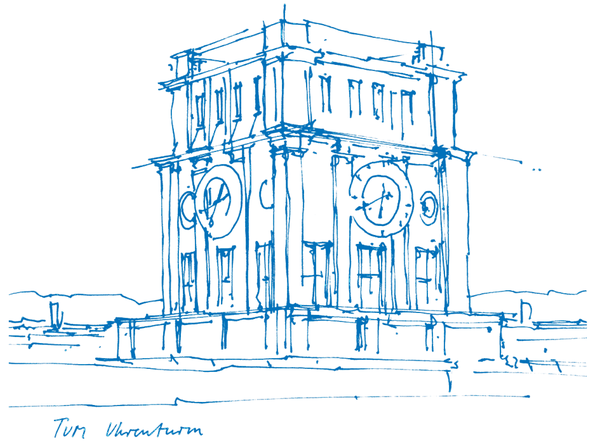
# High-Performance Compiler-based Automatic Differentiation

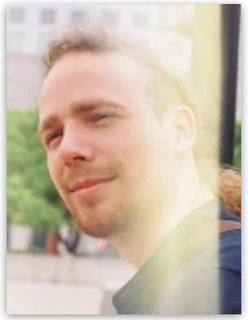
Ludger Paehler, TReNDs Group Talk, 14th of July 2023



# The Work of Many People

Ludger Paehler
(TUM)

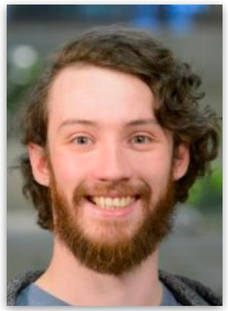
William S. Moses
(UIUC)



Valentin Churavy
(MIT)

Jan Hückelheim
(ANL)

Michel Schanen
(ANL)



Johannes Doerfert
(LLNL)

Sri Hari Krishna
Narayanan
(ANL)

Lukas Heinrich
(TUM)

Adrian Maldonado
(ANL)

Nikolaus A. Adams
(TUM)

# Coinstac Acceleration with Numba

## Compilation with LLVM

```python
from numba import jit

@jit(nopython=True)
def remote_stats(MSE, varX_matrix_global, avg_beta_vector):
    my_shape = avg_beta_vector.shape
    ts = np.zeros(my_shape)

    for voxel in prange(my_shape[0]):
        var_covar_beta_global = MSE[voxel] * np.linalg.inv(varX_matrix_global)
        se_beta_global = np.sqrt(np.diag(var_covar_beta_global))
        ts[voxel, :] = avg_beta_vector[voxel, :] / se_beta_global

    return ts
```

# Table of Contents

# Automatic Differentiation

# Calculus: Revisited

- Derivatives compute the rate of change of a function's output with respect to input(s)

$$f'(x) = \lim_{h \longrightarrow 0} \frac{f(a+h) - f(a)}{h}$$

- Derivatives (& generalizations like gradients) used widely across science:
  - Machine learning (back-propagation, Bayesian inference, uncertainty quantification)
  - Scientific computing (modeling, simulation, gradient-based optimization)

# Reverse-Mode Differentiation

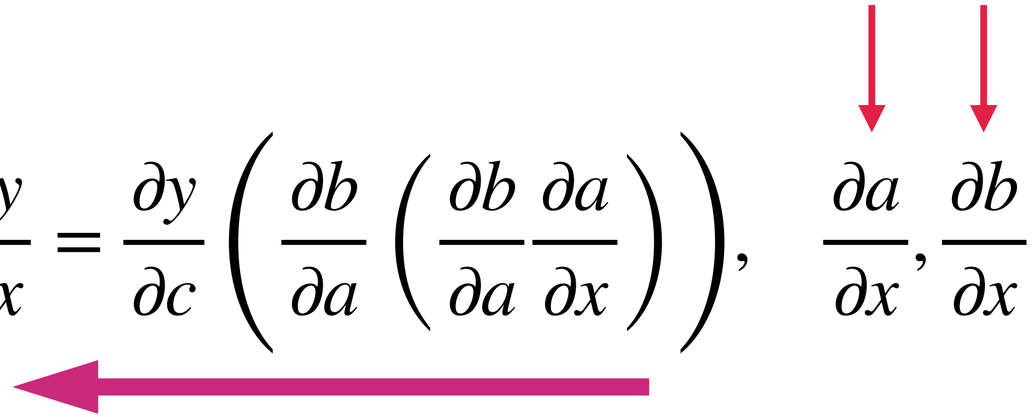
- Enzyme's main mode of differentiation is reverse-mode differentiation
- Reverse-mode differentiation
  - Evaluation performed left to right
  - Information flow in opposite direction as computation, elaborate caching mechanism required
  - Efficient for function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, where $m \ll n$

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial c}\left(\frac{\partial c}{\partial b}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\right)\right)$$

# Forward-Mode Differentiation

- Evaluation of the gradient is performed from the right to the left
- Accumulates the Jacobians of the intermediate variables w.r.t. the input x
- Information flows in the same direction as the computation
- Efficient for function $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, where m $\gg$ n

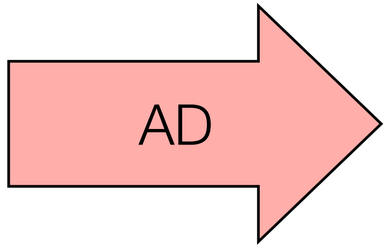
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial c}\left(\frac{\partial b}{\partial a}\left(\frac{\partial b}{\partial a}\frac{\partial a}{\partial x}\right)\right), \quad \frac{\partial a}{\partial x}, \frac{\partial b}{\partial x}$$



# Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs

```
double relu3(double x){
    if (x > 0)
        return pow(x, 3);
    else
        return 0;
}
```



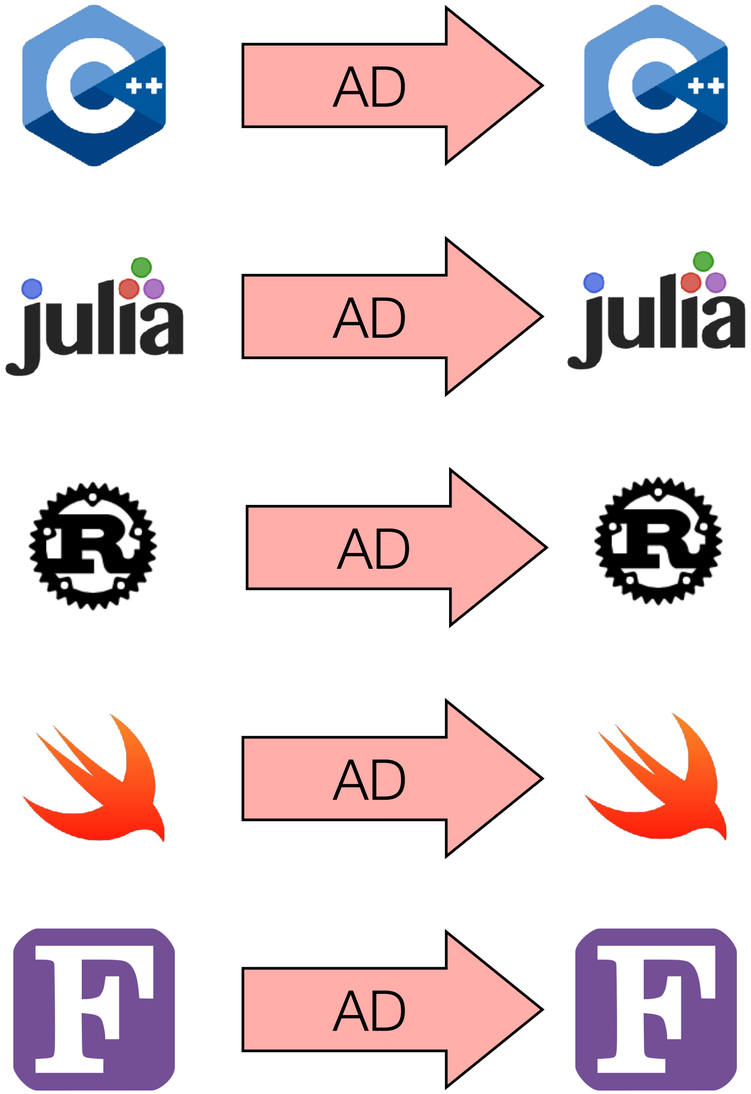

```
double grad_relu3(double x){
    if (x > 0)
        return 3 * pow(x, 2);
    else
        return 0;
}
```

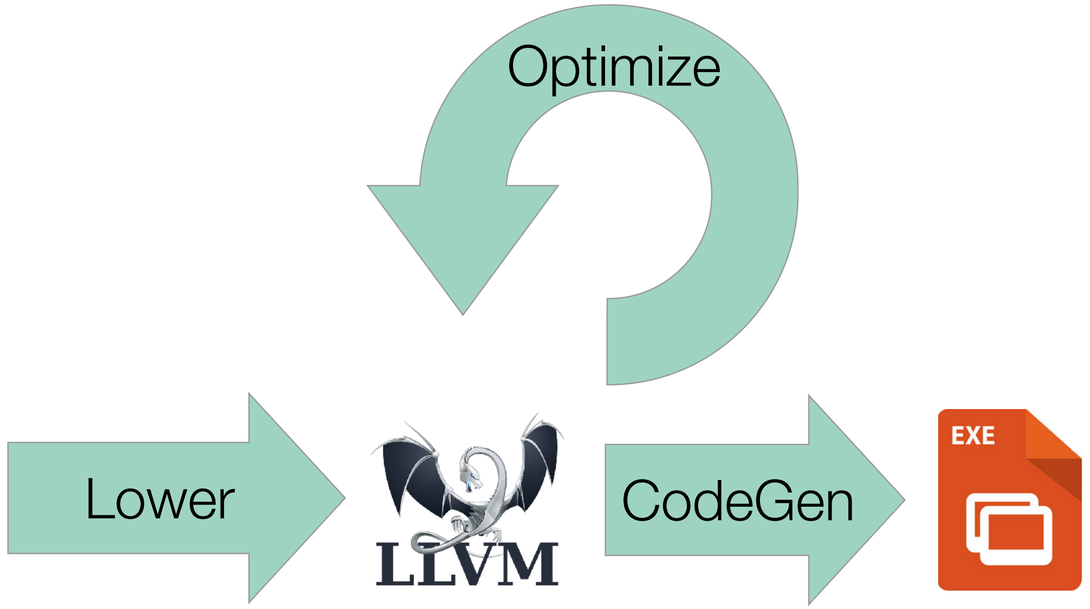
- Unlike numerical approaches, automatic differentiation (AD) can compute the derivatives of all inputs (or outputs) at once, without approximation error!

```
// Numeric differentiation
double grad_input[100];

for (int i=0; i<100; i++){
    double input2[100] = input;
    input2[i] += 0.01;
    grad_input[i] = (f(input2) - f(input))/0.001;
}
```

```
// Automatic differentiation
double grad_input[100];

grad_f(input, grad_input);
```

# Existing AD Pipelines



# Case Study: Vector Normalization

```
// Compute magnitude in O(n)
double mag(double[] x);

// Compute norm in O(n^2)
void norm(double[] out, double[] in){

    for (int i=0; i<n; i++){
        out[i] = in[i] / mag(in);
    }
}
```

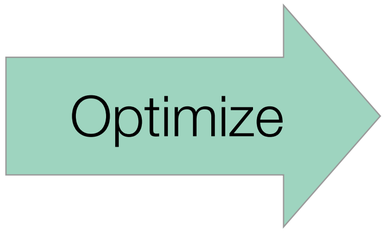
# Case Study: Vector Normalization

```
// Compute magnitude in O(n)
double mag(double[] x);

// Compute norm in O(n^2)
void norm(double[] out, double[] in){
    double res = mag(in);
    for (int i=0; i<n; i++){
        out[i] = in[i] / mag(in);
    }
}
```

# The Relation of Optimization & AD

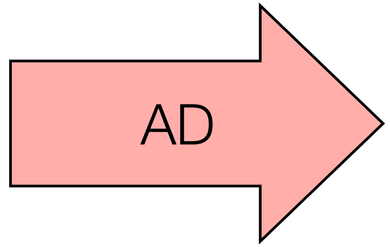
$O(n^2)$

```
for i=0..n {
    out[i] /= mag(in)
}
```




$O(n)$

```
res = mag(in)
for i=0..n {
    out[i] /= res
}
```




$O(n)$

```
d_res = 0.0
for i=0..n {
    d_res += d_out[i]...
}
d_mag(d_in, d_res)
```

# The Relation of Optimization & AD

$O(n^2)$

```
for i=0..n {
    out[i] /= mag(in)
}
```

Optimize

$O(n)$

```
res = mag(in)
for i=0..n {
    out[i] /= res
}
```

AD

$O(n)$

```
d_res = 0.0
for i=0..n {
    d_res += d_out[i]...
}
d_mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {
    out[i] /= mag(in)
}
```

AD

$O(n^2)$

```
for i=0..n {
    d_res += d_out[i]...
    d_mag(d_in, d_res);
}
```

# The Relation of Optimization & AD

$O(n^2)$

```
for i=0..n {
    out[i] /= mag(in)
}
```

Optimize

$O(n)$

```
res = mag(in)
for i=0..n {
    out[i] /= res
}
```

AD

$O(n)$

```
d_res = 0.0
for i=0..n {
    d_res += d_out[i]...
}
d_mag(d_in, d_res)
```

