Automatic Differentiation: Applications, Theory, and Implementations,
Bücker, M.; Corliss, G.; Hovland, P.; Naumann, U.; Norris, B. (Eds.)
Lecture Notes in Computational Science & Engineering, Volume 50, p111-120, Springer, 2006.
ISBN 3-540-28403-6

1

Improving the Performance of the Vertex Elimination Algorithm for Derivative Calculation*

M. Tadjouddine[1], F. Bodman[2], J.D. Pryce[2], and S.A.Forth[1]

[1] Applied Mathematics & Operational Research, ESD
Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK
{M.Tadjouddine, S.A.Forth}@cranfield.ac.uk
[2] Department of Information Systems,
Cranfield University (RMCS Shrivenham), Swindon SN6 8LA, UK
j.d.pryce@cranfield.ac.uk

Summary. In previous work [TOMS, 2004, 30(3), 266–299], we used Markowitz-like heuristics aiming to find elimination sequences that minimise the number of floating-point operations (flops) for vertex elimination Jacobian code. We also used the depth-first traversal algorithm to reorder the statements of the Jacobian code with the aim of reducing the number of memory accesses. In this work, we study the effects of reducing flops or memory accesses within the vertex elimination algorithm for Jacobian calculation. On RISC processors, we observed that for data residing in registers, the number of flops gives a good estimate of the execution time, while for out-of-register data, the execution time is dominated by the time for memory access operations. We also present a statement reordering scheme based on a greedy-list scheduling algorithm using ranking functions. This statement reordering will enable us to trade-off the exploitation of the instruction level parallelism of such processors with the reduction in memory accesses.

Key words: vertex elimination, Jacobian accumulation, performance analysis, statement reordering, greedy-list scheduling algorithms

1.1 Introduction

Many scientific applications require the first derivatives (at least) of a function $\mathbf{f} : \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{y} \in \mathbb{R}^m$ represented by computer code. This can be obtained using automatic differentiation (AD)[1, 2]. We assume the function code has no loops or branches; alternatively, our work applies to *basic blocks* of more complicated code. From the program, we build up the data dependence

*This work was partly supported by EPSRC under grant GR/R21882

graph (DDG), or computational graph, of the function **f** as a Directed Acyclic Graph (dag) $G=(V,E)$, with vertex-set V and edge-set E. A vertex v_i represents a floating-point assignment of the original code; an edge $(v_j, v_i) \in E$ represents the data dependence relationship $v_j \prec v_i$ meaning v_j appears on the right hand side of the assignment that computes v_i. Logically, E and the relation $\prec$ are the same. Code may contain overwrites of variables; we assume these removed by converting to Static Single Assignment form [3], so that a variable may be identified with the statement that computes it. We have $|V|=n+p+m=N$ where n, p, m are respectively the number of independent, intermediate and dependent vertices. We 'linearise' G by labelling its edges with local partial derivatives. Finally, we eliminate, in some order termed the elimination sequence, all intermediate vertices so that G is rendered bipartite. This process, the *vertex elimination approach*, can be found in [2, 4, 5, 6].

As shown in [2, 4], the linearised graph G can be viewed as an $N \times N$ sparse lower triangular matrix $\mathbf{C}=(c_{ij})$ and $\mathbf{C}-\mathbf{I}_N$ is called the *extended Jacobian*. The Jacobian $\mathbf{J}$ can be obtained by solving a sparse, triangular linear system with coefficient matrix $\mathbf{C}-\mathbf{I}_N$ using some form of Gaussian elimination.

Since p, the number of intermediate vertices, tends to be large even in medium-sized applications, the performance of the vertex elimination algorithm can be degraded by fill-in. The floating point operations (flops) performed, and the fill-in, are determined by the elimination sequence. The question one would *ideally* like to answer is "Which elimination sequence gives the fastest code on a particular platform?". As a platform-independent approximation to this problem one may ask "Which elimination sequence minimises fill-in [respectively flop-count]?". For a sparse symmetric positive definite system of linear equations, the fill-in problem is NP-complete [7] and it is suspected the same holds for our problem. Therefore, in practice a near-optimal sequence must be found by some heuristic algorithm. Our premiss is that such sequences allow us to generate faster Jacobian code.

