Multi-Accelerator Automatic Differentiation

William S. Moses

wsmoses@illinois.edu

PPoPP DiffPP

Feb 1, 2026

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 [△]

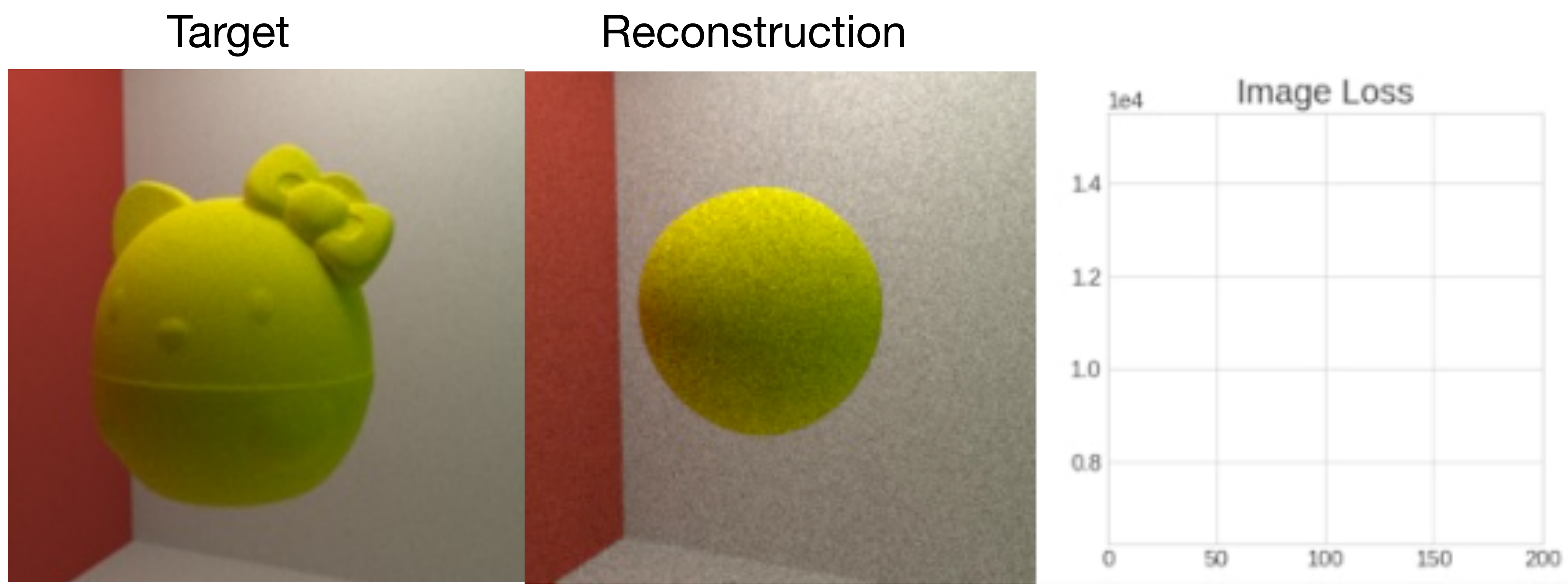
Outline

- Compiler-Based Differentiation (Enzyme-LLVM)
- Modern Computing Infrastructure
- Raising Primal Code to Run on Accelerators
- Distributed Accelerated Differentiation

Differentiation: Connecting Science and AI

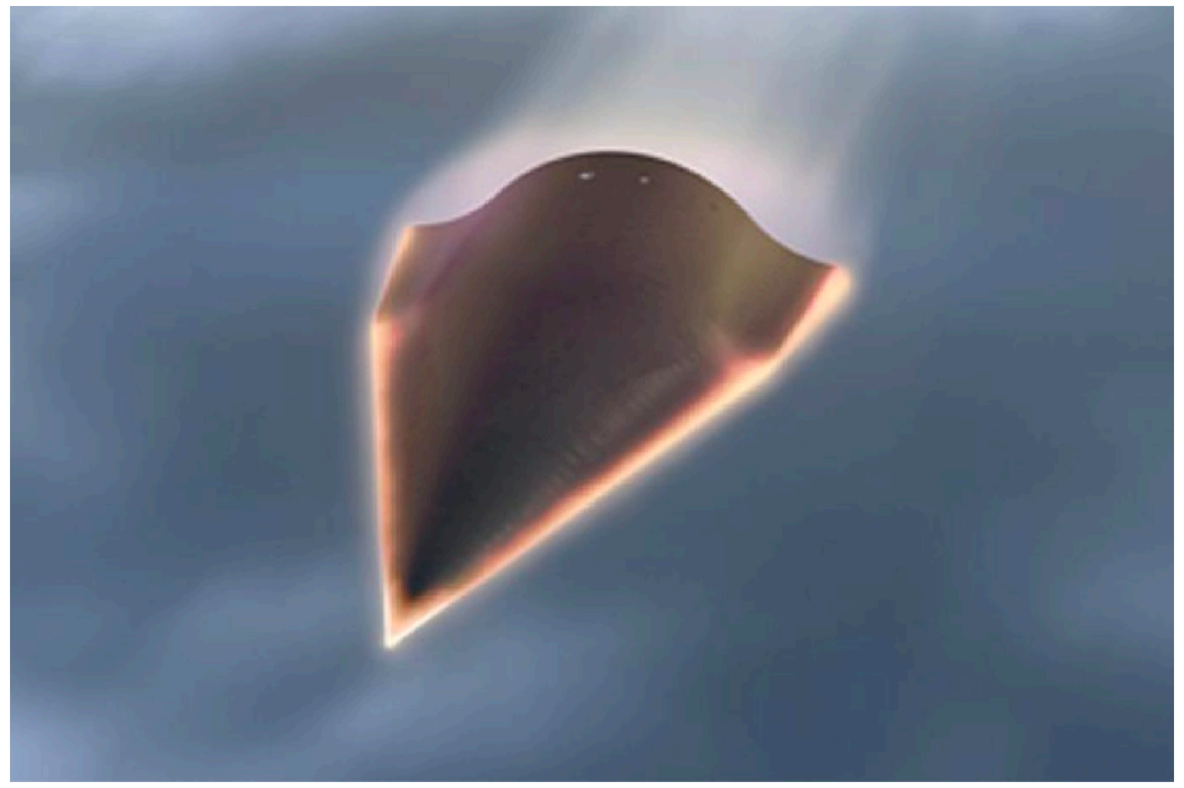
Derivatives are key to science + ML

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

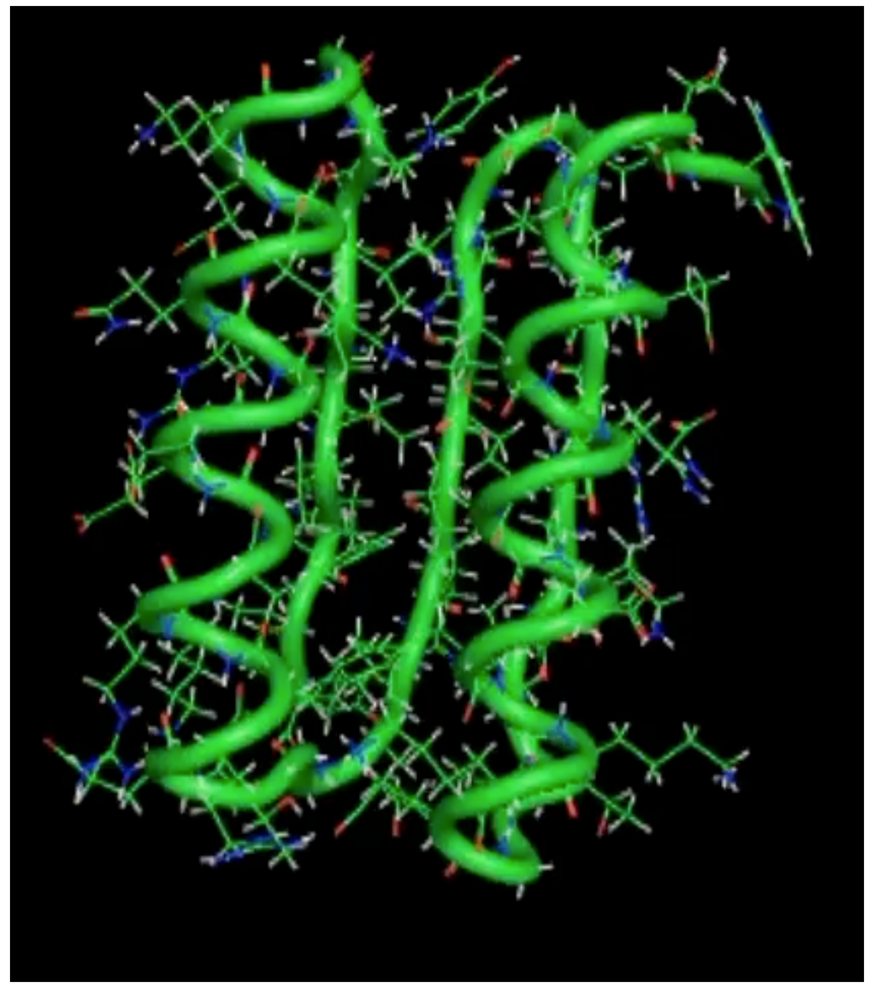


from Efficient Differentiation of Pixel Reconstruction Filters for Path-Space Differentiable Rendering, SIGGRAPH Asia 2022, Zihan Yu et al

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



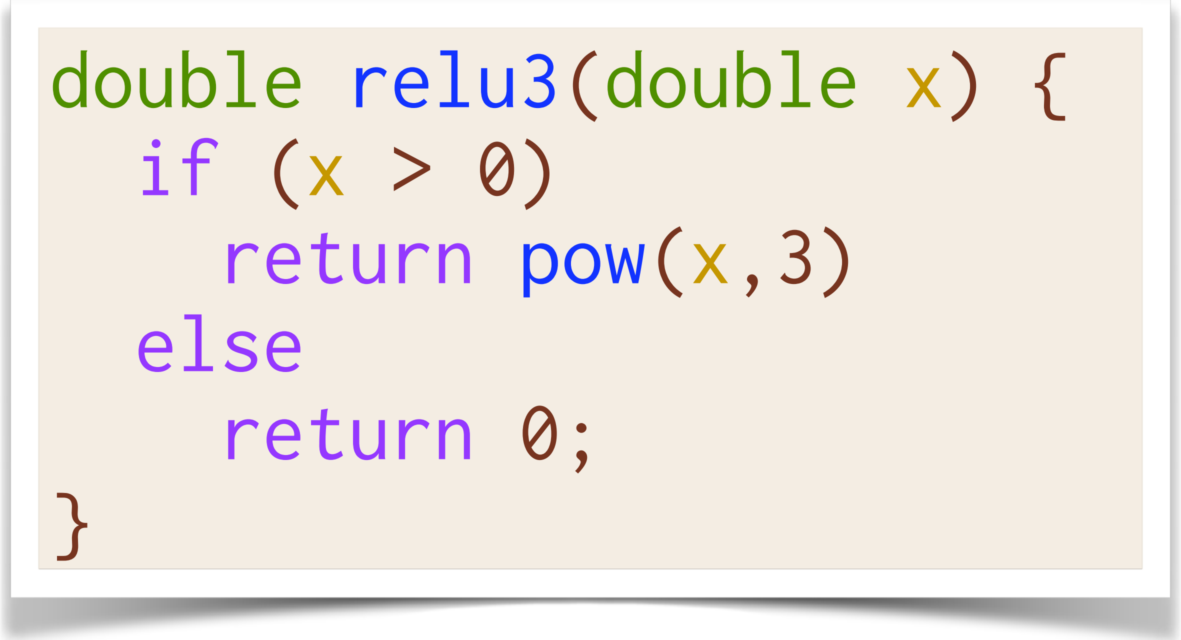
from Center for the Exascale Simulation of Materials in Extreme Environments



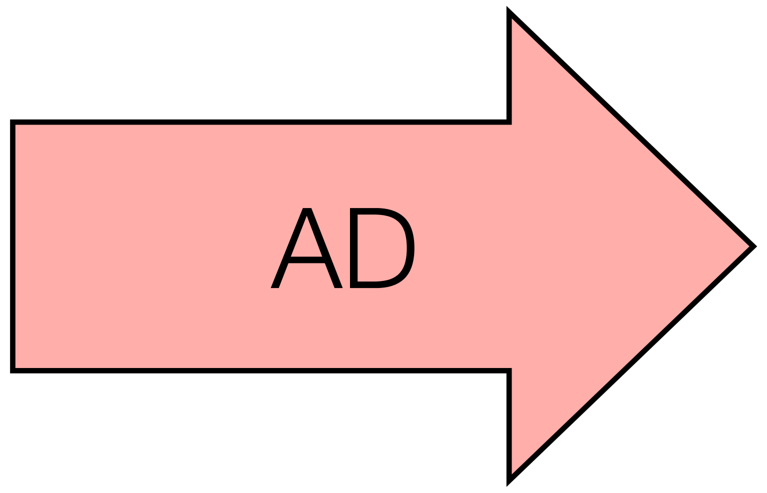
from Differential Molecular Simulation with Molly.jl, EnzymeCon 2023, Joe Greener (Cambridge)

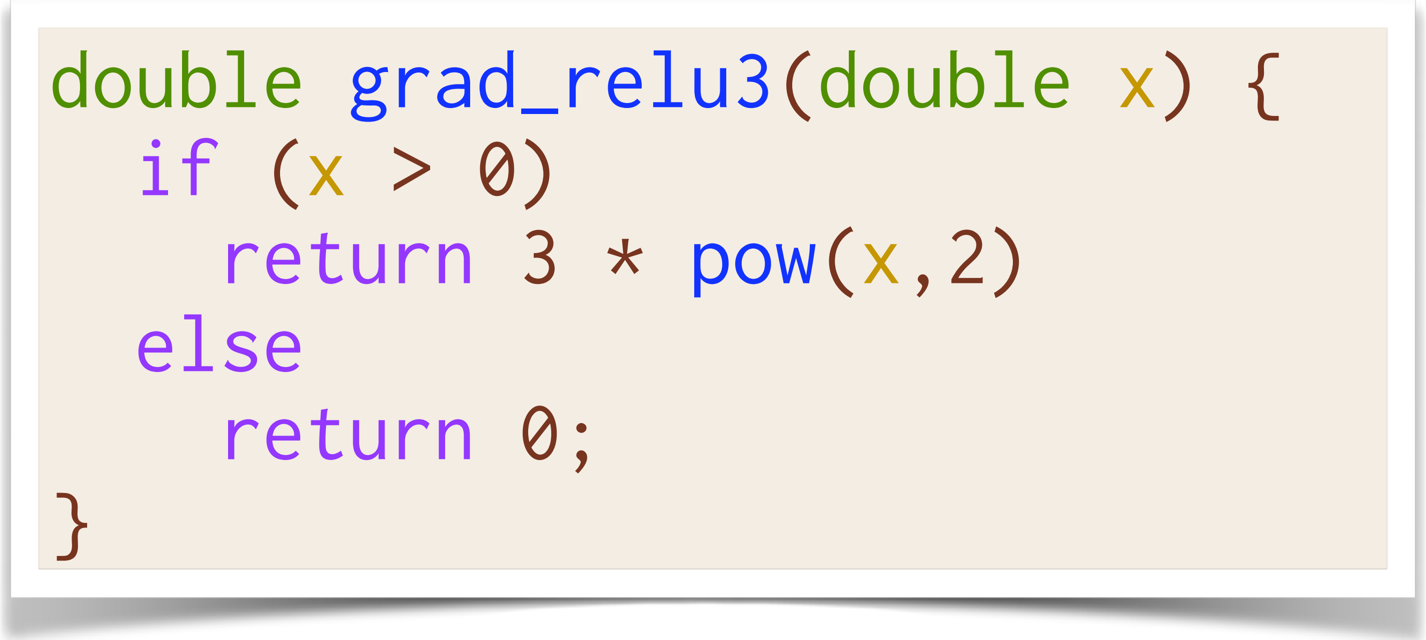
Automatic Derivative Generation

- Derivatives can be generated automatically from definitions within programs



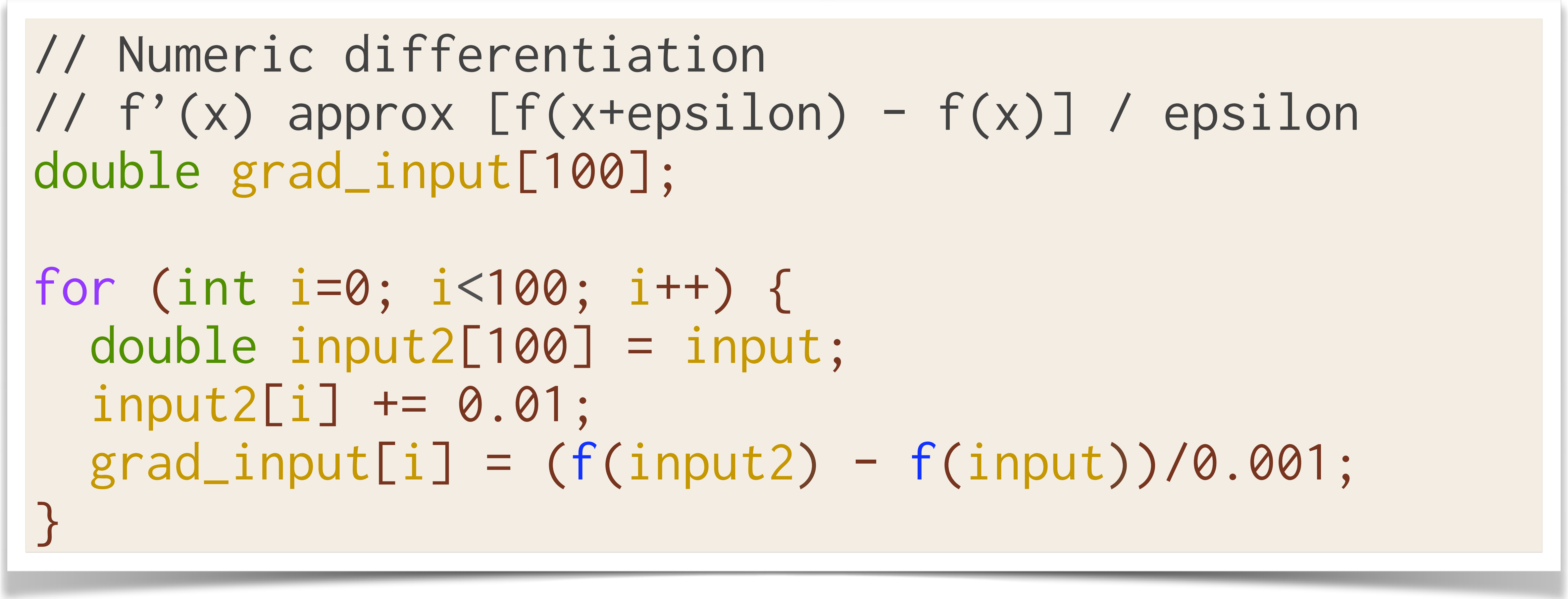
```
double relu3(double x) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```



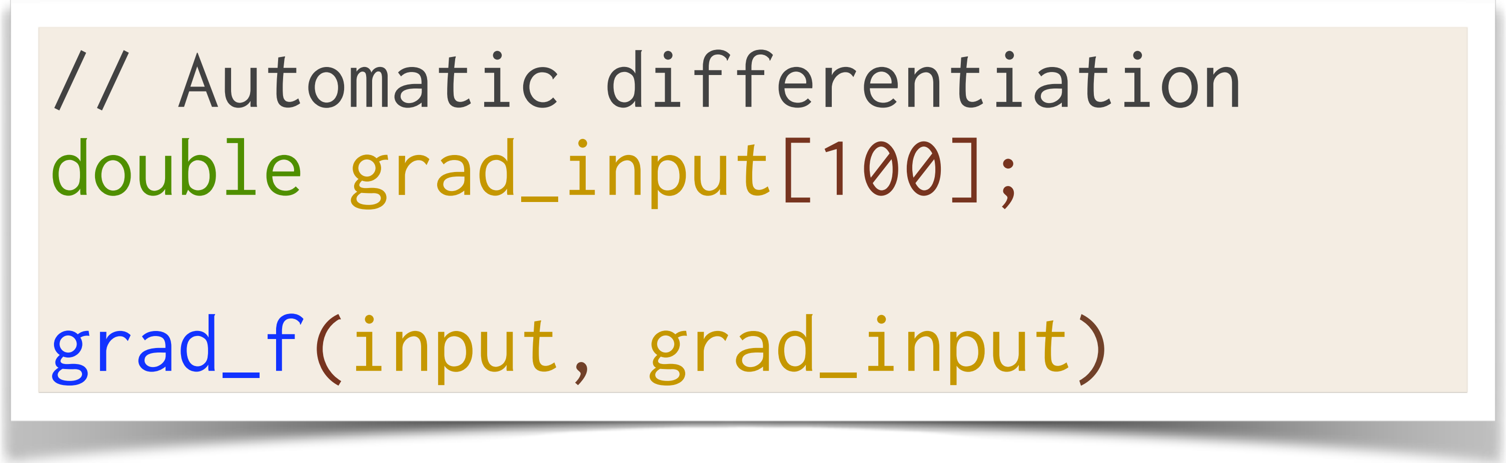


```
double grad_relu3(double x) {
  if (x > 0)
    return 3 * pow(x,2)
  else
    return 0;
}
```

- 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)
```

