Algorithmic Differentiation of Structured Mesh Applications

Gábor Dániel Balogh
Supervisor: dr. István Reguly

Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

October 20, 2020

Algorithmic Differentiation - AD

- Algorithmic Differentiation is used to evaluate derivatives of the function which defined by a computer program

Algorithmic Differentiation - AD

- Algorithmic Differentiation is used to evaluate derivatives of the function which defined by a computer program

Why AD?

- Numerical methods require derivatives
- There are three main ways of automating computation of derivatives
  - Finite differentiation - slow for high dimension, lower accuracy
  - Symbolic differentiation - cannot handle some programming constructs
  - Algorithmic Differentiatoin - exact solution, fast

Algorithmic Differentiation - AD

Our program: $f : \mathbb{R}^n \to \mathbb{R}^m$, from som input $u \in \mathbb{R}^n$ generates some output $w \in \mathbb{R}^m$

Algorithmic Differentiation - AD

Our program: $f : \mathbb{R}^n \to \mathbb{R}^m$, from som input $u \in \mathbb{R}^n$ generates some output $w \in \mathbb{R}^m$

Our goal is to get the Jacobian J (or a part of it):

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

Algorithmic Differentiation - AD

Our program: $f : \mathbb{R}^n \to \mathbb{R}^m$, from som input $u \in \mathbb{R}^n$ generates some output $w \in \mathbb{R}^m$

Our goal is to get the Jacobian J (or a part of it):

$$J_{ij} = \frac{\partial f_i}{\partial x_j}$$

But how to get there?

Algorithmic Differentiation - AD

Assume that we can write f as a composite of K functions:

$$f = f^K \circ f^{K-1} \circ \ldots \circ f^1$$

Algorithmic Differentiation - AD

Assume that we can write f as a composite of K functions:

$$f = f^K \circ f^{K-1} \circ \ldots \circ f^1$$

Then we can write the Jacobian as:

$$J = J_L \cdot J_{L-1} \cdot \ldots \cdot J_1$$

Algorithmic Differentiation - AD

Assume that we can write f as a composite of K functions:

$$f = f^K \circ f^{K-1} \circ \ldots \circ f^1$$

Then we can write the Jacobian as:

$$J = J_L \cdot J_{L-1} \cdot \ldots \cdot J_1$$

There are two mode of AD:

- Forward (tangent) mode computes $J \cdot u = J_L \cdot J_{L-1} \cdot \cdots \cdot J_1 \cdot u$, for $u \in \mathbb{R}^n$
- Backward (adjoint) mode computes $J^T \cdot w = J_1^T \cdot J_2^T \cdot \cdots \cdot J_K^T \cdot w$, for $w \in \mathbb{R}^m$

Adjoint mode AD

Backward (adjoint) mode computes $J^T \cdot w = J_1^T \cdot J_2^T \cdot \cdots \cdot J_K^T \cdot w$, for $w \in \mathbb{R}^m$

Use w such that the i^{th} element of w is 1, and the others are 0.

- $J^T \cdot w$ will produce the i^{th} row of J

Adjoint mode AD

Backward (adjoint) mode computes $J^T \cdot w = J_1^T \cdot J_2^T \cdot \cdots \cdot J_K^T \cdot w$, for $w \in \mathbb{R}^m$

Use w such that the i^{th} element of w is 1, and the others are 0.

- $J^T \cdot w$ will produce the i^{th} row of J

Evaluate it one step at a time

- Only need the derivative of one function of the chain
- If we choose the f_i decomposition carefully, we can implement AD efficiently

Adjoint mode AD

Backward (adjoint) mode computes $J^T \cdot w = J_1^T \cdot J_2^T \cdot \dots \cdot J_K^T \cdot w$, for $w \in \mathbb{R}^m$

Use w such that the i^{th} element of w is 1, and the others are 0.

- $J^T \cdot w$ will produce the i^{th} row of J

Evaluate it one step at a time

- Only need the derivative of one function of the chain
- If we choose the f_i decomposition carefully, we can implement AD efficiently
- But to get J_i we need the state of the program at f_i

Adjoint mode AD

Backward (adjoint) mode computes $J^T \cdot w = J_1^T \cdot J_2^T \cdot \dots \cdot J_K^T \cdot w$, for $w \in \mathbb{R}^m$

Commonly implemented with operator overloading

- f_i is an elementary operation, easy to compute $J_i^T \cdot w'$
- But we produce enormous chains, and need to store every state of every variable

