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):
  stack.push(x)
  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)
```

```
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(1)

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

```
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

```
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)
```

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)
```

with some heuristics, for this case we can achieve

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

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

time complexity: O(nlog(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

output

```
x1 = sin(x0)
```

```
x10 = sin(x9)
```

```
x20 = sin(x19)
```

"program counter"
of the primal program

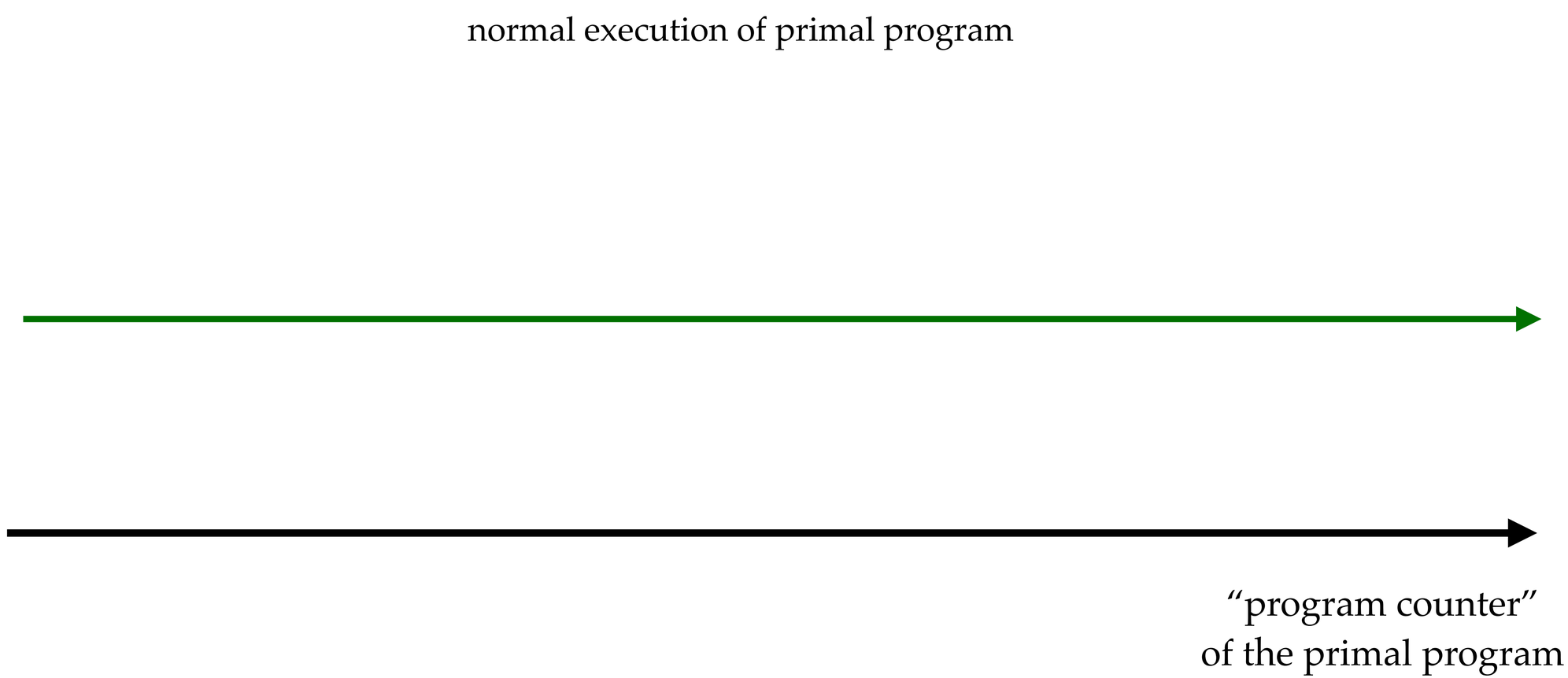
Divide-and-Conquer Checkpointing for Arbitrary Programs with No User Annotation

Jeffrey Mark Siskind[a*] and Barak A. Pearlmutter[b]

[a] *School of Electrical and Computer Engineering, Purdue University, 465 Northwestern Avenue, West Lafayette, IN, USA;* [b] *Dept of Computer Science, Maynooth University, Ireland*

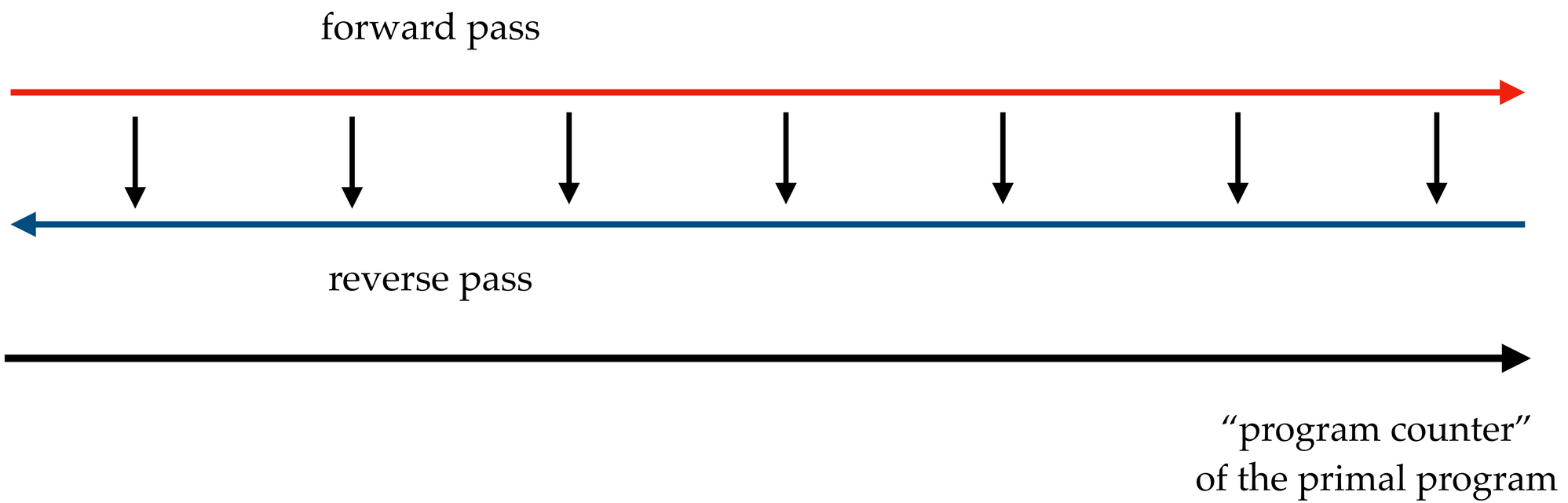
idea taken from this paper (2017!)

