

Improving the Performance of the Vertex Elimination Algorithm for Derivative Calculation

M. Tadjouddine^a, F. Bodman^b, J.D. Pryce^b and S.A.Forth^a

^a Engineering Systems Department, Cranfield University (Shrivenham Campus), Swindon SN6 8LA, UK.

^b Department of Information Systems, Cranfield University (Shrivenham Campus), Swindon SN6 8LA, UK.

Abstract

We study two aspects of the vertex elimination algorithm in calculating Jacobians. First, we used Markowitz-like heuristics aiming at minimising the number of floating-point operations to find elimination sequences and then generate the Jacobian code. Second, we used the depth-first traversal algorithm to reorder the statements of the Jacobian code so as to reduce the number of memory accesses. On RISC processors, we observed that for in-cache data, the number of floating point operations gives a good estimate of the execution time, while for out-of-cache data, the execution time is dominated by the memory accesses. We also present a statement reordering scheme based on ranking functions, which will enable the exploitation of the instruction level parallelism of such processors and maximise performance.

1 Introduction

Many scientific applications require the first derivative (at least) of a function $\mathbf{f} : \mathbf{x} \in \mathbb{R}^n \mapsto \mathbf{y} \in \mathbb{R}^m$ represented by a computer program. This can be obtained using Automatic Differentiation(AD)[8]. Typically, from the program, we can first, build up the computational graph of the function \mathbf{f} as a directed acyclic graph $G=(V, E)$, where V is the set of vertices v_i and E , the set of edges (v_j, v_i) . A vertex v_i represents an instruction of the original code; an edge $(v_j, v_i) \in E$, the data dependence relationship \prec from v_j to v_i meaning v_i depends on v_j . We have $|V|=n+p+m=N$ where n, p, m are respectively the number of input, intermediate and output vertices. Second, we linearise G by attaching its edges with local partial derivatives. Finally, we eliminate, in some order, all intermediate vertices so as G is rendered bipartite. We refer this process as the *vertex elimination approach* and can be found in [4, 8, 13].

As detailed in [4, 8], the graph G can be viewed as a $N \times N$ sparse lower triangular matrix $\mathbf{C}=(c_{ij})$ called *extended Jacobian*. The Jacobian \mathbf{J} can be obtained by solving a rather large linear system using some form of Gaussian elimination.

Since the number p of intermediate vertices tends to be enormous 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]?”. The fill-in problem was shown to be NP-complete in [17] and we suspect the same holds for the flop-count 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 [7] 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, memory traffic appears to be the hurdle to overcome. In this paper, we study two aspects of the vertex elimination algorithm. First, we study how the number of floating point operations (flops) 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 that purpose, 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 the number of flops, and of memory traffic (loads and stores). We observed:

- A reordering of statements of the code can improve the performance of the Jacobian code by a significant percentage when this reduces the memory traffic.
- For in-cache 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 cache data, the execution time is dominated by the number of load and store operations and a reordering that reduced these memory access operations enhanced Jacobian code's performance.

Similar behaviour is found in other analyses of numerical codes, see for example [7]. This paper presents the argument in a context of semantic augmentation of numerical codes as it is done in AD of computer programs. We also describe planned work to improve performance of Jacobian code, produced by vertex elimination, by reordering the statements following standard instruction scheduling algorithms.

2 Heuristics

Over the past four decades, several heuristics aimed at producing elimination orderings with low fill have been investigated. These algorithms have the desired effects of reducing work as well. The most widely used are *nested dissection* [3, 5] and *minimum degree*. The latter originated with the *Markowitz method* [11] and is studied for example in [1]. Nested dissection, a recursive algorithm, starts by finding a *balanced separator*. That is a set of vertices that, when removed, partition the graph into two or more components composed of vertices which, when eliminated, do not create fill-in in any of the other components. The vertices are ordered in each component and the vertices in the separator at the end. The process is then repeated on the components.

On the other hand, Markowitz-like algorithms, unlike nested dissection that examines the entire graph before reordering it, perform local optimisations. At each elimination step, they select a vertex with minimum cost in some sense, eliminate it and look for the next vertex with minimum cost in the new graph.

We applied various elimination sequences, with the names Forward, Reverse, Markowitz, VLR, Markowitz [resp. VLR] with Pre-elimination as described in [4, 13] to graphs obtained using statement level (SL) and code list (CL) differentiation [4, 16]. We studied performance with and without applying statement reordering as described in Section 4, see [16].

