Checkpointing

UCSD CSE 291
Tzu-Mao Li
The memory cost of reverse mode AD

for i in range(n):
    x = sin(x) → ?
The memory cost of reverse mode AD

for i in range(n):
    x = sin(x)

for i in range(n):
    x = stack.pop()
    dx = dx * cos(x)

time complexity: O(n)
space complexity: ?

time complexity: O(n)
space complexity: ?
The memory cost of reverse mode AD

```
for i in range(n):
    x = sin(x)
```

time complexity: $O(n)$
space complexity: $O(1)$

```
for i in range(n):
    x = stack.pop()
    dx = dx * cos(x)
```

time complexity: $O(n)$
space complexity: $O(n)$

reverse mode preserves time complexity, but doesn’t preserve memory complexity!
Reverse mode’s memory cost can be prohibitive in practice

Imagine if \( x \) is an array with many elements here, then when \( n \) is large, we can quickly run out of RAM.

```python
for i in range(n):
    stack.push(x)
    x = \sin(x)

for i in range(n):
    x = stack.pop()
    dx = dx * \cos(x)
```

How can we reduce the memory usage for this case?
Reducing memory cost by recomputation

```python
for i in range(n):
    stack.push(x)
    x = sin(x)

for i in range(n):
    x = stack.pop()
    dx = dx * cos(x)
```

```python
for i in range(n):
    x = x0
    for j in range(i):
        x = sin(x)
        dx = dx * cos(x)
```
Reducing memory cost by recomputation

for i in range(n):
    stack.push(x)
    x = sin(x)

for i in range(n):
    x = stack.pop()
    dx = dx * cos(x)

time complexity: O(n)
space complexity: O(n)

now it becomes too slow :

for i in range(n):
    x = x0
    for j in range(i):
        x = sin(x)
        dx = dx * cos(x)

time complexity: O(n^2)
space complexity: O(1)
Checkpointing: exploring tradeoffs between memoization and recomputation

for i in range(n):
    stack.push(x)
    x = sin(x)

for i in range(n):
    x = stack.pop()
    dx = dx * cos(x)

for i in range(n):
    x = x0
    for j in range(i):
        x = sin(x)
        dx = dx * cos(x)

with some heuristics, for this case we can achieve

time complexity: $O(n)$
space complexity: $O(n)$

time complexity: $O(n\log(n))$
space complexity: $O(\log(n))$

time complexity: $O(n^2)$
space complexity: $O(1)$

uniform checkpointing wouldn’t achieve this!
A visualization of checkpointing

for any program, we can see the execution trace of it as a linear progression of a program counter

input

\[ x_1 = \sin(x_0) \]

\[ x_{10} = \sin(x_9) \]

\[ x_{20} = \sin(x_{19}) \]

output

"program counter"
of the primal program

idea taken from this paper (2017!)
A visualization of checkpointing

normal execution of primal program

"program counter"
of the primal program
A visualization of checkpointing

standard reverse mode, i.e., store everything

forward pass

reverse pass

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all → checkpoint

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all

execute without storing for reverse pass

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all → execute without storing for reverse pass → store all

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all

execute without storing for reverse pass

store all

reverse pass

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all → execute without storing for reverse pass → deallocate

reverse pass

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all

execute without storing for reverse pass

store all

deallocate

reverse pass

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

“program counter” of the primal program
A visualization of checkpointing

one possible checkpointing scheme

store all

execute without storing for reverse pass

deallocate

reverse pass

reverse pass

“program counter” of the primal program
A visualization of checkpointing

what is the peak memory use of the previous checkpointing scheme?

“program counter” of the primal program
A visualization of checkpointing

what is the peak memory use of the previous checkpointing scheme?

\[ \max(a + b, a + c) \]

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

store all

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

store all

execute without storing for reverse pass

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

"program counter" of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

“program counter” of the primal program
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

“program counter” of the primal program
“program counter” of the primal program

store all

execute without storing for reverse pass

deallocate

reverse pass

store all

deallocate

deallocate

we can recursively apply checkpointing to allow even more flexibility

Multilevel checkpointing
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

peak memory use: $a + b + \max(c, d, e)$

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree
Bisection-based Checkpointing

