Memory Requirements

Laurent Hascoët
Paul Hovland
Jan Hückelheim
Sri Hari Krishna Narayanan

SIAM AN24 Student Days

Memory requirements of AD

Computing derivatives increases the program memory footprint. We must store:

- the intermediate and final derivatives
- *(depending on AD mode)* intermediate values, partial derivatives, sequence of primal operations, trace of control flow...

Alternatives to AD (continuous adjoints, FD) face similar issues.
A challenge especially for reverse AD (cheap gradients, but "no free lunch").

Overloading AD

Overloading AD is traditional name for taping a trace of the full forward run, e.g. for each *run-time* statement `x := a Op b`

- `[long int]`: an index representing `a` (and `a'`),
- `[long int]`: an index representing `b` (and `b'`),
- `[short int]`: a code for `Op`,
- `[double]`: the value of `x` *before* the statement.

Tape grows linearly with execution time.
Tape contains everything $\Rightarrow$ no backward sweep code, only a conceptually simple tape interpreter.

Overloading AD is probably the most memory-consuming AD.

Source-Transformation AD

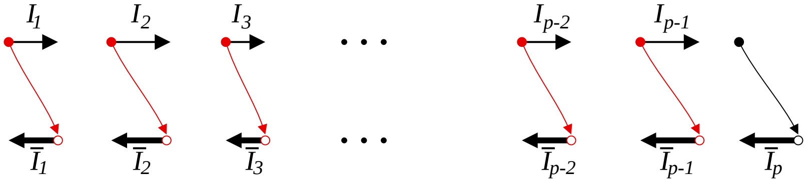
ST-AD generates a new source program P',
replacing a good part of the tape with source in P'.

- No indices stored, nor op-codes:
  only new variables `a'`, `b'`, `x'`
- Stack memory footprint of P (only) doubled
- End-user *and* compiler can take a look at P'

Reverse AD still tapes values for data-flow reversal
$\Rightarrow$ tape is smaller, still grows linear with run-time.
What can we do?

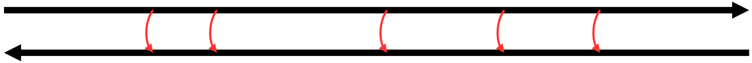
Memory requirements of Data-flow reversal

Recall the concept of Data-flow reversal:



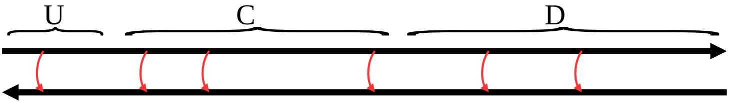
Assume red-arrow magic done by storing on a stack

Step back a little:

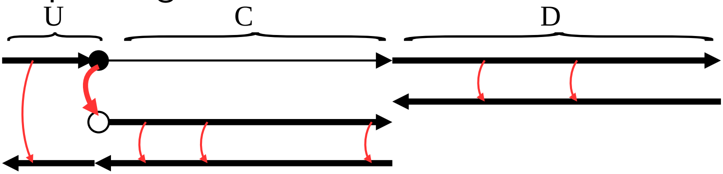


Stack grows like run-time, reaching maximum at turn point (on the right).

Checkpointing: trading recomputation for storage



Checkpointing C is:

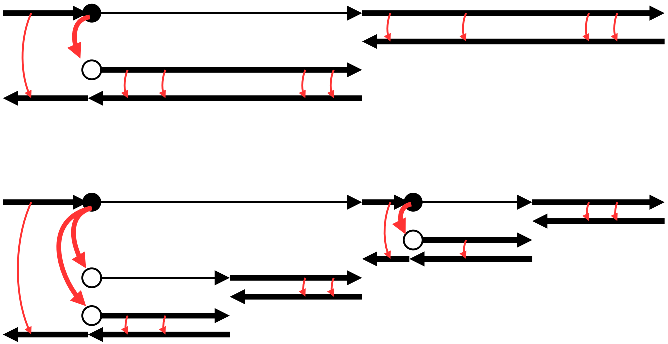


- reduces peak storage
- at the cost of duplicate execution
- also costs a memory "Snapshot", small enough:

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

Nesting Checkpoints

On large codes, checkpoints can/must be nested.



