Automated Derivative Sparsity via Dead Code Elimination

Kevin Mu

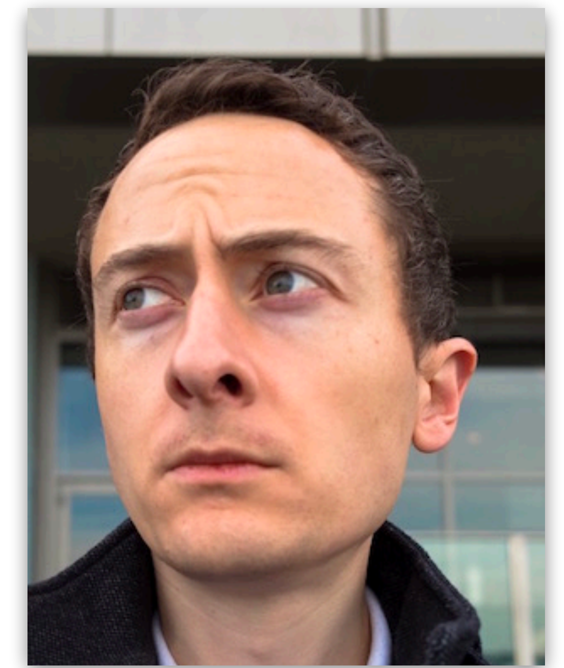
Jessie Michel

William S. Moses

Shoaib Kamil

Zachary Tatlock

Alec Jacobson



Jonathan Ragan-Kelley

wsmoses@illinois.edu

Winter EuroAD 2023

Sparsity is Common in AD

- Computing the Jacobian with Forward or Reverse-mode AD

$$J_{fwd}(\vec{x}) = \begin{pmatrix} f_{fwd}(\vec{x}, [1, 0, 0, ...]) \\ f_{fwd}(\vec{x}, [0, 1, 0, ...]) \\ f_{fwd}(\vec{x}, [0, 0, 1, ...]) \end{pmatrix}$$

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$J_{rev}(\vec{x}) = \begin{pmatrix} f_{rev}(\vec{x}, [1, 0, ...]) \\ f_{rev}(\vec{x}, [0, 1, ...]) \end{pmatrix}^T$$

Sparsity is Common in (High-Order) AD

- Computing the Hessian

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$\begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

Sparsity is Common in (High-Order) AD

- Computing the Hessian

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$\begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

```
void hessian(double* in, double* outputs) {
  for(int i=0; i<n; I++) {
   fwddiff(revdiff(f),
           in, &outputs[i * n]);
  }
}
```

Sparsity is Common in (High-Order) AD

- Computing the Hessian

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$\begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

```
void hessian(double* in, double* outputs) {
  for(int i=0; i<n; I++) {
   fwddiff(revdiff(f),
          in, &outputs[i * n]);
  }
}
```

$O(n)$

Sparsity is Common in (High-Order) AD

- Computing the Hessian

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$\begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

```
void hessian(double* in, double* outputs) {
  for(int i=0; i<n; I++) {
   fwddiff(revdiff(f),
          in, &outputs[i * n]);
  }
}
```

$O(n)$

$O\left(n^2\right)$

Sparsity is Common in (High-Order) AD

- Computing the Hessian

$$f(\vec{x}) = \sum_i a_i x_i^2$$

$$\begin{pmatrix} 2a_0 & 0 & 0 & \dots & 0 \\ 0 & 2a_1 & 0 & \dots & 0 \\ 0 & 0 & 2a_2 & \dots & 0 \\ 0 & 0 & 0 & \dots & 2a_n \end{pmatrix}$$

$$O\left(n^2\right)$$

```
void hessian(double* in, double* outputs) {
  for(int i=0; i<n; I++) {
   fwddiff(revdiff(f),
          in, &outputs[i * n]);
  }
}
```

