Elimination Techniques on Linearized Computational Graphs and Dual Graphs with an Emphasis on Data Locality

Andrew Lyons
Department of Electrical Engineering and Computer Science
Vanderbilt University
2201 West End Avenue, Nashville, TN 37235, USA
andrew.m.lyons@vanderbilt.edu

Abstract

Use of the chain rule in the preaccumulation of Jacobian matrices yields a computationally complex search space of elimination sequences. Current techniques attempt to minimize arithmetic operations in the generated code, which is generally considered to be an NP-hard problem, though no proof currently exists.

The heuristics described in this paper focus on generating code that makes use of cache and other fast memory to speed execution. We describe heuristics that focus on data locality for vertex and edge elimination on linearized computational graphs, as well as heuristics for face elimination on dual graphs.

1. Introduction

The heuristic model presented here has been implemented as part of a piece of software called xaifBooster, which is written in C++ and uses the boost graph library. xaifBooster receives xml code that describes directed acyclic graphs known as linearized computational graphs (LCGs). These graphs represent the code (FORTRAN or C++) that describes some vector function F whose Jacobian matrix F' we seek.

The component described in this paper creates a copy G of the linearized computational graph and performs a series of l eliminations on it that reduce it to a bipartite graph $G^{(l)}$ which represents the Jacobian of F. Each of these eliminations generates a corresponding Jacobian Accumulation Expression graph (JAE graph) which is eventually turned into a line of Jacobian accumulation code. The sequence of eliminations performed on G, the LCG of F, determines (uniquely) a block of Jacobian accumulation code, which is computationally substantial.

Current popular heuristics for Jacobian accumulation [1, 2] focus solely on minimizing floating point operations, the result is that they greedily choose elimination targets from around the graph and the accumulation code they produce will not take advantage of the benefits of caching. The heuristics described here are designed to produce code that will reuse bits of data immediately, while they still reside in fast memory.

2. Determination of Elimination Sequences

For a LCG G, a set of elimination targets $\Theta_G = \{\theta | \theta$ is a valid elimination target in $G\}$ is constructed. A heuristic, denoted h_k, is a map from one such set $\Theta_G^{(k-1)}$ to a subset $\Theta_G^{(k)}$. Elements in the target set $\Theta_G^{(k)}$ constitute the most favorable equivalence class for h_k, based on its particular criteria. Thus if all of the elements in $\Theta_G^{(k-1)}$ are in the same equivalence class with respect to h_k, then $\Theta_G^{(k)} = \Theta_G^{(k-1)}$.

In the event of a heuristic h_k returning target set $\Theta_G^{(k)}$ such that $|\Theta_G^{(k)}| > 1$, $\Theta_G^{(k)}$ must be passed to another heuristic h_{k+1}, and so on until the target set consists of a single element. The elimination of this single element transforms $G^{(h)}$ into $G^{(h+1)}$. In this way, what is defined as a hierarchy of heuristics $(h_1, h_2, ..., h_q)$ will make a distinct sequence of l eliminations, thereby turning G into the bipartite graph $G^{(l)}$.

Thus, for any particular graph G the unique elimination target chosen by a hierarchy of q heuristics is determined by the following formula:

$$\Theta_G^{(q)} = h_q(h_{q-1}(...h_1(\Theta_G^{(h-1)})...)).$$

In order to ensure that $|\Theta_G^{(q)}| = 1$, we require that the last heuristic in the hierarchy always return a unique selection (Placing such a heuristic anywhere but last in the hierarchy would render all subsequent heuristics useless). Forward or reverse mode heuristics will always return a unique selection, as they are implemented based on one particular topological sort of the original graph G. see [3] for more information on forward and reverse heuristics.

3. Vertex Elimination Heuristics

Vertex elimination is executed on linearized computational graphs $G = (V, E)$. Three sets of vertices partition V: the set of independent vertices V_I, the set of dependent vertices V_D, and the set of "eliminatable" (or intermediate) vertices V_E. For vertex elimination on a LCG G, the set of eliminatable targets is defined by

$$\Theta_G = \{\theta | \theta \in V_E\}.$$

Definition 1 *$\forall v_i \in V$ the* predecessor set *of v_i, denoted P_{v_i}, is $\{v_j | v_j \in V, (j, i) \in E\}$.*

