Automatic Differentiation for Adjoint Stencil Loops

Jan Hückelheim\textsuperscript{1}  Navjot Kukreja\textsuperscript{1}  Sri Hari Krishna Narayanan\textsuperscript{2}
Fabio Luporini\textsuperscript{1}  Gerard Gorman\textsuperscript{1}  Paul Hovland\textsuperscript{2}

October 3, 2019

\textsuperscript{1}Imperial College London, UK
\textsuperscript{2}Argonne National Laboratory, USA
• Automatic Differentiation (AD)
• AD for parallel programs
• Stencil loops
• Our work: AD for stencil loops
Automatic differentiation (AD)

- Given a program ("primal") that implements some function

\[ J = F(\alpha), \]

- AD generates a new program that implements its derivative.
Why would we want AD?

• Example: A fluid dynamics code that computes pressure loss in a pipe, subject to pipe geometry.

• AD computes derivative of pressure loss wrt. design parameters.

• We can automatically modify shape to minimise pressure loss

• Applications: Engineering optimisation, Imaging, Machine learning, ...
AD approaches

There are many ways of implementing AD:

**Source-to-source transformation**

- Creates code that computes partial derivative of each operation, and assembles them with chain-rule.
- Fast, efficient, but hard to get right. Mainly Fortran/C

**Operator overloading**

- Trace the computation at runtime, compute adjoints based on trace. Slow, huge memory footprint, easy to implement. Works for most high-level languages.

**High level, manual or automated**

- Start with problem definition, derive adjoint problem, implement the adjoint code separately.
Algorithmic differentiation (AD)

There are two fundamentally different modes:

**Tangent mode, Forward mode**

- Computes the Jacobian-vector product
  
  \[ \dot{J} = (\nabla F(x)) \cdot \dot{\alpha}. \]

- Derivatives are propagated along with the original computation.

**Adjoint mode, Reverse mode, backpropagation**

- Computes the transpose Jacobian-vector product
  
  \[ \bar{\alpha} = (\nabla F(x))^T \cdot \bar{J}. \]

- Path through original computation is traced, derivatives are propagated in reverse order.
Forward vs. reverse

- Tangent mode is simple to understand and implement, but: Need to re-run for every input.
- Adjoint mode is cheaper for many inputs and few outputs (run once, get all directional derivatives).
Challenge: derivative parallelisation in reverse mode

• If a shared memory region is read concurrently in original program, then the corresponding derivative will be updated concurrently.
• We can only easily parallelise adjoint if primal had exclusive read access.
• How can we detect this?
• What can we do otherwise?

* Förster (2014): Algorithmic Differentiation of Pragma-Defined Parallel Regions: Differentiating Computer Programs Containing OpenMP
Exclusive read access examples

- Do these loops have exclusive read access?

  ! Example loop 1

  real, dimension(10) :: b,c

  !$omp parallel do
do i=1,10
   do i=1,10
     b(i) = sin(c(i))
   end do
end do

- Answer: Yes
• Do these loops have exclusive read access?

! Example loop 2:

```fortran
real :: a
real, dimension(10) :: b, c

!$omp parallel do
do i=1,10
  b(i) = a+c(i)
end do
```

• Answer: No
Exclusive read access examples

• Do these loops have exclusive read access?

    ! Example loop 3:

    real, dimension(10) :: b, c
    integer, dimension(10) :: neigh
    call read_from_file(neigh)

    !$omp parallel do
    do i=1,10
        b(i) = c(neigh(i))
    end do

• Answer: Depends on file contents
Solutions?

• Detecting exclusive read access is impossible in general
• Without exclusive read access, we must pay a price:
  • Use reductions (extra memory)
  • Use atomics (extra time)
  • Some combination
• Can we do better in special cases?
Figure 1: AD on a gather produces a scatter
The Stencil is originally a gather operation

```c
#pragma omp parallel for private(i)
for ( i=1; i<=n - 1; i++ ) {
    r[i] = c[i]*(2.0*u[i-1]-3.0*u[i]+4*u[i+1]);
}
```
AD converts it to a scatter

```c
for ( i=1; i<=n-1; i++ ) {
    ub[i-1] += 2.0 * c[i] * rb[i];
    ub[i] -= 3.0 * c[i] * rb[i];
    ub[i+1] += 4.0 * c[i] * rb[i];
}
```
The scatter can be split into individual updates

```plaintext
for ( i=1; i<=n-1; i++ ) {
    ub[i-1] += 2.0 * c[i] * rb[i];
}
for ( i=1; i<=n-1; i++ ) {
    ub[i] -= 3.0 * c[i] * rb[i];
}
for ( i=1; i<=n-1; i++ ) {
    ub[i+1] += 4.0* c[i] * rb[i];
}
```
1D Stencil Example

```plaintext
Shift indices to write to loop counter element

```for ( j=0; j<=n-2; j++ ) {
    ub[j] += 2.0 * c[j+1] * rb[j+1];
}
```for ( j=1; j<=n-1; j++ ) {
    ub[j] -= 3.0 * c[j] * rb[j];
}
```for ( j=2; j<=n; j++ ) {
    ub[j] += 4.0 * c[j-1] * rb[j-1];
}
```
#pragma omp parallel for private(j)
for ( j=2; j<=n-2; j++ ) {
    ub[j] += 2.0 * c[j+1] * rb[j+1];
    ub[j] -= 3.0 * c[j] * rb[j];
    ub[j] += 4.0 * c[j-1] * rb[j-1];
}
ub[0] += 2.0 * c[1] * rb[1];
// ... other remainders: ub[1], ub[n-1], ub[n]
Higher dimensions

In higher dimensions, we need remainders for edges and corners
Figure 2: Speedups for the wave equation solver on a Broadwell processor, using up to 12 threads. The conventional adjoint code with manual parallelisation does not scale at all. The primal and PerforAD-generated adjoint benefit from using all 12 cores.
Figure 3: Absolute runtimes for wave equation primal and adjoint stencils and conventional adjoints in serial, as well as best observed primal and adjoint stencil run time in parallel. The best-observed performance of adjoint stencils was with 12 threads and is faster than the conventional adjoint by a factor of $3.4 \times$. 
• We release tool with this paper to generate these loop nests
  • https://github.com/jhueckelheim/PerforAD

```python
import sympy as sp; import perforad

# Define symbols

c = sp.Function("c")
u_1 = sp.Function("u_1"); u_1_b = sp.Function("u_1_b")
u_2 = sp.Function("u_2"); u_2_b = sp.Function("u_2_b")
i,j,k,D,n = sp.symbols("i,j,k,D,n")

# Build stencil expression

u_xx = u_1(i-1) - 2*u_1(i) + u_1(i+1)
expr = 2.0*u_1(i) - u_2(i) + c(i)*D*u_xx
lp = perforad.makeLoopNest(lhs=u(i), rhs=expr, counters = [i], bounds={i:[1,n-2]})
perforad.printfunction(name="wave1d_perf_b", loopnestlist=lp.diff({u:u_b, u_1:u_1_b, u_2: u_2_b}))
```
PerforAD-generated adjoint stencils preserve scalability of original program.

Paper discusses differentiation and code generation in more detail.

We also discuss reproducibility and floating point associativity.

See paper for full details, and runtimes on KNL.

Future work:

- Explore other code generation strategies (e.g. fewer remainder loops, but with branches)
- ML workloads
- SIMD and GPU programs
- Explore other polyhedral transformations in AD context
Thank you

Questions?