A visualization of checkpointing



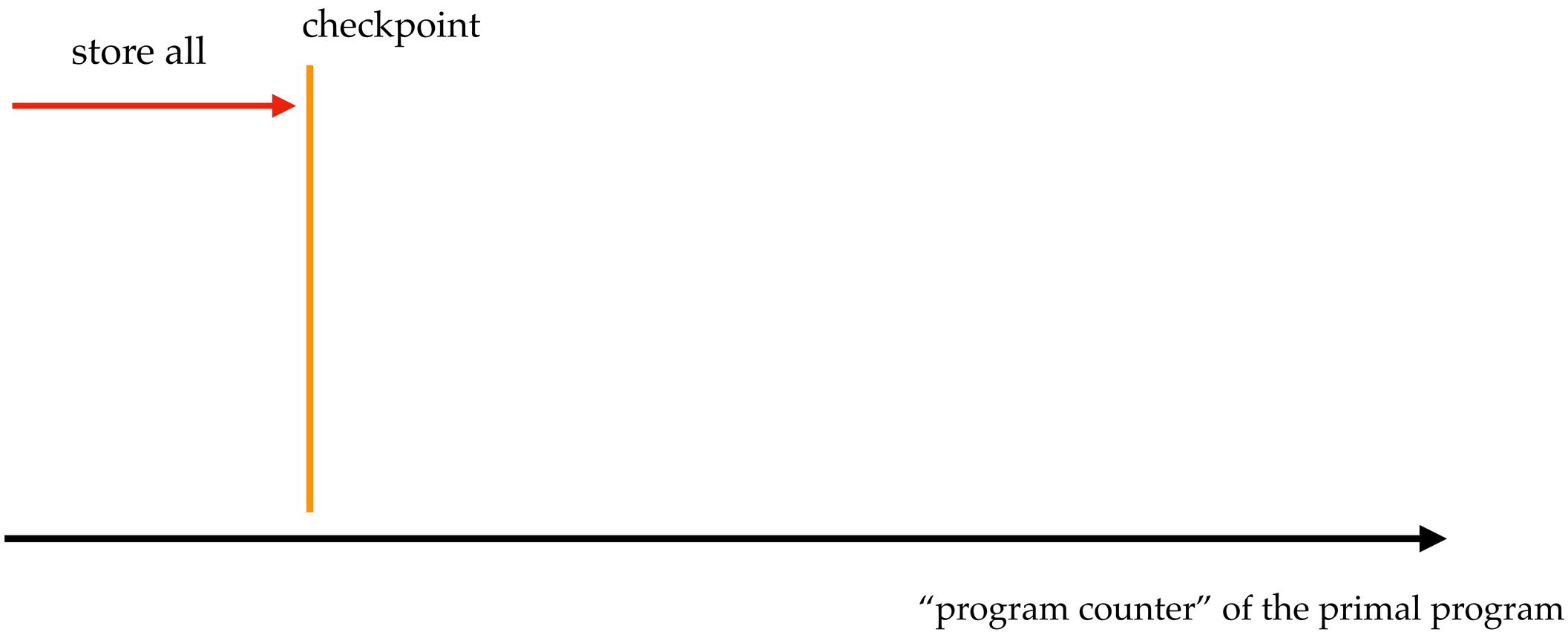
A visualization of checkpointing

standard reverse mode, i.e., store everything



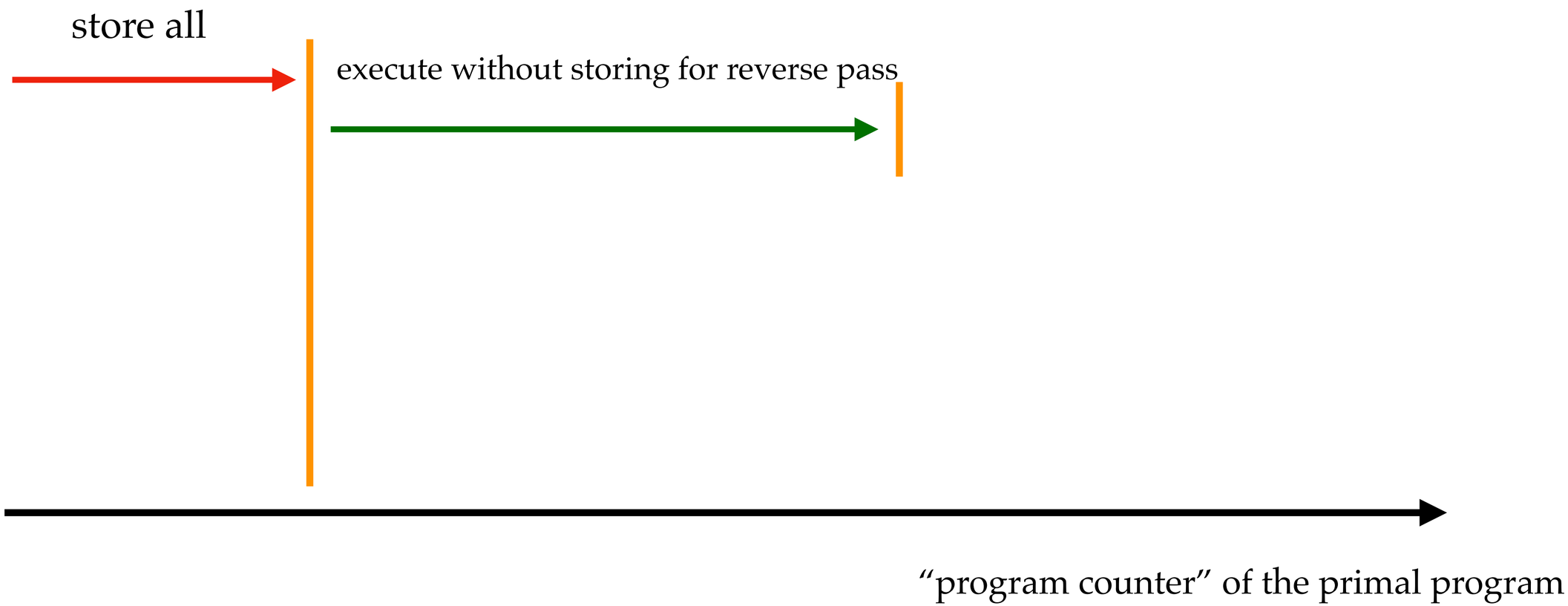
A visualization of checkpointing

one possible checkpointing scheme



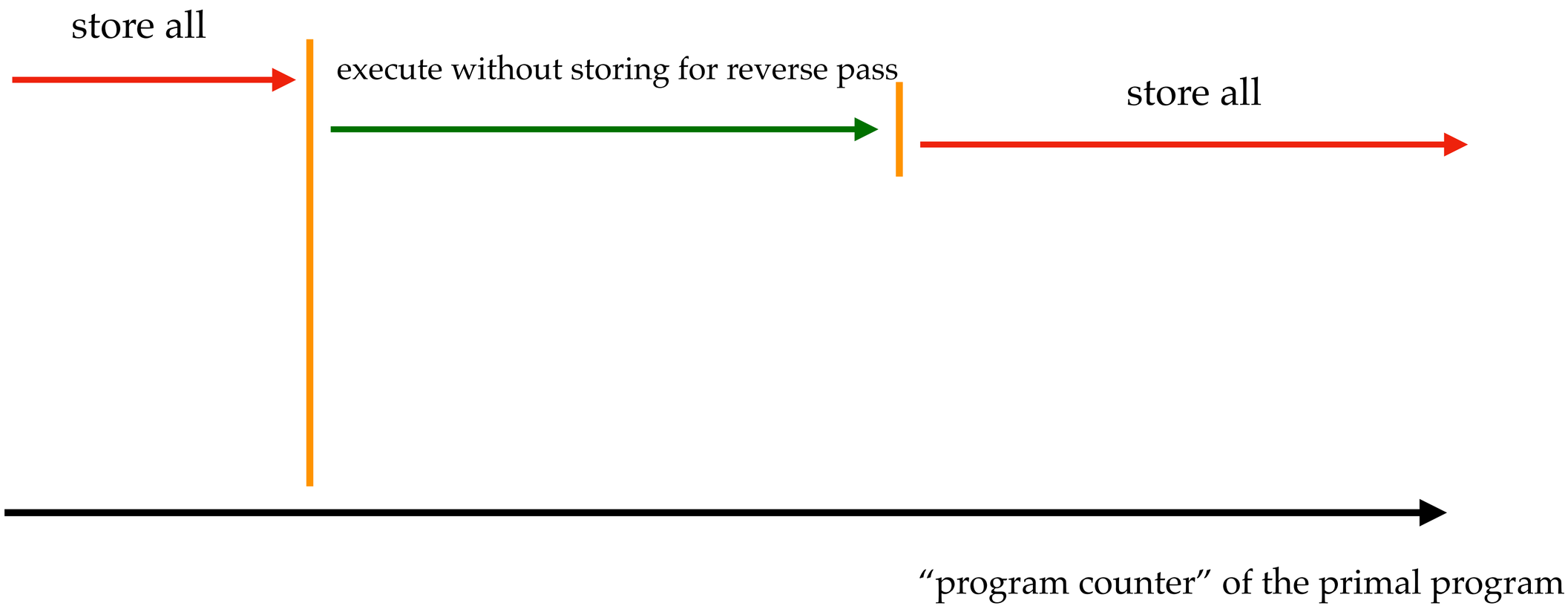
A visualization of checkpointing

one possible checkpointing scheme



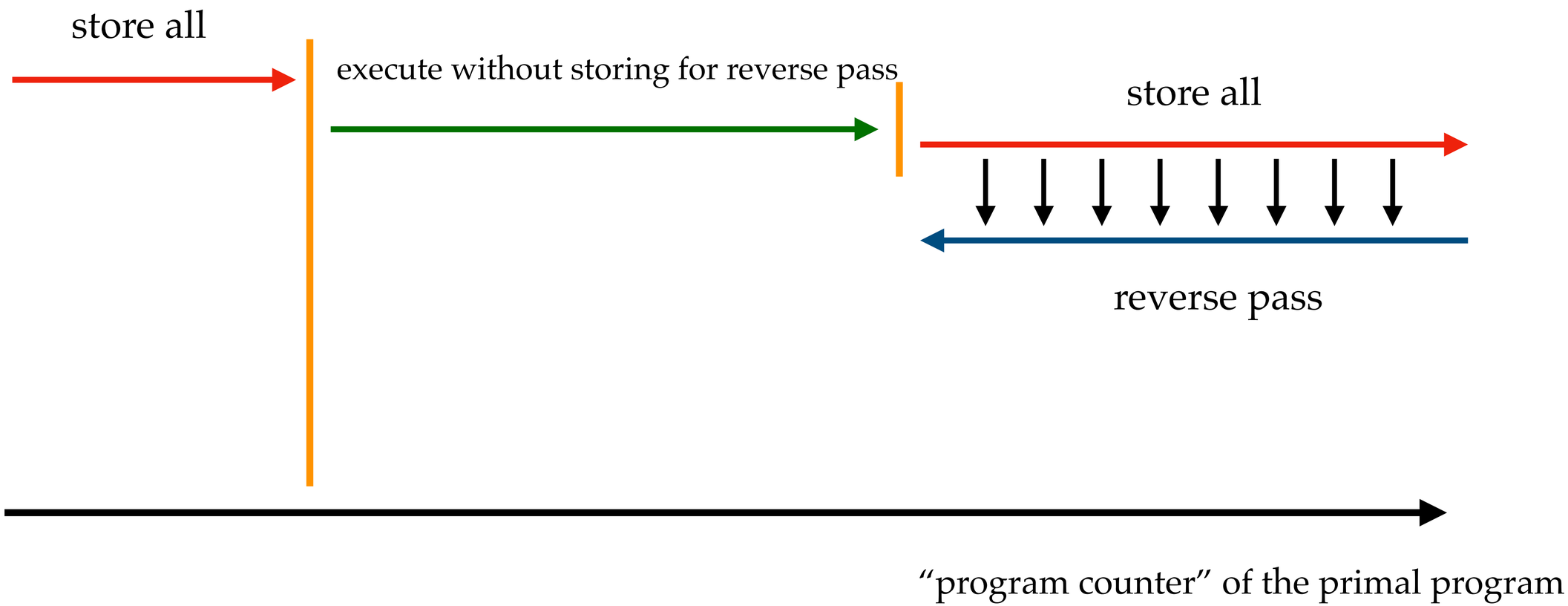
A visualization of checkpointing

one possible checkpointing scheme



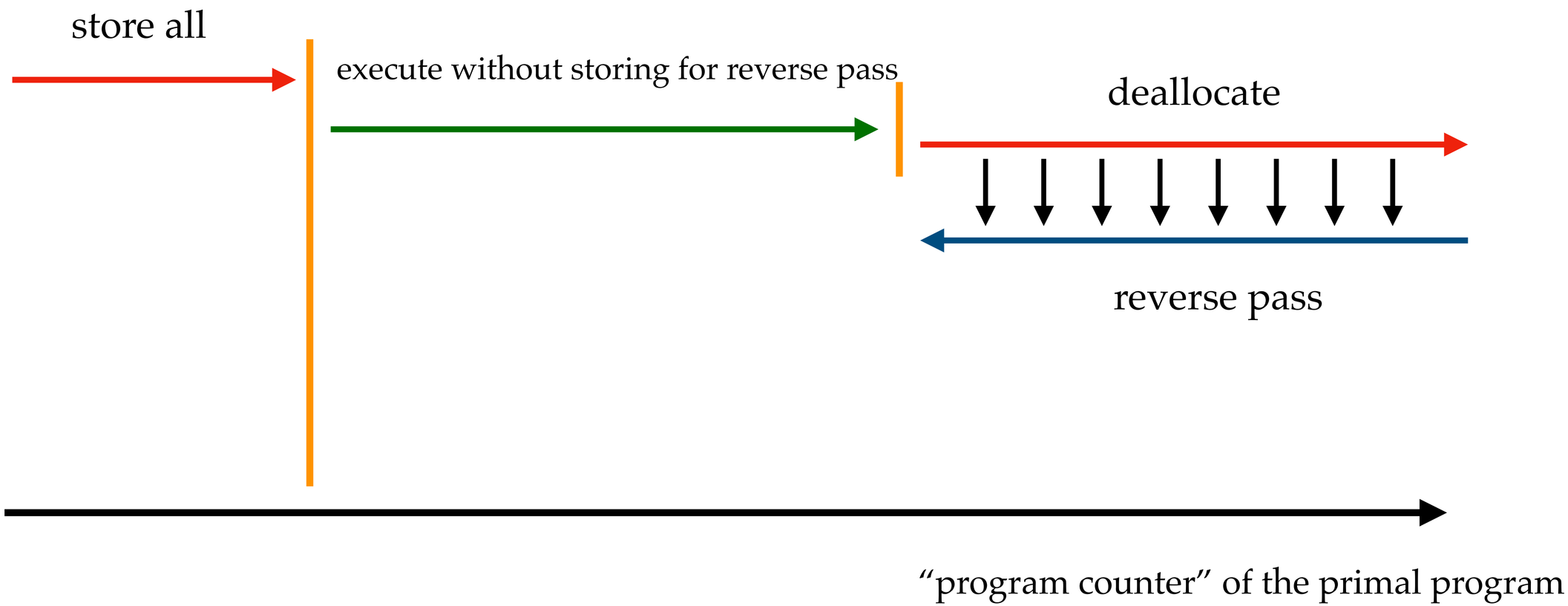
A visualization of checkpointing

one possible checkpointing scheme



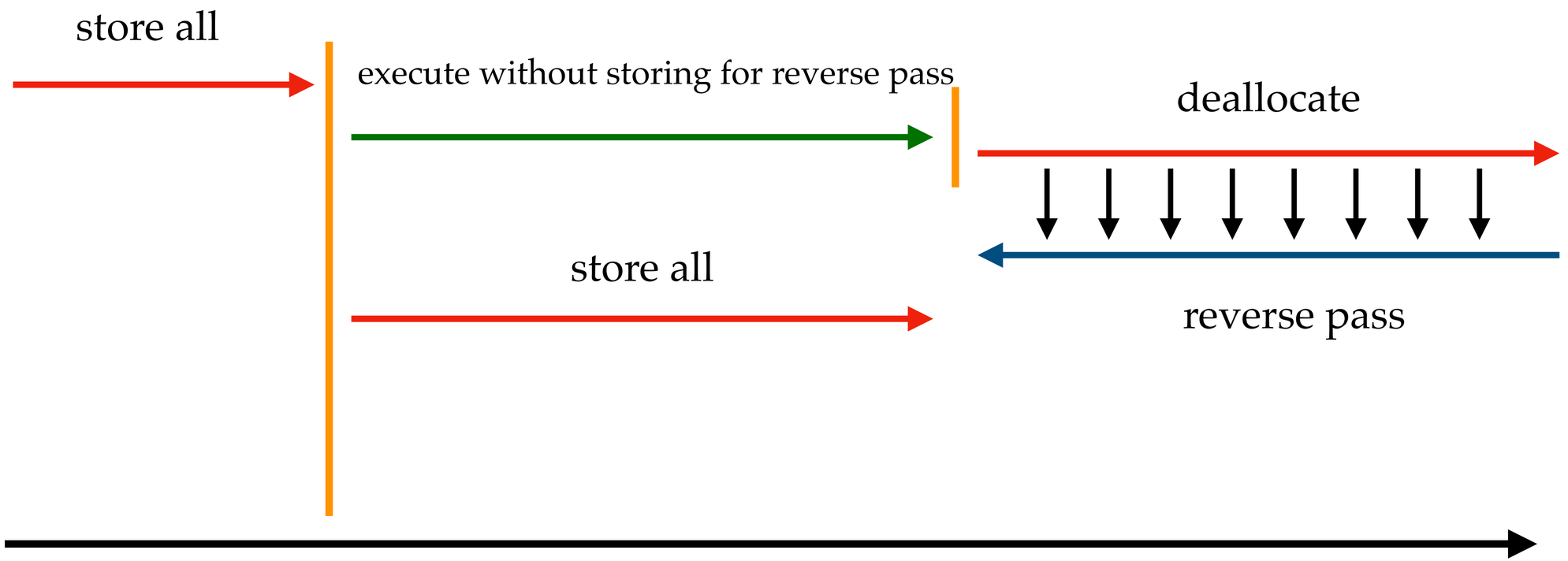
A visualization of checkpointing

one possible checkpointing scheme



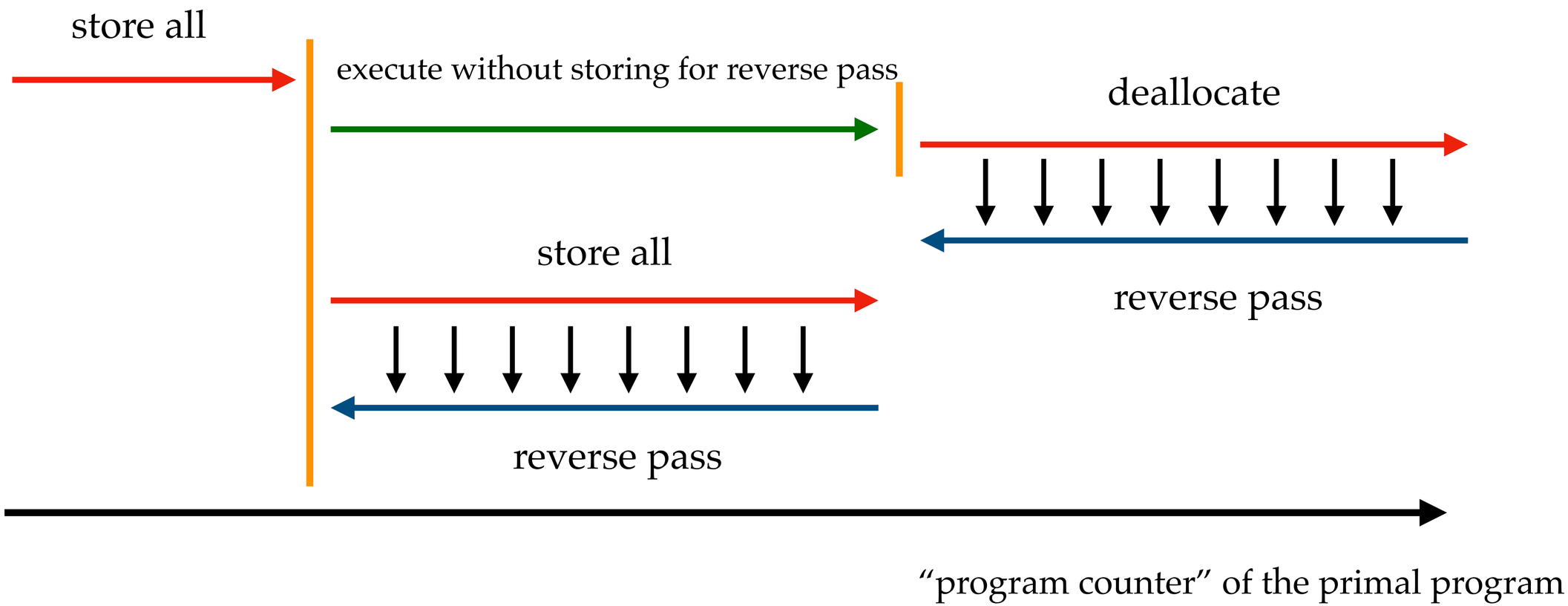
A visualization of checkpointing

