Towards Polyhedral Automatic Differentiation

Jan Hückelheim[1,2] Navjot Kukreja[1]

December 14, 2019

[1] Imperial College London, UK
[2] Argonne National Laboratory, USA

Recap: Automatic differentiation (AD)

AD modes

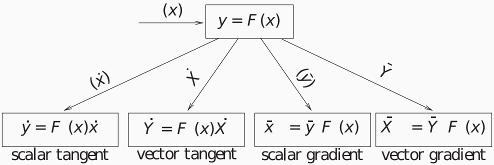


Figure 3.1: Basic Calculations of Tangents and Gradients

Andreas Griewank, Andrea Walther: Evaluating Derivatives

Forward or reverse?

- Infinitely many ways to implement primal, tangent, gradient
- Some of them are more useful than others
- Success story of AD: take inspiration from given program, which is hopefully a reasonable implementation of F
- In this work: Derive efficient gradient/tangent from efficient primal?

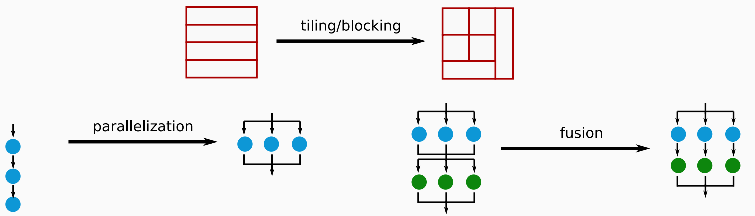
Motivation: Efficiency

”End of Moore’s law”

- Serial performance not growing for the last decade
- Code does not get faster just by waiting a few years

How to compute more?

- Adapt to different processors (GPU, TPU, ...)
- Expose and use parallelism
- Use cache hierarchy well, e.g. tiling, cache blocking
- Minimize passes over memory, e.g. fusion



AD approaches

There are ways to categorize AD tools, for example:

High level

- ML frameworks, Halide, BLAS: define high-level operations, hide implementation details under the hood
- AD operates on high level of abstraction
- Problem: Limited expressiveness, someone needs to write gradient operators, composition of existing blocks is not always efficient

Low level

- Directly operate on low-level language (e.g. C)
- Very expressive, general
- Performance optimizations are not abstracted away, mixed into computation

Question:

How should Automatic Differentiation respond?

- Can we maintain correctness?
- Can we maintain performance?

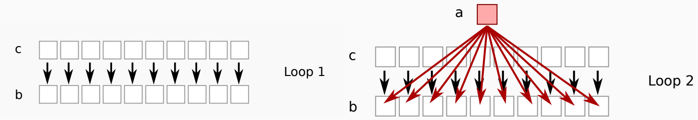
Question:

How should Automatic Differentiation respond?

- Can we maintain correctness?
- Can we maintain performance?

Something we can not do:

- Just re-use the primal parallelization



- In reverse-mode AD, shared read (ok) becomes shared write (not ok)

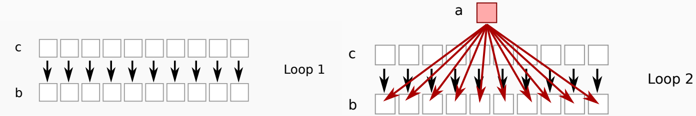
Question:

How should Automatic Differentiation respond?

- Can we maintain correctness?
- Can we maintain performance?

