Some highlights on Source-to-Source Adjoint AD

Laurent Hascoët

Université Côte d'Azur, INRIA, France

NIPS 2017, Long Beach, California, USA

Workshop:
Future of gradient-based ML software & techniques

Motivation

Adjoint derivatives by Algorithmic Differentiation (AD):

- compute gradients of numerical models,
- from the models source program,
- more or less automatically,
- at a cost independant of #inputs,
- ...but facing serious challenges.

ML Back Propagation shares issues with adjoint AD.
$\Rightarrow$ Can we share a few solutions ?

Challenges of adjoint AD

Gradients are propagated backwards,
using info from the (forward) primal code
$\Rightarrow$ Instruction flow reversal
$\Rightarrow$ Data flow reversal

For the record, there are other challenges:

- non-smoothness *[Griewank et al.]*
- stochastic or chaotic parts *[Wang]*
- higher derivatives (cost, size...) *[Walther, Wang, Pothen]*
- . . .

AD models

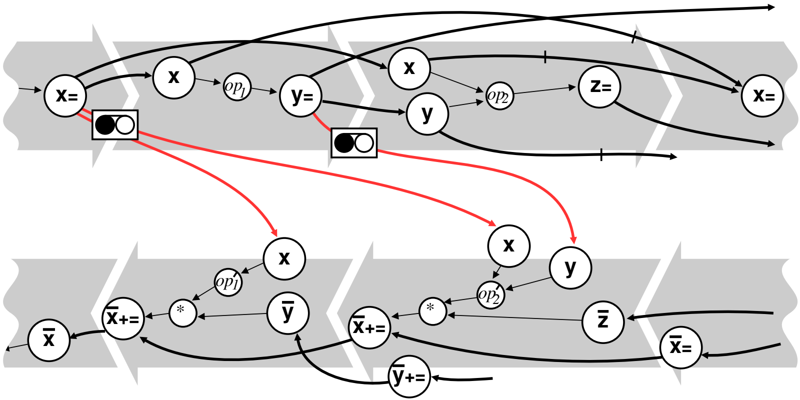
Different AD models ($\Rightarrow$ different AD tools) explore different strategies:

- Instruction flow reversal
  - $\Rightarrow$ either by storing runtime trace
  - $\Rightarrow$ or by writing a new source
- Data-Flow reversal
  - $\Rightarrow$ either by storing values or partials
  - $\Rightarrow$ or by recomputing them

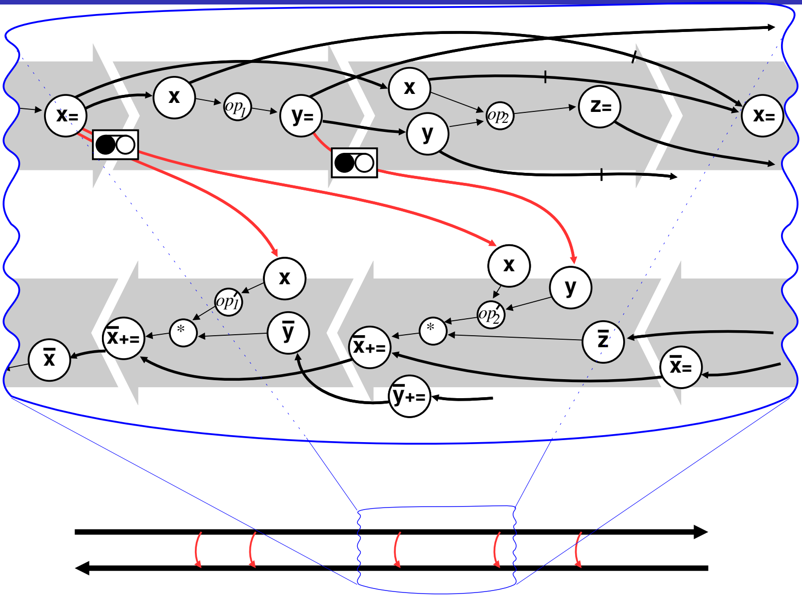
Our directions:

write a new source (Source-to-Source),
store intermediate values (Store-All) upon value kill

Source-to-Source adjoint AD, Store-All



Source-to-Source adjoint AD, Store-All



The memory challenge

We are happy not to store the runtime instruction trace,
– but we still need to store the intermediate values,
– at a memory cost that grows linearly with runtime.