one possible checkpointing scheme



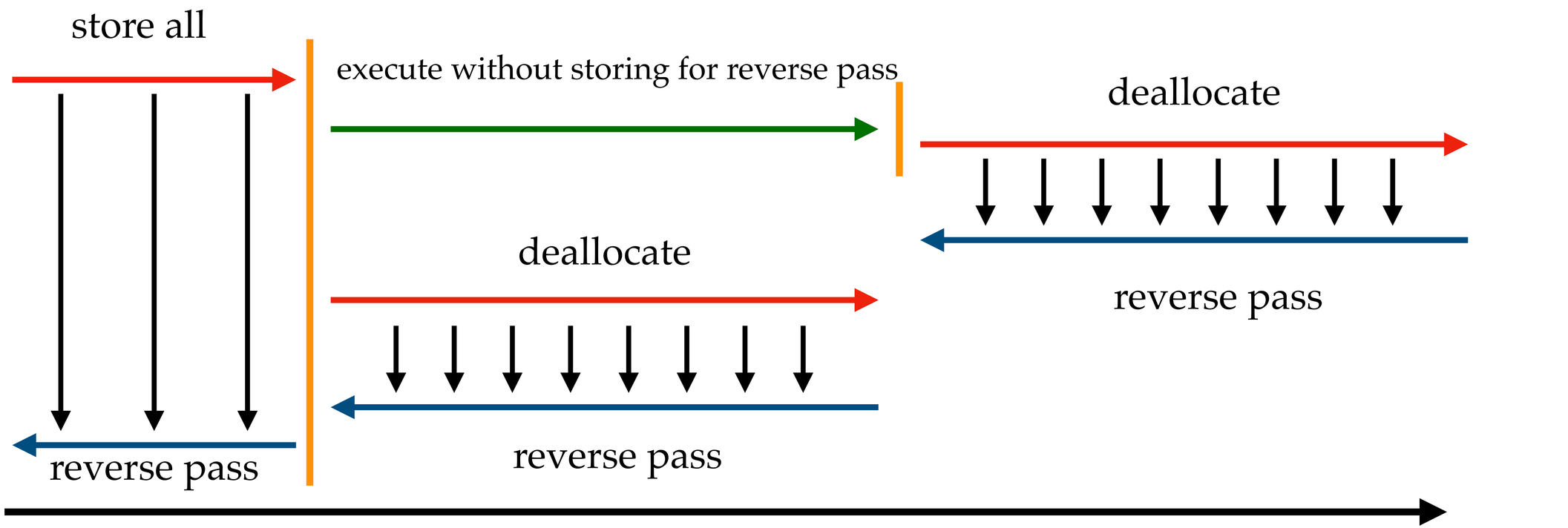
A visualization of checkpointing

one possible checkpointing scheme



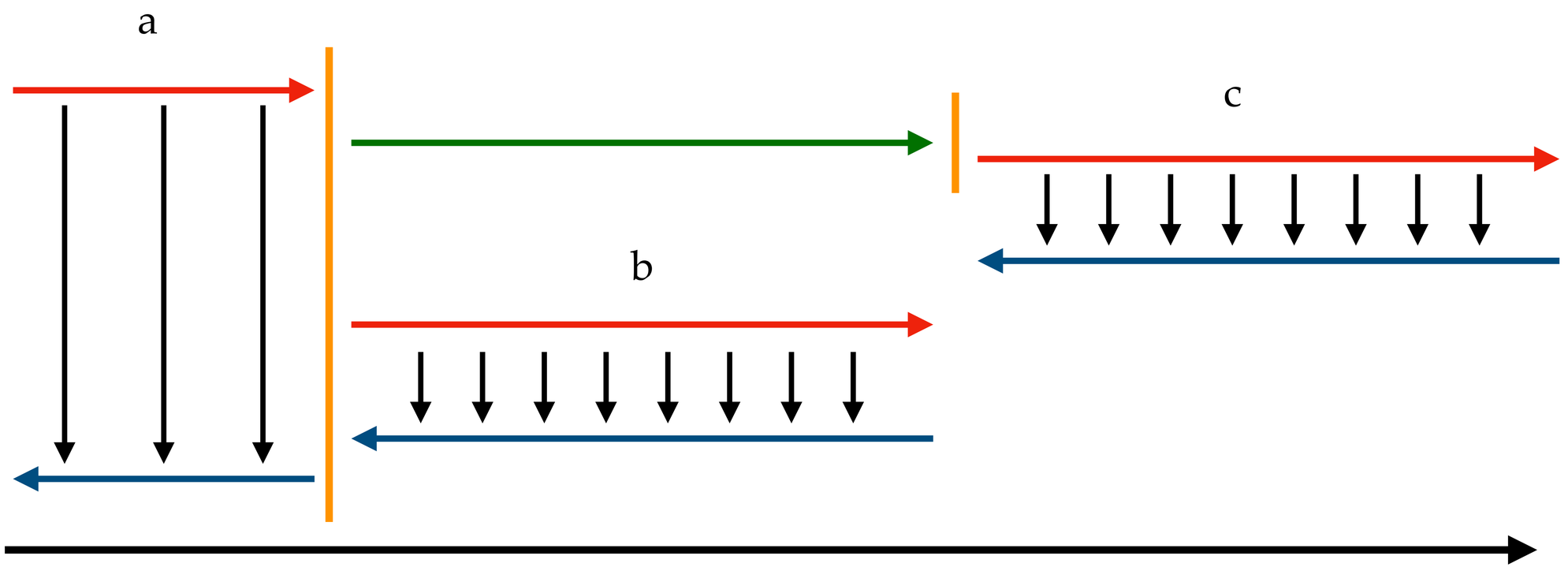
A visualization of checkpointing

one possible checkpointing scheme



A visualization of checkpointing

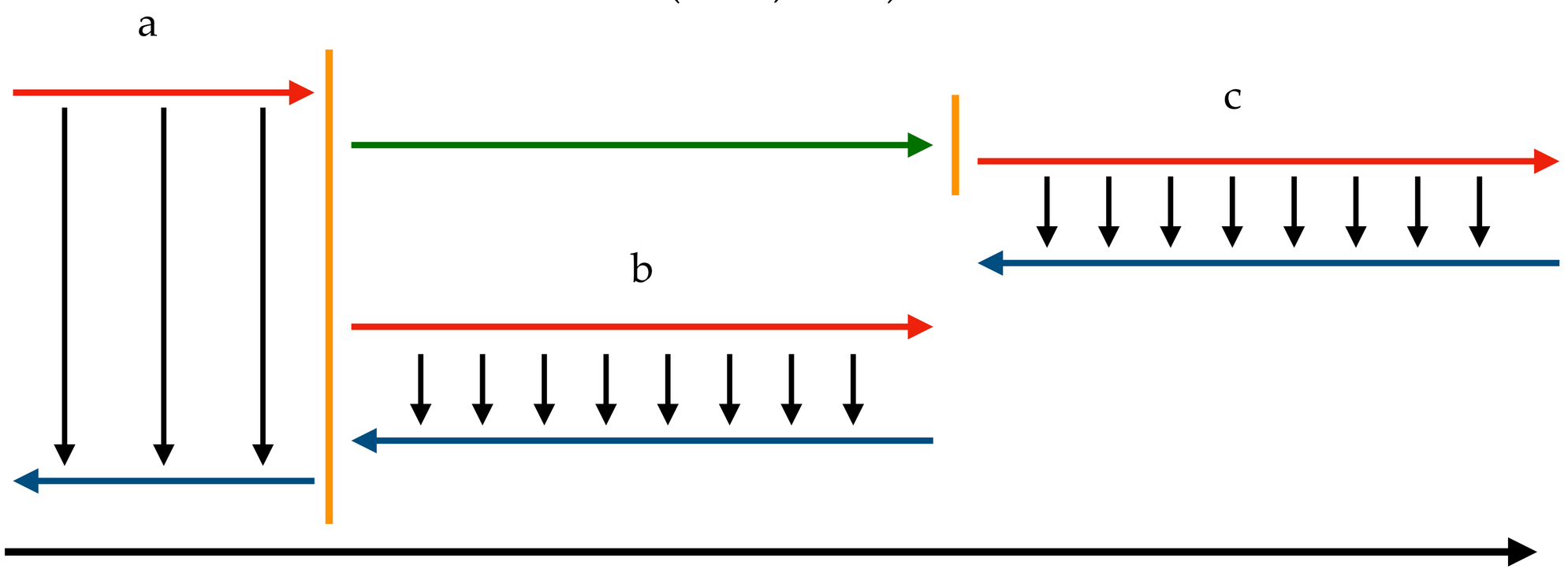
what is the peak memory use of the previous checkpointing scheme?



A visualization of checkpointing

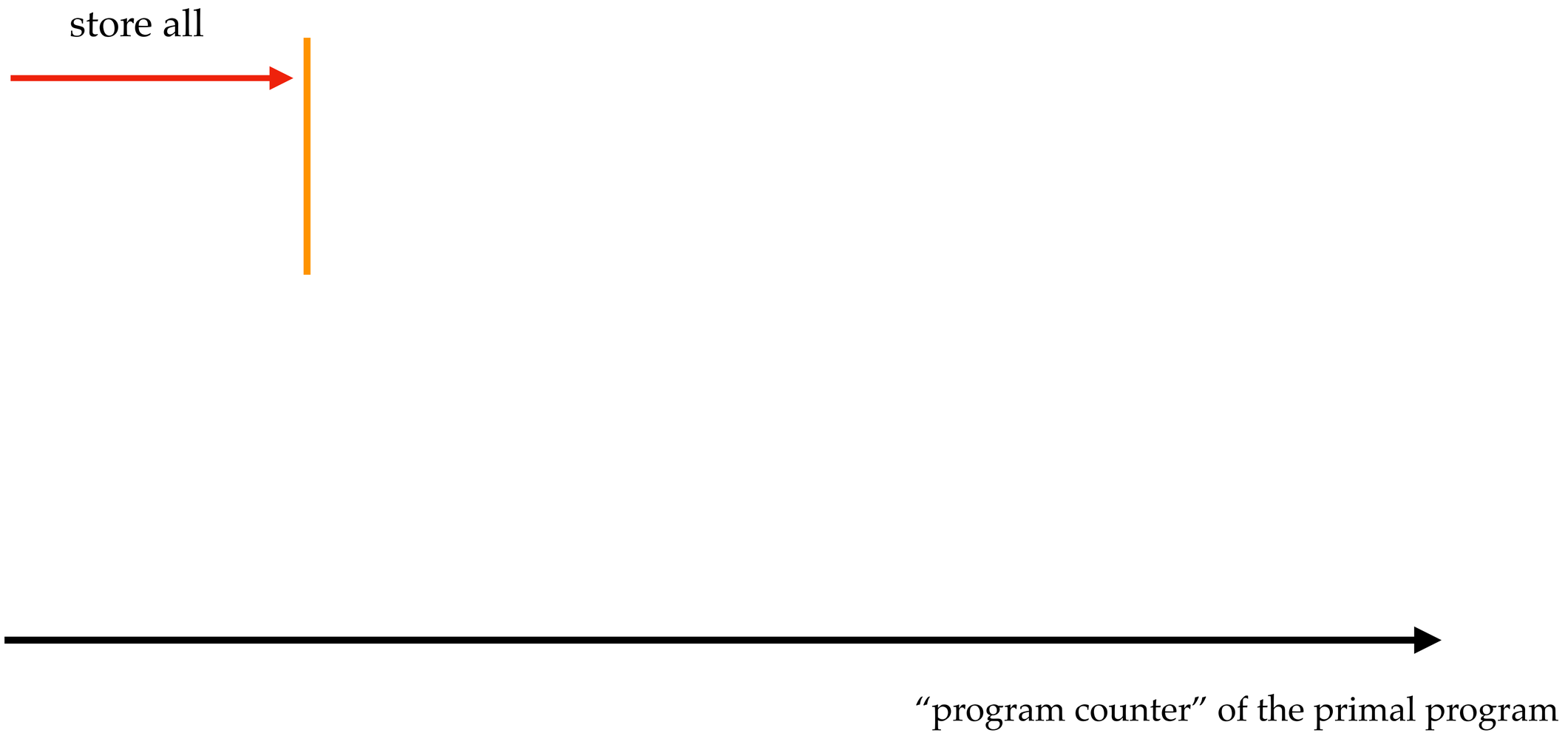
what is the peak memory use of the previous checkpointing scheme?

max(a + b, a + c)



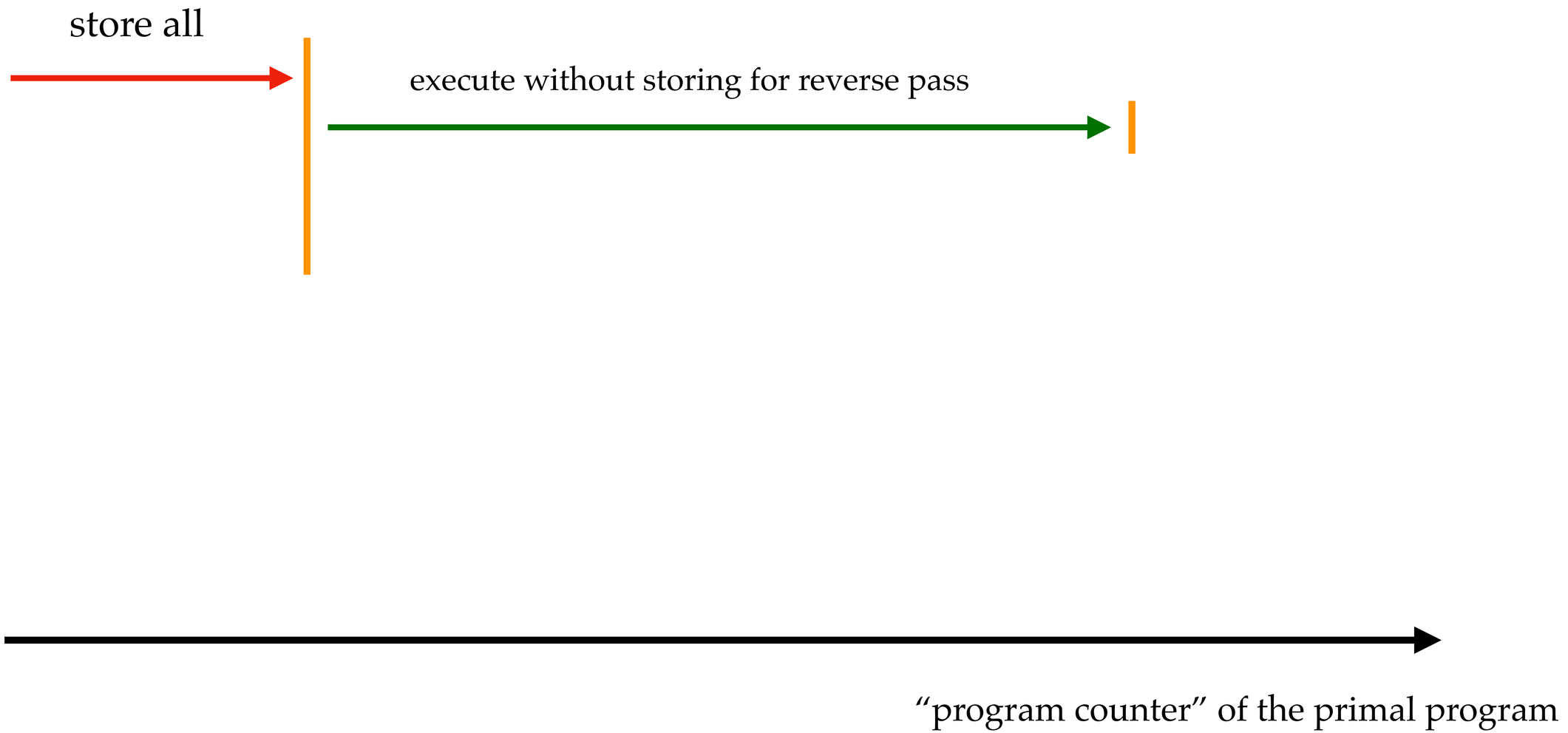
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



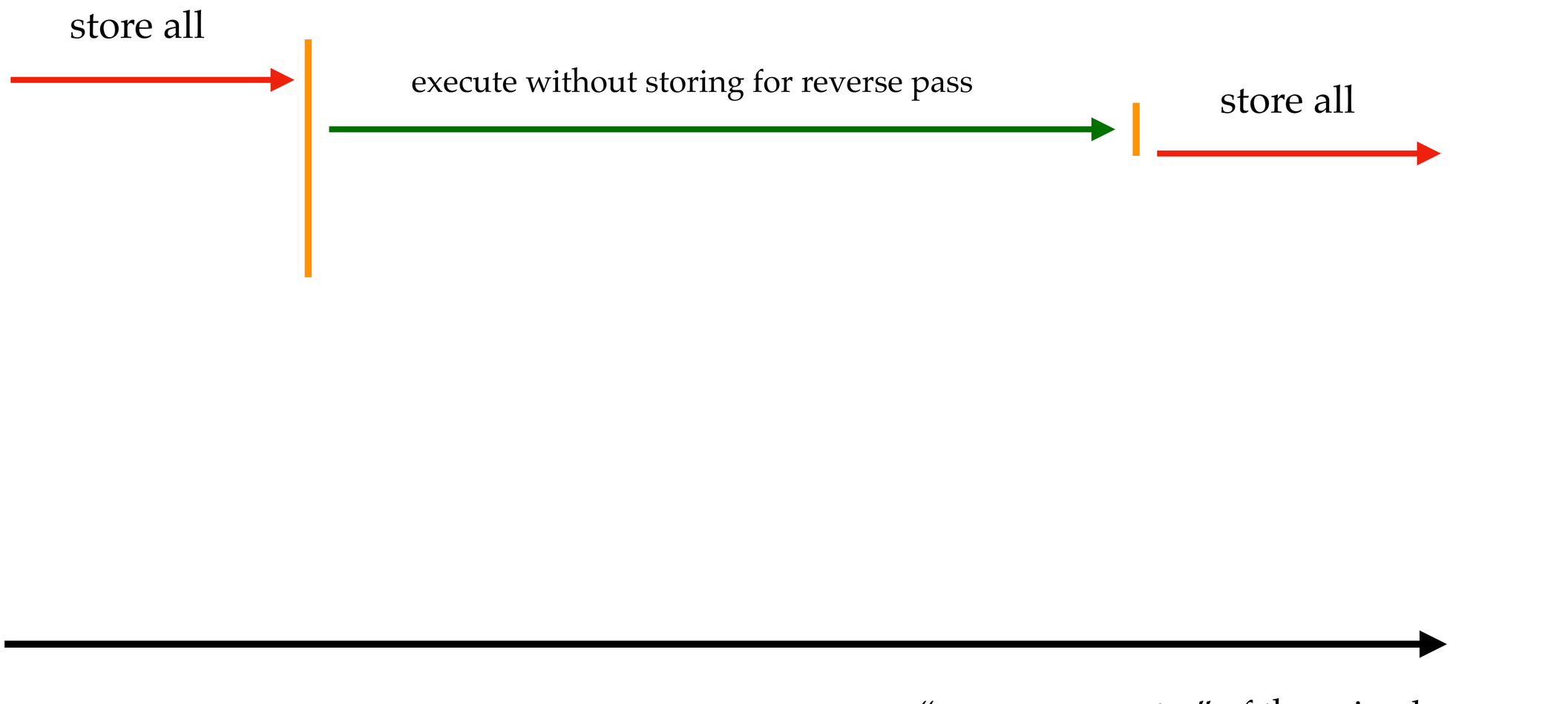
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