3 Performance Analysis

We consider two of the test problems reported in [4]: the Human Heart Dipole (HHD) from the Minpack 2 test suite and the Roe flux calculation (ROE) [15]. These routines were differentiated using ADIFOR [2] and TAMC [6] and via the AD tool ELIAD [4, 16] using the heuristics listed in Section 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, we studied the assembler from different platforms, counting the number of loads, stores and floating point operations. We then modelled the execution time in terms of loads and stores and floating point operations. Figure 1 and Figure 2 show the results of our study.

In Figure 1 and Figure 2, ULTRA10 represents the SUN Ultra 10 processor with 440 MHz, 32 KB L1 cache, 2 MB L2 cache, and using the Workshop f90 6.0 Compiler ; SGI the R12000 processor, 300 MHz, 64KB L2 cache, 8 MB L2 cache, and using the f90 MIPS Pro 7.3 compiler. On the abscissa axis, the range 1 – 4 are forward and reverse orderings using SL and CL differentiation; the range 5 – 8 the same heuristics preceded by pre-elimination of vertices with single successor; the range 9 – 12 are the heuristics 5 – 8 followed by statement reordering; the range 13 – 16 represents Markowitz and VLR using SL and CL differentiation; the range 17 – 20 are 13 – 16 followed by statement reordering, the range 20 – 24 are respectively Hand-coded, ADIFOR(forward), TAMC(forward) and TAMC(reverse).

The observed time *Obs-Time* is the CPU time obtained by averaging a certain number of evaluations and runs, see [4] for details. If a is the number of floating point operations, b the number of loads and stores, k_1, k_2 respectively the number of floating point operations and memory accesses performed per cycle, we

