Parallel Automatic Differentiation

UCSD CSE 291
Tzu-Mao Li
The SIMD parallel model

```python
@simd
def par_add(x: In[Array[float]],
y: In[Array[float]],
z: Out[Array[float]]):
    i = thread_id()
    z[i] = x[i] * x[i] + y[i] * y[i]
```

```
parallel_for (i) {
    z[i] = x[i] * x[i] + y[i] * y[i]
}
```
Differentiating the SIMD parallel model

```python
@simd
def par_add(x : In[Array[float]], y : In[Array[float]], z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * x[i] + y[i] * y[i]

rev_par_add = rev_diff(par_add)
```

what is the derivative?
Differentiating the SIMD parallel model

```python
@simd
def rev_par_add(x : In[Array[float]],
y : In[Array[float]],
z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * x[i] + y[i] * y[i]

rev_par_add = rev_diff(par_add)
```

```python
@simd
def rev_par_add(x : In[Array[float]],
y : In[Array[float]],
z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * x[i] + y[i] * y[i]

dx[i] = dx[i] + dz[i] * 2 * x[i]
dy[i] = dy[i] + dz[i] * 2 * y[i]
```

is it correct?
Differentiating the SIMD parallel model

```python
@simd
def par_add(x : In[Array[float]],
y : In[Array[float]],
z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * x[i] + y[i] * y[i]
    return z

rev_par_add = rev_diff(par_add)
```

```python
@simd
def rev_par_add(x : In[Array[float]],
dx : Out[Array[float]],
y : In[Array[float]],
dy : Out[Array[float]],
dz : In[Array[float]]):
    i : int = thread_id()
    # z[i] = x[i] * x[i] + y[i] * y[i]
    dx[i] = dx[i] + dz[i] * 2 * x[i]
    dy[i] = dy[i] + dz[i] * 2 * y[i]

is it correct? yes
```
Differentiating the SIMD parallel model

@simd
def par_mul(x : In[Array[float]], y : In[float], z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * y

rev_par_mul = rev_diff(par_mul)
Differentiating the SIMD parallel model

```python
@simd
def par_mul(x : In[Array[float]], y : In[float], z : Out[Array[float]]):
    i = thread_id()
    z[i] = x[i] * y

rev_par_mul = rev_diff(par_mul)
```

```python
@simd
def rev_par_mul(x : In[Array[float]],
                dx : Out[Array[float]],
                y : In[float],
                dy : Out[float],
                dz : In[Array[float]]):
    i = thread_id()
    z[i] = x[i] * y
    dx[i] = dx[i] + dz[i] * y
    dy = dy + dz[i] * x[i]
```

is it correct?
Differentiating the SIMD parallel model

```python
@simd
def par_mul(x: In[Array[float]],
y: In[float],
z: Out[Array[float]]):
    i: int = thread_id()
    z[i] = x[i] * y
    z = rev_par_mul(x, y, z)

rev_par_mul = rev_diff(par_mul)
```

```python
@simd
def rev_par_mul(x: In[Array[float]],
    dx: Out[Array[float]],
y: In[float],
    dy: Out[float],
dz: In[Array[float]]):
    i: int = thread_id()
    # z[i] = x[i] * y
    dx[i] = dx[i] + dz[i] * y
    dy = dy + dz[i] * x[i]
```

different threads will write to dy at the same time
— race condition!!
Differentiating the SIMD parallel model

```python
@simd
def par_mul(x : In[Array[float]], y : In[float], z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * y

rev_par_mul = rev_diff(par_mul)
```

```python
@simd
def rev_par_mul(x : In[Array[float]], dx : Out[Array[float]],
                y : In[float], dy : Out[float],
                dz : In[Array[float]]):
    i : int = thread_id()
    # z[i] = x[i] * y
    dx[i] = dx[i] + dz[i] * y
    atomic_add(dy, dz[i] * x[i])
```
In reverse mode, read -> write  
write -> read

