Automatic differentiation and performance portability for Oceananigans via Enzyme + Reactant

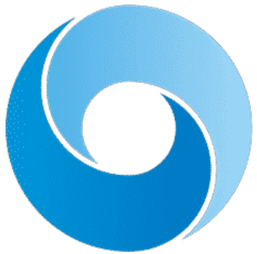
William S. Moses

wsmoses@illinois.edu

ECCO Meeting

May 29, 2026

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN



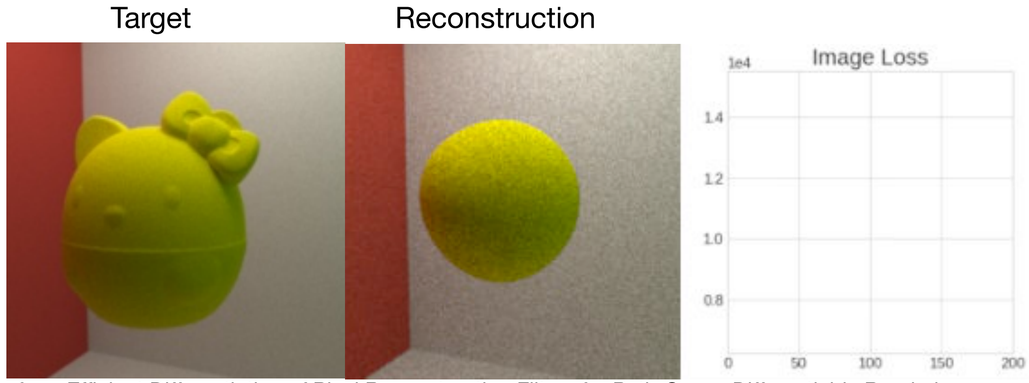
William S. Moses[†§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, Paul Berg[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], Son Vu, Sergio Sanchez-Ramirez[◇], Simone Silvestri, Nora Loose[♣], Ivan Ho, Vimarsh Sathia[†], Jan Hueckelheim[♠], Johannes 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 [△]

Differentiation: Connecting Science and AI

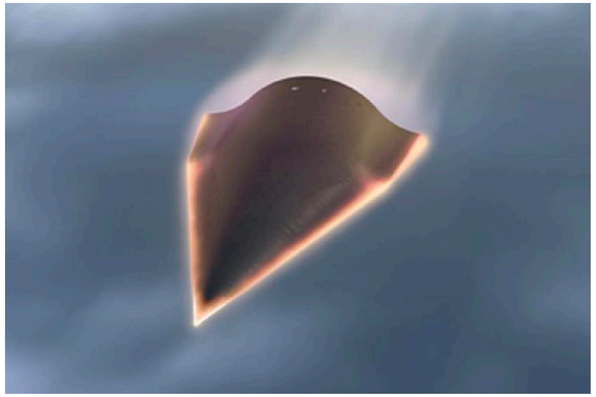
Derivatives are key to science + ML

- *Scientific Computing*: UQ, Differential Equation, Error Analysis
- *Machine Learning*: Back-Propagation, Bayesian Inference

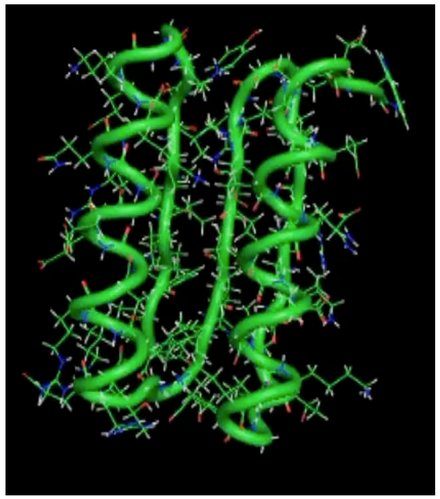


from Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering, SIGGRAPH Asia 2022, Zihan Yu et al

from CLIMA & NSF CSSI: Differentiable programming in Julia for Earth system modeling (DJ4Earth)



from Center for the Exascale Simulation of Materials in Extreme Environments

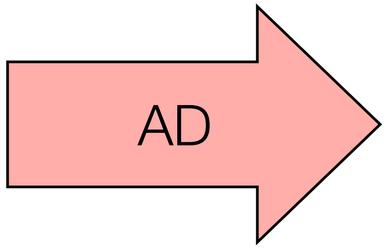


from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

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 derivative of ALL inputs (or outputs) at once, without approximation error!

```
// Numeric differentiation
// f'(x) approx [f(x+epsilon) - f(x)] / epsilon
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)
```

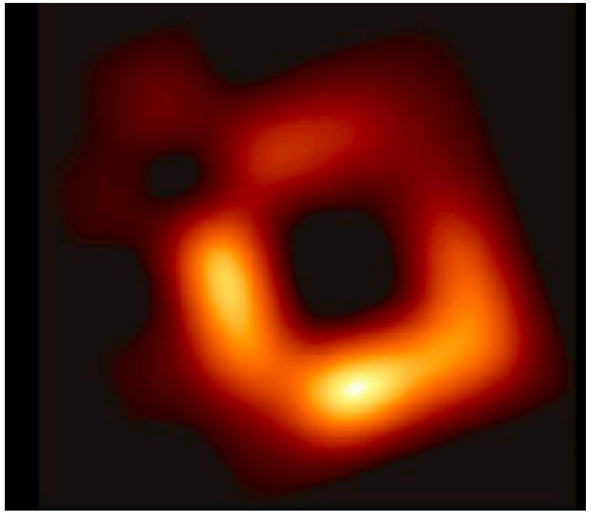
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

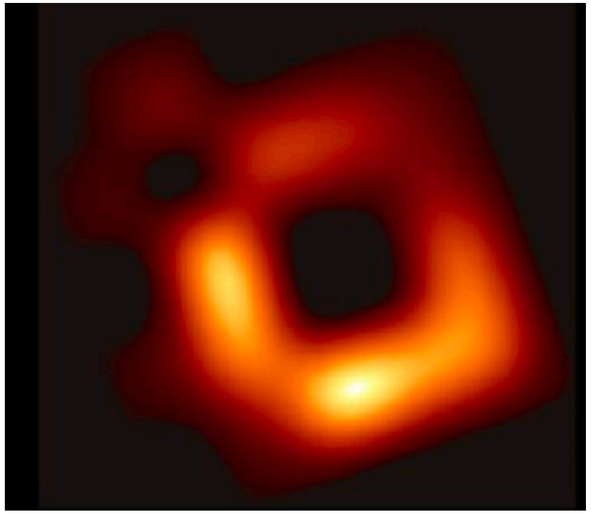
Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative

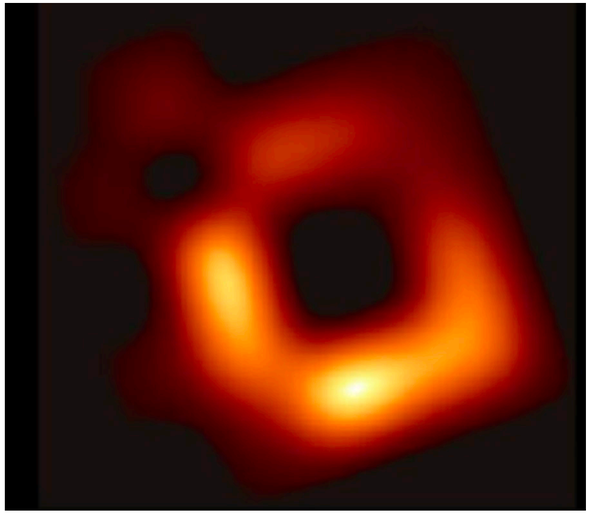


With Enzyme differentiation:
1 hour on 1 thread

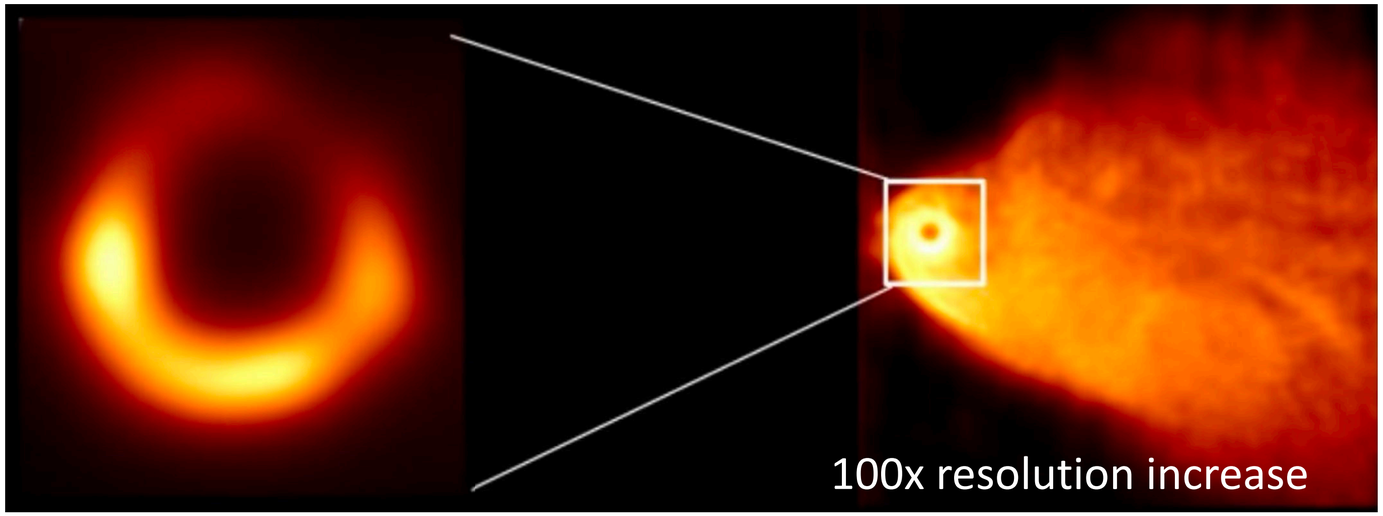
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

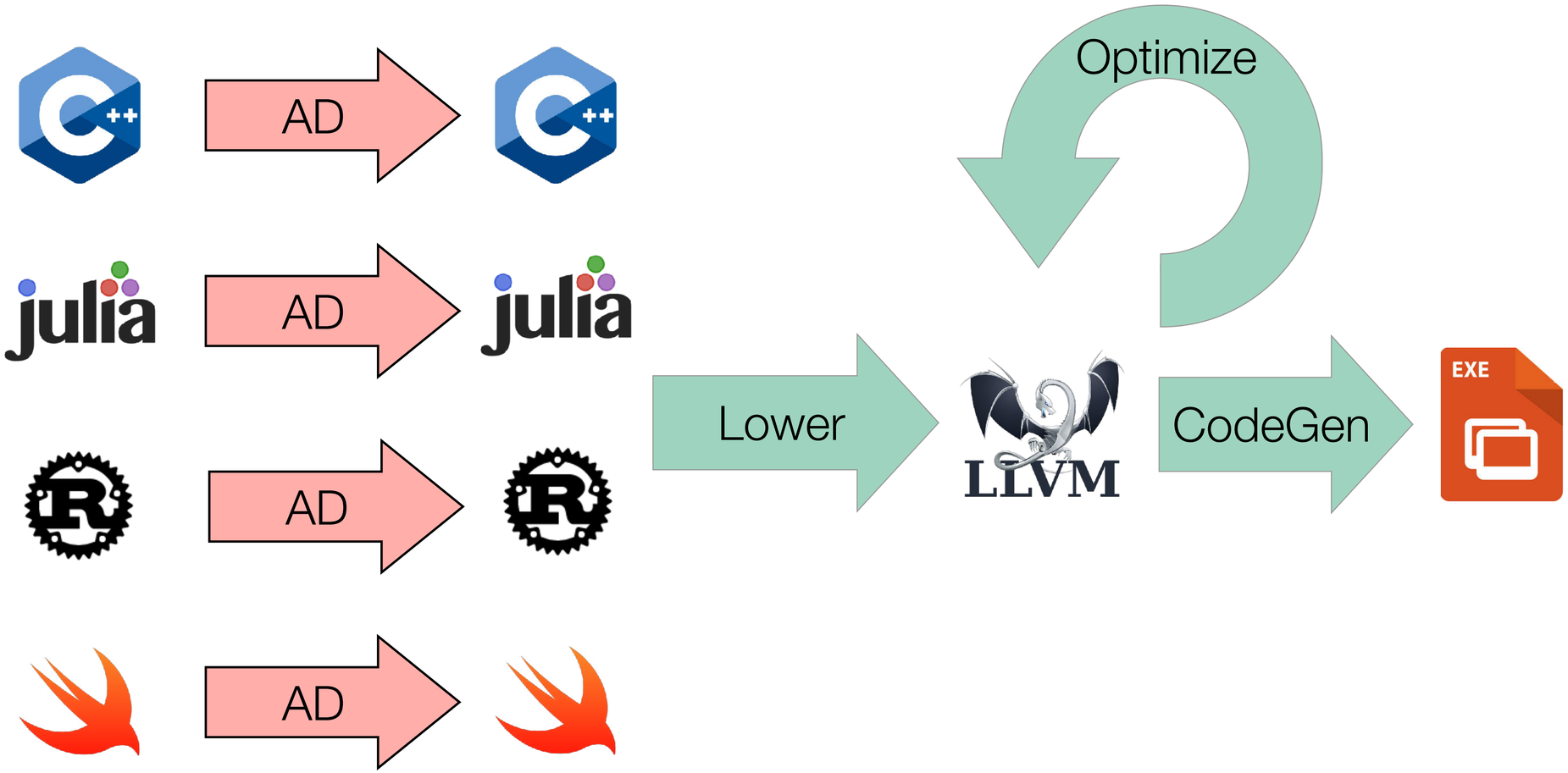
Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



With Enzyme differentiation:
1 hour on 1 thread



Existing Automatic Differentiation Pipelines

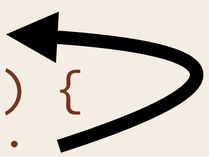


Case Study: Vector Normalization

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



Optimization & Automatic Differentiation

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

Optimization & Automatic Differentiation

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

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

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

AD

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

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimization & Automatic Differentiation

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

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

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

AD

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

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimize

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

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

Optimization & Automatic Differentiation

Differentiating after optimization can create ***asymptotically faster*** gradients!

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

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

Optimize

$O\left(n\right)$

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

AD

$O\left(n\right)$

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

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

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

AD

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

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

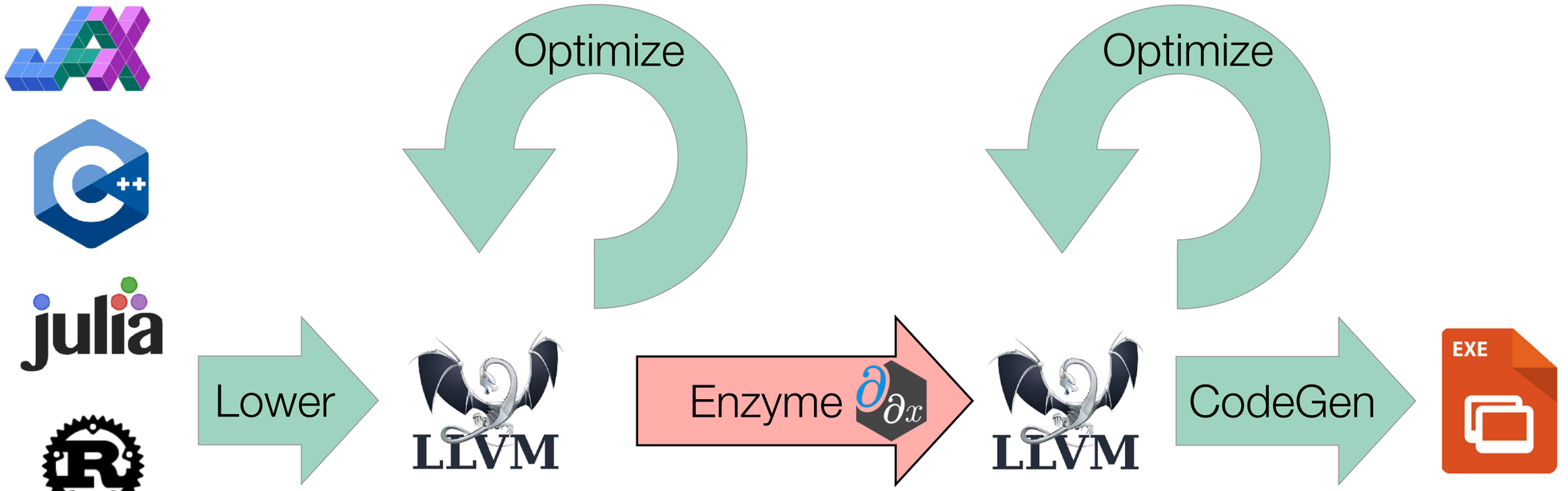
Optimize

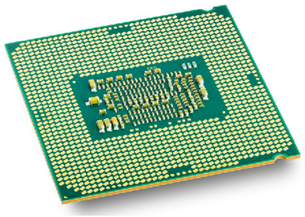
$O\left(n^2\right)$

```
for i=n..0 {
  d_res = d_out[i]…
  ∇mag(d_in, d_res)
}
```