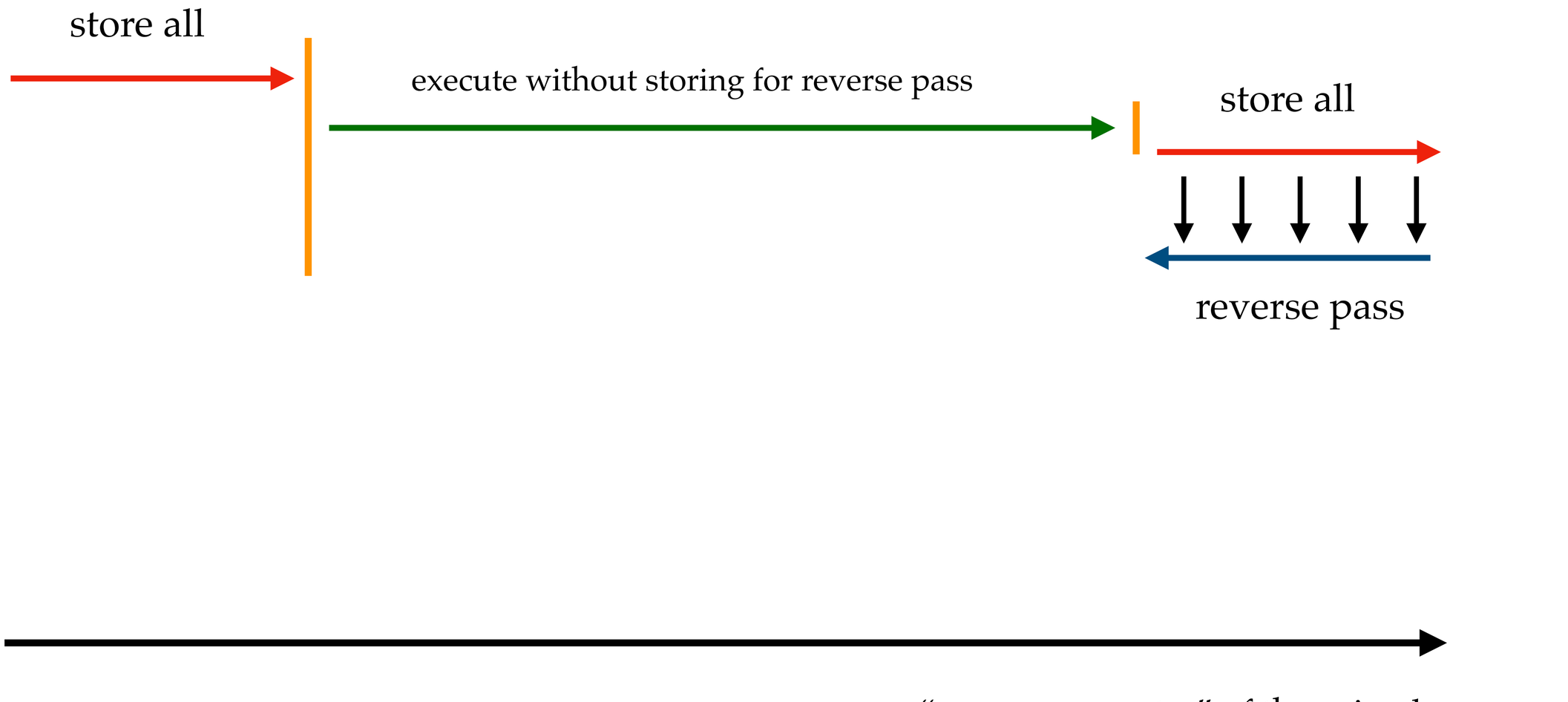
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



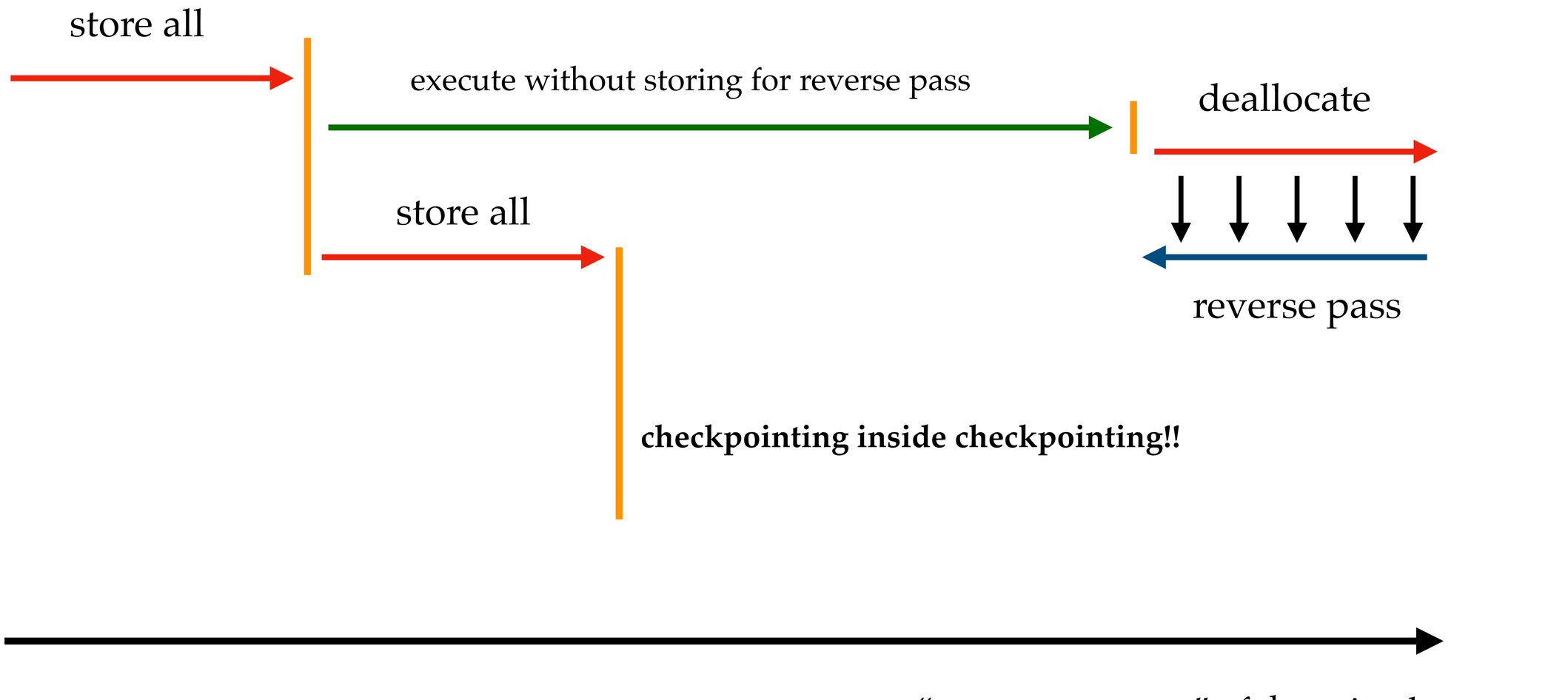
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



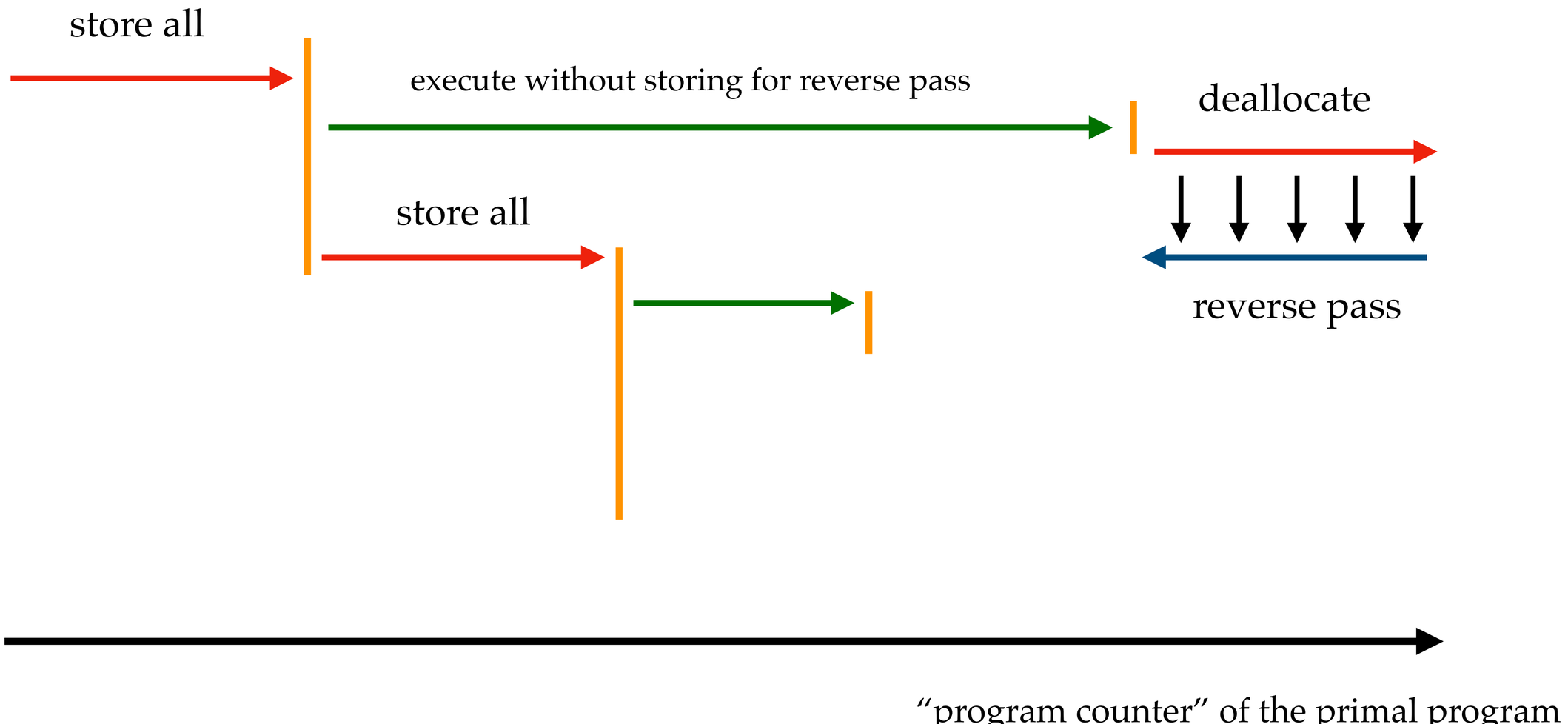
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



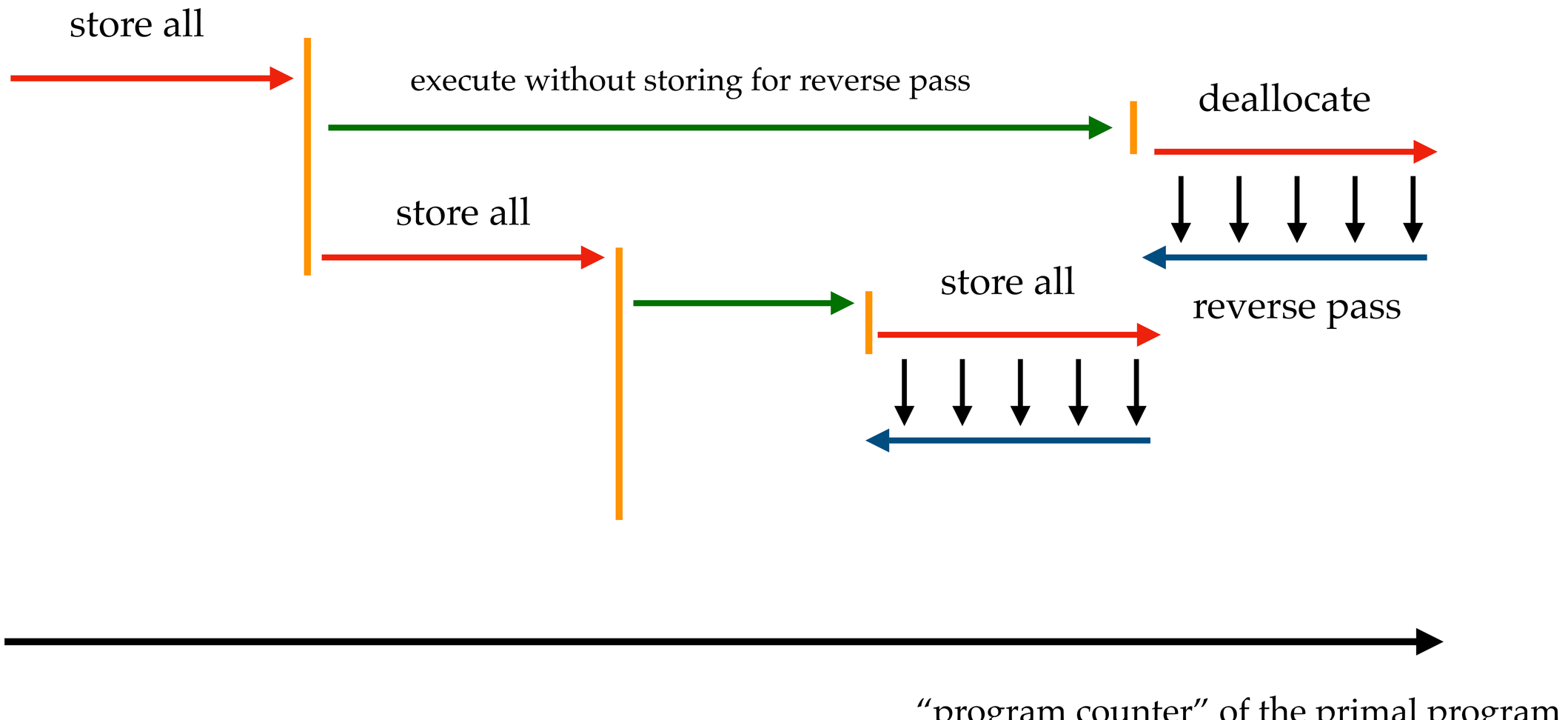
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



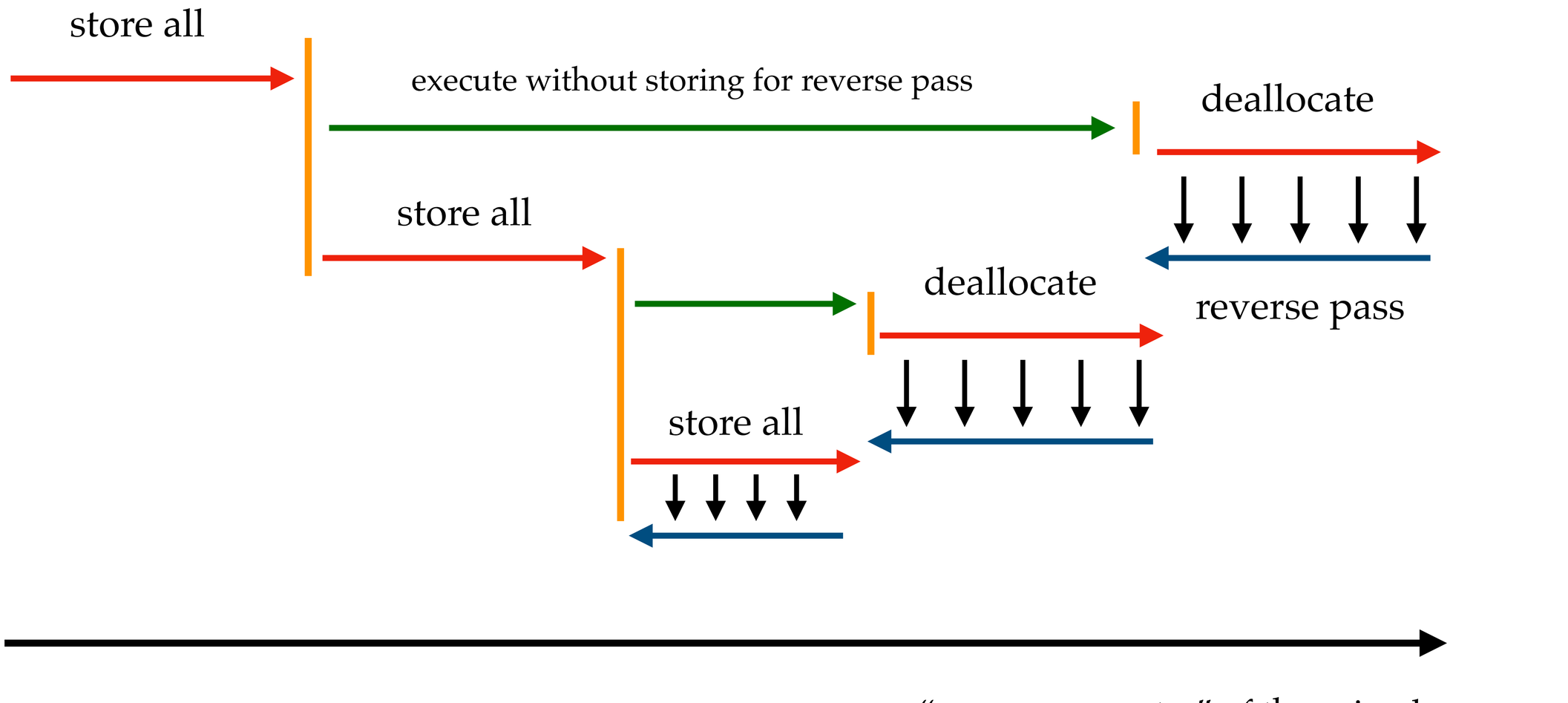
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



