Optimization of Sparse Generalized Jacobian Chain Products

CES-Master Seminar, SS 2020, RWTH Aachen University
Computational Engineering Science M.Sc.

TAMME CLAUS
RWTH Aachen University
Mat.Nr: 346980
tamme.claus@rwth-aachen.de

supervised by

UWE NAUMANN
Software and Tools for Computational Engineering (STCE)
RWTH Aachen University
naumann@stce.rwth-aachen.de

Attachements

- presentation: `TammeClaus_OptimizationOfSparseGeneralizedJacobianChainProducts_Presentation.pdf`
- source code: `GSJCPB/`
  - solver: `gsjcpb_solve.cpp`
  - problem generator: `gsjcpb_generate.cpp`
  - example problems: `problem_example`, `problem_suitesparse`, `problems*/*`
  - example solutions: `solutions*/*`

OPTIMIZATION OF SPARSE GENERALIZED JACOBIAN CHAIN PRODUCTS[*]

TAMME CLAUS[†]

Abstract. With growing application of algorithms from algorithmic/computational differentiation (AD) in various fields of computational engineering and science the need of its efficient application is evident. While being conceptually easy to understand, the efficient implementation of the AD methods is an open research question. In this work the optimization of sparse generalized jacobian chain products is considered as one approach to optimize the number of necessary fused multiply-add operations prior to evaluating the derivative using the AD modes adjoint and tangent. The result of this optimization can be considered as a mission plan for the subsequently executed AD-tool. The origin of the problem will be unveiled by briefly explaining the concepts of AD, then individual costs arising in the problem will be analyzed. Afterwards the optimization of the problem is implemented using a dynamic programming approach. A case study, based on the described optimization finally shows the need for AD mission planning by illustrating the unveiled reductions in the number of fused multiply add operations.

Key words. sparse generalized jacobian chain product, algorithmic differentiation, computational differentiation, ad mission planning, adjoint mode, tangent mode, dynamic programming, sparse matrix product, jacobian compression

AMS subject classifications. 65-04, 65D25, 90C27, 90C39

1. Introduction.

The discovery of the derivative back in the 1600s represents such a fundamental building block in the development of calculus, that even today there is controversy about who actually discovered it first.[13, 15] As the local rate of change of a (continuous, locally differentiable) function $y = F(x) : \mathbb{R}^n \to \mathbb{R}^m$, the derivative is nowadays build upon in most areas of mathematics, physics and engineering. Instead of simple mathematical expressions which can be differentiated by hand, $F(x)$ often hides the evaluation of an (arbitrarily) complex computer simulation or the application of iterative algorithms. Irrespective of whether only simple sensitivities, the application of numerical optimization methods or stochastic analysis are required, the need to calculate derivatives $F'(x)$ of functions $F(x)$ often arises.

Algorithmic/automatic differentiation (AD) provides tools for embedding the calculation of arbitrarily high, exact derivatives into existing program code, with little to no changes to the original code.[5, 10] While the two fundamental modes of AD, tangent and adjoint (see subsection 2.1) are conceptually easy to understand, their efficient application is challenging. Here, we measure efficiency in terms of fused multiply-add (`fma`) operations, since multiplication followed by an addition commonly occurs in tangent and adjoint AD and many general-purpose processors are equipped to execute the `fma` operation in one clock cycle.[4]

The accumulation of a jacobian, the matrix of all derivatives of a vector-valued function F, can be represented by the transformation of a labeled directed acyclic graph (DAG), with vertices for all intermediate elemental operations and edges labeled with respective partial derivatives, to a bipartite graph. The bipartite graph then holds edges from input variables to output variables labeled with the corresponding jacobian entries. The optimal accumulation of jacobians is shown to be a NP-complete problem in [9], including elimination techniques like vertex, edge or face elimination.[8] Therefore

[*]CES-Master Seminar, SS 2020

[†]RWTH Aachen University (Mat.Nr: 346980, tamme.claus@rwth.aachen.de). Supervised by Uwe Naumann, STCE, RWTH Aachen University (naumann@stce.rwth-aachen.de).

bear in mind that a calculation of the optimal accumulation is only effective if the time spent on *finding the optimum + optimal accumulation < naive accumulation.*

While an elimination approach may unveil a small number of required operations to accumulate the jacobian it might be inapplicable in practice, because the DAG can grow to unmanageable sizes. Furthermore the evaluation time of algorithms on modern computers, is not only determined by the `fma`s but also influenced by temporal and spatial data locality.[4] This motivates the approach to optimize on a higher level of granularity, as is done here in contrast to the graph elimination techniques in [8]. Obviously a higher granularity limits the `fma` reduction possibilities, but facilitates the solving of the optimization problem while still yielding significant `fma` reductions compared to a naive accumulation.

Consider a partitioning of F in

$$F = F_q \circ F_{q-1} \circ \ldots \circ F_1, \tag{1.1}$$

with compliant $z_i = F_i(z_{i-1}) : \mathbb{R}^{n_i} \to \mathbb{R}^{m_i}$, $m_i = n_{i+1}$, $n_1 = n$ and $m_q = m$, analogously $z_0 = x$ and $z_q = y$. A possible partitioning could be given by the individual functions of a computer program, however the individual factors F_i are purposefully let arbitrary here. By the multivariate chain rule, the jacobian $F' \in \mathbb{R}^{m \times n}$ is decomposed into the jacobian chain product

$$F' = F'_q \cdot F'_{q-1} \cdot \ldots \cdot F'_1, \tag{1.2}$$

with local jacobians $F'_i(z_{i-1}) \in \mathbb{R}^{m_i \times n_i}$. To calculate such a chained jacobian product, AD tools offer the opportunity to propagate tangents or adjoints through the individual factors F_i. This leaves us with the options to preaccumulate individual F'_i (using tangent or adjoint) followed by matrix-multiplication or the direct propagation of tangents and adjoints.