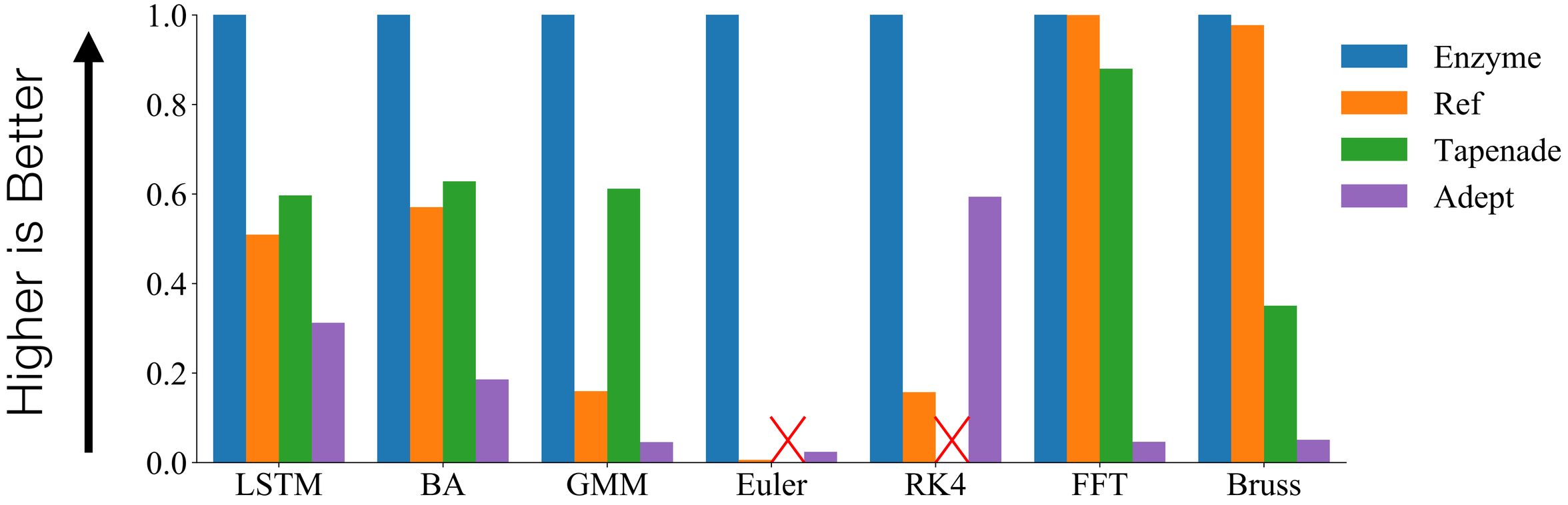
Enzyme Approach

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





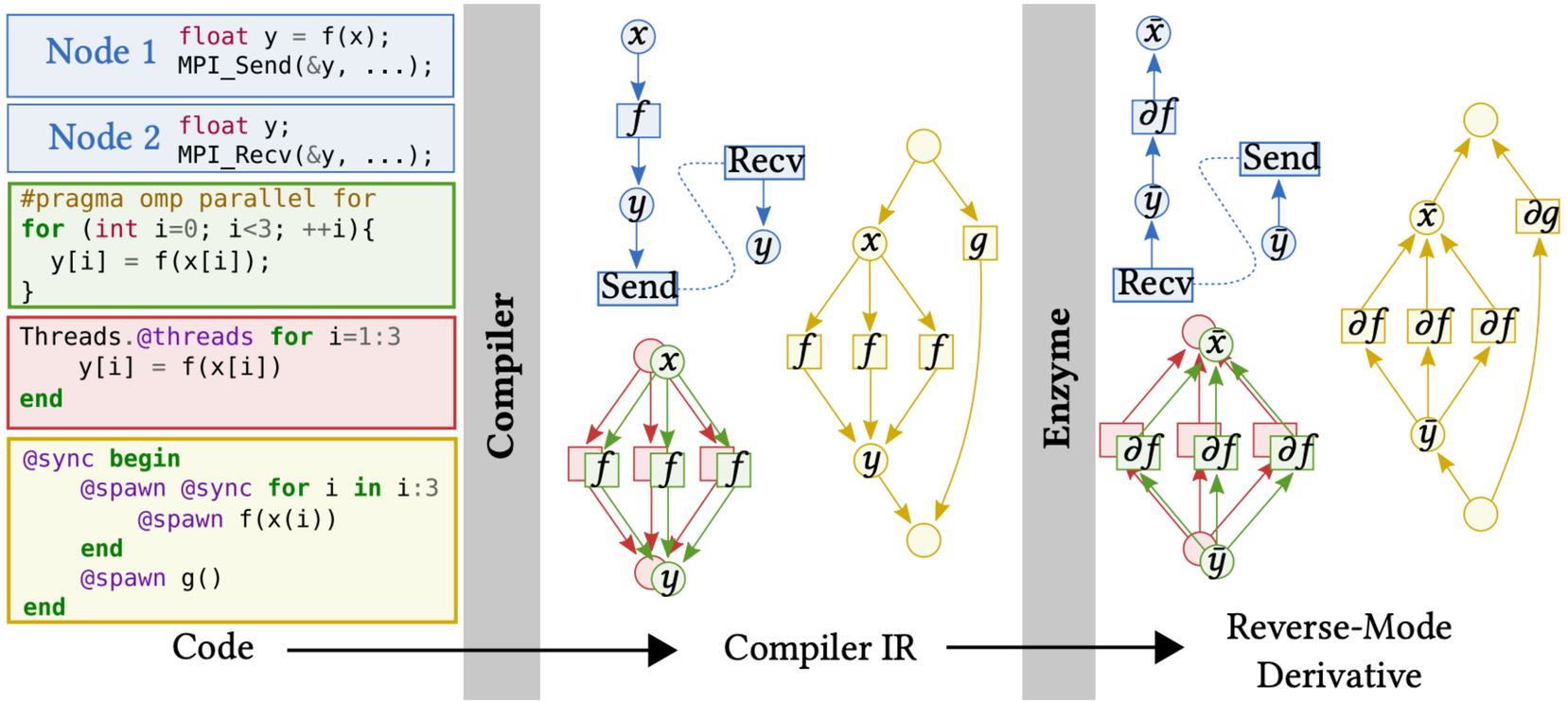
Enzyme CPU Speedups [NeurIPS'20]



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

Common Framework for Parallel AD [SC'22]

- Common infrastructure for supporting parallel AD (caching, race-resolution, gradient accumulation) enables parallel differentiation independent of framework or language.



- Enables differentiation of a combination of GPU (e.g. CUDA, ROCm), CPU (OpenMP, Julia Tasks, RAJA), Distributed (MPI, MPI.jl), and more

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?

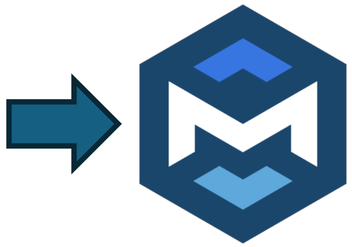
```
function f(x)
  return x*x + transpose(x*x)
end
```

```
@code_typed f(x)

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}
```

Reactant: Automatically Synthesize Efficient Code for All Devices

Reactant Compiler (C++, Julia, Python)

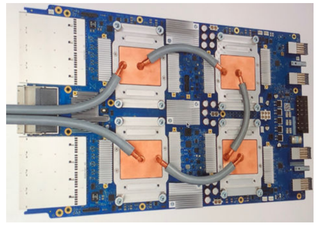


Domain specific optimizations & analyses (linear algebra, Enzyme, etc)

Generate MLIR

```
for i in axes(A, 1)
  tmp[i] = 0
  for j in axes(A, 2)
    tmp[i] += A[i, j] * x[j]
  end
  for j in axes(A, 2)
    y[j] += A[i, j] * tmp[i]
  end
end
```

Distributed Hardware-Specific Codegen + Optimization



Optimization passes

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)

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

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
```

How Does Raising & Tensor Transformations Impact AD?

Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients

How Does Raising & Tensor Transformations Impact AD?

- Biggest impact in three primary areas:
  - Work-Reduction + Fusion
  - Checkpointing
  - Communication

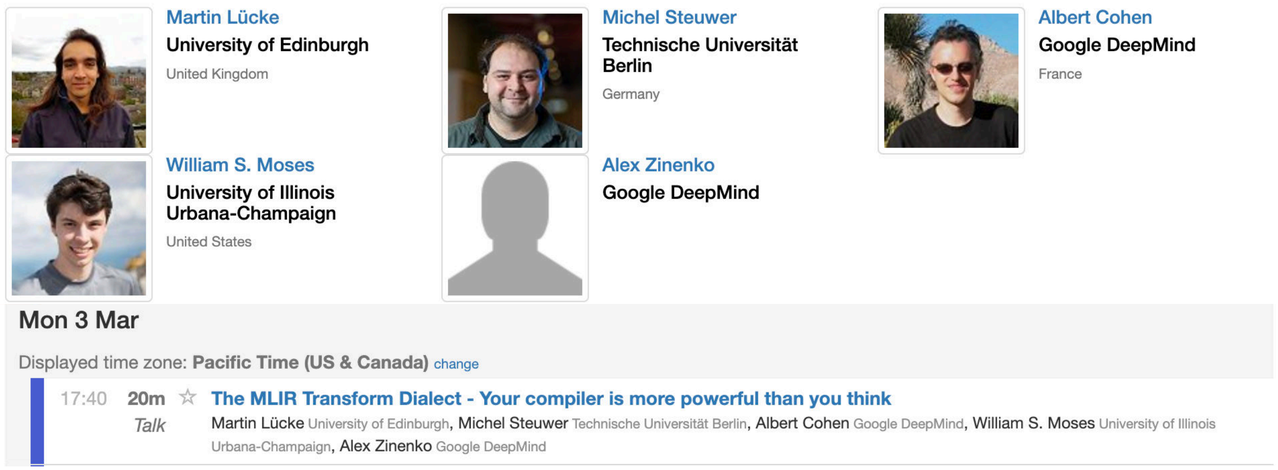
Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients

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

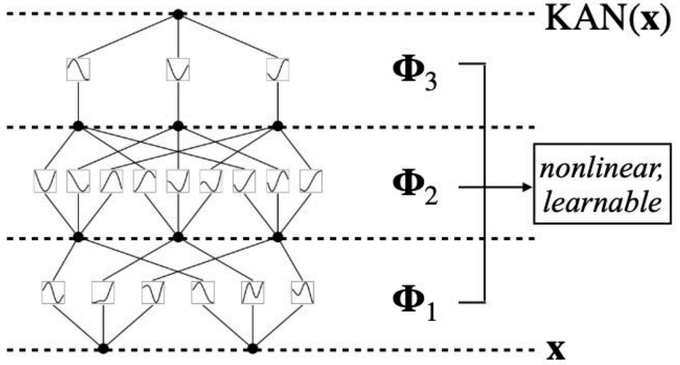



```
Step: 2 loss: 12.880576133728027
Step: 3 loss: 12.785988807678223
Step: 4 loss: 12.652521133422852
Step: 5 loss: 12.482083320617676
…
Step: 19 loss: 9.190348625183105
Step: 20 loss: 8.928218841552734
Step: 21 loss: 8.660679817199707

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
```

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)

CUDA KAN network

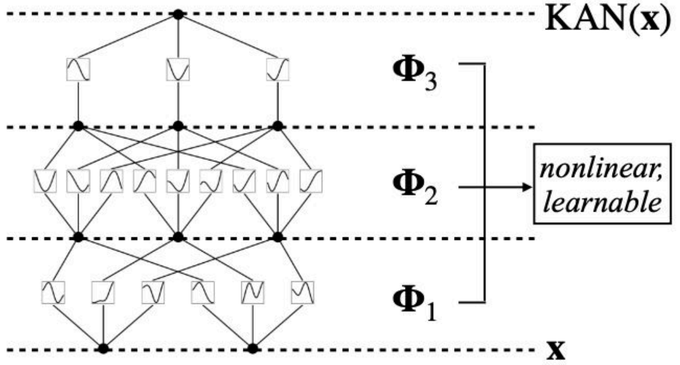



```
Forward (regular Julia)
 47.586 us ( 248 allocations)
234.233 us (1022 allocations)
134.028 us ( 668 allocations)

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)
```

EnzymeMLIR in Julia (via Reactant.jl MLIR Frontend)

CUDA KAN network



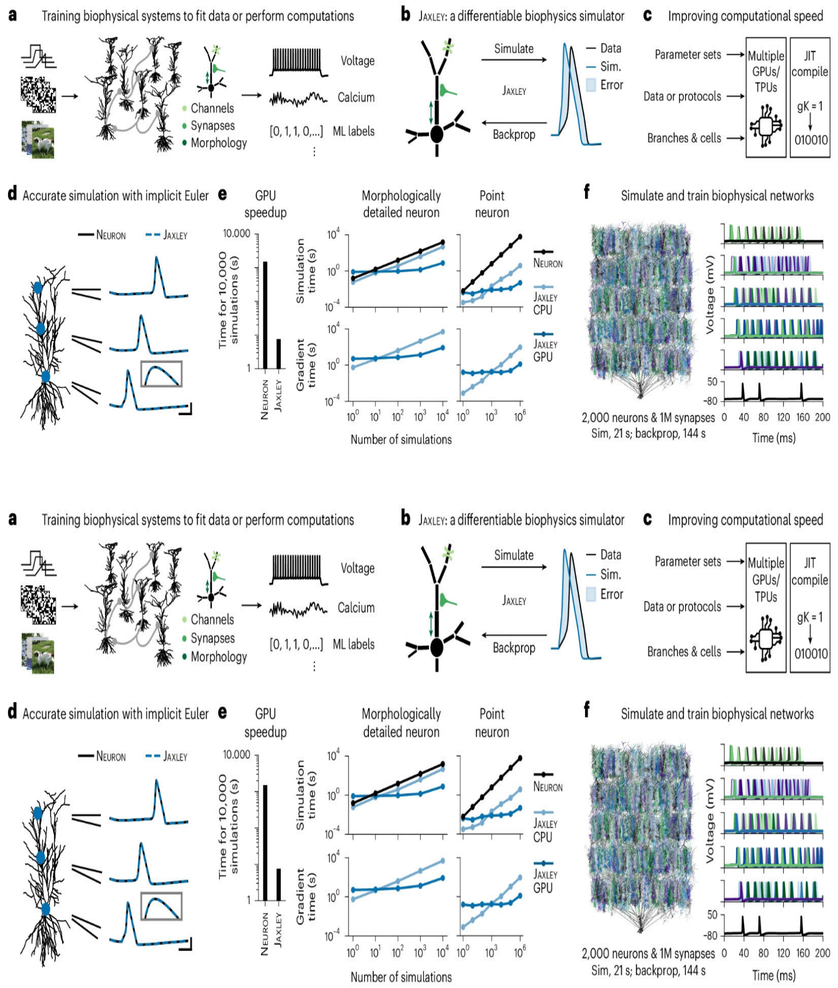

```
Forward (regular Julia)
 47.586 us ( 248 allocations)
234.233 us (1022 allocations)
134.028 us ( 668 allocations)

