Reactant: Optimize Julia functions with MLIR & XLA

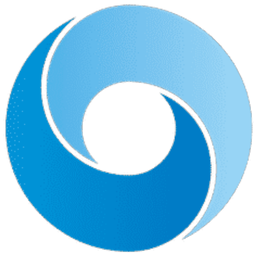
William S. Moses

wsmoses@illinois.edu

JuliaCon

Jul 25, 2025

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN



William S. Moses[†§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, Paul Berg[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], Son Vu, Sergio Sanchez-Ramirez[◇], Simone Silvestri, Nora Loose[♣], Ivan Ho, Vimarsh Sathia[†], Jan Hueckelheim[♠], Johannes De Fine Licht, Kevin Gleason[§], Ludovic Rass, Gabriel Baraldi, Dhruv Apte[#], Lorenzo Chelini[♦], Jacques Pienaar[§], Gaetan Lounes, Valentin Churavy, Sri Hari Krishna Narayanan[♠], Navid Constantinou, William R. Magro[§], Michel Schanen[♠], Alexis Montoison[♠], Alan Edelman[‡], Samarth Narang, Tobias Grosser, Keno Fischer[♮], Robert Hundt[§], Albert Cohen[§], Oleksandr Zinenko[§ *]

UIUC [†], Google [§], UCL [★], MIT [‡], NVIDIA [♦], UT Austin [#], [C]Worthy [♣], BSC [◇], Argonne National Laboratory [♠], LBNL [♡], Cambridge [♭], JuliaHub [♮], University of Mainz [♯], BFH [▽], Ghent University [△]

The Code I Want To Write != The Code I Want To Run

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```

The Code I Want To Write != The Code I Want To Run

- Allocation for result of x*x
  - Matmul computation
- Allocation for result of second x*x
  - Matmul computation
- No allocation of transpose (yay types!)
- Allocation for result of add
  - Add computation

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```

The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```

The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add
-

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```

- 2 x Allocs; 1 x Matmuls; 1 x Mul

```
function f2(x: Symmetric)
  return 2*x*x
end
```

The Code I Want To Write != The Code I Want To Run

- 3 x Allocs; 2 x Matmuls; 1 x Add

```
function f(x: Symmetric)
  return x*x + transpose(x*x)
end
```

- 2 x Allocs; 1 x Matmuls; 1 x Mul

```
function f2(x: Symmetric)
  return 2*x*x
end
```

- 1 x Allocs; 1 x Matmuls

```
function f3(x: Symmetric)
  out = similar(x)
  mul!(out, x, x, 2, 0)
end
```

Why Can’t the Compiler Fix this for Me?

```
function f(x)
  return x*x + transpose(x*x)
end
```

```
@code_typed f(y)
CodeInfo(
1 ── %1 = invoke Main.:*(x::Matrix{Float64}, x::Matrix{Float64})::Matrix{Float64}
 │    %2 = invoke Main.:*(x::Matrix{Float64}, x::Matrix{Float64})::Matrix{Float64}
 │    %3 = %new(Transpose{Float64, Matrix{Float64}}, %2)::Transpose{Float64, Matrix{Float64}}
 │    %4 = invoke Main.:+(%1::Matrix{Float64}, %3::Transpose{Float64, Matrix{Float64}})::Matrix{Float64}
 └────       return %4
) => Matrix{Float64}
```

```
julia> @code_typed y*y
CodeInfo(
1 ─── %1  = Base.arraysize(A, 1)::Int64
│    %2  = Base.arraysize(B, 2)::Int64
│    %3  = $(Expr(:foreigncall, :(:jl_alloc_array_2d), Matrix{Float64}, svec(Any, Int64, Int64), 0, :(:ccall), Matrix{Float64}, :(%1), :(%2), :(%2), :(%1)))::Matrix{Float64}
└────       goto #25 if not true
2 ──── %5  = φ (#1 => 'N', #23 => %47)::Char
│    %6  = φ (#1 => 2, #23 => %48)::Int64
└────       goto #13 if not true
3 ──── %8  = φ (#2 => 'N', #12 => %26)::Char
│    %9  = φ (#2 => 2, #12 => %27)::Int64
│    %10 = Base.bitcast(Base.UInt32, %8)::UInt32
│    %11 = Base.bitcast(Base.UInt32, %5)::UInt32
│    %12 = (%10 === %11)::Bool
└────       goto #5 if not %12
4 ───       goto #14
5 ─── %15 = Base.sle_int(1, %9)::Bool
└────       goto #7 if not %15
6 ─── %17 = Base.sle_int(%9, 3)::Bool
└────       goto #8
7 ───       nothing::Nothing
8 ──── %20 = φ (#6 => %17, #7 => false)::Bool

…

24 ──       goto #26
25 ──       goto #26
26 ── %55 = φ (#24 => false, #25 => true)::Bool
└────       goto #27
27 ──       goto #33 if not %55
28 ──       goto #30 if not true
29 ──       nothing::Nothing
30 ──       goto #32 if not true
31 ──       nothing::Nothing
32 ── %62 = invoke LinearAlgebra.gemm_wrapper!(%3::Matrix{Float64}, 'N'::Char, 'N'::Char, A::Matrix{Float64}, B::Matrix{Float64}, $(QuoteNode(LinearAlgebra.MulAddMul{true, true, Bool, Bool}(true,
false)))::LinearAlgebra.MulAddMul{true, true, Bool, Bool})::Matrix{Float64}
└────       goto #37
33 ──       goto #36 if not true
34 ──       goto #36 if not true
35 ──       nothing::Nothing
36 ── %67 = invoke LinearAlgebra._generic_matmatmul!(%3::Matrix{Float64}, 'N'::Char, 'N'::Char, A::Matrix{Float64}, B::Matrix{Float64}, $(QuoteNode(LinearAlgebra.MulAddMul{true, true, Bool, Bool}(true,
false)))::LinearAlgebra.MulAddMul{true, true, Bool, Bool})::Matrix{Float64}
└────       goto #37
37 ── %69 = φ (#32 => %62, #36 => %67)::Matrix{Float64}
└────       goto #38
38 ──       goto #39
39 ──       return %69
) => Matrix{Float64}
```

What is MLIR?