Something we can not do:

- Just re-use the primal parallelization



- In reverse-mode AD, shared read (ok) becomes shared write (not ok)

Another thing we can not do

- AD, then hand everything to optimizing compiler

Idea: Generate fast code inspired by primal

- Out of all possible codes, generate the one that closely mimics the primal
- Get as much information as possible from primal, to parallelize the derivative

Example: AD on a Stencil

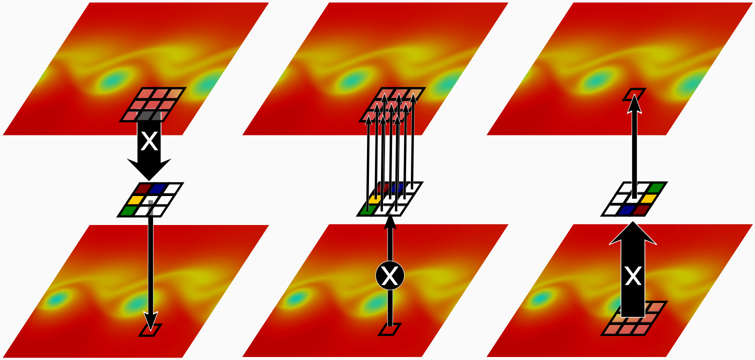
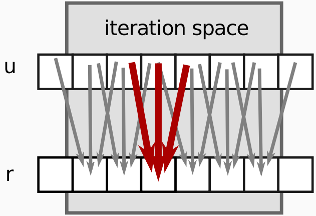


Figure 1: AD on a gather produces a scatter

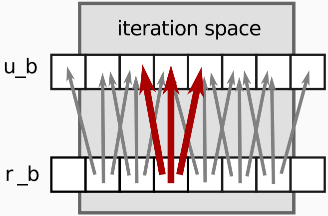
1D Stencil Example



The Stencil is originally a gather operation

```
#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]);
}
```

1D Stencil Example



AD converts it to a scatter

```
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];
}
```

Can we auto-optimize?

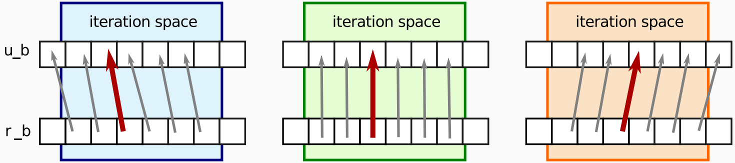
```
for ( i=1; i<=n-1; i++ ) {
  ub[i-1] += 2.0 * c[i] * rb[i];
  ub[i] -= 3.0 * c[i] * rb[i];
  ub[i+1] += 4.0 * c[i] * rb[i];
}
```

- Looked at in isolation, there are challenges:
- Is the trip count large enough to make parallelization profitable?
- Are ub, c, rb aliased?
- So many ways to transform this, which one is best?
- Would tiling help? What parameters are optimal?

PerforAD

- Prototype to generate gradient code that looks like primal code
- `https://github.com/jhueckelheim/PerforAD`
- Primal and gradient performance end up being similar
- Looks at loops in terms of iteration space, and statements
- We are free to restructure code, as long as statement is applied to same overall iteration space

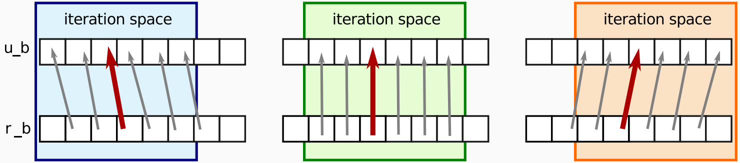
1D Stencil Example



The scatter can be split into individual updates

```
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];
}
```