$O(n^2)$

```
for i=0..n {
    out[i] /= mag(in)
}
```

AD

$O(n^2)$

```
for i=0..n {
    d_res += d_out[i]...
    d_mag(d_in, d_res);
}
```

Optimize

$O(n^2)$

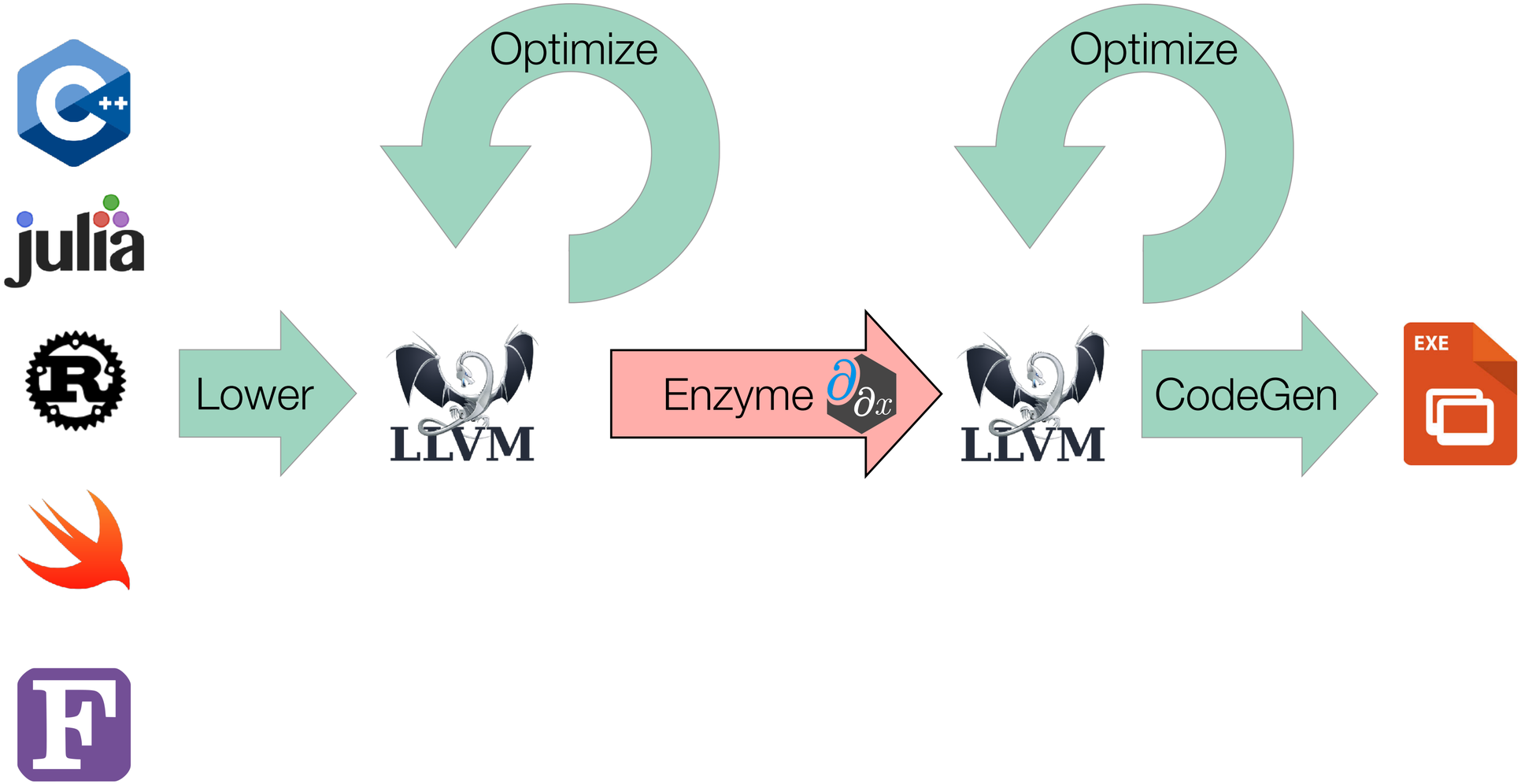
```
for i=0..n {
    d_res = d_out[i]...
    d_mag(d_in, d_res);
}
```

# Differentiating after optimization can create asymptotically faster gradients

# Compiler-based Automatic Differentiation

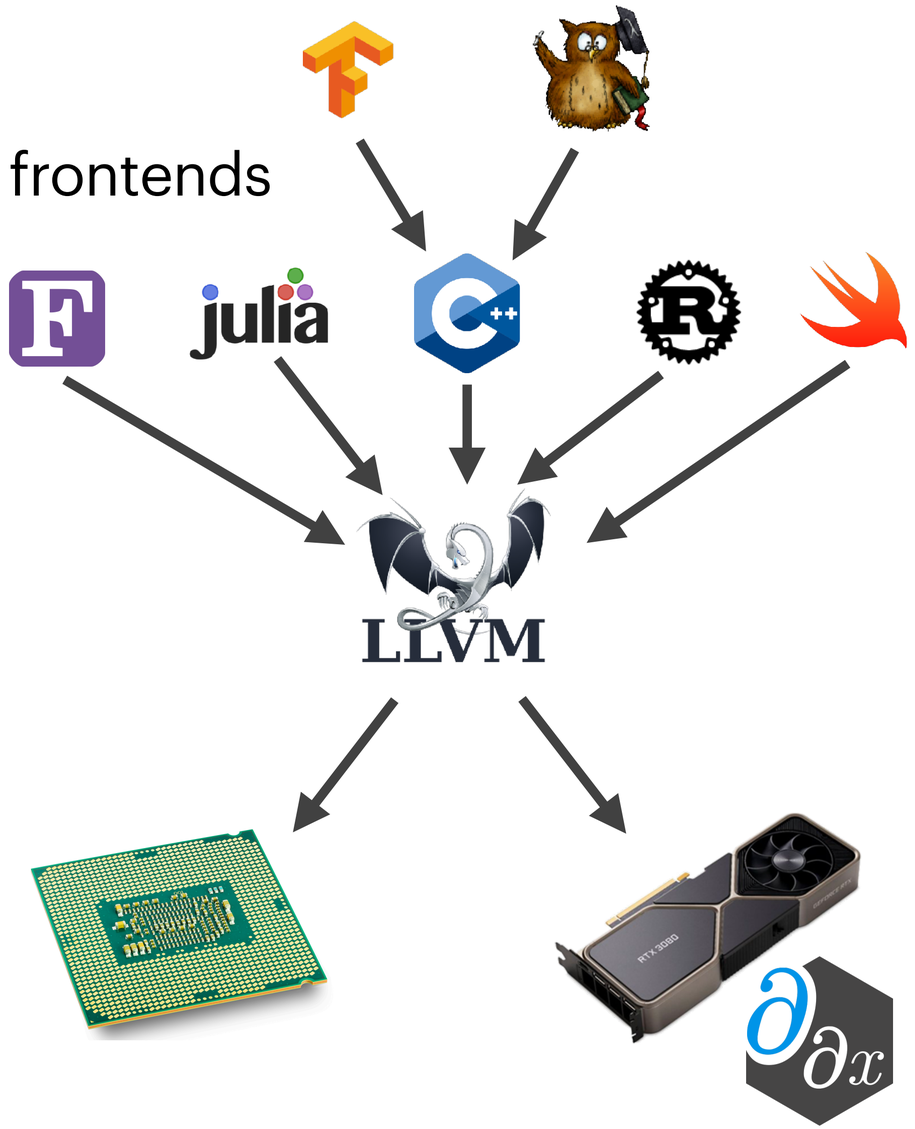
Performing AD at a low level lets us work on optimized code!

# Enzyme's Approach



# Why Does Enzyme Use LLVM?

- Generic low-level compiler infrastructure with many frontends
  - “Cross platform assembly”
  - Many backends
    - CPU (X86, ARM, RISC-V)
    - GPU (NVIDIA, AMD, Intel)
- Well-defined semantics
- Large collection of optimizations and analyses
  - Ability to write platform-independent passes



# Advantages of Compiler-based AD

- 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 our own Just-in-Time (JIT) Compiler
  - Will become more important later on!

# Case Study: ReLU3

## C Source

```c
double relu3(double x){
    double result;
    if (x > 0)
        result = pow(x, 3);
    else
        result = 0;
    return result;
}
```

## Enzyme Usage

```c
double diffe_relu3(double x){
    return __enzyme_autodiff(relu3, x);
}
```

## Resulting in the gradient

```c
double diffe_relu3(double x){
    double result;
    if (x > 0)
        result = 3 * pow(x, 2);
    else
        result = 0;
    return result;
}
```

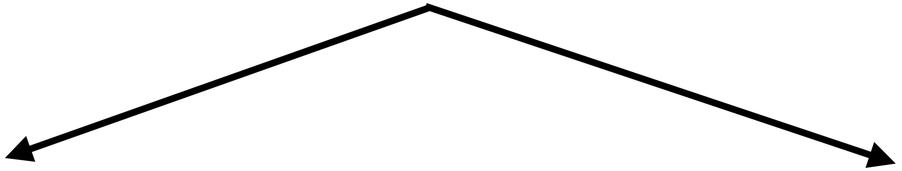
Essentially the optimal hand-written gradient!

But...

Low-level code **lacks** information necessary to compute adjoints

# Challenges of Low-Level AD on LLVM-IR

```
void f(void* dst, void* src){
    memcpy(dst, src, 8);
}
```



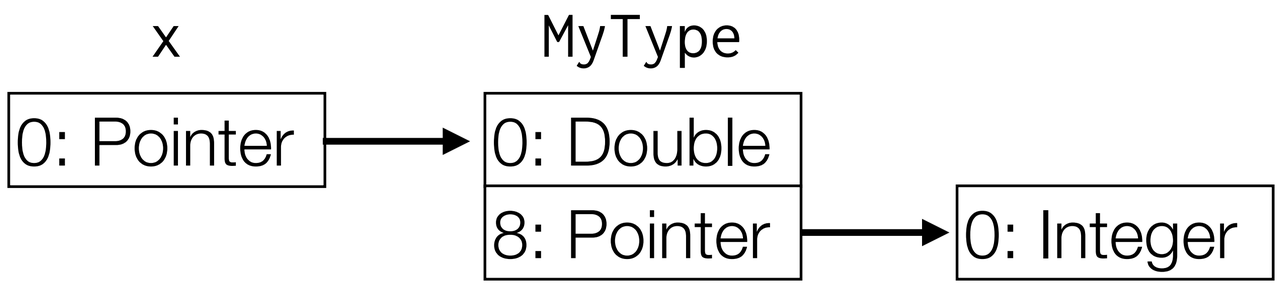
```
void grad_f(double* dst, double* grad_dst,
            double* src, double* grad_src){
    // Forward Pass
    memcpy(dst, src, 8);

    // Reverse Pass
    grad_src[0] += grad_dst[0];
    grad_dst[0] = 0;
}
```

```
void grad_f(float* dst, float* grad_dst,
            float* src, float* grad_src){
    // Forward Pass
    memcpy(dst, src, 8);

    // Reverse Pass
    grad_src[0] += grad_dst[0];
    grad_dst[0] = 0;
    grad_src[1] += grad_dst[1];
    grad_dst[1] = 0;
}
```

# Type Analysis

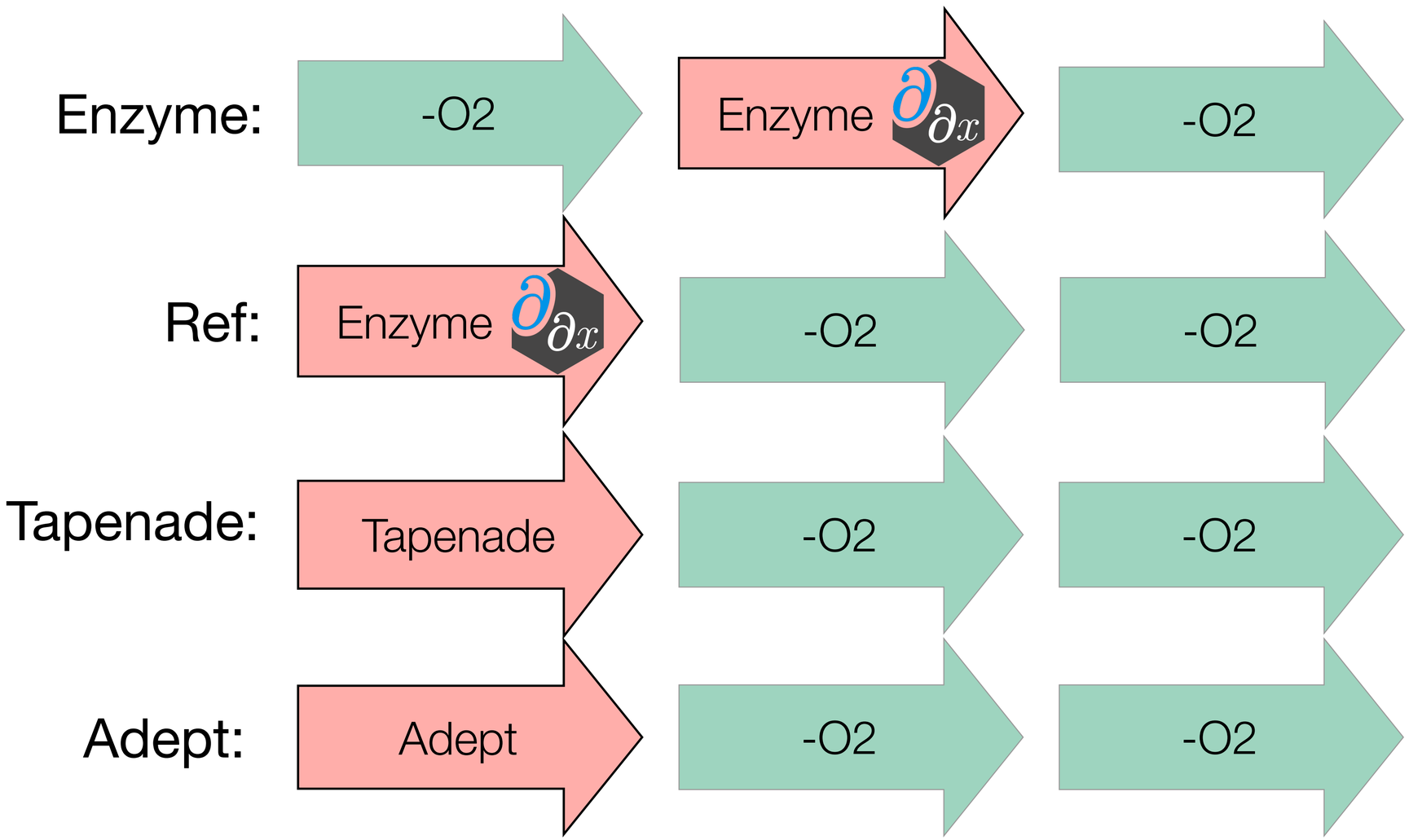
- New interprocedural dataflow analysis that detects the underlying type of data
- Each value has a set of memory offsets: type
- Perform series of fixed-point updates through instructions

```
struct MyType {
    double;
    int*;
}

x = MyType*;
```