On a well-balanced nesting (e.g. a well-balanced call tree), memory and CPU grow like *log*(runtime)

Technical constraints

- start/end of `C` in the same procedure
- start/end of `C` in the same control

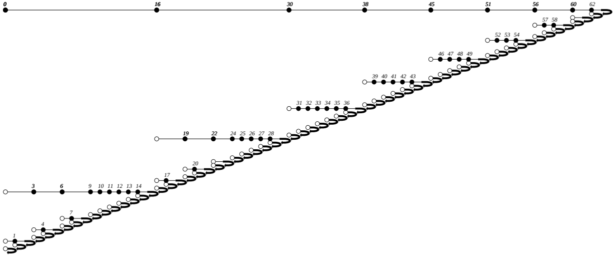
Also, `C` must be reentrant, i.e. one can restore exactly its initial state:

- if `C` contains a `malloc`, it must contain its `free`
- if `C` contains a `send`, it must contain its `recv`
- if `C` contains a `isend`/`irecv`, it must contain its `wait`
- if `C` contains a `open`, it must contain its `close`

Most often `C`'s are procedures, time steps ...

Checkpointing on Time-stepping loops

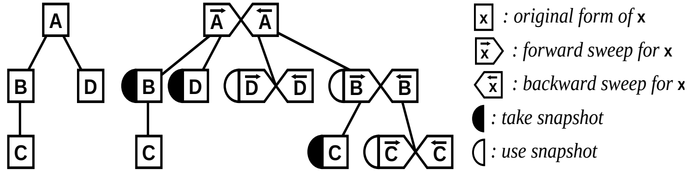
Binomial checkpointing nesting is optimal for time-stepping loops (uniform step cost, depend on previous step only, negligible snapshot time):



- peak memory storage and duplicate execution still grow like *log*(runtime), but proved optimal.
- in real life, storage is fixed to q snapshots, execution duplication grows like $\sqrt[q]{\text{runtime}}$

Checkpointing on the Call Tree

The Call Tree is the natural support to define your checkpointing strategy:



- Costs still grow like *log*(runtime) if call tree well balanced.
- Ill-balanced call trees require not checkpointing some calls.
- Small leaf procedures better not checkpointed.

Checkpointing on the Call Tree

foo not checkpointed (aka Split)	foo checkpointed (aka Joint)

...
  $\overrightarrow{\mathtt{foo}}\mathtt{(a)}$
...
  $\overleftarrow{\mathtt{foo}}\mathtt{(a,\ \bar{a})}$
...
 	
$\overrightarrow{\mathtt{foo}}\mathtt{(x)}$
  push x
  x = sin(x)
  push x
  x = x*x
}
$\overleftarrow{\mathtt{foo}}\mathtt{(x,\ \bar{x})}$
  pop x
  $\bar{\mathtt{x}}$ = 2*x*$\bar{\mathtt{x}}$
  pop x
  $\bar{\mathtt{x}}$ = cos(x)*$\bar{\mathtt{x}}$
}
	...
	  push a
	  foo(a)
	...
	  pop a
	  $\overline{\mathtt{foo}}\mathtt{(a,\ \bar{a})}$
	...
	
	$\overline{\mathtt{foo}}\mathtt{(x,\ \bar{x})}$
	  push x
	  x = sin(x)
	  $\bar{\mathtt{x}}$ = 2*x*$\bar{\mathtt{x}}$
	  pop x
	  $\bar{\mathtt{x}}$ = cos(x)*$\bar{\mathtt{x}}$
	}

Advanced research on Data-flow reversal

- combine storage of intermediates/partials, and inversion / forward recalculation.
- periodic checkpointing
- binary checkpointing
- optimal strategies if non-zero snapshot time
- combination with resilience checkpoints