Multi-Level Intermediate Representation (MLIR)

- New Compiler IR with user-defined instructions, optimizations, analyses
  - Linear Algebra
  - GPU Programming
  - Fully Homomorphic Encryption
- Mix and match dialects and optimizations from multiple dialects
- Core infrastructure of modern ML frameworks (JaX, PyTorch, TensorFlow)
- Frontends for C++ (Polygeist), Julia (Reactant.jl, this talk :))

```
func @set(%arg0: memref<?xi32>, %arg1: i32) {
    affine.for %arg2 = 0 to 10 {
      affine.store %arg1, %arg0 [2 * %arg2] : memref<?xi32>
    }
    return
  }
}
```

GPU Programming via LLVM

- Mainstream compilers do not have a high-level representation of parallelism, making optimization difficult or impossible
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly and synchronization is treated as a complete optimization barrier.

```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

Host Code

```
target triple = "x86_64-unknown-linux-gnu”

define void @_Z6launchPiS_i(i32* %out,
                            i32* %in,
                            i32 %n) {
  call i32 @pushCallConfiguration(…)
  call i32 @cudaLaunch(@_device_stub, …)
  ret void
}
```

Device Code

```
target triple = ”nvptx”

define void @_Z9normalize(i32* %out,
                          i32* %in, i32 %n) {
  %4 = call i32 @llvm.tid.x()
  %5 = icmp slt i32 %4, %n
  br i1 %5, label %6, label %13

6:
  %8 = getelementptr i32, i32* %in, i32 %4
  %9 = load i32, i32* %8, align 4
  %10 = call i32 @_Z3sumPii(i32* %in, i32 %n)
  %11 = sdiv i32 %9, %10
  %12 = getelementptr i32, i32* %out, i32 %4
  store i32 %11, i32* %12, align 4
  br label %13

13:
  ret void
}
```

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index

  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]
    %sum = call @_Z3sumPii(%in, %n)
    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs, PPoPP'23

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int *in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index
  %sum = call @_Z3sumPii(%in, %n)
  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]

    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs, PPoPP'23

Preserving the GPU parallel structure [1]

- Polygeist/MLIR maintains GPU parallelism in a compiler-friendly form
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization
- Optimizations on primal => ***outsized impact for derivatives***

```
__global__ void normalize(int *out, int *in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index
  %sum = call @_Z3sumPii(%in, %n)
  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]

    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

[1] High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs, PPoPP'23

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

```
x + 0 -> x

transpose(transpose(x)) -> x

transpose(matmul(a,b)) ->
   matmul(b, a)
```

Often require program context

```
transpose(convert(reshape(x)))
 <-> reshape(convert(transpose(x))

slice(add(a, b)) ->
 add(slice(a), slice(b))


mul(pad(x, 0), y) ->
 pad(mul(x, slice(y)), 0)
```

```
x, y : tensor<100000xf32>

a = dot(x, y)

b = mul(a, z)

c = add(b, 4)

return c[0:10]
```

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

```
x + 0 -> x

transpose(transpose(x)) -> x

transpose(matmul(a,b)) ->
   matmul(b, a)
```

Often require program context

```
transpose(convert(reshape(x)))
 <-> reshape(convert(transpose(x))

slice(add(a, b)) ->
 add(slice(a), slice(b))


mul(pad(x, 0), y) ->
 pad(mul(x, slice(y)), 0)
```

```
x, y : tensor<100000xf32>

a = dot(x, y)

b = mul(a, z)

c = add(b[0:10], 4)

return c
```

Linear Algebra Optimizations

Wrote >200 different patterns!

Simplify code where possible

```
x + 0 -> x

transpose(transpose(x)) -> x

transpose(matmul(a,b)) ->
   matmul(b, a)
```

Often require program context

```
transpose(convert(reshape(x)))
 <-> reshape(convert(transpose(x))

slice(add(a, b)) ->
 add(slice(a), slice(b))


mul(pad(x, 0), y) ->
 pad(mul(x, slice(y)), 0)
```

```
x, y : tensor<100000xf32>

a = dot(x, y)

b = mul(a[0:10], z[0:10])

c = add(b, 4)

return c
```

Performance Engineering a Toy LLM

- Introduction of linear-algebra specific optimizations made a significant impact on training performance
  - 53% speedup of a run with 32x accelerators
  - 14-18.5% speedup for single accelerator