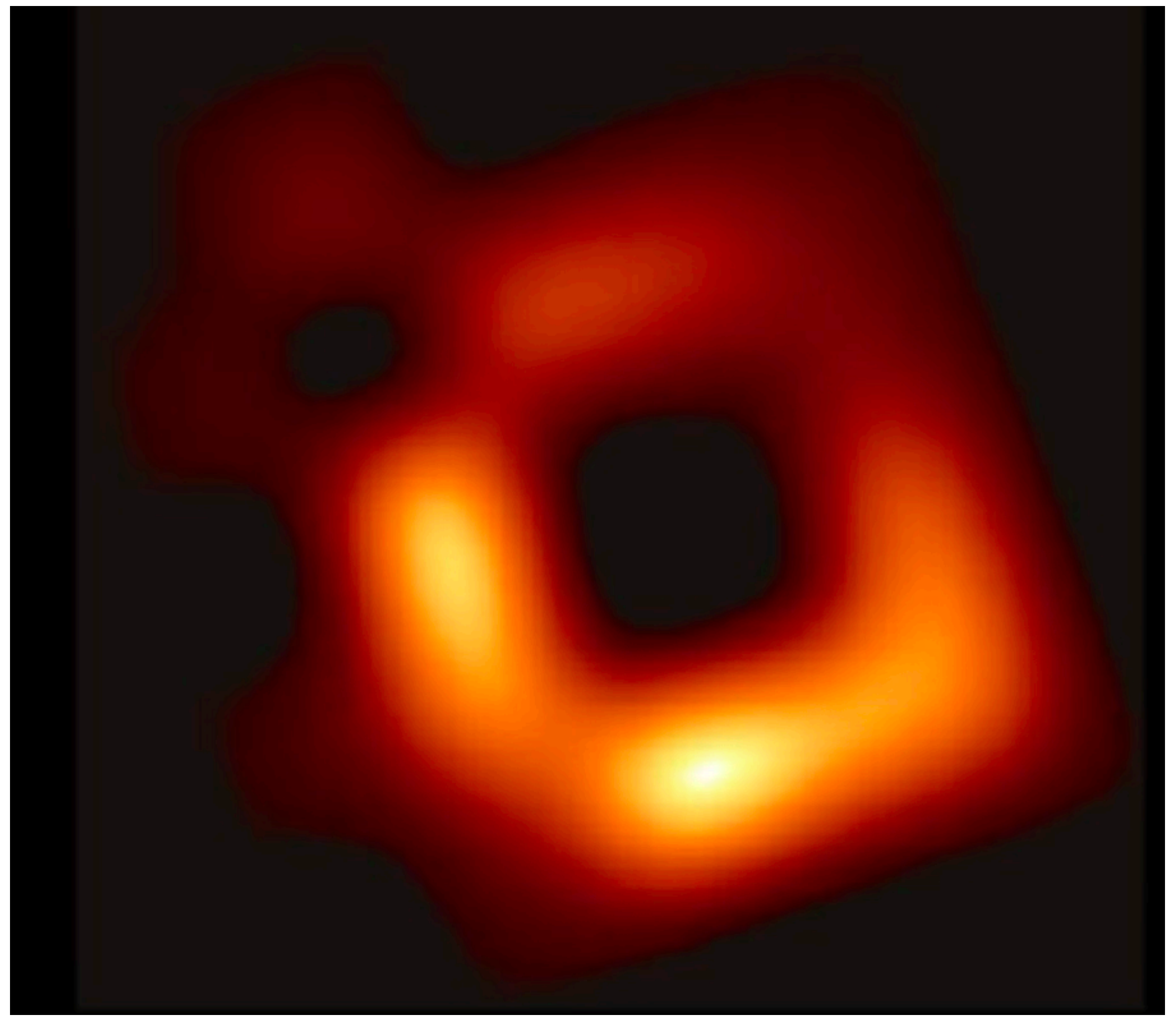
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

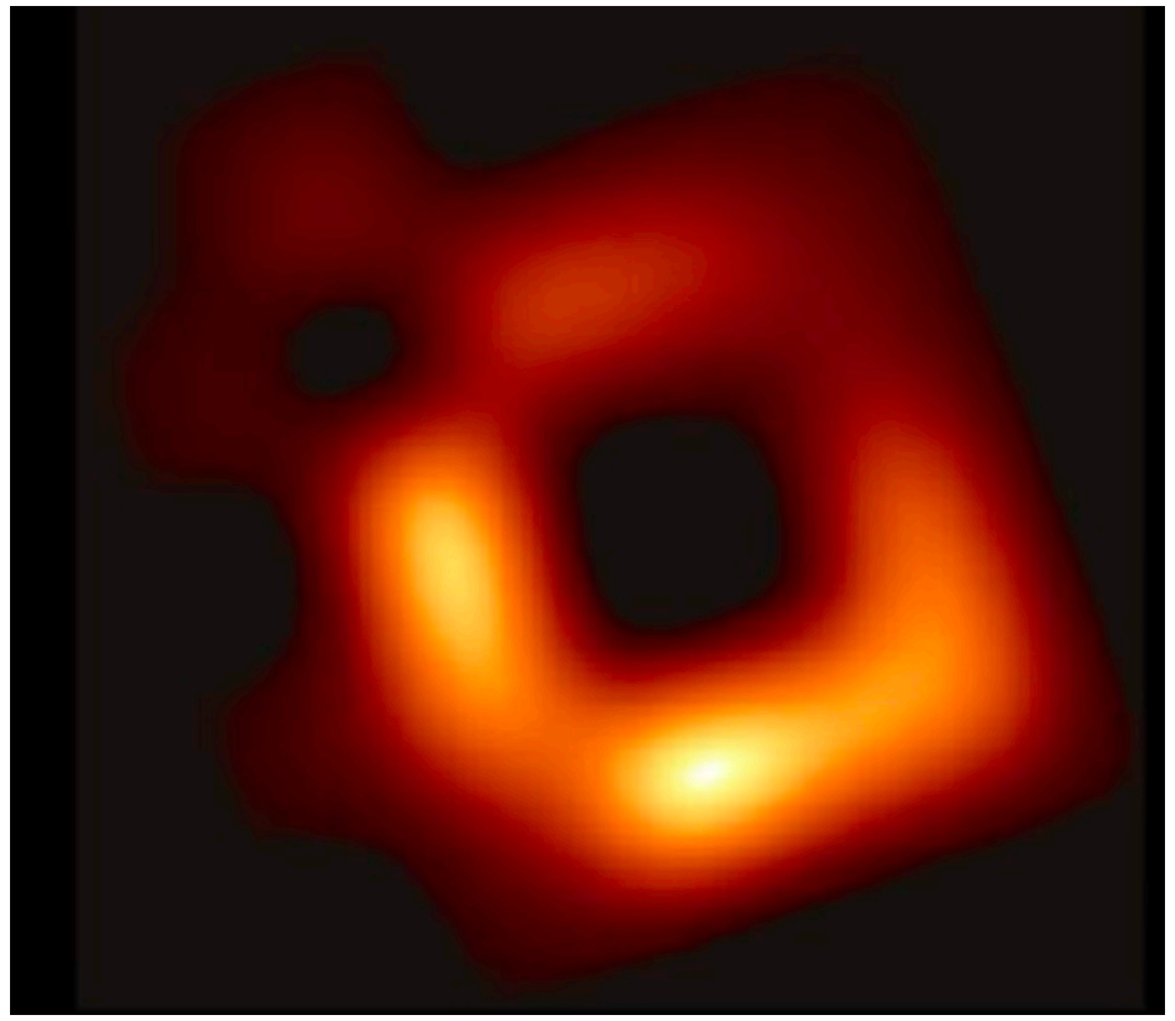
Reconstructed image of M87
~1 week on cluster
Majority runtime is derivative



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

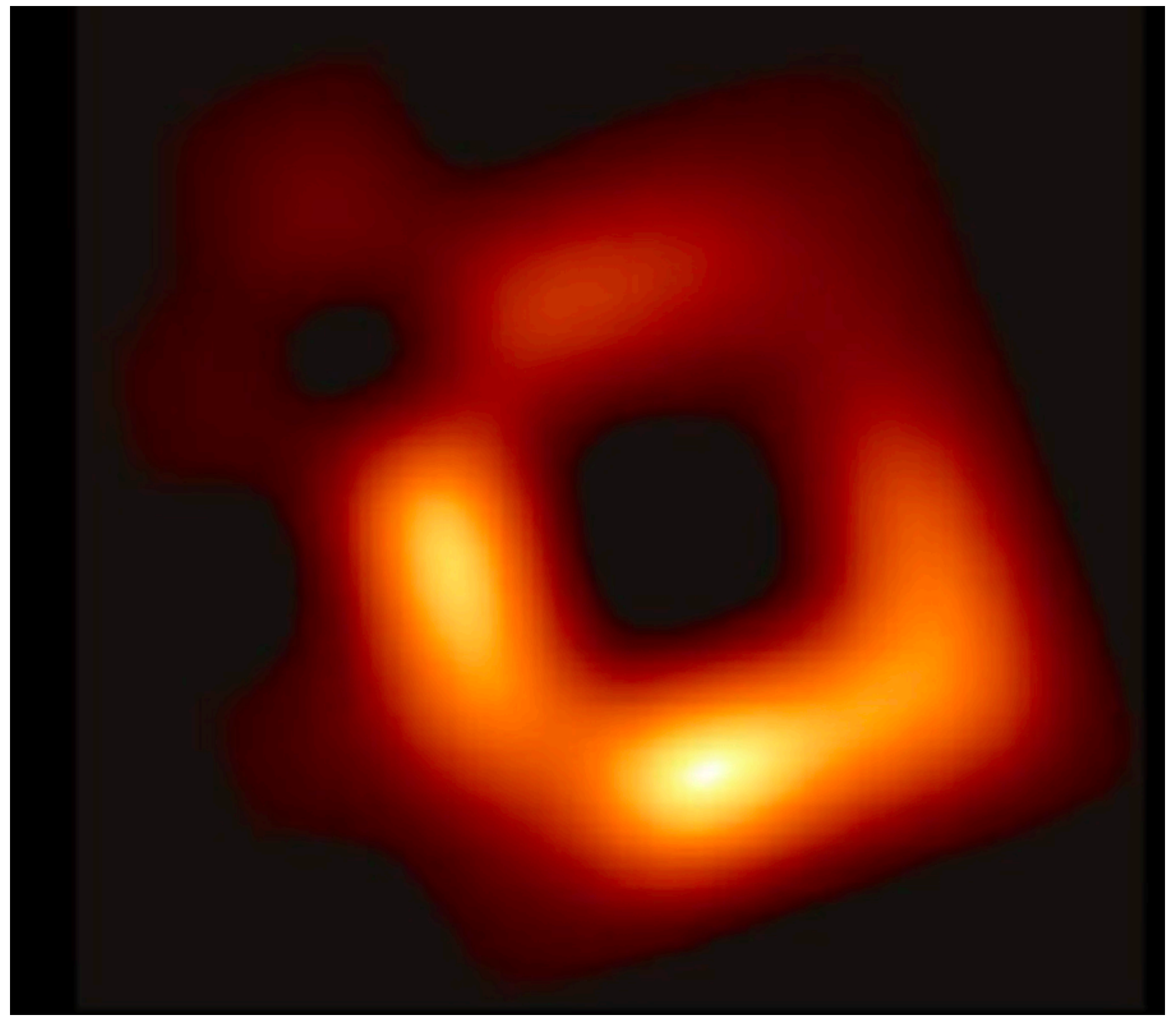


With Enzyme differentiation:
1 hour on 1 thread

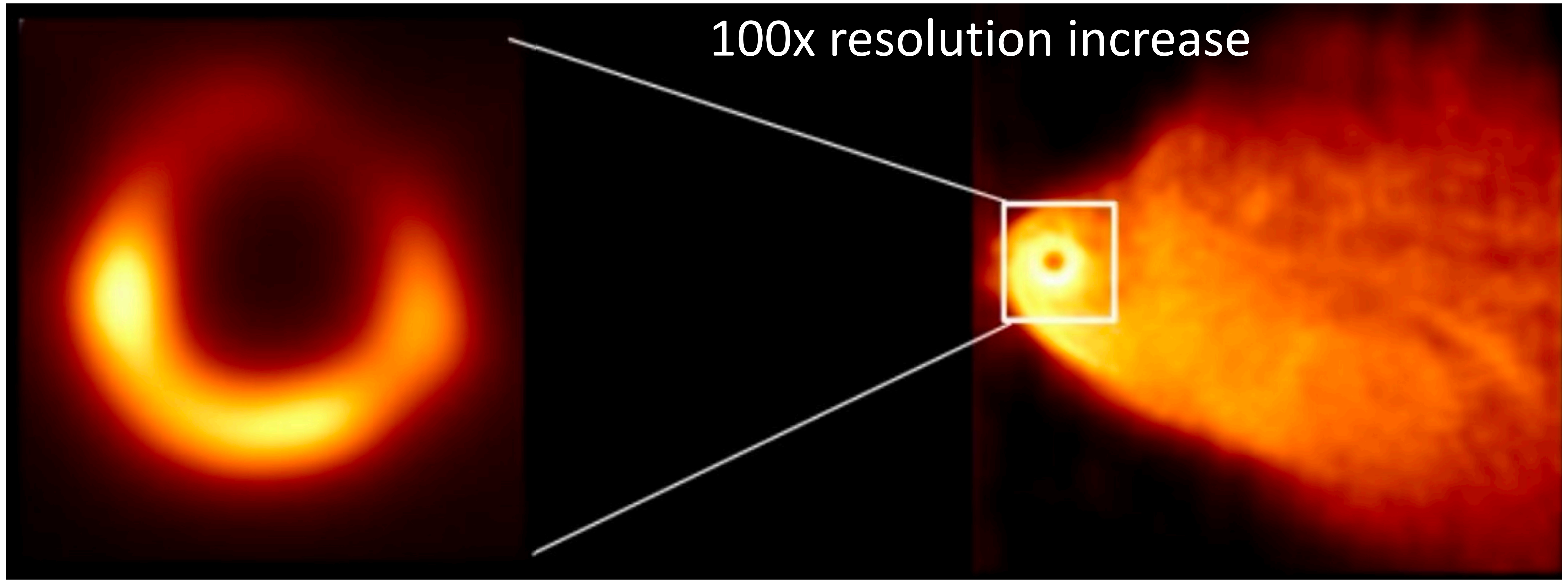
Differentiation is Expensive

Derivatives are the most costly and difficult to use algorithms

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

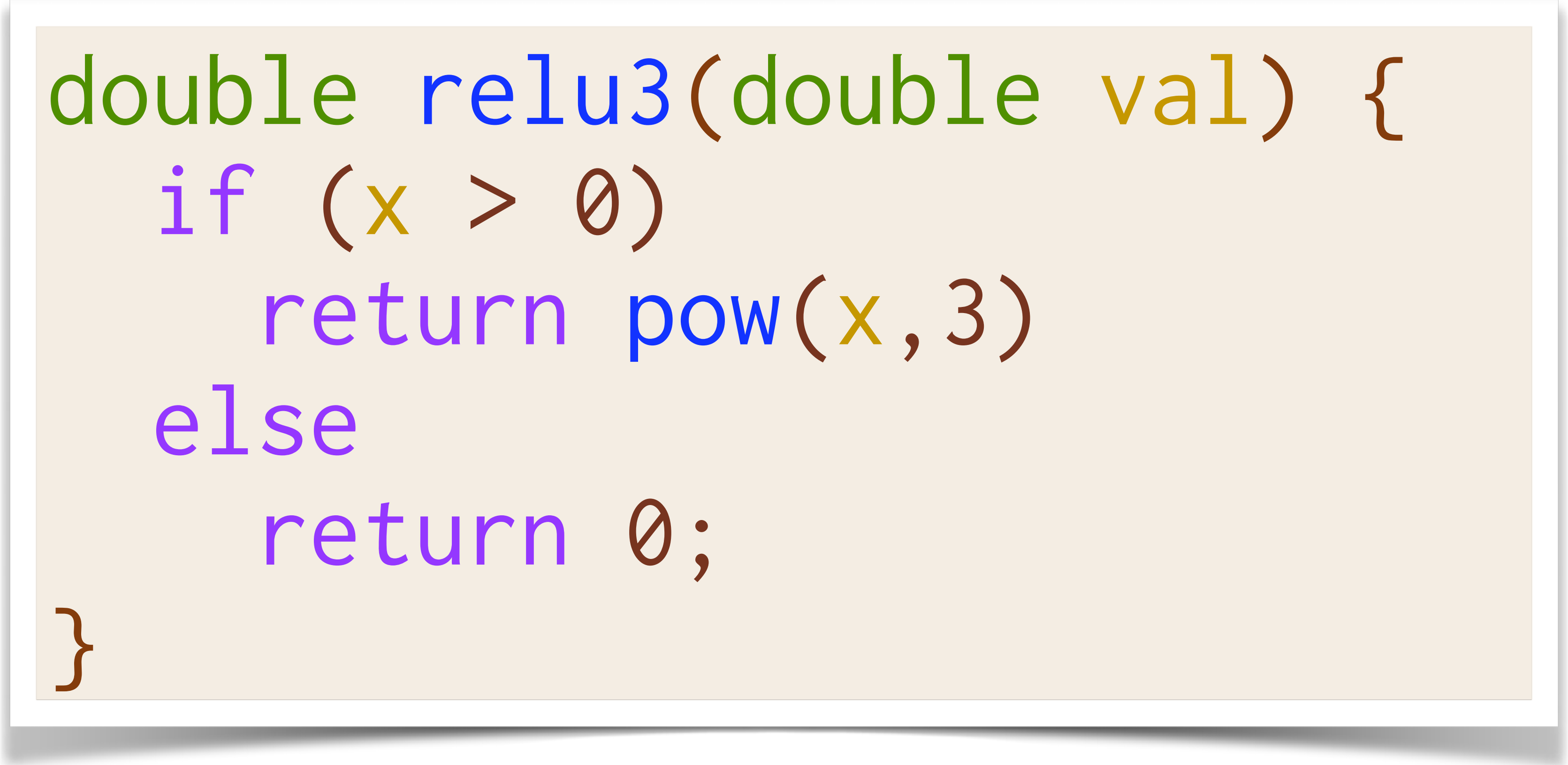


With Enzyme differentiation:
1 hour on 1 thread

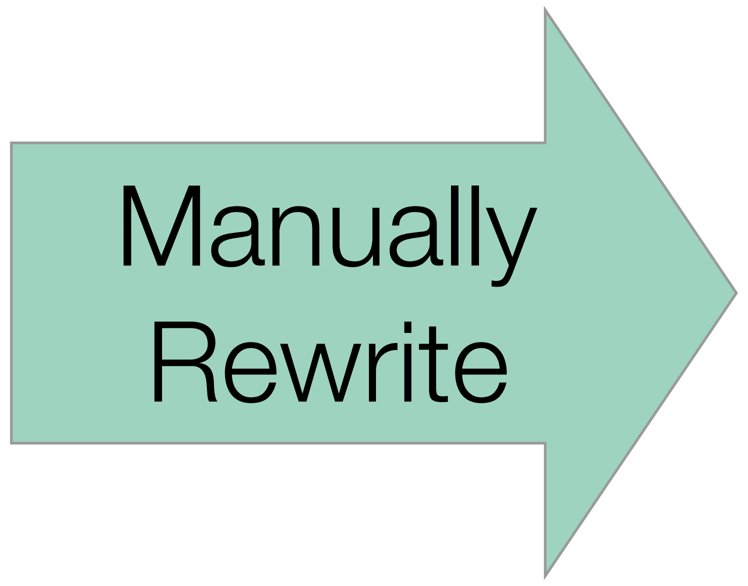


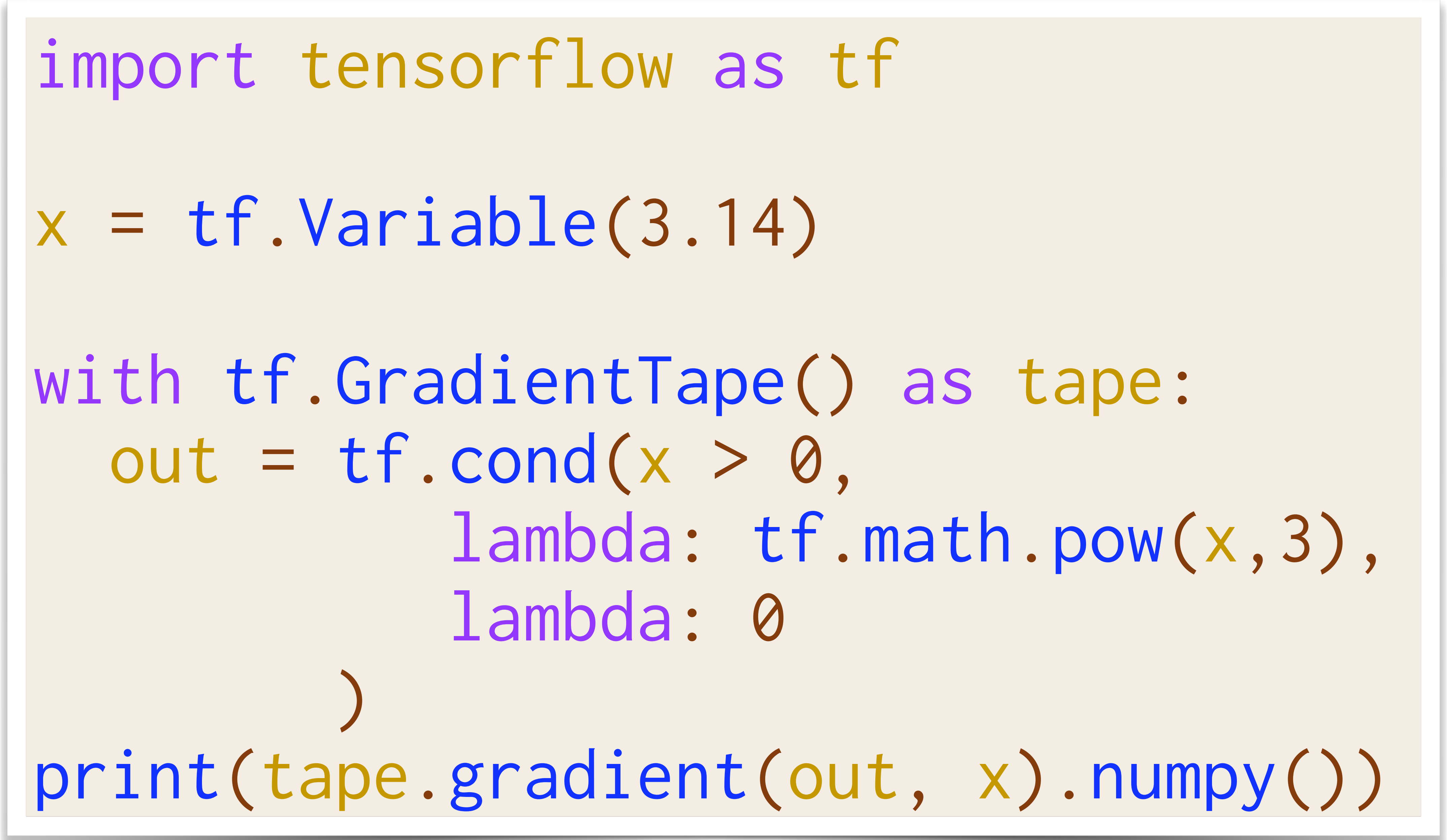
Existing AD Approaches (1/3)