Goedecker and Hoisie [8] report that performance of numerically intensive codes on many processors is a low percentage of nominal peak performance. There is a gap between CPU performance growth (around 55% per year) and memory performance growth (about 7% per year) [9].

To enhance performance, it would appear crucial to keep the memory traffic low. In this paper, we study two aspects of the vertex elimination algorithm. First, we study how the number of floating point operations in the Jacobian code relates to its performance on various platforms. Second, we study how reordering the statements of the Jacobian code affects memory accesses and register usage. For these purposes, we generated Jacobian codes using Markowitz-like strategies and statement reordering and inspected the assembler from different processors and compilers. We studied how the execution time is affected by the number of flops, and amount of memory traffic (loads and stores). We observed:

- A reordering of the Jacobian code's statements can improve its performance by a significant percentage when this reduces the memory traffic.
- For in-register data, the execution time is dominated by the number of floating point operations and a reduction of floating point operations gave further performance improvement.
- For out-of-register data, the execution time is dominated by the number of load and store operations and a reordering that reduced these memory access operations enhances Jacobian code performance.

Similar behaviour is found in performance analysis of other numerical codes, see for example [8]. This paper presents the argument in the context of semantic augmentation of numerical codes as is carried out in AD of computer programs. We also describe planned work to improve performance of Jacobian code, produced by vertex elimination, by reordering the statements using standard instruction scheduling algorithms.

1.2 Heuristics

Solving large linear systems by Gaussian elimination can be prohibitive due to the amount of fill-in. As said above, heuristic approximate solutions are used to the NP-complete problem of finding an elimination ordering to minimise fill-in. Over the past four decades several heuristics aimed at producing low-fill orderings have been investigated. These algorithms have the desired effects of reducing work as well. The most widely used are *nested dissection* [10, 11] and *minimum degree*: the latter, originating with the *Markowitz method* [12], is for example studied in [13].

Nested dissection, first proposed in [11], is a recursive algorithm which starts by finding a *balanced separator*. A balanced separator is a set of vertices that when removed partition the graph into two or more components, each composed of vertices whose elimination does not create fill-in in any of the other components. Then the vertices of each component are ordered, followed by the vertices in the separator. Unlike nested dissection that examines the entire graph before reordering it, the minimum degree or Markowitz-like algorithms tend to perform local optimisations. At each elimination step, such a method selects a vertex with minimum cost or degree, eliminates it and looks for the next vertex with the smallest cost in the new graph.

As described in [4, 14], we built up the linearised computational graph in the following two ways:

1. Statement Level (SL) in which local derivatives are computed for each statement, no matter how complex its right-hand side.
2. Code List (CL) in which local derivatives are computed for each statement after the code has first been rewritten so that each statement performs a single unary or binary operation.

Then, we applied the following heuristics [4, 5] to the resulting graphs:

- Forward (F): where intermediate vertices are eliminated in forward order.
- Reverse (R): where intermediate vertices are eliminated in reverse order.
- Markowitz (M): at each elimination stage a vertex v_j of smallest Markowitz cost is eliminated. This cost is defined as the product of the number of predecessors of v_j times the number of its successors in the current, partly eliminated, graph .
- VLR (V): as Markowitz but using the VLR cost function defined by

$$\mathrm{VLR}(v_j) = \mathrm{mark}(v_j) - \mathrm{bias}(v_j),$$

  with $\mathrm{bias}(v_j)$ a fixed value for v_j, namely the product of the number of independent vertices and the number of dependent vertices that v_j is connected to.
- Any of the above with Pre-elimination (P): vertices with single successor are eliminated first and then one of Forward, Reverse, Markowitz or VLR order is applied to those remaining.

We also used a Depth-First Traversal (DFT) algorithm [14] to reorder statements of the obtained Jacobian code, without altering dependencies between statements, in the hope of further performance improvement.

1.3 Performance Analysis

We consider two of the test problems reported in [4]: the Human Heart Dipole (HHD) from the Minpack 2 test suite [15] and the Roe flux calculation (ROE) [16]. These routines were differentiated using the AD tool ELIAD [4, 14] using the heuristics listed in Section 1.2. All the Jacobian codes were compiled on different platforms with maximum optimisation level, and run for a number of times carefully calculated for each platform [4].