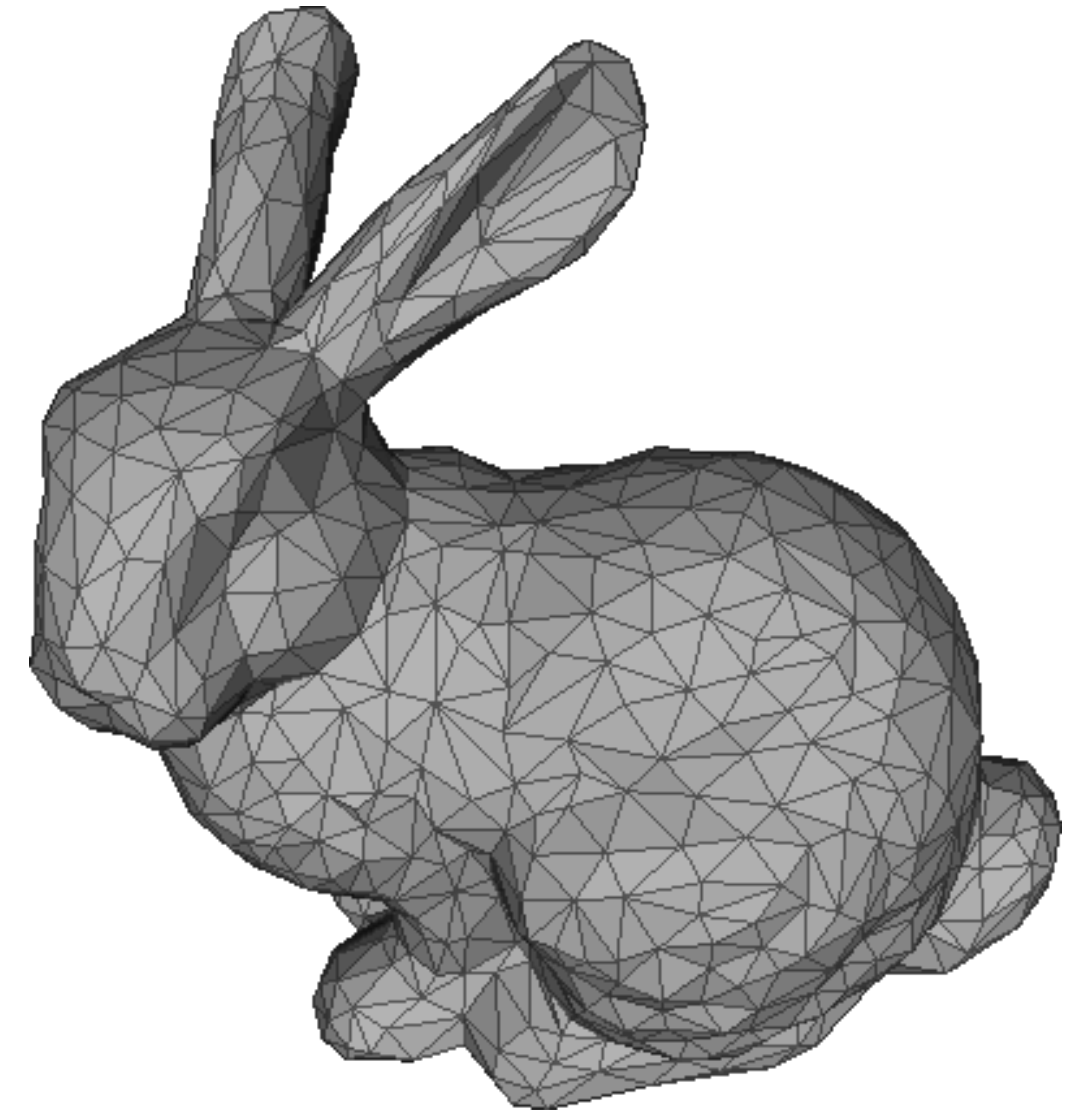
$O(n)$

$O\left(n^2\right)$

Sparsity is Common in Graphics

- Many graphics problems are decomposed into a list of faces/edges

```
def f(input):
  sum = 0
  for shape in shapes(input):
   sum += local(shape)
  return sum
```