- 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;
}
```

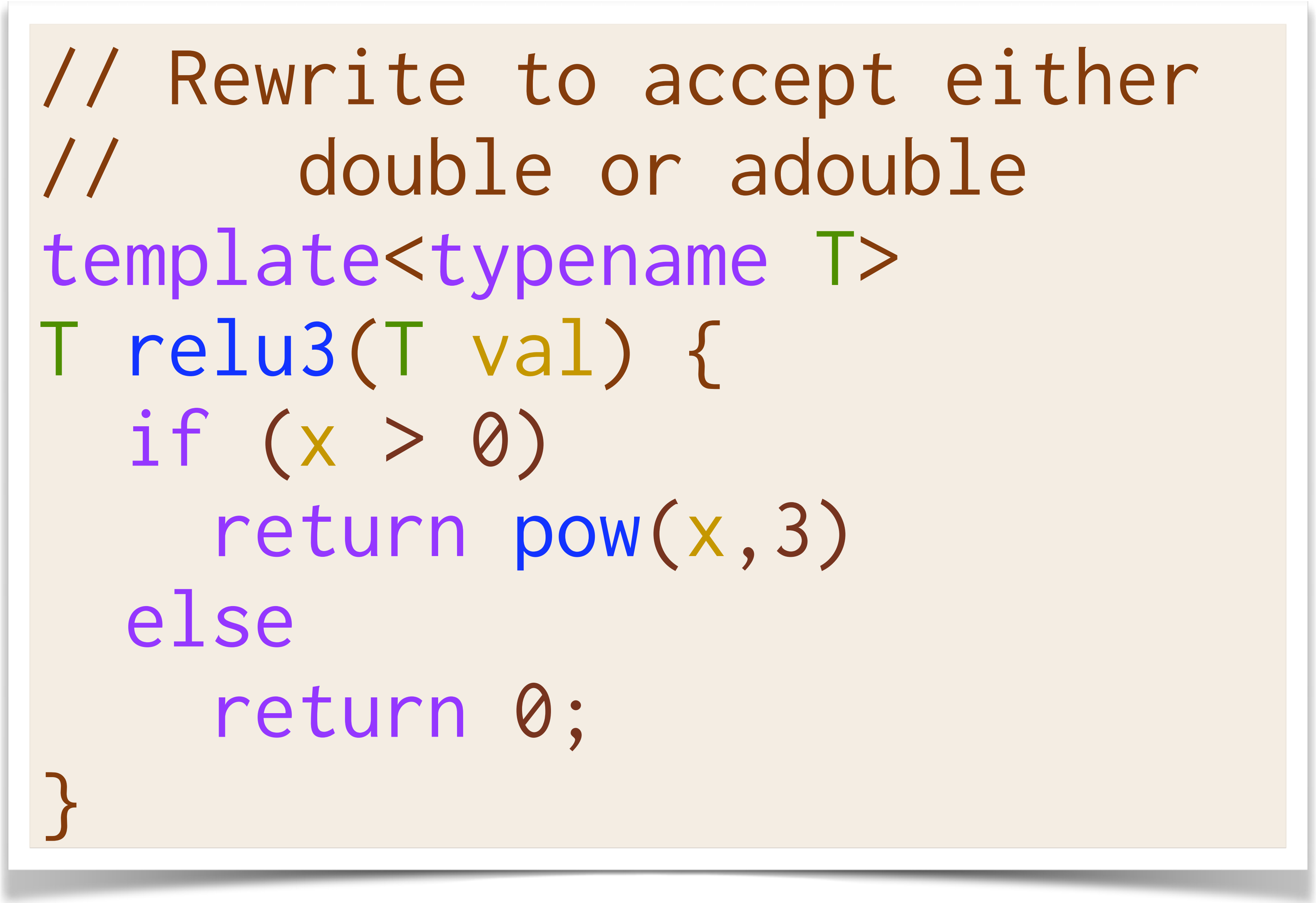




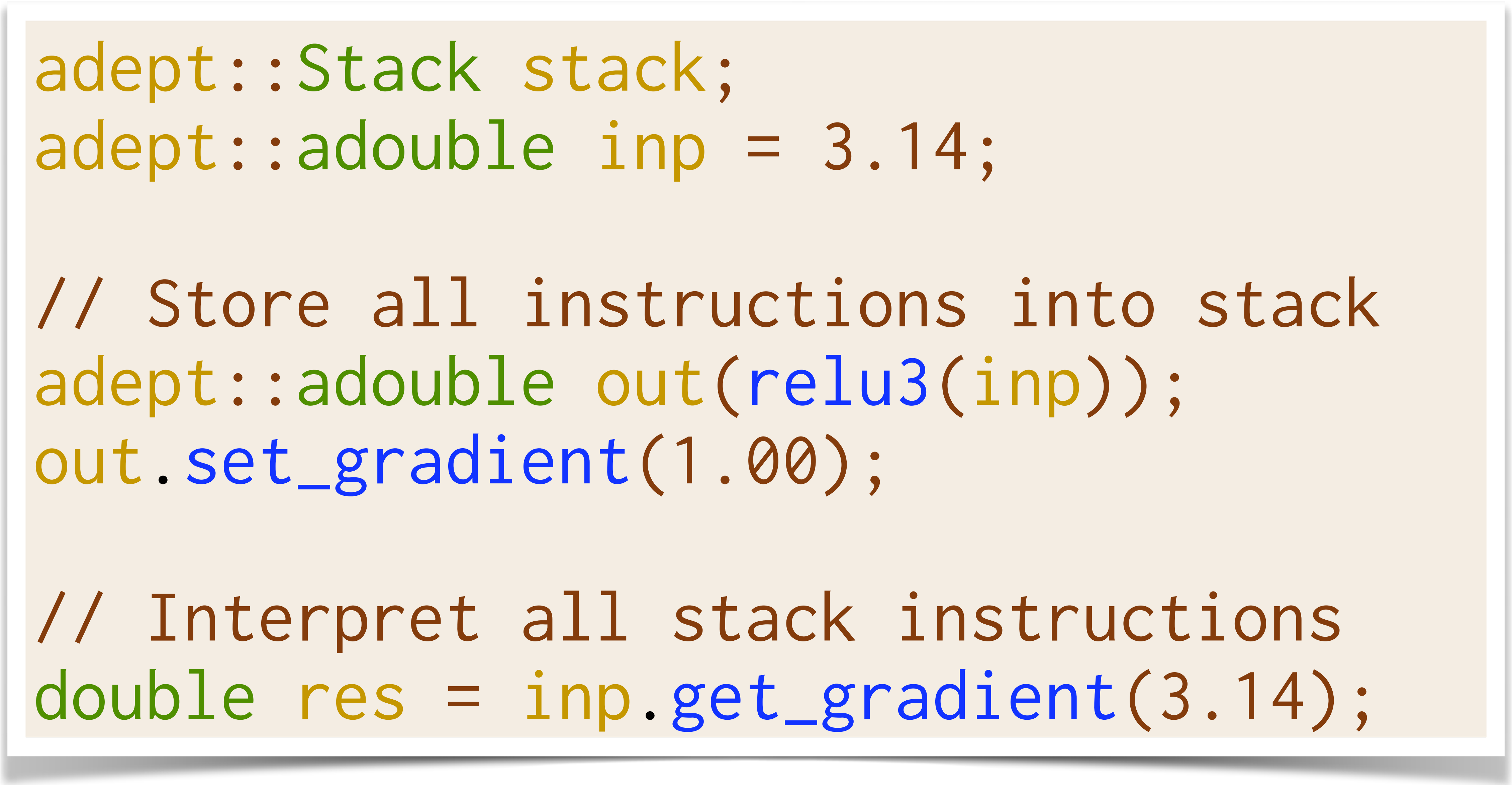
```
import tensorflow as tf

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



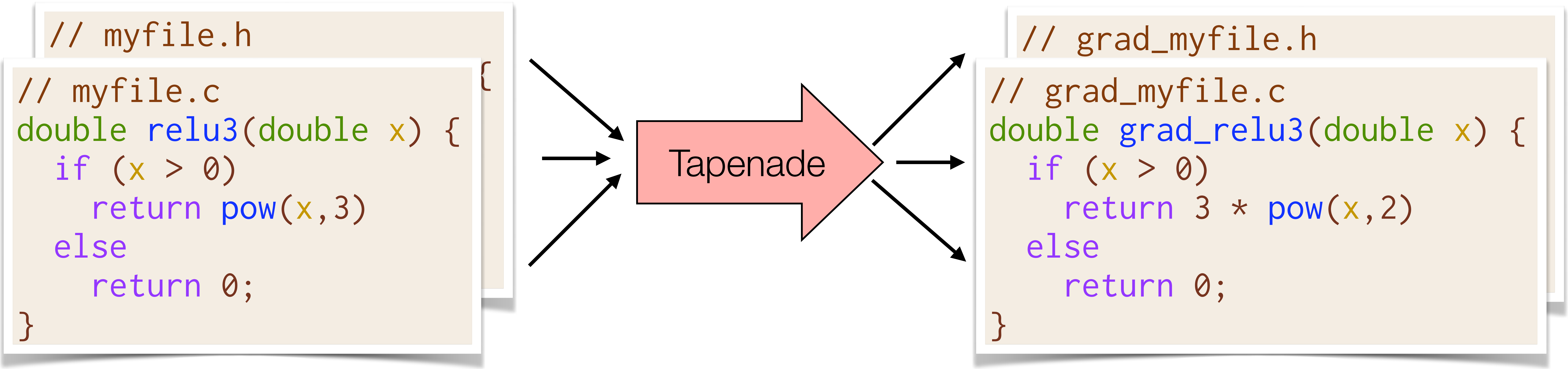
```
// Rewrite to accept either
//    double or adouble
template<typename T>
T relu3(T val) {
  if (x > 0)
    return pow(x,3)
  else
    return 0;
}
```



```
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



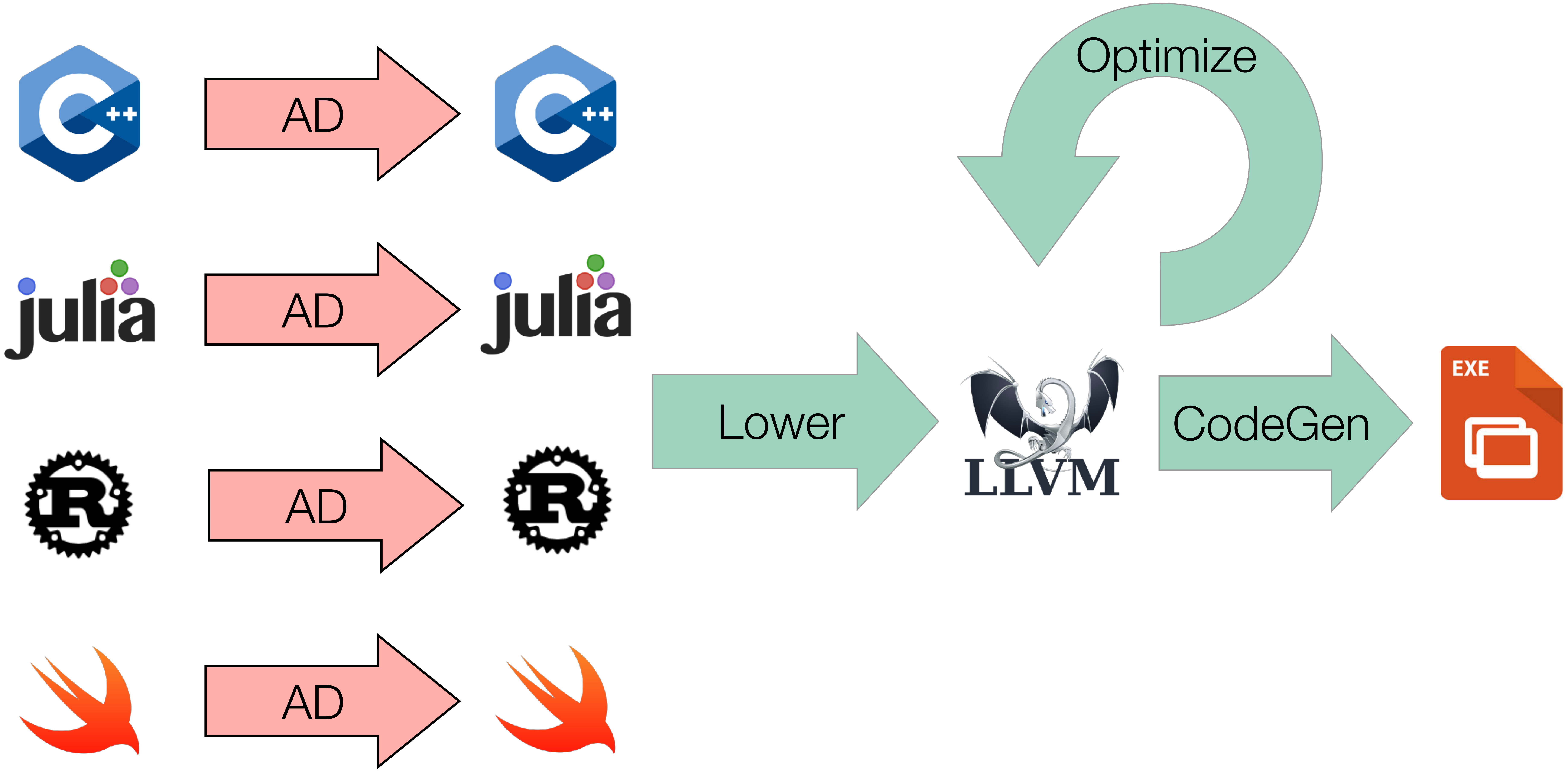
```
// myfile.h
```

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

```
// grad_myfile.h
```

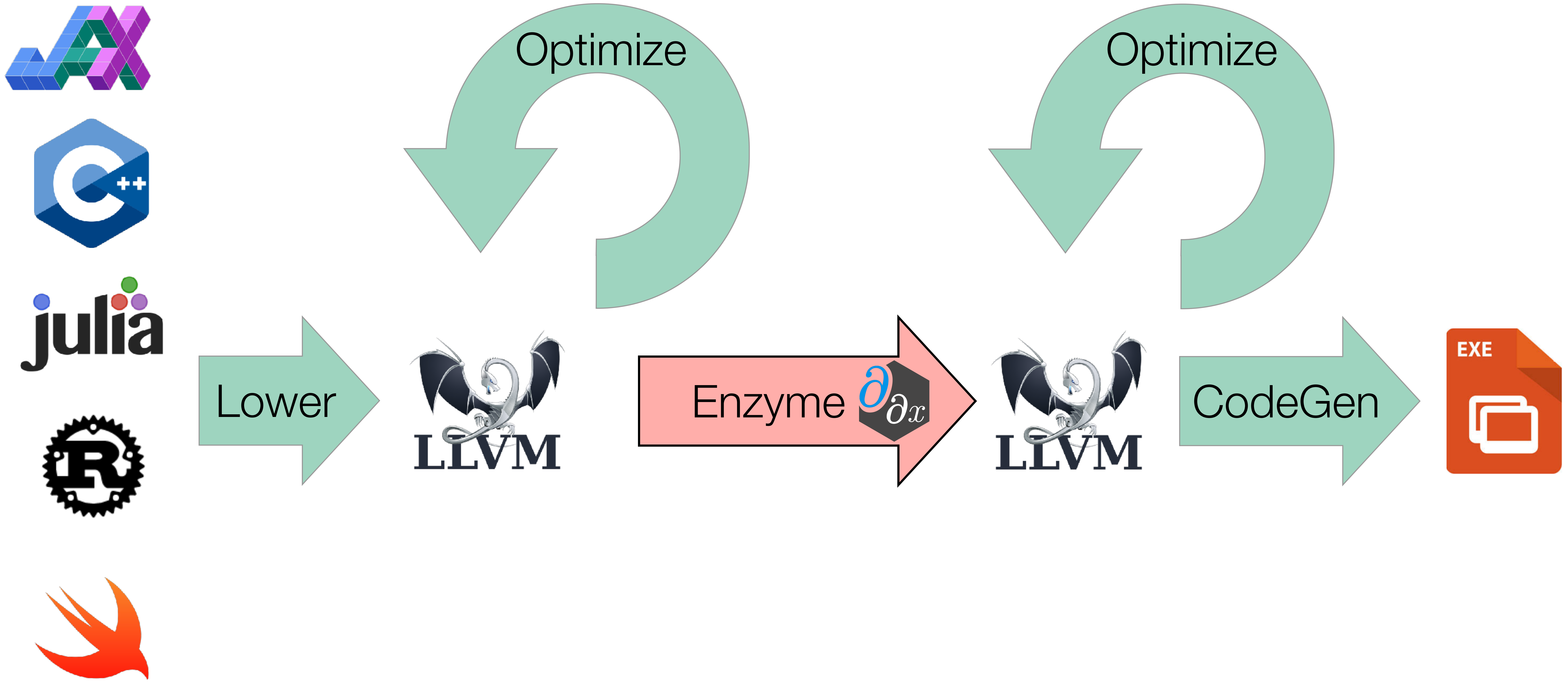
```
// grad_myfile.c
double grad_relu3(double x) {
  if (x > 0)
    return 3 * pow(x,2)
  else
    return 0;
}
```

Existing Automatic Differentiation Pipelines

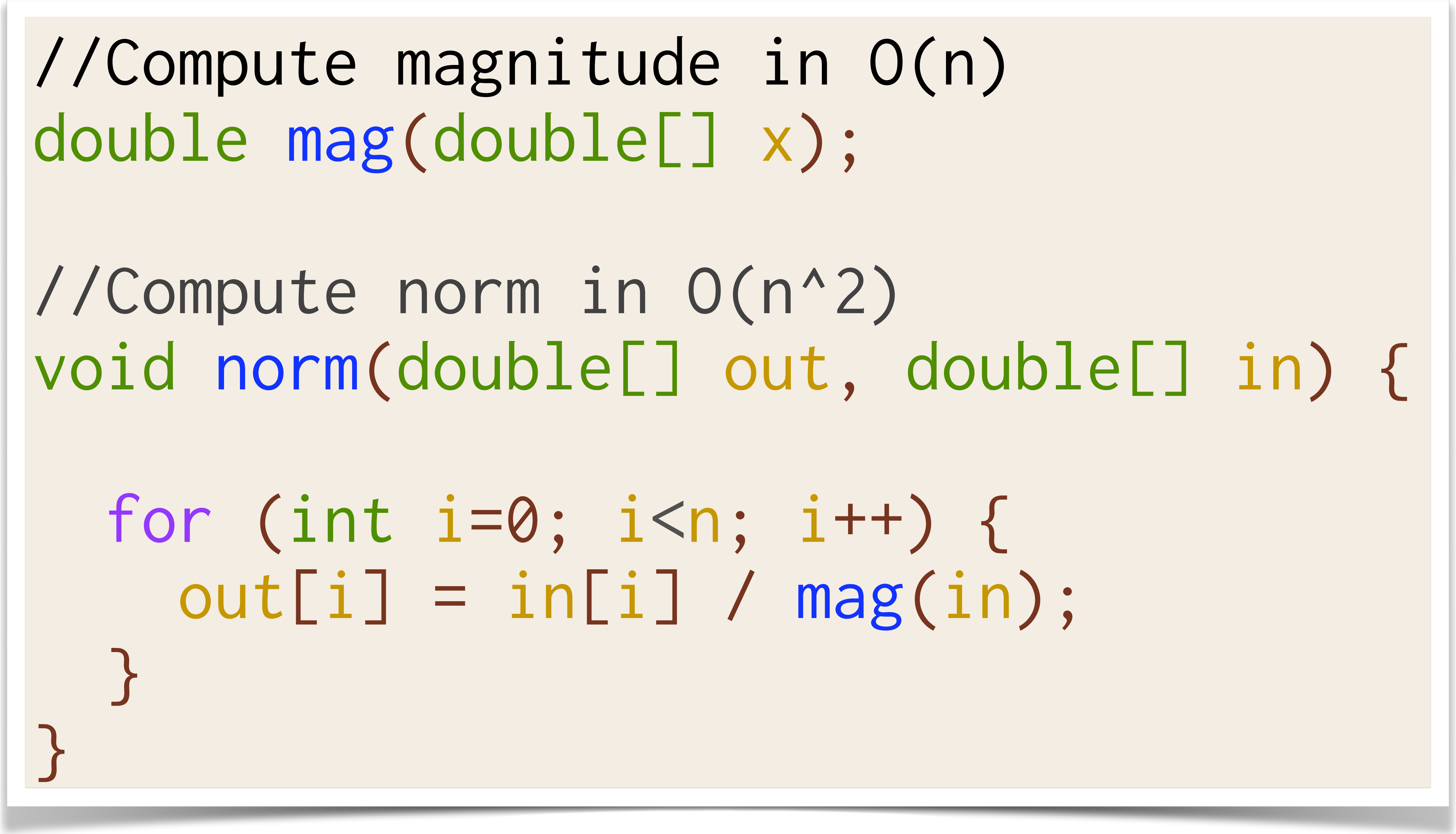


Enzyme Approach

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

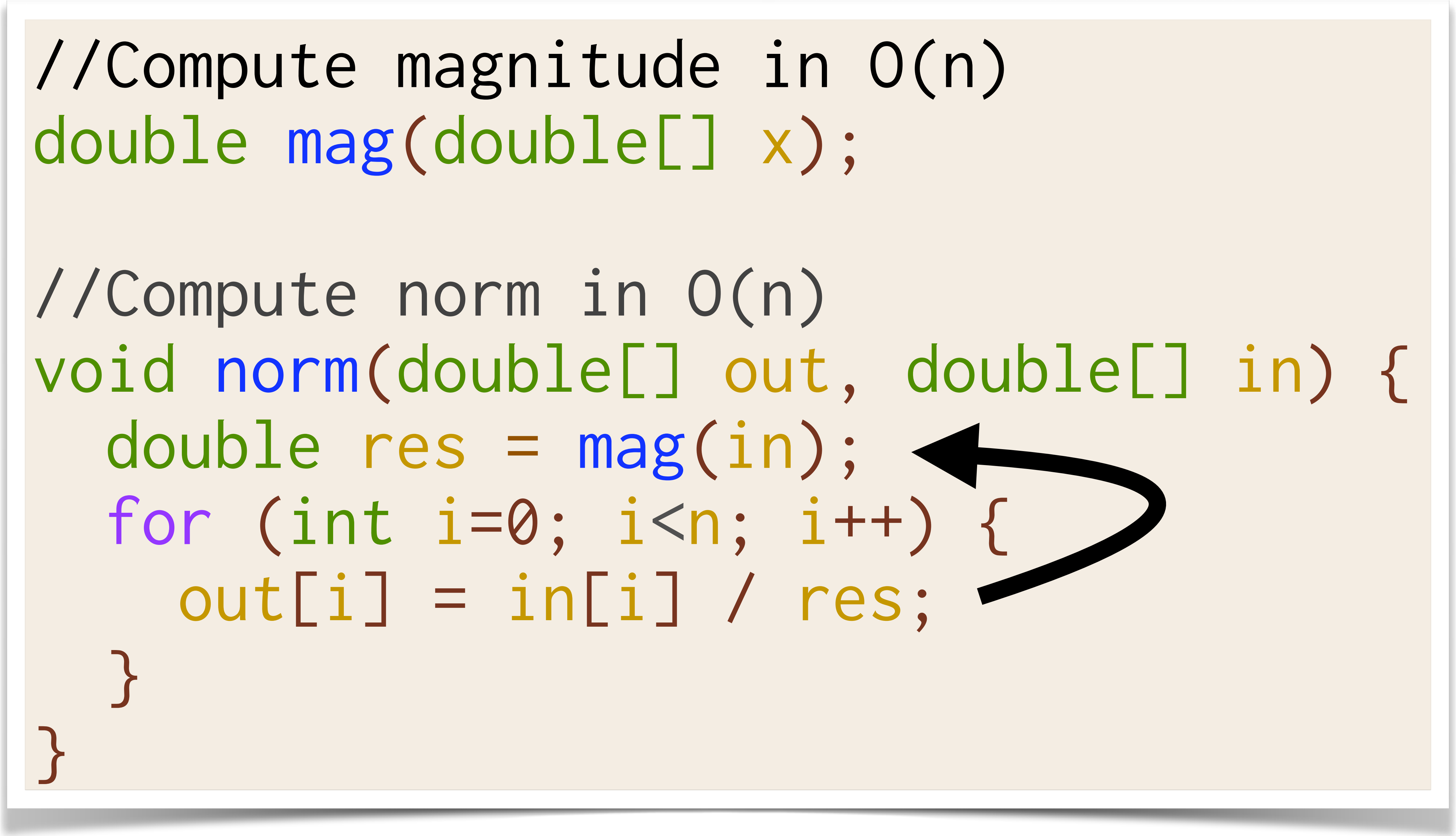


Case Study: Vector Normalization



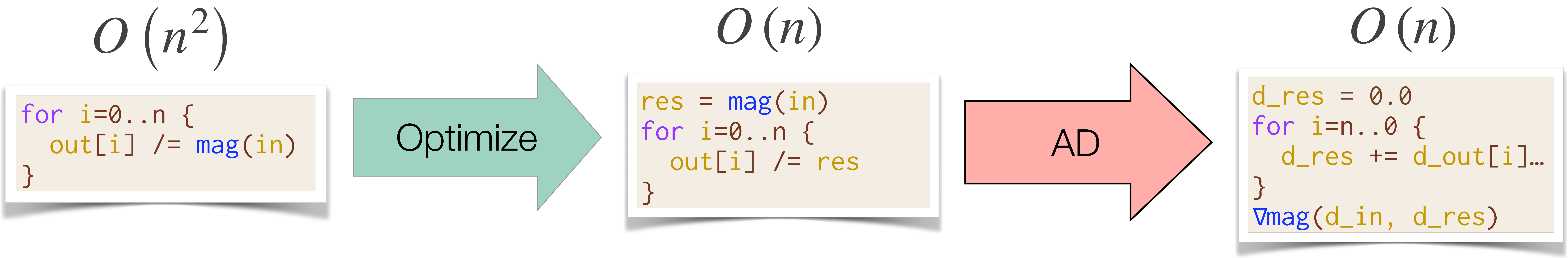
```
//Compute magnitude in O(n)
double mag(double[] x);