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)
```

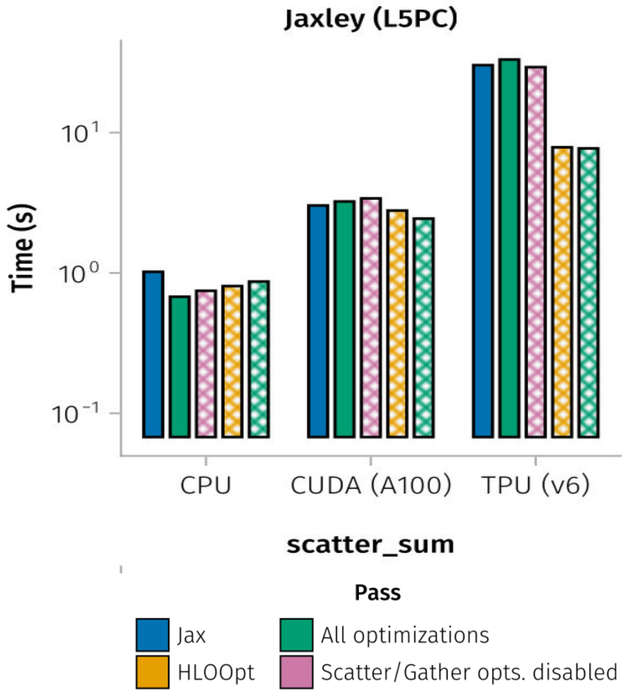
2.14x speedup (Primal)

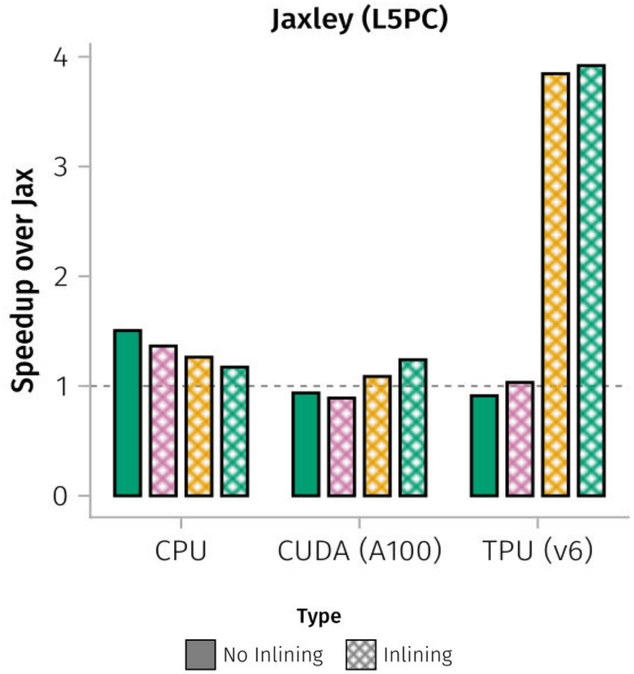
13.57x speedup (Derivative)

Work Reduction Benchmark: Jaxley



1.15x speedup on CPU
1.33x speedup on A100
3.92x speedup on TPU v6





[1] Deistler, Michael, et al. "Jaxley: differentiable simulation enables large-scale training of detailed biophysical models of neural dynamics." Nature Methods (2025): 1-9.

Computing Hardware is No Longer For Everybody

Computing Hardware is No Longer For Everybody

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer’s Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World’s Smallest AI Supercomputer Capable of Running 200B-Parameter Models

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By **Katie Paul** and Krystal Hu

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium December 6, 2022. REUTERS/Yves Herman/File Photo Purchase Licensing Rights

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By **Katie Paul** and Krystal Hu

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium December 6, 2022. REUTERS/Yves Herman/File Photo Purchase Licensing Rights

ANTHROP\C

Claude API Solutions Research Commitments Learn News Try Claude

Product

Claude 3.5 Haiku on AWS Trainium2 and model distillation in Amazon Bedrock

Dec 3, 2024 • 3 min read

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By **Katie Paul** and Krystal Hu

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium December 6, 2022. REUTERS/Yves Herman/File Photo Purchase Licensing Rights

ANTHROP\C

Claude API Solutions Research Commitments Learn News Try Claude

Product

Claude 3.5 Haiku on AWS Trainium2 and model distillation in Amazon Bedrock

Dec 3, 2024 • 3 min read

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

Elon Musk's xAI is reportedly trying to borrow $12,000,000,000 for even more Nvidia GPUs, an impulse all PC gamers can truly understand

News By Andy Edser published 23 July 2025

I've checked down the back of the sofa, and I'm not sure I can cover it.

Comments (2)

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By **Katie Paul** and Krystal Hu

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgium December 6, 2022. REUTERS/Yves Herman/File Photo Purchase Licensing Rights

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

ANTHROP\C Claude API Solutions Research Commitments Learn News Try Claude

Product

Claude 3.5 Haiku on AWS Tr… model distillation in Amaz…

Dec 3, 2024 • 3 min read

OpenAI's Sam Altman is dreaming of running 100 million GPUs in the future - 100x more than it plans to run by December 2025

News By Efosa Udinmwen published July 26, 2025

OpenAI scale-up will give its investors something to think about

Comments (0)

Elon Musk's xAI is reportedly trying to borrow $12,000,000,000 for even more Nvidia GPUs, an impulse all PC gamers can truly understand

News By Andy Edser published 23 July 2025

I've checked down the back of the sofa, and I'm not sure I can cover it.

Comments (2)

Computing Hardware is No Longer For Everybody

Exclusive: Meta begins testing its first in-house AI training chip

By **Katie Paul** and **Krystal Hu**

March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[1/2] The logo of Meta Platforms' business group is seen in Brussels, Belgi
Herman/File Photo Purchase Licensing Rights

ANTHROP\C Claude API Solutions Research Commitments Learn News Try Claude

Product

OpenAI's Sam Altman is dreaming of running 100 million GPUs in the future - run by December

nething to think about

Ironwood: The first Google TPU for the age of inference

- When scaled to 9,216 chips per pod for a total of 42.5 Exaflops, Ironwood supports more than 24x the compute power of the world's largest supercomputer – El Capitan – which offers just 1.7 Exaflops per pod. Ironwood delivers the massive parallel processing power necessary for the most demanding AI workloads, such as super large size dense LLM or MoE models with thinking capabilities for training and inference. Each individual chip boasts peak compute of 4,614 TFLOPs. This represents a monumental leap in AI capability. Ironwood's memory and network architecture ensures that the right data is always available to support peak performance at this massive scale.

NVIDIA Puts Grace Blackwell on Every Desk and at Every AI Developer's Fingertips

NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

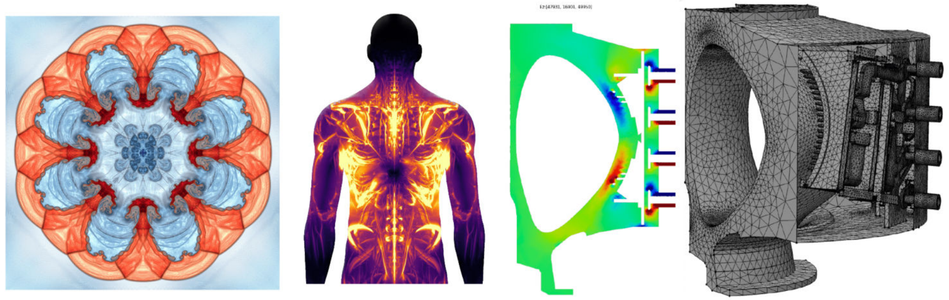
more Nvidia GPUs, an impulse all PC gamers can truly understand

News By Andy Edser published 23 July 2025

I've checked down the back of the sofa, and I'm not sure I can cover it.

Comments (2)

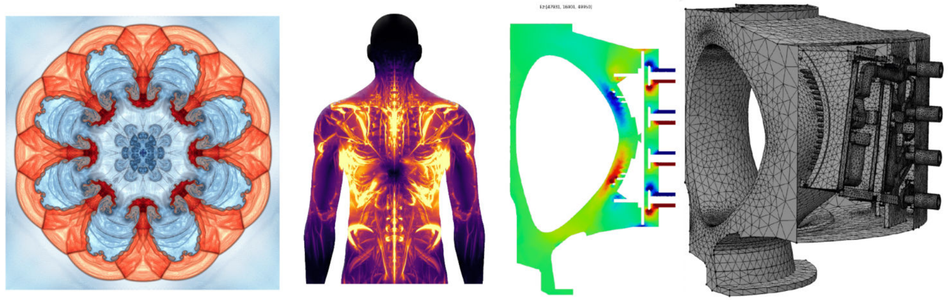
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* nodeElemCornerList,
                                    const Real_t* fx_elem,
                                    const Real_t* fy_elem,
                                    const Real_t* fz_elem,
                                    Real_t* fx_node,
                                    Real_t* fy_node,
                                    Real_t* fz_node,
                                    const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
      Index_t g_i = tid;
      Int_t count=nodeElemCount[g_i];
      Int_t start=nodeElemStart[g_i];
      Real_t fx,fy,fz;
      fx=fy=fz=Real_t(0.0);

      for (int j=0;j<count;j++)
      {
          Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
          fx += fx_elem[pos];
          fy += fy_elem[pos];
          fz += fz_elem[pos];
      }


      fx_node[g_i]=fx;
      fy_node[g_i]=fy;
      fz_node[g_i]=fz;
    }
}
```

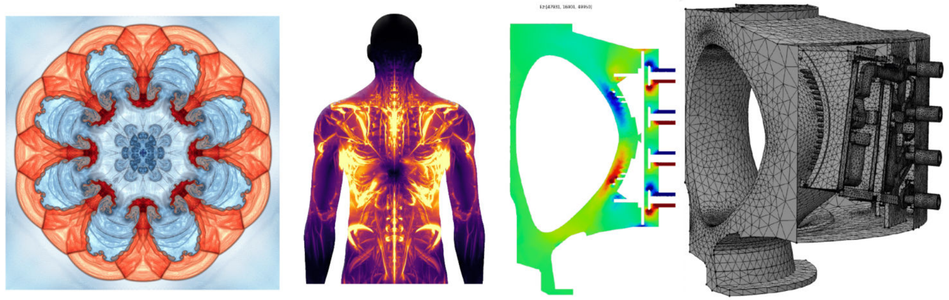
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* nodeElemCornerList,
                                    const Real_t* fx_elem,
                                    const Real_t* fy_elem,
                                    const Real_t* fz_elem,
                                    Real_t* fx_node,
                                    Real_t* fy_node,
                                    Real_t* fz_node,
                                    const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
      Index_t g_i = tid;
      Int_t count=nodeElemCount[g_i];
      Int_t start=nodeElemStart[g_i];
      Real_t fx,fy,fz;
      fx=fy=fz=Real_t(0.0);

      for (int j=0;j<count;j++)
      {
          Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
          fx += fx_elem[pos];
          fy += fy_elem[pos];
          fz += fz_elem[pos];
      }


      fx_node[g_i]=fx;
      fy_node[g_i]=fy;
      fz_node[g_i]=fz;
    }
}
```

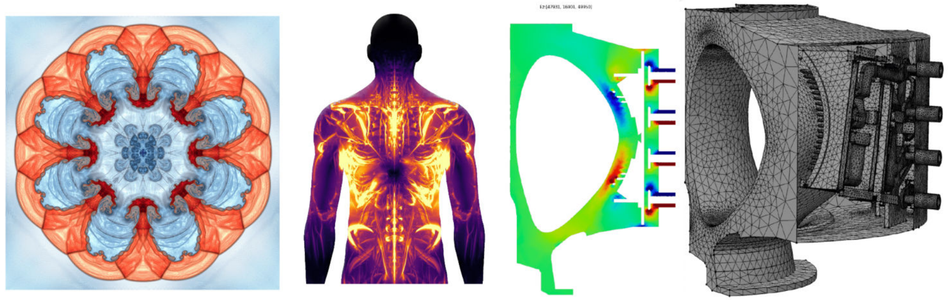
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)
  - Templated

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* nodeElemCornerList,
                                    const Real_t* fx_elem,
                                    const Real_t* fy_elem,
                                    const Real_t* fz_elem,
                                    Real_t* fx_node,
                                    Real_t* fy_node,
                                    Real_t* fz_node,
                                    const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
      Index_t g_i = tid;
      Int_t count=nodeElemCount[g_i];
      Int_t start=nodeElemStart[g_i];
      Real_t fx,fy,fz;
      fx=fy=fz=Real_t(0.0);

      for (int j=0;j<count;j++)
      {
          Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
          fx += fx_elem[pos];
          fy += fy_elem[pos];
          fz += fz_elem[pos];
      }


      fx_node[g_i]=fx;
      fy_node[g_i]=fy;
      fz_node[g_i]=fz;
    }
}
```

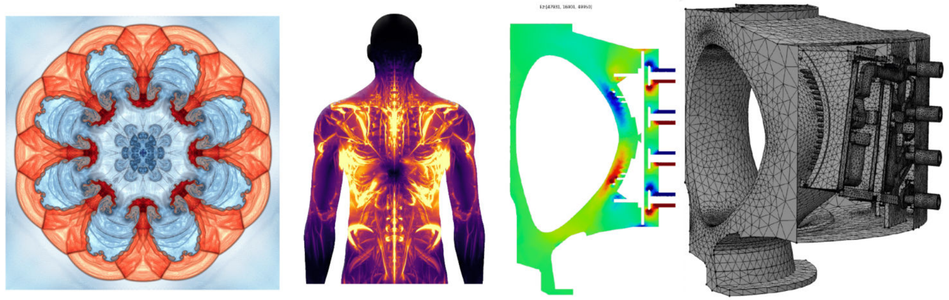
Lingua Franca of Scientific Computing



- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)
  - Templated
  - Not in Python

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* nodeElemCornerList,
                                    const Real_t* fx_elem,
                                    const Real_t* fy_elem,
                                    const Real_t* fz_elem,
                                    Real_t* fx_node,
                                    Real_t* fy_node,
                                    Real_t* fz_node,
                                    const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
      Index_t g_i = tid;
      Int_t count=nodeElemCount[g_i];
      Int_t start=nodeElemStart[g_i];
      Real_t fx,fy,fz;
      fx=fy=fz=Real_t(0.0);

      for (int j=0;j<count;j++)
      {
          Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
          fx += fx_elem[pos];
          fy += fy_elem[pos];
          fz += fz_elem[pos];
      }


      fx_node[g_i]=fx;
      fy_node[g_i]=fy;
      fz_node[g_i]=fz;
    }
}
```

Lingua Franca of Scientific Computing



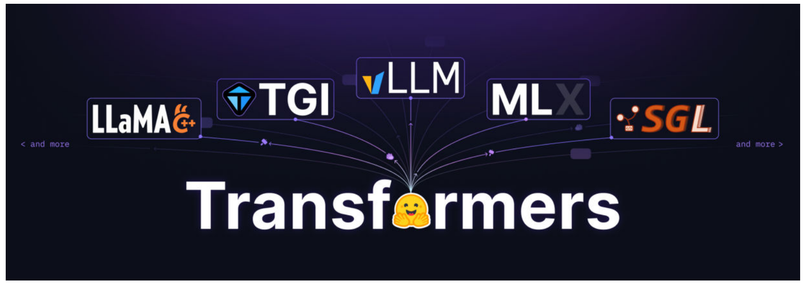
- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)
  - Templated
  - Not in Python
  - Sometimes* in CUDA

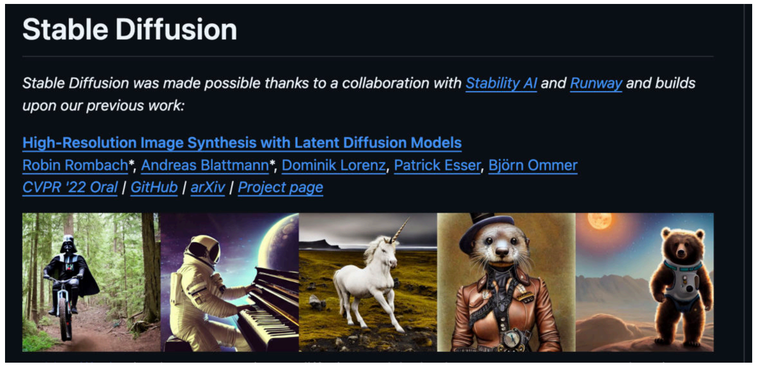
```
template <>
struct RajaCuWrap<3>
{
   template <const int BLCK = MFEM_CUDA_BLOCKS, typename DBODY>
   static void run(const int N, DBODY &&d_body,
                   const int X, const int Y, const int Z, const int G)
   {
      RajaCuWrap3D(N, d_body, X, Y, Z, G);
   }
};
```

```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* nodeElemCornerList,
                                    const Real_t* fx_elem,
                                    const Real_t* fy_elem,
                                    const Real_t* fz_elem,
                                    Real_t* fx_node,
                                    Real_t* fy_node,
                                    Real_t* fz_node,
                                    const Int_t num_threads)
{
    int tid=blockDim.x*blockIdx.x+threadIdx.x;
    if (tid < num_threads)
    {
      Index_t g_i = tid;
      Int_t count=nodeElemCount[g_i];
      Int_t start=nodeElemStart[g_i];
      Real_t fx,fy,fz;
      fx=fy=fz=Real_t(0.0);

      for (int j=0;j<count;j++)
      {
          Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here
          fx += fx_elem[pos];
          fy += fy_elem[pos];
          fz += fz_elem[pos];
      }


