Lazy K-Way Linear Combination Kernels For Efficient Runtime Sparse Jacobian Matrix Evaluations In C++

Rami M. Younis and Hamdi A. Tchelepi

Abstract The most notoriously expensive component to develop, extend, and maintain within implicit PDAE-based predictive simulation software is the Jacobian evaluation component. While the Jacobian is invariably sparse, its structure and dimensionality are functions of the point of evaluation. The application of Automatic Differentiation to develop these tools is highly desirable. The challenge presented is in providing implementations that treat dynamic sparsity efficiently without requiring the developer to have any *a priori* knowledge of sparsity structure. Under the context of dynamic sparse Operator Overloading implementations, we develop a direct sparse lazy evaluation approach. In this approach, an efficient runtime variant of the classic Expression Templates technique is proposed to support sparsity. The second aspect is the development of two alternate multi-way Sparse Vector Linear Combination kernels that yield efficient runtime sparsity detection and evaluation.

Key words: Implicit, Simulation, Sparsity, Jacobian, Thread-Safety

1 Introduction

A focal area of scientific computing is the predictive simulation of complex physical processes. Implicit simulation methods require the evaluation and solution of large systems of nonlinear residual equations and their Jacobian matrices. In the context of emerging simulation applications, the Jacobian matrix is invariably large and sparse. Moreover the actual sparsity structure and dimensionality may both be functions of the point of evaluation. Additionally, owing to the model complexity, the evaluation of the Jacobian matrix typically occurs over numerous modules and stages, requiring the storage of resultants from a wide range of intermediate calculations. The resultants of such calculations vary dramatically in terms of their level

Department of Energy Resources Engineering, Stanford University, Stanford, CA, USA
`[ryounis,tchelepi]@stanford.edu`

of sparsity, ranging from univariate variables to dense and block sparse multivariates. Finally, given an interest in rapidly modifiable codes to include new physics or alternate sets of independent unknowns, the most notoriously expensive software component to develop, extend, and maintain is the Jacobian matrix evaluation component. Dynamic, sparse Automatic Differentiation (AD) offers a clearly recognized potential solution to the design and development challenges faced by implicit simulator developers. Several comprehensive introductions to AD are available [9, 7, 15]. The efficient runtime computation of dynamic sparse Jacobian matrices is the topic of several recent contributions. There are two broad approaches to dynamic sparse AD.

The first approach uses results from sparsity pattern analysis by graph coloring techniques in order to obtain the Jacobian from a compressed intermediate dense matrix [13, 8]. This is accomplished by inferring the sparsity pattern of the Jacobian and analyzing it to determine an appropriate compression operator that is referred to as the *seed matrix*. The dense intermediate matrix is computed using AD, and the target sparse Jacobian is backed-out from it using the seed matrix. Since the AD operations are performed in a dense format, they can be implemented efficiently. Advances in efficient dense AD implementations include the Operator Overloading (OO) tools as described in [14, 1]. These approaches report the use of a lazy evaluation generic metaprogramming technique known as Expression Templates (ET) [11, 10] in order to attain close to optimal dense AD operation efficiencies. The computational costs of the compression and de-compression however can be significant and can involve heavy sparse memory bandwidth limited operations. In situations where the sparsity pattern is constant or is known *a priori*, this cost may be amortized since the seed matrix remains unchanged. In the context of general purpose predictive simulation this is not the case.

The second approach is intrinsically dynamic, and it uses sparse vector datastructures to represent derivatives. The core computational kernel of direct runtime sparse AD is a SParse-vector Linear Combination (SPLC). This is because the derivative of any expression with $k > 0$ arguments can be expressed as a linear combination of the k sparse vector derivatives of the expression arguments; $c_1\mathbf{f}_\mathbf{1}^{'} + \ldots + c_k\mathbf{f}_\mathbf{k}^{'}$. SPLC operations perform sparsity structure inference along with the computation of the sparse Jacobian entries. Examples of implementations with such a capability include the SparsLinC module [4] within the Source Transformation tools ADIFOR [3] and ADIC [5]. Direct sparse treatment offers complete runtime flexibility with transparent semantics. On the other hand, since the computational kernel consists of a set of sparse vector operations, it is a challenge to attain reasonable computational efficiency on modern computer architectures. Sparse algebra operations involve a heavy memory bandwidth requirement leading to notoriously memory-bound contexts[12]. Existing codes such as the SparsLinC module provide various sparse vector datastructures that attempt to hide the costs of dynamic memory and memory latency to some extent.