To assess the performance of the ELIAD generated Jacobians, we studied the assembler from different platforms, counting the number of loads, stores and flops ('L', 'S' and 'Flops' in the tables) after compiler's optimisations.

Table 1.1 shows the results of our study from the SUN Ultra 10 processor with 440 MHz, 32 KB L1 cache, 2 MB L2 cache, and using the Workshop f90 6.0 Compiler. The observed time Obs-Time is the CPU time obtained by averaging a certain number of evaluations and runs, see [4] for details.

Table 1.2 shows some runtime predictions using a very simple model approximating the runtime via the memory access count and the flops count. This approximate model estimates the following quantities: T_F, the time taken by the floating point operations

$$T_F = \frac{\text{Flops}}{\text{flops rate}} \times \text{cycle time} \times \text{latency} \tag{1.1}$$

and T_M, the time taken by memory access operations

Table 1.1. Performance data for the HHD and the Roe flux test cases on the Ultra10 platform, Obs-Time in μs

	HHD			ROE		
Technique	Obs-Time	Flops	L+S	Obs-Time	Flops	L+S
SL-F	0.77	150	179	11.38	1732	2489
SL-R	0.79	148	188	7.26	1432	1600
CL-F	0.73	150	184	16.57	1843	3406
CL-R	0.80	148	201	6.98	1496	1655
SL-P-F	0.83	172	205	6.64	1580	1718
SL-P-R	0.73	172	182	6.11	1382	1626
CL-P-F	0.72	150	174	6.24	1580	1662
CL-P-R	0.71	148	182	5.81	1382	1609
SL-P-F-DFT	0.78	168	214	7.49	1584	1855
SL-P-R-DFT	0.66	164	180	5.73	1382	1466
CL-P-F-DFT	0.80	168	200	7.42	1587	1923
CL-P-R-DFT	0.66	164	167	5.84	1387	1305
SL-P-M	0.69	150	181	6.91	1524	1803
SL-P-M	0.83	168	214	5.71	1365	1507
CL-P-V	0.69	150	181	7.40	1524	1824
CL-P-V	0.83	168	214	6.17	1364	1503
SL-P-M-DFT	0.73	150	184	8.03	1529	1958
SL-P-V-DFT	0.80	168	200	6.19	1366	1375
CL-P-M-DFT	0.73	150	184	7.58	1532	1945
CL-P-V-DFT	0.80	168	200	5.59	1369	1362

$$T_M = \frac{(\mathrm{L} + \mathrm{S})}{\text{memory access rate}} \times \text{cycle time} \times \text{latency}. \tag{1.2}$$

The Ultra 10 processor can perform up to 2 flops per cycle (its flops rate is 2) with a latency of 3 cycles and 1 load or 1 store (its memory access rate is 1) with a latency of 2 cycles, and uses in-order execution [8] of instructions.

In Table 1.2, we represent the performance measures for a sample of methods shown in Table 1.1. The column 'Nom. flops' is the nominal flops count obtained from the source text. This table illustrates the following observations:

- A small reduction of flops count does not necessarily imply a reduction of the actual runtime Obs-time.
- T_M tends to be a better estimate of Obs-time than is T_F.
- The statement reordering improved performance when it reduced the number of memory accesses.

These results have led us to believe that the runtime is more correlated with the memory accesses (loads and stores) than with the flops count. To further investigate this, we performed a linear regression analysis using the `regress` function of `MATLAB`'s statistics toolbox [17]. For both test cases in Table 1.1, we form the linear model:

$$T = aX + b + \epsilon \tag{1.3}$$

Table 1.2. A sample of methods applied to Roe flux on the Ultra 10 (timings in μs)

Technique	Nom. flops	Flops	L+S	T_F	T_M	Obs-time
SL-P-V-DFT	1462	1366	1375	4.65	6.26	6.19
CL-P-V-DFT	1578	1369	1362	4.68	6.20	5.59
CL-P-F	1742	1580	1662	5.40	7.56	6.24
CL-P-F-DFT	1742	1587	1923	5.40	8.74	7.42
SL-P-R	1505	1382	1626	4.71	7.40	6.11
SL-P-R-DFT	1505	1382	1466	4.71	6.66	5.73

in which X represents the vector of flops or memory accesses (loads + stores), b a constant vector, ϵ a residual vector and a a vector of parameters. Table 1.3 shows the 'explained variability' that is one of the statistics returned by `regress`, and the norm of the residual ϵ from the regression.