This work extends the optimization of generalized jacobian chain products assuming dense local factors F'_i given in [12] to sparse local factors. The dynamic programming approach to solve the optimization problem from [12] is extended to sparse local jacobians. Sparsity patterns of all local factors are assumed to be given, but can for a given implementation be determined automatically.[3, 6] Without further information about the factors F_i, their sparsity patterns allow a more efficient implementation of the matrix-matrix product and the application of jacobian compression techniques for less costly accumulation using AD. Dynamic programming has already been successfully applied to similar problems in the efficient application of AD [7] and in the field of sparse-linear algebra [11].

2. Costs.

2.1. Costs of AD modes - tangent and adjoint. Both modes of AD, tangent and adjoint, are fundamentally based on the chain rule. Each local factor $z_i = F_i(z_{i-1})$ can be split up in multiple single assignments $\varphi_j^{(i)}$ representing the elemental arithmetic operations $\{+, -, \sin, \ldots\}$. Then the chain rule allows us to write the derivative of F_i as multiplication of the derivatives of the elemental operations with respect of their direct dependents, which are assumed to be available. Let us formally write the function F_i as the sequence of assignments

$$(v_1^{(i)}, \ldots, v_{n_i}^{(i)})^T = z_{i-1} \tag{2.1a}$$

$$v_j^{(i)} = \varphi_j^{(i)}((v_l^{(i)})_{l \prec j}) \tag{2.1b}$$

$$z_i = (v_{|V_i|-m_i}^{(i)}, \ldots v_{|V_i|}^{(i)})^T, \tag{2.1c}$$

where $v^{(i)}$ are intermediate variables and $|V_i|$ is the necessary number of single assignments. Then the two AD-modes basically calculate

$$\text{(2.2a)} \qquad \dot{v}_j^{(i)} = \sum_{k \prec j} \frac{\partial \varphi_j^{(i)}}{\partial v_k^{(i)}} \dot{v}_k^{(i)} \qquad\qquad \text{(2.2b)} \qquad \bar{v}_k^{(i)} = \sum_{j \succ k} \frac{\partial \varphi_j^{(i)}}{\partial v_k^{(i)}} \bar{v}_j^{(i)},$$

with corresponding initialization of tangents $\dot{v}^{(i)}$ from $\dot{z}_{i-1}$ and adjoints $\bar{v}^{(i)}$ from $\bar{z}_i$. Note the reversed dependencies of tangent and adjoint accumulations. While for the tangent $\dot{v}_j^{(i)}$ all preceding, $k \prec j$, tangents $\dot{v}_k^{(i)}$ and respective derivatives of the elemental operation $\varphi_j^{(i)}$ are considered, for the adjoint $\bar{v}_k^{(i)}$ we consider all succeeding, $j \succ k$, adjoints $\bar{v}_j^{(i)}$ with respective derivatives.

Both modes can be visualized using the DAG $G_i = (V_i, E_i)$ of F_i with vertices V_i representing the intermediate variables $v^{(i)}$. Simultaneously the edges E_i represent direct dependencies between $v^{(i)}$ through $\varphi^{(i)}$ and are labeled with the corresponding partial derivatives. An edge directed from $v_k^{(i)}$ into $v_j^{(i)}$ visualizes direct dependence of j from k. Then for the tangent mode we sum over all incoming edges, while for the adjoint mode we use all outgoing edges.

To accumulate the jacobian F_i' the crucial difference between both modes is, that for the tangent mode one has to initialize/seed input tangents $\dot{z}_{i-1}$ while for the adjoint mode one seeds the outputs $\bar{z}_i$. We denote the simultaneous propagation of multiple tangents, known as vector tangent mode with $\dot{Z}_i = \dot{F}_i \cdot \dot{Z}_{i-1}$. Similarly the simultaneous propagation of multiple adjoints, vector adjoint mode with $\bar{Z}_{i-1} = \bar{Z}_i \cdot \bar{F}_i$.

costs to accumulate jacobians. To populate all intermediate variables and their partial derivatives one forward pass of F is always necessary, its costs can therefore be neglected in the optimization, as done in [12]. Furthermore we assume sufficient memory availability to record all intermediate variables and partial derivatives. While tangents can be computed alongside the forward pass, adjoints require their storage (or recomputation, see checkpointing [10]).

A naive accumulation of the jacobian F_i' using vector tangent mode can be achieved by seeding the identity matrix $I_{n_i} \in \mathbb{R}^{n_i \times n_i}$. Each tangent in I_{n_i} is propagated through the DAG G_i, yielding one `fma` operation per edge. Consequentially the cost for naive vector tangent mode is $|E_i| \cdot n_i$. Given the sparsity pattern of F_i' structurally orthogonal columns can be identified and their jacobian entries can be evaluated simultaneously. Given there are $t_i \leq n_i$ groups of structurally orthogonal columns, this results in a compressed seeding matrix $T_i \in \mathbb{R}^{n_i \times t_i}$ and the cost of accumulating the whole jacobian is given by $|E_i| \cdot t_i$.

