Automated Batching and Differentiation of Scalar Code in Enzyme

Euro LLVM 2022
May 11, 2022
Overview

1. Compiler-based Automatic Differentiation
2. Design of Vector-Mode / Batching
3. Usage Examples
4. Benchmarks
5. Future Work
Compiler-based Automatic Differentiation
Forward- and Reverse-Mode

- **Forward-Mode**
  - Evaluation performed right to left
  - Information flow in same direction as computation, no caching required
  - Efficient for functions \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) where \( m \gg n \)

- **Reverse-Mode**
  - Evaluation performed left to right
  - Information flow in opposite direction as computation, elaborate caching required
  - Efficient for functions \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) where \( m \ll n \)

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial c} \left( \frac{\partial c}{\partial b} \left( \frac{\partial b}{\partial a} \frac{\partial a}{\partial x} \right) \right)
\]
Compiler-based Automatic Differentiation
The Advantages

• Ability to access and modify the program at any point during the compilation process
  • Ability to run optimizations before, and after differentiation
  • Identify source-line information from metadata
  • Rewrite & modify library calls
• Ability to run a JIT-compiler
Enzyme’s Approach
Optimize then Differentiate
Enzyme Frontends
Deeply Rooted in the LLVM Ecosystem

• Generic low-level compiler infrastructure with many frontends
  • “Cross-platform assembly”
  • Many Backends (CPU, CUDA, AMDGPU, ..)
• Well-defined semantics
• Large collection of optimizations and analyses which can be utilised within the automatic differentiation
- **Forward Mode** computes the derivative of $f$ with respect to one input
- **Reverse Mode** computes the gradients of $f$ for all inputs
Vector Mode AD
Forward Mode

- Vector Mode maps perfectly to SIMD
Vector Mode AD

Use cases

- Make optimal use of available hardware
- Forward-mode AD, compute derivatives for all inputs
- Reverse-mode AD, compute gradients with respect to multiple outputs
Vector Mode AD

Benefits

• Avoid re-evaluation of the original function
• Cache values from the original function only once
Vector Mode AD
Case study: Expensive call

- Runtime of $f$ mostly bound by call to slow
- Value of slow() takes place in derivative computation but does not propagate derivatives

```c
float f(float x[2]) {
    float sum = x[0] + x[1];
    float val = slow();
    sum *= val;
    return sum;
}
```

```c
float df(float x[2], float d_x[2]) {
    float d_sum = d_x[0] + d_x[1];
    float val = slow();
    d_sum *= val;
    return d_sum;
}
```
Vector Mode AD
Case study: Expensive call

float f(float x[2]) {
    float sum = x[0] + x[1];
    float val = slow();
    sum *= val;
    return sum;
}

float[2] df_batch(float x[2],
                  float d_x[2][2]) {
    float d_sum[2];
    d_sum[0] += d_x[0][0] + d_x[0][1];
    d_sum[1] += d_x[1][0] + d_x[1][1];
    float val = slow();
    d_sum[0] *= val;
    d_sum[1] *= val;
    return d_sum;
}
Vector Mode AD
Simulating reverse mode

• Compute gradient with respect to
  • \( x_1 \) by setting \( \mathbf{d} \mathbf{x} \) to \((1,0)\)
  • \( x_2 \) by setting \( \mathbf{d} \mathbf{x} \) to \((0,1)\)

```c
float[2] grad_f_batch(float x[2]) {
  float d_x[2][2] = {
    {1, 0},
    {0, 1}
  };
  return df_batch(x, d_x);
}
```

```c
float[2] df_batch(float x[2],
                   float d_x[2][2]) {
  float d_sum[2];
  d_sum[0] += d_x[0][0] + d_x[0][1];
  d_sum[1] += d_x[1][0] + d_x[1][1];
  float val = slow();
  d_sum[0] *= val;
  d_sum[1] *= val;
  return d_sum;
}
```
Vector Mode
Design

1. Perform **Activity Analysis**, determine which instructions are required to compute the derivatives
2. Perform **Type Analysis**

4. **Synthesis**: Propagate with each active value a vector of derivatives
5. For each pointer store a vector of shadow pointers
6. Create shadow allocations for each heap data structure

8. Emit instructions for the application of the chain rules n-times according to vector width
9. Let the LLVM vectoriser do the rest
Usage

LLVM IR

```llvm
declare [2 x double] @__enzyme_fwddiff(...)

define double @square(double %x) {
  %mul = fmul double %x, %x
  ret double %mul
}

define [2 x double] @dsquare(double %x, double %dx1, double %dx2) {
  %call = call [2 x double] (...), @__enzyme_fwddiff(double (double)* @square, metadata !"enzyme_width", i64 2, double %x, double %dx1, double %dx2)
  ret [2 x double] %call
}

define [2 x double] @dsquare(double %x, double %dx1, double %dx2) {
  %factor.i = fmul double %x, 2.0
  %1 = fmul double %factor.i, %dx1
  %2 = insertvalue [2 x double] undef, double %1, 0
  %3 = fmul double %factor.i, %dx2
  %4 = insertvalue [2 x double] %2, double %3, 1
  ret [2 x double] %4
}
```
Optimizations

- Activity analysis, doesn’t propagate derivatives for inactive values
- Combine calls to allocation / free functions

```c
float f(float x[2]) {
    float sum = x[0] + x[1];
    float val = slow(); // inactive
    sum *= val;
    return sum;
}
```
Vectorizing LLVM types
Type Trees

Vectorized at the root node  [2 x { double*, float, [3 x double], %struct.Foo }]

Vectorized at the leaf nodes  { [2 x double]*, [2 x float], [3 x [ 2 x double]], %struct.Foo2 }
Vectorizing LLVM types

Discussion

• Vectorisation of the root node
  • Better compatibility, no modification of existing structs necessary

• Vectorisation of leaf nodes
  • Better memory locality, vectorized loads and stores
# Benchmarks

A selection of AD Tools

<table>
<thead>
<tr>
<th></th>
<th>Forward Mode</th>
<th>Reverse Mode</th>
<th>Forward Vector Mode</th>
<th>Reverse Vector Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enzyme</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Tapenade</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>CoDiPack</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
<tr>
<td>Clad</td>
<td>✅</td>
<td>✅</td>
<td>❌</td>
<td>❌</td>
</tr>
<tr>
<td>Adept</td>
<td>✅</td>
<td>✅</td>
<td>❌</td>
<td>❌</td>
</tr>
</tbody>
</table>
• Vector width of 1 to 64
• Problem too big to vectorize at once
• Up to 4x speedup, due to activity analysis
Benchmarks
AD Bench - GMM

- Vector width of 1 to 32
- Problem too big to vectorize at once
- Scaling issues beyond vector width of 8
Benchmarks
LIBOR

• Vector width = problem size
• 46% speedup over Tapenade
Future Work

• Dynamic vector width
• Type tree vectorized at the leaf nodes
• Support for OpenMP & MPI parallelism
• Support for GPUs (PTX, ROCm)
Summary

• Tool for performing reverse and forward-mode AD of statically analyzable LLVM IR

• Differentiates code in languages going to the LLVM IR (C, C++, Fortran, Julia, Rust, Swift, Haskell, etc) and parallel frameworks (OpenMP, MPI, CUDA, ROCm, Julia Threads)

• Work in progress

• Open source:  https://github.com/EnzymeAD/Enzyme

• Compiler Explorer:  https://enzyme.mit.edu/explorer
Questions