Table 1.3. MATLAB's `regress` results of the regression analysis of data of Table 1.1

Model	variability	$\|\epsilon\|_2$
$T = a_1\text{Flops} + b_1 + \epsilon_1$	0.87	8.7
$T = a_2(\text{L+S}) + b_2 + \epsilon_2$	0.99	3.2

The flops explain about 87% of the variability in the observed time T, whereas the loads and stores explain about 99%. Furthermore, the (loads and stores) model has a better residual compared to the flops model. It is important to reduce the flops count in numerical calculations but it is even more crucial to minimise memory traffic. These experiments suggest consideration of code reordering techniques, data structures, and other optimisation techniques that reduce the amount of memory accesses if we aim to generate efficient derivative code even for medium-sized applications.

1.4 A Statement Reordering Scheme

In [4, 14], we used a Statement Reordering Algorithm (SRA) based on G', the DDG of the statements of the derivative code. By depth first traversal, for each statement s it tries to place the statements on which s depends, close to s. It was hoped this would speed up the code by letting the compiler perform better register usage since, in our test cases, cache misses were shown not to be a problem [14]. The benefits were inconsistent, probably because this does not account for the instruction level parallelism of modern cache-based machines and the latencies of certain instructions.

In this work, we plan to encourage the compiler to exploit instruction level parallelism and use registers better, by a SRA that gives priority to certain statements via a ranking function. The compiler's instruction scheduling and register allocation work on a dependency graph G'' whose vertices are machine

code instructions. We have no knowledge of G'', so we work on the DDG G' on the premiss that our 'preliminary' optimisation will help the compiler generate optimised machine code. In the next sections, we shall use the instruction scheduling approach used in for instance [18], and the ranking function ideas of [19, 20, 21, 22] on a simple virtual processor.

1.4.1 The Processor Model

We consider the following simple model of a superscalar machine, similar to that of [23]. It has an unlimited number of floating-point (and other) 'registers'. It has one pipelined functional unit (FU) that can perform any scalar assignment-statement in our code, however complicated, in 2 clock cycles, including loading any number of operands from registers, computing, and storing the result in a register. An assignment-statement that does no processing (a simple copy) is assumed to take 1 cycle. A statement can be issued to the FU at each cycle, however data dependencies may 'stall' it: e.g. if the code `c=a*b; d=a+c` is begun at time $t = 0$, we cannot issue the second statement at $t = 1$ because `c` is not yet available.

We develop an algorithm that, within this model, tries to remove stalling by re-ordering code statements. Coarse-grained though the model is, we hope it imitates enough relevant behaviour of current superscalar architectures, to produce re-orderings that give speedup in practice.

1.4.2 The Derivative Code and its Evaluation

Since the original code was assumed branch- and loop-free the same is true of the derivative code. It includes original statements v_i of f's code as well as statements to compute the local derivatives c_{ij} that are the nonzeros of the extended Jacobian $\mathbf{C}$ before elimination. But the bulk of it is *elimination statements*. As originally defined in [6], these take the basic forms $c_{ij} = c_{ik}c_{kj}$ or $c_{ij} = c_{ij} + c_{ik}c_{kj}$, and typically a c_{ij} position is 'hit' (updated) more than once, needing non-trivial renaming of variables to put it into the needed single-assignment form.

Though not strictly necessary, we assume the VE process has been rewritten in 'one hit' form, e.g. by the inner product approach of [24]. That is, that each c_{ij} occurring in it is given its final value in a single statement, of the form either $c_{ij} := c^0_{ij} + \sum_{k\in K} c_{ik}c_{kj}$ if updating an original elementary derivative, or $c_{ij} := \sum_{k\in K} c_{ik}c_{kj}$ if creating fill-in. Here K is a set of indices that depends on i, j and on the elimination order used, and c^0_{ij} stands for an expression that computes the elementary derivative, $\partial v_i/\partial v_j$. The result is that the derivative code is automatically in single-assignment form.