Definition 2 *$\forall v_i \in V$ the* successor set *of v_i, denoted S_{v_i}, is $\{v_j | v_j \in V, (i, j) \in E\}$.*

Elimination of a vertex θ is executed by creating $(p, s) \in E$, $\forall v_p \in P_\theta$, $v_S \in S_\theta$. If (p, s) already exists, we conduct what is known as a fused multiply-add (FMA), and increment the existing edge with the value of the newly created one. θ and its incident edges are then removed from the graph. Thereby, a complete sequence of vertex eliminations reduces G to a bipartite graph with $V = V_I \cup V_D$ and edges $(i, d) \in E$ (where $v_i \in V_I$ and $v_d \in V_D$) whose labels represent the Jacobian entries. For more information on vertex elimination, see [1].

In addition to the set of eliminatable targets Θ, vertex elimination heuristics are made aware of P_{θ^-} and S_{θ^-}, the sets of predecessors and successors of θ^-, the most recently eliminated target.

3.1. Highest Sibling

Highest vertex sibling degree, or HS_v, selects targets that have the highest *sibling degree*, denoted sd_{max}, with respect to the previous elimination, θ^-. $\forall \theta \in \Theta_G$, the sibling degree of θ, denoted $sd_{\theta^-}(\theta)$, is

$$sd_{\theta^-}(\theta) = |S_\theta \cap S_{\theta^-}| * |P_\theta \cap P_{\theta^-}| \quad .$$

The maximum sibling degree is defined as follows:

$$sd_{max}(\Theta_G) = \max_{\forall \theta \in \Theta_G} \{sd_{\theta^-}(\theta)\} \quad .$$

HS_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, sd_{\theta^-}(\theta) = sd_{max}(\theta^-)\}$. In the case when $sd_{max} = 0, \Theta_G^{(k)} = \Theta_G^{(k-1)}$.

The elimination of a target θ^+ directly following the elimination of a target θ with $sd_\theta(\theta^+) > 0$ creates code that stipulates $sd_\theta(\theta^+)$ additions to edges created during the previous elimination θ^-, which should still reside in fast memory.

3.2. Successor/Predecessor

Successor/Predecessor, or SP_v, makes selections based on the following criteria:

If $|P_{\theta^-} \cap \Theta_G^{(k-1)}| > 0 \wedge |S_{\theta^-} \cap \Theta_G^{(k-1)}| > 0$,
  If $|P_{\theta^-}| > |S_{\theta^-}|$,
    SP_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \theta \in S_{\theta^-}\}$.
  If $|S_{\theta^-}| > |P_{\theta^-}|$,
    SP_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \theta \in P_{\theta^-}\}$.
  If $|S_{\theta^-}| = |P_{\theta^-}|$,
    SP_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \theta \in P_{\theta^-} \cup S_{\theta^-}\}$.
If $|P_{\theta^-} \cap \Theta| > 0 \wedge |S_{\theta^-} \cap \Theta| = 0$,
  SP_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \theta \in P_{\theta^-}\}$.
If $|P_{\theta^-} \cap \Theta| = 0 \wedge |S_{\theta^-} \cap \Theta| > 0$,
  SP_v selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \theta \in S_{\theta^-}\}$.

When an element from $S_{\theta^-}(P_{\theta^-})$ is selected, $|P_{\theta^-}|$ ($|S_{\theta^-}|$) of its edges should still be in fast memory because they were created (or incremented) during the previous elimination.

3.3. Example

In the interest of simplifying this example, we will assume that only vertices without outedges are considered dependent vertices. In practice, this may not always be the case, for reasons beyond the scope of this paper.

As an example, consider G as depicted in Fig. 1 (a) and the following sequence of vertex elimination heuristics: h_1 = Sibling, h_2 = Successor/Predecessor, h_3 = Markowitz, h_4 = Reverse.

For the first application of this sequence to our graph G, h_1 and h_2 will return the same set Θ_G that they receive, because there is no previous elimination θ^- associated with the first elimination.

h_3 will thus receive the set $\Theta''_G = \{v_3, v_4, v_5, v_6\}$ and will return the set $\Theta'''_G = \{v_3\}$ because v_3 has the lowest Markowitz degree (1 inedge * 1 outedge = 1).

Because a single elimination target has now been chosen, we can make our first elimination by creating new edges (or correspondingly incrementing existing edges) from every predecessor of v_3 to every successor of v_3. The resulting graph G' is shown in Fig. 1 (a).