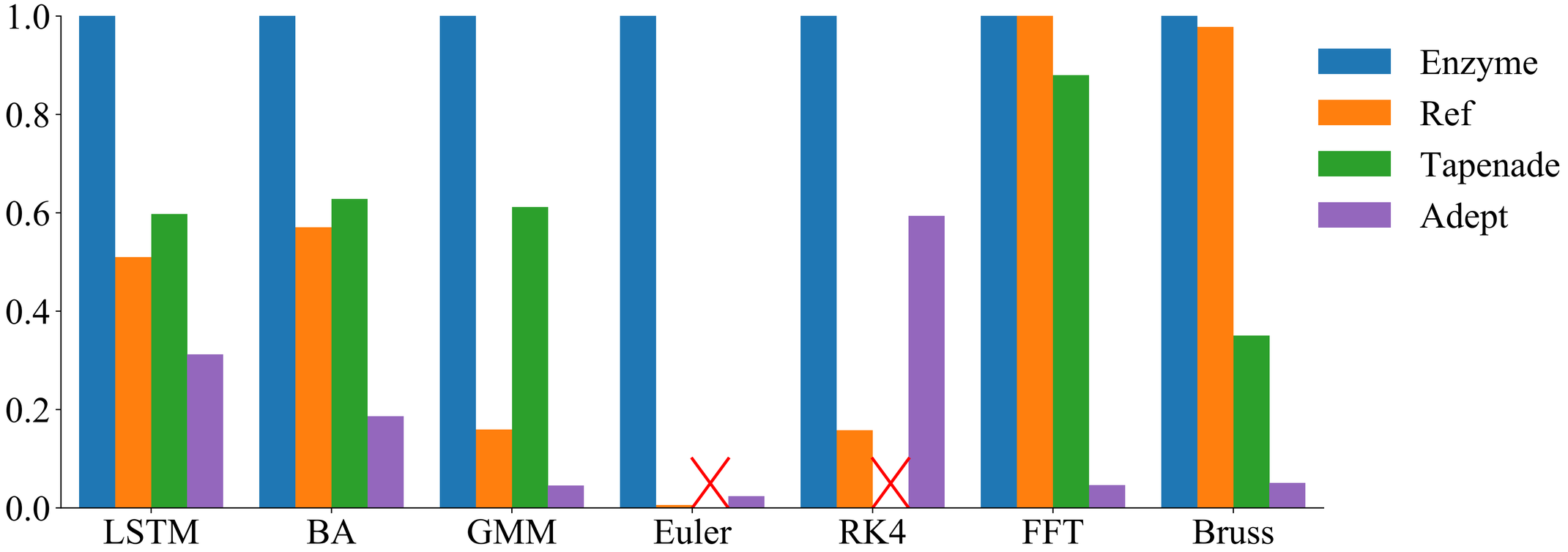



types(x) = {[0]:Pointer, [0,0]:Double, [0,8]:Pointer, [0,8,0]:Integer}

# Experimental Setup with ADBench



# Speedup of Enzyme



Enzyme is **4.2x faster** than the Reference!

# Automatic Differentiation on GPUs

# Reverse-Mode AD on GPUs

- Prior work has not explored reverse-mode AD of existing GPU kernels
  1. Reversing parallel control flow can lead to incorrect results
  2. Complex performance characteristics make it difficult to synthesize efficient code
  3. Resource limitations can prevent kernels from running at all

# Challenges of Parallel AD

- The adjoint of an instruction increments the derivative of its input
- Benign read races in forward pass ⇒ Write race in reverse pass (undefined behaviour)

```
void set(double* ar, double val){
    parallel_for(int i=0; i<10; i++)
        ar[i] = val;
}
```

Read Race

Write Race

```
double gradient_set(double* ar, double* d_ar,
                    double val){
    double d_val = 0.0;

    parallel_for(int i=0; I<10; I++)
        ar[i] = val;

    parallel_for(int I=0; I<10; I++){
        d_val += d_ar[i];
        d_ar[i] = 0.0;
    }

    return d_val;
}
```

# The Complex GPU Memory Hierarchy

| Per Thread | Per Block | Per GPU |
| --- | --- | --- |
| Register | Shared Memory | Global Memory |
| ~Bytes | ~KBs | ~GBs |
| Use Limits Parallelism | Use Limits Parallelism | |

Slower, larger amount of memory

# Efficient GPU Code

- To maintain correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Just like on the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations are **not** sufficient
- Novel GPU- and AD-specific optimizations can speed up computation by several orders of magnitude

```
// Forward Pass

out[i] = x[i] * x[i];

x[i] = 0.0f;

// Reverse (gradient) pass
...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

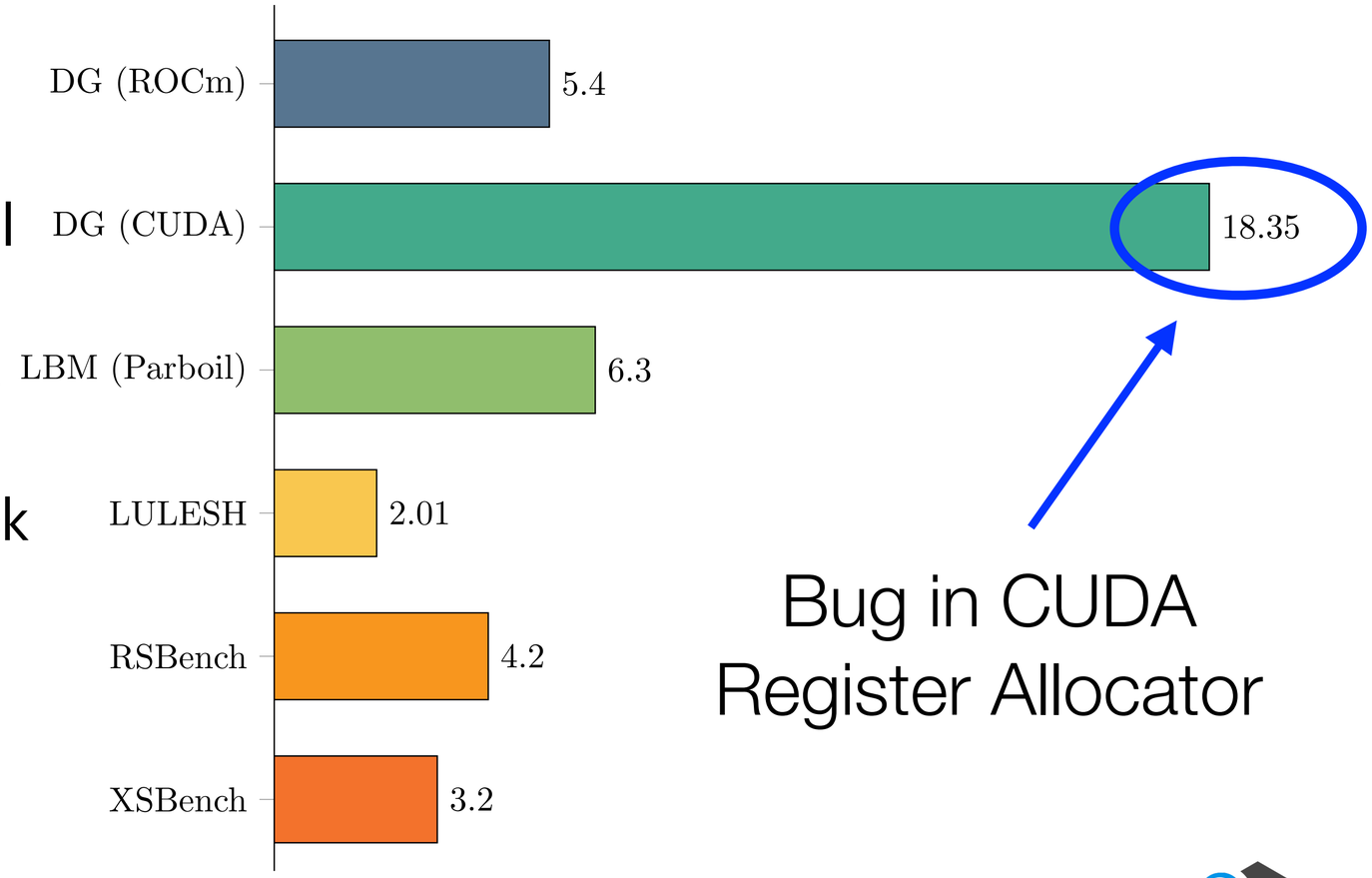
# ~~Efficient~~ Correct GPU Code

- To maintain correctness, Enzyme may need to cache values in order to compute the gradient
  - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Just like on the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations are **not** sufficient
- Novel GPU- and AD-specific optimizations can speed up computation by several orders of magnitude

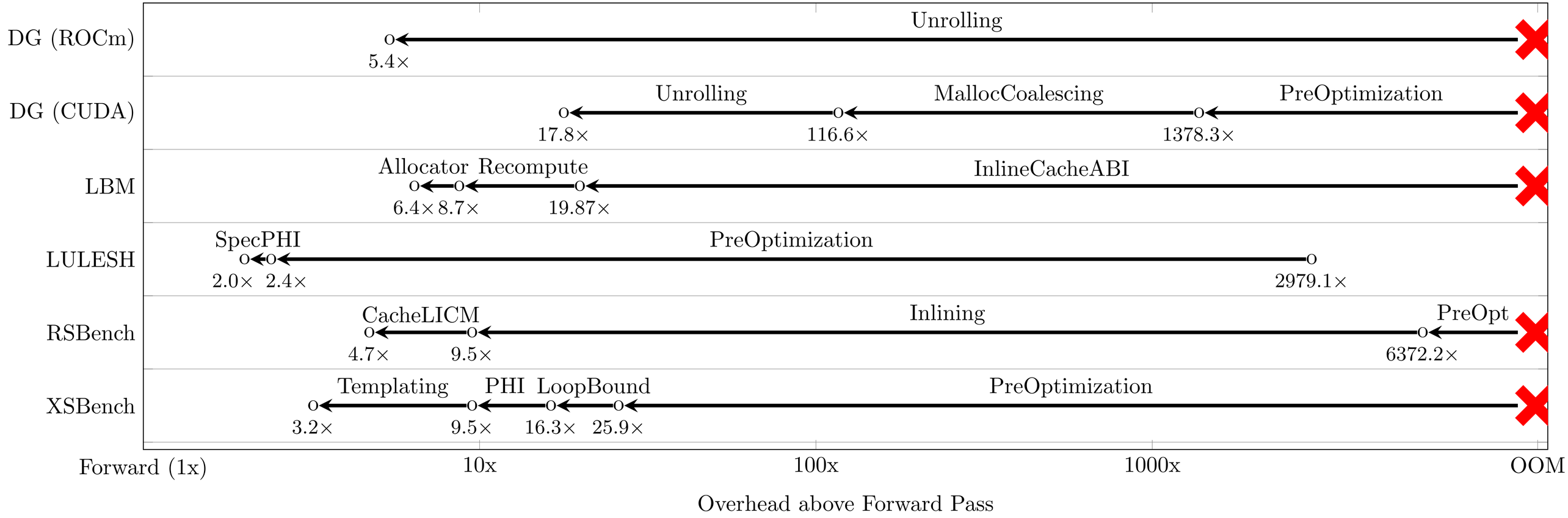
```
double* x_cache = new double[...];

// Forward Pass
out[i] = x[i] * x[i];
x_cache[i] = x[i];

x[i] = 0.0f;

// Reverse (gradient) pass
...
grad_x[i] += 2 * x_cache[i]
                * grad_out[i];
...