Its graph $G' = (V', \prec')$ — where V' is the set of statements and $\prec'$ the data dependence relationship — is a dag. A *schedule* π for G' assigns to each statement (denoted s in this section) a start-time in clock-cycle units, respecting data dependencies subject to the constraints of our processor model. It is

a one-to-one (as the FU only does one statement at a time) mapping of V' to the integers $\{0, 1, 2, \cdots\}$. Write $t(s)$ for the execution time of s (1 or 2 in our model). Then to respect dependencies it is necessary and sufficient that:

$$s_1 \prec s_2 \;\Rightarrow\; \pi(s_1) + t(s_1) \leq \pi(s_2). \tag{1.4}$$

for s_1 and s_2 in V'. The *completion time* of π is

$$T(\pi) = \max_{s \in V'} \{\pi(s) + t(s)\}. \tag{1.5}$$

Our aim is to find π to minimise the quantity $T(\pi)$ subject to (1.4). This optimisation problem is NP-hard [18, 25].

1.4.3 The DDG of the derivative code

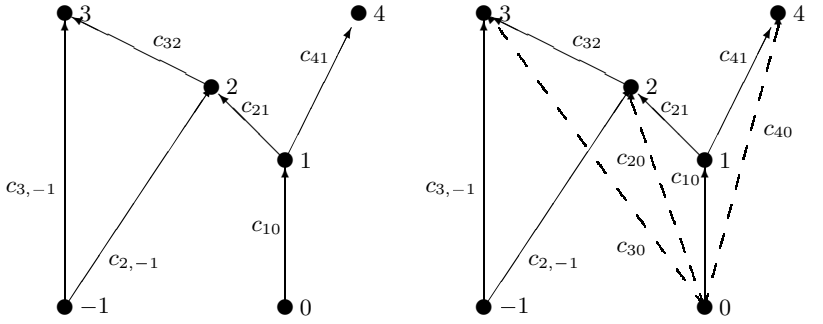
The classical way of constructing the derivative code's DDG would be to parse the code, build up an abstract representation and deduce the dependences between all statements, see for instance [26, 25]. Since derivative code is generated from function-code, its DDG can be constructed more easily by using data that is available during code generation. We omit details here.

Consider the code fragment of Fig. 1.1. Its computational graph is represented by the dag G on the left of Fig. 1.2, with, on the right, the extra edges produced on eliminating the intermediates v_1 and v_2 in that order.

Fig. 1.1. A code fragment

$$\begin{aligned} v_{-1} &= x_1 \\ v_0 &= x_2 \\ v_1 &= \sin(v_0) \\ v_2 &= v_1 - v_{-1} \\ v_3 &= v_{-1} v_2 \\ v_4 &= \sqrt{v_1} \end{aligned}$$

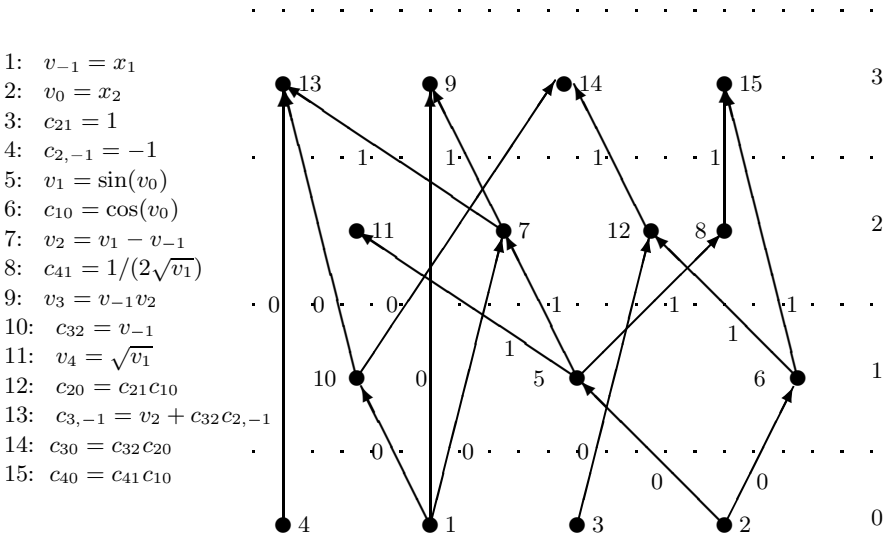
Fig. 1.2. Graph augmentation process: eliminated vertices are kept and fill-in is represented by dashed arrows



