as follows:

$$n = m = 50$$

$$v_{51} = \sum_{i=1}^{50} u_i$$
$$v_i = \sin(v_{i-1}) \qquad i = 52 \ldots 10,050$$

$$z_j = v_{10,050} + j \qquad j = 1 \ldots 50$$

The function F_4 is a constraint function vector evaluation taken from a standard trajectory problem. This is then expressed as a sum of squared terms and solved by using a least squares optimization algorithm from the OPTIMA subroutine library [12], linked to a reverse accumulation package. For further details, visit our WorldWideWeb page at `http://www.cs.herts.ac.uk/~matqmb/ad.html`.

Table 1 shows the time in seconds to evaluate the Jacobian matrix of each test function using a variety of different reverse accumulation approaches. In the first column, the Jacobian is evaluated without using vector arithmetic at all, by repeated traversal of the computational graph, once for each component. In the last three columns, a vector is associated with each adjoint: in column two, full vectors are used; in column three, sparse vectors are used; and in column four, dirty vectors (pointing to sparse vectors) are used. The target platform was a DEC Alpha.

Problem	Multipass Scalar	Full Vector	Sparse Vector	Dirty Vector
F_1	3.1	4.0	4.6	4.6
F_2	5.3	6.3	0.4	0.4
F_3	6.1	4.0	5.0	0.3
F_4	0.28	0.25	0.21	0.15

TABLE 1

Table 1 : Jacobian matrix evaluation (time in seconds)

The function F_1 is designed in such a way that every intermediate node is read twice and the adjoint vectors are full at each node. As expected, dirty vectors give no improvement.

The function F_2 contains no shared threads, and each node affects only one of the dependent variables. Each adjoint vector contains a single nonzero component. The use of sparse vectors gives a big improvement over full vectors: but, as expected, the use of dirty vectors gives no further improvement.

The function F_3 contains a single long thread that affects all dependent variables. The majority of adjoint vectors are full. In this case the use of dirty vectors produces a dramatic improvement, in spite of the fact that the dirty vector figure includes the overhead of using sparse vectors.

Even though the internal sparsity in F_4 fills out quite rapidly over successive timesteps, the use of dirty vectors produces a 40% saving over the use of full vectors.

4 Going Further

Dirty vectors represent an attractive proposition because they require few mechanisms not already present in a sophisticated AD implementation, and they can provide a substantial performance benefit at little or no cost. But the benefit of dirty vectors in the form so

far described is limited. To obtain further sizable performance improvements, we need to extend the dirty vector technique to vectors that are linear combinations of a small number of other (long) vectors. This extension amounts to a run-time technique for *preaccumulation* of shared threads of computation [14] (also called *interface narrowing* [10]), corresponding to the change-of-variable rule from calculus.

Under certain circumstances, it is advantageous to "preaccumulate" a section of the computational graph [4, 7], replacing the section by a much smaller set of nodes representing the output values from that segment of the graph, together with their derivatives with regard to the input variables for that section. These preaccumulated values are then used as the basis of application for the chain rule on a subsequent accumulation pass (which may itself be a recursive preaccumulation of a larger part of the graph.)

The dirty vector technique can be generalized and combined with an existing sparse vector AD package to give a cheap implementation of certain forms of preaccumulation. We introduce a (short) sparse vector each of whose elements is a record of type `dirty_vector`, with the pointer field `pv` being used as the sparse component index. (The use of pointers as index values is a device to minimize page faults on traversal.) The adjoint of a node is a sparse vector of the new type.

The user gives a hint about the location of a convenient graph section for preaccumulation by making a call to a special routine before and after evaluating f. The decision whether actually to preaccumulate a graph section or subgraph can be made at run time. For example, in reverse mode, consider a subfunction f of F. Suppose that f produces q outputs each of which affects r outputs of F, where $r > n$. Suppose also that f has p inputs which may themselves be further subfunctions or independent variables (i.e. inputs to F). If the computational effort of evaluating the Jacobian of f is equivalent to W vector multiply-and-add operations, then it will certainly be better to preaccumulate f if

$$W > \frac{pqr}{r-q}.$$

If the Jacobian of f is sparse, or if dirty vectors are being used within f to obtain the Jacobian, then preaccumulation may give a time saving under even weaker conditions. In effect, we are treating the outputs of f as new dependent variables and eliminating them (by multiplying them out) when the beginning of f is reached on the reverse pass.

By making forward quantities sparse vectors, and adjoint quantities pointers to nodes, as described in [5], the techniques discussed here can also be adapted to extract Hessians and higher derivatives in the same way as there.

References

[1] M. C. Bartholmew-Biggs, L. Bartholomew-Biggs and B. Christianson, 1994, Optimization and Automatic Differentiation in Ada: Some Practical Experience, Optimization Methods and Software **4** 47–73

[2] C. Bischof, 1992, Issues in Parallel Automatic Differentiation, pp 100–113 *in* Automatic Differentiation of Algorithms: Theory, Implementation and Applications, A. Griewank and G.F. Corliss (eds), SIAM : Philadelphia

[3] C. Bischof and P. Hovland, 1992, Using ADIFOR to Compute Dense and Sparse Jacobians, ANL/MCS-TM-158 Argonne National Laboratory, Illinois

[4] B. Christianson, 1992, Automatic Hessians by Reverse Accumulation, IMA Journal of Numerical Analysis **12** 135–150

[5] B. Christianson, 1993, Reverse Accumulation of Functions Containing Gradients, Numerical Optimisation Centre Technical Report 278, School of Information Sciences, University of Hertforshire, Hatfield, UK.

[6] B. Christianson, 1994, Reverse Accumulation and Attractive Fixed Points, Optimization Methods and Software **3** 311-326

[7] B. Christianson and L.Dixon, 1992, Reverse Accumulation of Jacobians in Optimal Control, Numerical Optimisation Centre Technical Report 267, School of Information Sciences, University of Hertforshire, Hatfield, UK.

[8] A. Griewank, 1989, On Automatic Differentiation, pp 83–108 in Mathematical Programming: Recent Developments and Applications, M. Iri and K. Tanabe (eds), Kluwyer, Tokyo, Japan

[9] A. Griewank, 1993, Some Bounds on the Complexity of Gradients, Jacobians and Hessians, in Complexity in Numerical Optimization, P.M. Pardolos (ed.), World Scientific

[10] P. Hovland *et al*, 1995, Efficient Derivative Codes through Automatic Differentiation and Interface Contraction: An Application in Biostatistics", SIAM Journal of Scientific Computing, *to appear*

[11] M. Iri, 1984, Simultaneous Computation of Functions, Partial Derivatives and Estimates of Rounding Errors - Complexity and Practicality, Japan Journal of Applied Mathematics **1**(2) 223–252

[12] The OPTIMA Manual, 1989, Numerical Optimisation Centre, Faculty of Information Sciences, University of Hertfordshire : Hatfield, England, Europe.

[13] L.B. Rall and G.F. Corliss, 1996, An Introduction to Automatic Differentiation, this volume.

[14] Yu. M. Volin *and* G.M. Ostrovskii, 1985, Automatic Computation of Derivatives with the use of the Multilevel Differentiation Technique, Computers and Mathematics with Applications **11**(11) 1099–1114