@simd
def par_mul(x : In[Array[float]],  
y : In[float],  
z : Out[Array[float]]):
  i : int = thread_id()
  z[i] = x[i] * y
  dx[i] = dx[i] + dz[i] * y
  atomic_add(dy, dz[i] * x[i])

parallel read from the same memory location will become race condition
Falling back to atomic add is always safe in homework, we will do this.
Atomic add itself is easily differentiable

```python
@simd
def par_reduce(x : In[Array[float]],
z : Out[float]):
    i : int = thread_id()
    atomic_add(z, x[i] * x[i])

rev_par_reduce = rev_diff(par_reduce)
```
Atomic add itself is easily differentiable

```python
@simd
def par_reduce(x: Array[Float], z: Out[Float]):
    i: int = thread_id()
    atomic_add(z, x[i] * x[i])

rev_par_reduce = rev_diff(par_reduce)
```

(in homework, we use atomic add just to be safe)

```python
@simd
def rev_par_reduce(x: Array[Float],
                   dx: Out[Array[Float]],
                   dz: In[Float]):
    i: int = thread_id()
    dx[i] = dx[i] + dz * 2 * x[i]
```
Differentiating general thread fork and join

fork

thread A

thread B

thread C

join

primal code

2022!

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

William S. Moses
MIT CSAIL
Cambridge, MA
wmoves@mit.edu

Sri Hari Krishna Narayanan
Argonne National Laboratory
Lemont, IL
smn07@anl.gov

Ludger Patzler
Technical University of Munich
Munich, Germany
ludger.patzler@tum.de

Valentin Churavy
MIT CSAIL
Cambridge, MA
vchuravy@mit.edu

Michel Schanen
Argonne National Laboratory
Lemont, IL
mschanen@anl.gov

Jan Hucklebeim
Argonne National Laboratory
Lemont, IL
jhbuckelbein@anl.gov

Johannes Doerfler
Argonne National Laboratory
Lemont, IL
jdoerfler@anl.gov

Paul Hovland
Argonne National Laboratory
Lemont, IL
phovland@anl.gov
Differentiating general thread fork and join

primal code

fork

thread A

thread B

thread C

join

reverse pass

fork

DA

DB

DC

works for CPU, GPU, OpenMP, MPI, …

2022!

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation
Rest of today

__syncthreads();

synchronization

ways to detect/remove race conditions
Rest of today

__syncthreads();

synchronization

ways to detect/remove race conditions
the “parallel reduction” algorithm

```c
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
        syncthreads()
    }
}
```

*note that on GPUs, global synchronization is very expensive and should be avoided, instead, block synchronization is cheap*
Barrier in parallel programming

The “parallel reduction” algorithm

```c
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
        syncthreads()
    }
}
```

How do we differentiate the barrier?

*Note that on GPUs, global synchronization is very expensive and should be avoided, instead, block synchronization is cheap.*
Differentiating barriers

A must be completed before entering B

A()

sync_threads()  reverse mode

B()
Differentiating barriers

A must be completed before entering B

A()

sync_threads()

B()

DB must be completed before entering DA

DB()

sync_threads()

DA()

Why is this correct?

Reverse-Mode Automatic Differentiation and Optimization of
GPU Kernels via Enzyme

William S. Moses*, Valentin Churavy*, Ludger Paehler†, Jan Hückelheim†,
Sri Hari Krishna Narayanan‡, Michel Schanen‡, Johannes Doerfert‡
{wmoses,vcchuravy}@mit.edu, ludger.paehler@tum.de, {jhuckelheim,smnarayan,mschanen,jdoerfert}@anl.gov
MIT CSAIL*, Technical University of Munich†, Argonne National Laboratory‡
Proving correctness of barrier differentiation

A() 
sync_threads() 
B()

DB() 
sync_threads() 
DA()

the memory access pattern can be categorized into 4 cases 
when A() and B() access the same memory location
Proving correctness of barrier differentiation

the memory access pattern can be categorized into 4 cases when A() and B() access the same memory location

<table>
<thead>
<tr>
<th>Case</th>
<th>Code A</th>
<th>Code B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A: ... store %ptr ... sync_threads()</td>
<td>B: ... store %ptr ...</td>
</tr>
</tbody>
</table>

case 1
Proving correctness of barrier differentiation

the memory access pattern can be categorized into 4 cases when A() and B() access the same memory location

case 1

the sync_threads in reverse pass ensures that the adjoints are properly zeroed before entering DA

A:

... store %ptr
... sync_threads()

B:

... store %ptr

A:

... store %ptr
... sync_threads()

B:

... store %ptr
... sync_threads()

DB:

... load %d_ptr
... store %d_ptr = 0

DA:

... load %d_ptr
... store %d_ptr = 0
Proving correctness of barrier differentiation

the memory access pattern can be categorized into 4 cases when A() and B() access the same memory location

case 2

A:

... load %ptr
... sync_threads()

B:

... store %ptr
...
Proving correctness of barrier differentiation

the memory access pattern can be categorized into 4 cases when A() and B() access the same memory location

case 2

the sync_threads in reverse pass again ensures that the adjoints are properly zeroed before entering DA

A:

```
load %ptr
... sync_threads()
```

B:

```
store %ptr
...                     
```

DB:

```
load %d_ptr
store %d_ptr = 0
... sync_threads()
```

DA:

```
... atomicAdd %d_ptr
```
Proving correctness of barrier differentiation

A()
sync_threads()
B()

DB()
sync_threads()
DA()

the memory access pattern can be categorized into 4 cases
when A() and B() access the same memory location

case 3

A:
... store %ptr
... sync_threads()
B:
... load %ptr
...
Proving correctness of barrier differentiation

the memory access pattern can be categorized into 4 cases when A() and B() access the same memory location

case 3

the sync_threads in reverse pass ensures that DA loads the right differentials

A:
... 
store %ptr
... 
sync_threads()
B:
... 
load %ptr
... 

DB:
... 
atomicAdd %d_ptr
... 
sync_threads()

DA:
... 
load %d_ptr
store %d_ptr = 0
... 

A() sync_threads() B()

DB() sync_threads() DA()
Proving correctness of barrier differentiation

The memory access pattern can be categorized into 4 cases when A() and B() access the same memory location.

- **Case 4**: we actually don’t need barrier in this case but it’s fine.

```c
A:
    load %ptr
    ... sync_threads()
B:
    load %ptr
    ...
```

```c
DB:
    atomicAdd %d_ptr
    ... sync_threads()
DA:
    atomicAdd %d_ptr
    ...
```
Rest of today

ways to detect/remove race conditions
Atomic add is slow especially when thread contention is high needs to wait until previous threads finishing the increment

When the primal code reads from a thread local variable, we don’t need atomic add

```python
@simd
def f(x : In[Array[float]],
y0 : In[Array[int]],
y1 : In[Array[int]],
y2 : In[Array[int]],
z : Out[Array[float]]):
i : int = thread_id()
t : Array[float, 10]
t[y0[i]] = x[y1[i]]
z[i] = t[y2[i]]

rev_f = rev_diff(f)
```
When the primal code reads from a thread local variable, we don’t need atomic add.

```python
@simd
def f(x : In[Array[float]],
y0 : In[Array[int]],
y1 : In[Array[int]],
y2 : In[Array[int]],
z : Out[Array[float]]):
i : int = thread_id()
t : Array[float, 10]
t[y0[i]] = x[y1[i]]
z[i] = t[y2[i]]
rev_f = rev_diff(f)

@simd
def rev_f(…):
i : int = thread_id()
t : Array[float, 10]
dt : Array[float, 10]
t[y0[i]] = x[y1[i]]
z[i] = t[y2[i]]
dt[y2[i]] = dt[y2[i]] + dz[i]
atomic_add(dx[y1[i]], dt[y0[i]])
```
When all threads in primal code read from a single variable, parallel reduction is usually better.

```python
@simd
def par_mul(x : In[Array[float]],
            y : In[float],
            z : Out[Array[float]]):
    i : int = thread_id()
    z[i] = x[i] * y

rev_par_mul = rev_diff(par_mul)
```

```python
@simd
def rev_par_mul(x : In[Array[float]],
                dx : Out[Array[float]],
                y : In[float],
                dy : Out[float],
                dz : In[Array[float]]):
    i : int = thread_id()
    # z[i] = x[i] * y
    dx[i] = dx[i] + dz[i] * y
    atomic_add(dy, dz[i] * x[i])
```
When all threads in primal code read from a single variable, parallel reduction is usually better.

```python
@simd
def rev_par_mul(x : In[Array[float]],
dx : Out[Array[float]],
y : In[float],
dy : Out[float],
dz : In[Array[float]],
tmp : InOut[Array[float]]):
    i : int = thread_id()
    # z[i] = x[i] * y
    dx[i] = dx[i] + dz[i] * y
    tmp[i] = dz[i] * x[i]
    for (s=n/2; s>0; s/=2):
        if i < s:
            tmp[i] += tmp[i+s]
        syncthreads()
    if i == 0:
        dy = tmp[0]
```

![Diagram of thread operations](image)
Parallel reduction vs atomic add

However, for the general case, the optimal strategy is data-dependent.