Next, we construct $\Theta_{G'} = \{v_4, v_5, v_6\}$, set θ^- to v_3, and pass them both to h_1, which is HS_v. Observe that $sd_{v_3}(v_4) = |S_{v_4} \cap S_{v_3}| * |P_{v_4} \cap P_{v_3}| = |1| * |1| = 1$, whereas $sd_{v_3}(v_5) = sd_{v_3}(v_6) = |0| * |0| = 0$. In this way, HS_v chooses a single elimination target v_4, and the result of eliminating vertex v_4 is G'', shown in Fig. 1 (c).

As before, $\Theta_{G''} = \{v_5, v_6\}$ and $\theta^- = v_4$. HS_v is unable to choose between v_5 and v_6 because $sd_{v_4}(v_5) = sd_{v_4}(v_6) = 0$. $\Theta'_{G''} = \{v_5, v_6\}$ is then sent to h_2, which is

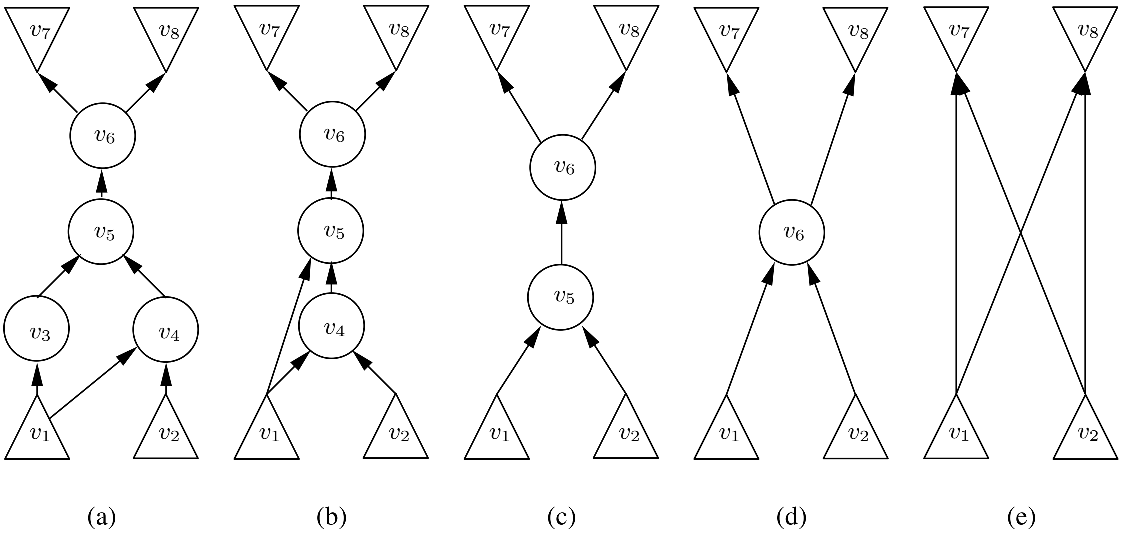


Figure 1. Vertex elimination example

Successor/Predecessor. h_2 determines that $|P_{v_4} \cap \Theta'_{G''}| = 1 > 0$ and $|S_{v_4} \cap \Theta'_{G''}| = 0$, so it selects $\Theta''_{G''} = \{v_5\}$. Fig. 1 (d) shows our LCG after the removal of v_5.

$|\Theta_{G'''}| = 1$, because the only remaining intermediate vertex is v_6, so we can go ahead and eliminate v_6 to obtain the bipartite graph $G^{(4)}$, shown in Fig. 1 (e)

4. Edge Elimination Heuristics

An edge (v_i, v_j) or short (i, j) can be either *front* or *back* eliminated, denoted by $(i, j)_f$ or $(i, j)_b$ respectively. Front elimination is executed by connecting all vertices in the predecessor set $P_{(i,j)_f} = \{v_i\}$, with all vertices in the successor set $S_{(i,j)_f} = S_{v_j}$. These new edges are $\{(i, k)|v_k \in S_{v_j}\}$. Only edges whose target is not an output can be front eliminated. Back elimination is executed by connecting all vertices in the predecessor set $P_{(j,k)_b} = P_{v_j}$ with all vertices in the successor set $S_{(j,k)_b} = \{v_k\}$. The new edges are $\{(i, k)|v_i \in P_{v_j}\}$. Only edges whose source is not an input variable can be back eliminated.