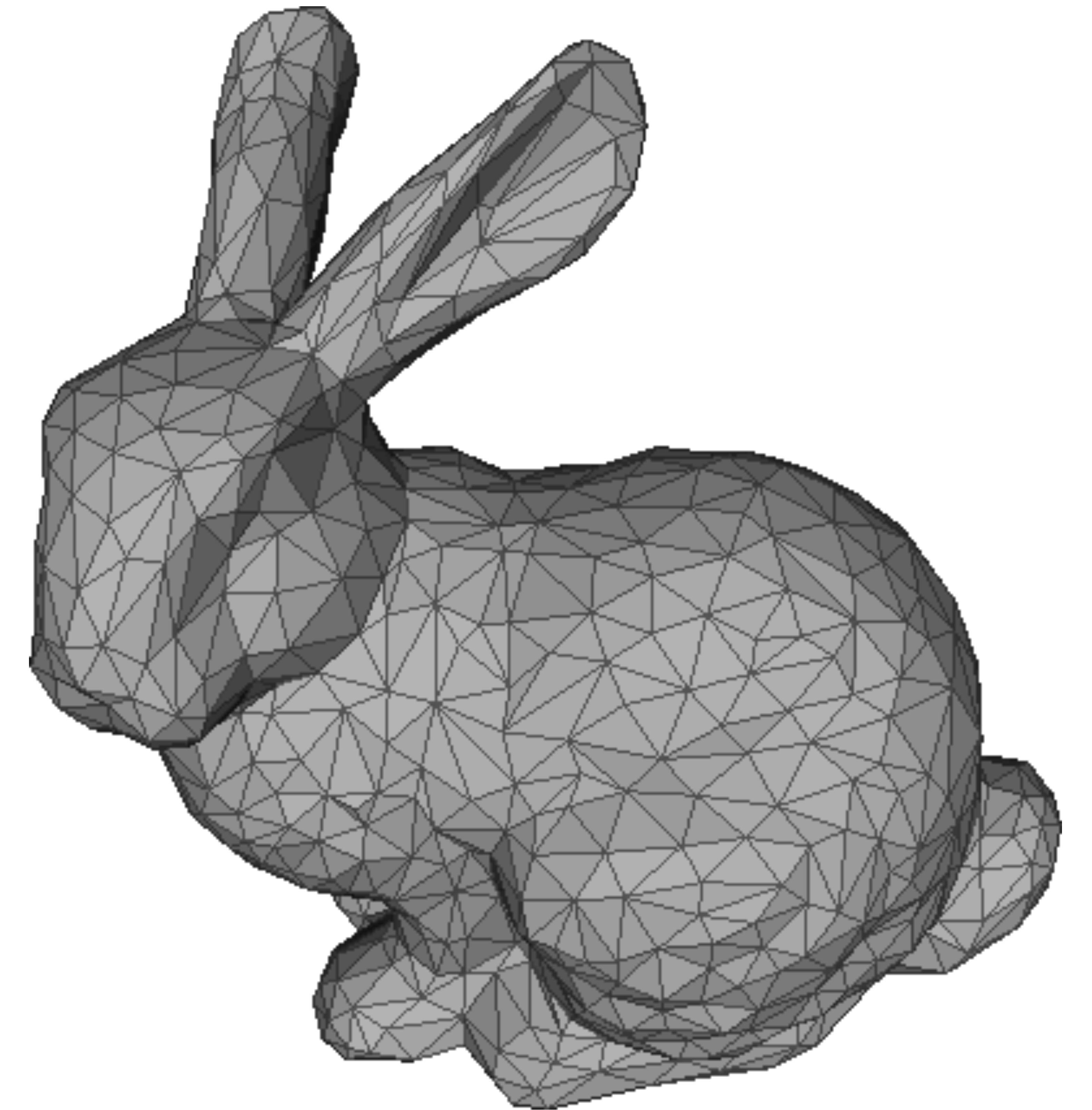


Sparsity is Common in Graphics

- Many graphics problems are decomposed into a list of faces/edges

```
def f(input):
  sum = 0
  for shape in shapes(input):
   sum += local(shape)
  return sum
```

$$H = \begin{pmatrix} \begin{pmatrix} s_0(0,0) & s_0(0,1) \\ s_0(1,0) & s_0(1,1) \\ 0 & \end{pmatrix} & 0 & 0 & \ldots \\ 0 & \begin{pmatrix} s_1(0,0) & s_1(0,1) \\ s_1(1,0) & s_1(1,1) \end{pmatrix} & 0 & \ldots \\ 0 & 0 & \begin{pmatrix} s_2(0,0) & s_2(0,1) \\ s_0(1,0) & s_0(1,1) \end{pmatrix} & \ldots \\ 0 & 0 & \ldots & \begin{pmatrix} s_n(0,0) & s_n(0,1) \\ s_n(1,0) & s_n(1,1) \end{pmatrix} \end{pmatrix}$$



Can we ***automatically*** leverage sparsity?