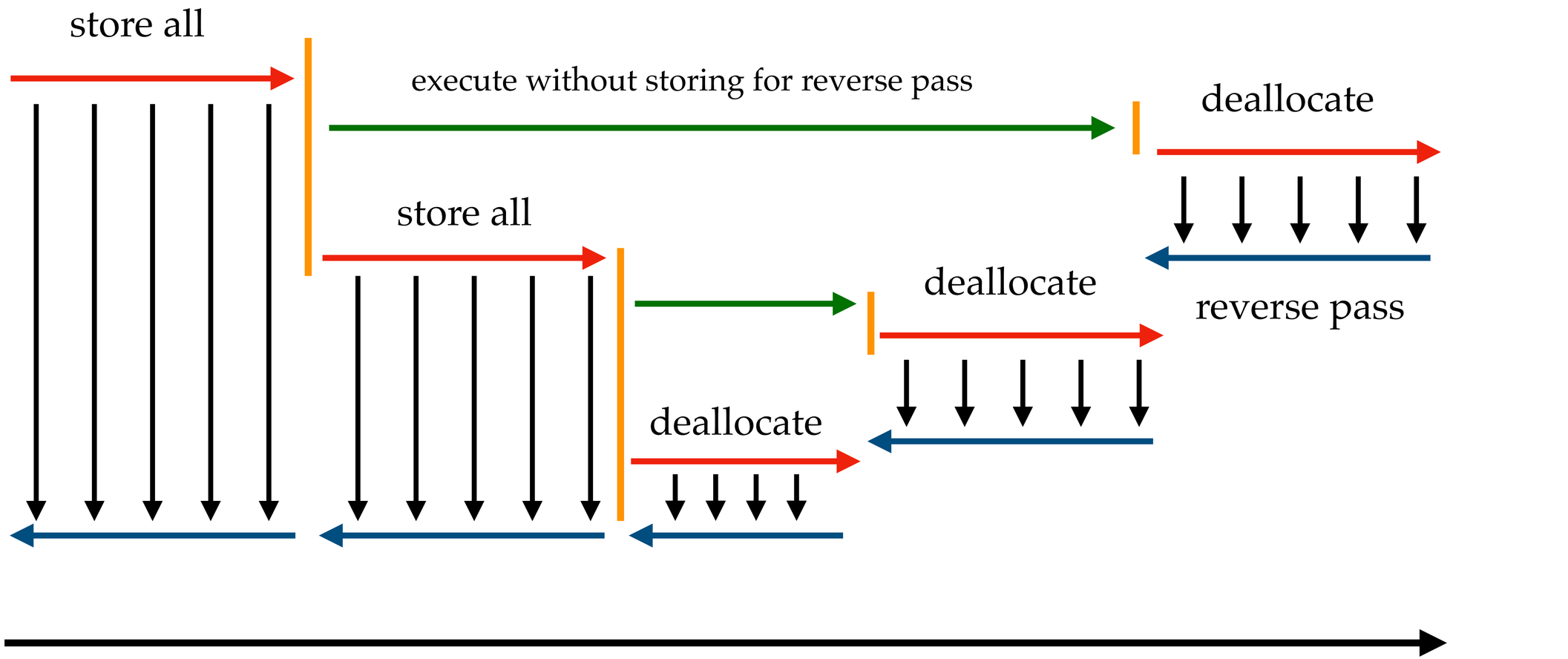
Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility



Multilevel checkpointing

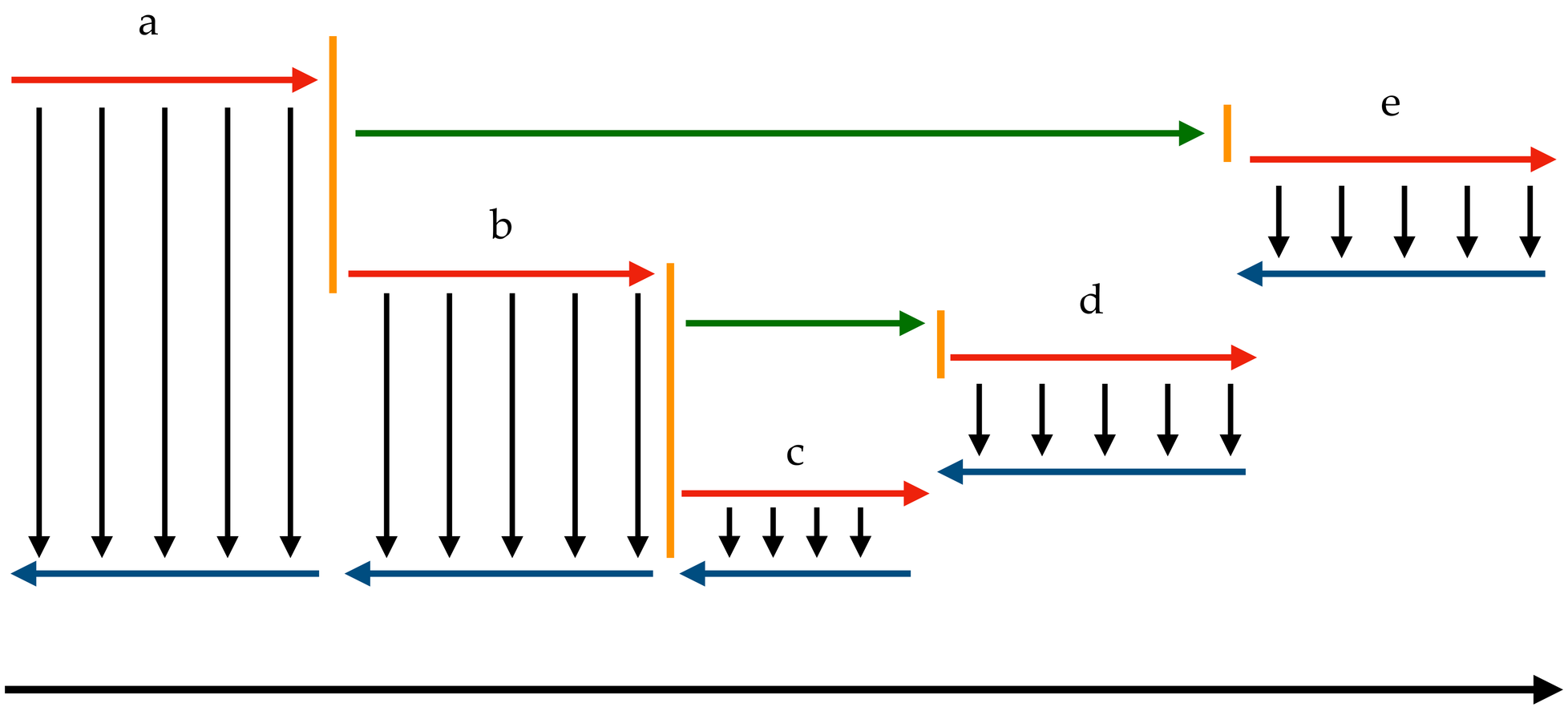
we can recursively apply checkpointing to allow even more flexibility



Multilevel checkpointing

we can recursively apply checkpointing to allow even more flexibility

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



Bisection-based Checkpointing [Griewank 1992]

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

ACHIEVING LOGARITHMIC GROWTH OF TEMPORAL AND SPATIAL COMPLEXITY IN REVERSE AUTOMATIC DIFFERENTIATION *

ANDREAS GRIEWANK

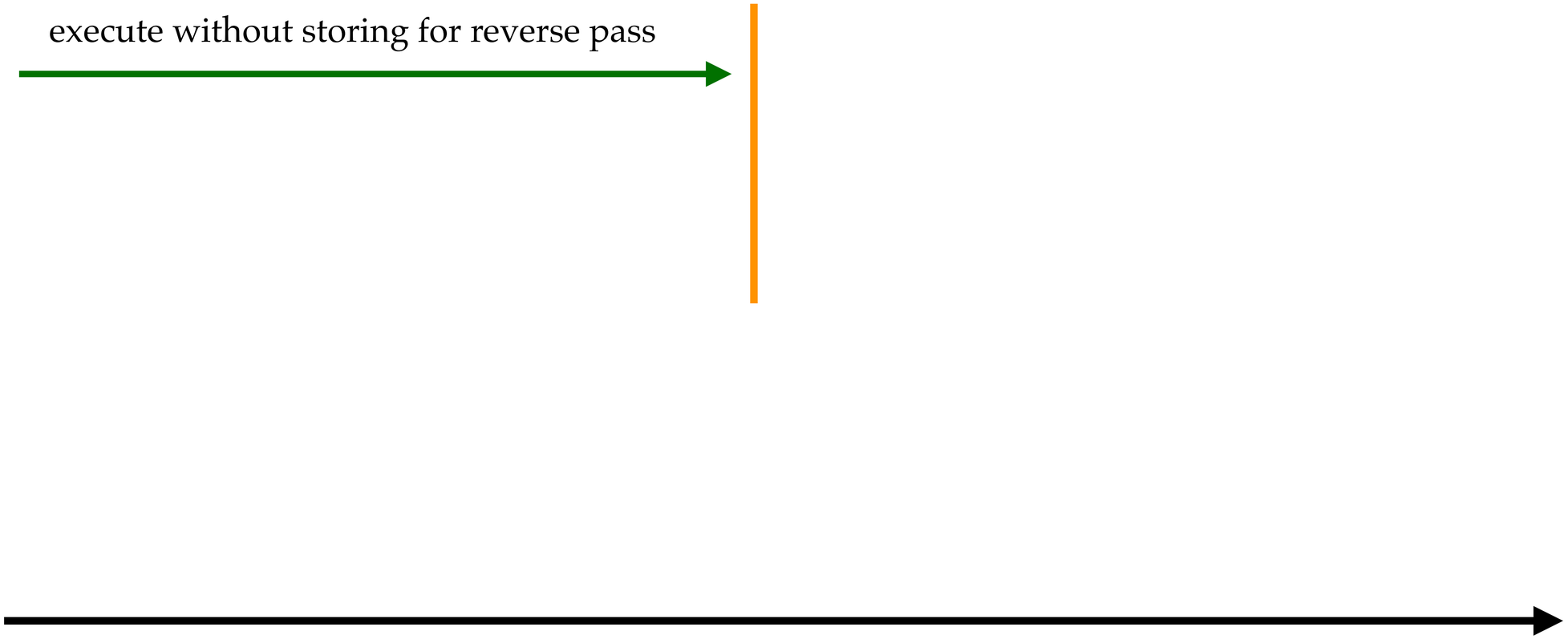
Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439

“program counter” of the primal program

Bisection-based Checkpointing [Griewank 1992]

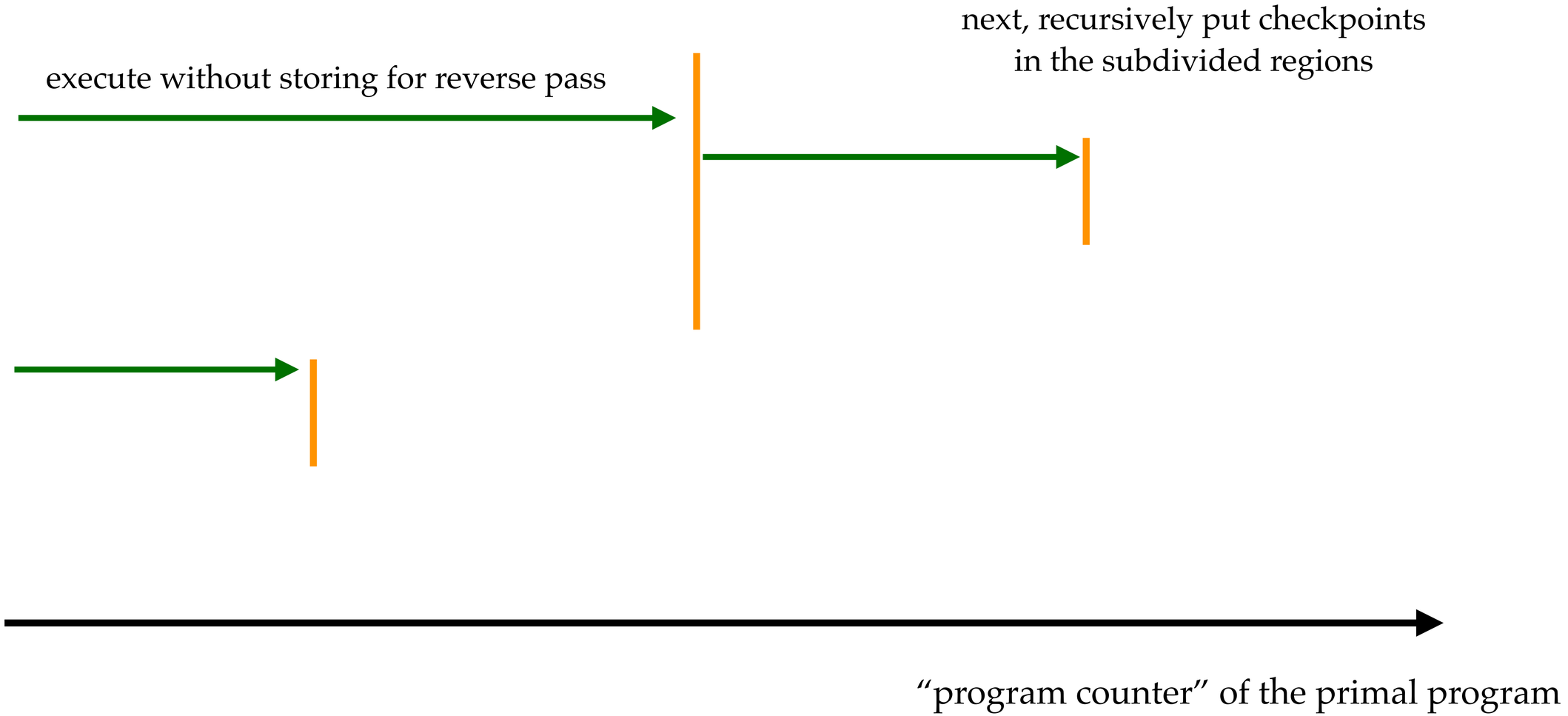
idea: apply multilevel checkpointing in a way that subdivide the regions into a balanced binary tree