In both case the new edges are labeled with the values $c_{ki} := c_{ji} * c_{kj}$ and the edge (i, j) is removed. If an edge elimination $(i, j)_f$ or $(j, k)_b$ would create an edge (i, k), where (i, k) already exists, the label of (i, k) is incremented $c_{ki} := c_{ki} + c_{ji} * c_{kj}$. This is referred to as *absorption* as opposed to the creation of new edges which represent *fill-in*.

If at any point during the elimination process an intermediate vertex has no more in- or out-edges, the vertex and all incident edges are removed from the graph. Thereby, a complete sequence of edge eliminations reduces $\mathcal{G}$ to a bipartite graph consisting only of vertices $\in \mathcal{X} \cup \mathcal{Y}$ and edges whose labels represent the Jacobian entries.

Each multiplication or combined increment/multiplication on the edge labels implies a Jacobian accumulation expression (JAE) which is stored in a list.

4.1. Highest Sibling

Highest edge sibling degree, or HS_e, will choose elimination targets that have the maximum *sibling degree* denoted by sd_{max}. $\forall \theta \in \Theta_G$, the sibling degree of θ with respect to the previous elimination θ^-, denoted $sd_{\theta^-}(\theta)$, is defined by

$$sd_{\theta^-}(\theta) = |S_\theta \cap S_{\theta^-}| * |P_\theta \cap P_{\theta^-}| \quad .$$

The maximum sibling degree is defined as follows:

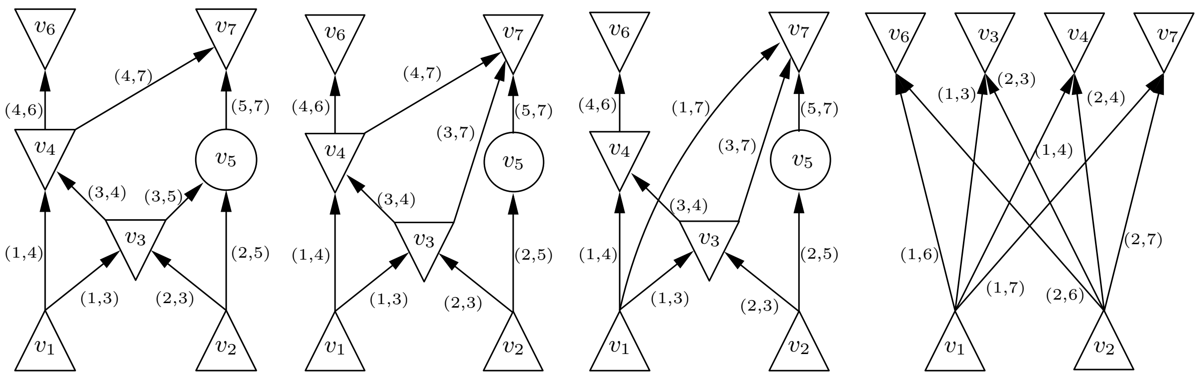
$$sd_{max}(\theta^-) = \max_{\forall \theta \in \Theta_G} \{sd_{\theta^-}(\theta)\} \quad .$$

HS_e selects $\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, sd_{\theta^-}(\theta) = sd_{max}(\theta^-)\}$. In the case when $sd_{max} = 0$, $\Theta_G^{(k)} = \Theta_G^{(k-1)}$. The elimination of a target θ^+ directly following the elimination of a target θ with $sd_\theta(\theta^+) > 0$ creates code that stipulates the immediate absorption of an edge created during the previous elimination θ^-, which should still reside in fast memory.

Note that if the last elimination was a front (back) elimination, any edge being considered for back (front) elimination must have a sibling degree of 1. Thus, HS_e can choose front (back) eliminations following a back (front) elimination only when the maximum sibling degree is 1.

4.2. Example

Consider the following sequence of edge elimination heuristics as applied to the LCG G depicted in Fig. 2 (a): h_1 = Sibling, h_2 = Markowitz, h_3 = Reverse



(a) (b) (c) (d)

Figure 2. Edge elimination example

Again, all elements in Θ_G are in the same equivalence class with respect to all data locality heuristics, thus h_2 (Lowest Markowitz) must be used to choose our first elimination. The Markowitz degree for any front or back edge elimination is defined as $|S|$ or $|P|$, respectively; see [1] for an in-depth description of Markowitz-type heuristics for edge elimination on a LCG. LM chooses both $(3,t)_f$ and $(2,t)_f$ because they are in the same equivalence class, with Markowitz degree 1.