//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)
}
```

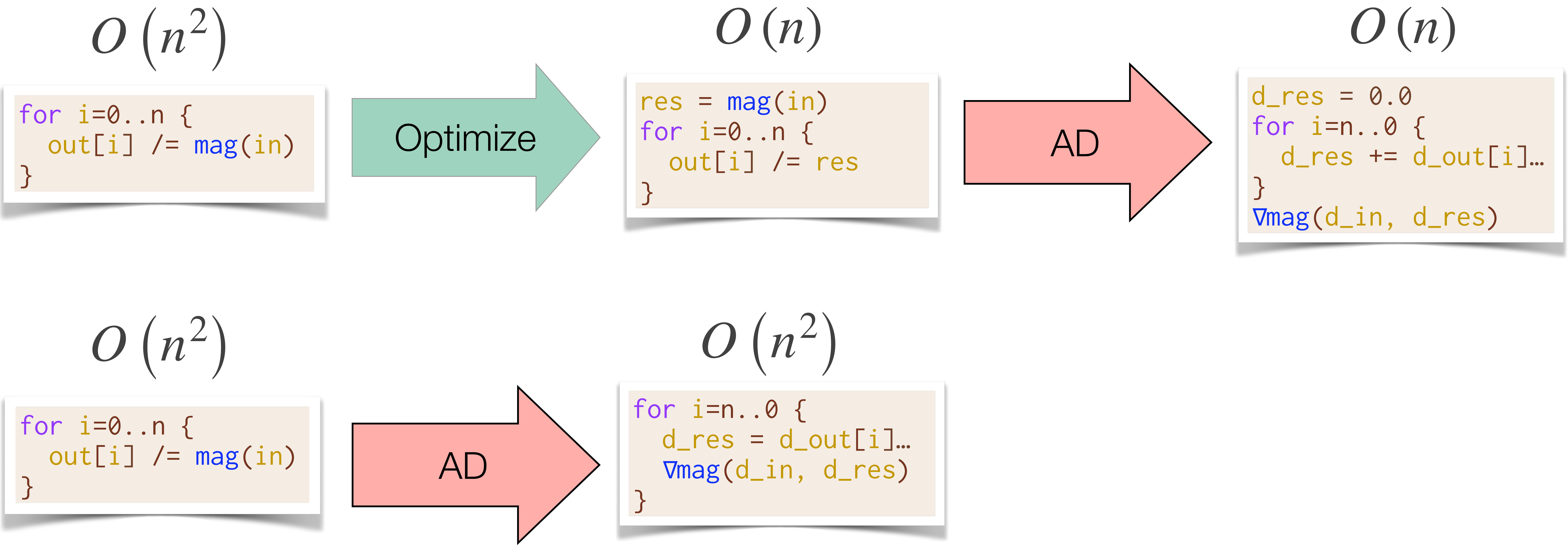
$O\left(n\right)$

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

$O\left(n\right)$

```
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)
}
```

$O\left(n\right)$

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

$O\left(n\right)$

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

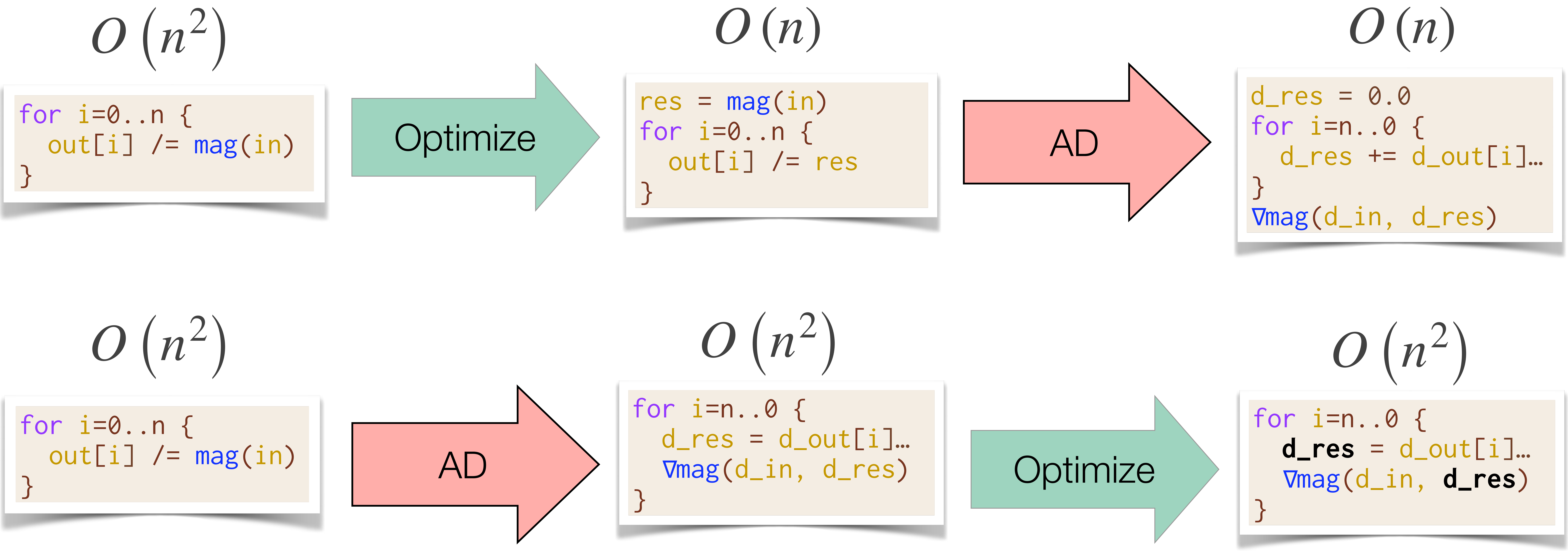
$O\left(n^2\right)$

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

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

```
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)
}
```

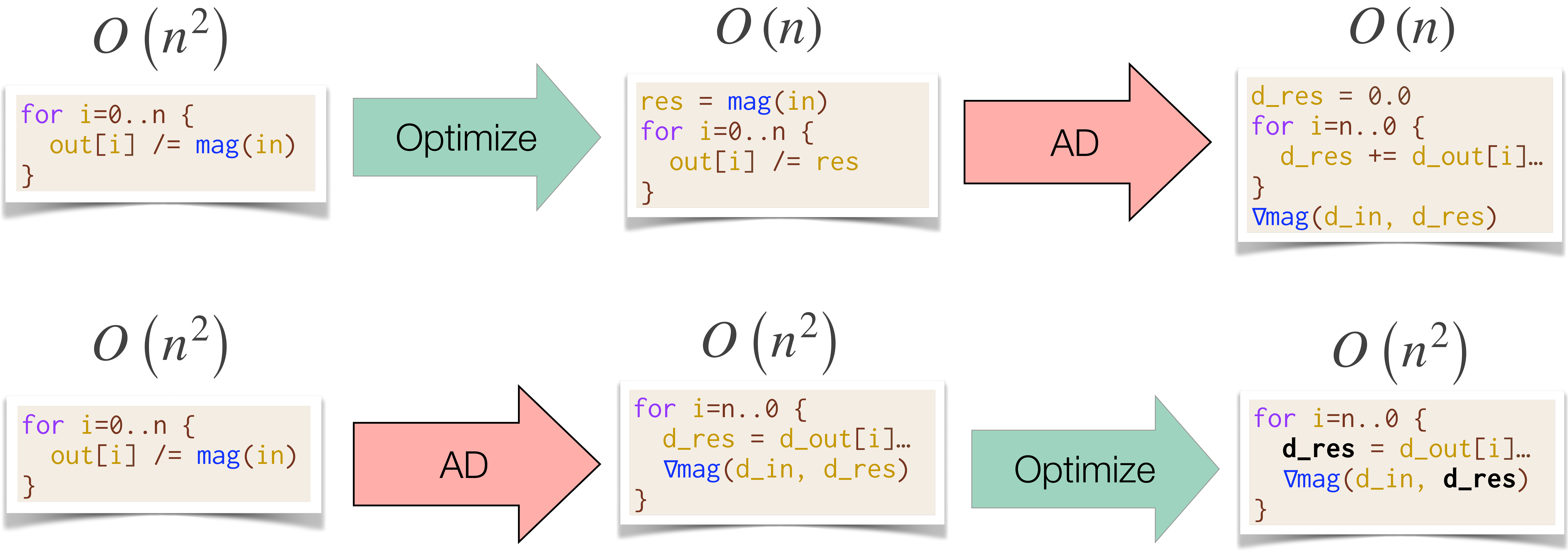
Optimize

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

```
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)
}
```

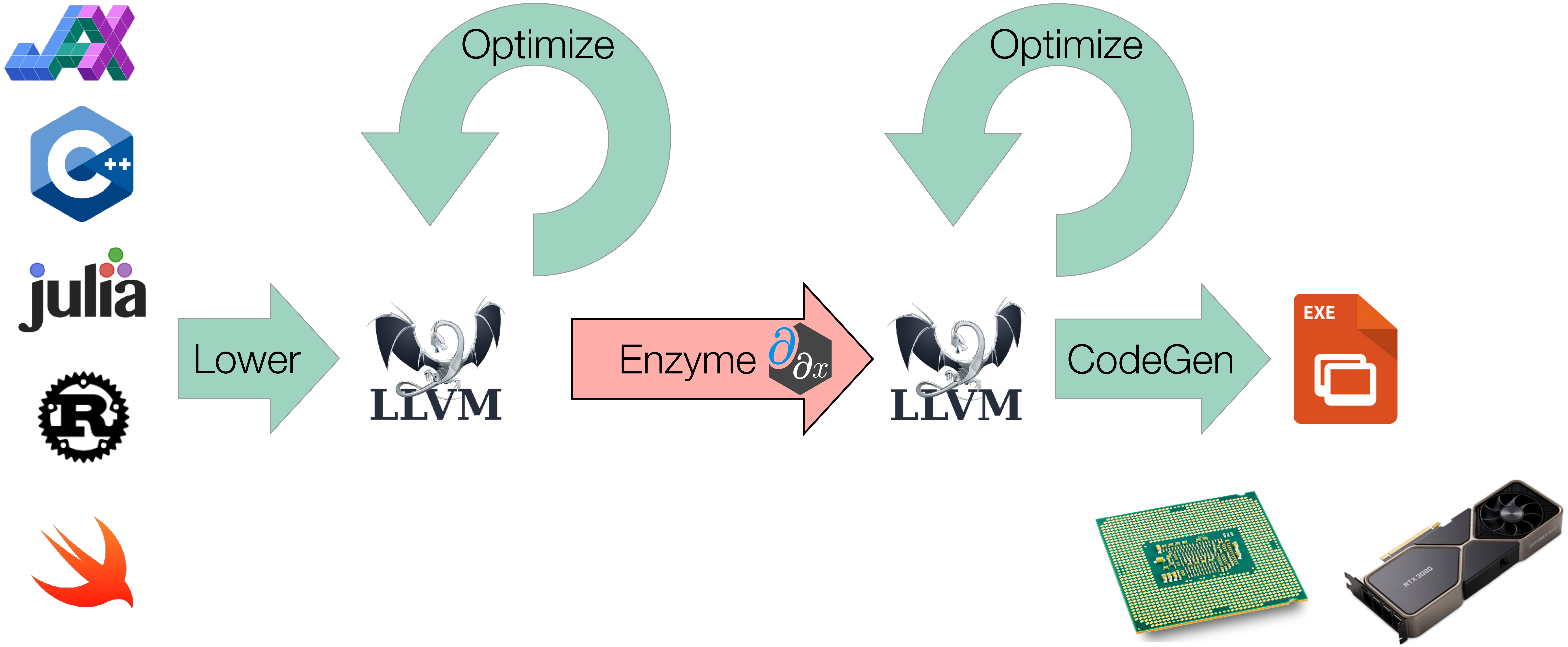
Optimize

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

```
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!



Automatic Differentiation & GPUs [MCPHNSD @ SC’21]

- Prior work has not explored reverse mode AD of existing GPU kernels

  1. Reversing parallel control flow can lead to incorrect results

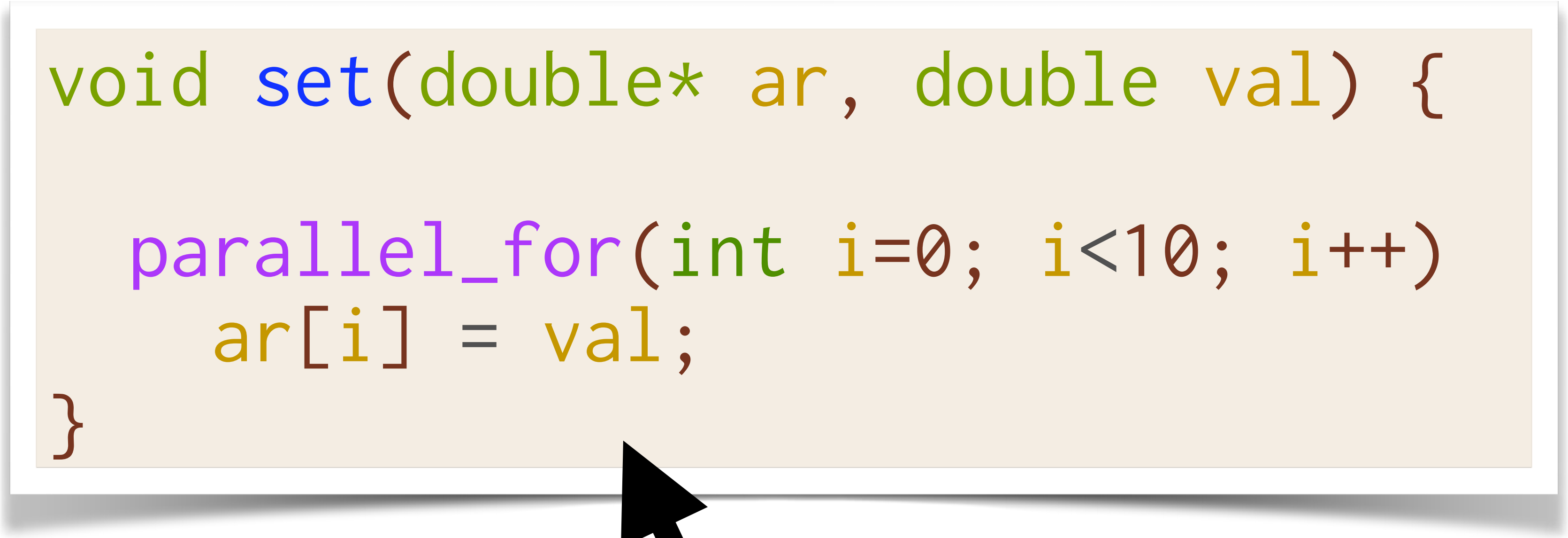
  2. Complex performance characteristics make it difficult to synthesize efficient code

  3. Resource limitations can prevent kernels from running at all

Challenges of Parallel AD

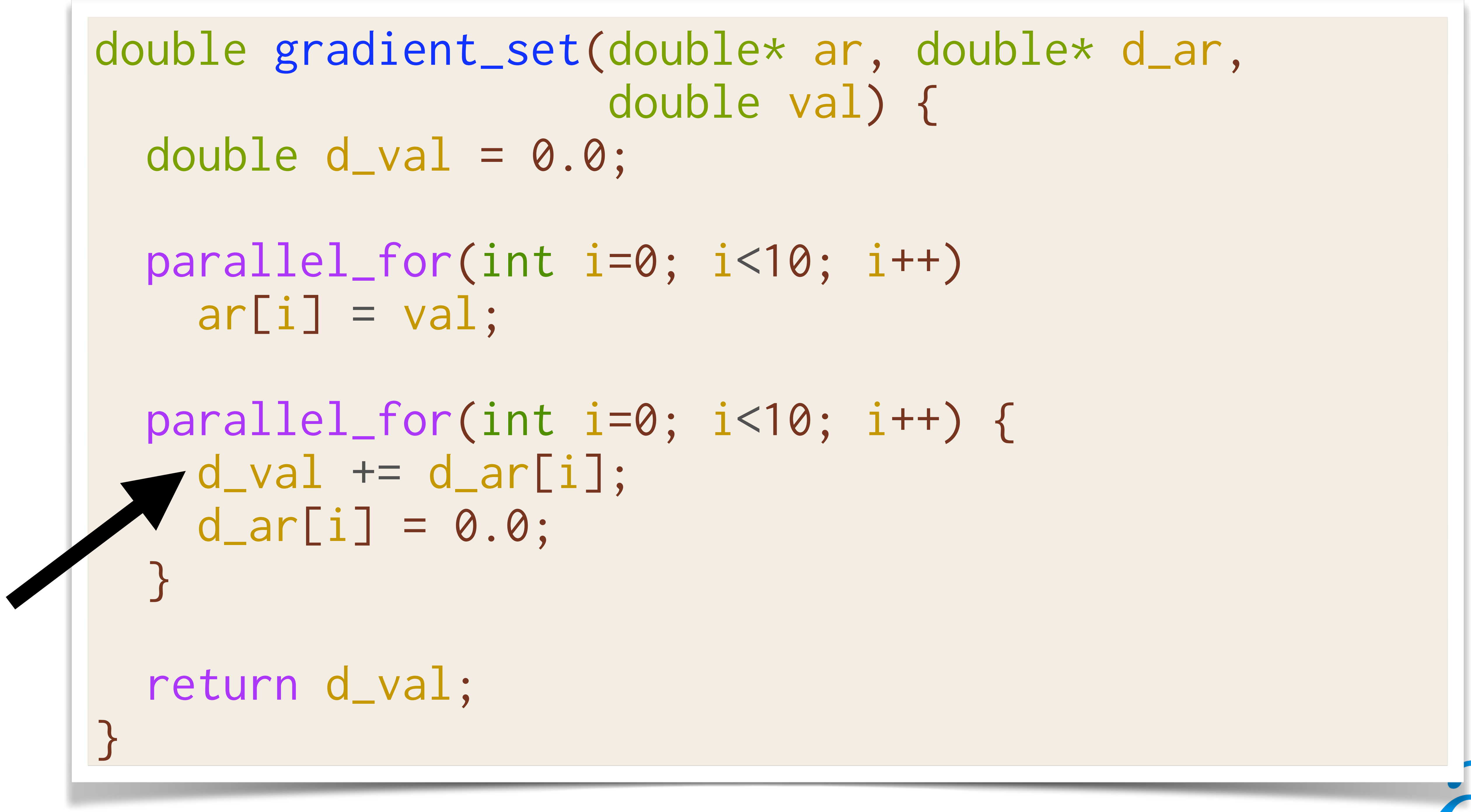
- The adjoint of an instruction increments the derivative of its input

- Benign read race in forward pass => Write race in reverse pass (undefined behavior)



```
void set(double* ar, double val) {

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

Read Race



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

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

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

  return d_val;
}
```

Write Race

GPU Memory Hierarchy

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

Slower, larger amount of memory

Correct and Efficient Derivative Accumulation

Thread-local memory

- Non-atomic load/store

```
__device__
void f(…) {

  // Thread-local var
  double y;


  …


  d_y += val;
}
```

Same memory location across all threads (some shared mem)

- Parallel Reduction

```
// Same var for all threads
double y;

__device__
void f(…) {


  …


  reduce_add(&d_y, val);
}
```

Others [always legal fallback]

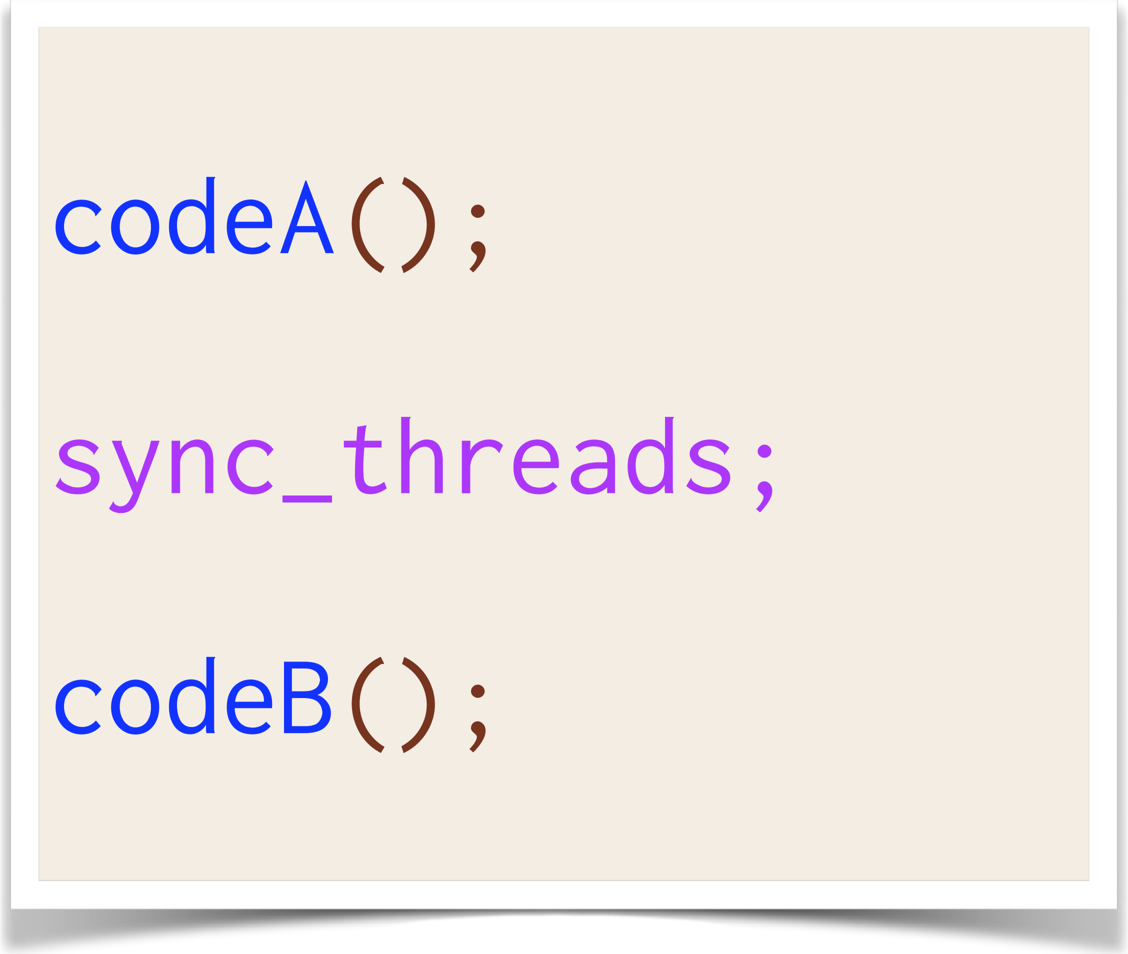
- Atomic increment

```
__device__
// Unknown thread-aliasing
void f(double* y) {


  …


  atomic { d_y += val; }
}
```

Slower

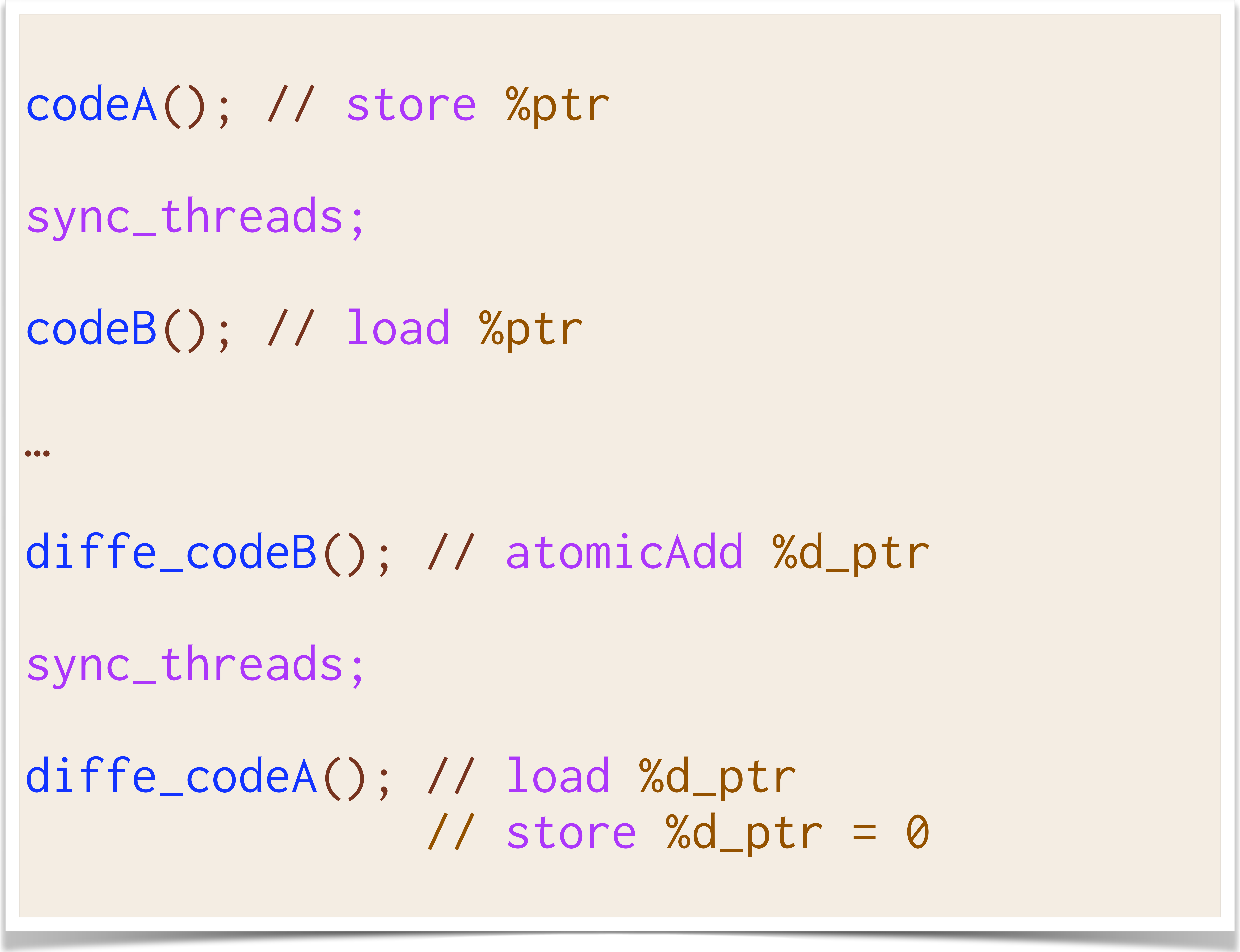
Synchronization Primitives

- Synchronization (`sync_threads`) ensures all threads finish executing codeA before executing codeB
- Sync is only necessary if A and B may access to the same memory
- Assuming the original program is race-free, performing a sync at the corresponding location in the reverse ensures correctness
- Prove correctness of algorithm by cases



```
codeA();

sync_threads;

codeB();
```

Case 1: Store, Sync, Load

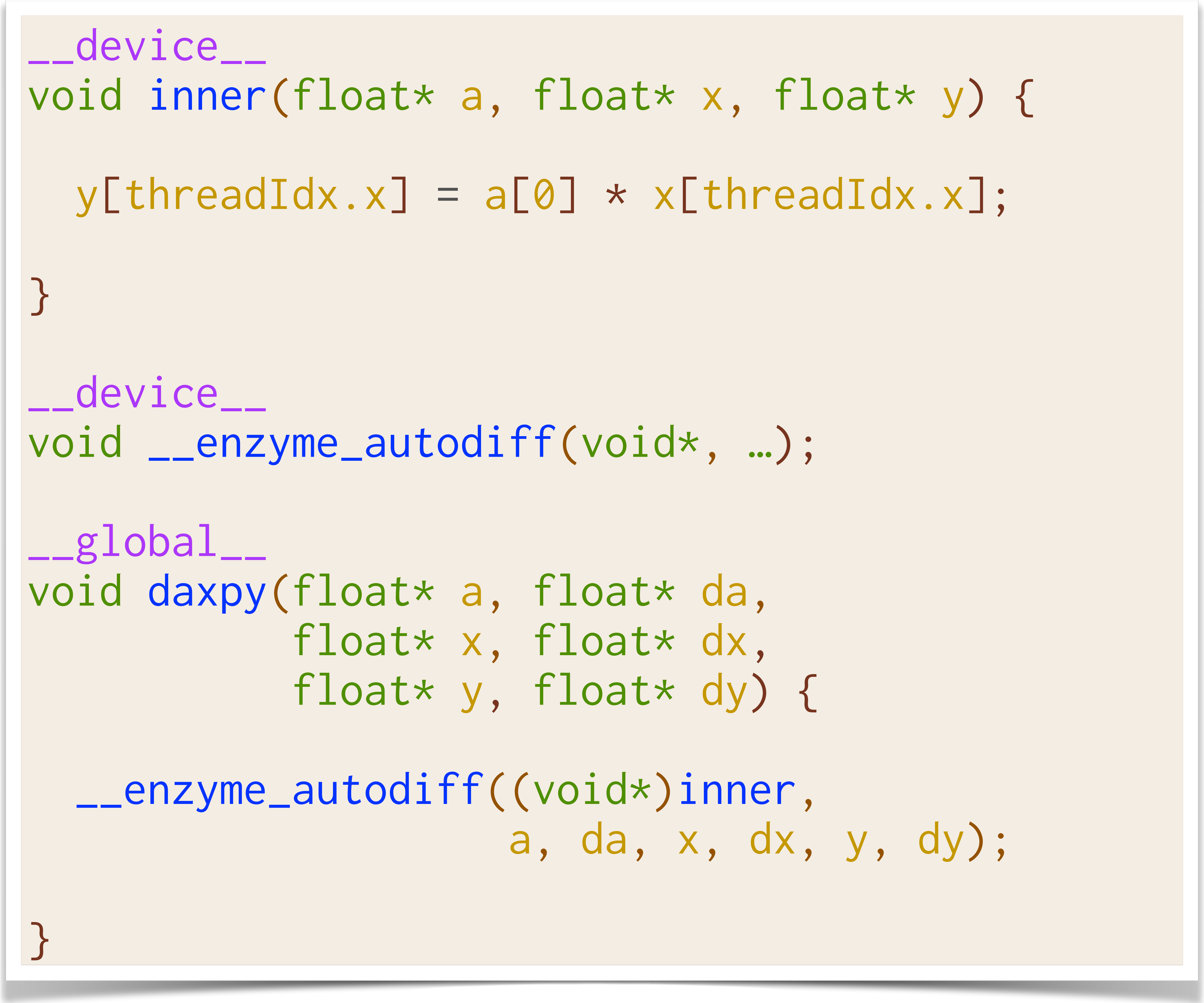


```
codeA(); // store %ptr

sync_threads;

codeB(); // load %ptr

…

diffe_codeB(); // atomicAdd %d_ptr

sync_threads;

diffe_codeA(); // load %d_ptr
               // store %d_ptr = 0
```

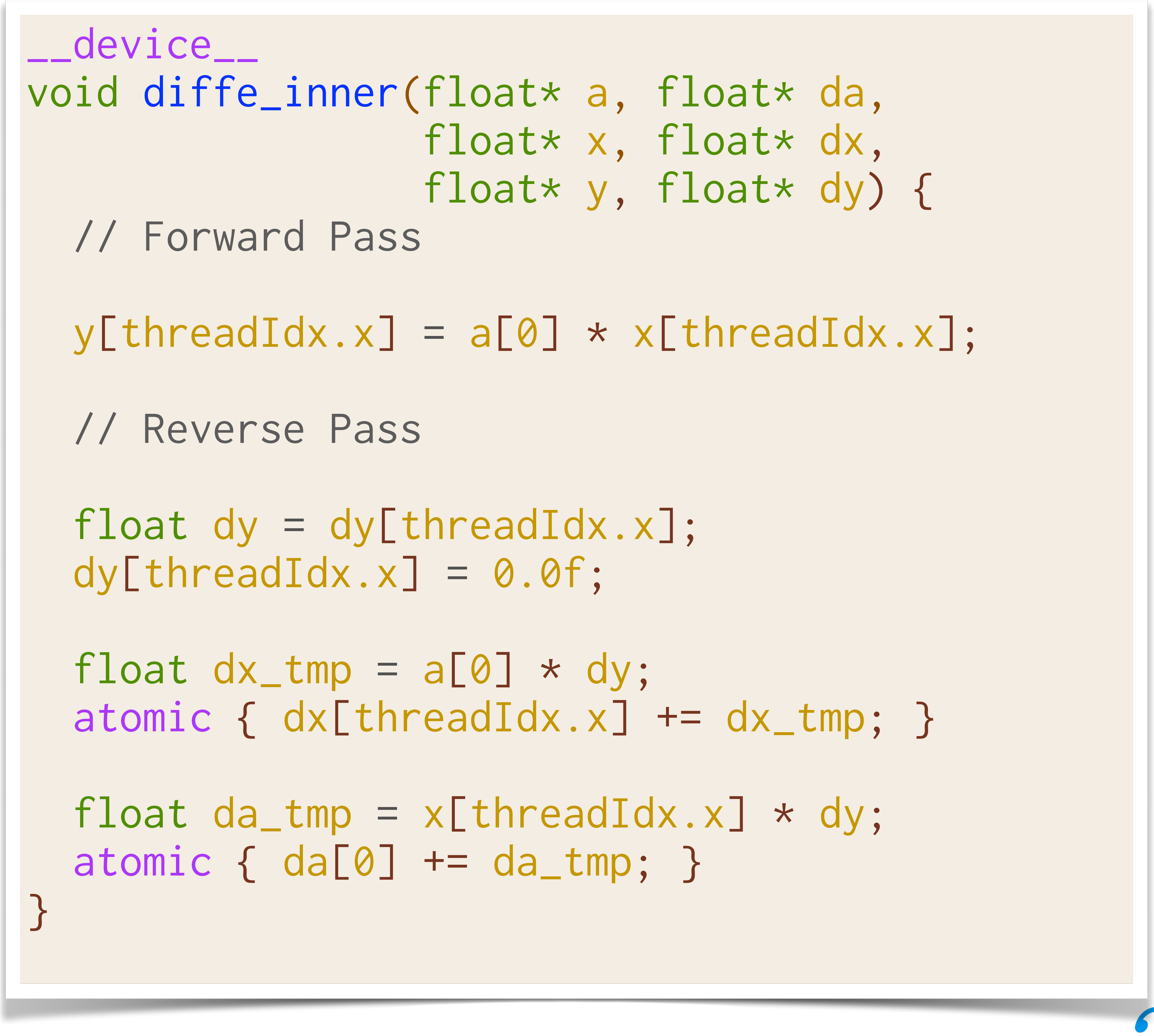
✓ Correct

- Load of `d_ptr` must happen after all atomicAdds have completed

CUDA Example

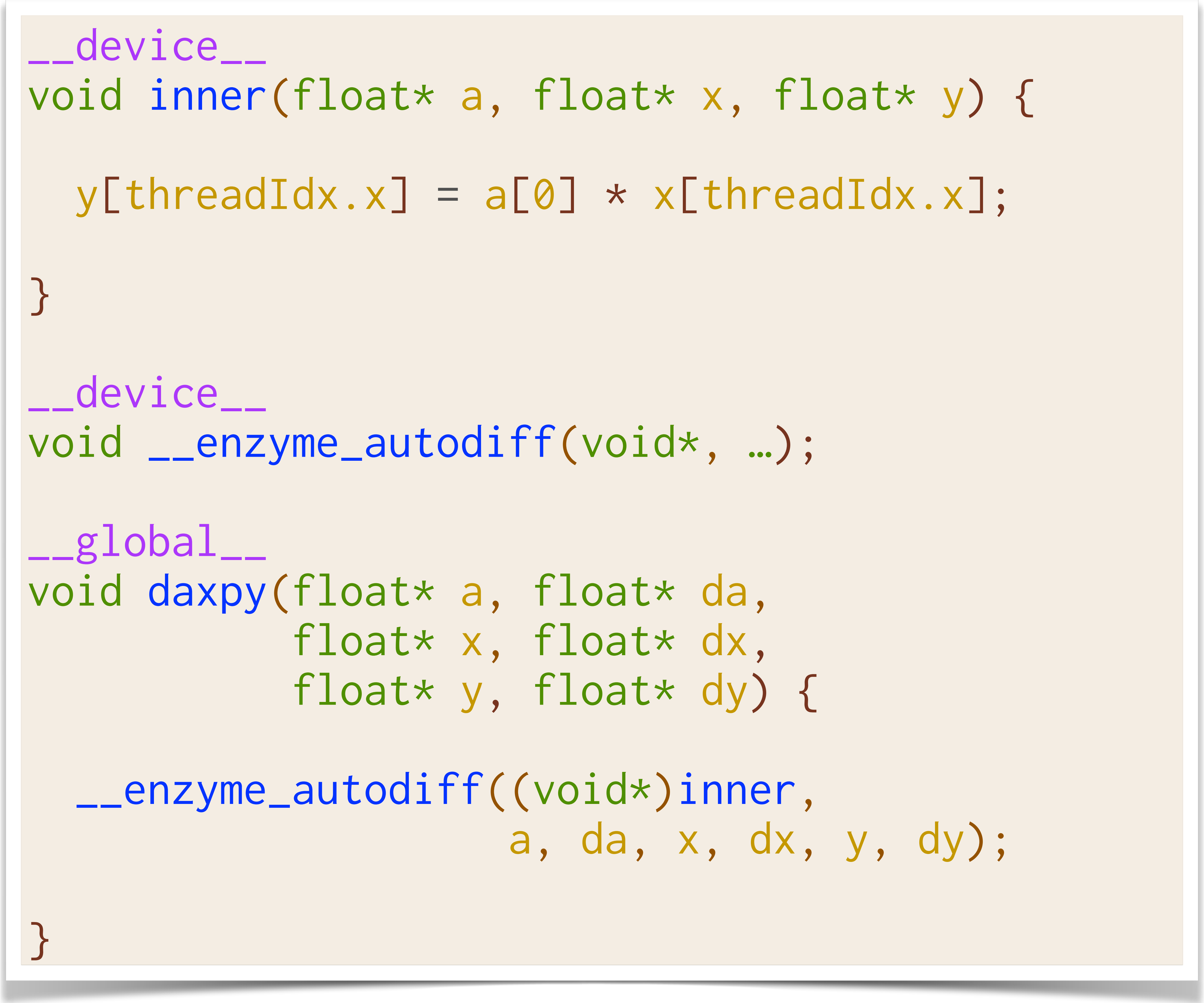


```
__device__
void inner(float* a, float* x, float* y) {

  y[threadIdx.x] = a[0] * x[threadIdx.x];

}

__device__
void __enzyme_autodiff(void*, …);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {

  __enzyme_autodiff((void*)inner,
                    a, da, x, dx, y, dy);

}
```