first, place a checkpoint at the center



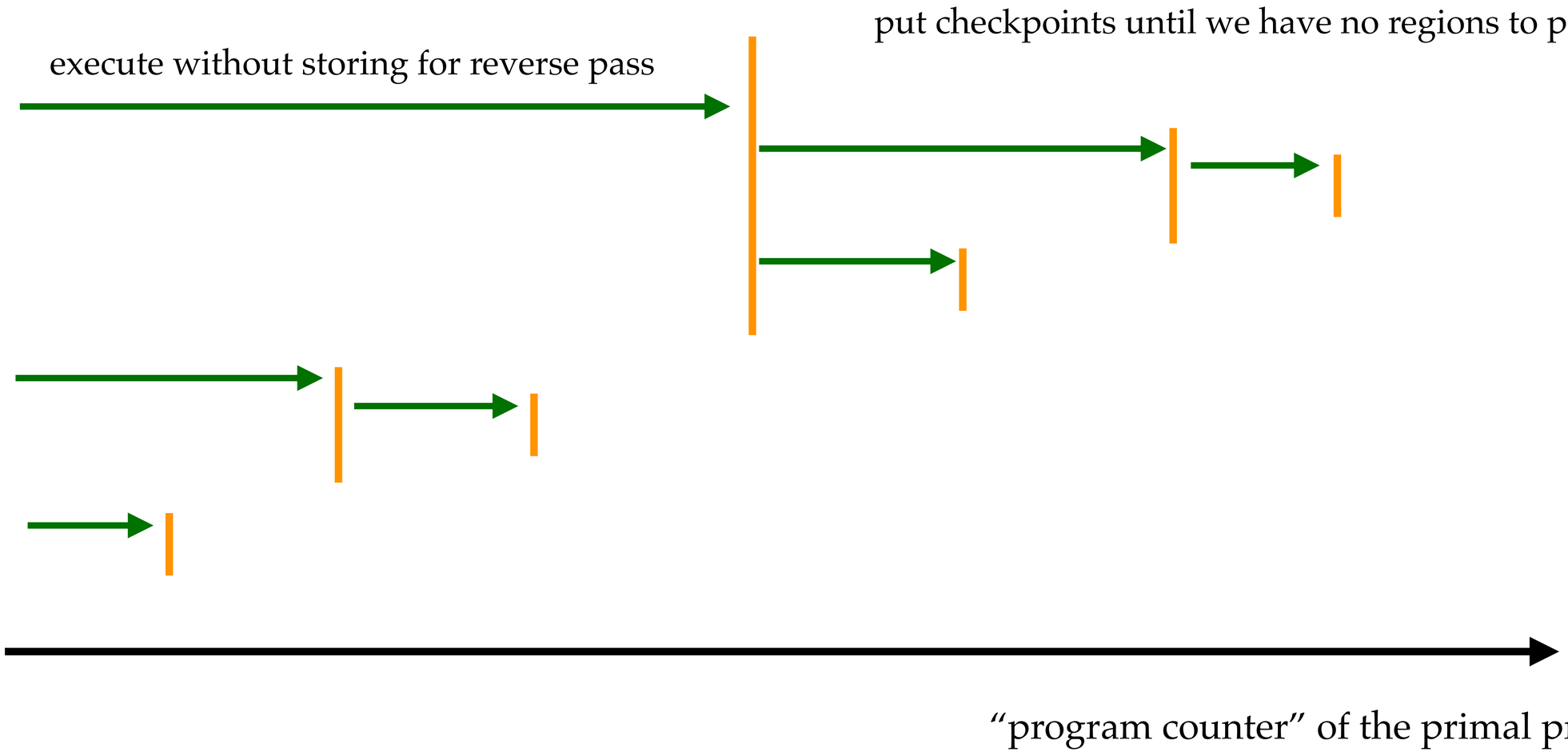
Bisection-based Checkpointing [Griewank 1992]

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



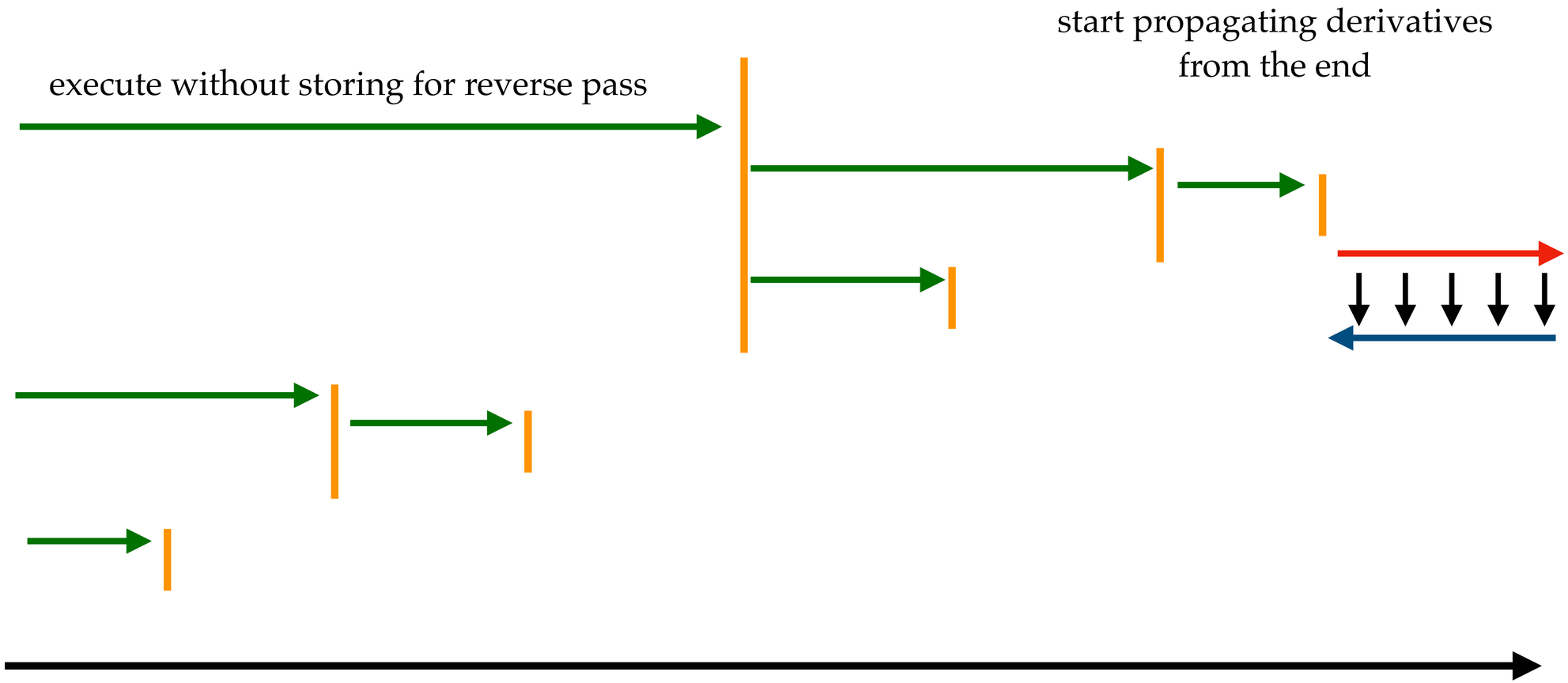
Bisection-based Checkpointing [Griewank 1992]

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



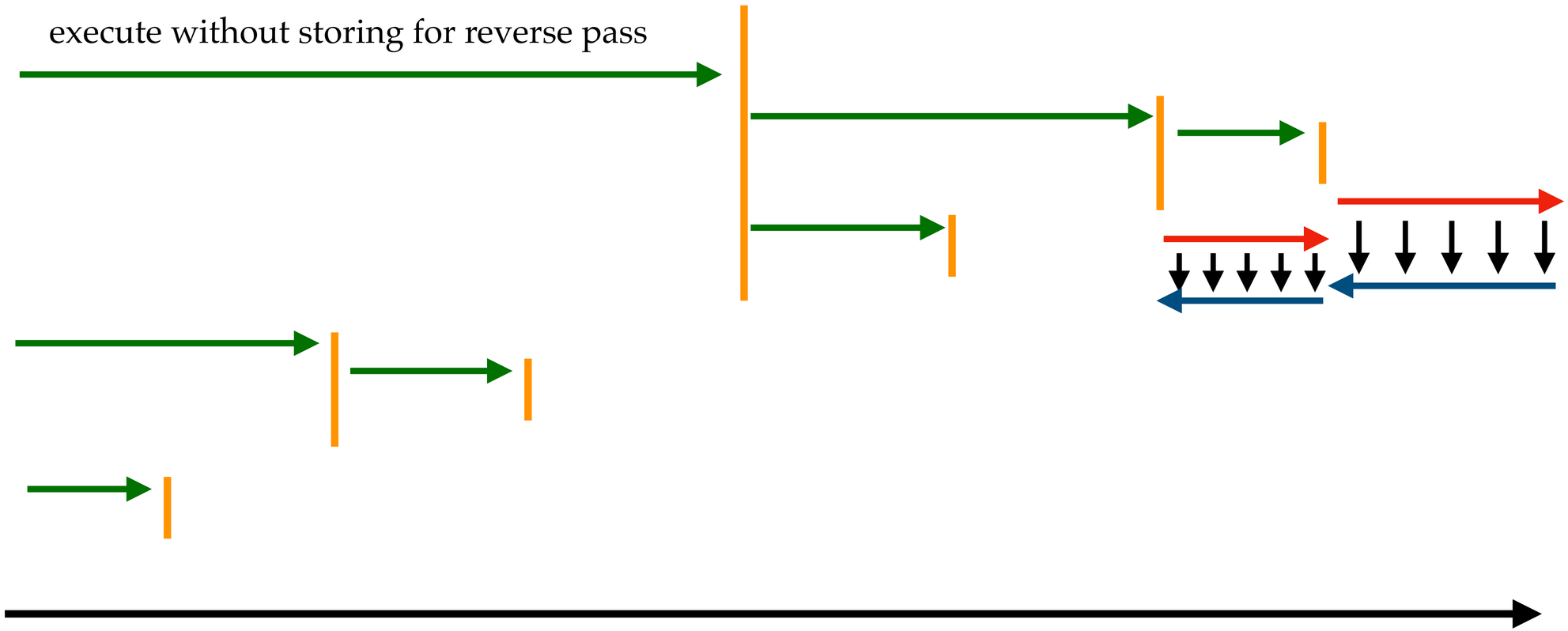
Bisection-based Checkpointing [Griewank 1992]

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



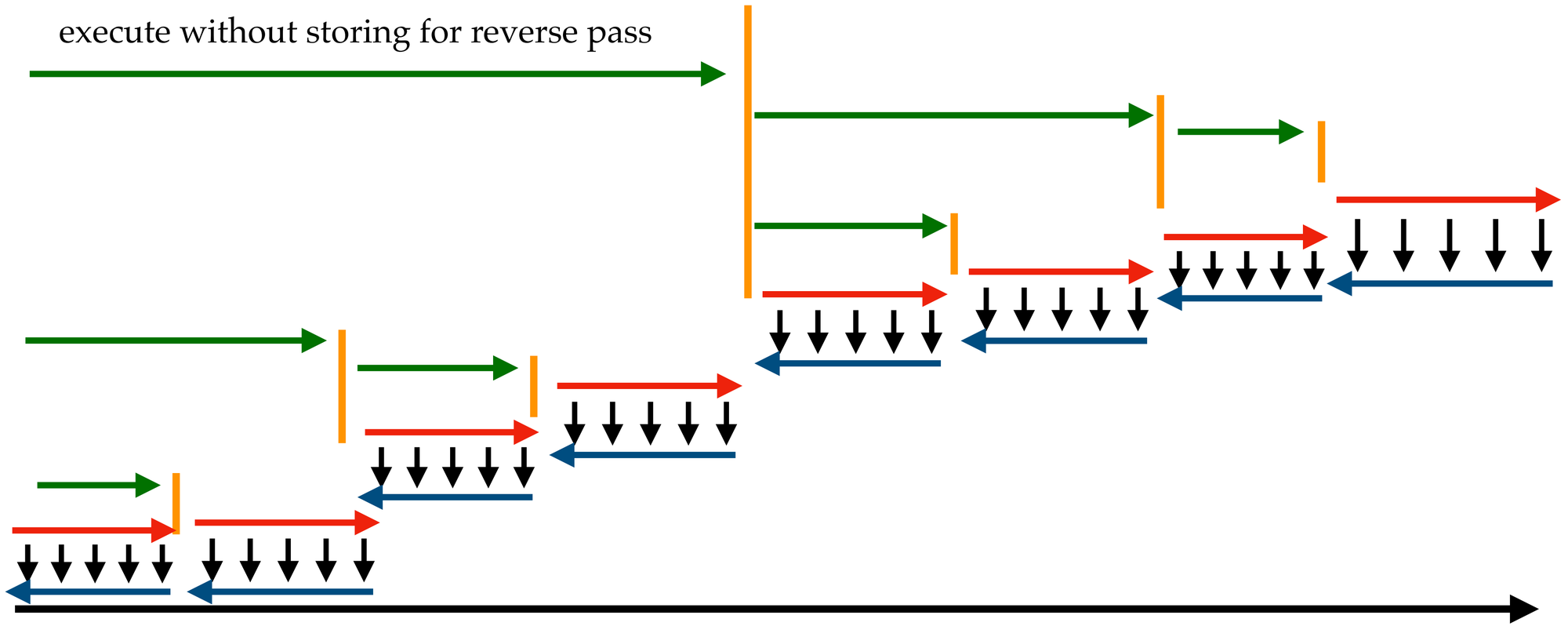
Bisection-based Checkpointing [Griewank 1992]

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



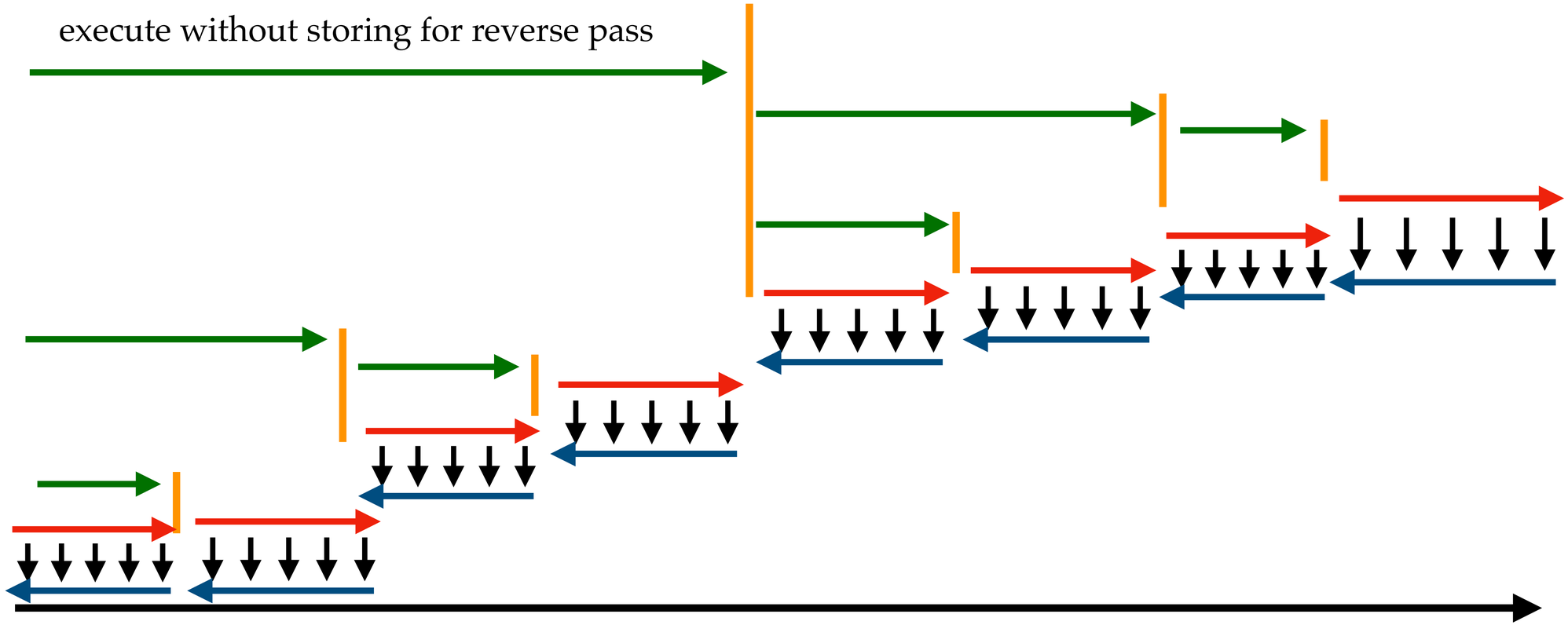
Bisection-based Checkpointing [Griewank 1992]