      fx_node[g_i]=fx;
      fy_node[g_i]=fy;
      fz_node[g_i]=fz;
    }
}
```

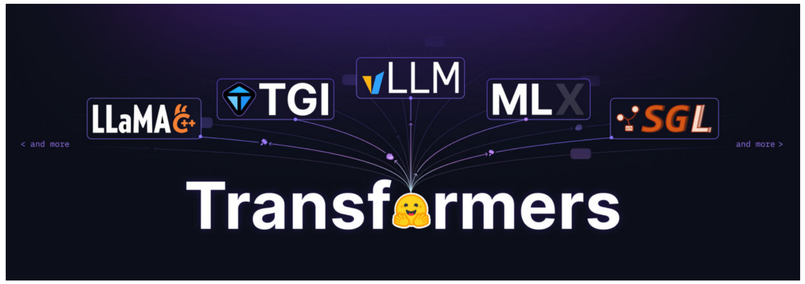
How do we write ML Accelerator code now?

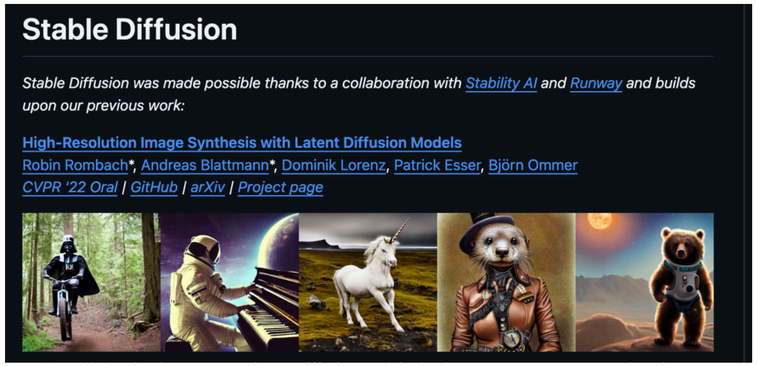
How do we write ML Accelerator code now?

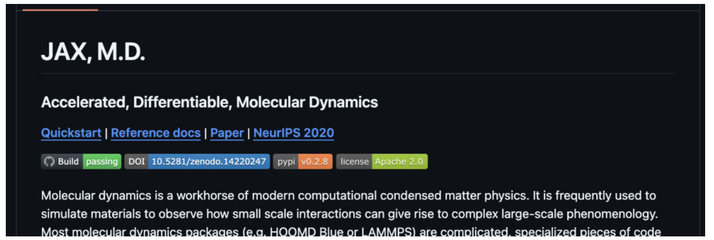


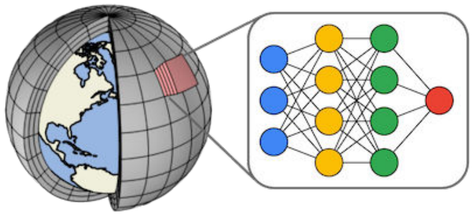


How do we write ML Accelerator code now?

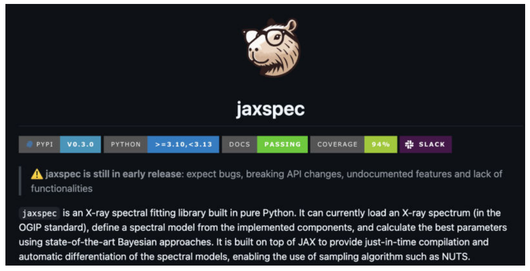




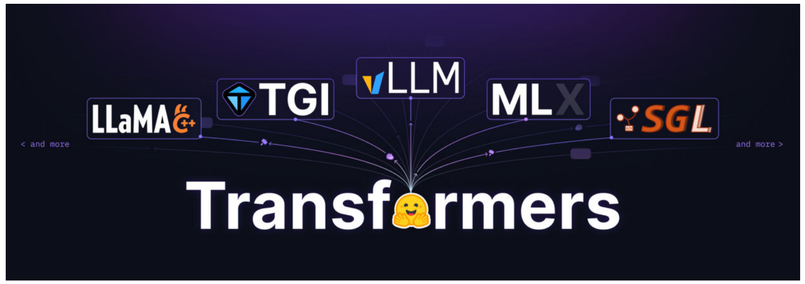


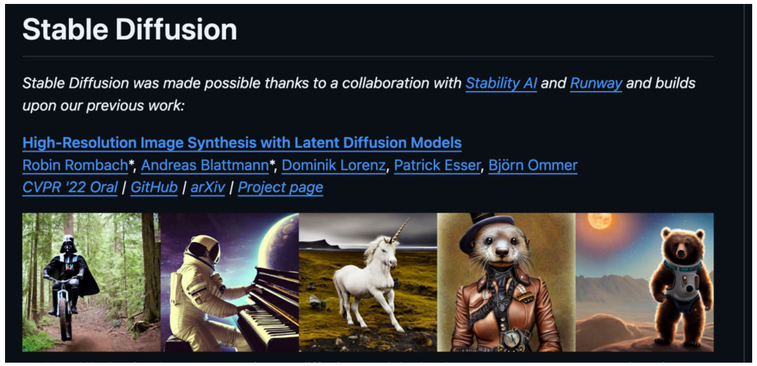


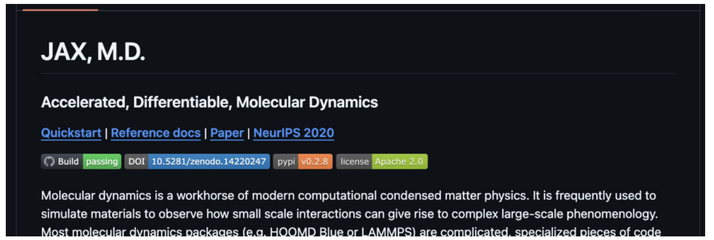
NeuralGCM

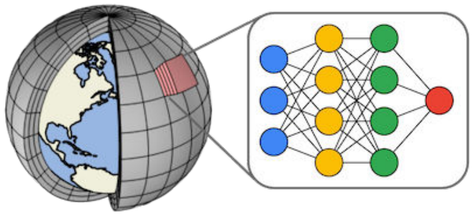


How do we write ML Accelerator code now?

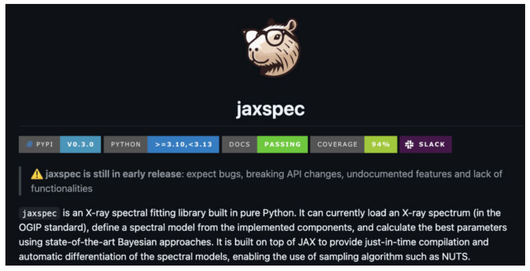








NeuralGCM



Rewrite it in JAX/PyTorch!

The Exascale Computing Project (ECP)

The ECP ran from 2016–2024 and was the largest software research, development, and deployment project managed to date by the US Department of Energy (DOE). The $1.8 billion project was a joint effort by the DOE Office of Science and the National Nuclear Security Administration that funded nearly 2,800 multidisciplinary individuals over the lifetime of the project to uplift the high-performance computing community toward capable exascale platforms, software, and application codes. The outcome was the delivery of an exascale computing ecosystem to provide breakthrough solutions that address future challenges in energy assurance, economic competitiveness, healthcare, and scientific discovery, as well as growing security threats. The ECP exascale ecosystem includes DOE mission-critical application codes, the underlying supporting software technologies, and mechanisms for their deployment and integration.

ECP was a grand convergence of advances in modeling and simulation, software tools and libraries, data analytics, machine learning, and artificial intelligence in support of delivering the world's first capable exascale ecosystem.

The payoff is here: exascale computing is revolutionizing nearly every domain of science.

ECP by the Numbers

Created to develop the nation's first capable exascale computing ecosystem, this unprecedented DOE research, development, and deployment project has already made a huge impact on computational science:

2,800 collaborators funded to develop exascale applications, software, and hardware.

Game-changing results in a broad spectrum of science and engineering application areas.

2 different GPU architectures now proven to work with exascale environments.

First and only open-source scientific software stack developed for scalability and available across all HPC platforms, including cloud computing.

The Exascale Computing Project (ECP)

The ECP ran from 2016–2024 and was the largest software research, development, and deployment project managed to date by the US Department of Energy (DOE). The $1.8 billion project was a joint effort by the DOE Office of Science and the National Nuclear Security Administration that funded nearly 2,800 multidisciplinary individuals over the lifetime of the project to uplift the high-performance computing community toward capable exascale platforms, software, and application codes. The outcome was the delivery of an exascale computing ecosystem to provide breakthrough solutions that address future challenges in energy assurance, economic competitiveness, healthcare, and scientific discovery, as well as growing security threats. The ECP exascale ecosystem includes DOE mission-critical application codes, the underlying supporting software technologies, and mechanisms for their deployment and integration.

ECP was a grand convergence of advances in modeling and simulation, software tools and libraries, data analytics, machine learning, and artificial intelligence in support of delivering the world's first capable exascale ecosystem.

The payoff is here: exascale computing is revolutionizing nearly every domain of science.

ECP by the Numbers

Created to develop the nation's first capable exascale computing ecosystem, this unprecedented DOE research, development, and deployment project has already made a huge impact on computational science:

2,800 collaborators funded to develop exascale applications, software, and hardware.

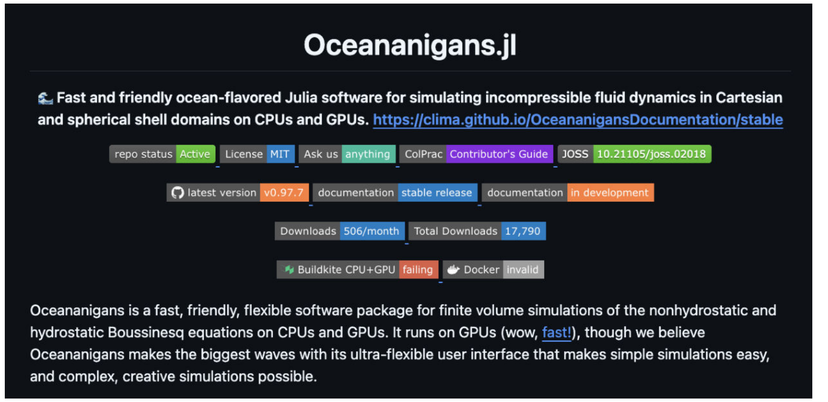
Game-changing results in a broad spectrum of science and engineering application areas.

2 different GPU architectures now proven to work with exascale environments.

First and only open-source scientific software stack developed for scalability and available across all HPC platforms, including cloud computing.

Looking More Deeply at Scientific Code

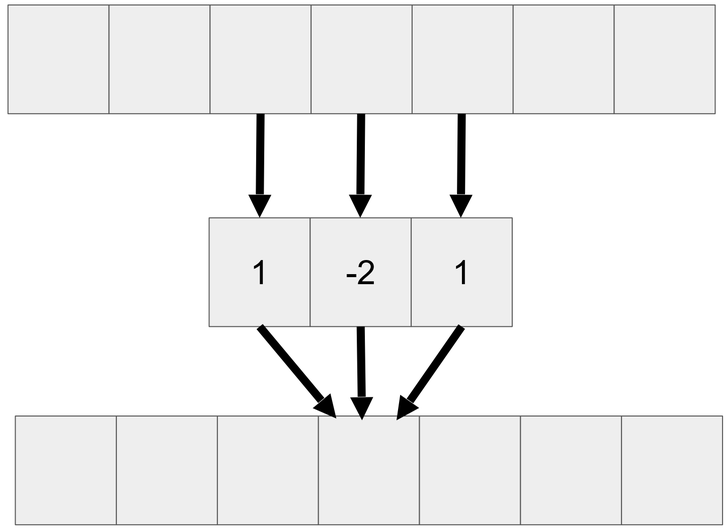
```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i + 1] + x[i + 2]
  end
end

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



> 277 such kernels

Looking More Deeply at 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
```



> 277 such kernels

CUDA to Accelerator IR (StableHLO)

- New framework for raising and optimizing the structure within existing kernels to stablehlo!
- 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 code 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]
  }
}
```

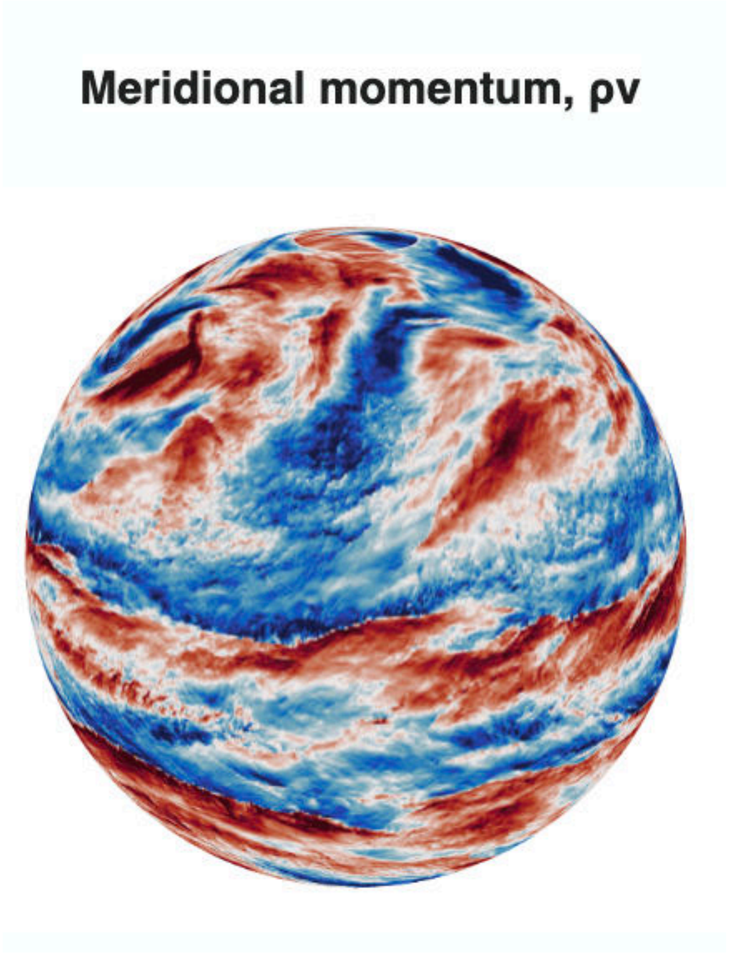
Multi-Dimensionalization

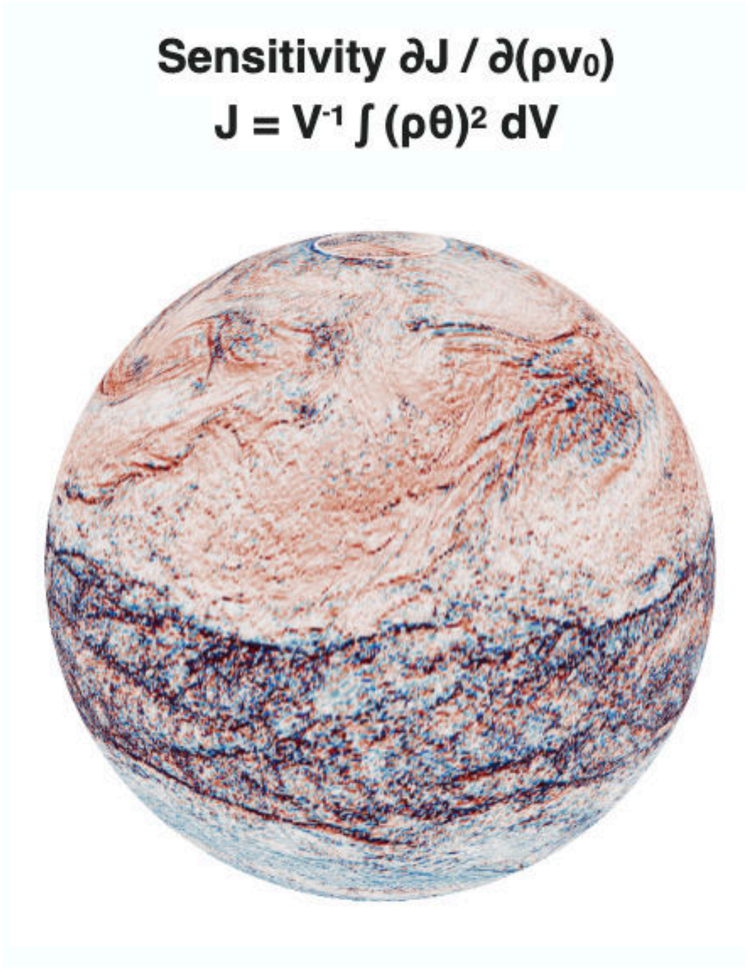
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

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

CUDA to Accelerator IR (StableHLO)






```
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]
  }
}
```

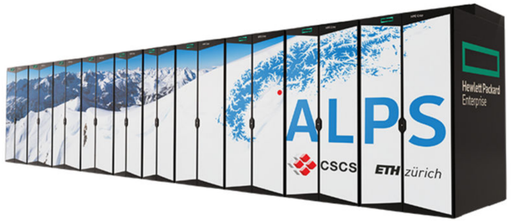
Multi-Dimensionalization

```
%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]>
```

Performance Results

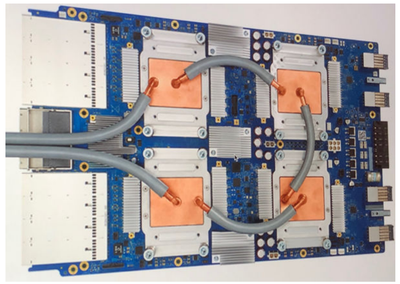


- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
  - Alps (2688 nodes x 4 NVIDIA GH200 GPUs)
  - Perlmutter (1536 nodes x 4 NVIDIA A100 GPUs)
  - 8,192 Google TPUs v7 (4614 F8 TFLOPS each)

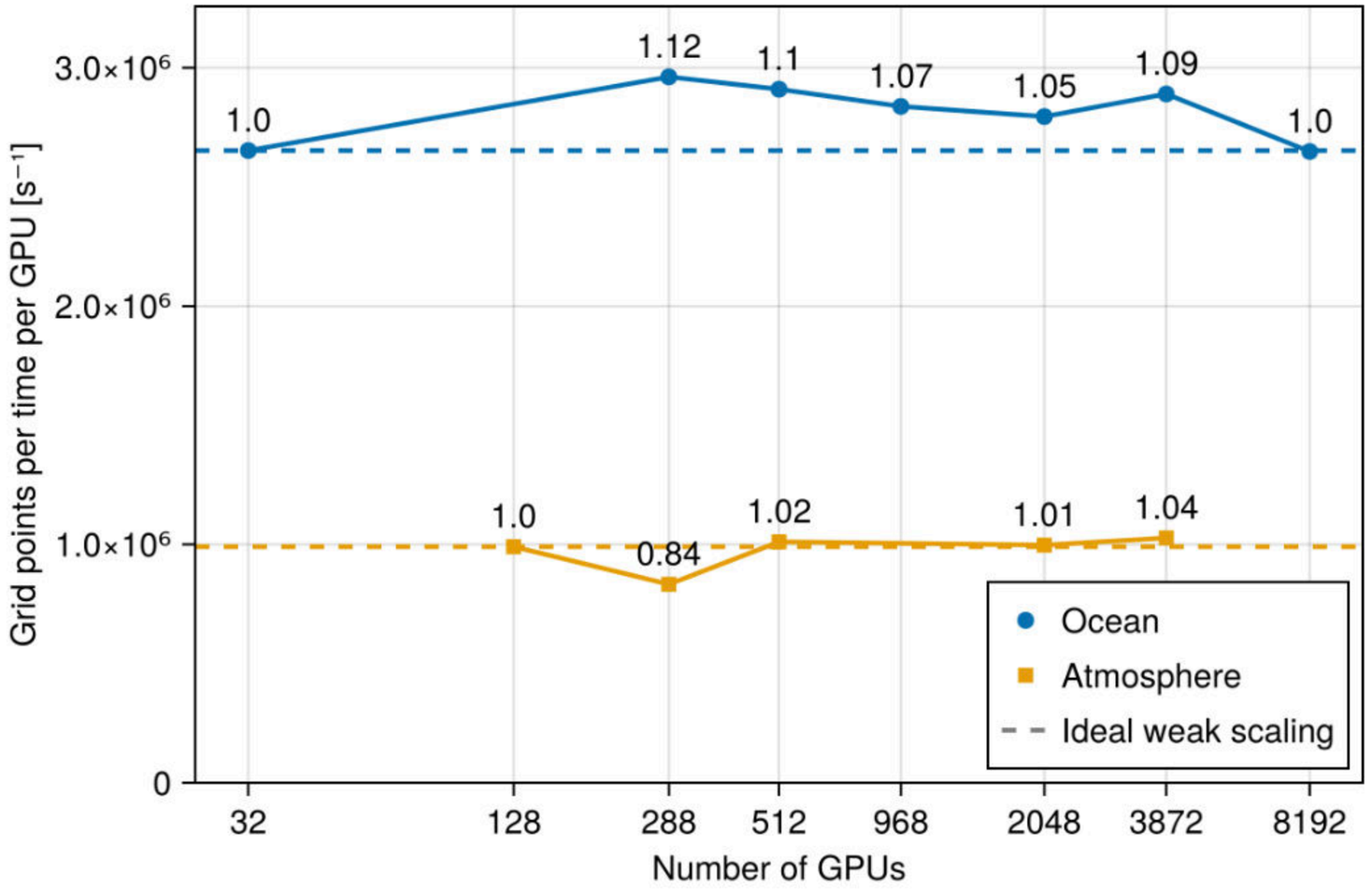
NVIDIA®

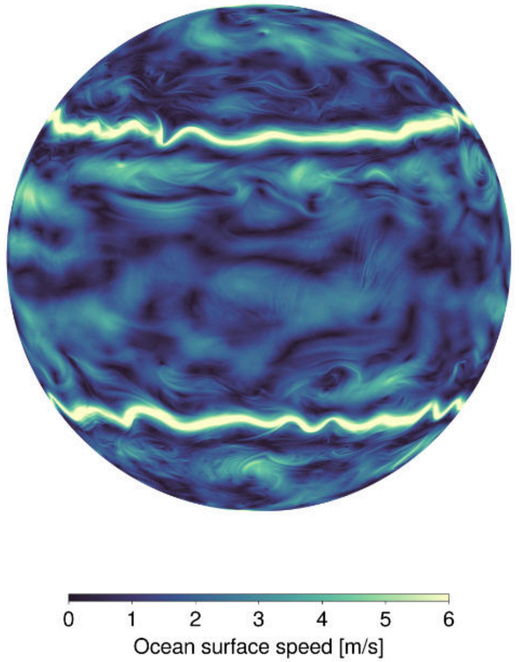
- Good Single-Node Perf (CPU)
  - Vanilla Model: 272.0seconds
  - Tensor Optims: 11.5seconds