Prior Work: TinyAD

- Rewrite the problem to explicitly iterate over shape
- Library performs block-wise sparse differentiation.

```cpp
// Set up a function with 2D vertex positions as variables
auto func = TinyAD::scalar_function<2>(mesh.vertices());

// Add an objective term per triangle. Each connecting 3 vertices
func.add_elements<3>(mesh.faces(), [&] (auto& element)
{
    // Element is evaluated with either double or TinyAD::Double<6>
    using T = TINYAD_SCALAR_TYPE(element);

    // Get variable 2D vertex positions of triangle t
    OpenMesh::SmartFaceHandle t = element.handle;
    Eigen::Vector2<T> a = element.variables(t.halfedge().to());
    Eigen::Vector2<T> b = element.variables(t.halfedge().next().to());
    Eigen::Vector2<T> c = element.variables(t.halfedge().from());

    return ...
});

// Evaluate the funcion using any of these methods:
double f = func.eval(x);
auto [f, g] = func.eval_with_gradient(x);
auto [f, g, H] = func.eval_with_derivatives(x);
auto [f, g, H_proj] = func.eval_with_hessian_proj(x);
```

Eurographics Symposium on Geometry Processing 2022
M. Campen and M. Spagnuolo
(Guest Editors)

Volume 41 (2022), Number 5

TinyAD:
Automatic Differentiation in Geometry Processing Made Simple

P. Schmidt[1] J. Born[1] D. Bommes[2] M. Campen[3] L. Kobbelt[1]

[1]RWTH Aachen University, Germany [2]University of Bern, Switzerland [3]Osnabrück University, Germany

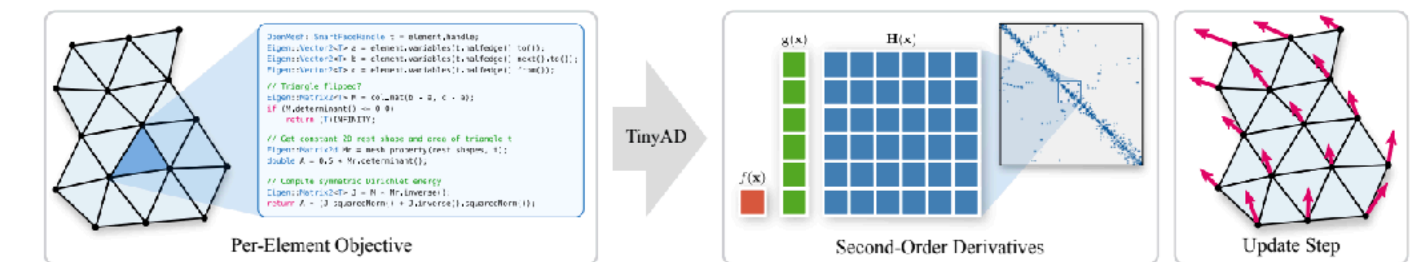


Figure 1: *TinyAD offers automatic differentiation targeted at geometry processing problems on meshes. Left: A user implements an objective function in C++ using the Eigen library (here per triangle, but arbitrary mesh primitives or stencils can be chosen as elements). Center: TinyAD efficiently computes second-order derivatives per element and automatically assembles the global gradient and sparse Hessian matrix. Right: The user computes an update step (here per vertex) using a standard linear solver. Via this simplified access to derivatives, we enable quick exploration of complex objectives in research workflows.*

Abstract

Non-linear optimization is essential to many areas of geometry processing research. However, when experimenting with different problem formulations or when prototyping new algorithms, a major practical obstacle is the need to figure out derivatives of objective functions, especially when second-order derivatives are required. Deriving and manually implementing gradients and Hessians is both time-consuming and error-prone. Automatic differentiation techniques address this problem, but can introduce a diverse set of obstacles themselves, e.g. limiting the set of supported language features, imposing restrictions on a program's control flow, incurring a significant run time overhead, or making it hard to exploit sparsity patterns common in geometry processing. We show that for many geometric problems, in particular on meshes, the simplest form of forward-mode automatic differentiation is not only the most flexible, but also actually the most efficient choice. We introduce TinyAD: a lightweight C++ library that automatically computes gradients and Hessians, in particular of sparse problems, by differentiating small (tiny) sub-problems. Its simplicity enables easy integration; no restrictions on, e.g., looping and branching are imposed. TinyAD provides the basic ingredients to quickly implement first and second order Newton-style solvers, allowing for flexible adjustment of both problem formulations and solver details. By showcasing compact implementations of methods from parametrization, deformation, and direction field design, we demonstrate how TinyAD lowers the barrier to exploring non-linear optimization techniques. This enables not only fast prototyping of new research ideas, but also improves replicability of existing algorithms in geometry processing. TinyAD is available to the community as an open source library.