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



Bisection-based Checkpointing [Griewank 1992]

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

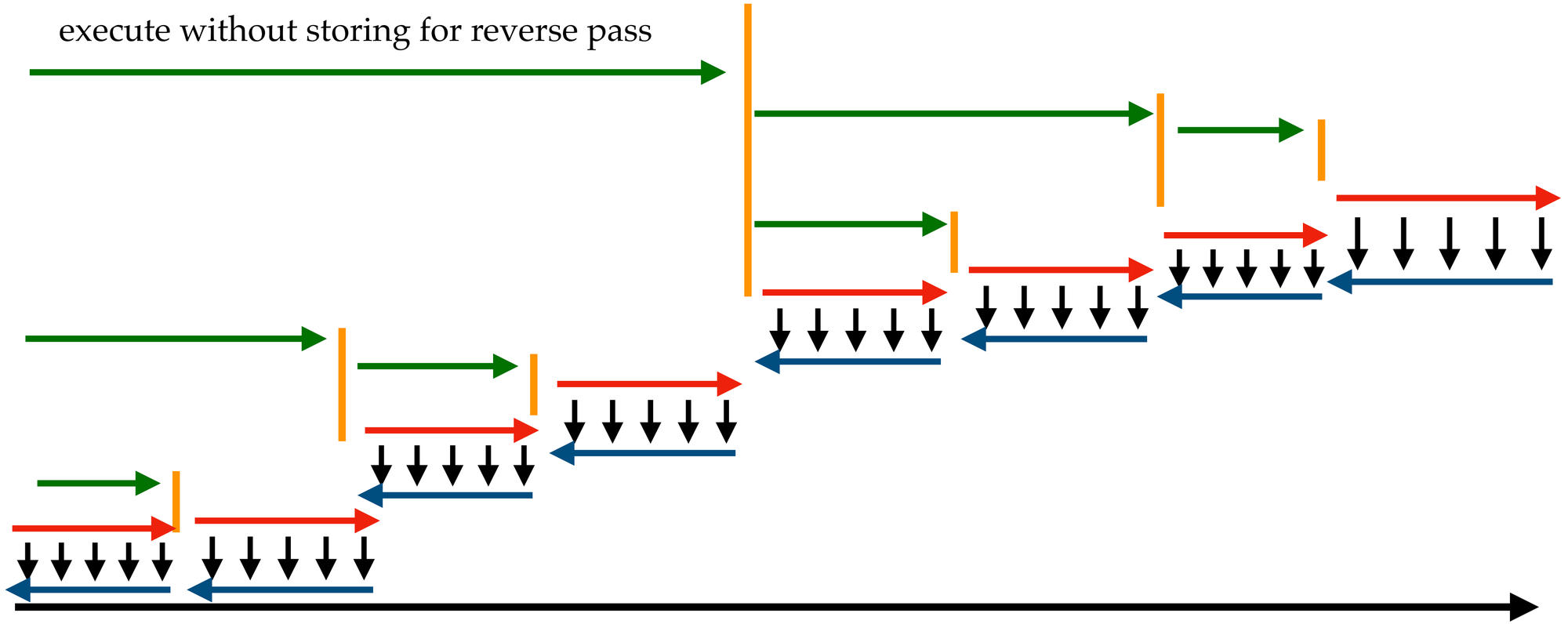


Bisection-based Checkpointing [Griewank 1992]

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

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

time complexity: sum of green lines + O(n) = O(nlog(n))



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

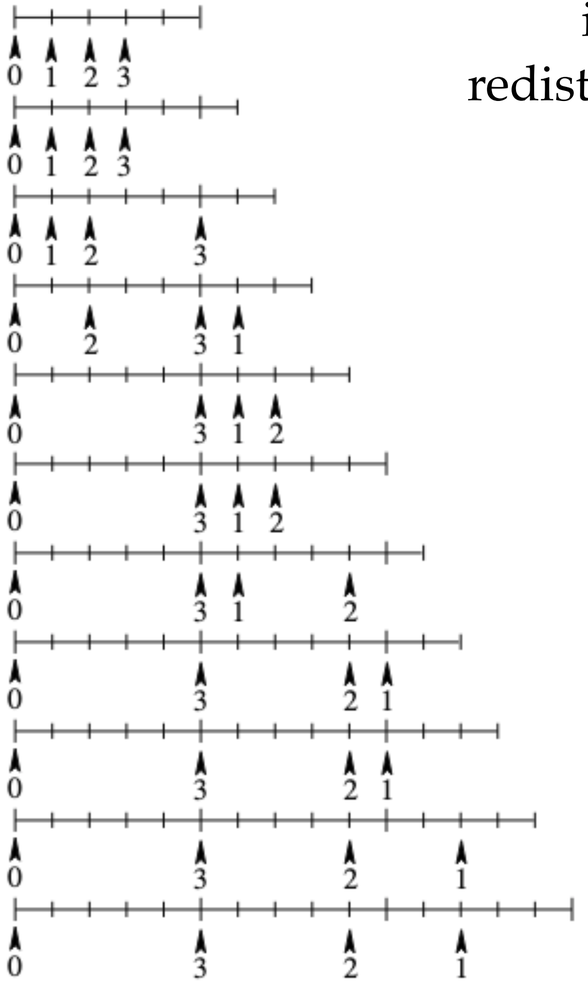
both bisection and binomial checkpointing requires knowing the number of steps taken in the primal code
these are known as “offline” checkpointing algorithms

“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

SIAM J. SCI. COMPUT.
Vol. 32, No. 2, pp. 836–854

© 2010 Society for Industrial and Applied Mathematics

NEW ALGORITHMS FOR OPTIMAL ONLINE CHECKPOINTING*

PHILIPP STUMM† AND ANDREA WALTHER‡

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

online checkpointing

CHECKMATE: BREAKING THE MEMORY WALL WITH OPTIMAL TENSOR REMATERIALIZATION

Paras Jain [*1] Ajay Jain [*1] Aniruddha Nrusimha [1]
Amir Gholami [1] Pieter Abbeel [1] Kurt Keutzer [1] Ion Stoica [1] Joseph E. Gonzalez [1]

offline checkpointing

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

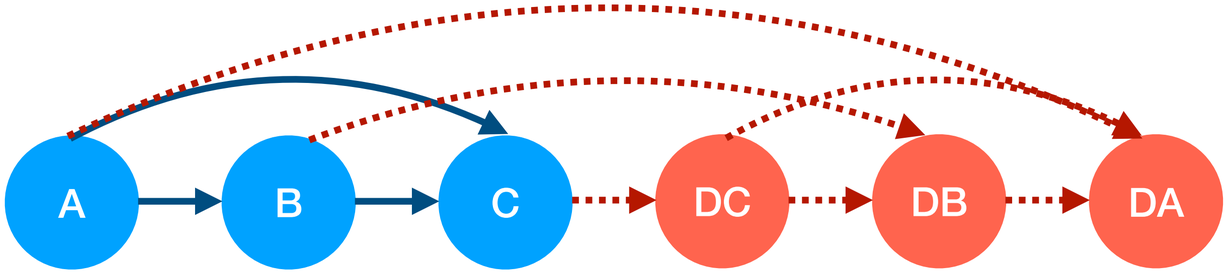
online checkpointing

CHECKMATE: BREAKING THE MEMORY WALL WITH OPTIMAL TENSOR REMATERIALIZATION

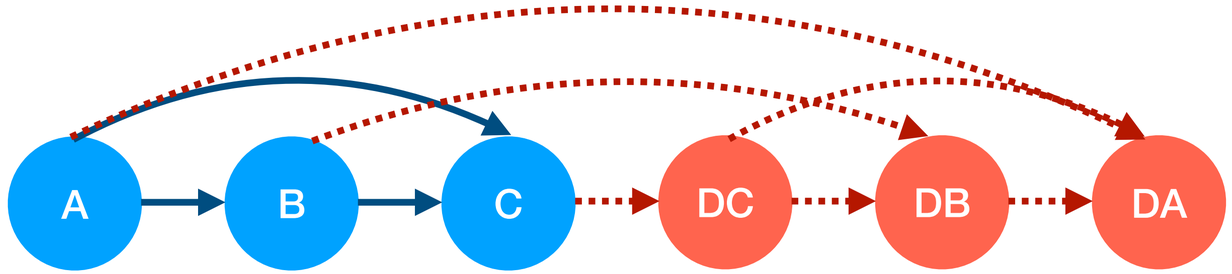
Paras Jain [*1] Ajay Jain [*1] Aniruddha Nrusimha [1]
Amir Gholami [1] Pieter Abbeel [1] Kurt Keutzer [1] Ion Stoica [1] Joseph E. Gonzalez [1]

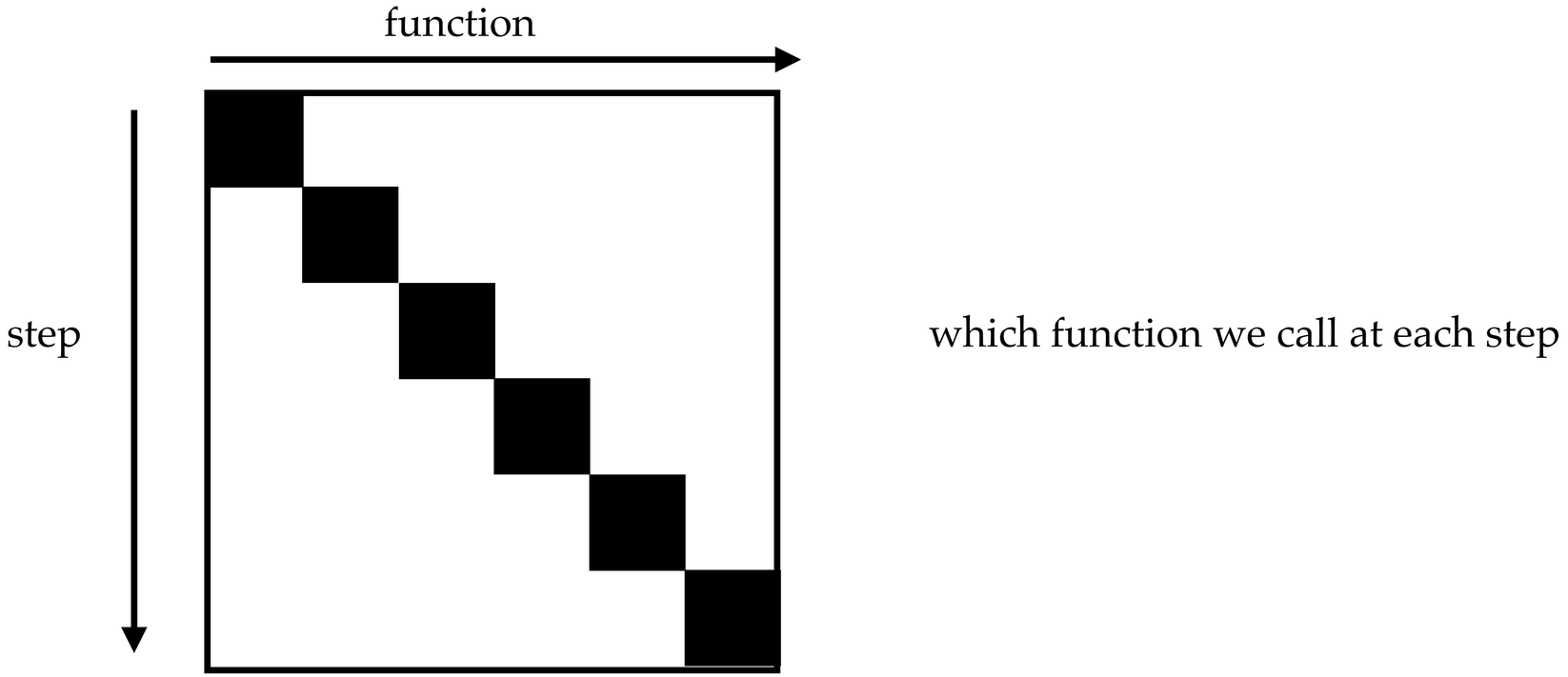
offline checkpointing

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

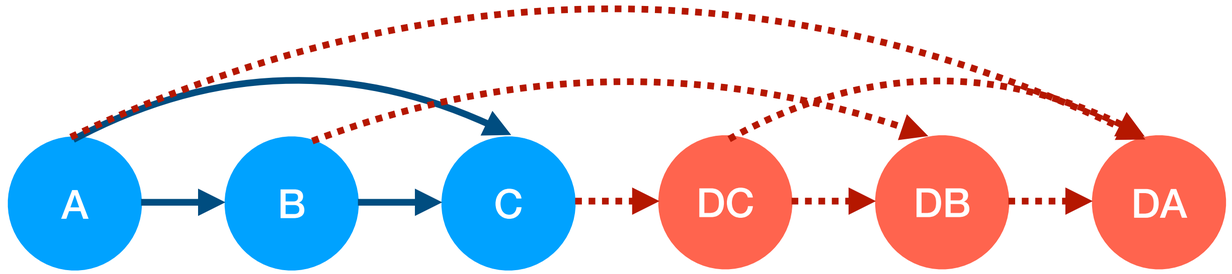


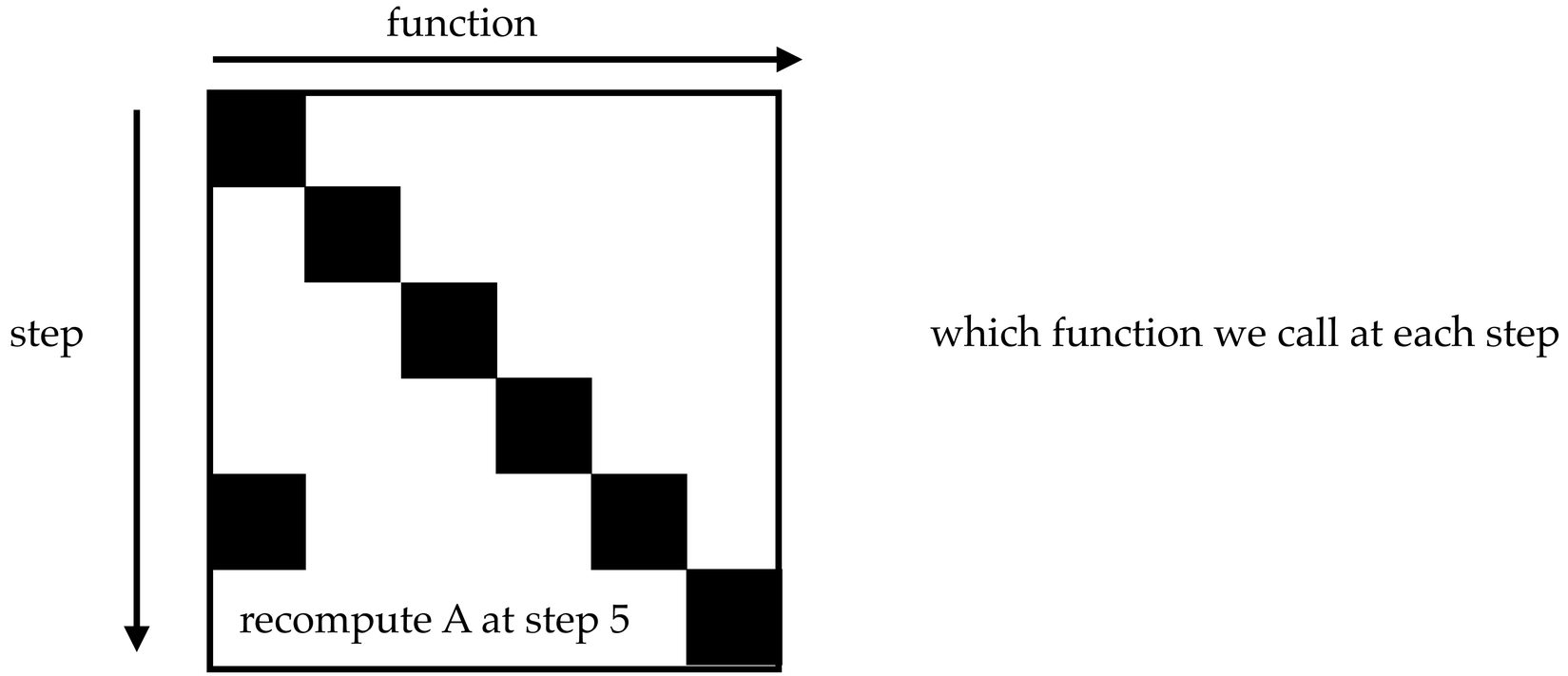
Checkmate: compute optimal strategy from a static data-flow graph