- Accessible now from github.com/EnzymeAD/Enzyme-JaX

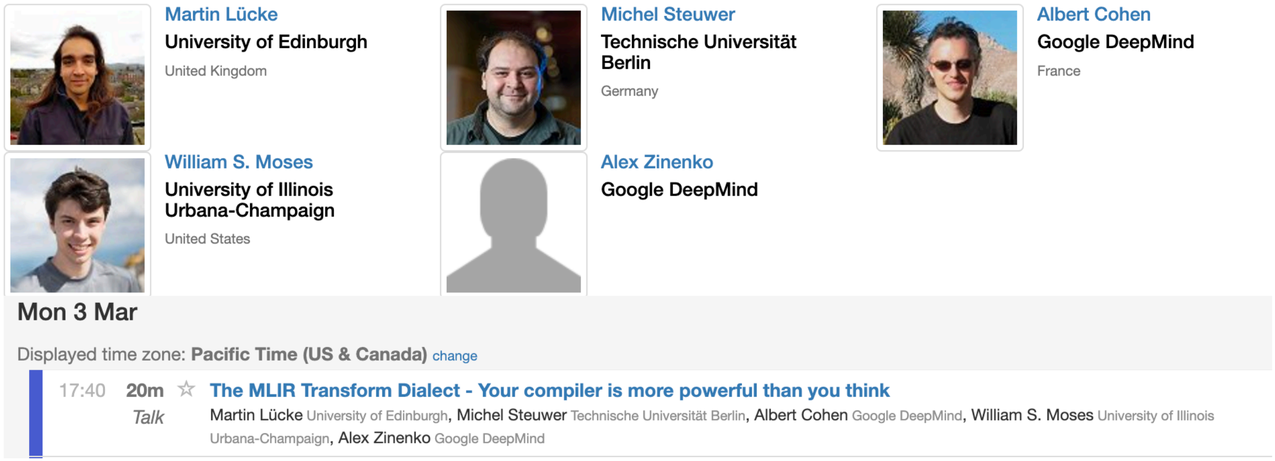
```
Step: 2 loss: 12.880576133728027
Step: 3 loss: 12.785988807678223
Step: 4 loss: 12.652521133422852
Step: 5 loss: 12.482083320617676
…
Step: 19 loss: 9.190348625183105
Step: 20 loss: 8.928218841552734
Step: 21 loss: 8.660679817199707

Unopt: 0.479 samples/sec
```

```
Step: 2 loss: 12.88175106048584
Step: 3 loss: 12.786417007446289
Step: 4 loss: 12.652612686157227
Step: 5 loss: 12.482114791870117
…
Step: 19 loss: 9.189879417419434
Step: 20 loss: 8.929146766662598
Step: 21 loss: 8.659639358520508

Opt: 0.736 samples/sec
```

Performance Engineering a Toy LLM

- Introduction of linear-algebra specific optimizations made a significant impact on training performance
  - 53% speedup of a run with 32x accelerators
  - 14-18.5% speedup for single accelerator

- Accessible now from github.com/EnzymeAD/Enzyme-JaX




```
Step: 2 loss: 12.880576133728027
Step: 3 loss: 12.785988807678223
Step: 4 loss: 12.652521133422852
Step: 5 loss: 12.482083320617676
…
Step: 19 loss: 9.190348625183105
Step: 20 loss: 8.928218841552734
Step: 21 loss: 8.660679817199707

Unopt: 0.479 samples/sec
```

```
Step: 2 loss: 12.88175106048584
Step: 3 loss: 12.786417007446289
Step: 4 loss: 12.652612686157227
Step: 5 loss: 12.482114791870117
…
Step: 19 loss: 9.189879417419434
Step: 20 loss: 8.929146766662598
Step: 21 loss: 8.659639358520508

Opt: 0.736 samples/sec
```

Reactant.jl

- Optimizing Compiler Framework For Julia, built on top of MLIR
- Preserves high-level structures from Julia code into the compiler
  - Effectively optimize programs
  - Effectively retarget programs (allowing them to run on distributed clusters of your favorite accelerator, including CPU/GPU/TPU)
- Can leverage XLA (state of the art runtime, used by TensorFlow, JaX, & PyTorch)
- Compatible with mutation, control flow(*), autodiff (Enzyme)
- All open source (GitHub.com/EnzymeAD/Reactant.jl)

Reactant.jl Usage

```
using Reactant

x = Reactant.to_rarray(ones(3,3))

function f(x)
   x * x + transpose(x * x)
end

r_f = @compile f(x)

r_f(x)
```

- Reactant has 3 core primitives:
  - Reactant.ConcreteArray
  - @compile
  - Reactant.TracedRArray

Reactant.jl ConcreteArrays

```
julia> x = Reactant.to_rarray(ones(3,3))

2025-07-25 09:09:49.875290: I external/xla/xla/service/service.cc:163] XLA
service 0x11e08f0 initialized for platform CUDA (this does not guarantee that
XLA will be used).

julia> r_f(x)
3×3 ConcretePJRTArray{Float64,2}:
 6.0  6.0  6.0
 6.0  6.0  6.0
 6.0  6.0  6.0

julia> Reactant.devices()
2-element Vector{Reactant.XLA.PJRT.Device}:
 Reactant.XLA.PJRT.Device(Ptr{Nothing} @0x00000000013d21a0, "CUDA:0 NVIDIA
GeForce RTX 5090")
 Reactant.XLA.PJRT.Device(Ptr{Nothing} @0x0000000000aefd90, "CUDA:1 NVIDIA
GeForce RTX 5090")
```