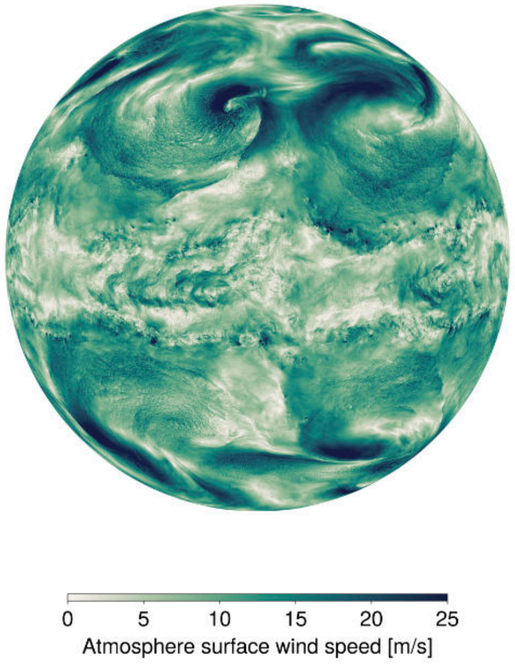
Google Cloud



Performance Results (Alps / NVIDIA GPU)





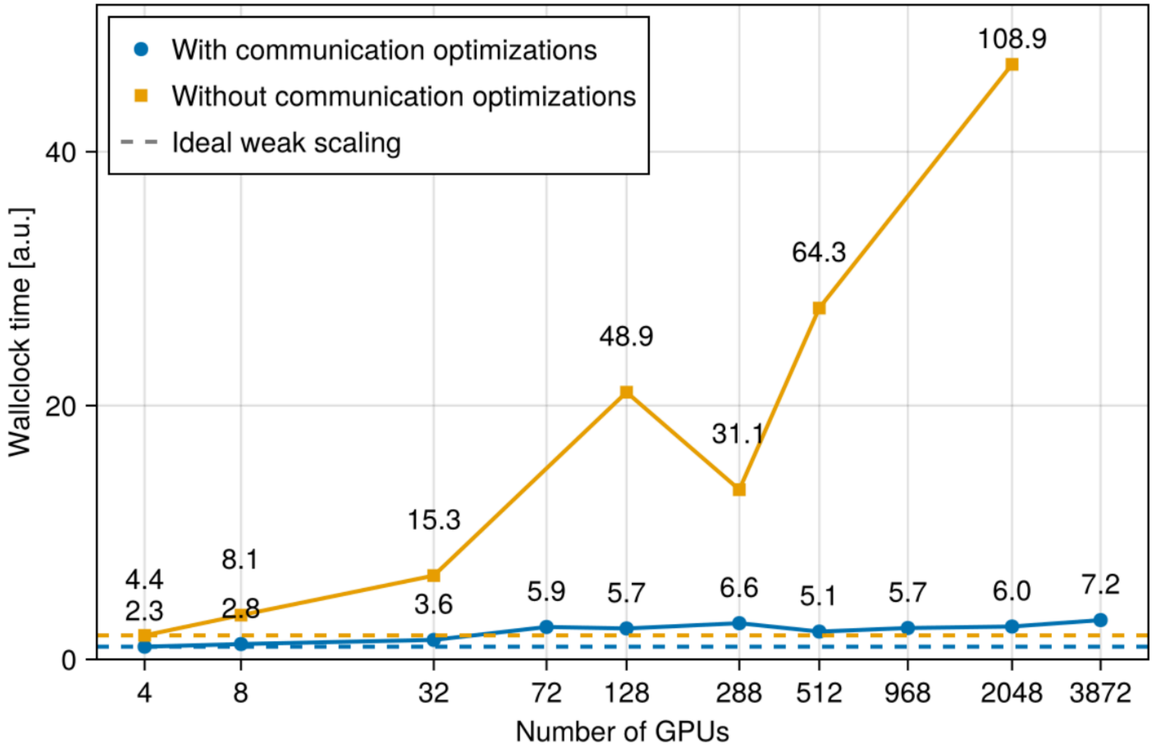


Distributed Optimizations

- Remove redundant communications (akin to automatic discovery of manual halo exchange)

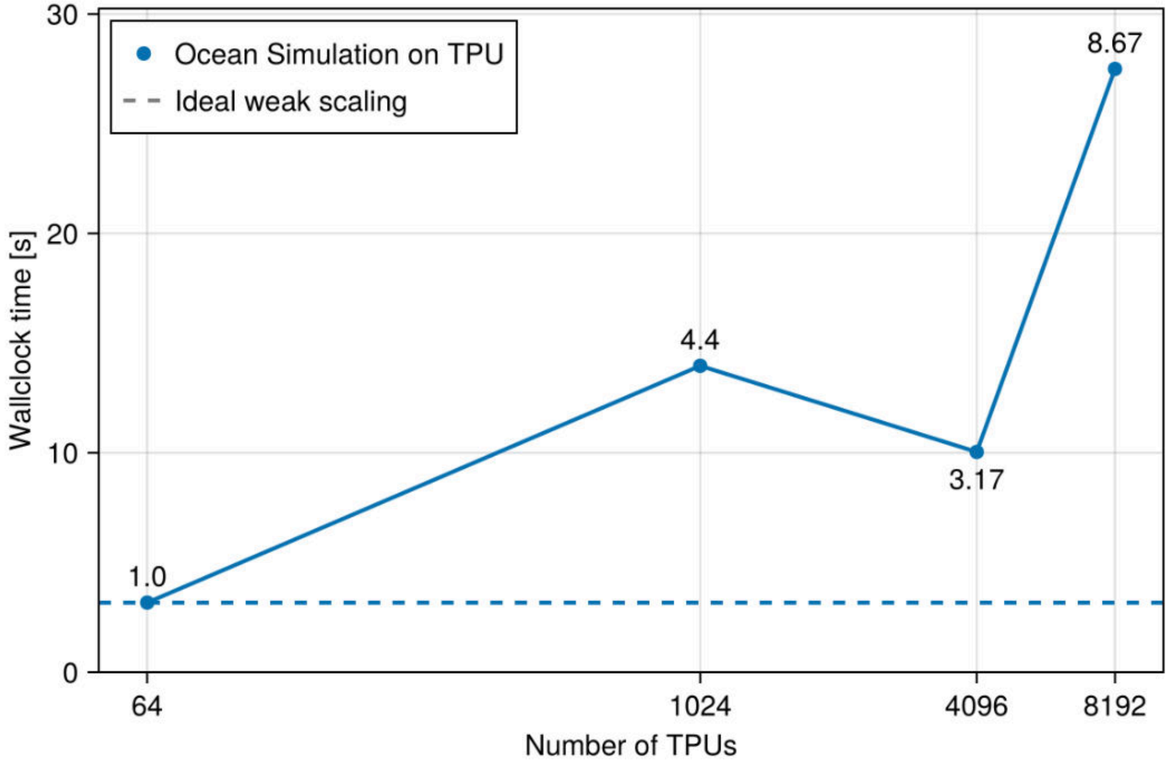
```
# Perform all rotates in a single communication [extending the halo]
%left1, %left2, %left3 = enzymexla.multi_rotate_left %x, 3

%result = %left1 + %left2 + %left3
```

- Fuse resharding operations into consumers to avoid unnecessary sends
- Follow sub tensor provenance & create novel communication partial-CSE to avoid (partially) redundant sends



Performance Results (Google Cloud / TPUs)

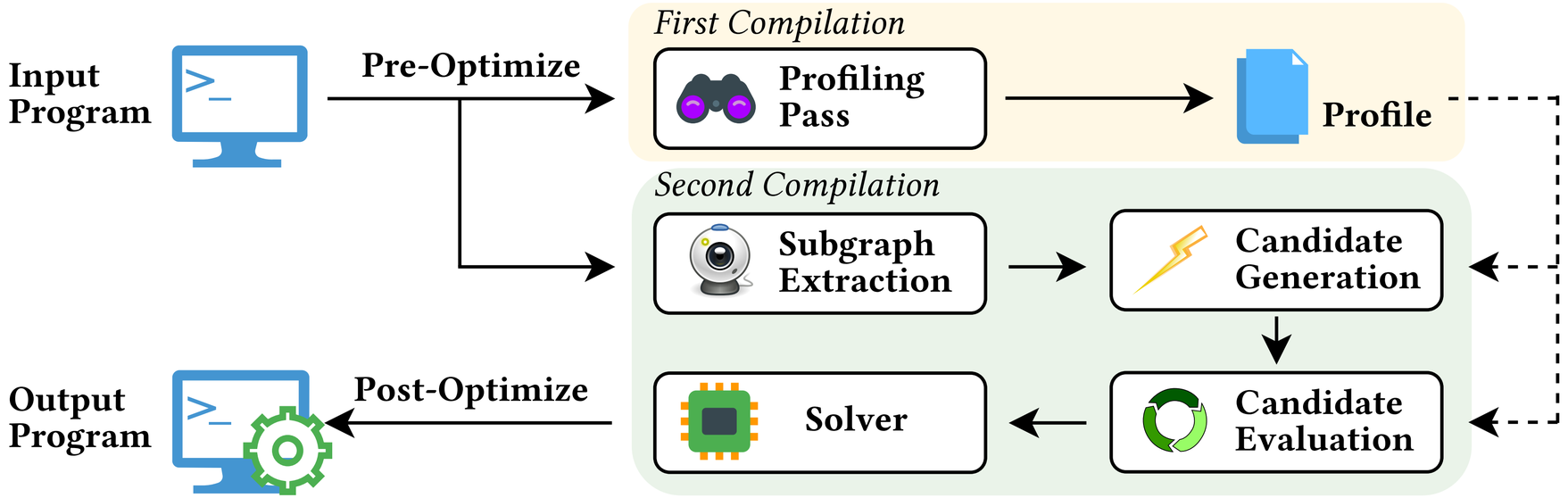
- At peak utilized of 97% of the TPU cluster, containing 36 FP8 exaFLOP!
- 1.493 PiB of HBM to store fields!
- 8x128 registers created some additional comms existing patterns don't yet handle (in progress)



Operation	Percent of Execution
Concatenate	39.04%
Reduce-Window	35.01%
Loop-Fusion 1	19.71%
Data Formatting	2.89%
Slice	1.59%
X64Combine	0.88%
Collective-Permute	0.48%

Jointly Considering Accuracy & Performance

Jointly Considering Accuracy & Performance



$$\Psi = \frac{\kappa}{2} A^4 - \frac{\mu}{2} \left(J_m \ln\left(1 - \frac{\mathrm{tr}(C) - 3}{J_m}\right) + 2\ln(J) \right)$$

$$\text{Profiled Bounds: } \left(1 - \frac{\mathrm{tr}(C) - 3}{J_m}\right) \in [0.972, 1.019] \; (\sim \text{ near } 1)$$

$$\text{Discovered Rewrite: } \ln\left(1 - \frac{\mathrm{tr}(C) - 3}{J_m}\right) \to \mathrm{log1p}\left(-\frac{\mathrm{tr}(C) - 3}{J_m}\right)$$

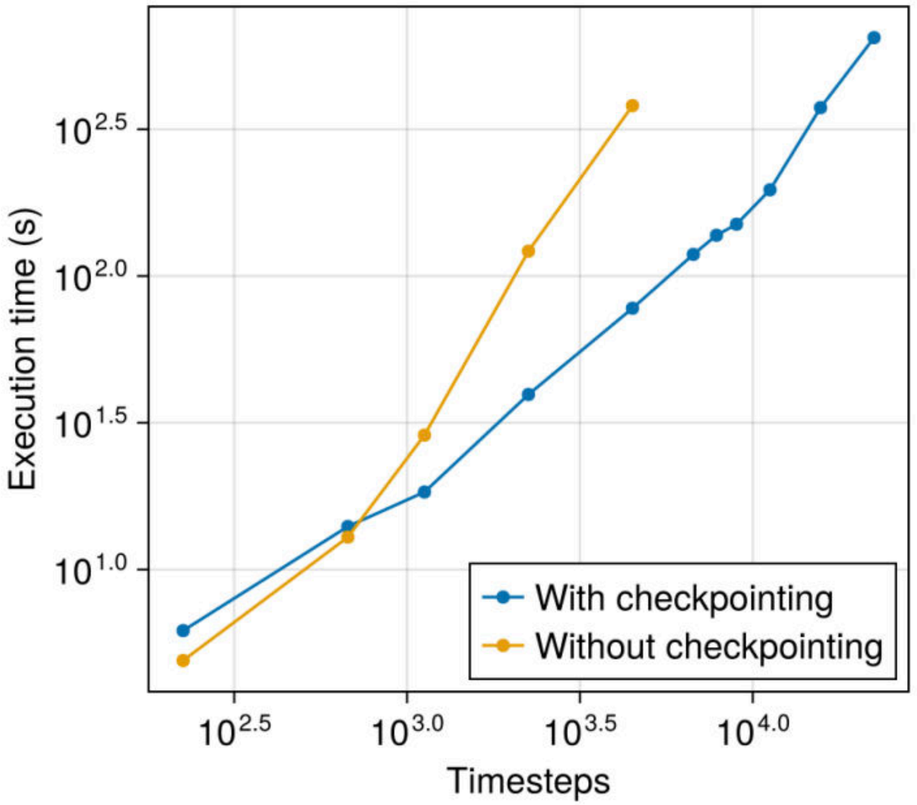
+2.15 bits of accuracy;
no slowdown

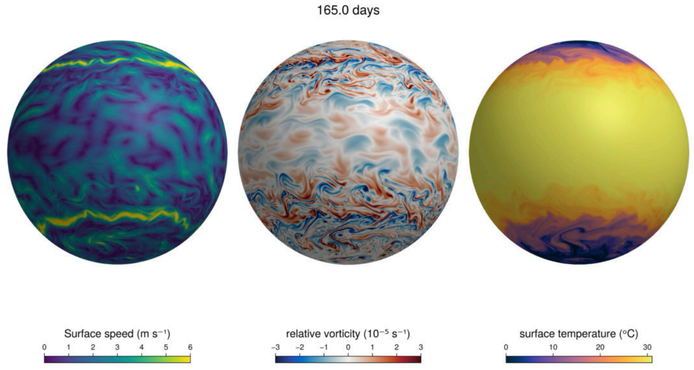
Takeaways

- Enzyme is a tool for performing reverse-mode AD of statically analyzable LLVM & MLIR
- Differentiates code in a variety of languages (C, C++, Fortran, Julia, Rust, Swift, etc)
- 4.2x speedup over AD before optimization on CPU, orders of magnitude improvements on GPU, and distributed
  - Interaction with Optimization is key!
- EnzymeMLIR preserves & optimizing custom semantics; impact compounds on accelerators
- Reactant takes existing code, removes type instabilities, optimizes the mathematics within, and runs it on distributed cluster of your favorite hardware (CPU, GPU, TPU)
- All open source (GitHub.com/EnzymeAD/{Enzyme,Enzyme.jl,Enzyme-JaX, Reactant, …} ; enzyme.mit.edu); weekly developer meetings!

Derivative Raising Performance Results

- Primal Perf (CPU)
  - Vanilla Model: 272.0seconds
  - Tensor Optims: 11.5seconds
- Derivative Performance
  - Similar performance to primal on single timestep, scaling with linearly time steps
  - Disabling tensor optimizations causes it to instantly oom the system
  - Tensor and whole-program optimizations are quite useful!





Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)

v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))
A = Reactant.to_rarray(rand(Float32, 1024, 1024))
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

- Without any optimization, we perform a scatter to create the diagonal, then a matmul

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) -> tensor<f32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>
    %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) ->
tensor<1024x1024xf32>
    %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>
    %2 = "stablehlo.scatter"(%cst_0, %1, %arg0) <{scatter_dimension_numbers =
#stablehlo.scatter<inserted_window_dims = [0, 1], scatter_dims_to_operand_dims = [0, 1],
index_vector_dim = 1>}> ({
    ^bb0(%arg3: tensor<f32>, %arg4: tensor<f32>):
      stablehlo.return %arg4 : tensor<f32>
    }) : (tensor<1024x1024xf32>, tensor<1024x2xi64>, tensor<1024xf32>) -> tensor<1024x1024xf32>
    %3 = stablehlo.dot_general %2, %arg1, contracting_dims = [1] x [1], precision = [DEFAULT,
DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) -> tensor<1024x1024xf32>
    %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>
    %5 = stablehlo.multiply %4, %4 : tensor<1024x1024xf32>
    %6 = stablehlo.reduce(%5 init: %cst) applies stablehlo.add across dimensions = [0, 1] :
(tensor<1024x1024xf32>, tensor<f32>) -> tensor<f32>
    return %6 : tensor<f32>
}
```

Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix

- Without any optimization, we perform a scatter to create the diagonal, then a matmul

- Differentiating this, results in gathers in the derivative, which cannot be removed via optimization.

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)

v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))
A = Reactant.to_rarray(rand(Float32, 1024, 1024))
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) → (tensor<1024xf32>, tensor<1024x1024xf32>, tensor<1024x1024xf32>)
{
    %cst = stablehlo.constant dense←2.000000e+00> : tensor<1024x1024xf32>
    %cst_0 = stablehlo.constant dense<2.000000e+00> : tensor<1024x1024xf32>
    %cst_1 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>
    %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) →
tensor<1024x1024xf32>
    %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>
    %2 = stablehlo.broadcast_in_dim %arg0, dims = [0] : (tensor<1024xf32>) →
tensor<1024x1024xf32>
    %3 = stablehlo.multiply %2, %arg1 : tensor<1024x1024xf32>
    %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>
    %5 = stablehlo.multiply %4, %cst_0 : tensor<1024x1024xf32>
    %6 = stablehlo.compare  GE, %4, %cst_1 : (tensor<1024x1024xf32>, tensor<1024x1024xf32>)
→ tensor<1024x1024xi1>
    %7 = stablehlo.multiply %4, %cst : tensor<1024x1024xf32>
    %8 = stablehlo.select %6, %5, %7 : tensor<1024x1024xi1>, tensor<1024x1024xf32>
    %9 = stablehlo.transpose %8, dims = [1, 0] : (tensor<1024x1024xf32>) →
tensor<1024x1024xf32>
    %10 = stablehlo.dot_general %8, %arg1, contracting_dims = [1] x [0], precision =
[DEFAULT, DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) →
tensor<1024x1024xf32>
    %11 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<1024xf32>) →
tensor<1024x1024xf32>
    %12 = stablehlo.multiply %8, %11 : tensor<1024x1024xf32>
    %13 = "stablehlo.gather"(%10, %1) <{dimension_numbers =
#stablehlo.gather<collapsed_slice_dims = [0, 1], start_index_map = [0, 1], index_vector_dim
= 1>, slice_sizes = array<i64: 1, 1>}> : (tensor<1024x1024xf32>, tensor<1024x2xi64>) →
tensor<1024xf32>
    return %13, %12, %9 : tensor<1024xf32>, tensor<1024x1024xf32>, tensor<1024x1024xf32>
}
```

Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix

- mul(diag(x), v) ->
  elementwise(x, v)

- Performing this prior to AD yields 2-3x performance!