CCS Concepts
• ***Mathematics of computing*** → *Automatic differentiation; Mathematical software;*
• ***Computing methodologies*** → *Mesh models;*

Can we leverage sparsity to reduce compute & memory?

Spadina

- Intersperse compiler optimizations and program optimizations with differentiation to find and leverage structural sparsity
- Implementation in Enzyme and JaX
- Applicable to general structural sparsity (Hessian, Jacobian, etc)

Present Requirements

- Accumulation of loop(s) of fixed size (e.g. no linked list, runtime-dependent values)
- Only cross-iteration dependencies are linear (e.g. sums)
- All inputs are recomputable [aka read-only]
- Induction variables are not needed for derivative computations [except indexing]
- Memory accesses are affine-indexed (e.g. `input[i + 10 * j]` not `input[I*i]`)
- May be able to relax these in future (some (like read-only) are required by JaX implementation, though not by Enzyme :))

Spadina Algorithm

1. Perform Dense AD, interspersed with optimization

2. Propagate sparsity from "one-hot" identity inputs

3. Extract structural derivative updates

```
double* X = __enzyme_todense(my_load, my_store, n);

print(X[I])

X[I] = 7;
```

```
double* X = __enzyme_todense(my_load, my_store, n);

print(my_load(I, n));

store(7, I, n));
```

Sparse AD Step 1: Dense AD

```
double f(double* in) {
  double loss = 0;
  for(int i=0; i<n-1; i++) {
   loss += ( in[i - 1] - in[I] ) ** 2;
  }
}
```

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
     +[](double* in, double* out) {
       __enzyme_autodiff(f, in, out);
     },
     enzyme_dup, in, &identity[i * n],
     enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

- ✓Accumulation of fixed-size loop(s)
- ✓Only linear cross-iteration dependencies
- ✓Read-only inputs
- ✓Induction variables are not needed for derivative
- ✓Affine memory accesses

Sparse AD Step 1: Dense AD

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
      +[](double* in, double* out) {
        __enzyme_autodiff(f, in, out);
      },
      enzyme_dup, in, &identity[i * n],
      enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

```
double f(double* in) {
  double loss = 0;
  for(int i=0; i<n-1; i++) {
   loss += ( in[i - 1] - in[I] ) ** 2;
  }
}
```

Sparse AD Step 1: Dense AD

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
      +[](double* in, double* out) {
        __enzyme_autodiff(f, in, out);
      },
      enzyme_dup, in, &identity[i * n],
      enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

```
double f(double* in) {
  double loss = 0;
  for(int i=0; i<n-1; i++) {
   loss += ( in[i - 1] - in[I] ) ** 2;
  }
}
```

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
      grad_f,
      enzyme_dup, in, &identity[i * n],
      enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

```
double grad_f(double* in, double* din) {
  double loss = 0;
  for(int i=0; i<n-1; i++) {
   din[i - 1] += 2 * (in[i - 1] - in[i]);
   din[i] += 2 * (in[i - 1] - in[i]);
  }
}
```

Sparse AD Step 1: Dense AD

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
      grad_f,
      enzyme_dup, in, &identity[i * n],
      enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

```
double grad_f(double* in, double* din) {
  for(int i=0; i<n-1; i++) {
   din[i - 1] += 2 * (in[i - 1] - in[i]);
   din[i] += 2 * (in[i - 1] - in[i]);
  }
}
```

```
void hessian(double* in, double* outputs) {

  for(int i=0; i<n; i++)
   hess_f(
      enzyme_dup, in, &identity[i * n],
      enzyme_dupnoneed, nullptr, &outputs[i * n]);
}
```

```
double hess_f(double* in,  double* fin,
              double* din, double* fdin) {
  for(int i=0; i<n-1; i++) {
   fdin[i - 1] += 2 * (fin[i - 1] - fin[i]);
   fdin[i] += 2 * (fin[i - 1] - fin[i]);
  }
}
```