(a) Original computational graph (b) Augmented computational graph

The left of Figure 1.3 shows derivative code from Figure 1.1, with a somewhat arbitrary order of the statements respecting dependencies. Note one

Fig. 1.3. The data dependence graph G' of the derivative code from the original dag G on the left of Fig. 1.2



statement, #13, that combines computation of an elementary derivative with elimination. On the right is its DDG, which can be constructed directly from the elimination order and the original graph on the left of Fig. 1.2. (It could be made smaller by propagating the constant values $c_{2,-1}$ and c_{21}.) Edges out of statement s are labelled by the value $(t(s) - 1)$, i.e. 0 or 1 representing the time delay imposed by s, in our processor model.

The depth-first traversal approach of [4, 14] gave the following schedule:

π_1 :

2	5		8	6		15	3	12	1	10	14	7	4	13	11	9	

π_1 contains 2 idle cycles and takes 18 cycles to complete including a cycle at the end: $T(\pi_1) = 18$. In Sect. 1.5, we produce a schedule that takes 16 cycles, the minimum possible. This new schedule combines the depth-first traversal property and the instruction level parallelism of pipelined processors via a ranking function.

1.5 A Greedy List Scheduling Algorithm

We use a greedy list scheduling algorithm as investigated in [21, 23]. We first preprocess the DDG G' to compute a ranking function that defines the relative priority of each vertex. Then a modification of the labelling algorithm of [23, 27] is used to iteratively schedule the vertices of G'.

Our ranking function uses the assumption that operations with more successors and which are located in a longer path should be given priority, being

likely to execute with a minimum delay and to affect more operations in the rest of the schedule. We use the functions $\text{height}(v), \text{depth}(v)$ defined to be the length of the longest path from v to an input (minimal) vertex and to an output (maximal) vertex respectively. $\text{height}(v)$ is defined by

1. for each input (minimal) vertex v, $\text{height}(v) = 0$;
2. for each other $v \in V'$, $\text{height}(v) = 1 + \max\{\text{height}(w), \text{ for all } w \succ v\}$.

$\text{depth}(v)$ is defined in the obvious dual way. For a vertex $v \in V'$ we define the ranking function by

$$\text{rank}(v) = a * \text{depth}(v) + b * \text{succ}(v). \tag{1.6}$$

where $\text{succ}(v)$ is the number of successors of v and a, b weights chosen on the basis of experiment. For $b = 0$, we recover the SRA using depth-first traversal as in [4, 14]. By combining the values $\text{depth}(v)$ and $\text{succ}(v)$, we aim to trade off between exploiting instruction level parallelism of modern processors and minimising register pressure.

The preprocessing phase of our algorithm is as follows.

1. Compute the heights and depths of the vertices of G'.
2. Compute the ranks of the vertices as in (1.6).

The iterative phase of the algorithm schedules the vertices of G' in decreasing order of rank. It constructs the mapping π defined in Sect. 1.4.2 by combining the rank of a vertex and its *readiness* using the following rule:

Rule 1
A vertex v is ready *to be scheduled if it has no predecessor or if all its predecessors have already completed.*

This ensures data on which the vertex v depends is available when v is scheduled.

Ties between vertices are broken using the following rule:

Rule 2
Among vertices of the same rank choose those with the minimum height. Among those of the same height, pick the first.