1.1 *This work*

This work extends the lazy evaluation performance optimization techniques that are applied in dense AD approaches to direct dynamic sparsity OO implementations. The extension requires two advances. The first, is to extend the datastructures and construction mechanism of the ET technique to suit sparsity. The second is to develop single pass algorithms to execute SPLC expressions more efficiently.

Section 2 introduces the challenges of extending the classic compile-time ET technique to support sparse arguments directly. A run-time alternative form of ET is developed. In particular, the run-time variant is designed to deliver competitive levels of efficiency compared to static approaches while directly supporting sparsity.

Section 3 reviews current SPLC algorithms and develops an alternate class of single-pass SPLC evaluation algorithm. The algorithms execute SPLC expressions involving *k* sparse vectors in one go while improving the ratio of memory and branch instructions to floating point operations over current alternatives.

Finally, Sect. 4 presents computational results using a large-scale Enhanced Oil Recovery simulation.

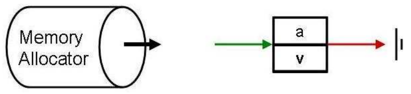
2 Lazy evaluation techniques for dynamic sparsity

The compile-time (static) ET technique is a lazy evaluation OO implementation that overcomes the well-recognized performance shortcoming of plain OO [6]. Along the *pairwise evaluation* process of OO, the ET technique generates expression graph nodes instead of dense vector intermediate resultants. The expression nodes are allocated on the stack, and they are meant to be completely optimized away by the compiler. The execution of the expression is delayed until the expression is assigned to a cached dense vector variable. At that point, the expression is executed with a single fused loop.

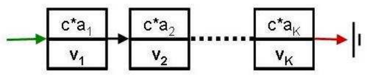
Since dense linear combinations involve vectors of the same dimension, the single-pass loop is performed entry-by-entry. Each iteration produces the appropriate scalar resultant by recursively querying parent nodes in the ET graph. The recursion terminates at the leaf dense vectors which simply return their corresponding entry. On the way out of the recursion, intermediate nodes perform scalar operations on the returned values and pass the resultant down along the recursion stack.

The extension of the classic ET technique to SPLCs requires a different ET datastructure. In sparse operations, non-zero entries do not always coincide, and subsequently the depth of the fused loop is not known until the entire SPLC is executed. Moreover, every node within the ET datastructure would need to maintain its own intermediate scalar state. This implies that the ET nodes for a sparse expression grow recursively in size on the stack with no constraints on the recursion depth. This is exacerbated by the fact that OO intermediates have a temporary life-cycle, and so ET nodes need to store parent nodes by value to avoid undefined behavior. The exception to this is the leaf nodes since they refer to vector arguments that are persistent in memory. This costly situation suggests a value in dynamic SPLC expression datastructures that are inexpensive to build at runtime.

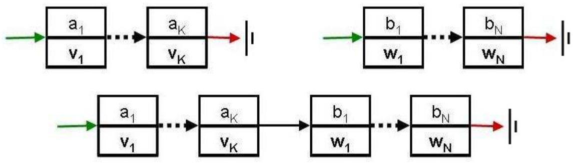
Once they are multiplied through, forward mode derivative expressions become vector linear combinations. The SPLCs can be represented by a list where each entry is a pair of a scalar weight and a sparse vector argument. Owing to their efficiency of concatenation, singly linked list datastructures can be used to efficiently store and represent runtime SPLC expressions. In the proposed approach, the scalar operators are overloaded to generate an SPLC list through three fundamental building blocks. These three operations are illustrated Fig. 1.