[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

first, place a checkpoint at the center

execute without storing for reverse pass

“program counter” of the primal program
Bisection-based Checkpointing
[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

execute without storing for reverse pass

next, recursively put checkpoints in the subdivided regions

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

“program counter” of the primal program
Bisection-based Checkpointing
[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

execute without storing for reverse pass

start propagating derivatives from the end

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

peak memory use: number of checkpoints: $O(\log(n))$

execute without storing for reverse pass

“program counter” of the primal program
Bisection-based Checkpointing

[Griewank 1992]

peak memory use: number of checkpoints: $O(\log(n))$

time complexity: sum of green lines + $O(n) = O(n\log(n))$

*assuming the memory/time use each step is $O(1)$

execute without storing for reverse pass

“program counter” of the primal program
Binomial-based Checkpointing

[Griewank 1992]

typically does better than bisection, but needs more parameters

parameters:
c: number of checkpoints
l: number of steps in primal execution

Theorem: assuming each step takes exactly the same computation time, the minimal number of step for reverse mode compute for storing c checkpoints of a length l primal computation is

\[ t(c, l) = rl - \beta(c + 1, r - 1) \]

where \( \beta(a, b) = \binom{a}{b} \) \( \beta(c, r - 1) < l \leq \beta(c, r) \)

(r is an unique integer that represents the number of repeated computation)
Offline vs online checkpointing

both bisection and binomial checkpointing requires knowing the number of steps taken in the primal code
these are known as “offline” checkpointing algorithms

“program counter” of the primal program

this can be obtained using a “pilot run” of the primal program, so it does not need to be known at compile time
Offline vs online checkpointing

both bisection and binomial checkpointing requires knowing the number of steps taken in the primal code
these are known as “offline” checkpointing algorithms

“program counter” of the primal program

this can be obtained using a “pilot run” of the primal program, so it does not need to be known at compile time

still, there are “online” variant of checkpointing algorithms that do not require knowing the step length
Online checkpointing

idea: when the run length increases, redistribute the current checkpoints to achieve optimal binomial checkpoints
Checkpointing in deep learning

Published as a conference paper at ICLR 2021

DYNAMIC TENSOR REMATERIALIZATION

Marisa Kirisame, Steven Lyubomirsky, Altan Haan, Jennifer Brennan, Mike He, Jared Roesch, Tianqi Chen, and Zachary Tatlock

{Jerry96, sslyu, altanh, jrb, dh63, jroesch}@cs.washington.edu, tqchen@cmu.edu, ztatlock@cs.washington.edu

CHECKMATE: BREAKING THE MEMORY WALL WITH OPTIMAL TENSOR REMATERIALIZATION

Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, Joseph E. Gonzalez

online checkpointing

offline checkpointing
Checkpointing in deep learning

Dynamic Tensor Rematerialization

Marisa Kirisame; Steven Lyubomirsky; Alan Haan; Jennifer Brennan;
Mike He; Jared Roesch; Tianqi Chen; and Zachary Tatlock

{jeremy96, sslyu, altanh, jrb, dh63, jroesch}@cs.washington.edu,
tqchen@cmu.edu, ztatlock@cs.washington.edu

Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization

Paras Jain *, Ajay Jain *, Aniruddha Nrusimha
Amir Gholami; Pieter Abbeel; Kurt Keutzer; Ion Stoica; Joseph E. Gonzalez

Online checkpointing

Offline checkpointing
Checkmate: compute optimal strategy from a static data-flow graph
Checkmate: compute optimal strategy from a static data-flow graph

which function we call at each step
Checkmate: compute optimal strategy from a static data-flow graph

which function we call at each step

recompute A at step 5
Checkmate: compute optimal strategy from a static data-flow graph

The optimal compute strategy can be formulated as an integer linear programming problem:

$$\min R \sum \sum C_{i}R_{t,i}$$

subject to memory limit

C is the cost

which function we call at each step

this is the R matrix

recompute A at step 5
Optimal R matrix for VGG19
Results
Checkpointing in deep learning

Published as a conference paper at ICLR 2021

Dynamic Tensor Rematerialization

Marisa Kirisame,*† Steven Lyubomirsky,*† Altan Haan,*† Jennifer Brennan,*† Mike He,*† Jared Roesch,*† Tianqi Chen,*† and Zachary Tatlock*†
{jerry96, sslyu, altanh, jrb, dh63, jroesch}@cs.washington.edu, tqchen@cmu.edu, ztatlock@cs.washington.edu

Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization

Paras Jain,*† Ajay Jain,*† Aniruddha Nrusimha,† Amir Gholami,*† Pieter Abbeel,*† Kurt Keutzer,*† Ion Stoica,*† Joseph E. Gonzalez,*†

online checkpointing

offline checkpointing
Dynamic tensor rematerialization

idea: override array allocation/access

allocate:
  if memory budget is sufficient, allocate as usual
  if not, “evict” an array from memory

access:
  if the array has been evicted, “rematerialize” it by computing the result from parent
  if the parent is also evicted, rematerialize recursively

in paper: prove that the eviction heuristics gives space complexity \(O(\log(n))\) and time complexity \(O(n)\) for feedforward networks
Dynamic tensor rematerialization
Checkpointing in optimal control
(aka differentiable physics)

Differentiable Elastic Object Simulation (3D)

30.5K particles, 512 time steps, 40 gradient descent iter.
Run time=4min. Red=extension blue=contraction.

Goal