Reactant.jl Compilation

```
julia> @code_hlo optimize=false f(x)
module @reactant_f attributes {mhlo.num_partitions = 1 : i64, mhlo.num_replicas = 1 : i64} {
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_1"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_2"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_3"(%arg0: tensor<f64>) -> tensor<f64> {
    return %arg0 : tensor<f64>
  }
  func.func private @"+_broadcast_scalar"(%arg0: tensor<f64>, %arg1: tensor<f64>) -> (tensor<f64>, tensor<f64>, tensor<f64>) {
    %0 = stablehlo.add %arg0, %arg1 : tensor<f64>
    return %0, %arg0, %arg1 : tensor<f64>, tensor<f64>, tensor<f64>
  }
  func.func @main(%arg0: tensor<3x3xf64> {tf.aliasing_output = 1 : i32}) -> (tensor<3x3xf64>, tensor<3x3xf64>) {
    %0 = stablehlo.transpose %arg0, dims = [1, 0] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %1 = stablehlo.convert %0 : tensor<3x3xf64>
    %2 = stablehlo.convert %0 : tensor<3x3xf64>
    %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %3 = stablehlo.broadcast_in_dim %1, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %4 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar"(%3) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %5 = stablehlo.convert %4 : tensor<3x3xf64>
    %cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %6 = stablehlo.broadcast_in_dim %2, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %7 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_1"(%6) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %8 = stablehlo.convert %7 : tensor<3x3xf64>
    %9 = stablehlo.dot_general %5, %8, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>, tensor<3x3xf64>) -> tensor<3x3xf64>
    %cst_2 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %10 = stablehlo.convert %0 : tensor<3x3xf64>
    %11 = stablehlo.convert %0 : tensor<3x3xf64>
    %cst_3 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %12 = stablehlo.broadcast_in_dim %10, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %13 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_2"(%12) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %14 = stablehlo.convert %13 : tensor<3x3xf64>
    %cst_4 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
    %15 = stablehlo.broadcast_in_dim %11, dims = [0, 1] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %16 = enzyme.batch @"Reactant.TracedUtils.TypeCast{Float64}()_broadcast_scalar_3"(%15) {batch_shape = array<i64: 3, 3>} : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %17 = stablehlo.convert %16 : tensor<3x3xf64>
    %18 = stablehlo.dot_general %14, %17, contracting_dims = [1] x [0], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>, tensor<3x3xf64>) -> tensor<3x3xf64>
    %cst_5 = stablehlo.constant dense<0.000000e+00> : tensor<3x3xf64>
```

Reactant.jl Compilation

```
julia> @code_hlo f(x)
module @reactant_f attributes {mhlo.num_partitions = 1 : i64, mhlo.num_replicas = 1 : i64} {
  func.func @main(%arg0: tensor<3x3xf64>) -> tensor<3x3xf64> {
    %0 = stablehlo.dot_general %arg0, %arg0, contracting_dims = [0] x [1], precision = [DEFAULT, DEFAULT] : (tensor<3x3xf64>,
tensor<3x3xf64>) -> tensor<3x3xf64>
    %1 = stablehlo.transpose %0, dims = [1, 0] : (tensor<3x3xf64>) -> tensor<3x3xf64>
    %2 = stablehlo.add %1, %0 : tensor<3x3xf64>
    return %2 : tensor<3x3xf64>
  }
}

julia> @code_xla f(x)
HloModule reactant_f, is_scheduled=true, entry_computation_layout={(f64[3,3]{1,0})->f64[3,3]{1,0}},
frontend_attributes={fingerprint_before_lhs="e0828ffd4833737b459c41de309ba417"}

%fused_add (param_0.1: f64[3,3]) -> f64[3,3] {
  %param_0.1 = f64[3,3]{1,0} parameter(0)
  %transpose.2 = f64[3,3]{1,0} transpose(%param_0.1), dimensions={1,0}
  ROOT %add.4.1 = f64[3,3]{1,0} add(%transpose.2, %param_0.1), metadata={op_name="add" source_file="/mnt3/wmoses/git/Reactant.jl/
src/Ops.jl" source_line=375}
}

ENTRY %main.5 (Arg_0.1: f64[3,3]) -> f64[3,3] {
  %Arg_0.1 = f64[3,3]{1,0} parameter(0), metadata={op_name="arg1 (path=(:args, 1))"}
  %custom-call.1 = (f64[3,3]{1,0}, s8[144]{0}) custom-call(%Arg_0.1, %Arg_0.1), custom_call_target="__cublas$gemm",
metadata={op_name="dot_general" source_file="/mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=808},
backend_config={"operation_queue_id":"0","wait_on_operation_queues":[],"gemm_backend_config":
{"alpha_real":1,"beta":0,"dot_dimension_numbers":{"lhs_contracting_dimensions":["0"],"rhs_contracting_dimensions":
["1"],"lhs_batch_dimensions":[],"rhs_batch_dimensions":[]},"alpha_imag":0,"precision_config":{"operand_precision":
["DEFAULT","DEFAULT"],"algorithm":"ALG_UNSET"},"epilogue":"DEFAULT","lhs_stride":"9","rhs_stride":"9","grad_x":false,"grad_y":false,
"damax_output":false},"force_earliest_schedule":false,"reification_cost":[]}
  %get-tuple-element.1 = f64[3,3]{1,0} get-tuple-element(%custom-call.1), index=0, metadata={op_name="dot_general" source_file="/
mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=808}
  ROOT %loop_add_fusion = f64[3,3]{1,0} fusion(%get-tuple-element.1), kind=kLoop, calls=%fused_add, metadata={op_name="add"
source_file="/mnt3/wmoses/git/Reactant.jl/src/Ops.jl" source_line=375}
}
```