delete[] x_cache;
```

# GPU Gradient Overhead

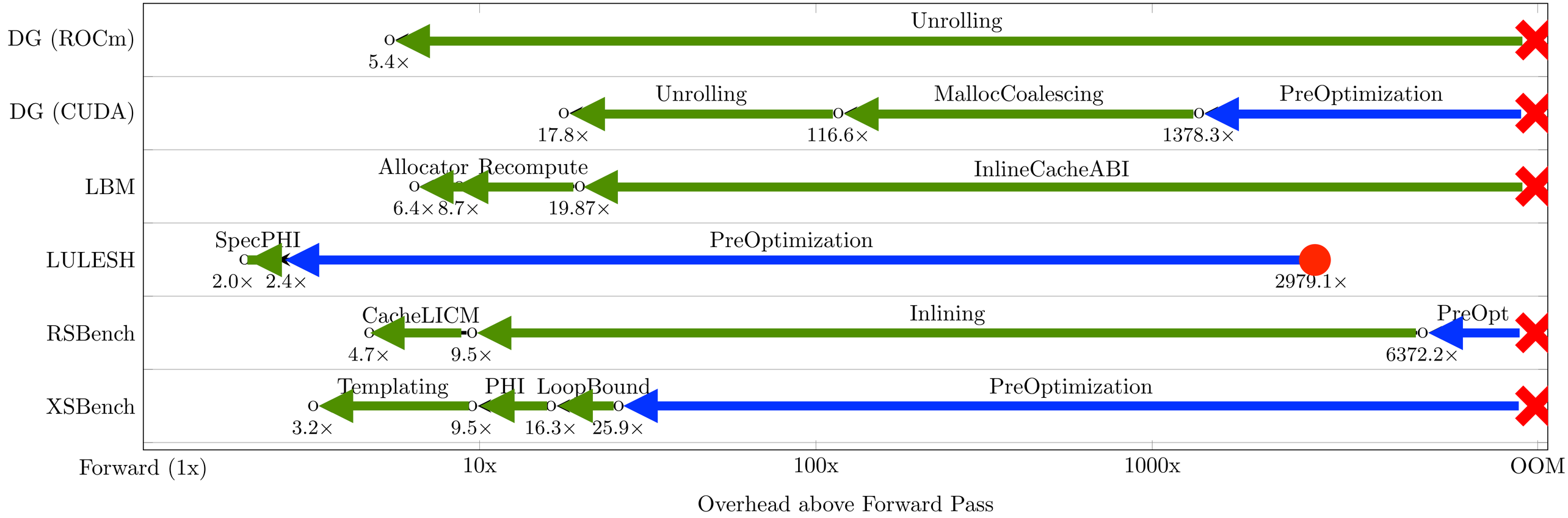
- Evaluation of both original code and gradient
  - DG: Discontinuous Galerkin Integral (Julia)
  - LBM: Particle-based fluid dynamics simulation
  - LULESH: Unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute-bound respectively)



# Ablation Analysis



# Ablation Analysis



# Just Enzyme and Numba: Differentiation of Numba

# Coinstac

## Compilation with ~~Numba~~ LLVM

```python
from numba import jit

@jit(nopython=True)
def remote_stats(MSE, varX_matrix_global, avg_beta_vector):
    my_shape = avg_beta_vector.shape
    ts = np.zeros(my_shape)

    for voxel in prange(my_shape[0]):
        var_covar_beta_global = MSE[voxel] * np.linalg.inv(varX_matrix_global)
        se_beta_global = np.sqrt(np.diag(var_covar_beta_global))
        ts[voxel, :] = avg_beta_vector[voxel, :] / se_beta_global

    return ts
```

# Background

## What is Numba?

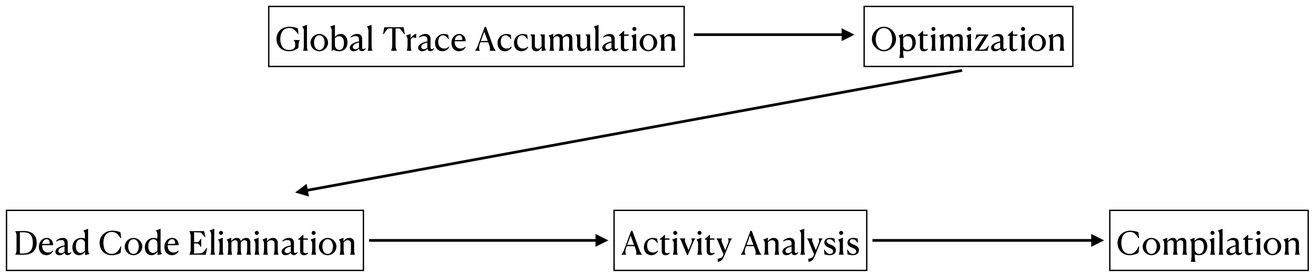
- Numba is a just-in-time (JIT) compiler for Python, which takes Python code, and is utilizing the LLVM compilation infrastructure to compile to different hardware targets such as
  - CPU
  - GPU
- Optimized for NumPy, and Stencil-Computation
  - Vectorization by Numba
  - OpenMP
  - Intel Threading Building Blocks (TBB)
  - CUDA

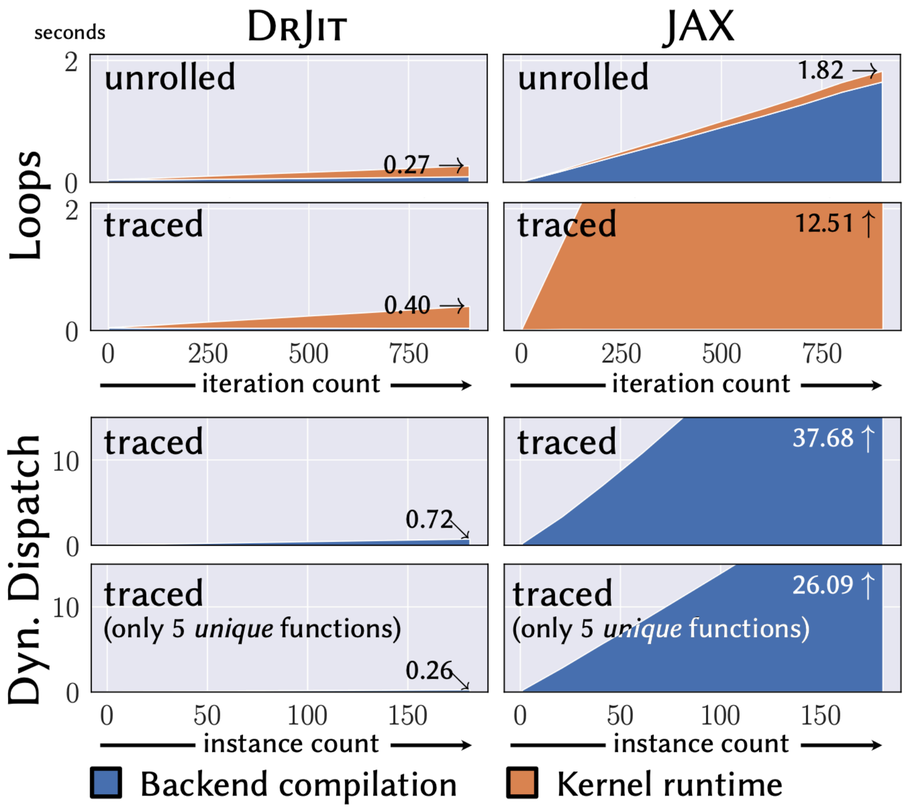
# Why would we care about another JIT-engine for Python when we already have JAX?

# Background

## Dr. JIT: A Just-In-Time Compiler for Differentiable Rendering

- JAX-JIT suboptimal for differentiable rendering workflows -> Write our own JIT!
- Tracing-based JIT for differentiable rendering with its own AD-system
  - Fusion of kernels into one mega-kernel




Source: Jakob, W., Speierer, S., Roussel, N., & Vicini, D. (2022). DR. JIT: a just-in-time compiler for differentiable rendering. *ACM Transactions on Graphics (TOG)*, *41*(4), 1-19.

# Background

## Loopy Code: Numba vs JAX

```python
import jax.numpy as jnp
from jax import jit


@jit
def test_func2(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += jnp.tanh(a[i, i])
    return a + trace
```

JAX

```python
from numba import njit
import numpy as np


@njit(nopython=True)
def test_function(a):
    trace = 0.0
    for i in range(a.shape[0]):
        trace += np.tanh(a[i, i])
    return a + trace
```

Numba

# Background

## Stencil Code: Numba vs JAX

```jax
from jax import jit
import time
import numpy as np
import kernex as kex


# Using kernex for stencil computation in JAX
@jit
@kex.kmap(
    kernel_size=(3, 3),
    padding = 'valid',
    relative=True)# For relative indexing
def stencil1_kernel(a):
    return 0.25 * (a[0, 1] + a[1, 0] + a[0, -1] + a[-1, 0])
```

JAX's Stencil Computation Package

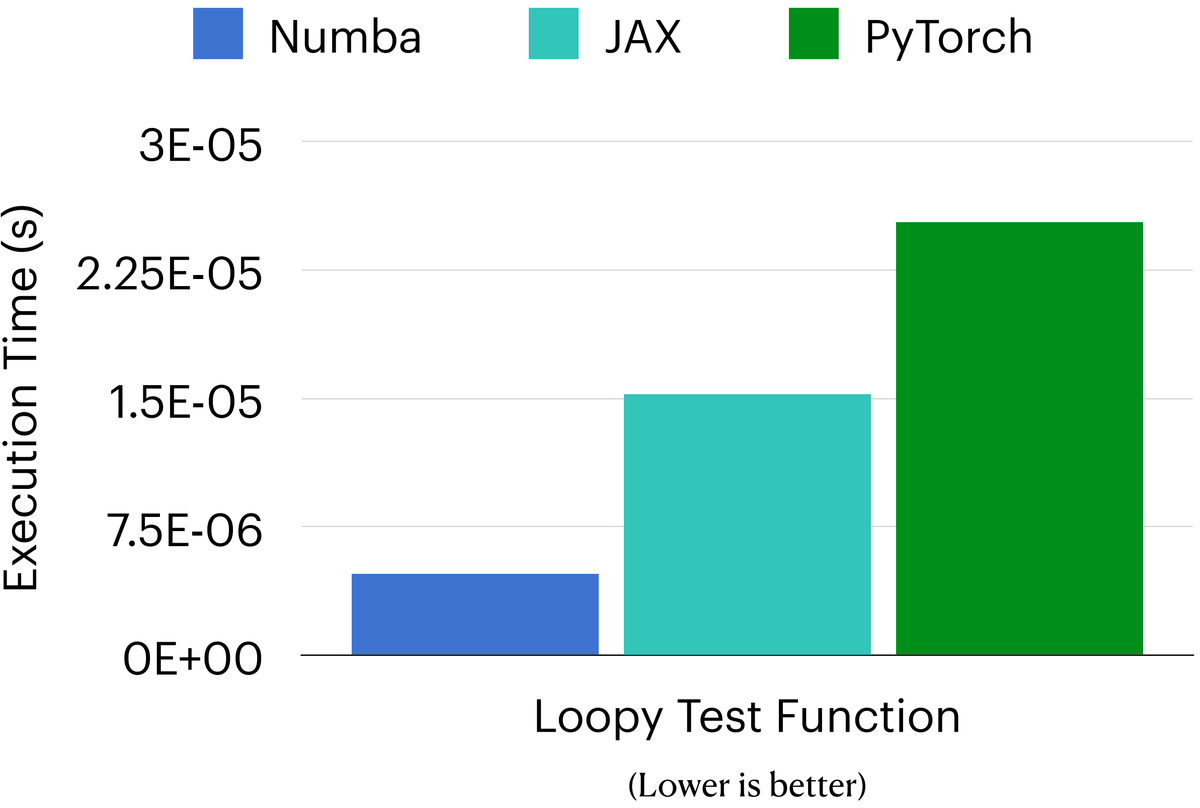
JAX

```numba
from numba import njit, stencil
import time
import numpy as np


# Using Numba's stencil decorator
@stencil
def _stencil1_kernel(a):
    return 0.25 * (a[0, 1] + a[1, 0] + a[0, -1] + a[-1, 0])