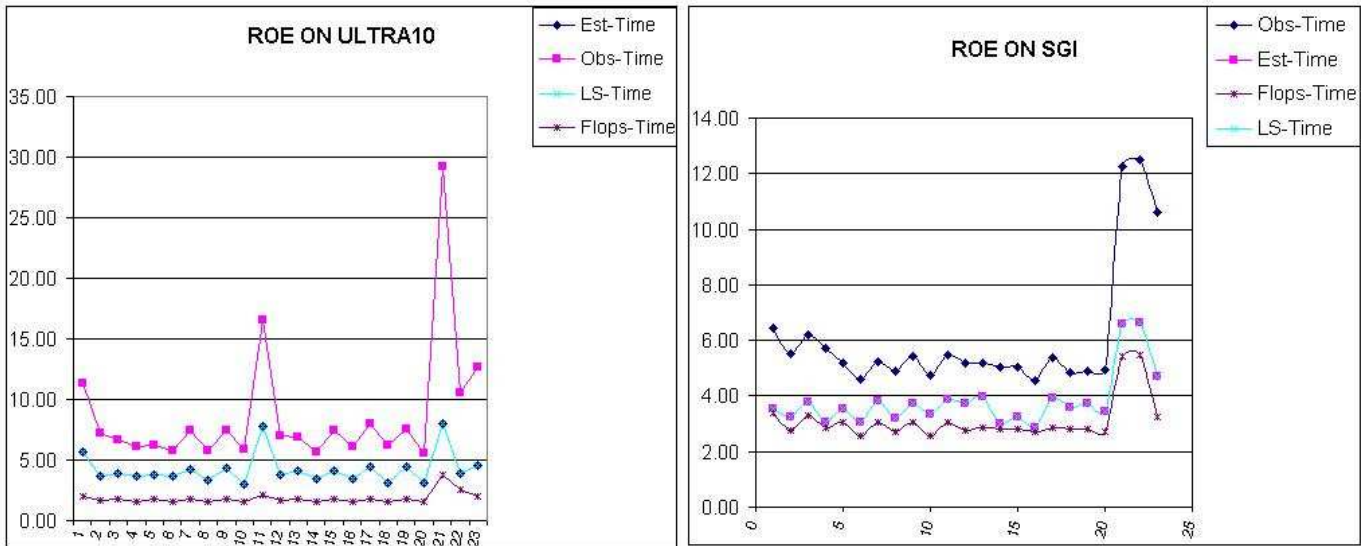


Figure 1: Performance modelling of the execution time: Roe case

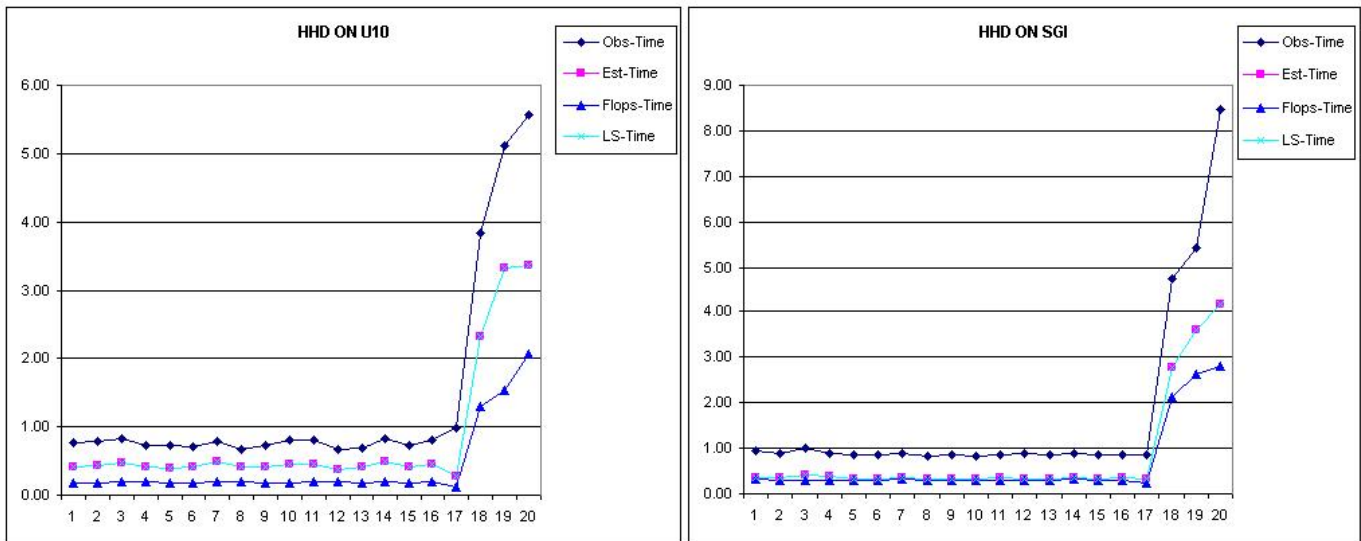


Figure 2: Performance modelling of the execution time: HHD case

calculated the following quantities: the estimated time $Est-Time = \max(a/k_1, b/k_2) * cycle\ time$, the $Flops-Time = a/k_1 * cycle\ time$ and the $LS-Time = b/k_2 * cycle\ time$. The Ultra 10 processor can perform up to 2 floating point operations per cycle with a latency of 4 cycles and 1 load or 1 store having 2 cycles latency, and uses in-order execution [10] of instructions.

The R12000 can perform up to 2 floating point operations per cycle, 1 memory access (load/store) both having a latency of 2 cycles, and uses out-of-order execution [7, 10] of instructions.

Figure 1 shows that the execution time is dominated by the memory access time, which coincides with the estimated time. The observed time is closely approximated by the estimated time multiplied by the latency of memory access operations. Moreover, the statement reordering performed better when it reduced the number of memory accesses. Figure 2 represents results from a small test case, which fits into the L1-cache. Though, we observed that memory accesses have played a central role, but importantly, a reduction of floating point operations affected the overall execution time and that the observed time is closely approximated by the $Flop-time$ multiplied by the latency of the floating point operation.

4 A Statement Reordering Scheme

The generated Jacobian code to calculate \mathbf{J} , includes \mathbf{f} 's original code as well as statements to compute the local derivatives c_{ij} that are the nonzeros of \mathbf{C} before elimination. But the bulk of it is *elimination statements*, of one of the forms $c_{ij} = c_{ik}c_{kj}$ or $c_{ij} = c_{ij} + c_{ik}c_{kj}$. The statements can be reordered in any way that respects dependencies, i.e. if statement s_1 depends on statement s_2 , then s_2 must precede s_1 . This defines the Jacobian code's data dependency graph $G' = (V', E')$, where V' is the set of statements.