Using a similar argumentation, the cost for naive vector adjoint mode, by seeding the identity $I_{m_i} \in \mathbb{R}^{m_i \times m_i}$, is $|E_i| \cdot m_i$. Analogously, using jacobian compression with $a_i \leq m_i$ groups of structurally orthogonal rows, by seeding $A_i \in \mathbb{R}^{a_i \times m_i}$ as adjoints, the cost is $|E_i| \cdot a_i$.

Determination of structurally orthogonal groups can be implemented using graph coloring methods. We refer to [2], where the compression used later in this work is covered as direct, unidirectional. More sophisticated compression, e.g. bidirectional or substitution methods are not considered here. Note that we neglect the cost of the subsequent unpacking of the jacobian, as for direct compression methods the unpacking is reordering of entries. Also we did not exploit internal structure of F_i, being aware of the fact that special structures of the DAG could reduce the number of necessary `fma`s even further.

2.2. Cost of sparse matrix multiplication. While jacobian compression offers one way of saving `fma`s, the sparse matrix-matrix product offers another. Consider the multiplication of two sparse matrices with sparsity patterns $A \in \{0,1\}^{m_2 \times n_2}$ and $B \in \{0,1\}^{m_1 \times n_1}$ with $n_2 = m_1$. Assuming an implementation that exploits sparsity, the cost to multiply the two matrices is

$$\texttt{fma} = \sum_{i=1}^{m_2} \sum_{j=1}^{n_1} \sum_{k=1}^{n_2} A_{ik} B_{kj} \leq m_2 n_1 n_2. \tag{2.3}$$

This reduces to counting the non-zero entries of both matrices which have to be multiplied. For dense matrices the cost is $m_2 n_1 n_2$.

The computation of the sparsity pattern $C \in \{0,1\}^{m_2 \times n_1}$ of the product will be used later, we formally state it here as

$$C_{ij} = \begin{cases} 1 & \sum_{k=1}^{m_1} A_{ik} B_{kj} > 0 \\ 0 & \text{else} \end{cases} \qquad i = 1 \ldots n_2, j = 1 \ldots m_1. \tag{2.4}$$

For notes on the efficient implementation of sparsity pattern multiplication and the respective cost of matrix multiplication we refer to section 4.

2.3. Cost of recompression. When using jacobian compression, the recompression of preaccumulated jacobians may be necessary. Consider the calculation of $F'_{2,1} = F'_2 \cdot F'_1 \in \mathbb{R}^{m_2 \times n_1}$ using tangent mode, where a compressed seed $T_{2,1} \in \mathbb{R}^{n_1 \times t_1}$ (compatible with the sparsity pattern of $F'_{2,1}$) and the jacobian $F'_1 \in \mathbb{R}^{m_1 \times n_1}$ are given.

$$\dot{F}_{2,1} \cdot (F'_1 \cdot T_{2,1}) \tag{2.5}$$

The seed $T_{2,1}$ can correspond to a seed T_1 (compatible with the sparsity pattern of F'_1), resulting in a free recompression, but in general, additions may be necessary. As cost of the recompression we assign one `fma` per addition occurring in the matrix-matrix product $F'_1 \cdot T_{2,1}$. Then the cost is given by

$$\texttt{fma} = \sum_{i=1}^{m_1} \sum_{j=0}^{t} \max\{0, \sum_{k=0}^{n_1} (S_1)_{ik} (T_{2,1})_{kj} - 1\}, \tag{2.6}$$

where $S_1 \in \{0,1\}^{m_1 \times n_1}$ is the sparsity pattern of F'_1. The cost for adjoint recompression can be defined analogous. For the implementation of recompression costs we refer to section 4.

3. Generalized sparse jacobian chain product - dynamic programming.

Having identified the two additional `fma` reductions of sparse jacobian chain products compared to dense products, the problem can be defined as follows.

problem statement. For the jacobian chain $F' = F_q \cdot \ldots \cdot F_1$ assume that tangent $\dot{F}_i \cdot \dot{Z}_{i-1}$ and adjoint $\bar{Z}_i \cdot \bar{F}_i$ implementations of the individual factors together with sparsity patterns $S_i \in \{0,1\}^{m_i \times n_i}$ are given. What is the optimal way (minimum number of `fma`) to accumulate F'?

This problem can, like the dense problem, be solved using dynamic programming. For the dense problem the formal proof is given in [12]. As the proof for the sparse problem would follow an almost identical path, it is omitted here and we will extend on the proof in [12] with regards to sparsity.

Consider some subchain $F'_{j,i} \in \mathbb{R}^{m_j \times n_i}$ of the jacobian chain $F' = F'_q \cdot \ldots F'_1$ split at position k with $q \geq j \geq k+1 > k \geq i \geq 1$

$$F'_{j,i} = \underbrace{F'_j \cdot \ \ldots \cdot F'_{k+1}}_{F'_{j,k+1}} \cdot \underbrace{F'_k \cdot \ \ldots \cdot F'_i}_{F'_{k,i}} \tag{3.1}$$

and sparsity pattern of the subchain $S_{j,i}$ with all possible row and column compressions $T^{(s)}_{j,i} \in \mathbb{R}^{n_i \times t^{(s)}_{j,i}}, s \in \mathcal{S}^t_{j,i}$ and $A^{(s)}_{j,i} \in \mathbb{R}^{a^{(s)}_{j,i} \times m_i}, s \in \mathcal{S}^a_{j,i}$. Assume that all jacobians of length $< j - i$ are given. The options to accumulate $F'_{j,i}$ are

$$F'_{j,k+1} \cdot F'_{k,i} \tag{3.2a}$$

$$\dot{F}_{j,i} \cdot T^{(s)}_{j,i} \quad \text{or} \quad \dot{F}_{j,k+1} \cdot (F'_{k,i} \cdot T^{(s)}_{j,i}) \tag{3.2b}$$

$$A^{(s)}_{j,i} \cdot \bar{F}_{j,i} \quad \text{or} \quad (A^{(s)}_{j,i} \cdot F'_{j,k+1}) \cdot \bar{F}_{k,i}. \tag{3.2c}$$

Equation (3.2a) uses the previously accumulated jacobians of the left $F'_{j,k+1}$ and right $F'_{k,i}$ subchain and uses matrix multiplication to calculate $F'_{i,j}$. The cost for this option is obtained from the cost to accumulate the jacobians of the two subchains $\texttt{fma}_{j,k+1}$ and $\texttt{fma}_{k,i}$ plus the cost of the matrix multiplication $\texttt{fma}_{j,k,i}$ (see subsection 2.2).

In equation (3.2b) we use tangent vector mode to propagate the compressed seed $T^{(s)}_{j,i}$ (homogeneous tangent) or $F'_{k,i} \cdot T^{(s)}_{j,i}$ through the whole (homogeneous) or the remaining left part of the subchain. The costs are given by $\sum_{\nu=i}^{j} |E_\nu| \cdot t^{(s)}_{j,i}$ (homogeneous) or the cost to accumulate the left subchain $\texttt{fma}_{k,i}$ with its recompression $\texttt{fma}^{(s)}_{j,k,i}$ (see subsection 2.3) plus the cost of tangent propagation $\sum_{\nu=k+1}^{j} |E_\nu| \cdot t^{(s)}_{j,i}$.

In equation (3.2c) we use adjoint vector mode to propagate the compressed seed $A^{(s)}_{j,i}$ (homogeneous adjoint) or $A^{(s)}_{j,i} \cdot F'_{j,k+1}$ through the whole (homogeneous) or remaining right part of the subchain. Analogously the costs are: $\sum_{\nu=i}^{j} |E_\nu| \cdot a^{(s)}_{j,i}$ (homogeneous) or the cost to accumulate the right subchain $\texttt{fma}_{j,k+1}$ with its recompression $\texttt{fma}^{(s)}_{j,k+1,i}$ plus the cost of adjoint propagation $\sum_{\nu=i}^{k} |E_\nu| \cdot a^{(s)}_{j,i}$.

dynamic programming recurrence. A solution of the dense generalized jacobian chain product bracketing problem can be computed using the following dynamic programming recurrence:

$$\texttt{fma}_{j,i} = \begin{cases} |E_j| \cdot \min\{\min\limits_{s\in\mathcal{S}^a_{j,j}} \{a^{(s)}_j\}, \min\limits_{s\in\mathcal{S}^t_{j,j}} \{t^{(s)}_j\}\} \quad \text{(I)} & i = j \\ \min \left\{ \begin{array}{l} \min\{\min\limits_{s\in\mathcal{S}^a_{j,i}} \{a^{(s)}_{j,i}\}, \min\limits_{s\in\mathcal{S}^t_{j,i}} \{t^{(s)}_{j,i}\}\} \sum\limits_{\nu=i}^{k} |E_\nu|, \quad \text{(II)} \\ \min\limits_{i\leq k<j} \left\{ \min \left\{ \begin{array}{l} \texttt{fma}_{j,k+1} + \texttt{fma}_{k,i} + \texttt{fma}_{j,k,i}, \quad \text{(III)} \\ \texttt{fma}_{j,k+1} + \min\limits_{s\in\mathcal{S}^a_{j,i}} \{\texttt{fma}^{(s)}_{j,k+1,i} + a^{(s)}_{j,i} \sum\limits_{\nu=i}^{k} |E_\nu|\}, \quad \text{(IV)} \\ \texttt{fma}_{k,i} + \min\limits_{s\in\mathcal{S}^t_{j,i}} \{\texttt{fma}^{(s)}_{j,k,i} + t^{(s)}_{j,i} \sum\limits_{\nu=k+1}^{j} |E_\nu|\} \quad \text{(V)} \end{array} \right\} \right\} \end{array} \right\} & j > i \end{cases} \tag{3.3}$$

As seen in [12] the proof is provided by induction over $j - i$, starting with the base cases $j = i$ and $j = i + 1$. Search space for the product of two sparse jacobians $F'_{i+1} \cdot F'_i$:

1. Homogeneous tangent $\dot{F}_{i+1} \cdot (\dot{F}_i \cdot T^{(s)}_{i+1,i})$ [covered by (II)]

2. Preaccumulation of F_i' using tangent mode $\dot{F}_i \cdot T_i^{(r)}$ and subsequent tangent propagation $\dot{F}_{i+1} \cdot (F_i' \cdot T_{i+1,i}^{(s)})$ [covered by (I) and (V)]
3. Preaccumulation of F_i' using adjoint mode $A_i^{(r)} \cdot \bar{F}_i$ and subsequent tangent propagation $\dot{F}_{i+1} \cdot (F_i' \cdot T_{i+1,i}^{(s)})$ [covered by (I) and (V)]
4. Homogeneous adjoint $(A_{i+1,i}^{(s)} \cdot \bar{F}_{i+1}) \cdot \bar{F}_i$ [covered by (II)]
5. Preaccumulation of F_{i+1}' using tangent mode $\dot{F}_{i+1} \cdot T_{i+1}^{(r)}$ and subsequent adjoint propagation $(A_{i+1,i}^{(s)} \cdot F_{i+1}') \cdot \bar{F}_i$ [covered by (I) and (IV)]
6. Preaccumulation of F_{i+1}' using adjoint mode $A_{i+1}^{(r)} \cdot \bar{F}_{i+1}$ and subsequent adjoint propagation $(A_{i+1,i}^{(s)} \cdot F_{i+1}') \cdot \bar{F}_i$ [covered by (I) and (IV)]
7. Preaccumulation of F_i' using tangent mode $\dot{F}_i \cdot T_i^{(s)}$ and preaccumulation of F_{i+1}' using tangent mode $\dot{F}_{i+1} \cdot T_{i+1}^{(s)}$ and subsequent matrix multiplication [covered by (I) and (III)]
8. Preaccumulation of F_i' using tangent mode $\dot{F}_i \cdot T_i^{(s)}$ and preaccumulation of F_{i+1}' using adjoint mode $A_{i+1}^{(s)} \cdot \bar{F}_{i+1}$ and subsequent matrix multiplication [covered by (I) and (III)]
9. Preaccumulation of F_i' using adjoint mode $A_i^{(s)} \cdot \bar{F}_i$ and preaccumulation of F_{i+1}' using tangent mode $\dot{F}_{i+1} \cdot T_{i+1}^{(s)}$ and subsequent matrix multiplication [covered by (I) and (III)]
10. Preaccumulation of F_i' using adjoint mode $A_i^{(s)} \cdot \bar{F}_i$ and preaccumulation of F_{i+1}' using adjoint mode $A_{i+1}^{(s)} \cdot \bar{F}_{i+1}$ and subsequent matrix multiplication [covered by (I) and (III)]

In contrast to [12], where case 1 could also be interpreted as case 2, here we have to separate them because of the potentially different seed compressions $T_{i+1,i}^{(s)}$ and $T_i^{(r)}$, which can yield different costs. Analogous for cases 4 and 6.

Furthermore, the superiority of the homogeneous preaccumulation cases 7, 8, 9 and 10 cannot be shown without knowledge of sparsity pattern and compressions. However, they are all covered by equation [I] and equation [III].

To complete the proof [12] shows $1 \leq l \implies l+1$ by proving the two properties: *overlapping subproblems* and *optimal substructure*. For the sparse problem both properties can be shown analogously.

3.1. Restricting the degrees of freedom. Allowing sparsity and especially the use of unidirectional jacobian compression techniques introduces a another level of complexity (compared to [12]) to the problem. For each factor, the sparsity pattern together with its compression matrices have to be computed, which extends the runtime of optimization in comparison to the dense problem. However, in the real world the rigorously fastest accumulation (minimum `fma`s) of the jacobian is not necessary, therefore further restrictions of the degrees of freedom in favor of the runtime are to be considered.

Instead of all possible jacobian compressions $T_{j,i}^{(s)}$ or $A_{j,i}^{(s)}$ it might be enough to consider only one, e.g the one which needs the fewest number of function evaluations (where $t_{j,i}^{(s)}$ or $a_{j,i}^{(s)}$ are minimal) or the one unveiled by a heuristic. In the dynamic programming recurrence (3.3) the search for the $\min_s$ is skipped and replaced with only one, e.g. $s_{j,i}^* = \arg\min_s t_{j,i}^{(s)}$.

Further, one can also completely neglect jacobian compressions and only consider the `fma` reductions caused by the sparse matrix multiplication. Then the specific consideration of homogeneous tangent and adjoint (equation (II)) in the dynamic programming recurrence (3.3) is not required and the recurrence reduces to the one given in [12] with the distinction in matrix-multiplication costs. In section 4 a case study with these both restrictions is carried out.

3.2. Example. An exemplary optimization is shown in Figure 1. It considers the `fma` minimization of accumulating the jacobian $F'_{5,1} \in \mathbb{R}^{4\times 3}$ of the function $F_5 \circ \ldots \circ F_1$ with different propagation costs $|E_i|$ and sparsity patterns. Unidirectional compression is considered, but as proposed in subsection 3.1 we restrict the search space to only one compression for tangent and adjoint. The coloring is represented in the Figure 1 by letters $a - d$ alongside the sparsity patterns. For each of the subchains the costs of all accumulation possibilities are notated, the optimal branches are underlined and the optimal accumulation cost is tabulated. With arrows the use of (smaller) preaccumulated subchains in the optimal accumulation is illustrated.

The optimal accumulation has a cost of $\texttt{fma}_{5,1} = 204$ and is given by:

- compressed preaccumulation of F'_1 using adjoint mode (cost 5`fma`)
- compressed preaccumulation of F'_2 using tangent mode (cost 15`fma`)
- preaccumulation of $F'_{5,2}$ using tangent mode $\dot{F}_5 \cdot \dot{F}_4 \cdot \dot{F}_3 \cdot F'_2$ (cost 172`fma`)
- dense matrix multiplication to calculate $F'_{5,1} = F'_{5,2} \cdot F'_1$ (cost 12`fma`)

For comparison, the cost of naive homogeneous adjoint is 424`fma`, the cost of naive homogeneous tangent is 318`fma` and the cost of homogeneous preaccumulation (of all factors followed by an optimal matrix product bracketing) is 328`fma`, which is a complexity improvement of 40% to 50%. This example is also attached to the work in file form and can be found in `problem_example`.

Without consideration of jacobian compression, the costs of preaccumulation of F'_1 and F'_2 would increase to 10`fma` and 30`fma`, which also changes the further optimal accumulations. Then the optimal accumulation has a cost of $\texttt{fma}_{5,1} = 222$ and is given by:

- preaccumulation of $F'_{5,2}$ using tangent mode (cost 202`fma`)
- adjoint propagation to calculate the jacobian $F'_{5,1} = F'_{5,2} \cdot \bar{F}_1$ (cost 20`fma`)

Restricting the problem further to only dense local factors does not worsen the optimum, because no matrix-multiplication is used in the optimal accumulation. Hence the accumulation of the jacobian for this problem assuming dense factors also yields a cost of 222`fma`.

4. Implementation and Case Study.

The implementation of the optimization extends the implementation seen in [12]. The dynamic programming routine is extended to trace sparsity patterns of each subchain. A sparsity pattern is stored as `std::vector<boost::dynamic_bitset<>>`, allowing a fast[14] and convenient implementation of the multiplication, cost and compression routines. Each sparsity pattern is stored twice, with row-major and column-major ordering, enabling a efficient sparsity pattern multiplication. The storage requirement of one sparsity pattern then is $\sim$ 2`bits` per matrix element, since each one is stored twice. The `boost::dynamic_bitset<>` class offers fast element-wise bit operations (`AND` and `OR`) as well as `.count()` which counts the number of `1` bits. Both are exploited to calculate multiplication and compression costs. In contrast to `std::bitset<>` the boost implementation is dynamic in size, which is advantageous in this application, since the size of each factor is problem dependent.

The algorithms to calculate row and column compressions follow the naive natural ordering heuristic. Starting with the first row/column all other ones are analyzed in ascending order, structurally orthogonal ones are identified and labeled with the same

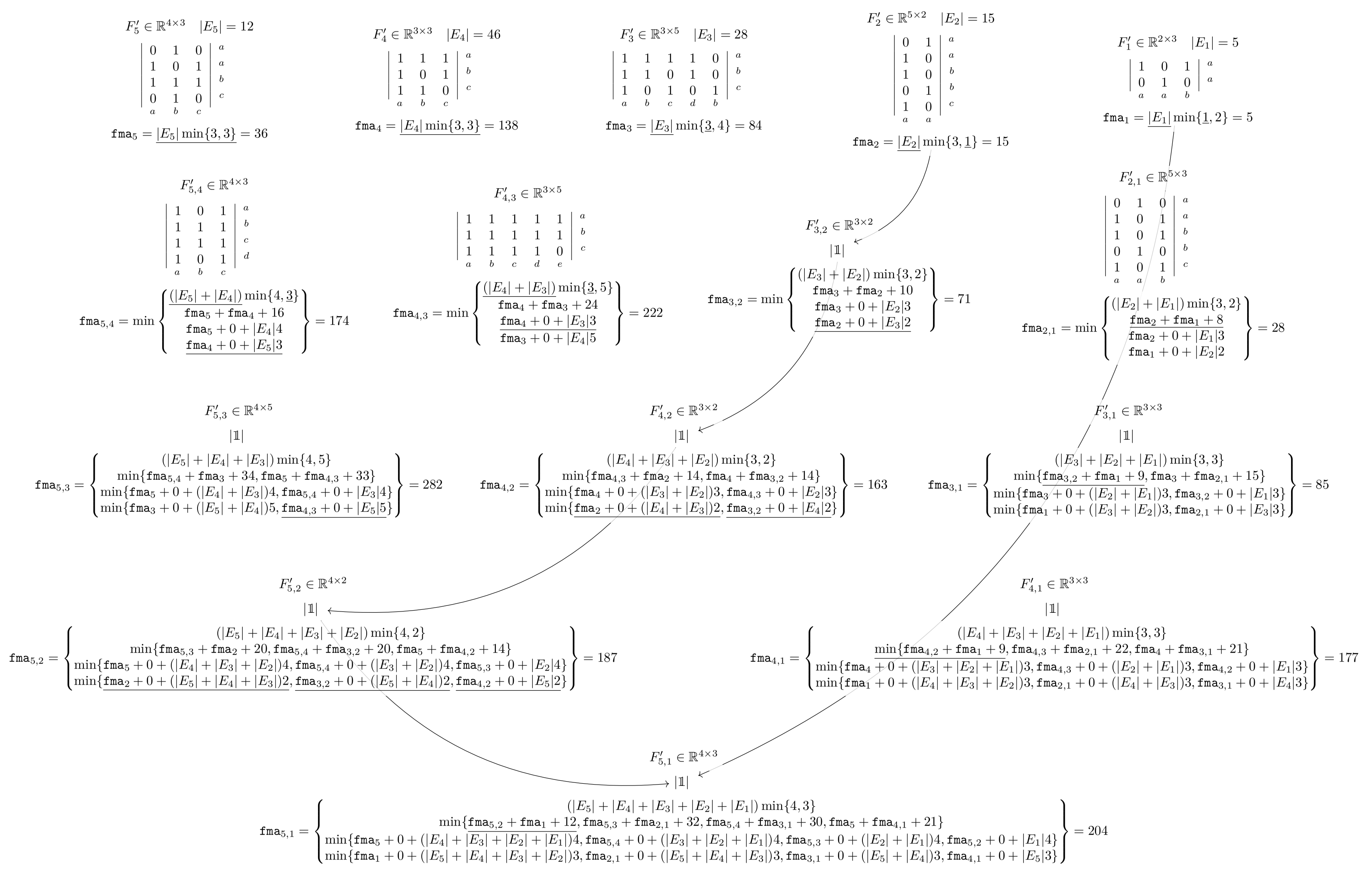
FIG. 1: *Example of the generalized sparse jacobian chain product with unidirectional jacobian compression. Sparsity patterns of the subchains are denoted with* $\{1,0\}^{m\times n}$ *with corresponding compressions* $(a-d)$. $|\mathbb{1}|$ *denotes a dense matrix. For each subchain the whole search space is notated, where the respective minima are underlined. Arrows indicate the use of preaccumulated smaller subchains in the optimal accumulation.*

```
q // n of factors (F_i's)
m_0 n_0 |E_0| // n of outputs, n of inputs, n of elemental function
  calls of F_0
0 1 0 ...// sparsity patterns (depend of the dimensions of each F_i)
0 1 1 ...
1 0 1 ...
...
m_1 n_1 |E_1| // n outputs, n inputs, n elemental fc's of F_1
//sparsity pattern of F_1
...
m_q n_q |E_q| // n outputs, n inputs, n elemental fc's of F_q
// sparsity pattern of F_q
...