Let’s use (differentiable) ray tracing as an example. When computing derivatives w.r.t. triangle position, many threads will write into the same memory address.
However, for the general case, the optimal strategy is data-dependent

computing derivatives w.r.t. position of each triangle
However, for the general case, the optimal strategy is data-dependent.

- Computing derivatives w.r.t. position of each triangle.
- RGB image vs. # of memory writes (log scale).
- High thread contention, should do parallel reduction.
- Low thread contention, should do atomicAdd.
- No universal strategy!
However, for the general case, the optimal strategy is data-dependent computing derivatives w.r.t. position of each triangle

moral of the story: should provide options for the user to decide which strategy to use!
In some cases, we can completely eliminate race conditions through rewrites

\[
\text{out}[i] = \text{in}[i - j] \times k[j]
\]

consider the case of 1D convolution
In some cases, we can completely eliminate race conditions through rewrites

consider the case of 1D convolution

\[
\text{out}[i] = \text{in}[i - j] * k[j]
\]

\[
\text{d_in}[i - j] += \text{d_out}[i] * k[j]
\]

der here, a gathering operation becomes a scattering operation
In some cases, we can completely eliminate race conditions through rewrites.

Consider the case of 1D convolution:

\[
\text{out}[i] = \text{in}[i - j] \times k[j]
\]

\[
\text{d_in}[i - j] += \text{d_out}[i] \times k[j]
\]

Solution: turn the scattering operation into a gathering operation.

No race condition.
In some cases, we can completely eliminate race conditions through rewrites.

Consider the case of 1D convolution:

\[ \text{out}[i] = \text{in}[i - j] \times k[j] \]

\[ \text{d_in}[i - j] += \text{d_out}[i] \times k[j] \]

\[ \text{d_in}[u] += \text{d_out}[u+j] \times k[j] \]

(let \( u = i - j \) -> solve -> \( i = u + j \))

Solution: turn the scattering operation into a gathering operation.

No race condition.
Works for things other than convolution

let $u = x + y$, $v = y$

$\text{out}[x, y] = \text{in}[x + y, y]$

$\Rightarrow \text{d\_in}[u, v] += \text{d\_out}(u - v, v)$
Works for things other than convolution

```
let u = x + y, v = y
out[x, y] = in[x + y, y] -> d_in[u, v] += d_out(u - v, v)

let u = int(x / 4)
out[x] = in[x/4] -> for j = 0 to 4:
    d_in[u] += d_out[4 * u + j]
```

Works for things other than convolution
Let’s go back to the parallel reduction code

```c
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
        syncthreads()
    }
}
```
Let’s go back to the parallel reduction code

```cpp
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
    }
}
```

complex loop condition
non-trivial array indexing
side effect
Let’s go back to the parallel reduction code

```c
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
    }
}
```

complex loop condition

non-trivial array indexing

side effect

barrier

most AD compilers probably would generate very complicated reverse mode code!
Let’s go back to the parallel reduction code

```
parallel_for(i) {
    for (s=n/2; s>0; s/=2) {
        if (i < s)
            x[i] += x[i+s]
    }
}
```

complex loop condition  
non-trivial array indexing  
side effect  
barrier  

most AD compilers probably would generate very complicated reverse mode code!

however, the analytical derivative is actually simple:

```
dx[i] += dx[0]
```
Parallel AD: it’s often better to differentiate high-level constructs

\[ y = \sum_{i} x_i \]

\[ \downarrow \]

\[ dx_i = dy \]
Parallel AD: it’s often better to differentiate high-level constructs and *schedule* the low-level implementation.

\[ y = \sum_{i} x_i \]

\[ dx_i = dy \]

A key idea in my PhD thesis: