AD source transformation
&
Performance Metrics

Ian Karlin (U of Colorado; summer student at Argonne)
Jean Utke (Argonne, MCS)

- what is automatic differentiation (AD)
- how is AD done with source transformation
- what variations need metrics
- how far have we come
- open issues
4 motivations for AD

! some numerical model given as a (large) program
? sensitivity analysis, optimization, parameter (state) estimation

1. don’t pretend we know nothing about the program (and take finite differences of an oracle?)

2. get machine precision derivatives (avoid approximation-versus-rounding problem)

3. the reverse mode (adjoint) yields “cheap” gradients

4. if the program is large, so is the adjoint, so is the effort to do it manually ... and it is easy to get wrong but hard to debug

get a tool to do it “automatically”
example - how do directional derivatives come about?

\[ f : y = \sin(a \times b) \times c \]
yields a graph representing the order of computation:

- intrinsics \( \phi(\ldots, w, \ldots) \) have local partial derivatives \( \frac{\partial \phi}{\partial w} \)
- e.g. \( \sin(t1) \) yields \( \cos(t1) \)
- code list→ intermediate values \( t1 \) and \( t2 \)
- all others already stored in variables
- data and statement-level code augmentation

\[ t1 = a \times b \]
\[ p1 = \cos(t1) \]
\[ t2 = \sin(t1) \]
\[ y = t2 \times c \]

What can we do with this?
forward with directional derivatives

\[ f(g(x)) \Rightarrow \dot{f}(g(x))\dot{g}(x) \hat{x} \] multiplications along paths

Assume a point \((a_0, b_0, c_0)\) and a direction \((\dot{a}, \dot{b}, \dot{c}) = (d_a, d_b, d_c)\)

variable and directional derivatives associated in pairs \((v, d_v):\)

\[
d_{a}b*p_{1}c+d_{b}a*p_{1}c+d_{c}t_{2}\]

has common subexpressions
interleave computations of directional derivatives

\[
t_{1} = a*b\\
d_{t1} = d_{a}*b + d_{b}*a\\
p_{1} = \cos(t_{1})\\
t_{2} = \sin(t_{1})\\
d_{t2} = d_{t1}*p_{1}\\
y = t_{2}*c\\
d_{y} = d_{t2}*c + d_{c}*t_{2}\]

What is in \(d_y\)?

note: graph-level code augmentation
forward with directional derivatives II

- if \((\dot{a}, \dot{b}, \dot{c}) = (1, 0, 0)\) then \(d_y = \frac{\partial f}{\partial a}(a_0, b_0, c_0)\)

\[
\begin{align*}
\text{t1} &= a \cdot b \\
d_{\text{t1}} &= d_a \cdot b + 0 \cdot a \\
p1 &= \cos(t1) \\
t2 &= \sin(t1) \\
d_{\text{t2}} &= d_{\text{t1}} \cdot p1 \\
y &= t2 \cdot c \\
d_y &= d_{\text{t2}} \cdot c + 0 \cdot t2
\end{align*}
\]

- 3 directions give \(\nabla f(a_0, b_0, c_0)\) and \(d_y = \nabla f^T(\dot{a}, \dot{b}, \dot{c}) = \nabla f^T \dot{x}\)

- floating point accuracy for derivative calculation!
- gradient calculation cost \(\sim n\)
reverse with adjoints

Assume variable and adjoints associated in pairs \((v, g_v)\):

append computations of adjoints

\[
\begin{align*}
t1 &= a*b \\
p1 &= \cos(t1) & \text{// push}(p1) \\
t2 &= \sin(t1) \\
y &= t2*c \\
g_c &= g_y*t2 \\
g_t2 &= g_y*c \\
g_t1 &= g_t2*p1 & \text{// pop()} \\
g_b &= g_t1*a \\
g_a &= g_t1*b
\end{align*}
\]

What is in \((g_a, g_b, g_c)\)? If \(g_y=1\), then \(\nabla f(a_0, b_0, c_0)\)!

notice the lifetime of \(p1\) ⇒ insert stack operations
transformation elements so far ...

√ data augmentation
√ linearized model - by intrinsic
√ accumulate derivatives - given the computational graph(s)
√ stack for certain values used in the adjoint computation

control flow / subroutine calls?