All Optimizations Enabled

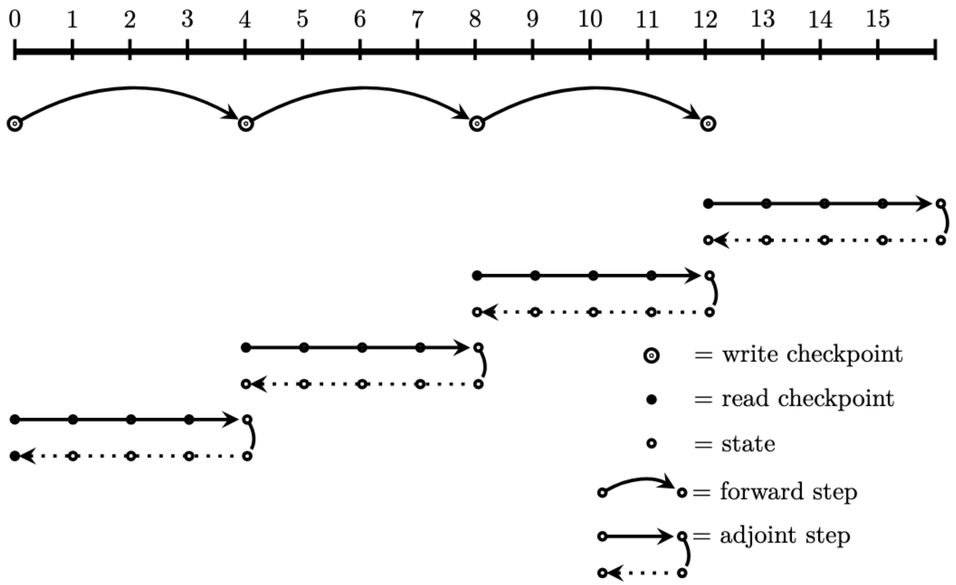
```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2: tensor<1024x1024xf32>)
→ tensor<f32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %0 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<1024xf32>) → tensor<1024x1024xf32>
    %1 = stablehlo.multiply %0, %arg1 : tensor<1024x1024xf32>
    %2 = stablehlo.add %1, %arg2 : tensor<1024x1024xf32>
    %3 = stablehlo.multiply %2, %2 : tensor<1024x1024xf32>
    %4 = stablehlo.reduce(%3 init: %cst) applies stablehlo.add across dimensions = [0, 1] :
(tensor<1024x1024xf32>, tensor<f32>) → tensor<f32>
    return %4 : tensor<f32>
}
```

```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) → tensor<f32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>
    %0 = stablehlo.transpose %arg2, dims = [1, 0] : (tensor<1024x1024xf32>) →
tensor<1024x1024xf32>
    %1 = stablehlo.iota dim = 0 : tensor<1024x2xi64>
    %2 = "stablehlo.scatter"(%cst_0, %1, %arg0) <{scatter_dimension_numbers =
#stablehlo.scatter<inserted_window_dims = [0, 1], scatter_dims_to_operand_dims = [0, 1],
index_vector_dim = 1>}> ({
    ^bb0(%arg3: tensor<f32>, %arg4: tensor<f32>):
      stablehlo.return %arg4 : tensor<f32>
    }) : (tensor<1024x1024xf32>, tensor<1024x2xi64>, tensor<1024xf32>) → tensor<1024x1024xf32>
    %3 = stablehlo.dot_general %2, %arg1, contracting_dims = [1] x [1], precision = [DEFAULT,
DEFAULT] : (tensor<1024x1024xf32>, tensor<1024x1024xf32>) → tensor<1024x1024xf32>
    %4 = stablehlo.add %3, %0 : tensor<1024x1024xf32>
    %5 = stablehlo.multiply %4, %4 : tensor<1024x1024xf32>
    %6 = stablehlo.reduce(%5 init: %cst) applies stablehlo.add across dimensions = [0, 1] :
(tensor<1024x1024xf32>, tensor<f32>) → tensor<f32>
    return %6 : tensor<f32>
}
```

Scatter Optimizations Disabled

Checkpointing

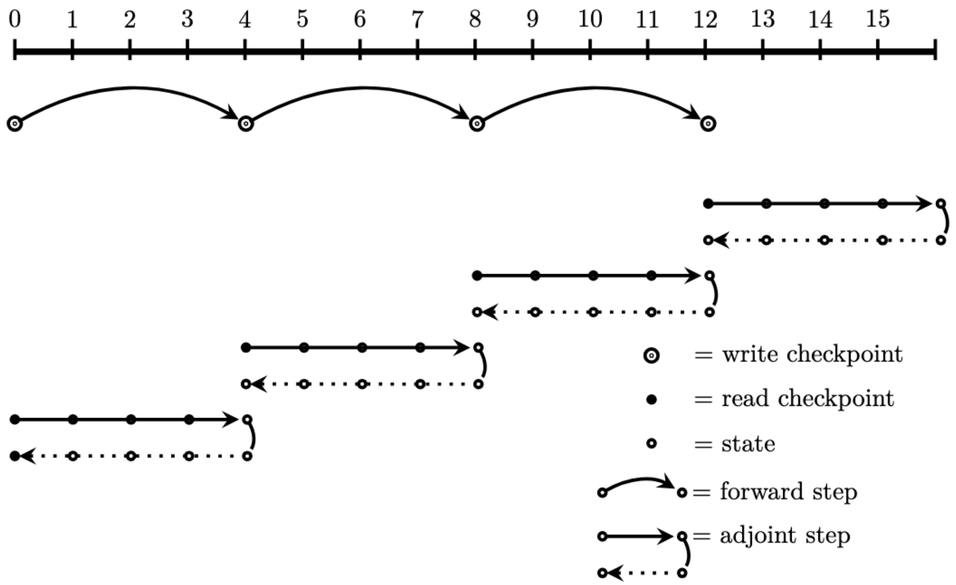
- Checkpointing is a technique for trading off memory and compute time in the derivative




```
cache = malloc N x f32
for i = 0:N {
   x = foo(x)
   cache[i] = x
}

for i = N:0 {
   x = cache[I]
   dx = grad_foo(x, dx)
}
```

Checkpointing

- Checkpointing is a technique for trading off memory and compute time in the derivative




```
cache = malloc M x f32
for i = 0:N/M {
   for j = 0:M {
     x = foo(x)
   }
   cache[i] = x
}

for i = N:0 {
   x = cache[I/M]
   for j in 0:i%M {
      x = foo(x)
   }
   dx = grad_foo(x, dx)
}
```

Checkpointing

- Checkpointing is a technique for trading off memory and compute time in the derivative

- Performing entire-program-level analysis, we can remove induction variables on the loop, reducing memory AND computation

```
x = tensor<100x100xf32>
for i = 0:steps {
   x[0,  :] = 0
   x[end,:] = 0
   y = foo(x, y)
}
```

```
if (steps > 0) {
  x[0,  :] = 0
  x[end,:] = 0

  for i = 0:steps {
    y = foo(x, y)
  }
```

Linear Algebra + AD

- Consider a simple code which performs a matmul and add on a Diagonal matrix

```
diagmm(v, A, x) = sum(abs2, v * A .+ x)

v = Reactant.to_rarray(Diagonal(rand(Float32, 1024)))
A = Reactant.to_rarray(rand(Float32, 1024, 1024))
x = Reactant.to_rarray(rand(Float32, 1024, 1024))
```

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 & synchronization is a barrier to 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);
}
```

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
}
```

GPU Programming via MLIR

- Preserve Host & Device code through frontend
  (Clang Plugin for C++, JIT Package for Julia, etc)
- 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

GPU Programming via MLIR

- Preserve Host & Device code through frontend

  (Clang Plugin for C++, JIT Package for Julia, etc)

- 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

GPU Programming via MLIR

```
func @launch(%h_out : memref<?xf32>, %h_in : memref<?xf32>, %n : i64) {

  parallel.for (%gx, %gy, %gz) = (0, 0, 0) to (grid.x, grid.y, grid.z) {

    %shared_val = memref.alloca : memref<f32>

    parallel.for (%tx, %ty, %tz) = (0, 0, 0) to (blk.x, blk.y, blk.z) {

      if %tx == 0 {
         store …, %shared_val[] : memref<f32>
      }

      polygeist.barrier(%tx, %ty, %tz)

      …
    }
  }
}
```

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

Synchronization via Memory

- Synchronization (sync_threads) ensures all threads within a block finish executing codeA before executing codeB
- The desired synchronization behavior can be reproduced by defining sync_threads to have the union of the memory semantics of the code before and after the sync.
- This prevents code motion of instructions which require the synchronization for correctness, but permits other code motion (e.g. index computation).

```
codeA(fib(idx));
sync_threads;
codeB(fib(idx));
```

```
off = fib(idx);
codeA(off);
sync_threads;
codeB(off);
```

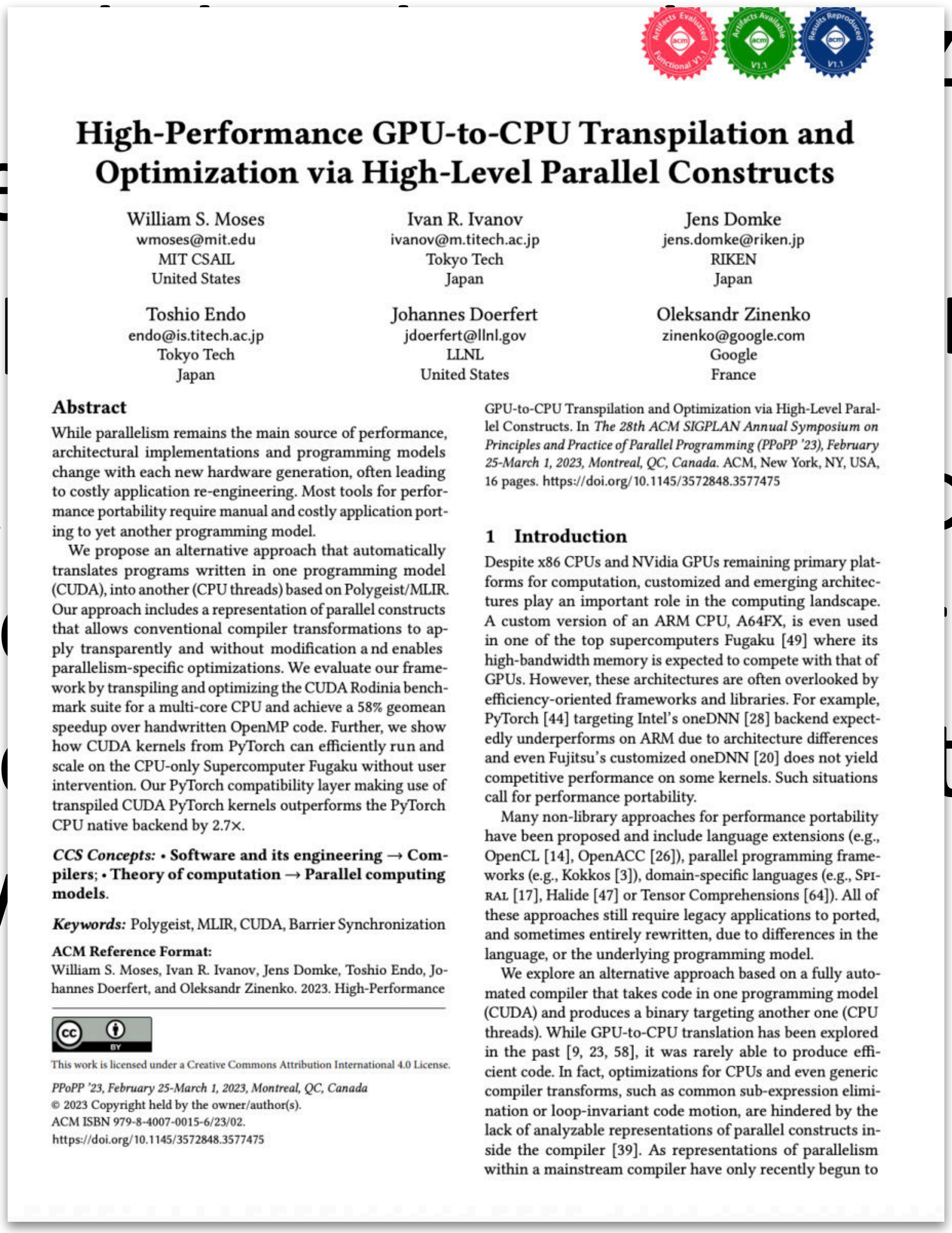
Synchronization via Memory

- High-level synchronization representation enables new optimizations, like sync elimination.
- A synchronize instruction is not needed if the set of read/writes before the sync don't conflict with the read/writes after the sync.

```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];

  if ( tx == 0 )
    node[ty] = input[index_in] ;

  // Unnecessary Barrier #1
  // None of the read/writes below the sync
  //  (weights, hidden)
  // intersect with the read/writes above the sync
  //  (node, input)
  __syncthreads();


  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];

  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  …
}
```

Synchronization via Memory

- H zation re es ne o nc eli
- A ction n ead/ b t cont w after



High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs

William S. Moses
wmoses@mit.edu
MIT CSAIL
United States

Ivan R. Ivanov
ivanov@m.titech.ac.jp
Tokyo Tech
Japan

Jens Domke
jens.domke@riken.jp
RIKEN
Japan

Toshio Endo
endo@is.titech.ac.jp
Tokyo Tech
Japan

Johannes Doerfert
jdoerfert@llnl.gov
LLNL
United States

Oleksandr Zinenko
zinenko@google.com
Google
France

Abstract

While parallelism remains the main source of performance, architectural implementations and programming models change with each new hardware generation, often leading to costly application re-engineering. Most tools for performance portability require manual and costly application porting to yet another programming model.

We propose an alternative approach that automatically translates programs written in one programming model (CUDA), into another (CPU threads) based on Polygeist/MLIR. Our approach includes a representation of parallel constructs that allows conventional compiler transformations to apply transparently and without modification a nd enables parallelism-specific optimizations. We evaluate our framework by transpiling and optimizing the CUDA Rodinia benchmark suite for a multi-core CPU and achieve a 58% geomean speedup over handwritten OpenMP code. Further, we show how CUDA kernels from PyTorch can efficiently run and scale on the CPU-only Supercomputer Fugaku without user intervention. Our PyTorch compatibility layer making use of transpiled CUDA PyTorch kernels outperforms the PyTorch CPU native backend by 2.7×.

CCS Concepts: • **Software and its engineering → Compilers**; • **Theory of computation → Parallel computing models**.