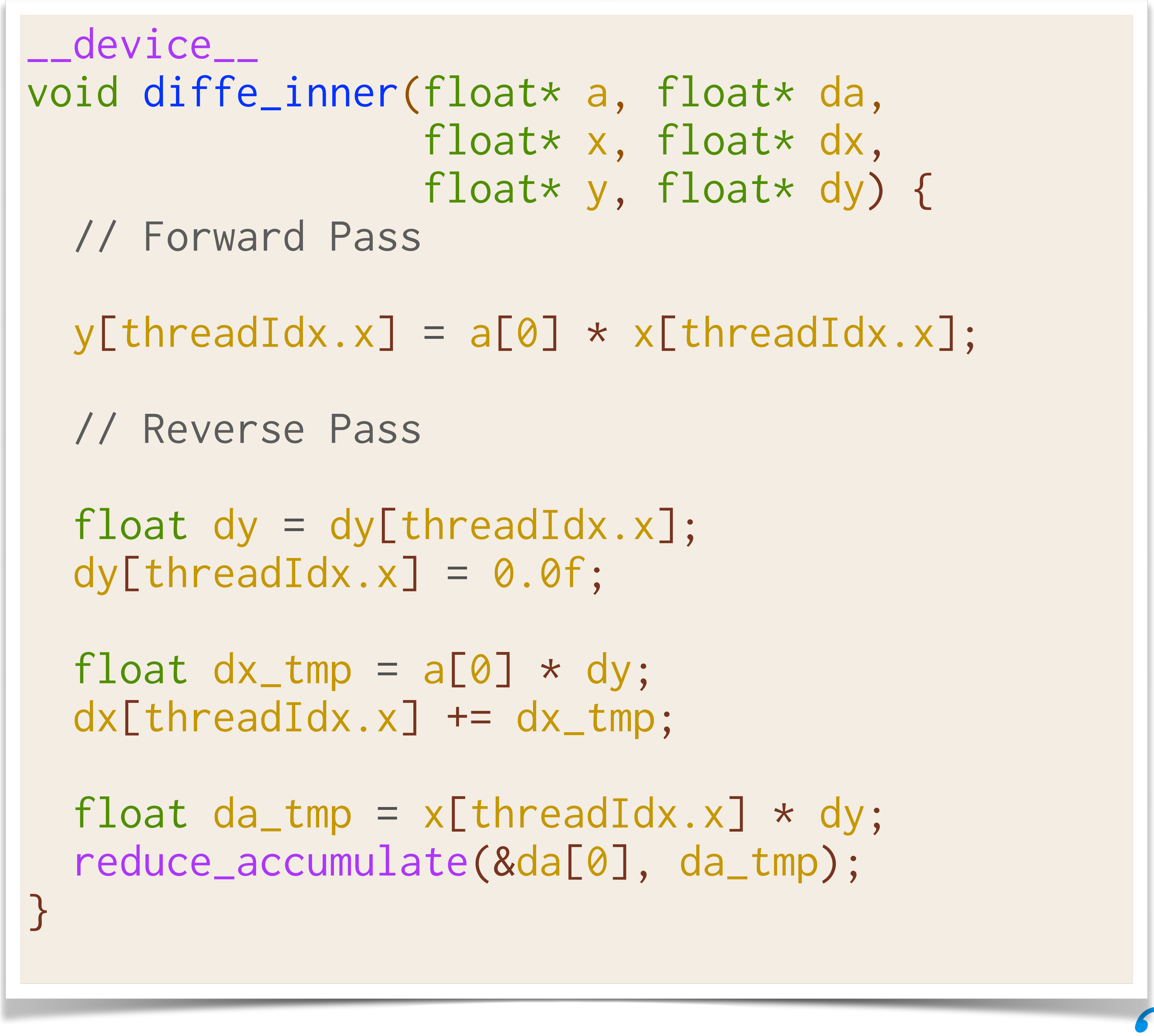


```
__device__
void diffe_inner(float* a, float* da,
                 float* x, float* dx,
                 float* y, float* dy) {
  // Forward Pass

  y[threadIdx.x] = a[0] * x[threadIdx.x];

  // Reverse Pass

  float dy = dy[threadIdx.x];
  dy[threadIdx.x] = 0.0f;

  float dx_tmp = a[0] * dy;
  atomic { dx[threadIdx.x] += dx_tmp; }

  float da_tmp = x[threadIdx.x] * dy;
  atomic { da[0] += da_tmp; }
}
```

CUDA Example

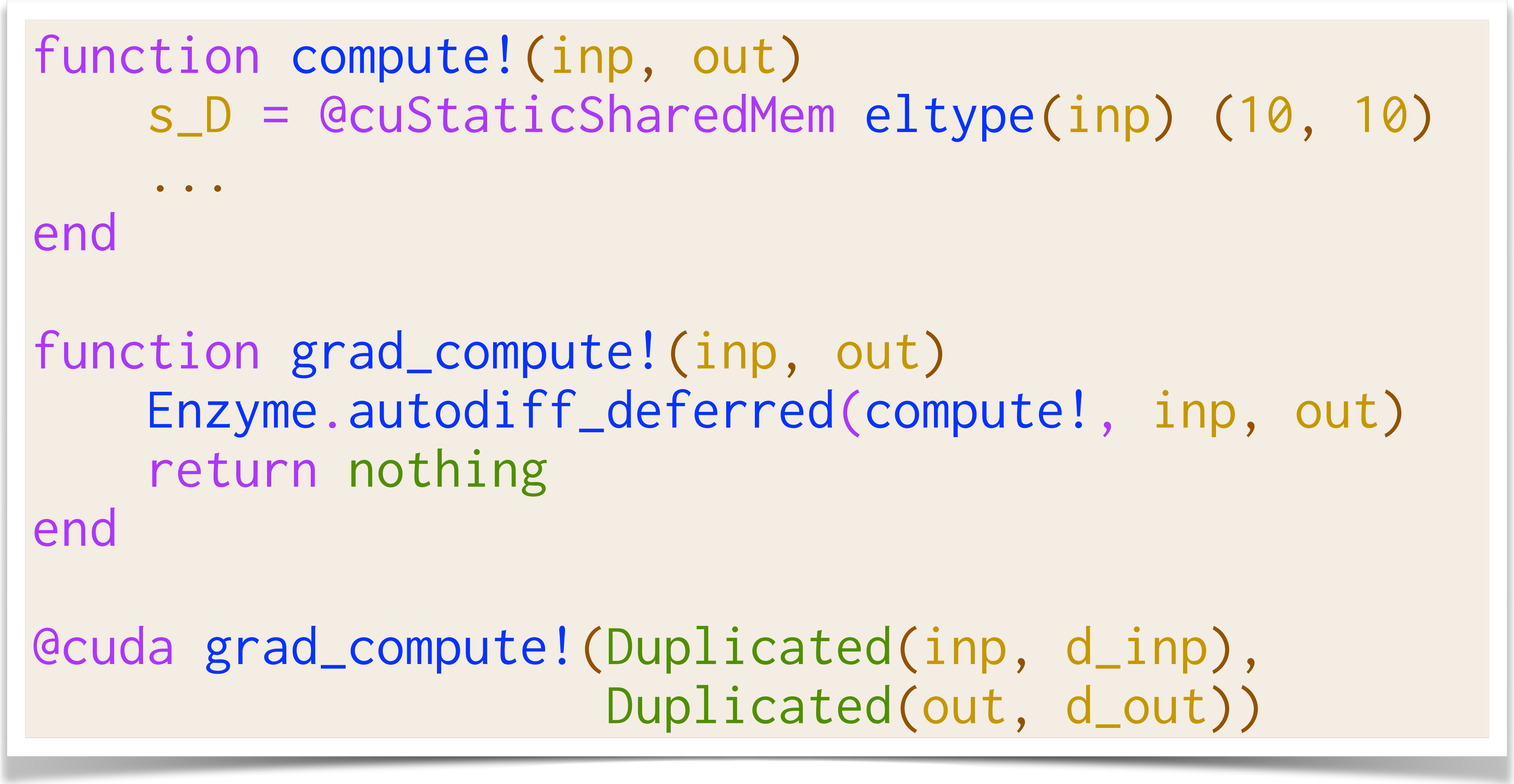


```
__device__
void inner(float* a, float* x, float* y) {

  y[threadIdx.x] = a[0] * x[threadIdx.x];

}

__device__
void __enzyme_autodiff(void*, …);

__global__
void daxpy(float* a, float* da,
           float* x, float* dx,
           float* y, float* dy) {

  __enzyme_autodiff((void*)inner,
                    a, da, x, dx, y, dy);

}
```

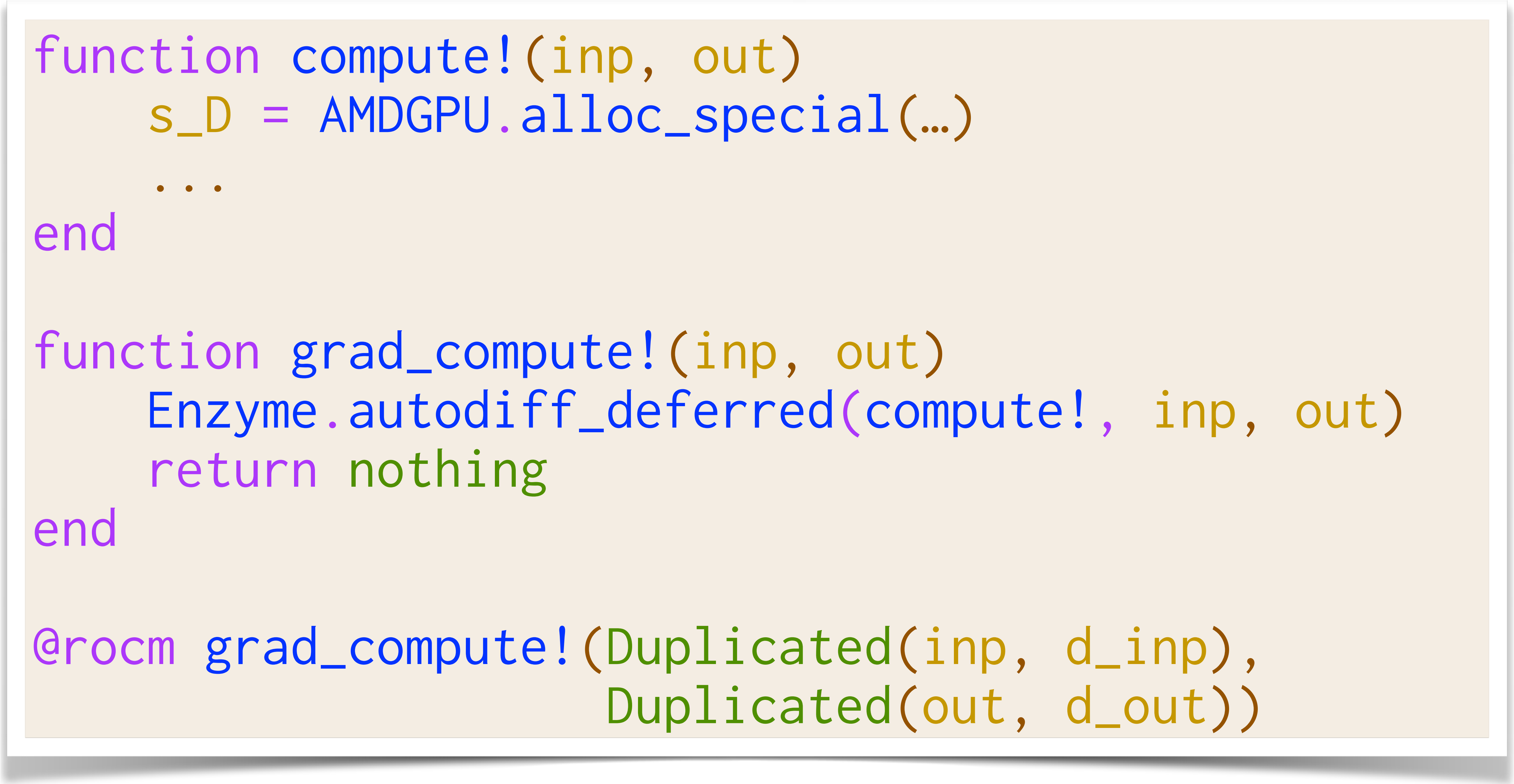


```
__device__
void diffe_inner(float* a, float* da,
                 float* x, float* dx,
                 float* y, float* dy) {
  // Forward Pass

  y[threadIdx.x] = a[0] * x[threadIdx.x];

  // Reverse Pass

  float dy = dy[threadIdx.x];
  dy[threadIdx.x] = 0.0f;

  float dx_tmp = a[0] * dy;
  dx[threadIdx.x] += dx_tmp;

  float da_tmp = x[threadIdx.x] * dy;
  reduce_accumulate(&da[0], da_tmp);
}
```

CUDA.jl / AMDGPU.jl Example



```julia
function compute!(inp, out)
    s_D = @cuStaticSharedMem eltype(inp) (10, 10)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@cuda grad_compute!(Duplicated(inp, d_inp),
                    Duplicated(out, d_out))
```



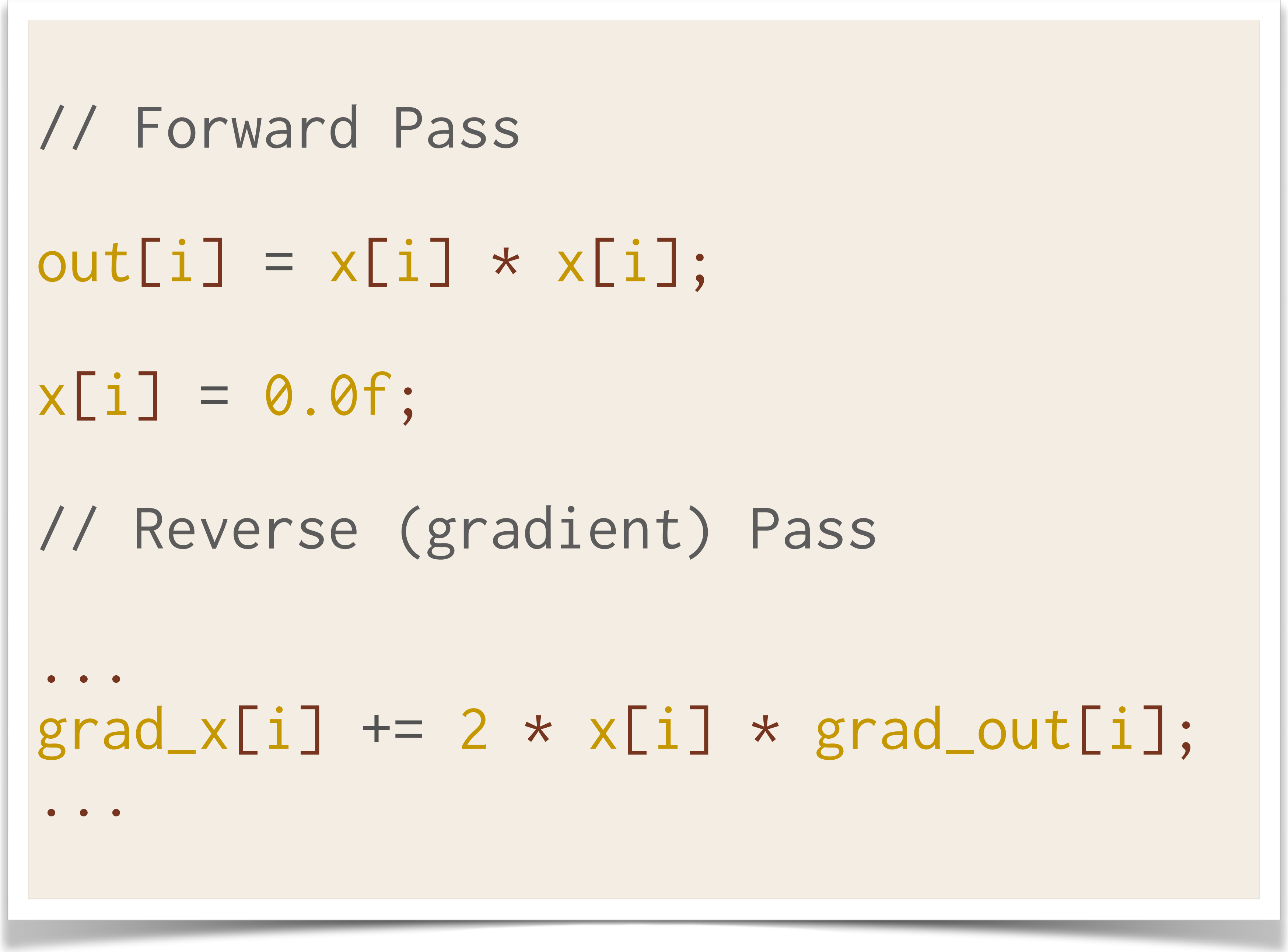
```julia
function compute!(inp, out)
    s_D = AMDGPU.alloc_special(…)
    ...
end

function grad_compute!(inp, out)
    Enzyme.autodiff_deferred(compute!, inp, out)
    return nothing
end

@rocm grad_compute!(Duplicated(inp, d_inp),
                    Duplicated(out, d_out))
```

See Below For Full Code Examples

https://github.com/wsmoses/Enzyme-GPU-Tests/blob/main/DG/

Efficient GPU Code

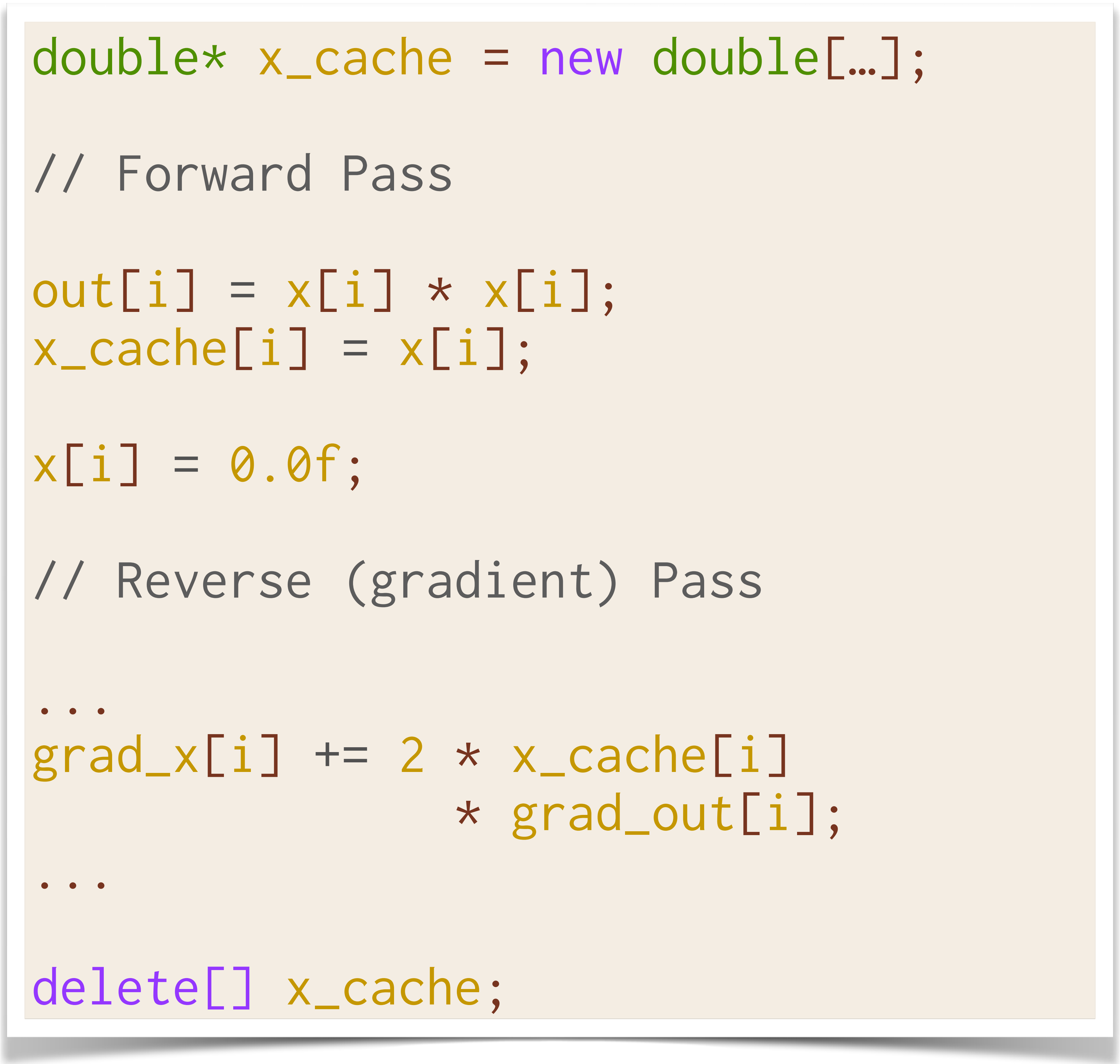
- For correctness, Enzyme may need to cache values in order to compute the gradient
    - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren’t sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude



```
// Forward Pass

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

x[i] = 0.0f;

// Reverse (gradient) Pass

...
grad_x[i] += 2 * x[i] * grad_out[i];
...
```

~~Efficient~~ Correct GPU Code

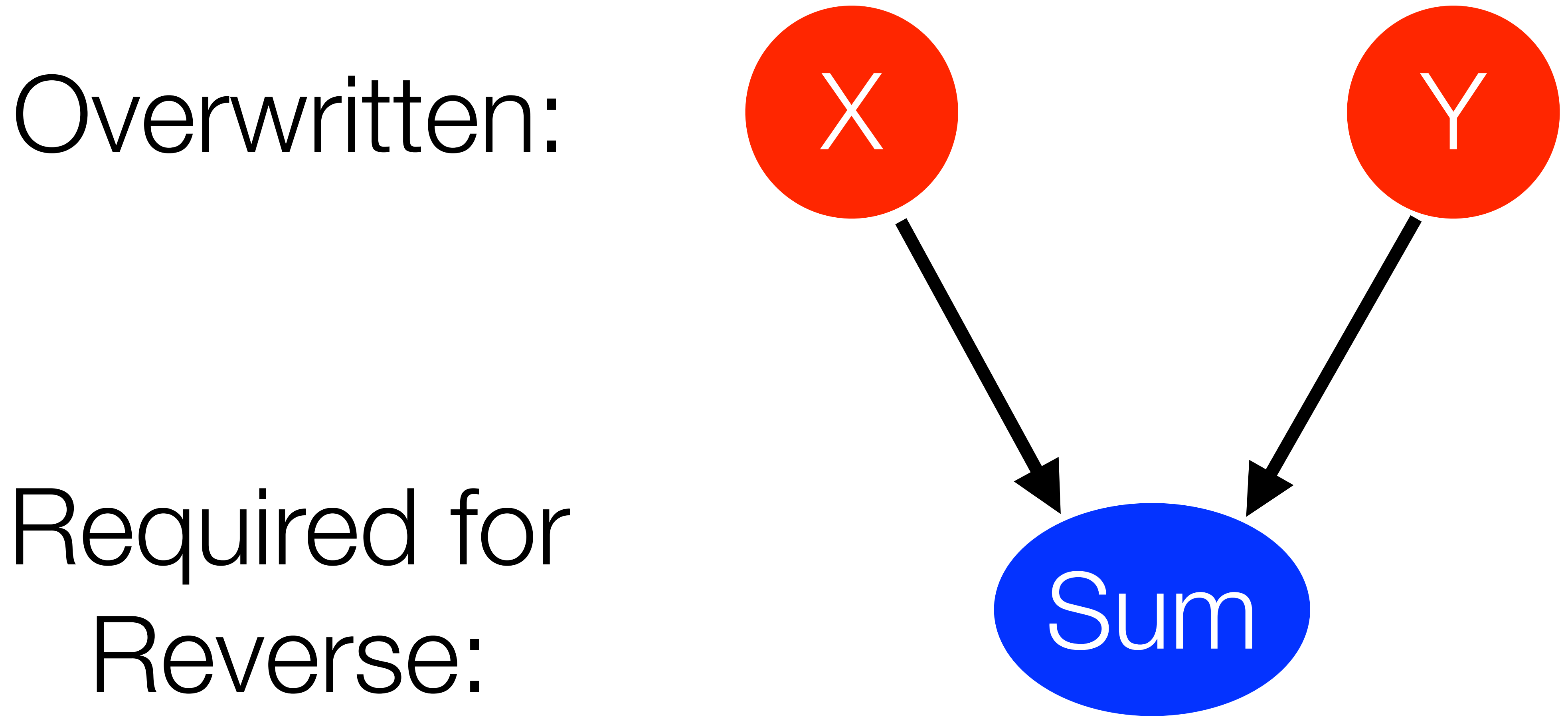
- For correctness, Enzyme may need to cache values in order to compute the gradient
    - The complexity of GPU memory means large caches slow down the program by several orders of magnitude, if it even fits at all
- Like the CPU, existing optimizations reduce the overhead
- Unlike the CPU, existing optimizations aren't sufficient
- Novel GPU and AD-specific optimizations can speedup by several orders of magnitude



```
double* x_cache = new double[…];

// Forward Pass

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

x[i] = 0.0f;

// Reverse (gradient) Pass

...
grad_x[i] += 2 * x_cache[i]
               * grad_out[i];
...

delete[] x_cache;
```

Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.