(a) $c.\mathbf{V}$



(b) $c.\sum_{i=1}^{k} a_i.\mathbf{V}_i$



(c) $\sum_{i=1}^{K} a_i.\mathbf{V}_i + \sum_{j=1}^{N} b_j.\mathbf{W}_j$

Fig. 1 SPLC expressions are represented by a one-directional linked list. The list is built by the OO pairwise evaluation process involving three fundamental operations only.

The first operation depicted in Fig. 1(a) involves the multiplication of a scalar weight and a sparse vector argument. This operation would be used for example whenever the chain rule is applied. Only in this operation is it necessary to allocate memory dynamically. Since the elements of SPLC expressions are allocated dynamically, their lifespan can be controlled explicitly. Subsequently, nodes can be made to persist beyond a statement's scope and it is only after the evaluation stage that the SPLC expressions need to be freed.

The second operation is the multiplication of a scalar and a SPLC sub-expression. As illustrated in Fig. 1(b), this is accomplished most efficiently by multiplying the weights in the linked list through, leaving the dynamic memory intact as is it returned by the operator.

Finally, Fig. 1(c) illustrates the third building block; the addition of two SPLC sub-expression, each containing one or more terms. The addition simply involves the re-assignment of the tail pointer of one sub-list to the head node of the other. In total, using dynamic memory pools, the run-time lists require $O\{k\}$ operations.

3 K-Way SPLC Evaluation kernels

Upon assignment to a resultant, the SPLC needs to be evaluated. In this section, we develop two evaluation algorithms that exploit the fact that all k arguments are already available.

The first algorithm employs a caching implicit binary tree to generalize the SparsLinC algorithms. The cached intermediate nodes store non-zero elemental information, thereby substantially reducing the number of non-zero index comparisons and the associated memory bandwidth requirements. The second algorithm is inspired by the seed matrix mappings used in other forms of sparse AD. Before presenting the two algorithms, we review the current approach to SPLC evaluation in AD tools.

The SparsLinC module generalizes a 2-way algorithm in a static manner to accommodate more arguments. The 2-way SPLC uses a running pointer to each of the two vector arguments. Initially both of the two pointers are bound to the first non-zero entry of its respective sparse vector argument. While both running pointers have not completely traversed their respective vector, the following sequence of operations is performed. The column indices of the two running pointers are compared. If the nonzero entry column indices are equal, the two entries are linearly combined and inserted into the resultant. Both running pointers are advanced. On the other hand, if they are not equal, then the entry with the smaller column index is inserted, and only its running pointer is advanced. At the end of the iteration, if one of the two sparse arrays involves any remaining untraversed entries, they are simply inserted into the resultant. The SparsLinC module executes K-Way combinations by repeating this 2-way algorithm in a pairwise process.

3.1 K-way SPLC kernel 1: Caching nodal binary tree

This approach generalizes the pairwise evaluation process used by SparsLinC in order to perform the evaluation in one pass while minimizing the number of index comparisons that are necessary. A binary tree is designed to maintain non-zero elemental state information at each node. This state information consists of a single nonzero entry (a pair of an integer column index and a value), as well as a logical index that maintains a node's *activation*. There are two types of nodes that make-up the tree.

1. Terminal leaves are the parent terminal nodes, and each leaf refers to a SPLC argument. A running pointer to the argument's sparse vector is maintained. The pointer is initialized to the sparse vector's first nonzero entry. Terminal leaves are *active* provided that their running pointer has not traversed the entire sparse vector.
2. Internal nodes, including the terminal root node, have two parents. Such nodes maintain the linear combination nonzero resultant of the parent nodes. Internal

nodes also maintain a coded activation variable that distinguishes between each of the following four scenarios:

a. Internal nodes are inactive if both parents are.
b. The left parent entry has a smaller column index than the right parent.
c. The right parent's column index is smaller than the left's.
d. The column indices of both parents are equal.