Can we master memory consumption ?

- use every possible Data-Flow analysis
  $\rightarrow$ can gain 40 to 70%... still linear memory cost
- trade recomputation/storage ("Checkpointing")
  $\rightarrow$ achieves logarithmic growth
- exploit profitable situations, (math or algorithm) e.g.
  - Linear solvers
  - Parallel loops
  - Fixed-Point iterations

Outline

1. Data-Flow Analysis
2. Checkpointing
3. Profitable Situations

Data-Flow Analysis

Naïve application of the adjoint AD model would

- execute all primal instructions
- store every value before it is overwritten
- execute the complete adjoint of each instruction

Forward constant propagation & backward slicing, specialized for the particular structure of adjoint codes

Use static data-flow analysis (classic + and −), on the primal code, then produce an optimized adjoint code

4 classic AD Data-Flow analyses

- **varied**: *[Fagan, Carle]*

  if current v depends on no "independent input", then $\overline{\mathrm{v}}$ is useless

  $\Rightarrow$ slice out computation of $\overline{\mathrm{v}}$

- **useful**:

  if current v influences no "dependent output", then $\overline{\mathrm{v}}$ is zero

  $\Rightarrow$ propagate constant $\overline{\mathrm{v}}$ and remove its initialization

- **diff-live**:

  if current v influences no useful derivative (may influence orig. result)

  $\Rightarrow$ slice out computation of v

- **TBR**: *[Naumann]*

  if current v not used in any derivative (e.g. only linear uses of v)

  $\Rightarrow$ slice out storage of v before it is overwritten

Discussion

- These are just special cases of classic code optim.
- Agressive compiler optim *[Pearlmutter, Siskind]* may be more systematic ($\Rightarrow$ are we missing adjoint data-flow analyses?)
- ... but there's a limit to the window of code that the compiler can examine, whereas fwd and bwd code are arbitrarily far apart
- Adjoint data-flow analyses use structural knowledge of adjoint codes, and run on the primal code. E.g.

$$\mathbf{TBR}^{+}(I) = \begin{cases} (\mathbf{TBR}^{-}(I) \cup \mathbf{use}(I')) \setminus \mathbf{kill}(I) & \text{if } I \text{ live} \\ \mathbf{TBR}^{-}(I) \cup \mathbf{use}(I') & \text{otherwise} \end{cases}$$

Summary: good, but not sufficient

Adjoint data-flow analyses

- are classical compiler analyses/optims specialized for adjoint codes.
- bring substantial benefit
  - 20% to 50% in runtime
  - 40% to 70% in memory space

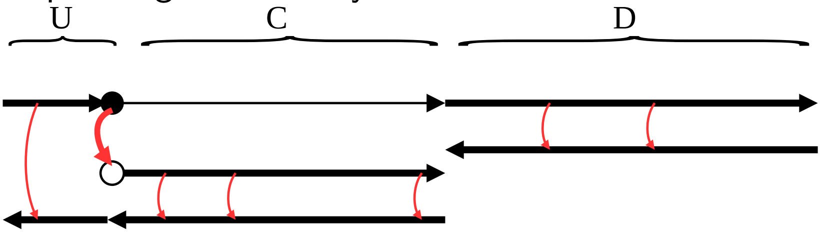
But memory still grows linearly with runtime
$\Rightarrow$ we need something else.