Keywords: Polygeist, MLIR, CUDA, Barrier Synchronization

ACM Reference Format:
William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *The 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23), February 25-March 1, 2023, Montreal, QC, Canada*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3572848.3577475

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

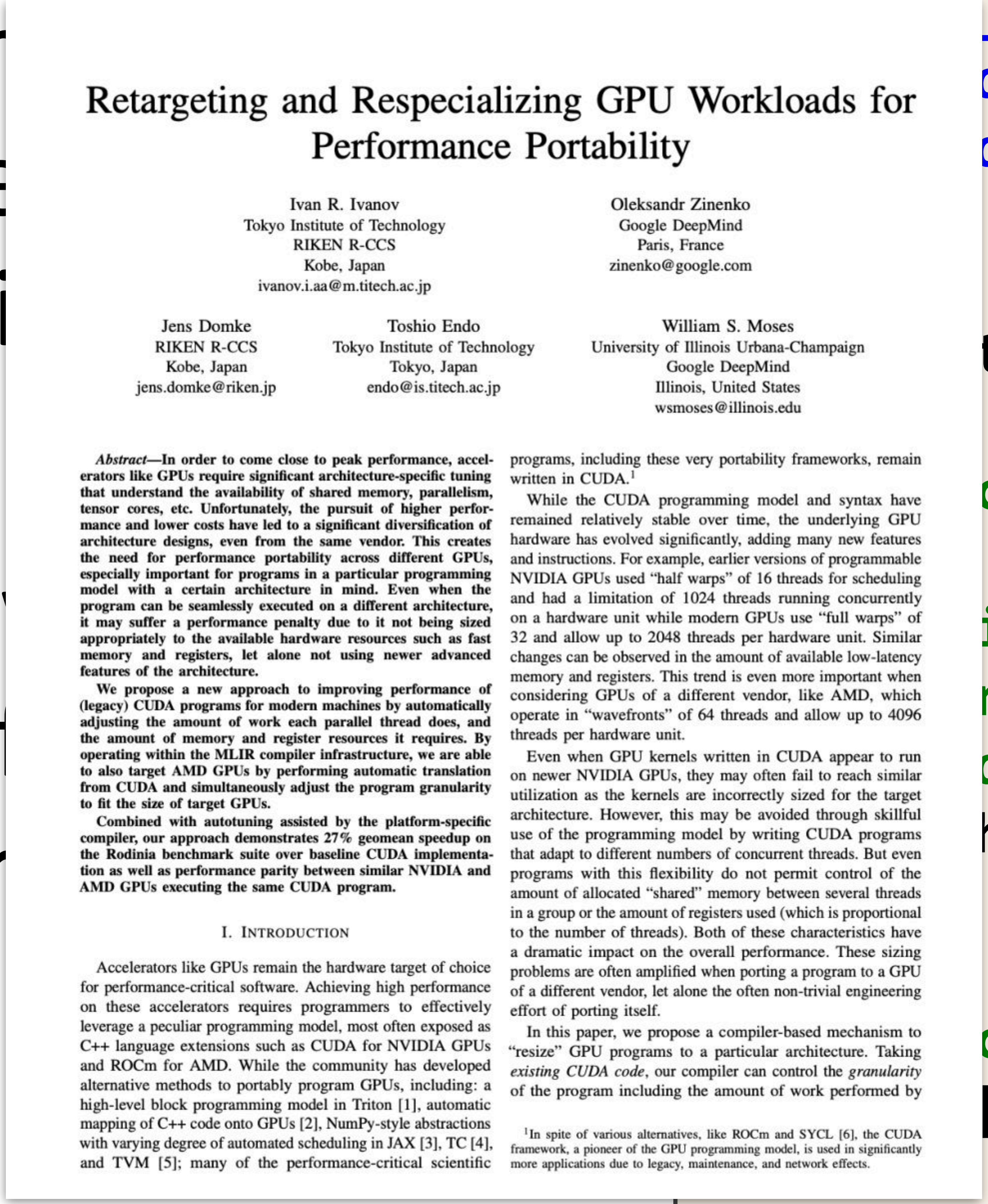
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577475

1 Introduction

Despite x86 CPUs and NVidia GPUs remaining primary platforms for computation, customized and emerging architectures play an important role in the computing landscape. A custom version of an ARM CPU, A64FX, is even used in one of the top supercomputers Fugaku [49] where its high-bandwidth memory is expected to compete with that of GPUs. However, these architectures are often overlooked by efficiency-oriented frameworks and libraries. For example, PyTorch [44] targeting Intel's oneDNN [28] backend expectedly underperforms on ARM due to architecture differences and even Fujitsu's customized oneDNN [20] does not yield competitive performance on some kernels. Such situations call for performance portability.

Many non-library approaches for performance portability have been proposed and include language extensions (e.g., OpenCL [14], OpenACC [26]), parallel programming frameworks (e.g., Kokkos [3]), domain-specific languages (e.g., Spiral [17], Halide [47] or Tensor Comprehensions [64]). All of these approaches still require legacy applications to ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU translation has been explored in the past [9, 23, 58], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transforms, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [39]. As representations of parallelism within a mainstream compiler have only recently begun to



Retargeting and Respecializing GPU Workloads for Performance Portability

Ivan R. Ivanov
Tokyo Institute of Technology
RIKEN R-CCS
Kobe, Japan
ivanov.i.aa@m.titech.ac.jp

Oleksandr Zinenko
Google DeepMind
Paris, France
zinenko@google.com

Jens Domke
RIKEN R-CCS
Kobe, Japan
jens.domke@riken.jp

Toshio Endo
Tokyo Institute of Technology
Tokyo, Japan
endo@is.titech.ac.jp

William S. Moses
University of Illinois Urbana-Champaign
Google DeepMind
Illinois, United States
wsmoses@illinois.edu

Abstract—**In order to come close to peak performance, accelerators like GPUs require significant architecture-specific tuning that understand the availability of shared memory, parallelism, tensor cores, etc. Unfortunately, the pursuit of higher performance and lower costs have led to a significant diversification of architecture designs, even from the same vendor. This creates the need for performance portability across different GPUs, especially important for programs in a particular programming model with a certain architecture in mind. Even when the program can be seamlessly executed on a different architecture, it may suffer a performance penalty due to it not being sized appropriately to the available hardware resources such as fast memory and registers, let alone not using newer advanced features of the architecture.**

We propose a new approach to improving performance of (legacy) CUDA programs for modern machines by automatically adjusting the amount of work each parallel thread does, and the amount of memory and register resources it requires. By operating within the MLIR compiler infrastructure, we are able to also target AMD GPUs by performing automatic translation from CUDA and simultaneously adjust the program granularity to fit the size of target GPUs.

Combined with autotuning assisted by the platform-specific compiler, our approach demonstrates 27% geomean speedup on the Rodinia benchmark suite over baseline CUDA implementation as well as performance parity between similar NVIDIA and AMD GPUs executing the same CUDA program.

I. Introduction

Accelerators like GPUs remain the hardware target of choice for performance-critical software. Achieving high performance on these accelerators requires programmers to effectively leverage a peculiar programming model, most often exposed as C++ language extensions such as CUDA for NVIDIA GPUs and ROCm for AMD. While the community has developed alternative methods to portably program GPUs, including: a high-level block programming model in Triton [1], automatic mapping of C++ code onto GPUs [2], NumPy-style abstractions with varying degree of automated scheduling in JAX [3], TC [4], and TVM [5]; many of the performance-critical scientific programs, including these very portability frameworks, remain written in CUDA.[1]

While the CUDA programming model and syntax have remained relatively stable over time, the underlying GPU hardware has evolved significantly, adding many new features and instructions. For example, earlier versions of programmable NVIDIA GPUs used "half warps" of 16 threads for scheduling and had a limitation of 1024 threads running concurrently on a hardware unit while modern GPUs use "full warps" of 32 and allow up to 2048 threads per hardware unit. Similar changes can be observed in the amount of available low-latency memory and registers. This trend is even more important when considering GPUs of a different vendor, like AMD, which operate in "wavefronts" of 64 threads and allow up to 4096 threads per hardware unit.

Even when GPU kernels written in CUDA appear to run on newer NVIDIA GPUs, they may often fail to reach similar utilization as the kernels are incorrectly sized for the target architecture. However, this may be avoided through skillful use of the programming model by writing CUDA programs that adapt to different numbers of concurrent threads. But even programs with this flexibility do not permit control of the amount of allocated "shared" memory between several threads in a group or the amount of registers used (which is proportional to the number of threads). Both of these characteristics have a dramatic impact on the overall performance. These sizing problems are often amplified when porting a program to a GPU of a different vendor, let alone the often non-trivial engineering effort of porting itself.

In this paper, we propose a compiler-based mechanism to "resize" GPU programs to a particular architecture. Taking *existing CUDA code*, our compiler can control the *granularity* of the program including the amount of work performed by

[1] In spite of various alternatives, like ROCm and SYCL [6], the CUDA framework, a pioneer of the GPU programming model, is used in significantly more applications due to legacy, maintenance, and network effects.

```
_ void bpnn_layerforward(...) {
d__ float node[HEIGHT];
d__ float weights[HEIGHT][WIDTH];

 == 0 )
ty] = input[index_in] ;

cessary Barrier #1
 of the read/writes below the sync
ights, hidden)
rsect with the read/writes above the sync
de, input)
hreads();


cessary Store #1
[ty][tx] = hidden[index];

  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  …
}
```

- 27% speedup on real code, 2.7x on PyTorch cross compilation!

Synchronization via Memory

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. historically OpenMP, now TPUs)
- Some backends do not have block synchronization
- Lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  codeB(%i);
}
```

```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  codeB(%i);
}
```

Synchronization via Memory

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. historically OpenMP, now TPUs)
- Some backends do not have block synchronization
- Lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

```
parallel_for %i = 0 to N {
  for %j = … {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
```

```
for %j = … {
  parallel_for %i = 0 to N {
    codeB1(%i, %j);
    sync_threads;
    codeB2(%i, %j);
  }
}
```

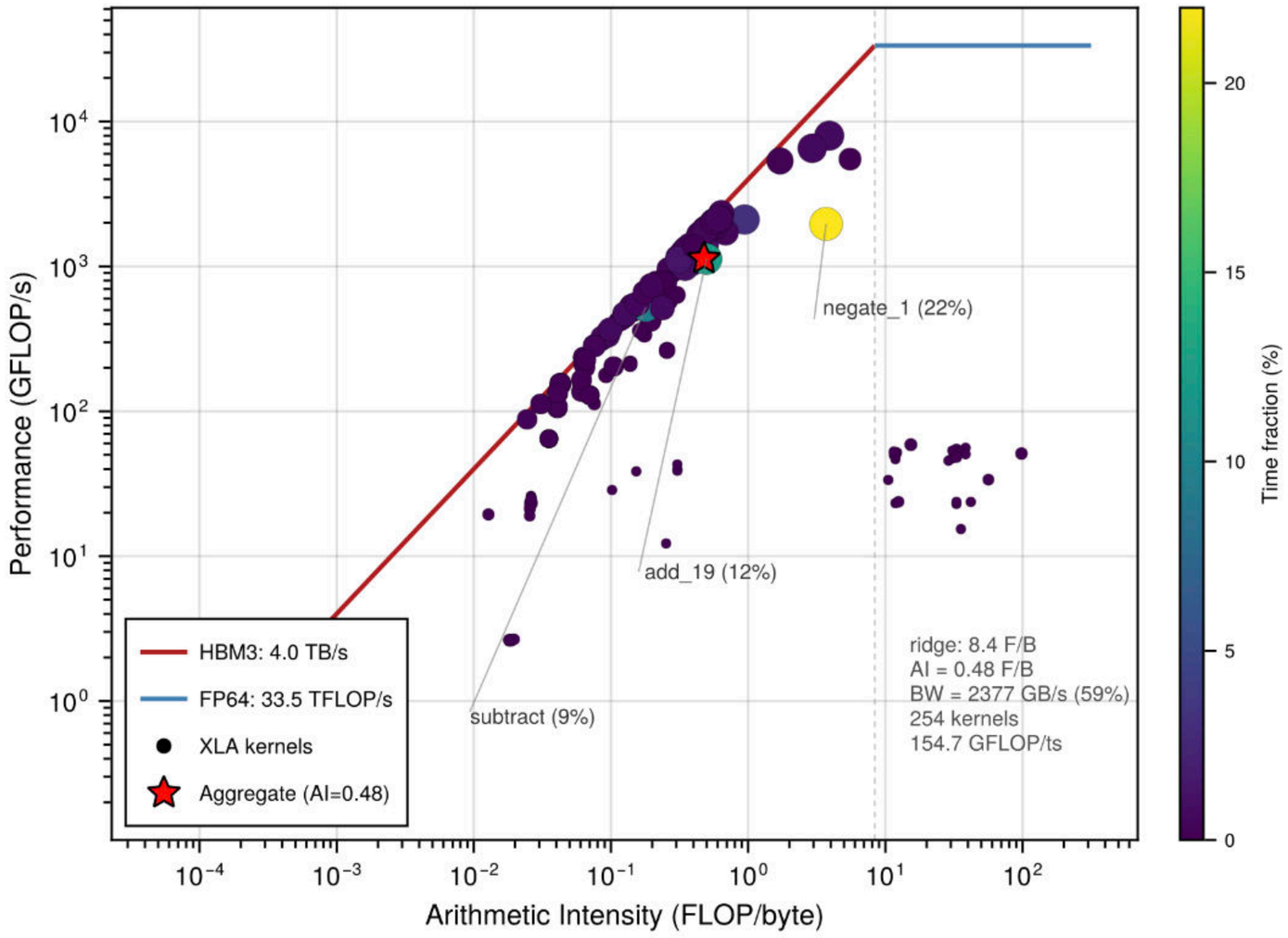
Allocation Merging

- Allocations (and any calls) on the GPU are expensive
- Given two allocations in the same scope, replace uses with a single allocation
- Beneficial for not just AD, but any GPU programs!

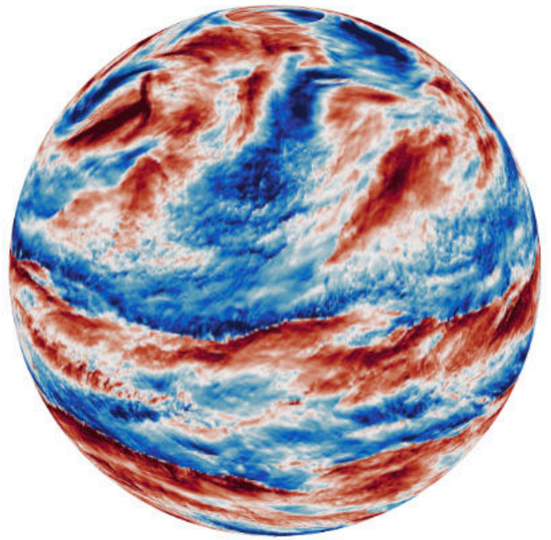
```
double* var1 = new double[N];
double* var2 = new double[M];

use(var1, var2);

delete[] var1;
delete[] var2;
```

```
double* var1 = new double[N + M];
double* var2 = var1 + N;

use(var1, var2);

delete[] var1;
```

Performance Results (Alps / NVIDIA GPU)

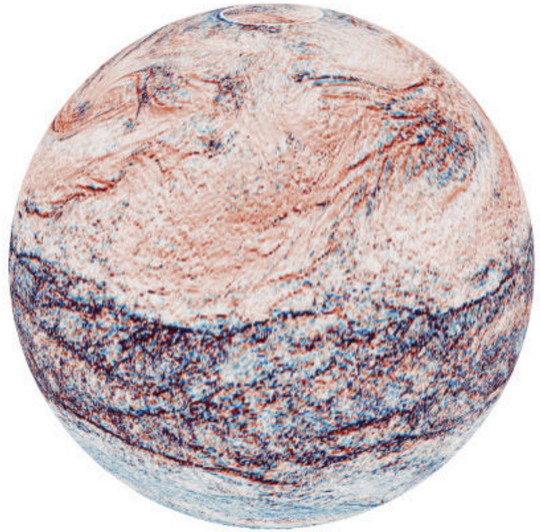


Meridional momentum, ρv



Sensitivity ∂J / ∂(ρv₀)

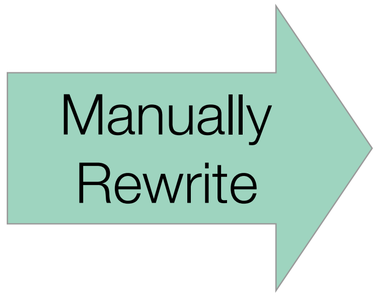
J = V⁻¹ ∫ (ρθ)² dV



Existing AD Approaches (1/3)

- Differentiable DSL (TensorFlow, PyTorch, DiffTaichi)
  - Provide a new language designed to be differentiated
  - Requires rewriting everything in the DSL and the DSL must support all operations in original code
  - Fast if DSL matches original code well

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




```
import tensorflow as tf

x = tf.Variable(3.14)

with tf.GradientTape() as tape:
  out = tf.cond(x > 0,
            lambda: tf.math.pow(x,3),
            lambda: 0
          )
print(tape.gradient(out, x).numpy())
```

Existing AD Approaches (2/3)

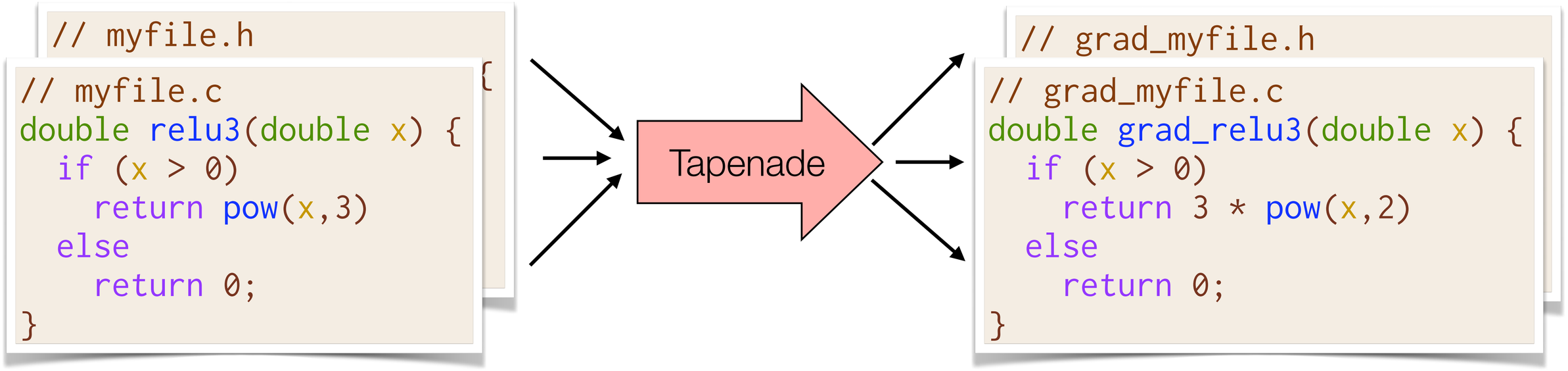
- Operator overloading (Adept, JAX)
  - Differentiable versions of existing language constructs (double => adouble, np.sum => jax.sum)
  - May require writing to use non-standard utilities
  - Often dynamic: storing instructions/values to later be interpreted

```
// Rewrite to accept either
//    double or adouble
template<typename T>
T relu3(T val) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```

```
adept::Stack stack;
adept::adouble inp = 3.14;

// Store all instructions into stack
adept::adouble out(relu3(inp));
out.set_gradient(1.00);

// Interpret all stack instructions
double res = inp.get_gradient(3.14);
```