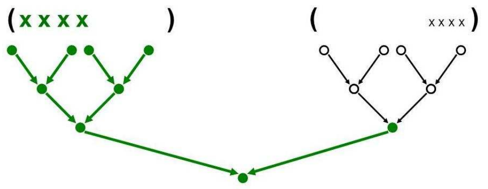


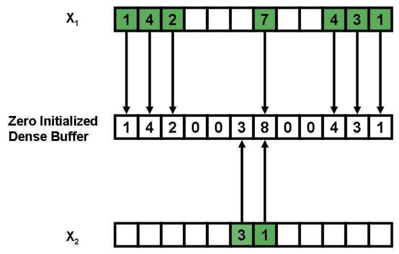
Fig. 2 A hypothetical SPLC expression and caching binary tree. The left sub-tree resultant is a sparse vector with nonzero entries with low column indices. The right sub-tree resultant has nonzero entries with large column indices. At the initial stages of the evaluation process (the first four iterations), only the left sub-tree is queried for column index comparisons.

The evaluation is performed in a single pass process starting from the root internal node. At each step in the fused evaluation loop, two reverse topological sweeps are executed by recursion. The first sweep is an *Analyze Phase* that labels the activation codes. The second sweep is an *Advance Phase* where all advance-able nodes are visited to evaluate their nonzero entry value and to update the running pointers of the active leaf nodes. The iteration continues so long as the root node remains active.

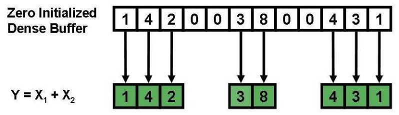
Consider the hypothetical SPLC scenario illustrated in Fig. 2. The proposed algorithm requires at most half of the number of comparisons and associated reads and writes as would be required by a SparsLinC kernel.

3.2 *K-way SPLC kernel 2: Prolong-and-restrict dense accumulation*

This algorithm is inspired by the seed matrix approaches to sparse AD. As illustrated in Fig. 3, the algorithm proceeds in a two stage process. In prolong phase (Fig. 3(a)), each of the k sparse array arguments is added into a zero-initialized dense buffer. In the restrict phase (Fig. 3(b)), the entire dense buffer is traversed to deduce the nonzero entries producing a sparse resultant. This algorithm performs poorly whenever the dimension of the required enclosing dense buffer is very large compared to the number of non-zero entries in the resultant. On the other hand, when that is not the case, this algorithm is very effective as it uses a more favorable memory locality profile and involves no branching in the prolong phase.



(a) Prolong phase; sparse vectors are added to a zero-initialized dense buffer.



(b) Restrict phase; the dense intermediate is mapped to a sparse resultant.

Fig. 3 An illustration of the two stages of the prolong-and-restrict k-way SPLC kernel. In this example, there are two SPLC arguments, $k = 2$, with unit weights.

3.3 Summary

Expressions involving multiple arguments ($k > 2$) can be evaluated more efficiently using k-way generalizations. In order to better characterize and compare the performance of the proposed algorithms, we introduce some diagnostic SPLC parameters which may all be computed efficiently during the SPLC list construction process. The first parameter is the *Apparent Dimension*, N_a, that is defined as the difference between the smallest nonzero entry column index and the largest column index in the resultant of the SPLC expression. The second parameter is the *Nonzero Density*, $0 < N_d \leq 1$, which is the ratio of the number of nonzero entries in the resultant of the SPLC to the Apparent Dimension. Finally, the third parameter is the number of arguments in the expression k.

The computational cost of the caching binary tree kernel is clearly independent of N_a. A worst-case scaling of the number of necessary memory reads and writes goes as $\log(k)N_d$. This cost is asymptotically favorable to that attained for example by the SparsLinC kernel which scales as $k.N_d$. On the other hand, the cost of the prolong-and-restrict dense accumulation kernel scales primarily with N_a since the prolong phase involves no branching.

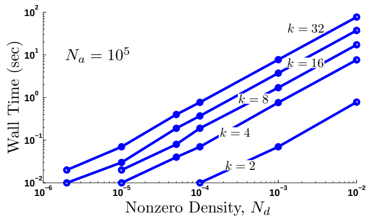
4 Computational Examples

The OO lazy evaluation techniques discussed in this work are all implemented in a comprehensive thread-safe generic C++ OO AD library [16] that computes runtime sparse Jacobians using the forward mode. The Automatically Differentiable Expression Templates Library (ADETL) provides generic datastructures to represent AD scalars and systems that can be univariate or multivariate (generically dense, sparse, or block sparse). The library handles cross-datatype operations and implements poly-algorithmic evaluation strategies. The choice of OO technique used depends on the type of derivatives involved in an AD expression. The ADETL treats univariates with a direct pairwise evaluation. Dense multivariate expressions involving more than two arguments are treated using the classic ET technique. Finally, sparse and block-sparse multivariate expressions are treated with the dynamic SPLC lists and are evaluated using either of the two proposed kernels.

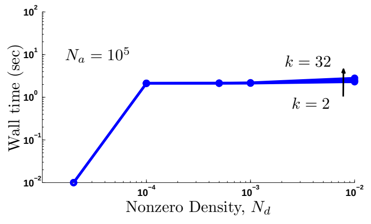
To illustrate the computational performance of the proposed algorithms for sparse problems, we consider a number of hypothetical SPLC expressions as well as the computation of a block structured Jacobian matrix arising from the numerical discretization of a system of PDAEs.

4.1 Model SPLC numerical experiments

In order to empirically validate the computational cost relations discussed in Sect. 3.3, we generate a number of synthetic SPLC expressions that span a portion of the three dimensional parameter space defines by k, N_a, and N_d. In particular, we execute a series of SPLC expressions $\sum_k c_k \mathbf{V_k}$ with $k = 2, 4, 8, 16,$ and 32 arguments. The argument sparse vectors $\mathbf{V_k}$ and coefficients c_k are generated randomly. By freezing the Apparent Dimension, $N_a = 10^5$, we can vary N_d simply by varying the number of nonzero entries used to generate the sparse vector arguments. We consider the range $10^{-6} < N_d < 10^{-1}$ that spans a wide range of levels of sparsity. Figure 4(a) and Figure 4(b) show the empirical results obtained using the binary tree and the prolong-restrict kernels respectively. The figures show plots of the wall execution time taken to construct and evaluate SPLC expressions with varying N_d. Each curve consists of results for a fixed k.



(a) Binary tree kernel



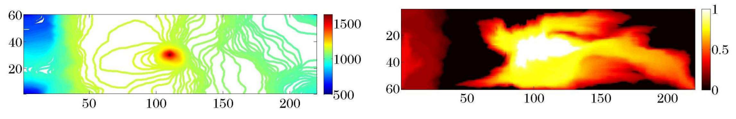
(b) Prolong-restrict kernel

Fig. 4 Performance comparisons of several SPLC evaluations using the two proposed kernels. The test sparse vectors are generated randomly and have an Apparent Dimension, $Na = 10^5$.

Clearly, the asymptotic behavior of the two algorithms is distinct. The prolong-restrict results show that for fairly large N_a, neither the number of arguments nor the level of sparsity N_d matter. These differences in computational cost lead to a performance crossover point. The ADETL exploits this by performing install-time measurements such as those presented in Fig. 4 in order to apply a poly-algorithmic evaluation strategy that automatically selects the better algorithm for a given situation.

4.2 Model Problem Simulation Jacobian

The nonlinear residual and Jacobian evaluation routines of a General Purpose Reservoir Simulator (GPRS) are re-written using the ADETL[16]. The original GPRS code was written using hand-coded Jacobian matrices including manual branch fragments that encode a dynamic sparsity pattern. The GPRS implements fully coupled implicit finite volume approximations of compressible compositional, thermal, multi-phase flow in porous media[2]. The system of equations is a collection of PDAEs of variable size and structure depending on the thermodynamic state.



(a) Pressure contour time snapshot (psi) (b) Gas saturation time snapshot

Fig. 5 Two sample state component snapshots for a simulation performed using the ADETL.

Figure 5 shows sample results obtained using the ADETL GPRS. During the course of the simulation, 735 Newton iterations are performed, each requiring the evaluation of the residual and Jacobian. The total wall clock times taken by the hand-differentiated and manually assembled GPRS is 238-secs. The time taken by the ADETL implementation is 287-secs, implying a total performance penalty of 21%. This penalty is considered minor compared to the improved maintainability and level of extendability of the new code.