Outline

1. Data-Flow Analysis
2. Checkpointing
3. Profitable Situations

Trading recomputation (time) for storage (memory)

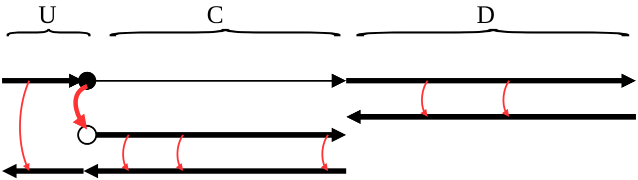
Checkpointing: elementary stitch



- reduces peak storage
- at the cost of duplicate execution
- also costs a memory "Snapshot", small enough:

$$\text{Snapshot} \subset \textbf{use}(\overline{\texttt{C}}) \cap \left(\textbf{out}(\texttt{C}) \cup \textbf{out}(\overline{\texttt{D}})\right)$$

Combining Checkpointing and TBR



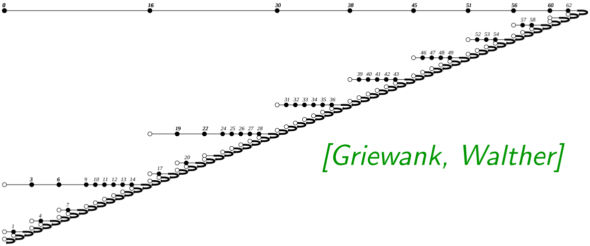
- The Snapshot may take care of TBR coming from U
- The TBR sent to D can take care of the Snapshot

A range of "optimal" combinations exist.
E.g., given **tbr**U coming from U, "lazy" snapshot:

- Snapshot $= \mathbf{out}(\mathtt{C}) \cap (\mathbf{use}(\overline{\mathtt{C}}) \cup \mathbf{tbr}\mathtt{U})$
- **tbr** to D $= (\mathbf{use}(\overline{\mathtt{C}}) \cup \mathbf{tbr}\mathtt{U}) \setminus \mathbf{out}(\mathtt{C})$
- **tbr** to C $= \mathbf{tbr}\mathtt{U}$

Nesting checkpoints

Checkpoints must be (carefully) nested.
Optimal nesting (binomial) exists for time-stepping loops:

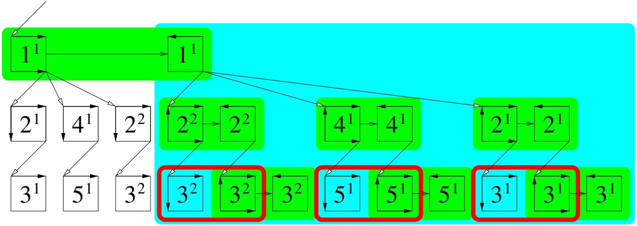


[Griewank, Walther]

- peak memory storage grows like log(runtime)
  execution duplication grows like log(runtime)
- in real life, storage is fixed to *q* snapshots,
  execution duplication grows like *q*th-root(runtime)

Checkpointing on calls

Nested checkpointing can be applied on procedure calls:



Sub-optimal(?), but still logarithmic if call tree is balanced.

Applies also to code sections that *could* be procedures.

A few limitations

- Checkpoints must respect code structure:
  - no checkpoint across procedures
  - no checkpoint across structured statements
  - *...well you could, but you need a flattened instruction tape*
- Checkpoints must contain both ends of system resources lifespan:

  `read/write`, `alloc/free`, `send/recv`, `isend/wait`...
- Checkpointed code must be reentrant

Outline

Profitable Situations

Take advantage of algorithmic or mathematic knowledge on parts of the code.

Just a selection:

- Adjoint of Linear Solvers
- Adjoint of Parallel Loops
- Adjoint of Fixed-Point iterations

Adjoint of Linear Solvers

Avoid differentiation inside the source of linear solvers

⇒ write their adjoint by hand, calling the solver itself!

```
SOLVE_B(A,Ab,y,yb,b,bb) {
  At = TRANSPOSE(A)
  SOLVE(At,tmp,yb)
  bb[:]  = bb[:]  + tmp[:]
  SOLVE(A,y,b)
  for each i and each j {
    Ab[i,j] = Ab[i,j] - y[j]*tmp[i]
  }
  yb[:]  = 0.0
}
```

[Giles]

Data-Dependence Graph of Adjoints

Data-Dependence Graph is key to loop rescheduling.
Fewer arrows in the DDG $\Rightarrow$ more rescheduling allowed.