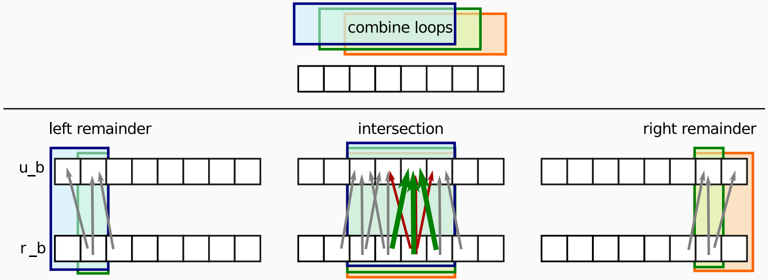
1D Stencil Example



Shift indices to write to loop counter element

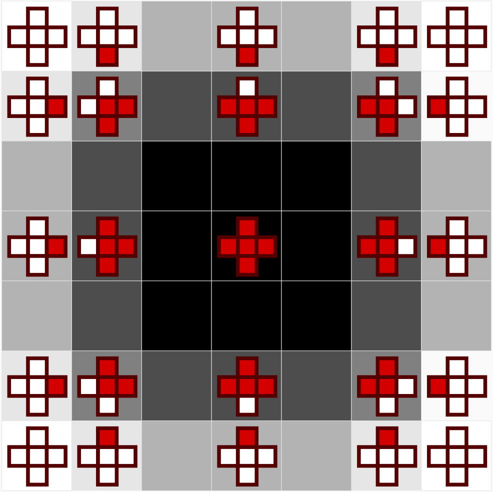
```
for ( j=0; j<=n-2; j++ ) {
  ub[j] += 2.0 * c[j+1] * rb[j+1];
}
for ( j=1; j<=n-1; j++ ) {
  ub[j] -= 3.0 * c[j] * rb[j];
}
for ( j=2; j<=n; j++ ) {
  ub[j] += 4.0 * c[j-1] * rb[j-1];
}
```

1D Stencil Example




```
#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

Performance Results - Scalability

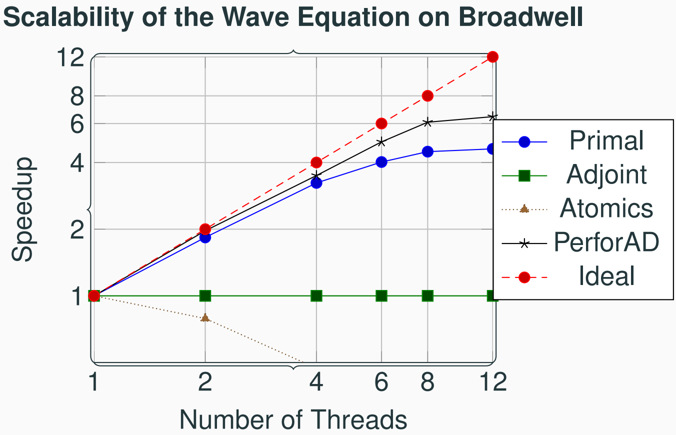
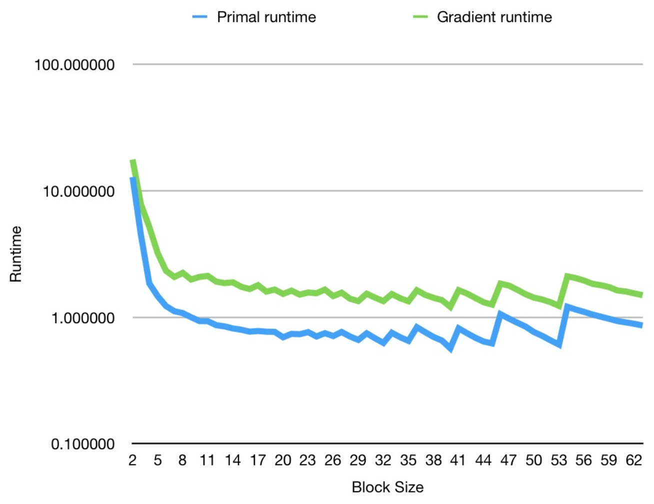


Figure 2: Speedups for the wave equation solver on a Broadwell processor, using up to 12 threads. The conventinal adjoint code with manual parallelisation does not scale at all. The primal and PerforAD-generated adjoint benefit from using all 12 cores.

Other optimizations

- Good block sizes for primal and gradient are related. This should be leveraged



Conclusion, Future Work

- We can automatically borrow ideas from primal to speed up gradient
- Can also use this for reproducibility, roundoff
- We have a paper: https://dl.acm.org/citation.cfm?doid=3337821.3337906
- Future work:
  - Try this with more examples
  - Try this with more diverse transformations
  - Need a better API to make this useful for more people

Thank you

Thank you

Questions?

References i