```

FIG. 2: *Problem file structure. Gray colored text is included as explanation here and should be omitted in the problem file.*

color. The same procedure is continued until all rows/columns have a color assigned. However, another heuristic can be implemented as a drop-in replacement for this naive implementation. Note that in the implementation the optimization is restricted to only one column and one row compression per factor, as proposed in subsection 3.1.

The optimization is implemented in `gsjcpb_solve.cpp` and can be executed using `gsjcpb_solve.exe problem_file s c` after building the project files using `make`. Thereby `s` toggles the optimization of the sparse problem or the dense problem and `c` toggles the jacobian compression. If `s` or `c` are any other character than `0`, the optimization runs in sparse or compression mode respectively. `problem_file` should point to a text file defining the specific problem. The assumed structure of the file is shown in Figure 2.

The project also includes a random problem generator `gsjcpb_generate.exe q max_nm d`, which prints a randomly generated problem to `stdout`. The number of factors of the problem to be generated can be defined by `q`, with `max_nm` the maximal dimension of each factor can be configured (each dimension is randomly selected between 1 and `max_nm`) and with `d` a density of the sparsity patterns can be configured. The sparsity pattern are generated randomly with the requirements that there is at least one non-zero element per row and per column. This assumes that no factor has constant outputs nor unused inputs, which is a valid assumption, because otherwise the respective dimension of the factor could be reduced. Then the remaining zero-elements are successively set to nonzero if $p \sim \text{Uni}([0,1))$, drawn from a uniform distribution, is $p < \frac{\mathtt{d}nm - \max\{n,m\}}{nm - \max\{n,m\}}$. This yields that in expectation the density of the sparsity pattern is equal to `d`, if `d` is greater than the minimum density specified by the mentioned requirement. Setting $d = 0$ will not generate any additional non-zeros.

case study. In Table 1 results of the accumulation costs for problems of growing size are presented. All results are based on the problems provided under the folder `/problems/problem*`, which were randomly generated by `gsjcpb_generate.exe q max_nm 0`, so no additional non-zeros apart from one per column/row were added to the sparsity pattern. In Table 1a all factors are assumed to be dense, while in Table 1b the sparsity is considered in the matrix-matrix product without further cost reductions using jacobian compression. Table 1c then additionally includes the cost reductions using unidirectional jacobian compression (unveiled by the naive coloring heuristic).

Besides the optimal cost unveiled by the previously described algorithm, all tables also include the cost of a naive tangent and naive adjoint accumulation as well as a naive preaccumulation approach, where initially all individual factors are preaccumulated and afterwards combined by an optimal matrix product bracketing.

Similarly to the results presented in [12], enormous reductions in the number of `fma`s are visible even for the dense case (Table 1a). The costs for naive tangent and naive adjoint do not change for the different restrictions, because for all problems the resulting jacobian is dense. For optimal preaccumulation the costs improve slightly from the dense case to the sparse case due to the reductions in sparse matrix multiplication (Table 1b). Allowing jacobian compression considerably improves the costs, as each individual factor can be accumulated in one tangent or adjoint pass (due to the problem generation, Table 1c). The optimal number of `fma`s does not significantly improve for both relaxations, only the cost for the problem ($\texttt{q} = 50, \texttt{max\_nm} = 50$) reduces using jacobian compression. This suggests, that the effectiveness is highly problem dependent and cannot be generalized. More case studies with random sparsity pattern, but a higher initial density (0.25 and 0.5) show similar results (the problem files can be found in `/problems025/problem*` and `problems05/problem*`).

In a test case, where the accumulation of the product of 5 jacobians (dimension between $\sim 200 - 1100$, random cost per factor between $100 - 1000$) with structured sparsity patterns from the SuiteSparse Matrix Collection [1] was considered, `fma` reductions are present. While for the dense problem 617210`fma` are calculated, the sparse problem without compression needed 608846`fma` but with compression only 413006`fma` are necessary. The problem file is also attached to the work (`problem_suitesparse`).

5. Conclusion. In this work we analyzed the cost reductions arising in the generalized jacobian chain product when assuming sparse factors. Building upon [12] the dynamic programming recurrence is expanded to also consider the cost reductions from sparse matrix-matrix multiplication and jacobian compression. With the aid of an example, the search space of each factor in the generalized sparse jacobian chain product is illustrated. Besides some comments on the implementation, the implementation is used to conduct a case study, where problems of growing size are analyzed with different restrictions on the optimization problem. The results show the importance of problem analysis, prior to applying algorithmic differentiation tolls to calculate jacobians.