Reverse mode chooses to eliminate $(3,t)_f$ because v_3 occurs after v_2 in every topological sort of G. The resulting graph G' is shown in Fig. 2 (b). Now that we have made an elimination and we have some data in fast memory, we can make use of data locality in order to expedite our accumulation.

Inspection of Fig. 2 (b) reveals that both $(3,4)_f$ and $(4,7)_b$ are siblings (of sibling degree 1) of $(3,t)_f$. However, we cannot eliminate edge $(3,4)_f$ because vertex 4 is a dependent vertex. For reasons dealing with implementation that won't be discussed here, any vertex with more than 1 out-edge must be considered dependent. Hence, every vertex in our graph except for v_5 will be treated as an output. This process continues until we have made l eliminations, and are left with the bipartite graph shown in Fig. 2 (b).

5. Face Elimination Heuristics - Elimination Techniques on the Dual Graph

Face elimination [2] is executed on $\tilde{G} = (\tilde{V}, \tilde{E})$, the dual graph (or line graph) of some LCG G. Edges $(i,j) \in \tilde{E}$, are referred to as "faces" to distinguish edges in G and $\tilde{G}$.

Definition 3 *A face $(i,j) \in \tilde{E}$ is said to be* similar *to a vertex $v \in \tilde{V}$ (denoted $\theta\ v$) if $P_v = P_i$ and $S_v = S_J$.*

A face (i,j) is eliminated by creating a new vertex v, and creating inedges from all $p \in P_i$ and outedges to all $s \in S_j$. If $\exists u \in \tilde{V} s.t.(i,j)\ u$, It's value is incremented by the value of v.

Definition 4 *A face (v_i, v_j) is said to be* intermediate *if $|P_{v_i}| > 0 \wedge |S_{v_j}| > 0$.*

Definition 5 *A vertex $v \in \tilde{V}$ is said to be* final *if there is no path in $\tilde{G}$ from any vertex $p \in P_v$ to any vertex $s \in S_v$ that does not go through v.*

Current implementations of face elimination in dual graphs require checking for cases in which vertices that have identical predecessor and successor sets, as they must be merged into a single vertex.

In our particular implementation, the set $\Theta_{\tilde{G}}$ is composed of faces that are both intermediate and incident with at least one final vertex. It is our belief that only eliminating such final edges will prevent the condition of two different vertices sharing the same predecessor and successor sets, thus alleviating the need for merges. That an optimal elimination sequence resides in the resulting metagraph hinges on proof of what is known as the 'no free refill conjecture'.

5.1. Absorption

Absorb mode, or AB_f, chooses $\Theta_G^{(k)}$ in the following way:

$$\Theta_G^{(k)} = \{\theta | \theta \in \Theta_G^{(k-1)}, \exists v \in \tilde{V} s.t. \theta\ v\}.$$

As always, in the case that $|\Theta_G^{(k)}| = 0$, we set $\Theta_G^{(k)} = \Theta_G^{(k-1)}$. In this way, we ensure that so long as there is such a face in $\tilde{G}$, our elimination will result in what is referred to as an "absorption". AB_f generates cache-friendly code by repeatedly choosing faces whose elimination does not create a new vertex in $\tilde{G}$. The intended consequence is that

$\tilde{G}$ will contain many paths as induced subgraphs, and eliminations straight along these paths are quite efficient with respect to data locality.

References

[1] A. Albrecht, P. Gottschling, and U. Naumann. Markowitz-type heuristics for computing Jacobian matrices efficiently. In *ICCS 2003*, volume 2658 of *LNCS*, pages 575–584, Berlin, 2003. Springer.

[2] U. Naumann. Optimal accumulation of Jacobian matrices by elimination methods on the dual computational graph. *Math. Prog.*, 3(99):399–421, 2004. Published online at `www.springerlink.com`.

[3] U. Naumann, J. Utke, A. Lyons, and M. Fagan. Control flow reversal for adjoint code generation. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2004)*, pages 55–64, Los Alamitos, CA, USA, 2004. IEEE Computer Society.