- (classical) No DDG arrow between successive `reads` of a variable.
- No DDG arrow either between successive `increments` of a variable. (assuming increments are atomic, or that memory is not shared)
- The adjoint of a `read(x)` is an `increment(`$\bar{x}$`)`
- The adjoint of an `increment(x)` is a `read(`$\bar{x}$`)`

> The DDG of the backward sweep is a subset of the DDG of the primal code, only with arrows reversed

Therefore adjoint AD preserves most parallel properties!

Application to Parallel Loops

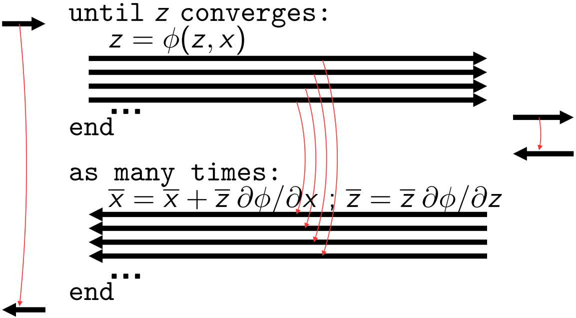
```
// Parallel loop:
for (i=0 ; i<=N ; ++i) {
  forward sweep iteration i
}
for (i=N ; i>=0 ; --i) {
  backward sweep iteration i
}
```

Loop #2 is parallel: reverse iterations, fuse with loop #1:

```
for (i=0 ; i<=N ; ++i) {
  forward sweep iteration i
  backward sweep iteration i
}
```

⇒ Reduces peak memory usage dramatically!

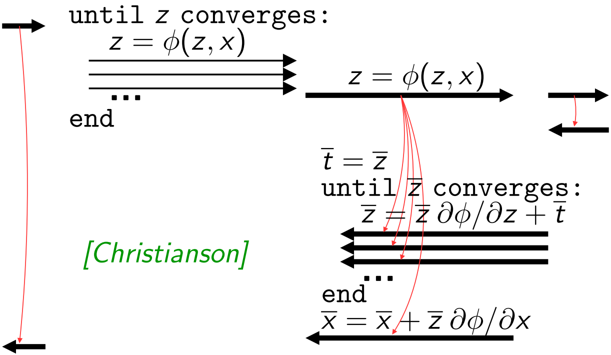
Adjoint of Fixed-Point iterations



You should not do that!

- all values from intermediate iterations are stored
- poor convergence guarantees of the adjoint sweep

Two-Phases Adjoint



- Only the converged primal iteration is stored, then is used several times.
- The adjoint iteration has its own convergence control
- Converges in one step if primal has quadratic convergence

Loosing Warm-Start

Suppose Fixed-Point is included in another iteration:

FP iterations: 16, 16, 16, 16, ...

Warm-Start uses previous converged z as next initial z.
$\Rightarrow$ convergence is reached earlier

FP iterations: 16, 9, 9, 9, ...

Standard adjoint "inherits" this Warm-Start effect
but the Two-Phases adjoint doesn't!

adjoint FP iterations: 46, 46, 46, 46, ...

Retrieving Warm-Start?

Two-Phases adjoint modifies $\overline{z}$ between two adjoint Fixed-Points.
Notice that $\overline{z}$ has two uses:

...
$\overline{t} = \overline{z}$
until $\overline{z}$ converges {
$\quad \overline{z} = \overline{z}\, \partial\phi/\partial z + \overline{t}$
}
$\overline{x} = \overline{x} + \overline{z}\, \partial\phi/\partial x$
modifications (reset!) on $\overline{z}$
...

(1) to set $\overline{t}$, that influences converged value ⇒ don't touch this!
(2) as the initial guess of the adjoint Fixed-Point ⇒ feel free to set it to previously converged!

adjoint FP iterations: 46, 20, 21, 20, . . .

Is that correct? Can we automate it?

Thank you for your attention!