Overwritten:

Required for Reverse:

X

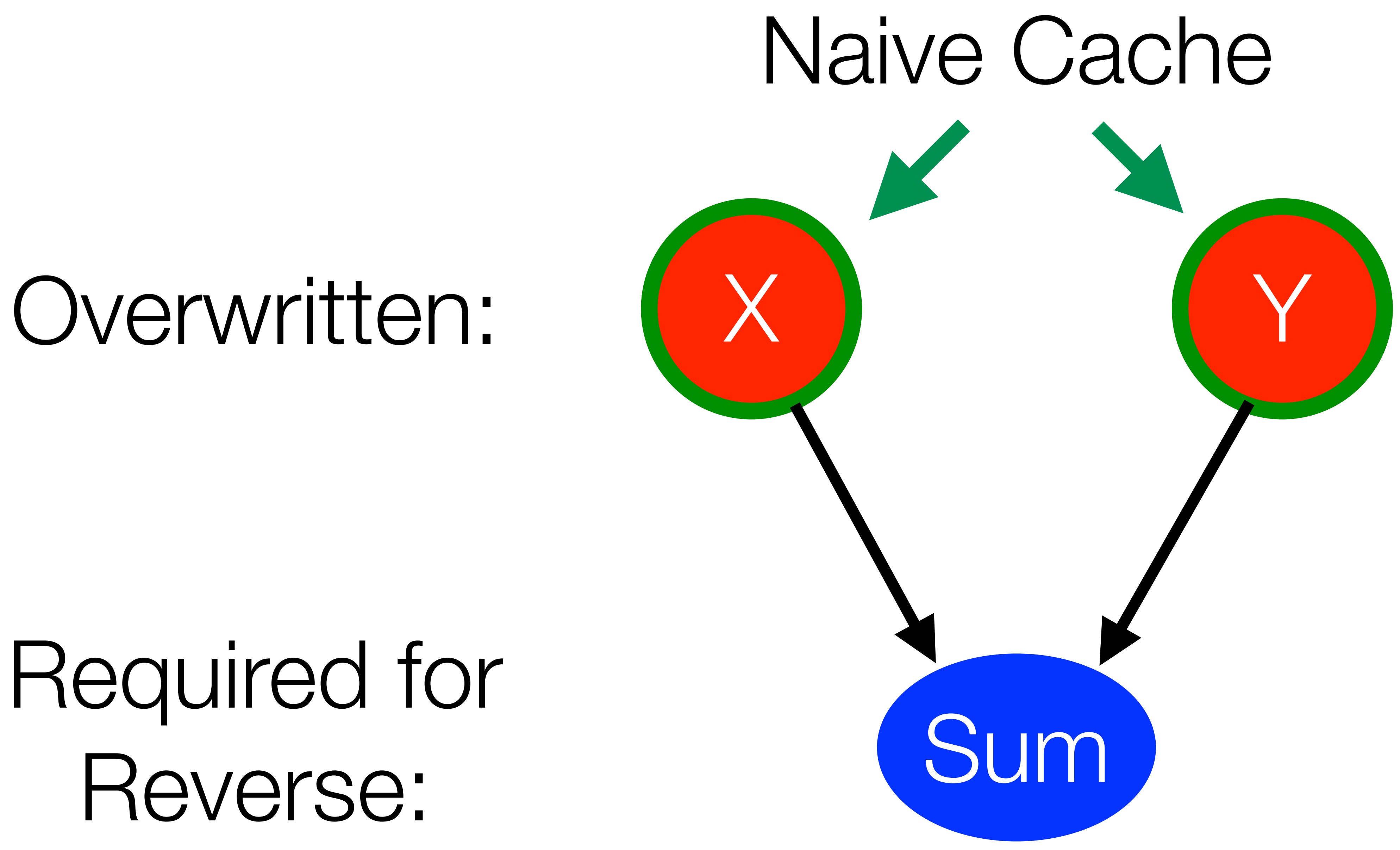
Y

Sum

```
for(int i=0; i<10; i++) {
  double sum = x[i] + y[i];


  use(sum);
}

overwrite(x, y);
grad_overwrite(x, y);

for(int i=9; i>=0; i--) {
  ...
  grad_use(sum);
}
```

Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.



Naive Cache

Overwritten:

X

Y

Required for Reverse:

Sum

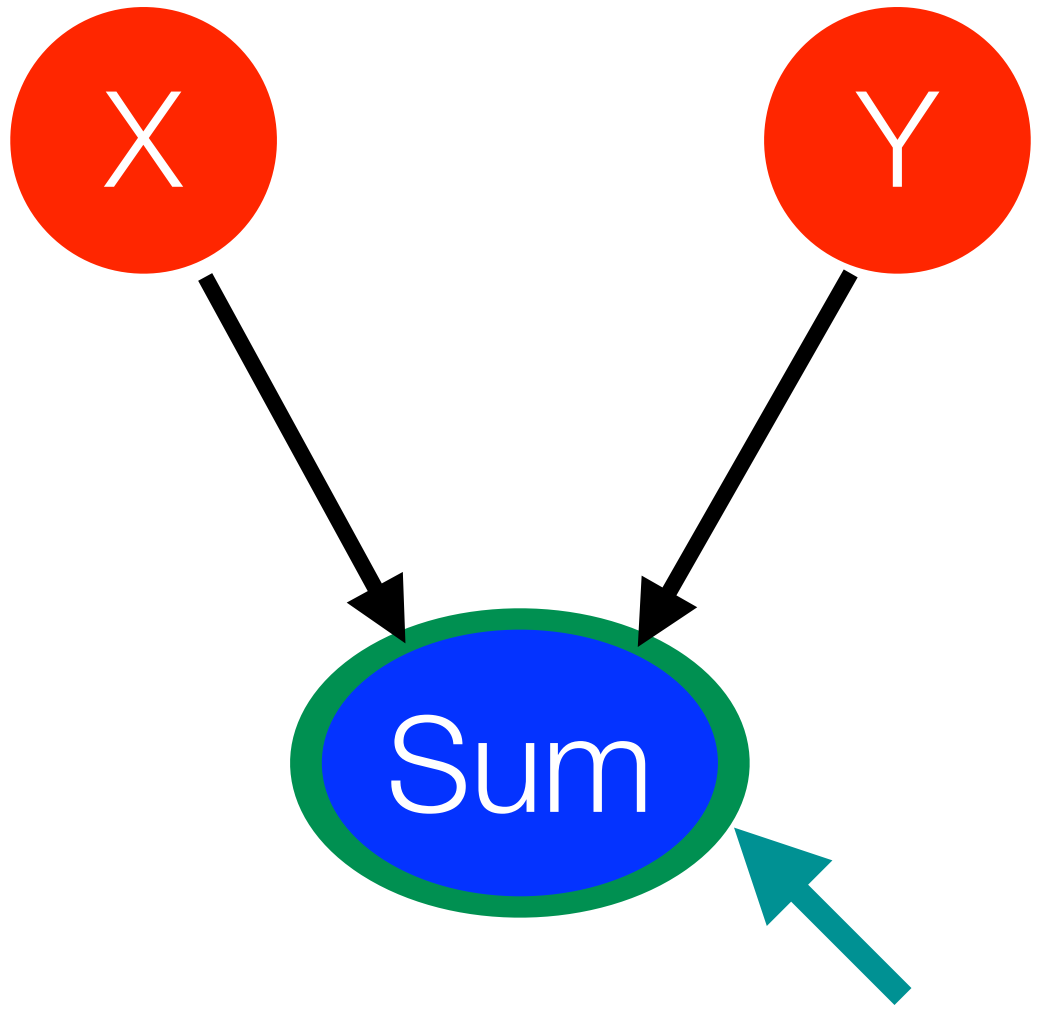
```
double* x_cache = new double[10];
double* y_cache = new double[10];

for(int i=0; i<10; i++) {
  double sum = x[i] + y[i];
  x_cache[i] = x[i];
  y_cache[i] = y[i];
  use(sum);
}

overwrite(x, y);
grad_overwrite(x, y);

for(int i=9; i>=0; i--) {
  double sum = x_cache[i] + y_cache[i];
  grad_use(sum);
}
```

Cache Reduction Example

- By considering the dataflow graph we can perform a min-cut to approximate smaller cache sizes.

Overwritten:

Required for Reverse:



Smallest Cache

```
double* sum_cache = new double[10];

for(int i=0; i<10; i++) {
  double sum = x[i] + y[i];
  sum_cache[i] = sum;

  use(sum);
}

overwrite(x, y);
grad_overwrite(x, y);

for(int i=9; i>=0; i--) {

  grad_use(sum_cache[i]);
}
```

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;
```

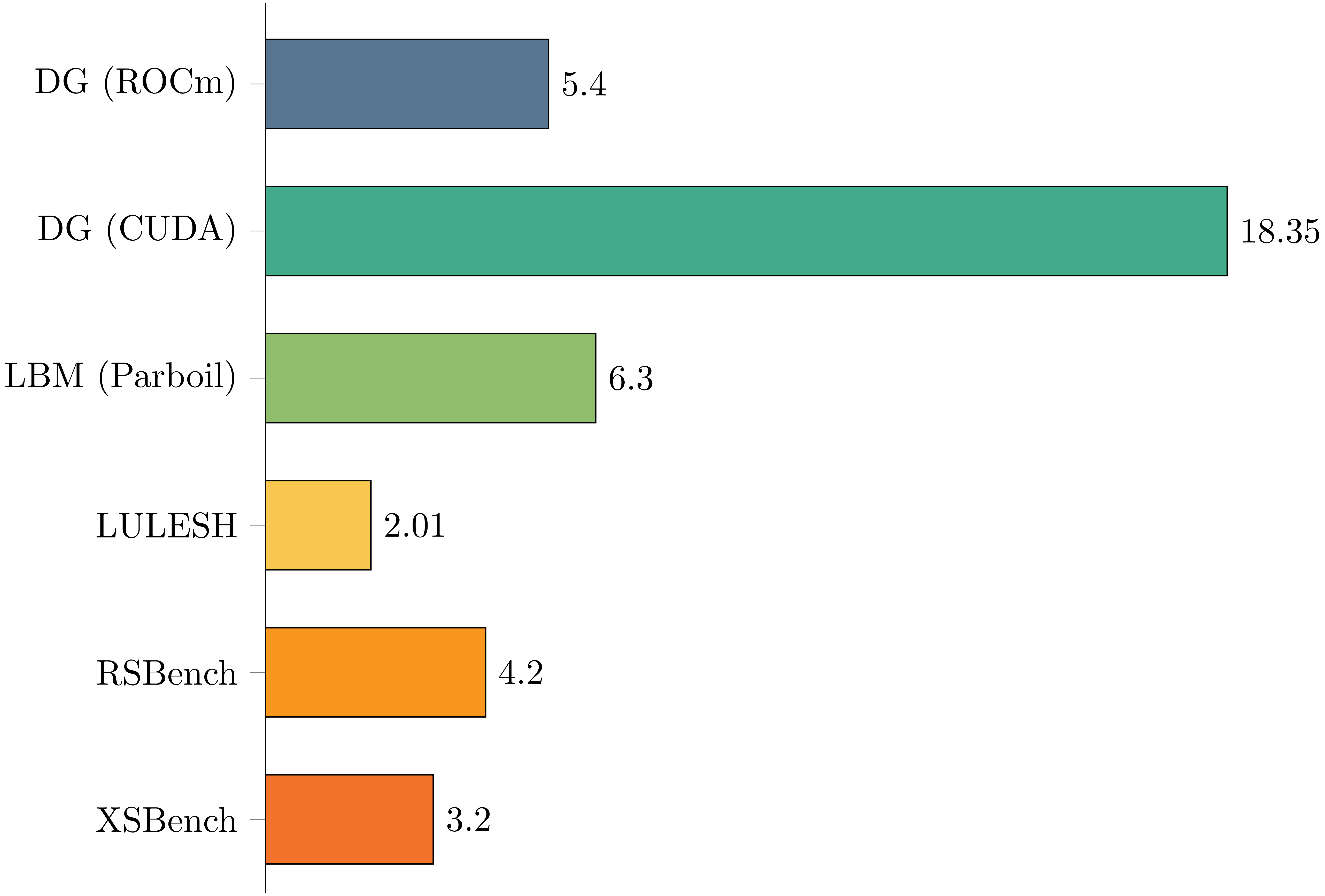
```
double* var1 = new double[N + M];
double* var2 = var1 + N;

use(var1, var2);