Step 2: Sparse Propagation

- In AD, the derivative result is always multiplied by the derivative input, e.g.
  d/dx (y = sin(x)) -> dy = cos(x) * dx

- Therefore all derivative results are multiplied by the original input, in this case a one-hot identity matrix encoding.

$$J_{fwd}(\vec{x}) = \begin{pmatrix} f_{fwd}(\vec{x}, [1, 0, 0, ...]) \\ f_{fwd}(\vec{x}, [0, 1, 0, ...]) \\ f_{fwd}(\vec{x}, [0, 0, 1, ...]) \end{pmatrix}$$

- Propagate this sparsity throughout the program to transform a dense AD code into a sparse AD code

```
y = sin(x)
z = f(y)
r = g(z)
```

```
dx = 1 if (i == j) else 0
dy = cos(x) * dx
dz = f’(y) * dy
dr = g’(z) * dz
```

```
dx = 1
dy = cos(x) * 1
dz = f’(y) * dy
dr = g’(z) * dz if (i == j) else 0
```

Sparse AD Step 2: Sparse Propagation

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++)
   __enzyme_fwddiff(
      +[](double* in, double* out) {
        __enzyme_autodiff(f, in, out);
      },
      enzyme_dup, in, __enzyme_todense(ident_load, ident_store, i, n),
      enzyme_dupnoneed, nullptr,
         __enzyme_todense(csr_load, csr_store, i, outputs, n));
}
```

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    double* fin = __enzyme_todense(ident_load, ident_store, I, n);
    double* dfin = __enzyme_todense(csr_load, csr_store, i, outputs, n);
    for(int j=0; j<n-1; j++) {
      fdin[j - 1] += 2 * (fin[j - 1] - fin[j]);
      fdin[j] += 2 * (fin[j - 1] - fin[j]);
    }
  }
}
```

Sparse AD Step 2: Sparse Propagation

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    double* fin = __enzyme_todense(ident_load, ident_store, I, n);
    double* dfin = __enzyme_todense(csr_load, csr_store, i, outputs, n);
    for(int j=0; j<n-1; j++) {
      fdin[j - 1] += 2 * (fin[j - 1] - fin[j]);
      fdin[j] += 2 * (fin[j - 1] - fin[j]);
    }
  }
}
```

```
void hessian(double* in, double* outputs) {
  for(int i=0; i<n; i++) {
    double* fin = __enzyme_todense(ident_load, ident_store, I, n);

    for(int j=0; j<n-1; j++) {
      csr_store(j - 1, 2 * (fin[j - 1] - fin[j]), outputs, n );
      csr_store(j    , 2 * (fin[j - 1] - fin[j]), outputs, n );
    }
  }
}
```

Sparse AD Step 2: Sparse Propagation

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    double* fin = __enzyme_todense(ident_load, ident_store, I, n);

    for(int j=0; j<n-1; j++) {
      csr_store(j - 1, 2 * (fin[j - 1] - fin[j]), outputs, n );
      csr_store(j    , 2 * (fin[j - 1] - fin[j]), outputs, n );
    }
  }
}
```

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
      csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
    }
  }
}
```

Sparse AD Step 2: Sparse Propagation

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
      csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
    }
  }
}
```

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      if ( 2 * ( ((j - 1) == i) - (j == i) ) != 0)
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
      if ( 2 * ( ((j - 1) == i) - (j == i) ) != 0)
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
    }
  }
}
```

Sparse AD Step 2: Sparse Propagation

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      if ( 2 * ( ((j - 1) == i) - (j == i) ) != 0)
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
      if ( 2 * ( ((j - 1) == i) - (j == i) ) != 0)
       csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
    }
  }
}
```