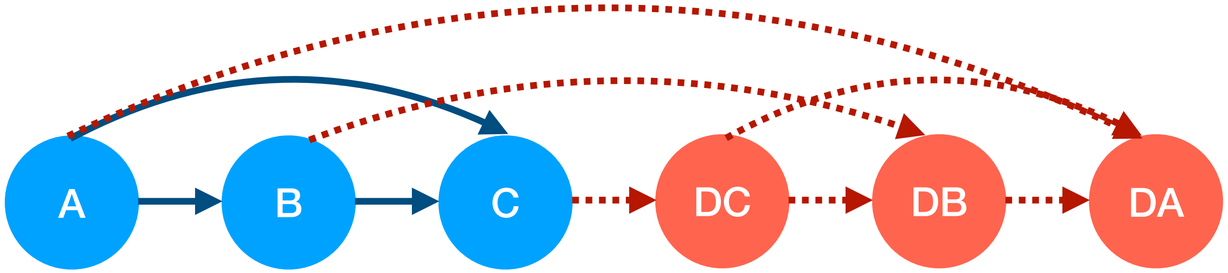


Checkmate: compute optimal strategy from a static data-flow graph





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

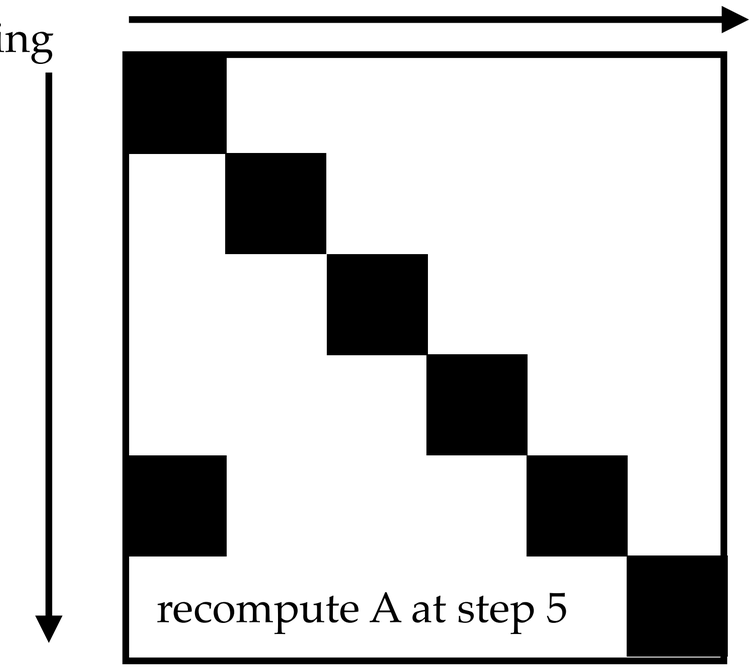


the optimal compute strategy can be formulated as an integer linear programming problem

$$\min_{R} \sum \sum C_i R_{t,i}$$

subject to memory limit

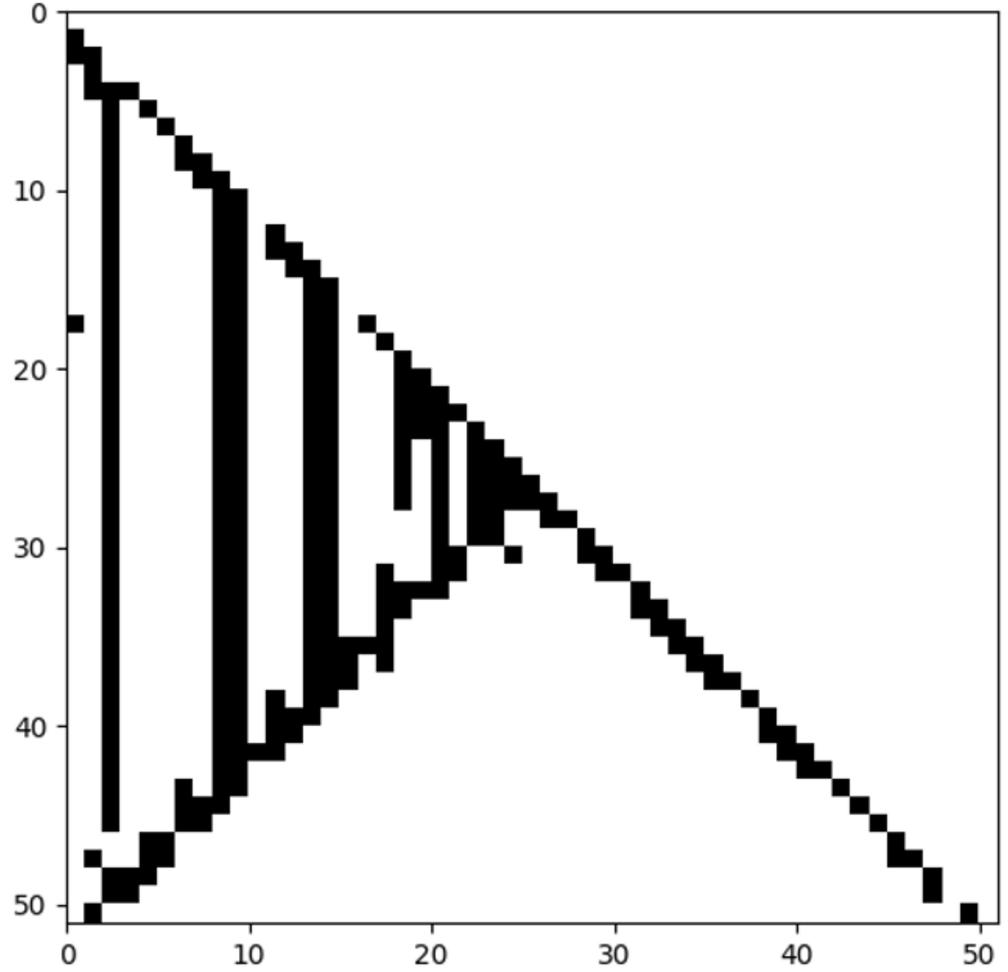
C is the cost



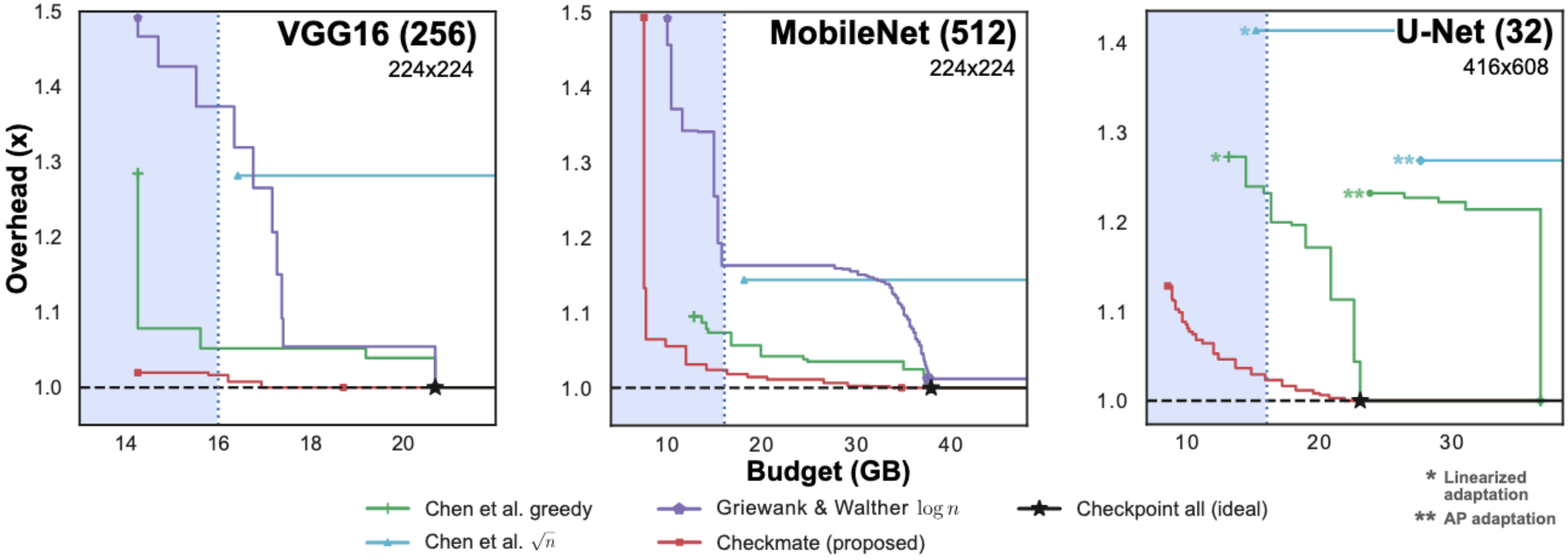
which function we call at each step

this is the R matrix

Optimal R matrix for VGG19



Results


VGG16 (256)
224x224
MobileNet (512)
224x224
U-Net (32)
416x608
Overhead (x)
Budget (GB)
Chen et al. greedy
Chen et al. $\sqrt{n}$
Griewank & Walther $\log n$
Checkmate (proposed)
Checkpoint all (ideal)
* Linearized adaptation
** AP adaptation

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

online checkpointing

CHECKMATE: BREAKING THE MEMORY WALL WITH OPTIMAL TENSOR REMATERIALIZATION

Paras Jain [*1] Ajay Jain [*1] Aniruddha Nrusimha [1]
Amir Gholami [1] Pieter Abbeel [1] Kurt Keutzer [1] Ion Stoica [1] Joseph E. Gonzalez [1]

offline checkpointing

Dynamic tensor rematerialization

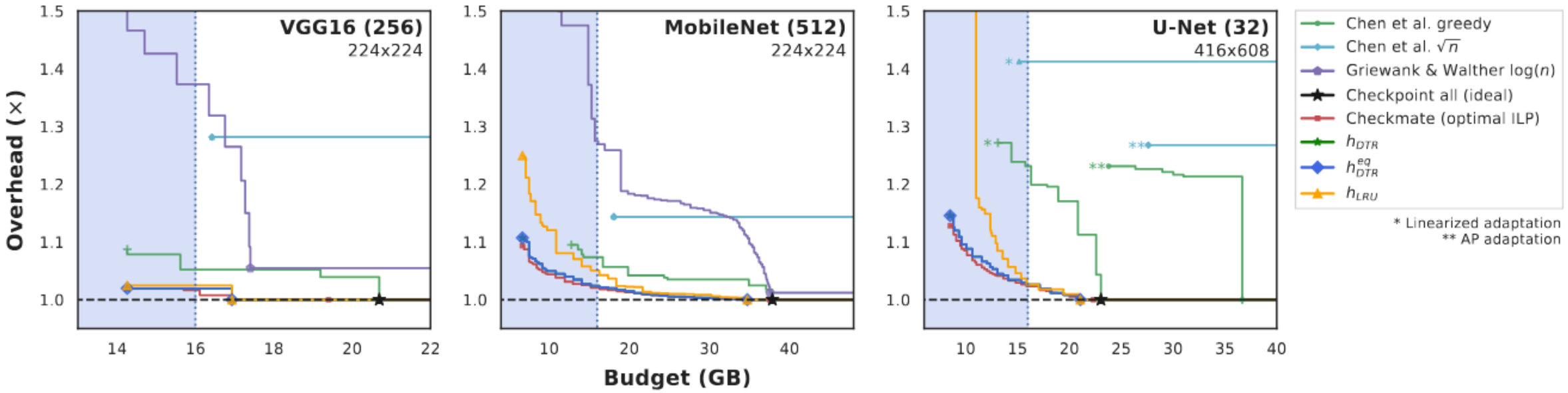
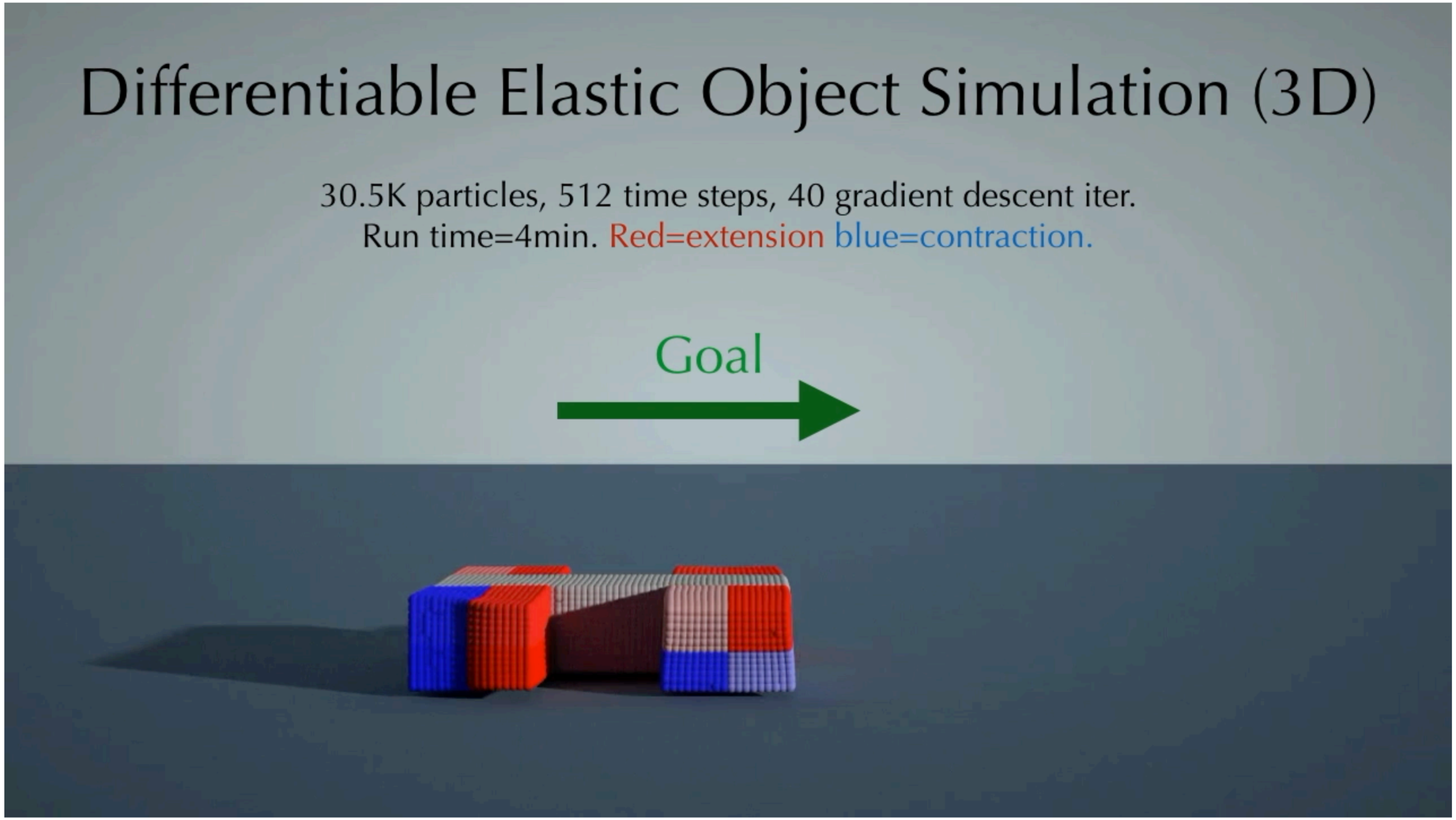
idea: override array allocation/access

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)