Reactant.jl TracedArrays

- ConcreteArray : Reactant-specific array type (which could have information stored across multiple devices)

- TracedArray: Reactant-specific array type used during compilation which represent data whose values are not compile-time constants

```
julia> function g(x)
           @show x
           x * x + transpose(x * x)
       end

julia> r_f = @compile g(x)
x = TracedRArray{Float64,2N}(((:args, 1),), size=(3, 3))
```

Reactant.jl TracedArrays

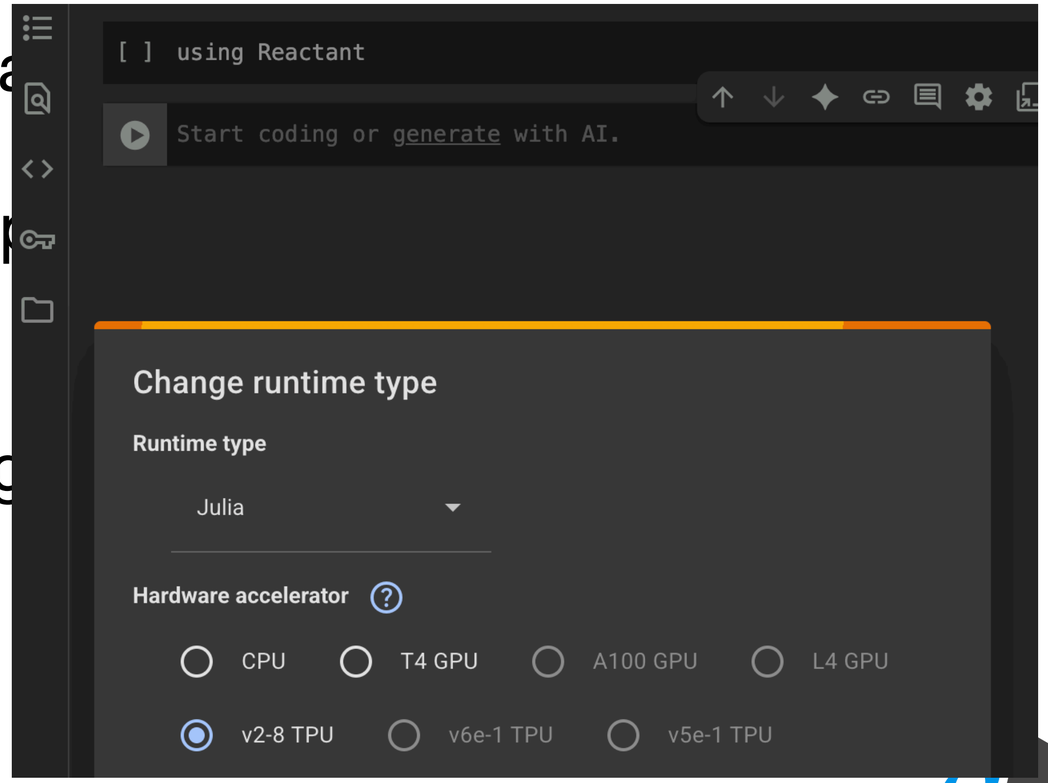
```
julia> function h(cond, x)
          if cond
             return x
          else
             return x*x
          end
       end
h (generic function with 2 methods)

julia> cond = Reactant.to_rarray(true; track_numbers=Number)
ConcretePJRTNumber{Bool}(true)

julia> @jit h(cond, x)
ERROR: TypeError: non-boolean (Reactant.TracedRNumber{Bool}) used in boolean context

julia> function h2(cond, x)
          @trace if cond
             x
          else
             x*x
          end
       end
h2 (generic function with 1 method)

julia> @jit h2(cond, x)
1-element Vector{ConcretePJRTArray{Float64, 2, 1}:
 ConcretePJRTArray{Float64, 2, 1}([1.0 1.0 1.0; 1.0 1.0 1.0; 1.0 1.0 1.0])
```

Reactant.jl Backend and Communication

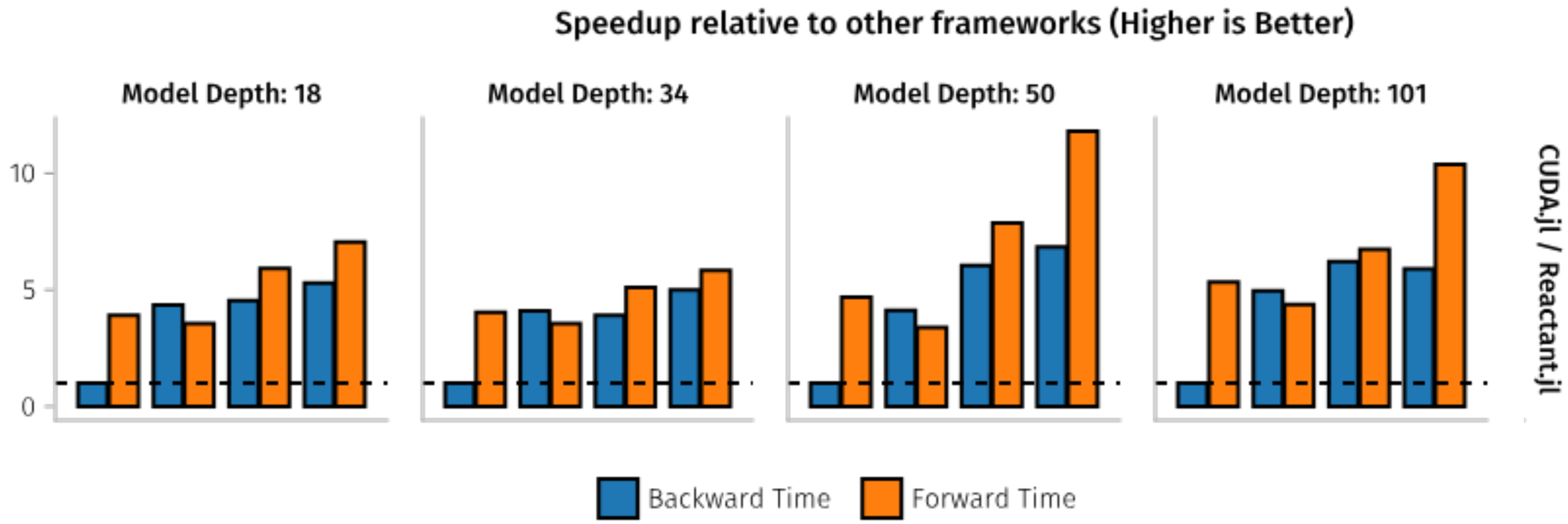
- Reactant enables you to write regular Julia code which will automatically be distributed across different accelerators
  - Under the hood, Reactant recognizes what devices you have available
  - Generate corresponding communication primitives (e.g. MPI/NCCL/etc), kernels, and library calls (BLAS, cuBLAS, tpuBLAS) to execute
- No need to modify your application to retarget different systems

Reactant.jl Backend and Communication

- Reactant enables you to write regular Julia code which will automatically be distributed across different accelerators
  - Under the hood, Reactant recognizes wha
  - Generate corresponding communication p kernels, and library calls (BLAS, cuBLAS,
- No need to modify your application to retarg



ML Performance Comparison



ML Performance Comparison

JuliaCon 2025

Schedule Sessions Speakers

Accelerating Machine Learning in Julia using Lux & Reactant

2025-07-25 11:30–12:00, Lawrence Room 107 - Method Room

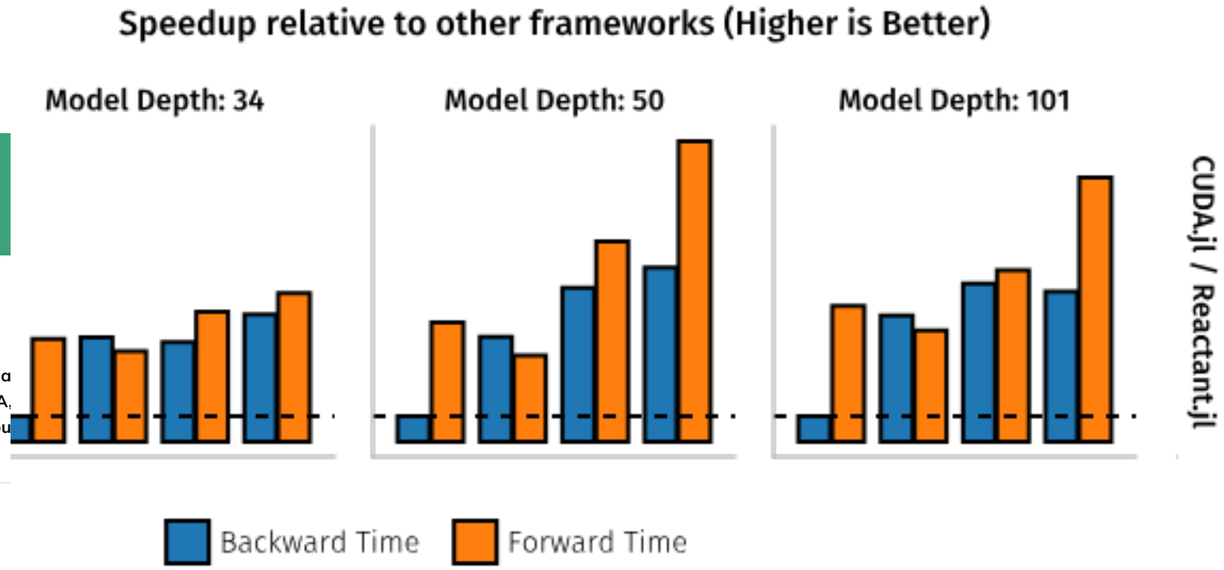
This talk will explore the latest advancements and current state of Lux.jl, a deep-learning framework in Julia. We will a to use Reactant.jl, a powerful tool that compiles Julia code to MLIR and executes it across various backends using XLA, session will highlight how Reactant.jl and Lux.jl enable training neural networks in Julia at speeds comparable to popu like JAX and PyTorch.

We will cover the following topics:

1. Compiling Lux models using Reactant and EnzymeJAX
2. Model-Parallel and Data-Parallel Training: Implement scalable training strategies in Lux.jl using Reactant.jl and Open distributing models and data across multiple devices.
3. Compiler Optimizations with Enzyme and JAX.
4. Debugging Performance with TensorBoard Profiler: Use TensorBoard Profiler to identify and resolve performance bott and inference pipelines.
5. Visualizing Models with Model Explorer: Explore and analyze neural network architectures interactively using Model E debugging and understanding.
6. Exporting Models to JAX and TensorFlow: Seamlessly export trained models from Lux.jl to JAX and TensorFlow for dep production environments.

Avik Pal

Ph.D. Student @ MIT

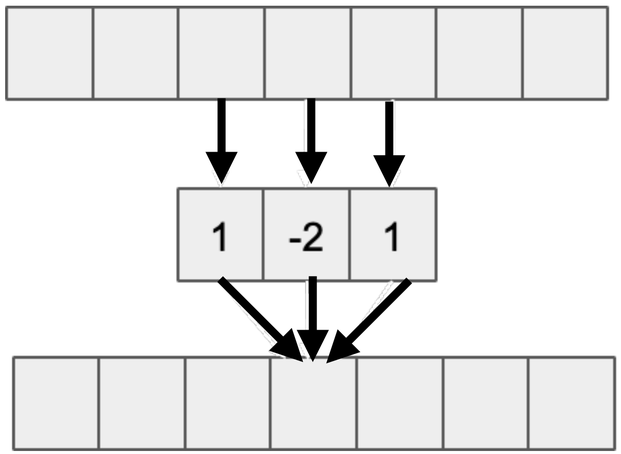


Accelerating Scientific Code: Raising Kernels to Convolutions

- Scientific codes (like state-of-the-art model, Oceananigans.jl) maintain hundreds of handwritten kernels, preventing them from using the advanced tensor capabilities of modern ML accelerators.

- Yet the core computations are often similar to ML (stencil kernel <-> convolution)

```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) *
blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i+1] + x[i+2]
  end
end

function model(...)
  @cuda threads=... blocks=... stencil_kernel(y, x)
  @cuda threads=... blocks=... stencil_kernel(x, y)
end
```



Accelerating Scientific Code

- New framework for raising and optimizing the structure within existing kernels to tensor IR!

  1) Compile Kernels to LLVM

  2) Raise the underlying structure in MLIR

  3) Multi-dimensionalize it into tensor operators

  4) Optimize

- Compiled single-node CUDA version of Oceananigans.jl to execute on thousands of distributed TPUs and GPUs

```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i+1] + x[i+2]
  end
end

function model(...)
  @cuda threads=... blocks=... stencil_kernel(y, x)
  @cuda threads=... blocks=... stencil_kernel(x, y)
end
```

Compilation

```
define void @julia_difference_kernel_890({}* %y, {}* %x) {
top:
  %3 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %4 = add nuw nsw i32 %3, 1
  ...
  br i1 %.not, label %common.ret, label %L31
}
```

Raising

```
func.func @kernel(%y : memref<100xf64>, %x : memref<100xf64>) {
  affine.parallel %arg1 = 0 to 100 {
    %x1 = affine.load %x[%arg1]
    %x2 = affine.load %x[%arg1 + 1]
    ...
    affine.store %sum, %y[%arg1]
  }
}
```

Multi-Dimensionalization

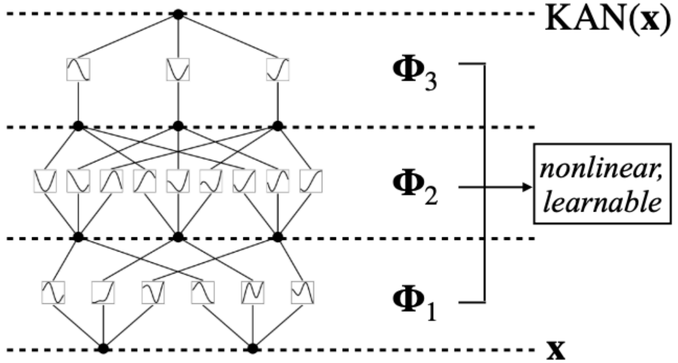
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

```
res = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)

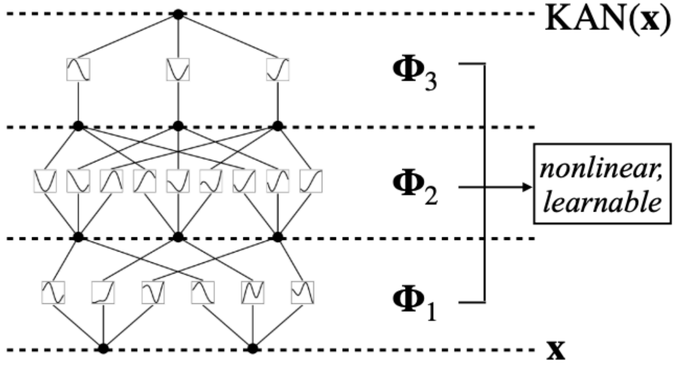
CUDA KAN network




```
Forward (regular Julia)
 47.586 us ( 248 allocations)
234.233 us (1022 allocations)
134.028 us ( 668 allocations)

Backwards (Zygote + Julia)
 289.319 us ( 575 allocations)
2099.000 us (1055 allocations)
1772.000 us ( 877 allocations)

Forward (Reactant)
39.873 us (  2 allocations)
68.439 us (  6 allocations)
55.889 us (  6 allocations)