Optimizing the accumulation of jacobians is only useful if the time spent on *finding the optimum + optimal accumulation < naive accumulation*. Therefore one has to balance between restrictions on degrees of freedom of the optimization and finding the rigorous minimum. The more different levels of granularity available, the better this balance can be found. Then an adaptive optimization, which can decide between different restrictions, could find the balance between optimization time and evaluation time.

Even more granularity, than discussed in this work, could be introduced to the problem by assuming the function, which is to be differentiated, to be a graph of smaller functions instead of the function composition considered here. Another more technical improvement would be the considerations of memory constraints, especially for adjoint propagations. Here we assumed, that sufficient memory is available to record all intermediate variables and partial derivatives of the whole function, which for large simulations obviously does not hold.

Ultimately we think that this work is one of many building blocks in the implementation of a AD mission planning framework, which coupled with an algorithmic differentiation package is simply useful.

length `q`	`max_mn`	Tangent	Adjoint	Preaccumulation	Optimum
10	10	296	2072	779	296
50	50	839925	604746	740518	135228
100	100	19160886	21611697	9034346	231143
150	150	56863838	47527984	53425505	852466
200	200	34648499	381133489	174604509	2041326

(A) *Optimal accumulation costs; assuming dense local factors. They are given for comparison with the work [12].*

length `q`	`max_mn`	Tangent	Adjoint	Preaccumulation	Optimum
10	10	296	2072	681	296
50	50	839925	604746	620148	134838
100	100	19160886	21611697	8808143	231143
150	150	56863838	47527984	52552009	852466
200	200	34648499	381133489	172585171	2041326

(B) *Optimal accumulation costs; considering the cost reductions due to the sparse matrix product. Jacobian compression is not considered here.*

length `q`	`max_mn`	Tangent	Adjoint	Preaccumulation	Optimum
10	10	296	2072	345	296
50	50	839925	604746	36767	36132
100	100	19160886	21611697	237339	231143
150	150	56863838	47527984	867098	852466
200	200	34648499	381133489	2067283	2041326

(C) *Optimal accumulation costs; considering both reductions, due to the sparse matrix product and jacobian compression. Here only the jacobian coloring unveiled by a naive natural ordering heuristic is applied, as proposed in subsection 3.1.*

TABLE 1: *Optimal accumulation costs for the generalized dense/sparse jacobian chain product with problems of different size and length for the restrictions discussed in subsection 3.1. `q` denotes the number of factors, `max_nm` the maximal number of dimensions. The accumulation costs using naive homogeneous tangent mode, naive homogeneous adjoint mode, a homogeneous preaccumulation (all local jacobians are preaccumulated and multiplied by optimal matrix product bracketing) and the optimal accumulation are tabulated.*

REFERENCES

[1] T. A. Davis and Y. Hu, *The university of florida sparse matrix collection*, ACM Trans. Math. Softw., 38 (2011), https://doi.org/10.1145/2049662.2049663.

[2] A. Gebremedhin, F. Manne, and A. Pothen, *What color is your jacobian? graph coloring for computing derivatives*, SIAM review, 47 (2005), pp. 629–705, https://doi.org/10.1137/S0036144504444711.

[3] R. Giering and T. Kaminski, *Automatic sparsity detection implemented as a source-to-source transformation*, in Computational Science – ICCS 2006, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, eds., Berlin, Heidelberg, 2006, Springer Berlin Heidelberg, pp. 591–598.

[4] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes*, Society for Industrial and Applied Mathematics, 2001, https://doi.org/10.1137/1.9780898718218.

[5] A. Griewank, *A mathematical view of automatic differentiation*, Acta Numerica, 12 (2003), p. 321–398, https://doi.org/10.1017/S0962492902000132.

[6] A. Griewank and C. Mitev, *Detecting jacobian sparsity patterns by bayesian probing*,

Math. Program., 93 (2002), pp. 1–25, https://doi.org/10.1007/s101070100281.

[7] A. Griewank and U. Naumann, *Accumulating jacobians as chained sparse matrix products*, Math. Prog, 3 (2002), p. 2003.

[8] U. Naumann, *Optimal accumulation of jacobian matrices by elimination methods on the dual computational graph*, Math. Program., 99 (2004), pp. 399–421, https://doi.org/10.1007/s10107-003-0456-9.

[9] U. Naumann, *Optimal jacobian accumulation is np-complete*, Mathematical Programming, 112 (2008), pp. 427–441, https://doi.org/10.1007/s10107-006-0042-z.

[10] U. Naumann, *The Art of Differentiating Computer Programs*, Society for Industrial and Applied Mathematics, 2011, https://doi.org/10.1137/1.9781611972078.

[11] U. Naumann, *On Sparse Matrix Chain Products*, 01 2020, pp. 118–127, https://doi.org/10.1137/1.9781611976229.12.

[12] U. Naumann, *Optimization of generalized jacobian chain products without memory constraints*, 2020, https://arxiv.org/abs/2003.05755.

[13] I. Newton, *Philosophiae naturalis Principia mathematica*, Cantabrigiae, [Cambridge University], editio secunda ed., 1713 [first edition 1687].

[14] V. Pieterse, D. Kourie, L. Cleophas, and B. Watson, *Performance of c++ bitvector implementations*, 01 2010, pp. 242–250, https://doi.org/10.1145/1899503.1899530.

[15] G. W. von Leibniz, *Nova methodus pro maximis et minimis, itemque tangentibus, quae nec fractas nec irrationales quantitates moratur, et singulare pro illis calculi genus*, Lipsiae, 1684.