5 Summary

The core kernel of runtime sparse Jacobian AD is a SPLC operation. We develop an OO implementation that combines a dynamic form of ET along with two alternate *k*-way evaluation algorithms. Extensive use of the ADETL in developing general purpose physical simulation software shows comparable performance compared to hand-crafted alternatives.

References

1. Aubert, P., Di Césaré, N., Pironneau, O.: Automatic differentiation in C++ using expression templates and application to a flow control problem. Computing and Visualization in Science **3**, 197–208 (2001)
2. Aziz, K., Settari, A.: Petroleum Reservoir Simulation. Elsevier Applied Science (1979)
3. Bischof, C.H., Carle, A., Corliss, G.F., Griewank, A., Hovland, P.D.: ADIFOR: Generating derivative codes from Fortran programs. Scientific Programming **1**(1), 11–29 (1992)
4. Bischof, C.H., Khademi, P.M., Bouaricha, A., Carle, A.: Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. Optimization Methods and Software **7**, 1–39 (1997)
5. Bischof, C.H., Roh, L., Mauer, A.: ADIC — An extensible automatic differentiation tool for ANSI-C. Software–Practice and Experience **27**(12), 1427–1456 (1997). DOI 10.1002/(SICI)1097-024X(199712)27:12⟨1427::AID-SPE138⟩3.0.CO;2-Q. URL `http://www-fp.mcs.anl.gov/division/software`
6. Bulka, D., Mayhew, D.: Efficient C++: performance programming techniques. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2000)
7. Fischer, H.: Special problems in automatic differentiation. In: A. Griewank, G.F. Corliss (eds.) Automatic Differentiation of Algorithms: Theory, Implementation, and Application, pp. 43–50. SIAM, Philadelphia, PA (1991)
8. Gebremedhin, A.H., Manne, F., Pothen, A.: What color is your jacobian? graph coloring for computing derivatives. SIAM Review **47**(4), 629–705 (2005). DOI 10.1137/S0036144504444711. URL `http://link.aip.org/link/?SIR/47/629/1`
9. Griewank, A.: On automatic differentiation. In: M. Iri, K. Tanabe (eds.) Mathematical Programming, pp. 83–108. Kluwer Academic Publishers, Dordrecht (1989)
10. Karmesin, S., Crotinger, J., Cummings, J., Haney, S., Humphrey, W.J., Reynders, J., Smith, S., Williams, T.: Array design and expression evaluation in POOMA II. In: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments, ISCOPE '98, pp. 231–238. Springer-Verlag, London, UK (1998)
11. Kirby, R.C.: A new look at expression templates for matrix computation. Computing in Science Engineering **5**(3), 66 – 70 (2003)
12. Lee, B., Vuduc, R., Demmel, J., Yelick, K.: Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply. In: Parallel Processing, 2004. ICPP 2004. International Conference on, pp. 169 – 176 vol.1 (2004)
13. Narayanan, S.H.K., Norris, B., Hovland, P., Nguyen, D.C., Gebremedhin, A.H.: Sparse jacobian computation using adic2 and colpack. Procedia Computer Science **4**, 2115 – 2123 (2011). DOI DOI:10.1016/j.procs.2011.04.231. URL `http://www.sciencedirect.com/science/article/pii/S1877050911002894`. Proceedings of the International Conference on Computational Science, ICCS 2011
14. Phipps, E.T., Bartlett, R.A., Gay, D.M., Hoekstra, R.J.: Large-scale transient sensitivity analysis of a radiation-damaged bipolar junction transistor via automatic differentiation. In: C.H. Bischof, H.M. Bücker, P.D. Hovland, U. Naumann, J. Utke (eds.) Advances in Automatic Differentiation, pp. 351–362. Springer (2008). DOI 10.1007/978-3-540-68942-3_31
15. Rall, L.B.: Perspectives on automatic differentiation: Past, present, and future? In: H.M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.) Automatic Differentiation: Applications, Theory, and Implementations, *Lecture Notes in Computational Science and Engineering*, vol. 50, pp. 1–14. Springer, New York, NY (2005). DOI 10.1007/3-540-28438-9_1
16. Younis, R.M.: Modern advances in software and solution algorithms for reservoir simulation. Ph.D. thesis, Stanford University (2002)