@njit()
def stencil1_kernel(a):
    return _stencil1_kernel(a)
```

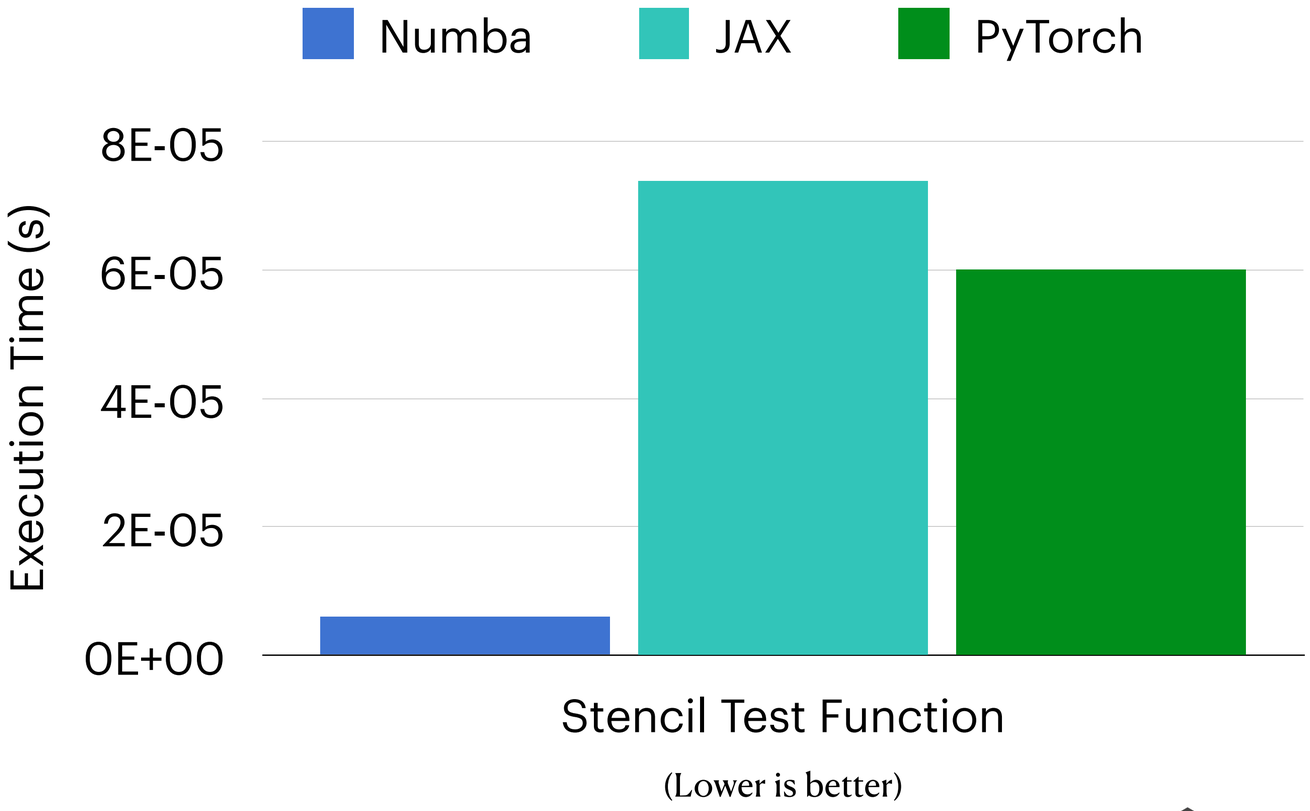
Numba

# Background

## Numba vs JAX




(Lower is better)




(Lower is better)

# Background

## Branchy Code: Numba vs JAX vs PyTorch

```python
import jax.numpy as jnp
from jax import jit, lax


def func1(_a):
    return _a**2 + 2 * _a


def func2(_b):
    return _b**4


@jit
def test_func3(_x):
    return lax.cond(jnp.sum(_x) < 100, func1, func2, _x)
```

Specialized Construct for Branching

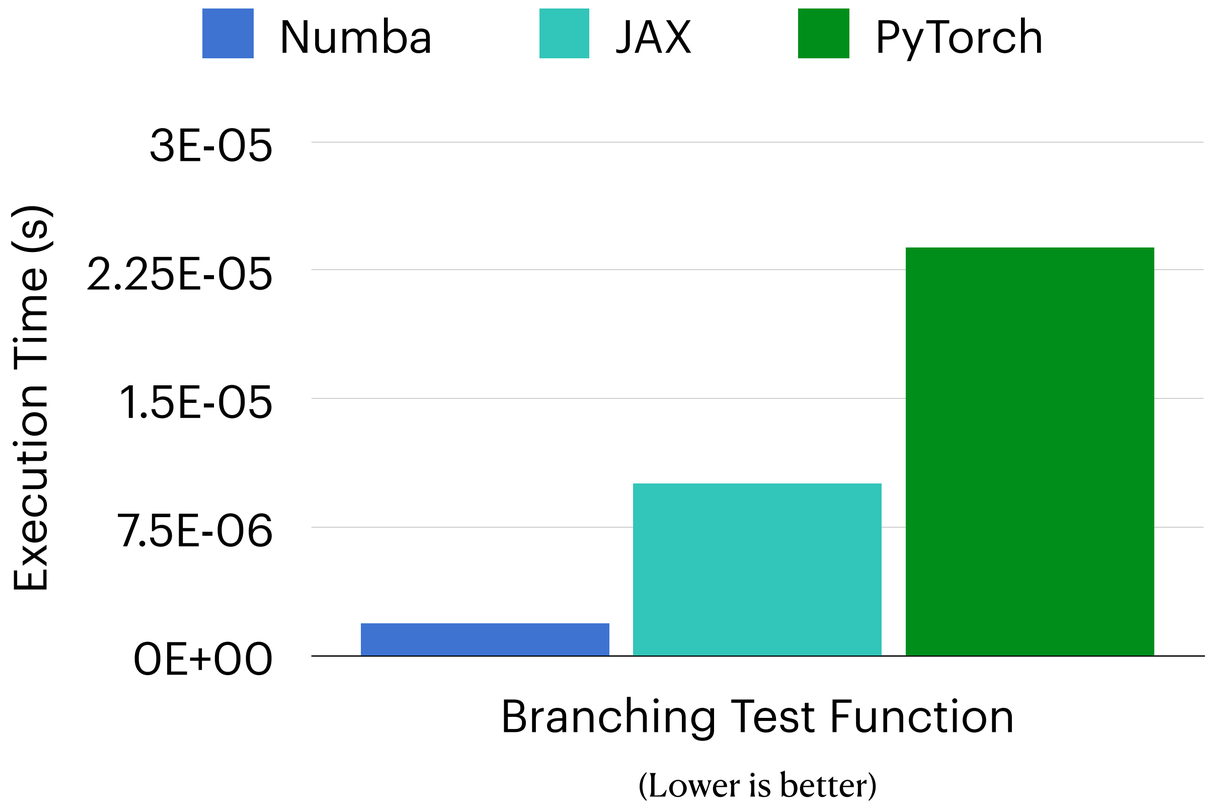
JAX

```python
from numba import njit
import numpy as np


@njit()
def test_function(_a):
    if np.sum(_a) < 100:
        return _a**2 + 2 * _a
    else:
        return _a**4
```

Numba

# Background

## Dynamism: Numba vs JAX vs PyTorch



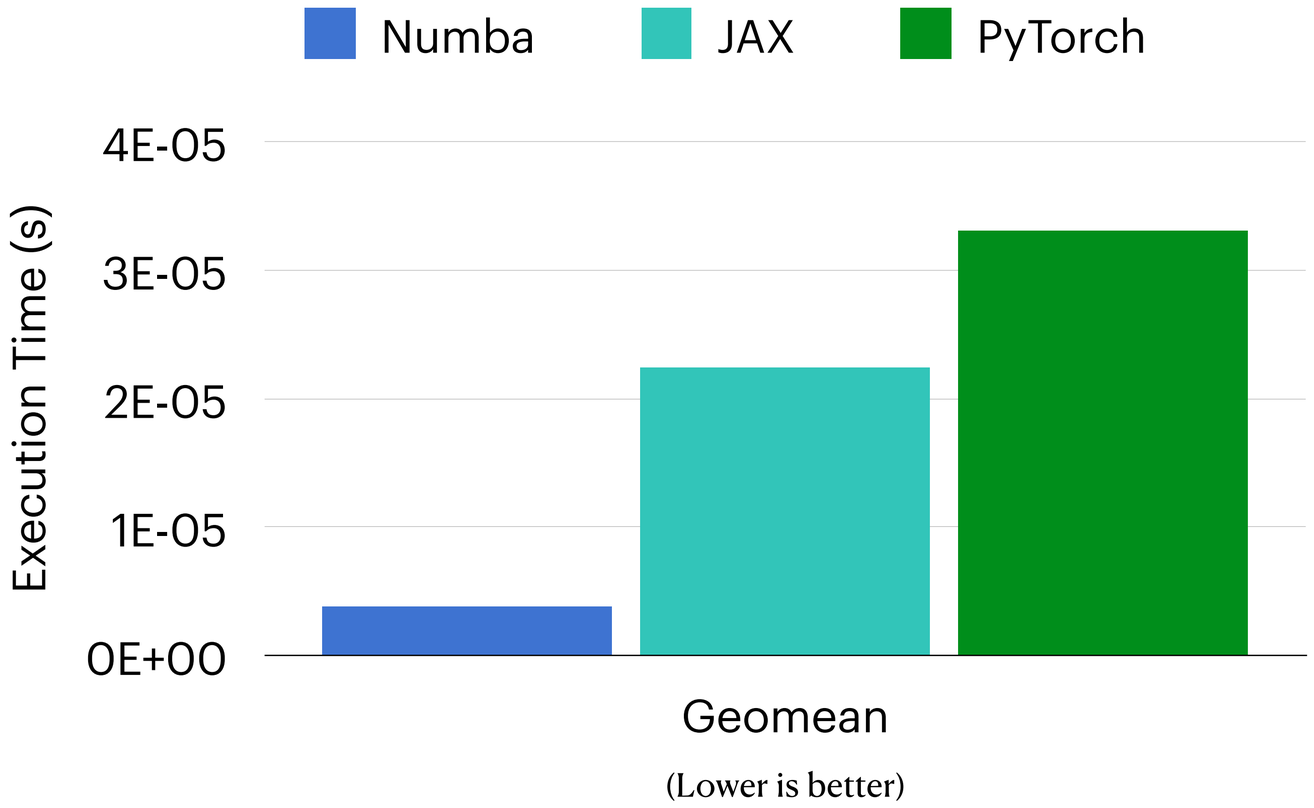

(Lower is better)

# Background

## Numba vs JAX Performance

- Numba an order of magnitude faster than JAX on this trivial benchmark
- Numba loves loopy code, whereas JAX does not

- **JIT engines conceived for very different use cases!**

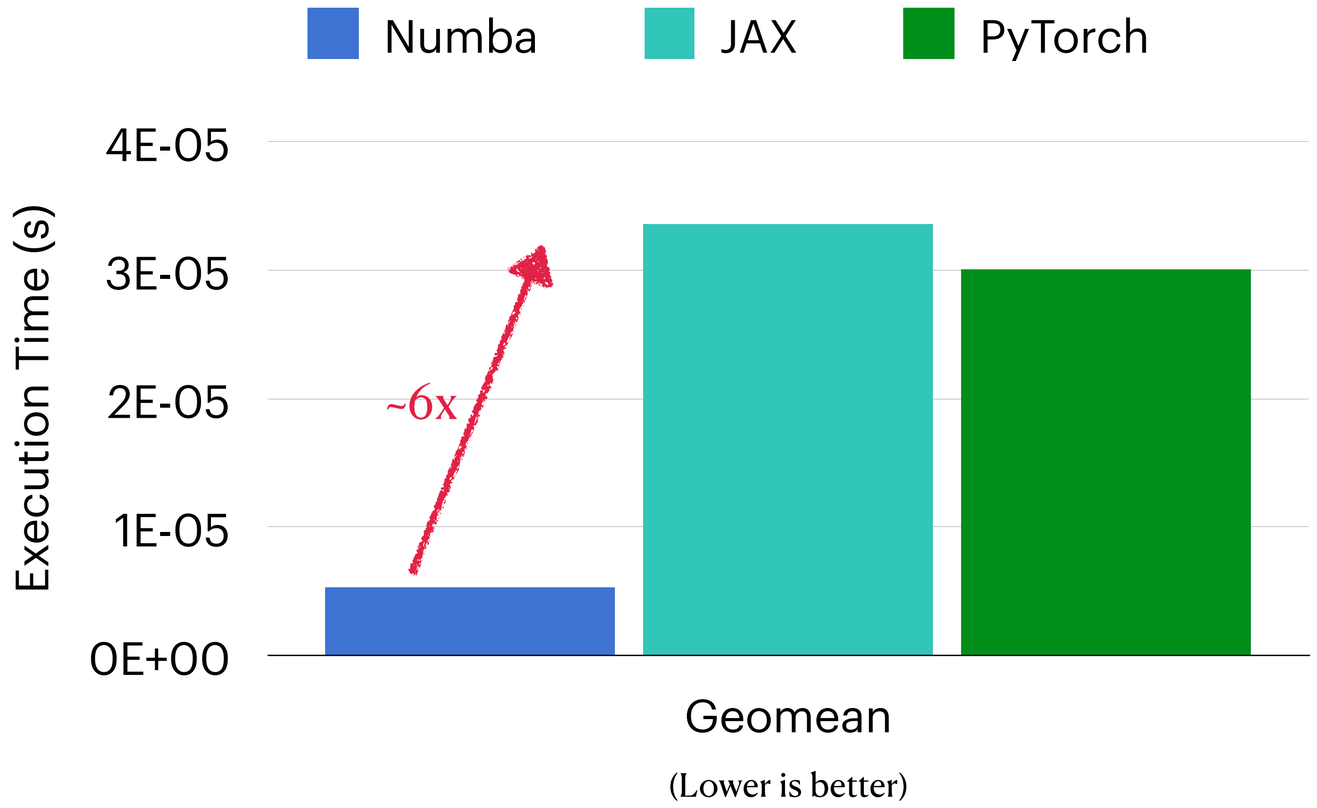


# Background

## Numba vs JAX Performance

- Numba an order of magnitude faster than JAX on this trivial benchmark
- Numba loves loopy code, whereas JAX does not

- **JIT engines designed for very different use cases!**

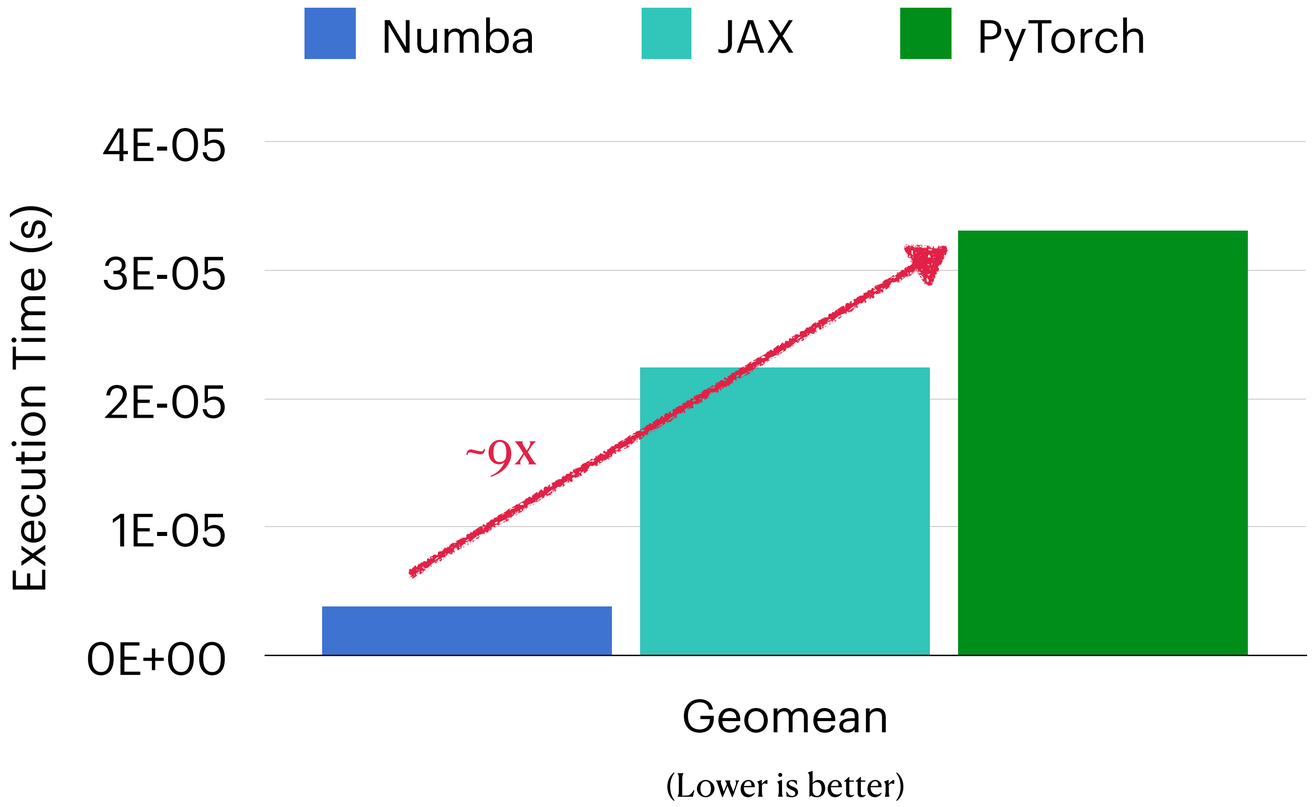


# Background

## Numba vs JAX Performance

- Numba an order of magnitude faster than JAX on this trivial benchmark
- Numba loves loopy code, whereas JAX does not

- **JIT engines conceived for very different use cases!**



# Background

## Trace-free vs Trace-based JIT

JAX frontend wrapping objects with “tracers” before distilling into jaxpr

Losing all Python side-effects

TorchDynamo tracing before handing off to TorchInductor

| Framework: | Numba | JAX | PyTorch | Dr. JIT |
|---|---|---|---|---|
| Trace-Free? | ✓ | ✗ | ✗ | ✗ |

Global Trace Accumulation As first step of JIT

# Numba-Enzyme gives us a *Trace-Free* Differentiable JIT

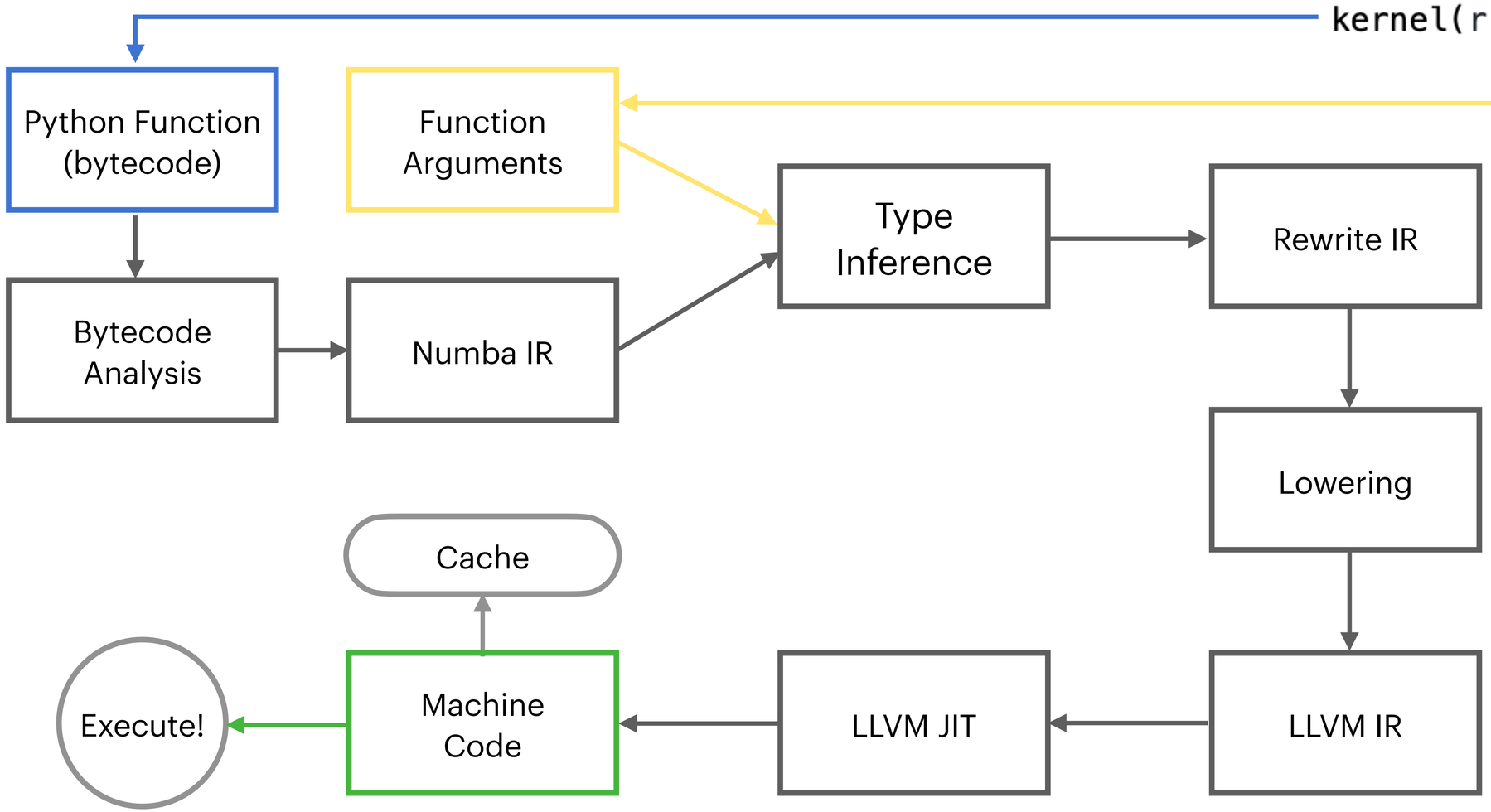
# Coinstac

## Compilation with ~~Numba~~ LLVM