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      if ( j == i + 1 || j == i)
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
      if ( j == i + 1 || j == i)
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ), outputs, n );
    }
  }
}
```

Sparse Solving

- We have now simplified all derivative updates in terms of the propagated sparse inputs
- For each accumulate call, solve for what indices could lead to non-zero indices
- Extract out the value being stored and call at exactly those indices
- Requires the accumulate function to be commutative, and append-like

Sparse AD Step 3: Sparse Solving

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      if ( j == i + 1 || j == i)
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ) );
      if ( j == i + 1 || j == i)
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ) );
    }
  }
}
```

```
void hessian(double* in, sparse<double>*
outputs) {
  for(int i=0; i<n; i++) {
     body(0, i, i ,    in, outputs);
     body(0, i, i + 1 ,in, outputs);
     body(1, i, i ,    in, outputs);
     body(1, i, i + 1 ,in, outputs);

   }
}
```

```
void body(int storenum, int i, int j,
          double* in, sparse<double>* outputs) {
  if( j >=0 && j<n) {
     if ( storenum == 0 )
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ) );
      if ( storenum == 1 )
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ) );
  }
}
```

Sparse AD Step 3: Sparse Solving

```
void hessian(double* in, sparse<double>* outputs) {
  for(int i=0; i<n; i++) {
    for(int j=0; j<n-1; j++) {
      if ( j == i + 1 || j == i)
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ) );
      if ( j == i + 1 || j == i)
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ) );
    }
  }
}
```

O(n) compute

O(n) data

```
void hessian(double* in, sparse<double>*
outputs) {
  for(int i=0; i<n; i++) {
     body(0, i, i ,    in, outputs);
     body(0, i, i + 1 ,in, outputs);
     body(1, i, i ,    in, outputs);
     body(1, i, i + 1 ,in, outputs);

  }
}
```

```
void body(int storenum, int i, int j,
          double* in, sparse<double>* outputs) {
  if( j >=0 && j<n) {
     if ( storenum == 0 )
        csr_store(j - 1, 2 * ( ((j - 1) == i) - (j == i) ) );
      if ( storenum == 1 )
        csr_store(j    , 2 * ( ((j - 1) == i) - (j == i) ) );
  }
}
```

Evaluation

Runtime performance (log-log)

seconds

JAX (dense)

$O(n^3)$

$O(n^2)$

Enzyme (dense)

$O(n)$

Our method in JAX

Our method in Enzyme

hand-written reference

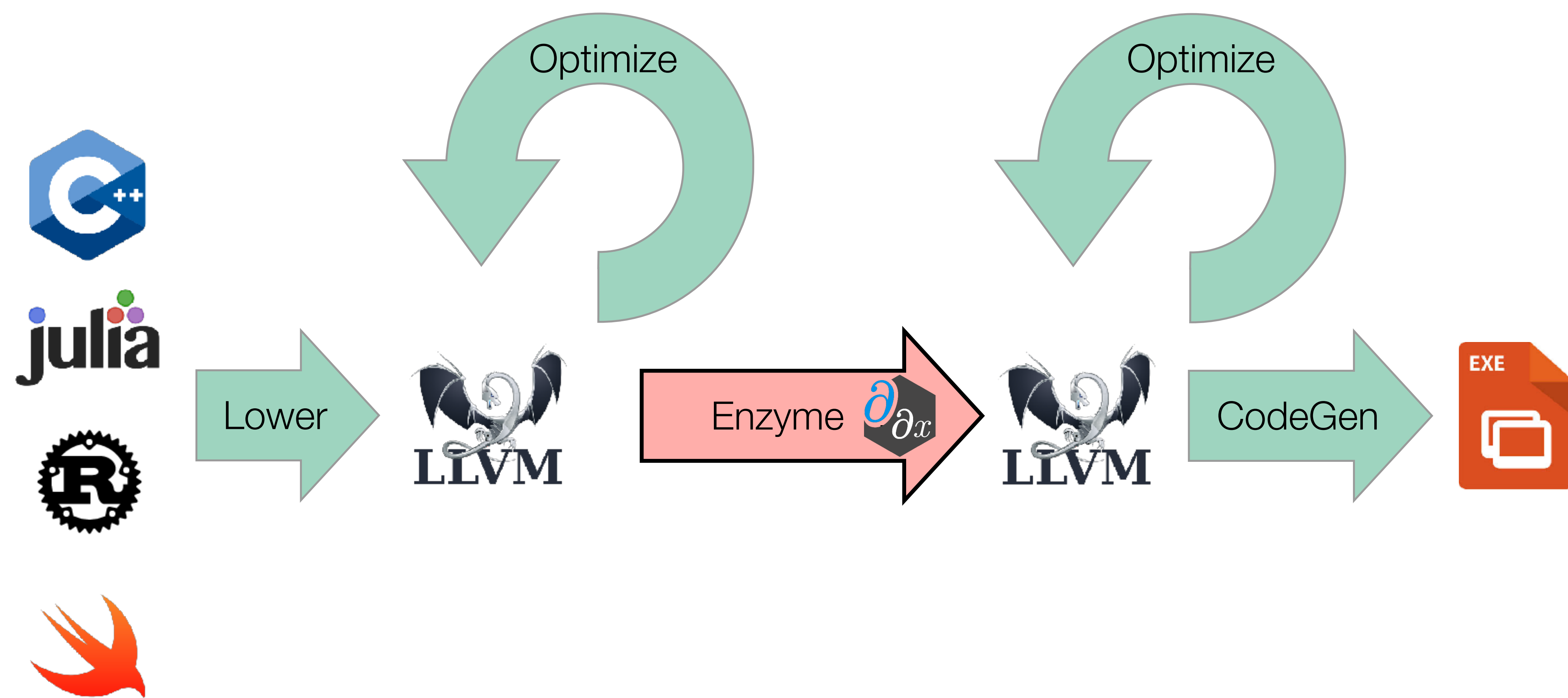
10^{1} 10^{0} 10^{-1} 10^{-2} 10^{-3} 10^{-4}