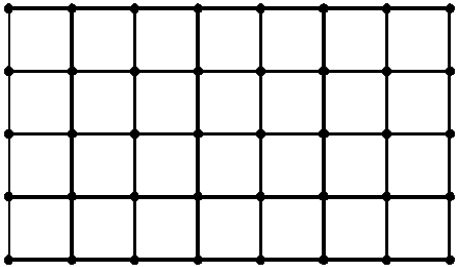
DSLs for future proof HPC applications

- Fast-changing hardware
  - infeasible to maintain code to support all of them
- Embedded Domain Specific Languages (eDSL) can hide platform specific details
  - future proof, perforamnce portable solutions for application developers

Oxford Parallel library for Structured mesh solvers

OPS (**O**xford **P**arallel library for **S**tructured mesh solvers)

- C++ library with high-level API calls for structured mesh applications
- High level concepts
  - grids
  - data on grids
  - loops over subgrid accessing data
- generate parallel implementations of loops for all hardware



OPS + AD

- Each loop that performs computation must be a call of `ops_par_loop`
  - takes the loop body as a function
  - descriptors of datasets: access type, stencil of access
- OPS generates the implementation for all `ops_par_loop`

OPS + AD

- Each loop that performs computation must be a call of `ops_par_loop`
  - takes the loop body as a function
  - descriptors of datasets: access type, stencil of access
- OPS generates the implementation for all `ops_par_loop`

If we have all these information, can we do AD?

OPS + AD

If we have all these information, can we do AD?

- OPS already has control over the sequence of parallel loops.

- If we have derivatives of these loops instead of the elementary operations we can evaluate the chain rule on the **loop level**.
  - We assume that either the user will supply the derivative or we can use source transformation to get it

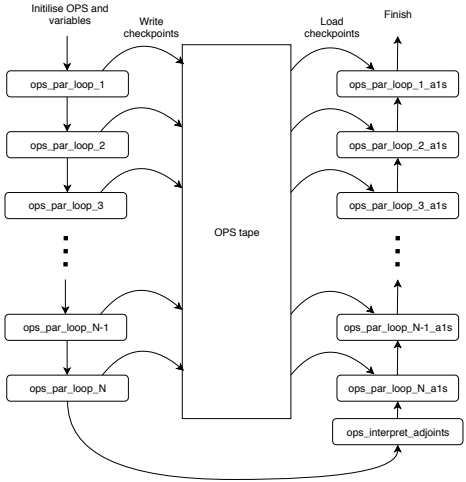
OPS + AD

If we have all these information, can we do AD?

- OPS already has control over the sequence of parallel loops.
- If we have derivatives of these loops instead of the elementary operations we can evaluate the chain rule on the **loop level**.
  - We assume that either the user will supply the derivative or we can use source transformation to get it
- Features missing to perform the backward sweep:
  - store intermediate states
  - reversing data flow inside loops

Intermediate state storage - tape

- Generated code registers loops and create copies of overwritten data to a data structure (tape).
- In the backward sweep we can execute the adjoints of each loop and load intermediate states of datasets.



Reversing data flow

The second problem is that we need to parallelise the adjoints of the stencil loops as well.

Figure 1: Example computational step in OPS given by the user (a) for compute results and (b) to compute the derivatives backwards

```
1 inline void mean_kernel(
2           const OPS_ACC<double> &u,
3           OPS_ACC<double> &u_2) {
4       u_2(0, 0) = (u(-1, 0) + u(1, 0)
5                  +  u(0, -1) + u(0, 1)) * 0.25;
6 }
```

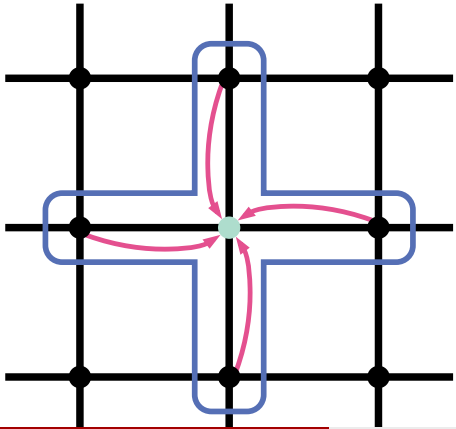
a: Compute the mean of neighbours for each grid point

```
1  inline void mean_kernel_adjoint(
2                              const OPS_ACC<double> &u,
3                              OPS_ACC<double> &u_a1s,
4                              const OPS_ACC<double> &u_2,
5                              OPS_ACC<double> &u_2_a1s) {
6       u_a1s(-1,0) += 0.25 * u_2_a1s(0, 0);
7       u_a1s(1, 0) += 0.25 * u_2_a1s(0, 0);
8       u_a1s(0,-1) += 0.25 * u_2_a1s(0, 0);
9       u_a1s(0, 1) += 0.25 * u_2_a1s(0, 0);
10 }
```

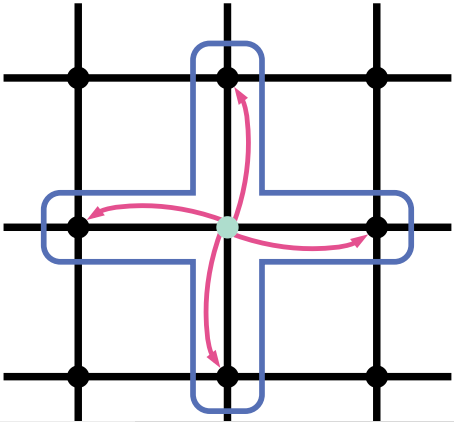
b: The corresponding adjoint kernel

Reversing data flow

Writing pattern in forward code.
No race condition allowed.