The core of the scheduling procedure is as follows:

1. Schedule first an input vertex v with the highest rank (break ties using Rule 2). That is, set time $\tau = 0$ and $\pi(v) = \tau$.
2. For $\tau > 0$, let v be the last vertex that was scheduled at times $< \tau$.
   a) Extract from the set of so far unscheduled vertices, the set A as follows:

$$\begin{aligned}
&S(v) = \{w : w \succ v\}\\
&\text{If } \ S(v) \ \text{ is nonempty, set}\\
&B(v) = \{u : \text{height}(u) < \max\{\text{height}(w) \text{ for } w \in S(v)\}\},\\
&\quad A = S(v) \cup B(v);\\
&\text{Otherwise}\\
&\quad A = \text{the set of remaining vertices.}
\end{aligned}$$

b) Extract from A, the set of vertices R that are ready to be scheduled.
c) If R is empty, do nothing (a no-op at this cycle). Otherwise, choose from R a vertex v with maximum rank (break ties by Rule 2), and set $\pi(v) = \tau$.
d) Set $\tau = \tau + 1$.

3. Repeat step 2 until all vertices are scheduled.

We can easily check that this algorithm determines a schedule π that satisfies (1.4), thus preserving the data dependences between vertices of the graph.

Let us apply this algorithm to the DDG of Fig. 1.3 to get a schedule π_2. We first compute the height, depth and rank of each vertex using the coefficients $a = b = 1$:

vertex	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
height	0	0	0	0	1	1	2	2	3	1	2	2	3	3	3
depth	2	3	2	1	2	2	1	1	0	1	0	1	0	0	0
succ	3	2	1	1	3	2	2	1	0	2	0	1	0	0	0
rank	5	5	3	2	5	4	3	2	0	3	0	2	0	0	0

To label the dag of Fig. 1.3, the algorithm starts with the input vertices and assigns $\pi_2(1) = 1$. Next it forms the set $A = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ of available statements, and the set $R = \{2, 3, 4, 10\}$ of ready statements. Using the ranking list, it assigns $\pi_2(2) = 2$; etc. The result of this algorithm for the dag of Fig. 1.3 is the following optimal schedule without idle cycles:

π_2 : | 1 | 2 | 5 | 6 | 10 | 3 | 4 | 7 | 12 | 8 | 11 | 13 | 9 | 14 | 15 | |

We observe that the completion time $T(\pi_2) = 16$, better than $T(\pi_1)$. The complexity of this labelling algorithm, which is similar to that of [23, 27] for a dag with n vertices and e edges, was initially proved to be $\mathcal{O}(n^2)$ [23, 27] and can be implemented in $\mathcal{O}(n + e)$ as shown in [28].

1.6 Conclusions and Further Work

We have presented a detailed performance analysis of Jacobian calculations using the vertex elimination algorithm. We have shown that for even medium-sized numerical applications the execution time is very much correlated with the memory accesses than with the number of floating point operations. We pointed out that though the vertex elimination algorithm reduced the number of floating point operations, it should be coupled with instruction scheduling heuristics to enable exploitation of the superscalar nature of modern processors so as to maximise the performance of the derivative code.

For that purpose, we described a statement reordering scheme based on a ranking function. We plan to implement it and test it using medium-sized problems on a range of superscalar processors. We may also look at ways of combining the two objectives of reducing flops and memory accesses in a single objective function.

Acknowledgements

The authors would like to thank Prof. J.K. Reid for enlightening discussions and one of the referees for thorough reading of our paper, many useful comments and suggestions.