10^{3} 10^{4} 10^{5} 10^{6}

number of vertices (n)

Spadina + {Enzyme, JaX}

- As Enzyme is within the compiler, the sparsification algorithm above is fully automatic from existing compliant code which uses Enzyme for dense AD
- Spadina-JaX requires rewriting to use special primitives, and must have the iteration space manually determined and specified




```
import jax.numpy as jnp

def local_energy(…): return jnp.foo(…)

energy = spadina.vmapsum(local_energy,
                                    elements)
spadina.sparse_grad(energy)
```

Spadina-Enzyme [and to a lesser extent Spadina-JaX]

- Automatically generate sparse versions of otherwise dense derivatives
- Exploits structural (not data) sparsity
- As Enzyme differentiates code in a variety of parallel frameworks (OpenMP, MPI, Julia Tasks, GPU), and languages (C, C++, Fortran, Julia, Rust, Swift, etc), can be widely applied
  - Haven't written nice syntactic sugar for frontends beyond magic calls, will do soon (sorry Julia folks)
- Open source (enzyme.mit.edu & join our mailing list)!
  https://github.com/EnzymeAD/Enzyme/blob/main/enzyme/test/Integration/Sparse/
- Lots more ongoing work including scheduling, checkpointing, and more

Acknowledgements

- Thanks to James Bradbury, Alex Chernyakhovsky, Lilly Chin, Hal Finkel, Marco Foco, Laurent Hascoet, Mike Innes, Tim Kaler, Charles Leiserson, Yingbo Ma, Chris Rackauckas, TB Schardl, Lizhou Sha, Yo Shavit, Dhash Shrivathsa, Nalini Singh, Vassil Vassilev, and Alex Zinenko

- William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DESC0019323. Valentin Churavy was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. HR0011-20-9-0016, and in part by NSF Grant OAC-1835443. Ludger Paehler was supported in part by the German Research Council (DFG) under grant agreement No. 326472365.

- This research was supported in part by LANL grant 531711; in part by the Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357; in part by the Exascale Computing Project (17-SC-20-SC). Research was sponsored by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000.

- The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government.

Spadina-Enzyme [and to a lesser extent Spadina-JaX]

- Automatically generate sparse versions of otherwise dense derivatives
- Exploits structural (not data) sparsity
- As Enzyme differentiates code in a variety of parallel frameworks (OpenMP, MPI, Julia Tasks, GPU), and languages (C, C++, Fortran, Julia, Rust, Swift, etc), can be widely applied
  - Haven't written nice syntactic sugar for frontends beyond magic calls, will do soon (sorry Julia folks)
- Open source (enzyme.mit.edu & join our mailing list)!
  https://github.com/EnzymeAD/Enzyme/blob/main/enzyme/test/Integration/Sparse/
- Lots more ongoing work including scheduling, checkpointing, and more

Enzyme