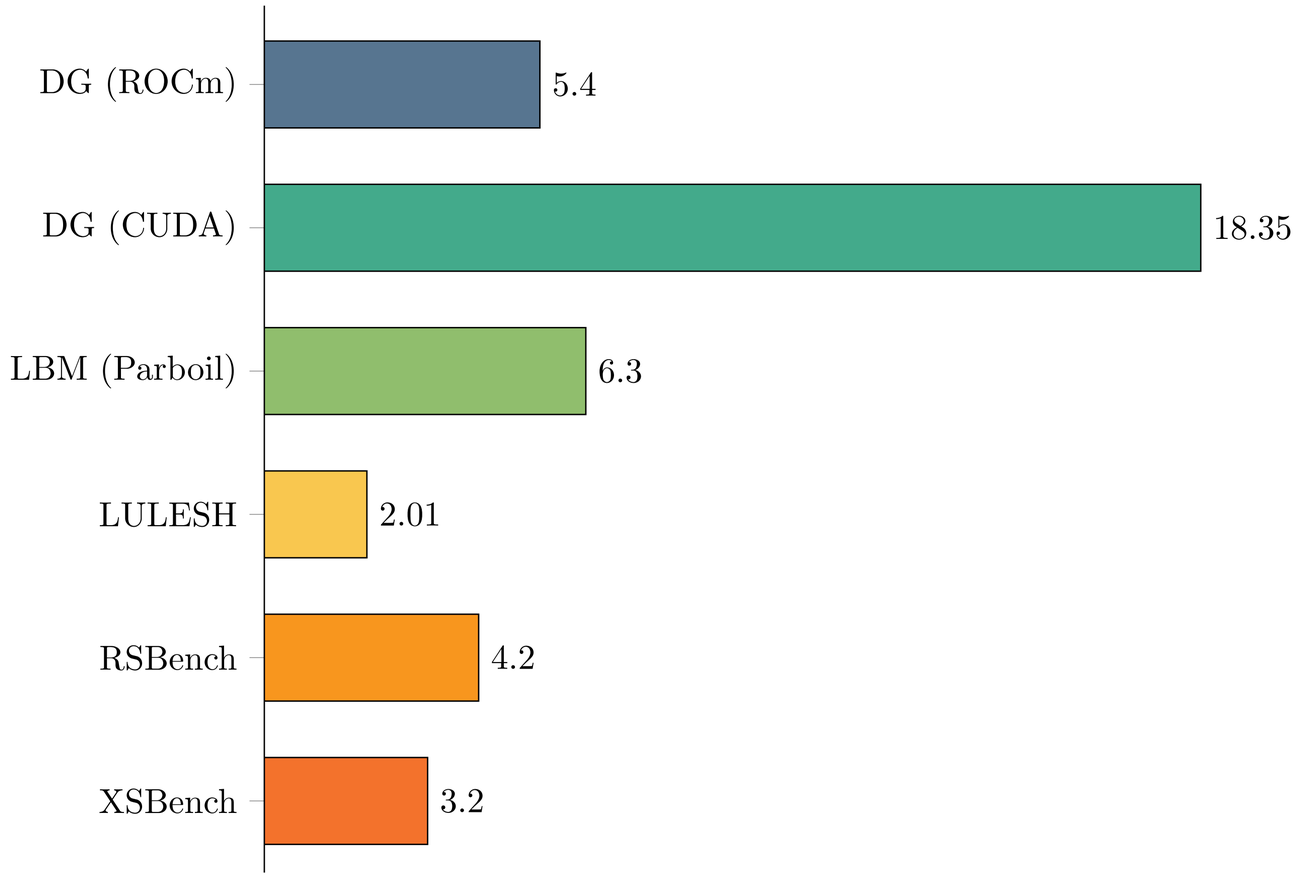
Existing AD Approaches (3/3)

- Source rewriting
  - Statically analyze program to produce a new gradient function in the source language
  - Re-implement parsing and semantics of given language
  - Requires all code to be available ahead of time => hard to use with external libraries



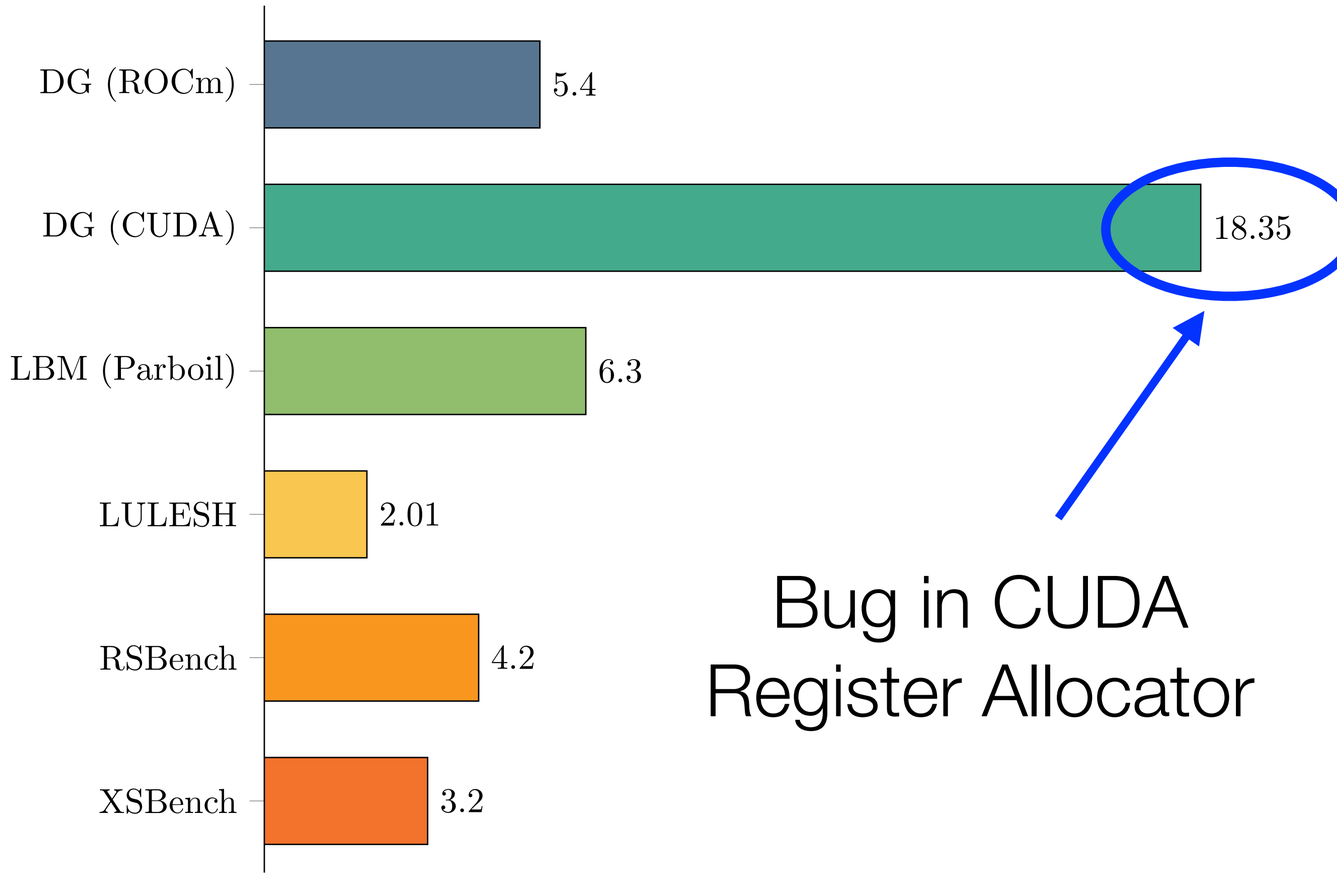
GPU Gradient Overhead [MCPHNMJ'21]

- 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)

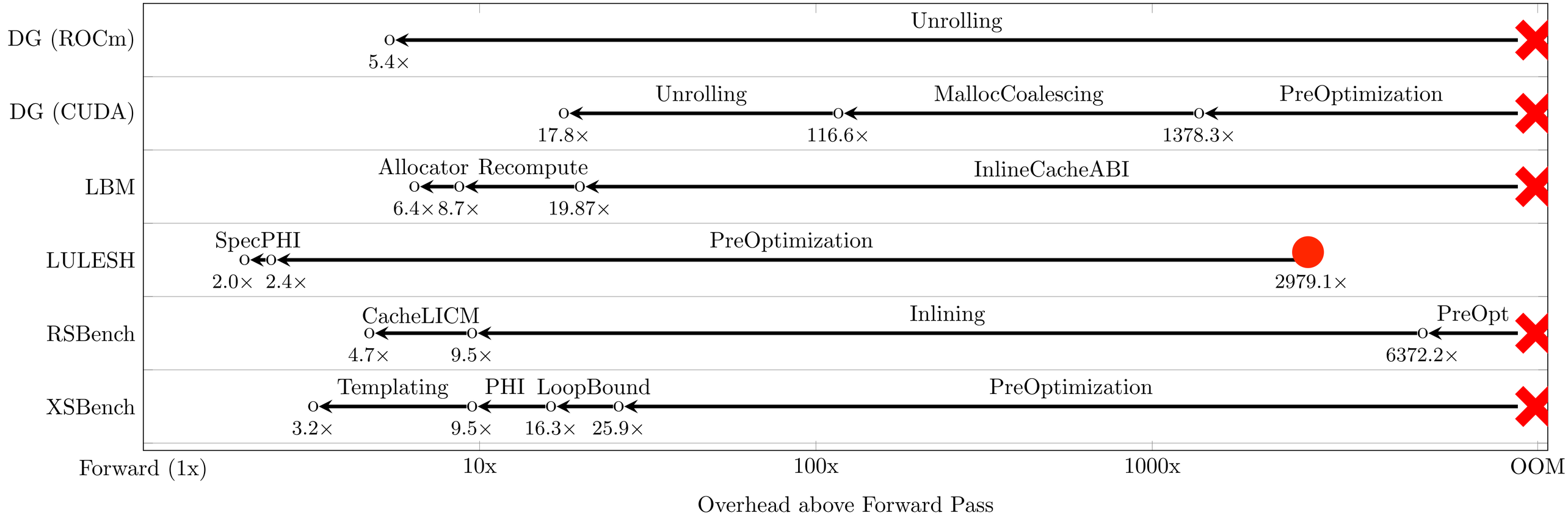


GPU Gradient Overhead [MCPHNMJ'21]

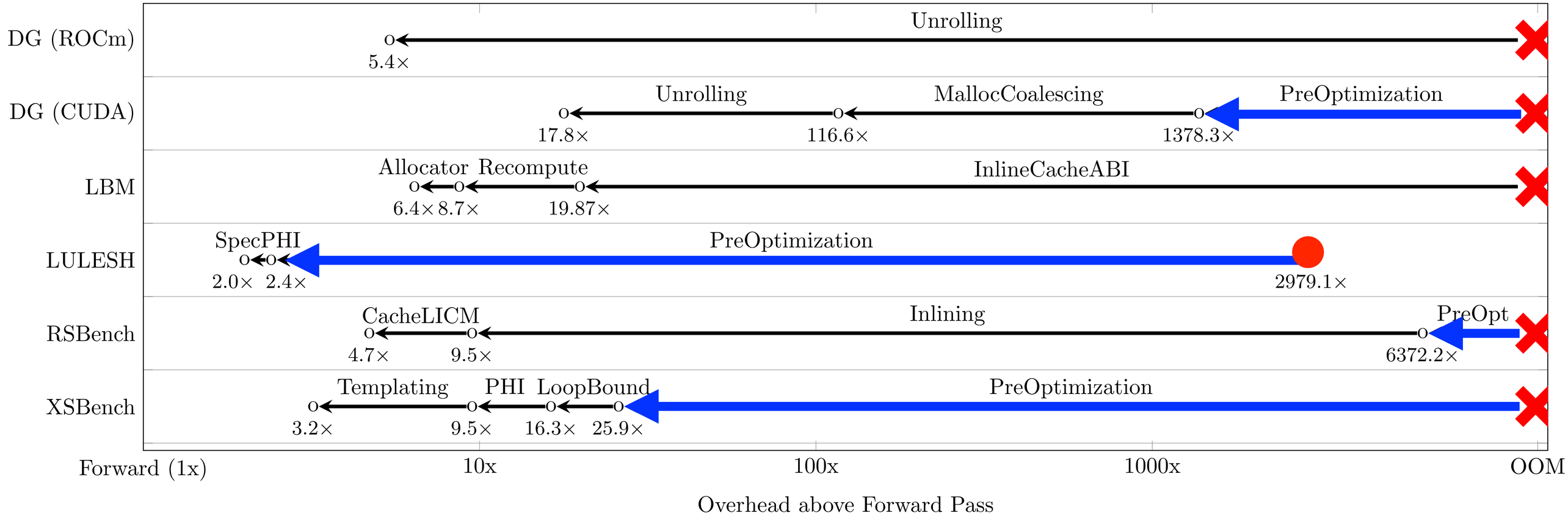
- Evaluation of both original code and gradient
  - DG: Discontinuous-Galerkin integral (Julia)
  - LBM: particle-based fluid dynamics simulation
  - LULESH: unstructured explicit shock hydrodynamics solver
  - XSBench & RSBench: Monte Carlo simulations of particle transport algorithms (memory & compute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



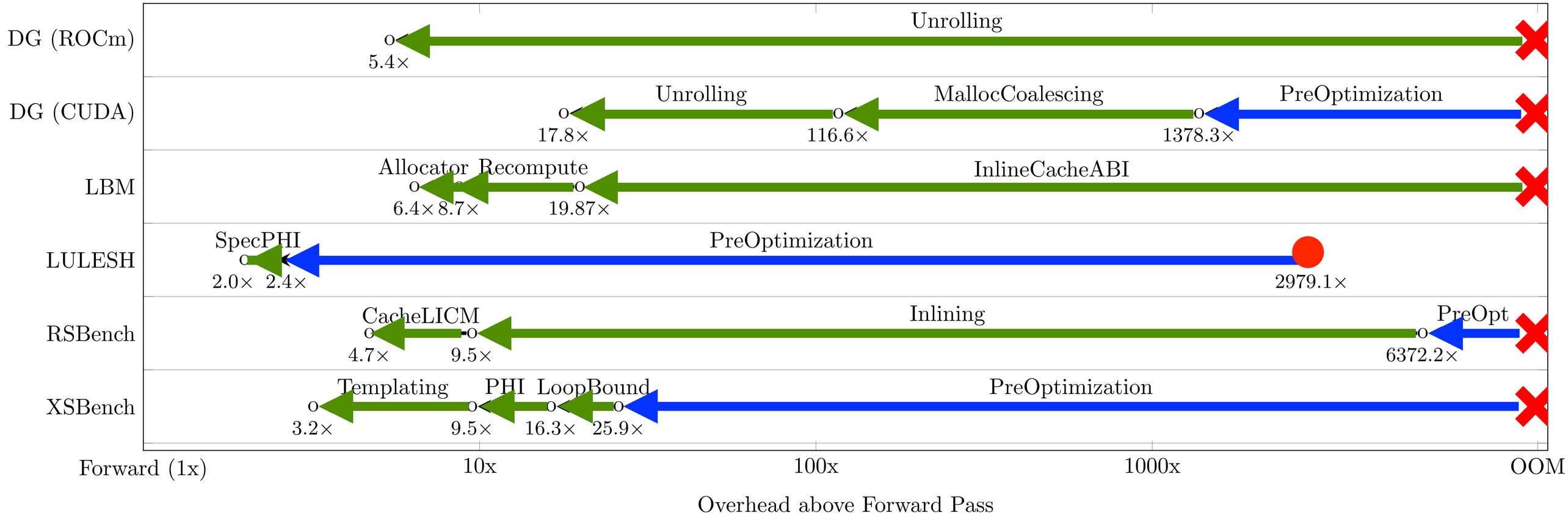
Ablation Analysis of Optimizations



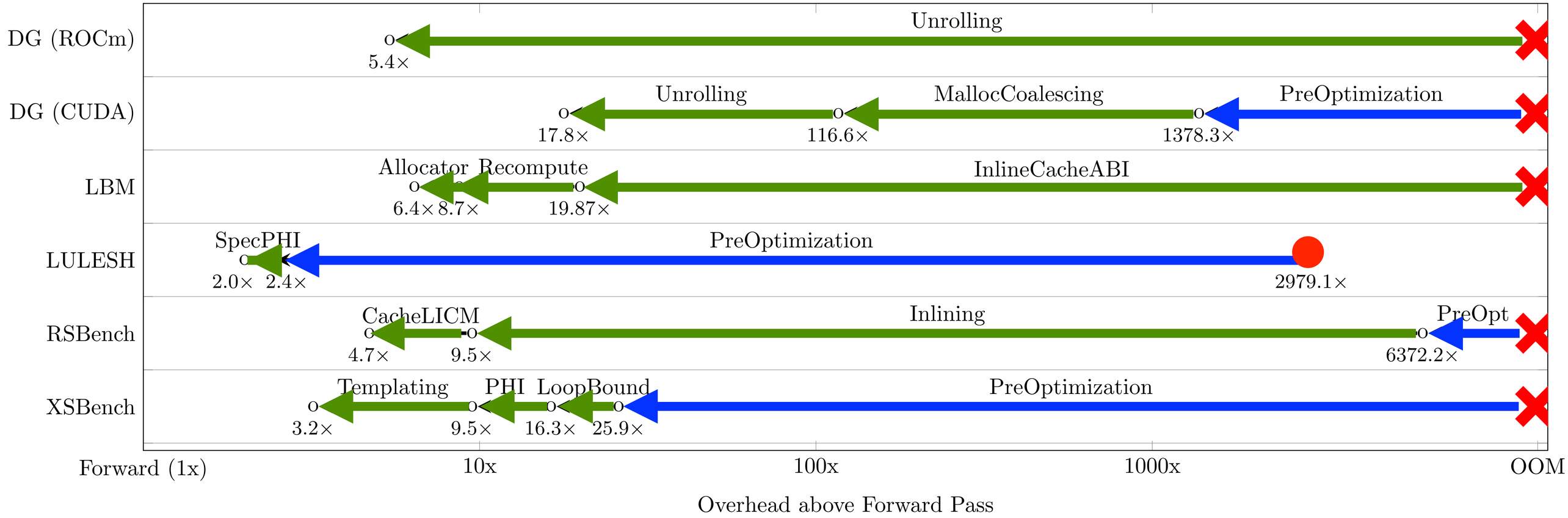
Ablation Analysis of Optimizations



Ablation Analysis of Optimizations



Ablation Analysis of Optimizations



GPU AD is Intractable Without Optimization!