References

1. Rall, L.B.: Automatic Differentiation: Techniques and Applications. Volume 120 of Lecture Notes in Computer Science. Springer-Verlag, Berlin (1981)
2. Griewank, A.: Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Number 19 in Frontiers in Appl. Math. SIAM, Philadelphia, PA (2000)
3. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13** (1991) 451–490
4. Forth, S.A., Tadjouddine, M., Pryce, J.D., Reid, J.K.: Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. ACM Transactions on Mathematical Software **30** (2004) 266–299
5. Naumann, U.: Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs. PhD thesis, Technical University of Dresden (1999)
6. Griewank, A., Reese, S.: On the calculation of Jacobian matrices by the Markowitz rule. In Griewank, A., Corliss, G.F., eds.: Automatic Differentiation of Algorithms: Theory, Implementation, and Application. SIAM, Philadelphia, Penn. (1991) 126–135
7. Yannakakis, M.: Computing the minimum fill-in is NP-complete. SIAM J. Alg. Disc. Meth. **2** (1981) 77–79
8. Goedecker, S., Hoisie, A.: Performance Optimization of Numerically Intensive Codes. SIAM Philadelphia (2001)
9. GroppItP, W., Kaushik, D., Keyes, D., Smith, B.: Improving the performance of sparse matrix-vector multiplication by blocking. Technical report, MCS Division, Argonne National Laboratory (2000) See `www-fp.mcs.anl.gov/petsc-fun3d/Talks/multivec_siam00_1.pdf`.
10. Bornstein, C., Maggs, B., Miller, G.: Tradeoffs between parallelism and fill in nested dissection. In: Proceedings of the SPAA'99, ACM (1999)
11. George, J., Liu, J.: An automatic nested dissection algorithm for irregular finite element problems. SIAM Journal of Numerical Analysis **15** (1978) 345–363
12. Markowitz, H.: The elimination form of the inverse and its application. Management Science **3** (1957) 257–269
13. Amestoy, P., Davis, T., Duff, I.: An approximate minimum degree ordering algorithm. SIAM J. Matrix Anal. Applic. **17** (1996) 886–905
14. Tadjouddine, M., Forth, S.A., Pryce, J.D., Reid, J.K.: Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In Sloot, P.M., ed.: Proceedings of the Second International Conference on Computational Science. Volume 2 of LNCS., Amsterdam, Springer-Verlag (2002) 1077–1086

15. Averick, B.M., Carter, R.G., Moré, J.J., Xue, G.L.: The MINPACK-2 test problem collection. Preprint MCS–P153–0692, ANL/MCS–TM–150, Rev. 1, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill. (1992) See `ftp://info.mcs.anl.gov/pub/MINPACK-2/tprobs/P153.ps.Z`.
16. Roe, P.L.: Approximate Riemann solvers, parameter vectors, and difference schemes. Journal of Computational Physics **43** (1981) 357–372
17. The Mathworks Inc: Statistics Toolbox User's Guide, Version 6.5. (2002) See `http://www.mathworks.com/access/helpdesk/help/toolbox/stats/`.
18. Motwani, R., Palem, K., Sarkar, V., Reyen, S.: Combining register allocation and instruction scheduling: (technical summary). Technical Report TR 698, Courant Institute, New York University, USA (1995) See `http://csdocs.cs.nyu.edu/Dienst/Repository/2.0/Body/ncstrl.nyu_cs/TR19%`95-698/postscript.
19. Hardnett, C., Rabbah, R., Palem, K., Wong, W.: Cache sensitive instruction scheduling. Technical Report CREST-TR-01-003, GIT-CC-01-15, Center for Research in Embedded Systems and Technologies (2001) See `citeseer.ist.psu.edu/hardnett01cache.html`.
20. Hennessy, J., Gross, T.: Postpass code optimization of pipeline constraints. ACM TOPLAS **5** (1983) 422–448
21. Palem, K., Simons, B.: Scheduling time-critical instructions on RISC machines. ACM TOPLAS **15** (1993) 632–658
22. Leung, A., Palem, K.V., Ungureanu, C.: Run-time versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and tradeoffs. In: Proceedings of the third International Conference of High Performance Computing, ACM (1996)
23. Bernstein, D., Gertner, I.: Scheduling expressions on a pipelined processor with a maximun delay of one cycle. ACM TOPLAS **11** (1989) 57–66
24. Pryce, J.D., Tadjouddine, M.: Cheap Jacobians by AD regarded as compact LU factorization. SIAM Journal on Scientific Computing (2004) To be submitted.
25. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, San Francisco, USA (1997)
26. Chapman, B., Zima, H.: Supercompilers for Parallel and Vector Computers. Addison-Wesley Publishing Company, Boston, USA (1991)
27. Cofman, E., Graham, R.: Optimal scheduling for two-processor systems. Acta Informatica **1** (1972) 200–213
28. Gabow, H.N., Tarjan, R.E.: A linear algorithm for a special case of disjoint set union. In: Proceedings of the 15th ACM Symposium on Theory of Computing, New York, ACM (1983) 57–66