```
from numba import jit

@jit(nopython=True)
def remote_stats(MSE, varX_matrix_global, avg_beta_vector):
    my_shape = avg_beta_vector.shape
    ts = np.zeros(my_shape)

    for voxel in prange(my_shape[0]):
        var_covar_beta_global = MSE[voxel] * np.linalg.inv(varX_matrix_global)
        se_beta_global = np.sqrt(np.diag(var_covar_beta_global))
        ts[voxel, :] = avg_beta_vector[voxel, :] / se_beta_global

    return ts
```

# Compilation

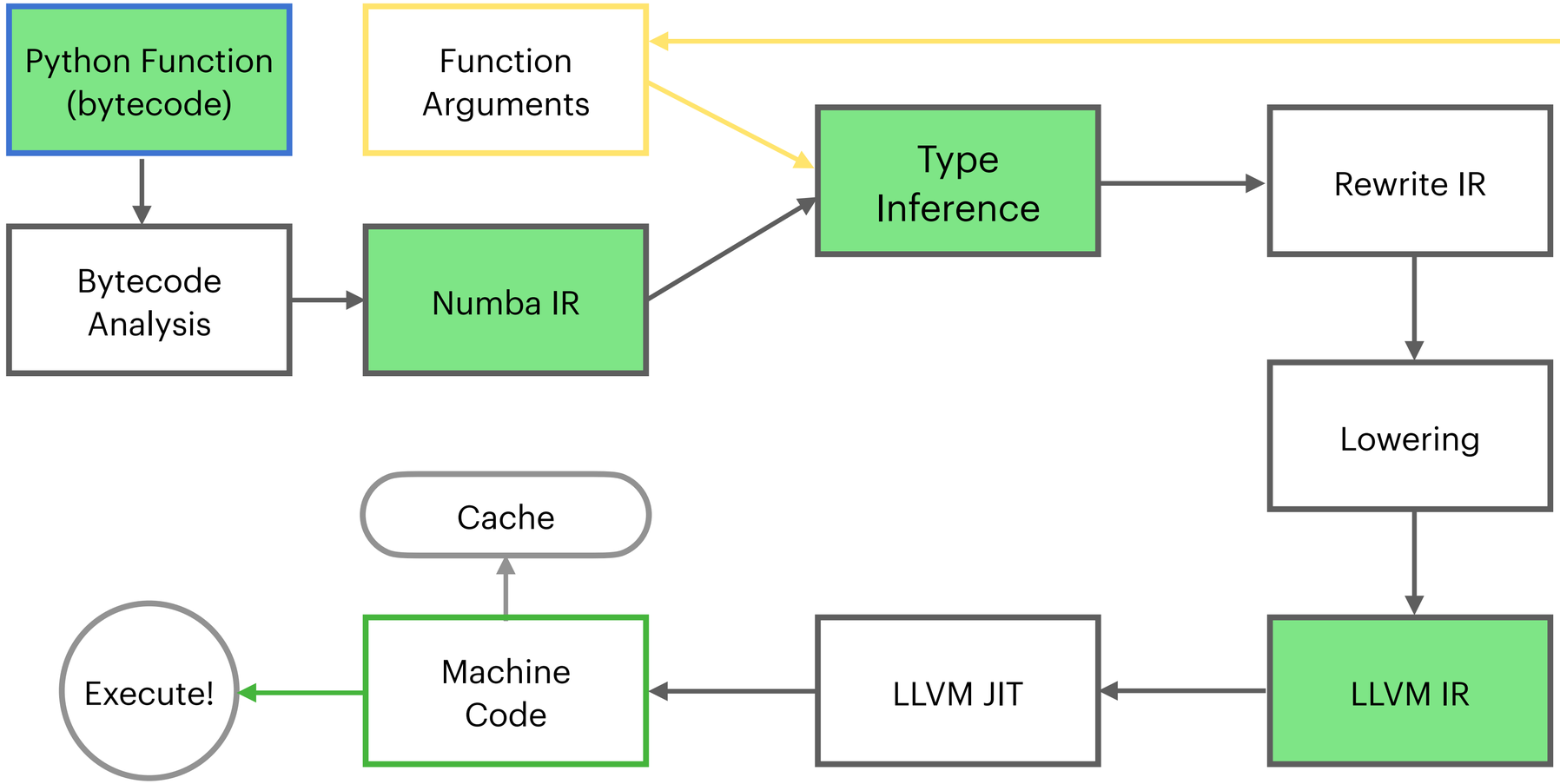
```
@njit
def kernel(r, a, x, y)
    ...

kernel(r, a, x, y)
```




Based on: Graham Markall "The Life of a Numba Kernel"

# Compilation

```
@njit
def kernel(r, a, x, y)
    ...

kernel(r, a, x, y)
```

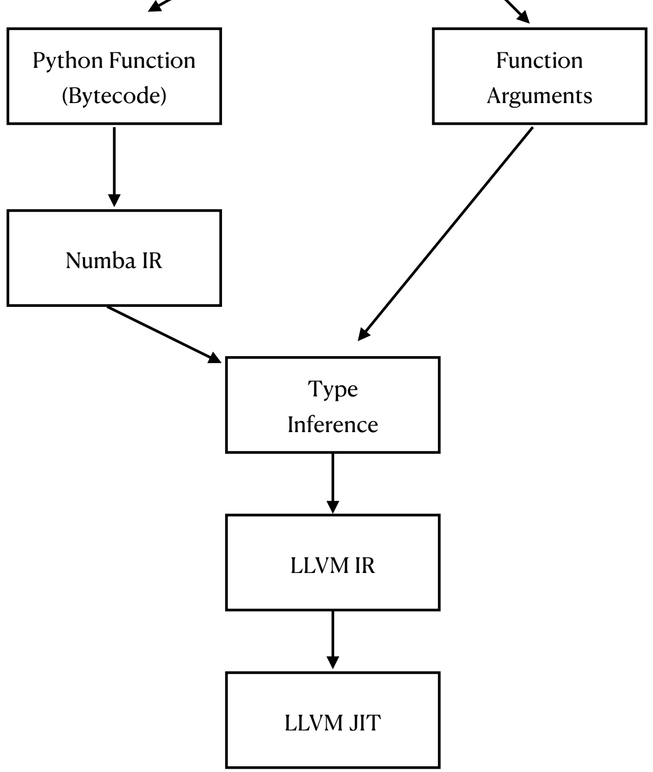



Based on: Graham Markall "The Life of a Numba Kernel"

# Numba Compilation

## Python