In [4, 16], we used a statement reordering algorithm (SRA) — using a depth-first traversal of G' — that for each $s_1 \in V'$, tries to place the s_2 's on which s_1 depends, close to s_1 . 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 be a problem [16]. The benefits were patchy, probably because this does not take account of the instruction-level parallelism (ILP) of modern cache-based machines and the latencies of certain instructions.

In this work, we plan to exploit ILP by a SRA that gives priority to certain statements via a ranking function. The compiler's optimisations work on a dependency graph G'' whose vertices are machine-code instructions. We have no knowledge of G'' so we shall work on G' . We shall use the combined register and instruction scheduling approach used in for instance [12], and the ranking function ideas of [10, 14]. Our ranking function uses the assumptions that operations, which have 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. The rank of a vertex $s \in V'$ is a weighted sum of $height(s)$, the length of the longest path out of s , and $succ(s)$, the number of successors of s . We plan to implement this reordering scheme and assess its impact on the overall performance of the vertex elimination algorithm in calculating Jacobians.

5 Conclusions and Further Work

We have presented a detailed performance analysis of Jacobian calculations using the vertex elimination algorithm. We have shown that for in-cache data the execution time is correlated with the number of floating point operations whereas for out-of-cache data it is correlated with the memory accesses. 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 ILP exploitation of modern processors, reducing memory accesses and maximising performance.

We described a statement reordering scheme based on ranking functions. We plan to implement it and test it using medium-sized problems on a range of RISC processors.

Acknowledgements

This work was supported by EPSRC and UK MOD under grant GR/R21882. The authors would like to thank Prof. J.K. Reid for enlightening discussions and comments about this work.

References

- [1] P. Amestoy, T. Davis, and I. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.
- [2] C. H. Bischof, A. Carle, P. Khademi, and A. Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.
- [3] C. Bornstein, B. Maggs, and G. Miller. Tradeoffs between parallelism and fill in nested dissection. In *SPAA'99*. ACM, 1999.
- [4] S. Forth, M. Tadjouddine, J. Pryce, and J. Reid. Jacobian code generated by source transformation and vertex elimination can be as efficient as hand-coding. *ACM Trans. on Math. Soft.*, To appear 2004.
- [5] J. George and J. Liu. An automatic nested dissection algorithm for irregular finite element problems. *SIAM Journal of Numerical Analysis*, 15:345–363, 1978.
- [6] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Trans. on Math. Soft.*, 24(4):437–474, December 1998.
- [7] S. Goedecker and A. Hoisie. *Performance Optimization of Numerically Intensive Codes*. SIAM Philadelphia, 2001.
- [8] A. Griewank. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Number 19 in *Frontiers in Appl. Math.* SIAM, Philadelphia, Penn., 2000.
- [9] W. Gropp, D. Kaushik, D. Keyes, and B. Smith. Improving the performance of sparse matrix-vector multiplication by blocking. Technical report, MCS Division, Argonne National Laboratory. www-fp.mcs.anl.gov/petsc-fun3d/Talks/multivec_siam00_1.pdf.
- [10] C. Hardnett, R. Rabbah, K. Palem, and W. Wong. Cache sensitive instruction scheduling. Technical Report CREST-TR-01-003, GIT-CC-01-15, Center for Research in Embedded Systems and Technologies, June 2001.
- [11] H. Markowitz. The elimination form of the inverse and its application. *Management Science*, 3:257–269, 1957.
- [12] R. Motwani, K. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling: (technical summary). Technical Report TR 698, Courant Institute, July 1995.
- [13] U. Naumann. *Efficient Calculation of Jacobian Matrices by Optimized Application of the Chain Rule to Computational Graphs*. PhD thesis, Technical University of Dresden, June 1999.
- [14] K. Palem and B. Simons. Scheduling time-critical instructions on RISC machines. *ACM TOPLAS*, 5(3), 1993.
- [15] P. L. Roe. Approximate Riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics*, 43:357–372, 1981.
- [16] M. Tadjouddine, S. A. Forth, J. D. Pryce, and J. K. Reid. Performance issues for vertex elimination methods in computing Jacobians using automatic differentiation. In P. M. Sloot, editor, *Proceedings of the Second International Conference on Computational Science*, volume 2 of *LNCS*, pages 1077–1086, Amsterdam, 2002. Springer-Verlag.
- [17] M. Yannakakis. Computing the minimum fill-in is NP-complete. *SIAM J. Alg. Disc. Meth.*, 2(1):77–79, 1981.