• graph sequence
  – elimination in graphs to bipartite ≈ preaccumulation of local Jacobians
  – propagation of Jacobians ≈ chained sparse matrix product

• explains the necessity of the value stack

• transformations for adjoint code cover control flow & call sequence reversal

Problem: value stack size ~ problem size & runtime
⇒ trade-off stack size (memory) for storing checkpoints (less memory) and recomputations from checkpoints (extra runtime)
trade memory consumption for recomputation

- checkpoint placement
- determine checkpoint contents using side effect analysis
- hierarchal checkpoints
- estimating checkpoint size vs. tape size reductions
- control irregular checkpointing schemes

6 checkpoints
forward with tape; length=1/7
reverse

code execution
variations for the source transformation (1)

variations are hierarchical, starting with the lowest level
the elimination order in the computational graph

- flop counts
- optimal solution known for single-expression-use graphs
- np-hard \(\Rightarrow\) comparing various heuristics

\((*,+)^*\)

<table>
<thead>
<tr>
<th>Test</th>
<th>Default</th>
<th>Vertex</th>
<th>Face</th>
<th>Switching</th>
<th>Savings</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RoehFlux</td>
<td>1469, 370</td>
<td>1245, 170</td>
<td>1400, 168</td>
<td>1217, 181</td>
<td>17.2%, 51.1%,</td>
<td>4.5%</td>
</tr>
<tr>
<td>RoehFlux_MO</td>
<td>2121, 1088</td>
<td>1175, 415</td>
<td>1495, 461</td>
<td>1175, 415</td>
<td>44.6%, 61.9%,</td>
<td>29.4%</td>
</tr>
<tr>
<td>mini1</td>
<td>450, 199</td>
<td>410, 91</td>
<td>383, 88</td>
<td>383, 88</td>
<td>14.9%, 55.8%</td>
<td></td>
</tr>
<tr>
<td>todd1</td>
<td>1461, 738</td>
<td>909, 254</td>
<td>1182, 316</td>
<td>909, 254</td>
<td>37.8%, 65.6%,</td>
<td>19.7%</td>
</tr>
</tbody>
</table>

- for smaller basic blocks and problem significant savings still occur though
  the largest savings are found in the largest basic blocks
example computational graph
next higher level is the scope of the individual computational graphs

- smaller scope ⇒ fewer flops for elimination within graphs
- smaller scope ⇒ more graphs in sequence, suspect more flops for propagation
- varying scope ⇒ varying number of nonzeros in sparse local Jacobian
- two practical choices, statement level (known optimal elimination) and maximal graphs

<table>
<thead>
<tr>
<th>test</th>
<th>basicblock ??</th>
<th>statement ??</th>
<th>switching ??</th>
<th>Δ time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*  + J_{ij}</td>
<td>*  + J_{ij}</td>
<td>*  + J_{ij}</td>
<td></td>
</tr>
<tr>
<td>RoehFlux</td>
<td>1217 181 615</td>
<td>308 0 278</td>
<td>310 0 277</td>
<td>28.9%</td>
</tr>
<tr>
<td>dfdcfj</td>
<td>103 5 34</td>
<td>91 5 41</td>
<td>103 5 34</td>
<td>-48.1%</td>
</tr>
<tr>
<td>todd1</td>
<td>909 254 280</td>
<td>75 1 165</td>
<td>75 1 165</td>
<td>-1.6%</td>
</tr>
</tbody>
</table>

- there can be small advantages to switching at the block level
- big advantage is that the better of statement level or block is picked without user effort
variations for the source transformation (3)

next higher level is the checkpoint placement for an adjoint code

• lower level choices determine value stack growth rate
• checkpoints close enough to keep stack size limited
• checkpoint size

transformation-time information by itself insufficient, need profile data because

• problem size impacts checkpoint size
• loop iteration counts impact stack size
• code generation inserts optional profiling code

checkpoint related transformation data is still useful.
checkpoint information

SubroutineName tape double:integer checkpoint double:integer:boolean

box_model_body 0:3752 0:0:0

box_final_state 0:2 6:0:0

box_forward 0:4 41:1:0

box_ini_fields 12:63 20:0:0

box_cycle_fields 0:25 12:0:0

box_robert_filter 12:25 10:0:0

box_timestep 11:1 19:0:0

box_transport 3:0 5:0:0

box_density 6:13 8:0:0

box_update 6:13 7:0:0

SubroutineName tape double:integer checkpoint double:integer:boolean
• **method** split

• **outer** and **inner** checkpoints

• **data** visibility
conclusions

- variations useful in practice
  (implemented in OpenAD, see www.mcs.anl.gov/openad)

- automatic heuristics consistent with runtime results

- transformation complexity is increasing
  - variants have a huge domain
  - transformation environment stays close to compiler IR
  - few abstractions possible at a higher level
  - e.g. the checkpointing scheme on the dynamic call tree

- gaining some insight into profile feedback to transformations (with user-hints)

- Can the AD niche tools and the general purpose STSs meet somewhere?
AD community

- most active: Germany (Aachen, Berlin, Dresden, Hamburg), US (Argonne, Rice U, MSU, Sandia), UK (Cranfield, RAL, Hatfield), France (INRIA)
- connections to application areas (engineering, oceanography, meteorology), numerical optimization, compiler research
- not many connections to “other” source transformation fields
- common problems: stable and up-to-date parsing/unparsing environments, advanced compiler analyses
- high level IR, but want e.g. type analysis
- semantic enhancements
- AD community has two informal 2 day workshops per year (next one is Dec 7/8 in Aachen), various workshops/minisymposia attached to conferences, the 5th International AD conference in 2008.
- community website: www.autodiff.org