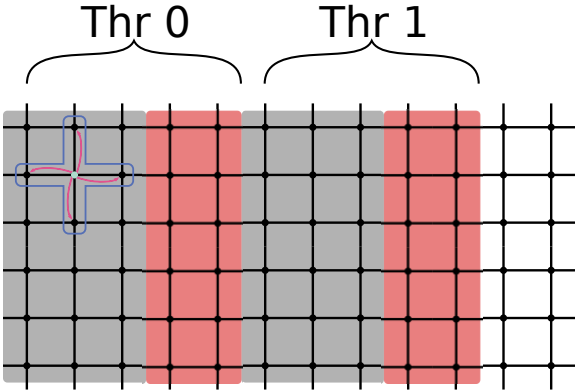


Reversed stencil for adjoints. Race conditions on each adjoint.



Avoiding race conditions

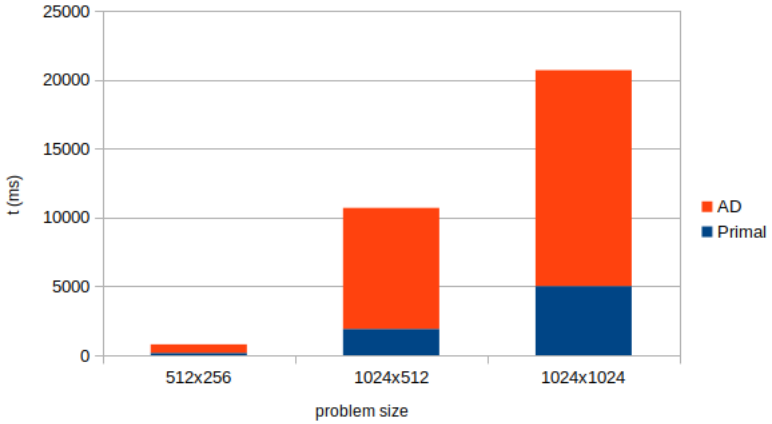
- CPU: Red black colouring creating red and black stripes for each thread



- CUDA: overloading operators to use atomics for adjoints

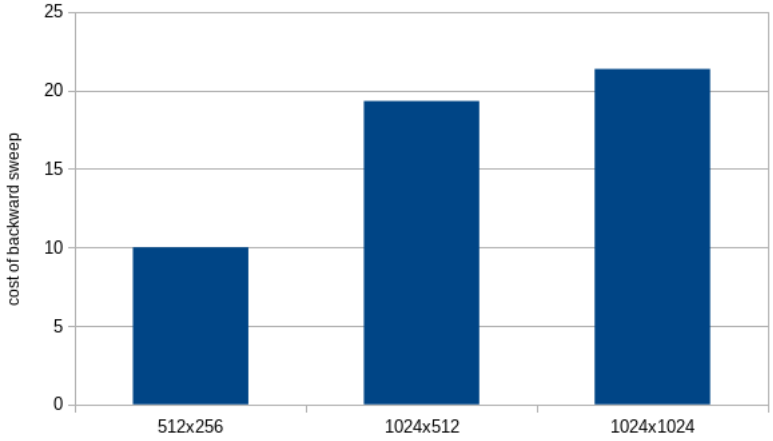
Performance: CPU

Our best performance on example apps produce derivatives under less then $6\times$ of the primal code, which is close to the theoretical optimum on a representative code from finance.



Performance: V100

Naive GPU implementation of the adjoint loops on typical problem sizes takes $10 - 25\times$ of the primal.



Current solution: Memory

Another problem of the current implementation is that checkpointing requires too much memory.

	Memory (GB)	With checkpointing (GB) iteration count		
Grid Size	primal	10	100	200
512×256	0.025	0.292	1.788	3.792
1024×512	0.094	0.892	6.902	12.90
1024×1024	0.188	1.946	13.04	26.04

Conclusions

We extended the OPS library with adjoint aware API.

- Successfully parallelised the computations of adjoints for CPUs and GPUs
- Showed promising runtime performance
- But the current implementation requires too much memory
  - Currently working on an implementation for recomputing loops

Supported by the ÚNKP-19-3-I New National Excellence Program of the Ministry for Innovation and Technology