Backwards (EnzymeMLIR + Reactant)
  51.691 us (  3 allocations)
 104.193 us (  3 allocations)
  80.020 us (  3 allocations)
```

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)

CUDA KAN network



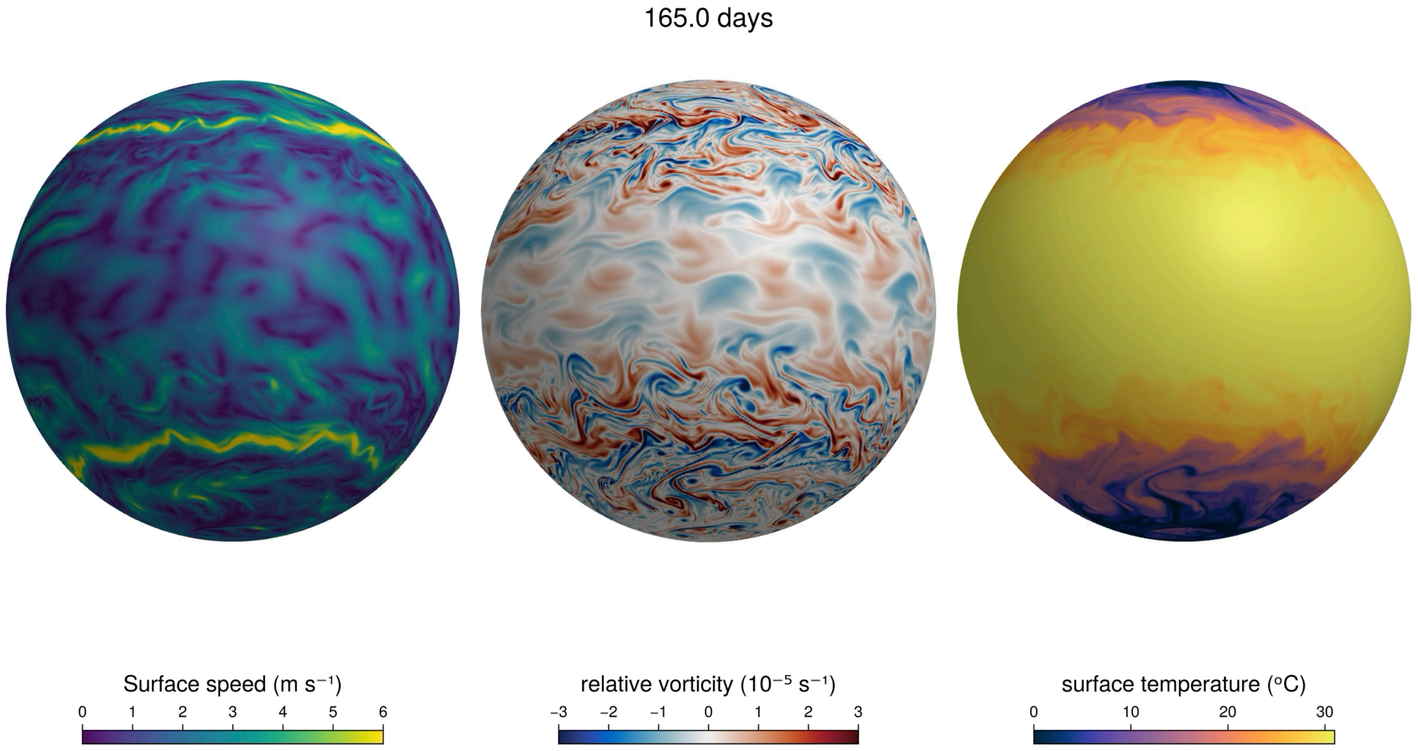

```
Forward (regular Julia)
 47.586 us ( 248 allocations)
234.233 us (1022 allocations)
134.028 us ( 668 allocations)

Forward (Reactant)
 39.873 us (  2 allocations)
 68.439 us (  6 allocations)
 55.889 us (  6 allocations)
```

```
Backwards (Zygote + Julia)
 289.319 us ( 575 allocations)
2099.000 us (1055 allocations)
1772.000 us ( 877 allocations)

Backwards (EnzymeMLIR + Reactant)
   51.691 us (  3 allocations)
  104.193 us (  3 allocations)
   80.020 us (  3 allocations)
```

2.14x speedup (Primal)

13.57x speedup (Derivative)

Accelerating Scientific Code




```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i+1] + x[i+2]
  end
end

function model(...)
  @cuda threads=... blocks=... stencil_kernel(y, x)
  @cuda threads=... blocks=... stencil_kernel(x, y)
end
```

Compilation

```
define void @julia_difference_kernel_890({}* %y, {}* %x) {
top:
  %3 = call i32 @llvm.nvvm.read.ptx.sreg.tid.x()
  %4 = add nuw nsw i32 %3, 1
  ...
  br i1 %.not, label %common.ret, label %L31
}
```

Raising

```
func.func @kernel(%y : memref<100xf64>, %x : memref<100xf64>) {
  affine.parallel %arg1 = 0 to 100 {
    %x1 = affine.load %x[%arg1]
    %x2 = affine.load %x[%arg1 + 1]
    ...
    affine.store %sum, %y[%arg1]
  }
}
```

Multi-Dimensionalization

```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

```
res = stablehlo.convolve %x, tensor<[1.0, -4.0, 6.0, -4.0, 1.0]>
```

Reactant.jl

- Optimizing Compiler Framework For Julia, built on top of MLIR
- Preserves high-level structures from Julia code into the compiler
  - Effectively optimize programs
  - Effectively retarget programs (allowing them to run on distributed clusters of your favorite accelerator, including CPU/GPU/TPU)
- Can leverage XLA (state of the art runtime, used by TensorFlow, JaX, & PyTorch)
- Compatible with mutation, control flow(*), autodiff (Enzyme)
- All open source (GitHub.com/EnzymeAD/Reactant.jl)