```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```

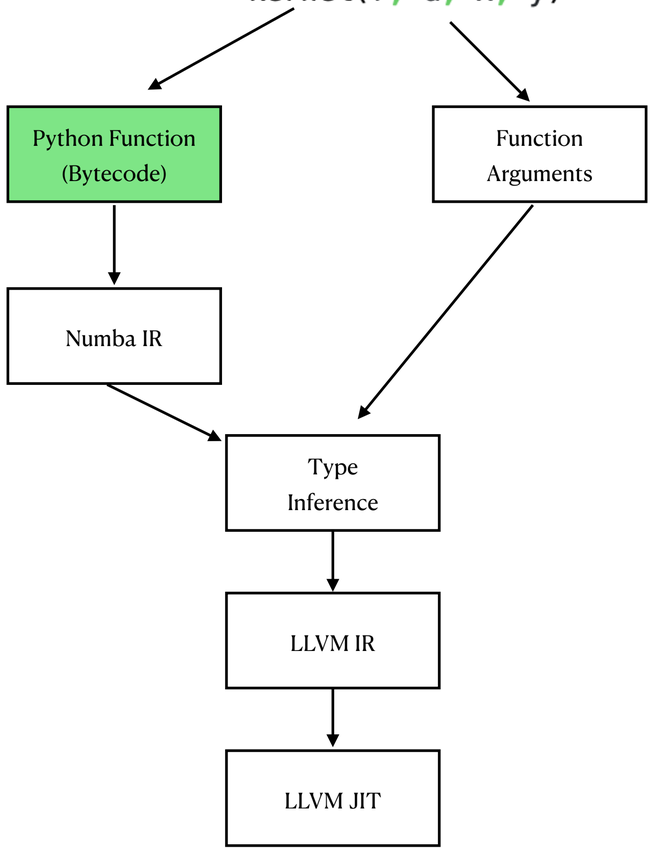



Based on: Graham Markall "The Life of a Numba Kernel"

# Numba Compilation

## Python Bytecode

```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```



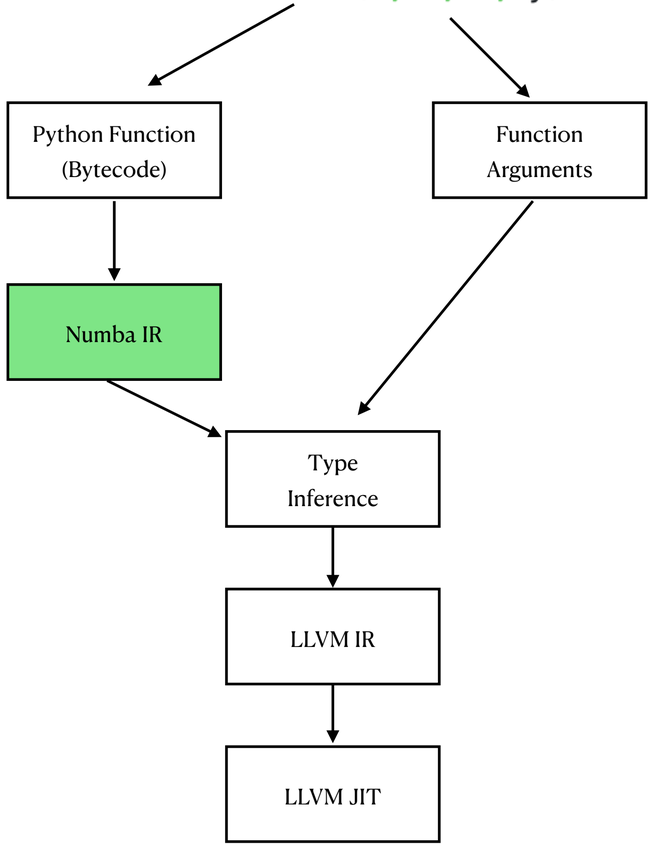

```
3           0 LOAD_GLOBAL              0 (cuda)
            2 LOAD_METHOD              1 (grid)
            4 LOAD_CONST               1 (1)
            6 CALL_METHOD              1
            8 STORE_FAST               4 (i)

5          10 LOAD_FAST                4 (i)
           12 LOAD_GLOBAL              2 (len)
           14 LOAD_FAST                0 (r)
           16 CALL_FUNCTION            1
           18 COMPARE_OP               0 (<)
           20 POP_JUMP_IF_FALSE       46

6          22 LOAD_FAST                1 (a)
           24 LOAD_FAST                2 (x)
           26 LOAD_FAST                4 (i)
           28 BINARY_SUBSCR
           30 BINARY_MULTIPLY
           32 LOAD_FAST                3 (y)
           34 LOAD_FAST                4 (i)
           36 BINARY_SUBSCR
           38 BINARY_ADD
           40 LOAD_FAST                0 (r)
           42 LOAD_FAST                4 (i)
           44 STORE_SUBSCR
      >>   46 LOAD_CONST               0 (None)
           48 RETURN_VALUE
```

Based on: Graham Markall "The Life of a Numba Kernel"

# Numba Compilation

## Untyped Numba IR

```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```



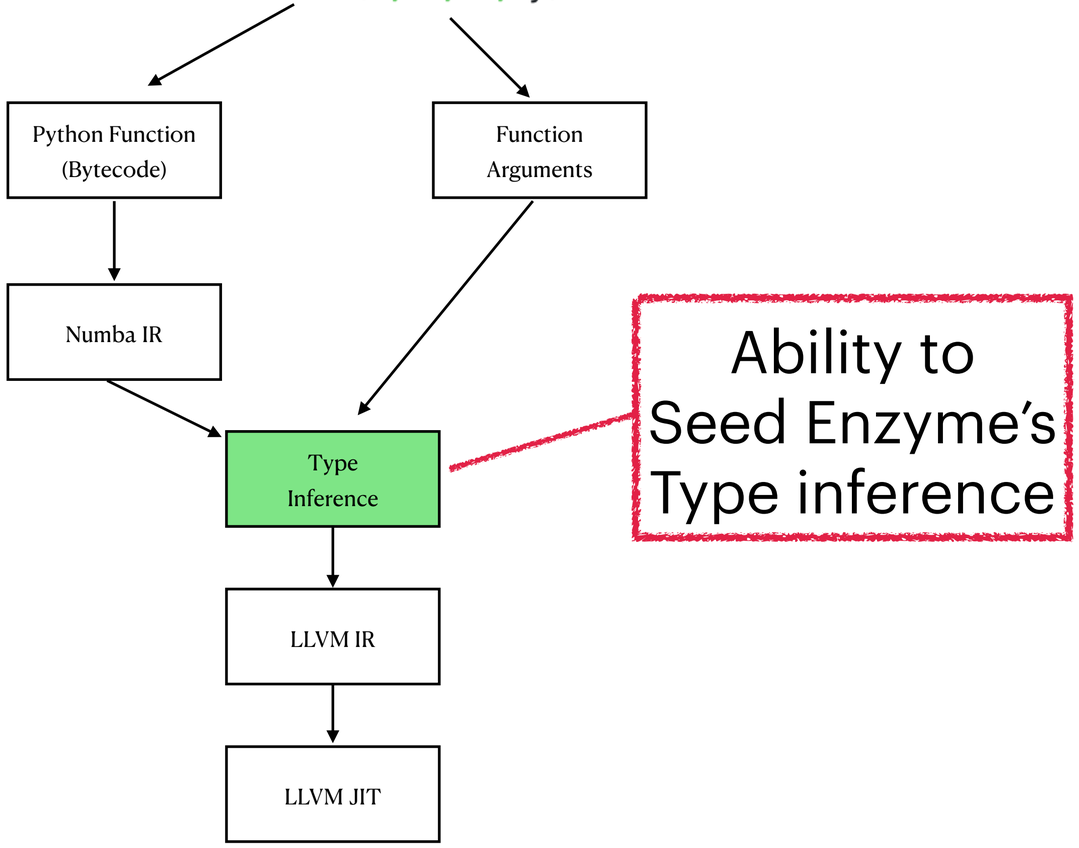

```
label 0:
    r = arg(0, name=r)                          ['r']
    a = arg(1, name=a)                          ['a']
    x = arg(2, name=x)                          ['x']
    y = arg(3, name=y)                          ['y']
    $2load_global.0 = global(cuda: <module 'numba.cuda' from '/home/gmarkall/numbadev/numba/numba/cuda/__in
    $4load_method.1 = getattr(value=$2load_global.0, attr=grid) ['$2load_global.0', '$4load_method.1']
    $const6.2 = const(int, 1)                   ['$const6.2']
    $8call_method.3 = call $4load_method.1($const6.2, func=$4load_method.1, args=[Var($const6.2, <ipython-i
ethod.1', '$8call_method.3', '$const6.2']
    i = $8call_method.3                         ['$8call_method.3', 'i']
    $14load_global.5 = global(len: <built-in function len>) ['$14load_global.5']
    $18call_function.7 = call $14load_global.5(r, func=$14load_global.5, args=[Var(r, <ipython-input-2-2505
'$18call_function.7', 'r']
    $20compare_op.8 = i < $18call_function.7 ['$18call_function.7', '$20compare_op.8', 'i']
    branch $20compare_op.8, 24, 48              ['$20compare_op.8']
label 24:
    $30binary_subscr.3 = getitem(value=x, index=i) ['$30binary_subscr.3', 'i', 'x']
    $32binary_multiply.4 = a * $30binary_subscr.3 ['$30binary_subscr.3', '$32binary_multiply.4', 'a']
    $38binary_subscr.7 = getitem(value=y, index=i) ['$38binary_subscr.7', 'i', 'y']
    $40binary_add.8 = $32binary_multiply.4 + $38binary_subscr.7 ['$32binary_multiply.4', '$38binary_subscr.
    r[i] = $40binary_add.8                      ['$40binary_add.8', 'i', 'r']
    jump 48                                     []
label 48:
    $const48.0 = const(NoneType, None)          ['$const48.0']
    $50return_value.1 = cast(value=$const48.0) ['$50return_value.1', '$const48.0']
    return $50return_value.1                    ['$50return_value.1']
```

Based on: Graham Markall "The Life of a Numba Kernel"

# Numba Compilation

## Typed Numba IR

```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```




Ability to Seed Enzyme's Type inference

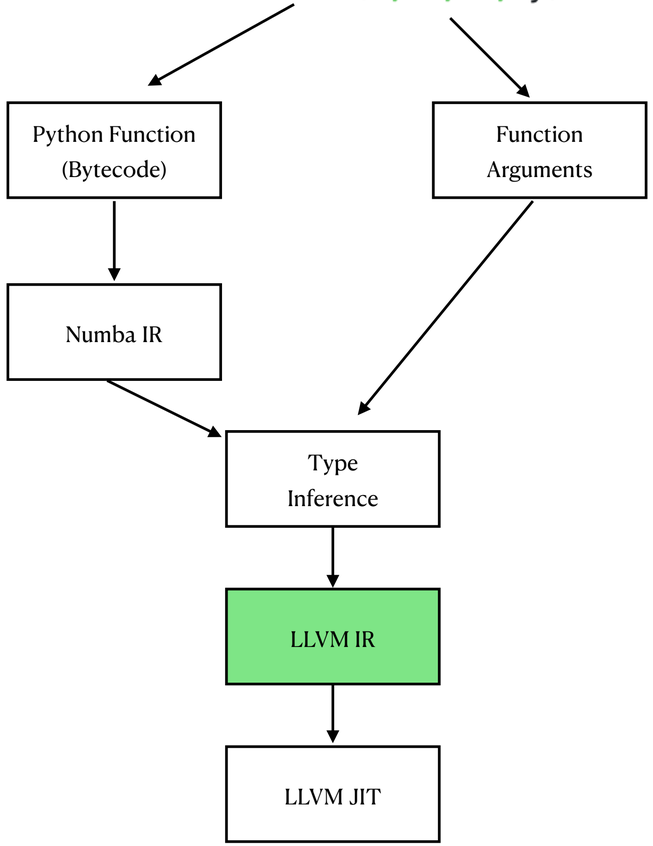
```
    # --- LINE 5 ---
    #   $14load_global.5 = global(len: <built-in function len>)  :: Function(<built-in f
    #   $18call_function.7 = call $14load_global.5(r, func=$14load_global.5, args=[Var(r
1d, C),) -> int64
    #   del $14load_global.5
    #   $20compare_op.8 = i < $18call_function.7  :: bool
    #   del $18call_function.7
    #   branch $20compare_op.8, 24, 48

    if i < len(r):

        # --- LINE 6 ---
        # label 24
        #   del $20compare_op.8
        #   $30binary_subscr.3 = getitem(value=x, index=i)  :: int32
        #   del x
        #   $32binary_multiply.4 = a * $30binary_subscr.3  :: float64
        #   del a
        #   del $30binary_subscr.3
        #   $38binary_subscr.7 = getitem(value=y, index=i)  :: int32
        #   del y
        #   $40binary_add.8 = $32binary_multiply.4 + $38binary_subscr.7  :: float64
        #   del $38binary_subscr.7
        #   del $32binary_multiply.4
        #   r[i] = $40binary_add.8  :: (array(int32, 1d, C), int64, float64) -> none
        #   del r
        #   del i
        #   del $40binary_add.8
        #   jump 48
        # label 48
        #   del y
        #   del x
        #   del r
        #   del i
        #   del a
        #   del $20compare_op.8
        #   $const48.0 = const(NoneType, None)  :: none
        #   $50return_value.1 = cast(value=$const48.0)  :: none
        #   del $const48.0
        #   return $50return_value.1

        r[i] = a * x[i] + y[i]
```

Based on: Graham Markall "The Life of a Numba Kernel"

# Numba Compilation

## LLVM IR

```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```

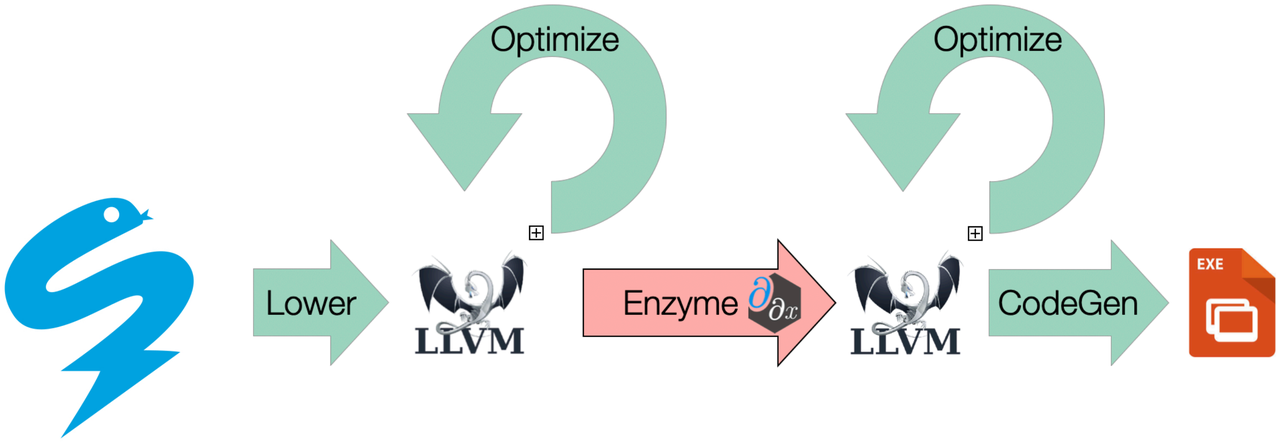



Based on: Graham Markall "The Life of a Numba Kernel"

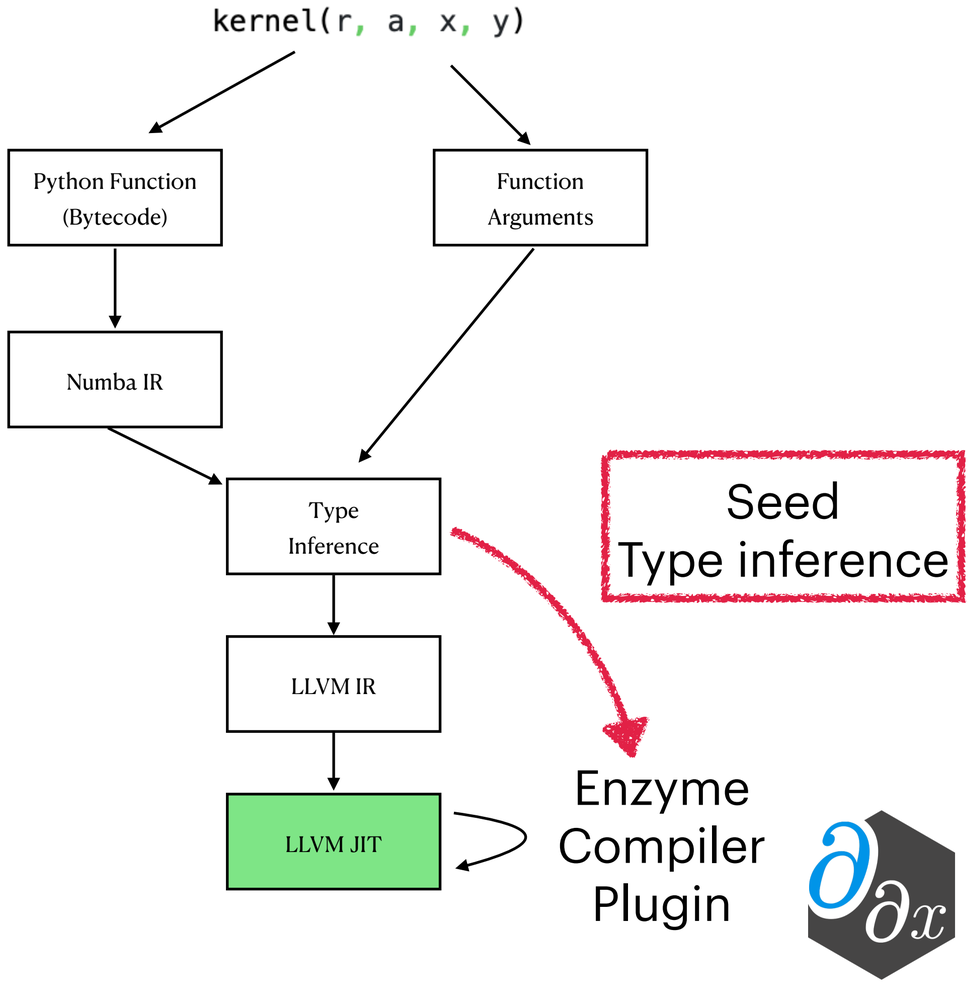
```
; Function Attrs: nounwind
define linkonce_odr i32 @"_ZN8__main__8axpy$241E5ArrayIiLi1E1C7mutable7alignedEd5ArrayIi
t, i8* %arg.r.0, i8* %arg.r.1, i64 %arg.r.2, i64 %arg.r.3, i32* %arg.r.4, i64 %arg.r.5.0
64 %arg.x.3, i32* %arg.x.4, i64 %arg.x.5.0, i64 %arg.x.6.0, i8* %arg.y.0, i8* %arg.y.1,
ocal_unnamed_addr #0 {
entry:
  %.57 = tail call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %.58 = tail call i32 @llvm.nvvm.read.ptx.sreg.ntid.x()
  %.59 = tail call i32 @llvm.nvvm.read.ptx.sreg.ctaid.x()
  %.60 = mul i32 %.59, %.58
  %.61 = add i32 %.60, %.57
  %.93 = sext i32 %.61 to i64
  %.94 = icmp slt i64 %.93, %arg.r.5.0
  br i1 %.94, label %B24, label %B48

B24:                                              ; preds = %entry
  %.119 = icmp slt i32 %.61, 0
  %.120 = select i1 %.119, i64 %arg.x.5.0, i64 0
  %.121 = add i64 %.120, %.93
  %.134 = getelementptr i32, i32* %arg.x.4, i64 %.121
  %.135 = load i32, i32* %.134, align 4
  %.143 = sitofp i32 %.135 to double
  %.144 = fmul double %.143, %arg.a
  %.168 = select i1 %.119, i64 %arg.y.5.0, i64 0
  %.169 = add i64 %.168, %.93
  %.182 = getelementptr i32, i32* %arg.y.4, i64 %.169
  %.183 = load i32, i32* %.182, align 4
  %.191 = sitofp i32 %.183 to double
  %.192 = fadd double %.144, %.191
  %.217 = select i1 %.119, i64 %arg.r.5.0, i64 0
  %.218 = add i64 %.217, %.93
  %.231 = getelementptr i32, i32* %arg.r.4, i64 %.218
  %.232 = fptosi double %.192 to i32
  store i32 %.232, i32* %.231, align 4
  br label %B48

B48:                                              ; preds = %B24, %entry
  store i8* null, i8** %.ret, align 8
  ret i32 0
}

; Function Attrs: nounwind readnone
declare i32 @llvm.nvvm.read.ptx.sreg.tid.x() #1
```

# Numba-Enzyme Pipeline

- Integration into Numba at the llvmlite-level to tie into the JIT-engine
  - Enzyme able to operate its own JIT
- No transformation passes at higher level of the pipeline for now for auto-shadows.




```
@njit
def kernel(r, a, x, y)
   ...

kernel(r, a, x, y)
```



# API

## Design Goals

- Minimally intrusive
- Explicit handling of shadows
- Full support of Numba's Numpy implementation
- Close to design example set by Enzyme.jl (GitHub.com/EnzymeAD/Enzyme.jl)
- Preserve Numba-typical handling of compiled kernels
  - Taking cues from the syntax of the CUDA-JIT:

```
@cuda.jit
def increment_a_2D_array(an_array):
    x, y = cuda.grid(2)
    if x < an_array.shape[0] and y < an_array.shape[1]:
        an_array[x, y] += 1

increment_a_2D_array[blockspergrid, threadsperblock](an_array)
```

# API

## Reverse-Mode Automatic Differentiation

```
from numba_enzyme import enzyme_reverse

@enzyme_reverse(enzyme_const="x",
                enzyme_dup="y",
                val_and_grad=False)
def product(x, y):
    return x*y

grad = product(x, y)
```

Reverse-Mode Gradient

```
from numba_enzyme import enzyme_reverse

@enzyme_reverse(enzyme_const="x",
                enzyme_dup="y",
                val_and_grad=True)
def product(x, y):
    return x*y

val, grad = product(x, y)
```

Primal + Reverse-Mode Gradient

# API

## Forward-Mode Automatic Differentiation

```
from numba_enzyme import enzyme_forward

@enzyme_forward(enzyme_const="x",
                enzyme_dup="y",
                val_and_grad=False)
def product(x, y):
    return x*y

tangent = product(x, y)
```

Forward-Mode Gradient,
Often also called the “Tangent”

```
from numba_enzyme import enzyme_forward

@enzyme_forward(enzyme_const="x",
                enzyme_dup="y",
                val_and_grad=True)
def product(x, y):
    return x*y

val, tangent = product(x, y)
```

Primal + Forward-Mode Gradient

# API

## Custom Gradient Interface

```python
from numba import njit

@njit
def handwritten_adjoint(..):

    ...

    return ...
```

Hand-written Adjoint

```python
from numba_enzyme import enzyme_reverse, custom_adjoint

@enzyme_reverse(...,
                val_and_grad=True,
                custom_adjoint=True)
def function(...):
    ...
    @custom_adjoint("handwritten_adjoint")
    b = subfunc(...)
    return ...

val, grad = function(...)
```

Custom Decorator

Hand-written adjoint inside of Reverse-mode gradient

# API

## Cross-Compilation into JAX and PyTorch

```python
from jax import jit
from numba_enzyme import enzyme_reverse

@enzyme_reverse(...,
                val_and_grad=True,
                torch_compat=True)
def numba_func(...):
    ...
    return ...

@jit
def jax_func(...):
    ...
    b = numba_func(...)
    return ...

input = ...
jax_func(input)
```

```python
import torch
from numba_enzyme import enzyme_reverse

@enzyme_reverse(...,
                val_and_grad=True,
                torch_compat=True)
def numba_func(...):
    ...
    return ...

def torch_func(...):
    ...
    b = numba_func(...)
    return ...

torch.compile(torch_func)
```

Seamless Interoperability: Automatic Registration with PyTorch & JAX

# Coinstac

## Differentiable Compilation with ~~Numba~~ LLVM

```
from numba_enzyme import enzyme_reverse

@enzyme_reverse(..). # Gradient through decorator
def remote_stats(MSE, varX_matrix_global, avg_beta_factor):
    my_shape = avg_beta_vector.shape
    ts = np.zeros(my_shape)

    for voxel in prange(my_shape[0]):
        var_covar_beta_global = MSE[voxel] * np.linalg.inv(varX_matrix_global)
        se_beta_global = np.sqrt(np.diag(var_covar_beta_global)
        ts[voxel, :] = avg_beta_vector[voxel, :] / se_beta_global

    return ts

grad = remote_stats(...)
```

# Summary

## Enzyme

- Compiler plugin for reverse-mode AD of statically analyzable LLVM IR
- Differentiates code of all languages going to the LLVM IR (C, C++, FORTRAN, Julia, Rust, Swift, Python, etc.)
- 4.2x speedup over AD before optimizations on CPU
- State-of-the-art performance with existing tools
- Very first general purpose reverse-mode AD on GPUs
- Novel GPU and AD-specific optimizations improve runtime by several orders of magnitude
- Full support for CPU parallelism (OpenMP, MPI)

# Summary

## Numba-Enzyme

- Gradient-providing JIT-Compiler in Numba
  - Forward-Mode
  - Reverse-Mode
  - Custom Gradient Interface
  - Seamless integration of kernels into JAX & PyTorch
- Performance highly competitive!
- Minimally intrusive to existing Numba code
- Full support of Numba's Numpy set
- JIT-characteristics targeted at computational science
  - Dynamism, Branches, Loops, Array mutation

# Get & Stay in Touch!

- Fully open-source
- Join our mailing list
- Join our weekly developers meeting at 1pm ET on Tuesdays!
- Official LLVM Incubator Project
  - Channel on LLVM Discord
  - Category on the LLVM Discourse