delete[] var1;
```

Novel AD + GPU Optimizations

- See our SC’21 paper for more (https://c.wsmoses.com/papers/EnzymeGPU.pdf)
  Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. SC, 2021
- [AD] Cache LICM/CSE
- [AD] Min-Cut Cache Reduction
- [AD] Cache Forwarding
- [GPU] Merge Allocations
- [GPU] Heap-to-stack (and register)
- [GPU] Alias Analysis Properties of SyncThreads
- …

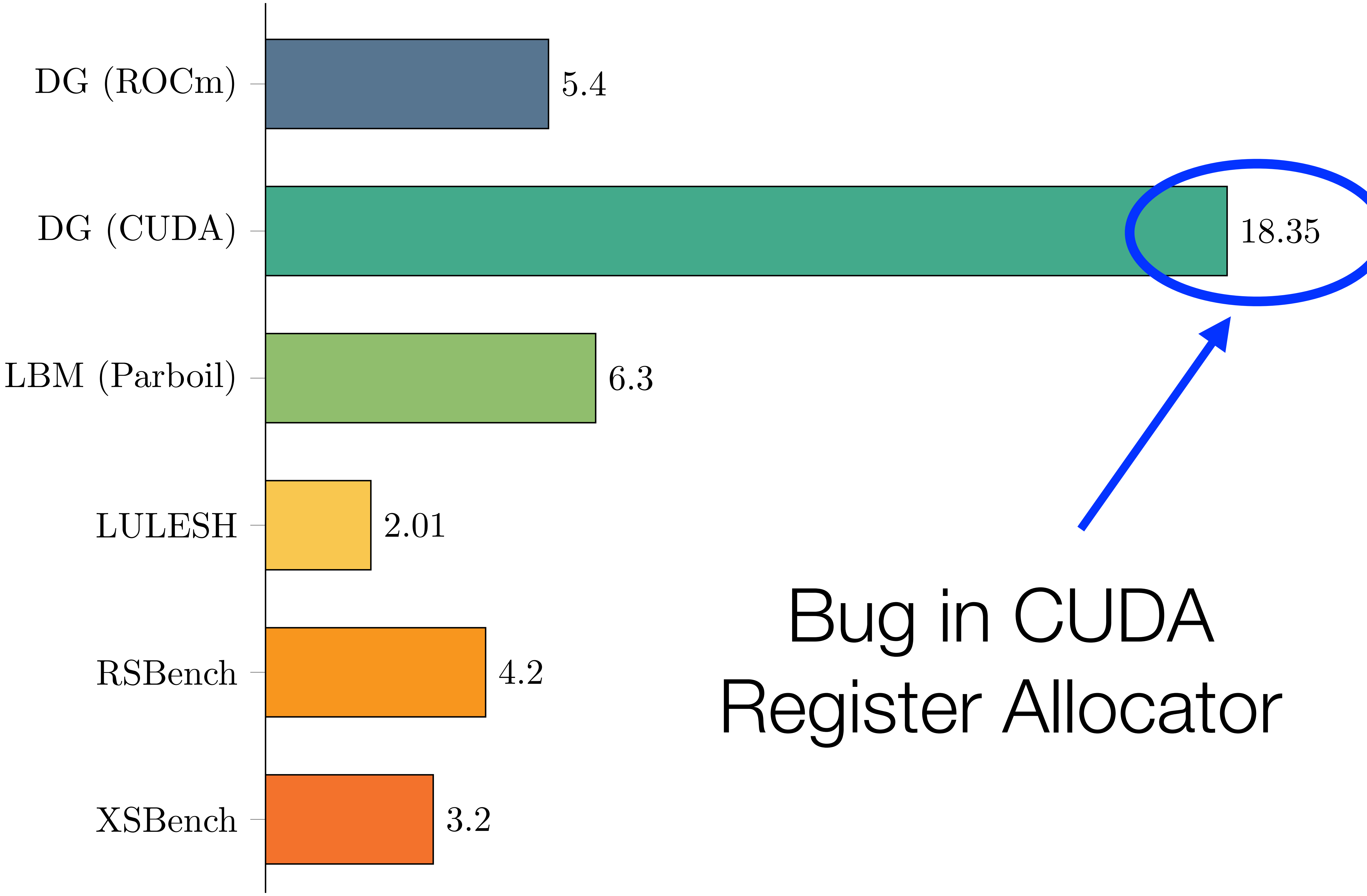
GPU Gradient Overhead [MCPHNMJ’21]

- 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)



Bug in CUDA Register Allocator

Ablation Analysis of Optimizations

DG (ROCm)
Unrolling
5.4×
DG (CUDA)
Unrolling
MallocCoalescing
PreOptimization
17.8×
116.6×
1378.3×
LBM
Allocator
Recompute
InlineCacheABI
6.4×
8.7×
19.87×
LULESH
SpecPHI
PreOptimization
2.0×
2.4×
2979.1×
RSBench
CacheLICM
Inlining
PreOpt
4.7×
9.5×
6372.2×
XSBench
Templating
PHI
LoopBound
PreOptimization
3.2×
9.5×
16.3×
25.9×
Forward (1x)
10x
100x
1000x
OOM
Overhead above Forward Pass

Ablation Analysis of Optimizations

DG (ROCm)
DG (CUDA)
LBM
LULESH
RSBench
XSBench
Unrolling
5.4×
Unrolling
MallocCoalescing
PreOptimization
17.8×
116.6×
1378.3×
Allocator
Recompute
InlineCacheABI
6.4×
8.7×
19.87×
SpecPHI
PreOptimization
2.0×
2.4×
2979.1×
CacheLICM
Inlining
PreOpt
4.7×
9.5×
6372.2×
Templating
PHI
LoopBound
PreOptimization
3.2×
9.5×
16.3×
25.9×
Forward (1x)
10x
100x
1000x
OOM
Overhead above Forward Pass

Ablation Analysis of Optimizations

DG (ROCm)
Unrolling
5.4×
DG (CUDA)
Unrolling
MallocCoalescing
PreOptimization
17.8×
116.6×
1378.3×
LBM
Allocator
Recompute
InlineCacheABI
6.4×
8.7×
19.87×
LULESH
SpecPHI
PreOptimization
2.0×
2.4×
2979.1×
RSBench
CacheLICM
Inlining
PreOpt
4.7×
9.5×
6372.2×
XSBench
Templating
PHI
LoopBound
PreOptimization
3.2×
9.5×
16.3×
25.9×
Forward (1x)
10x
100x
1000x
OOM
Overhead above Forward Pass

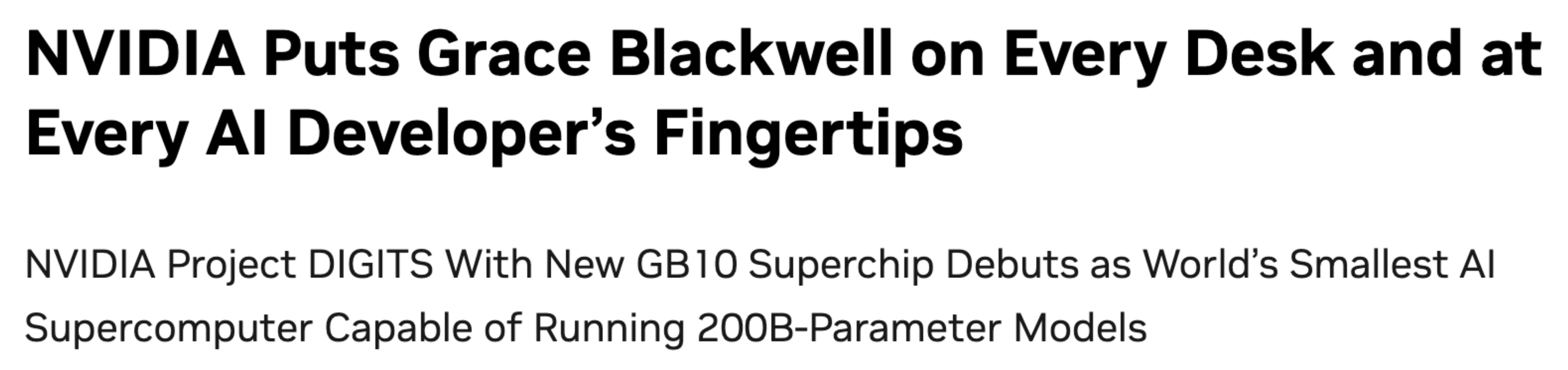
Ablation Analysis of Optimizations

DG (ROCm)
Unrolling
5.4×
DG (CUDA)
Unrolling
MallocCoalescing
PreOptimization
17.8×
116.6×
1378.3×
LBM
Allocator
Recompute
InlineCacheABI
6.4×
8.7×
19.87×
LULESH
SpecPHI
PreOptimization
2.0×
2.4×
2979.1×
RSBench
CacheLICM
Inlining
PreOpt
4.7×
9.5×
6372.2×
XSBench
Templating
PHI
LoopBound
PreOptimization
3.2×
9.5×
16.3×
25.9×
Forward (1x)
10x
100x
1000x
OOM
Overhead above Forward Pass

GPU AD is Intractable Without Optimization!

Computing Hardware is No Longer For Everybody

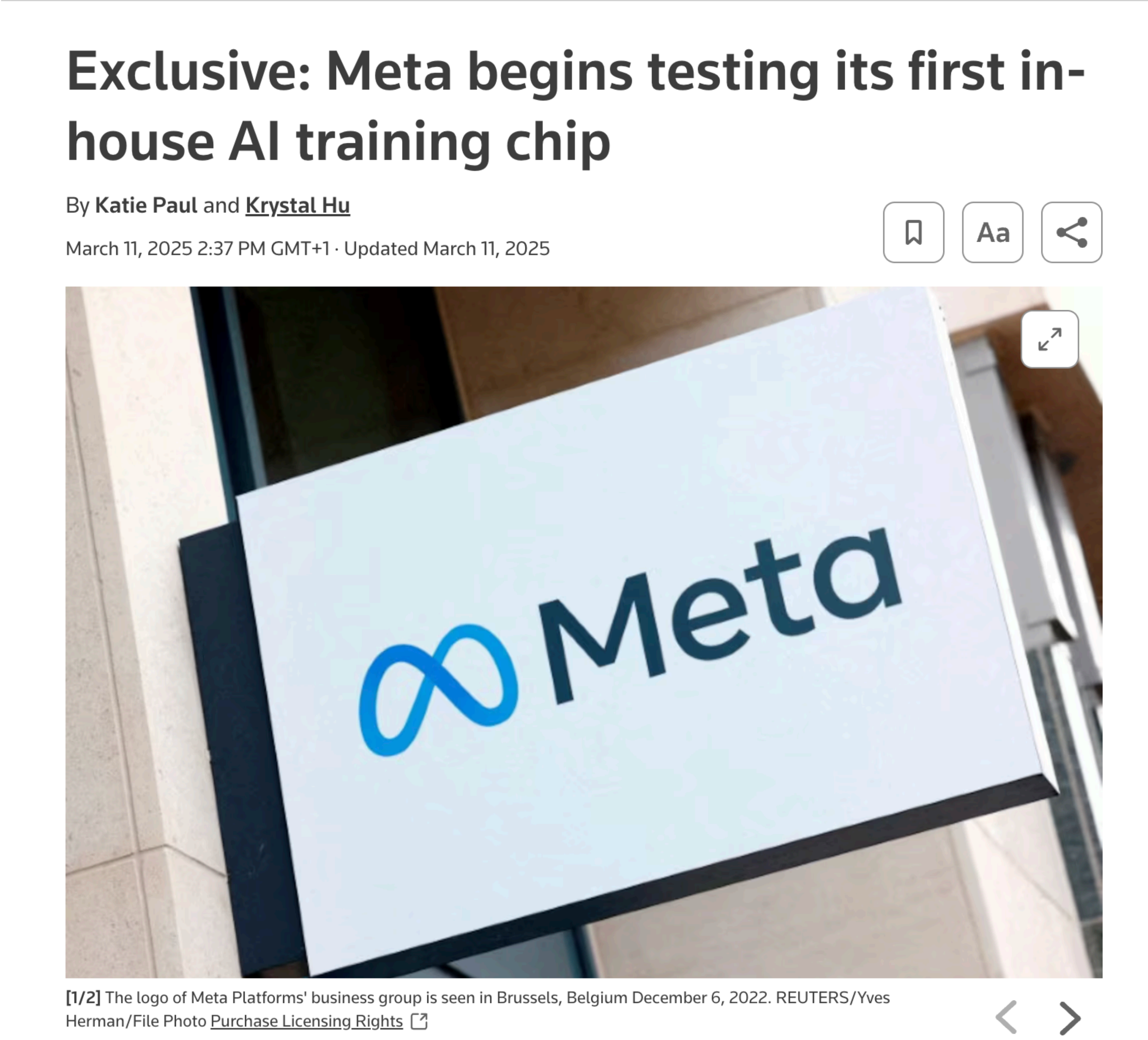
Computing Hardware is No Longer For Everybody



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

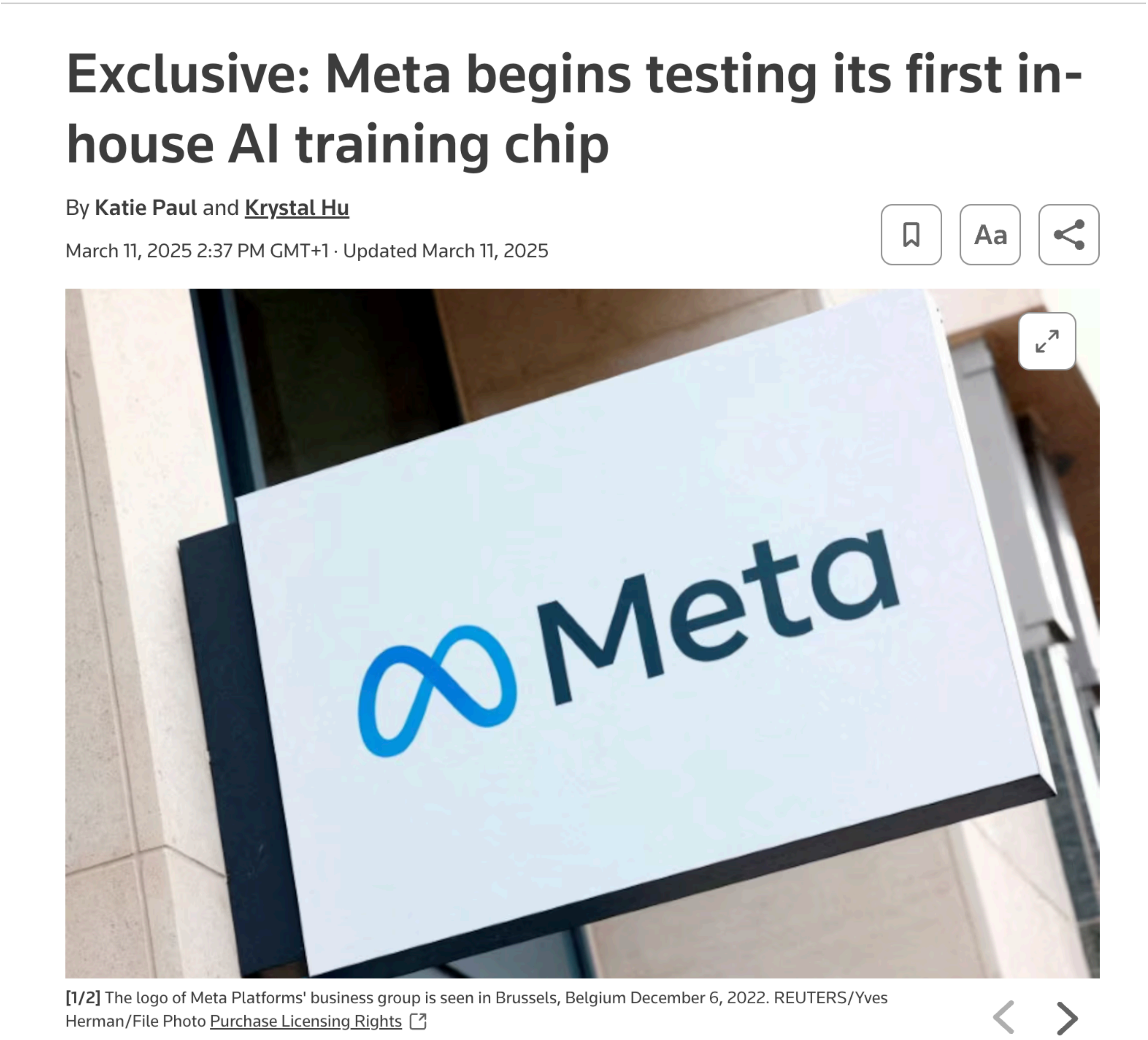
March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[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

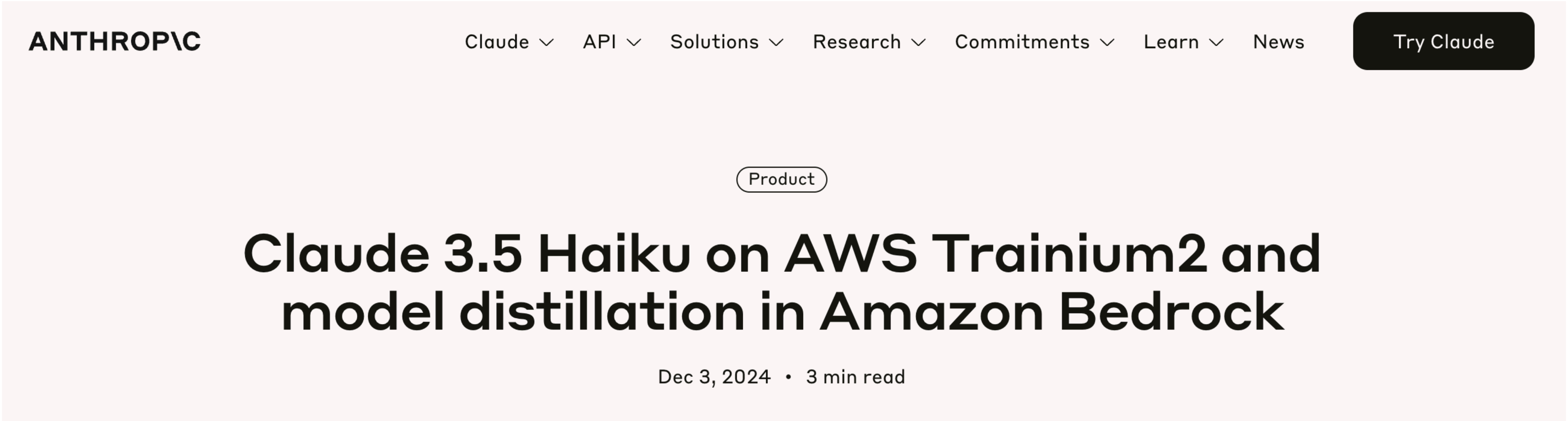


Exclusive: Meta begins testing its first in-house AI training chip

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



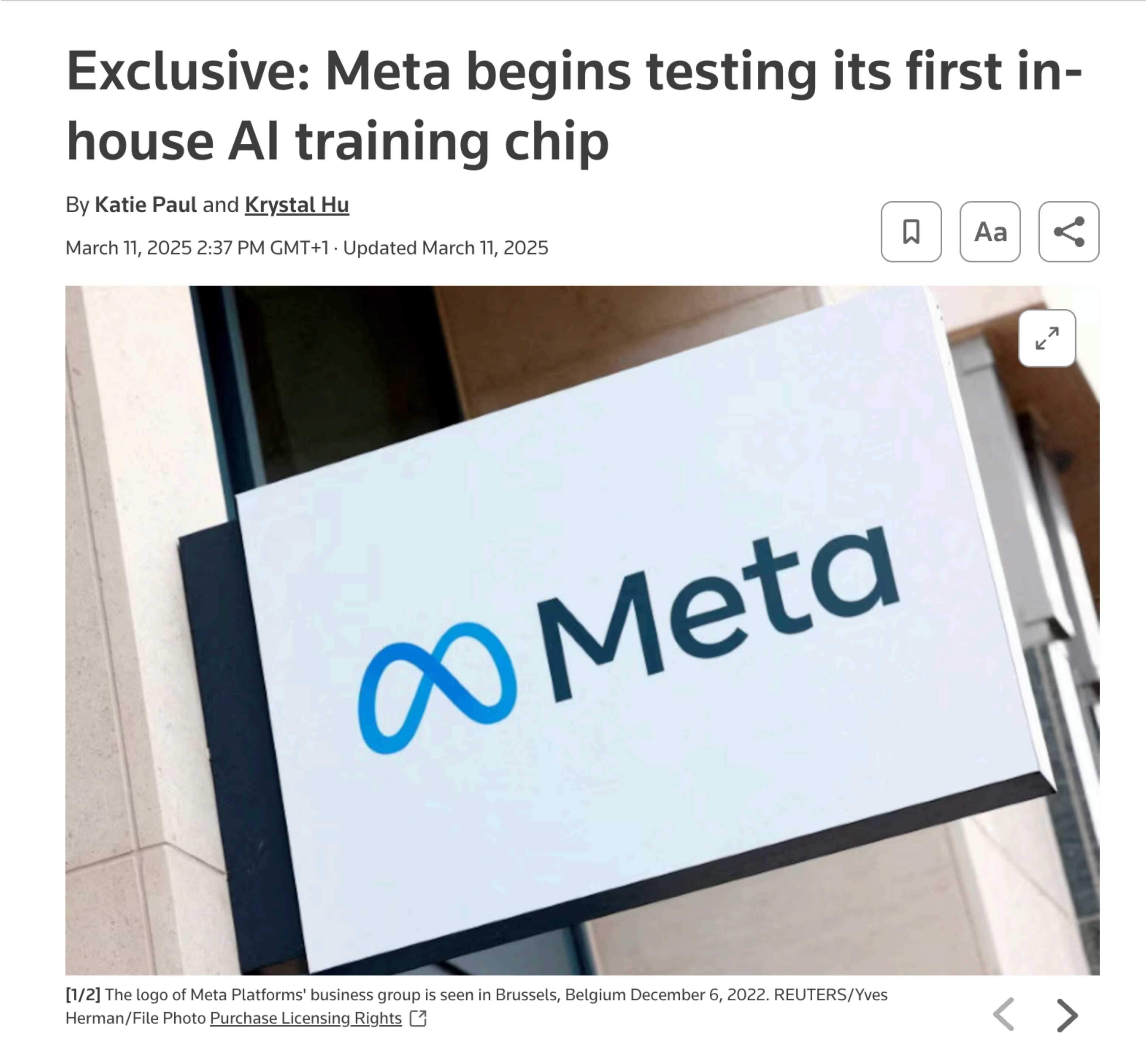
Claude 3.5 Haiku on AWS Trainium2 and model distillation in Amazon Bedrock

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

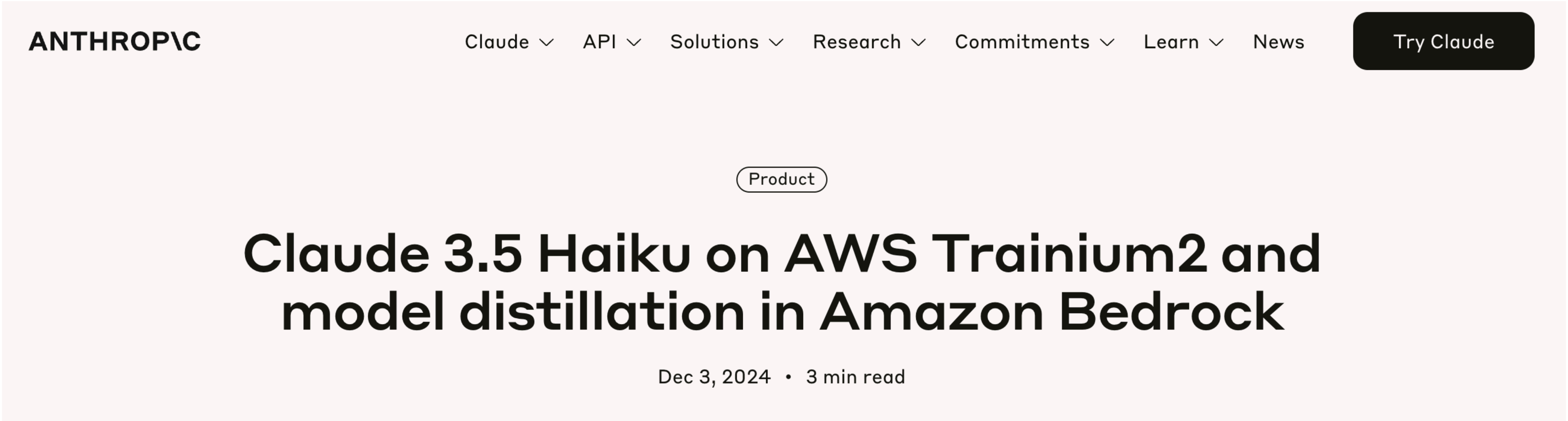


Exclusive: Meta begins testing its first in-house AI training chip

By Katie Paul and Krystal Hu

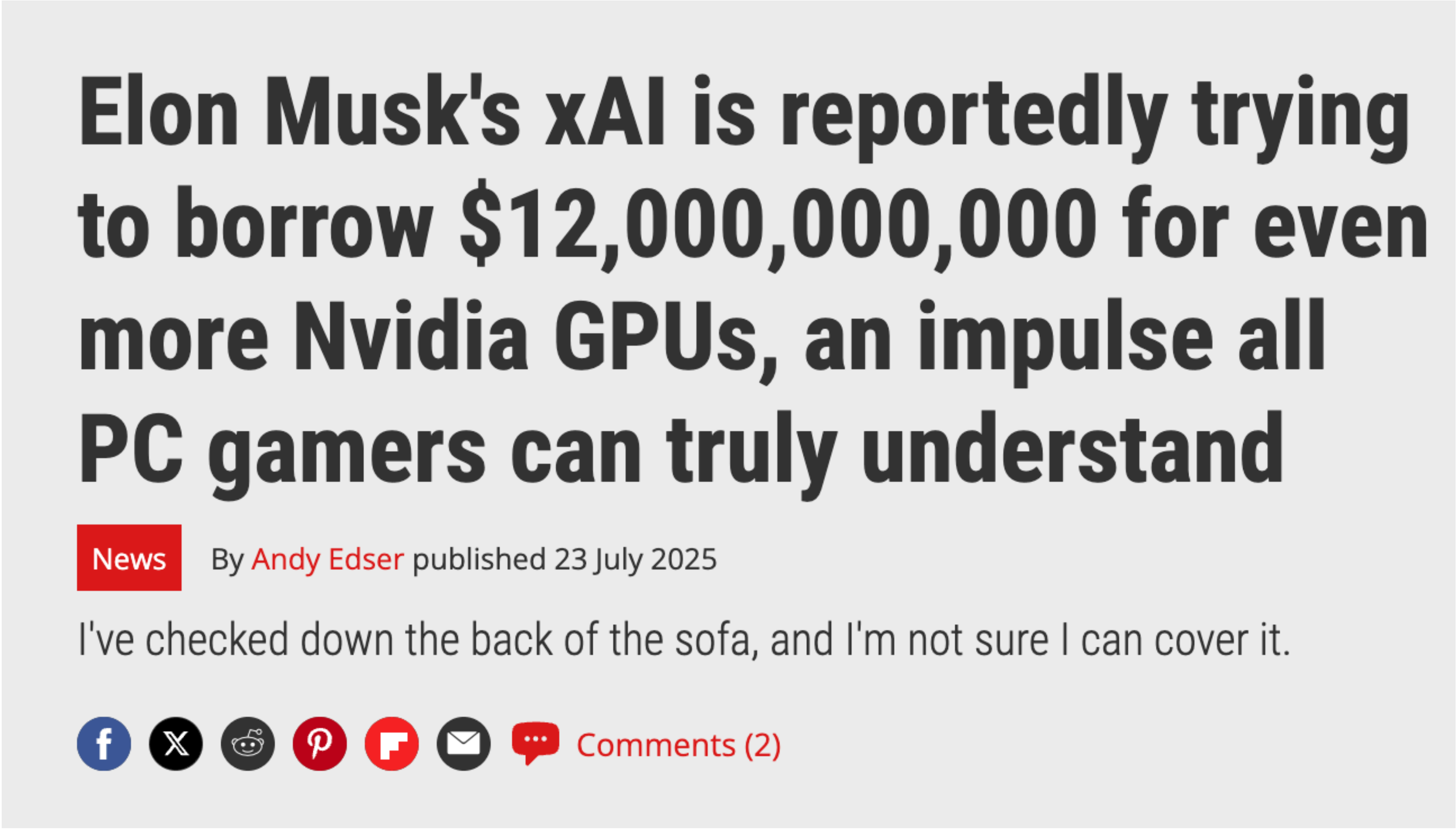
March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[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



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

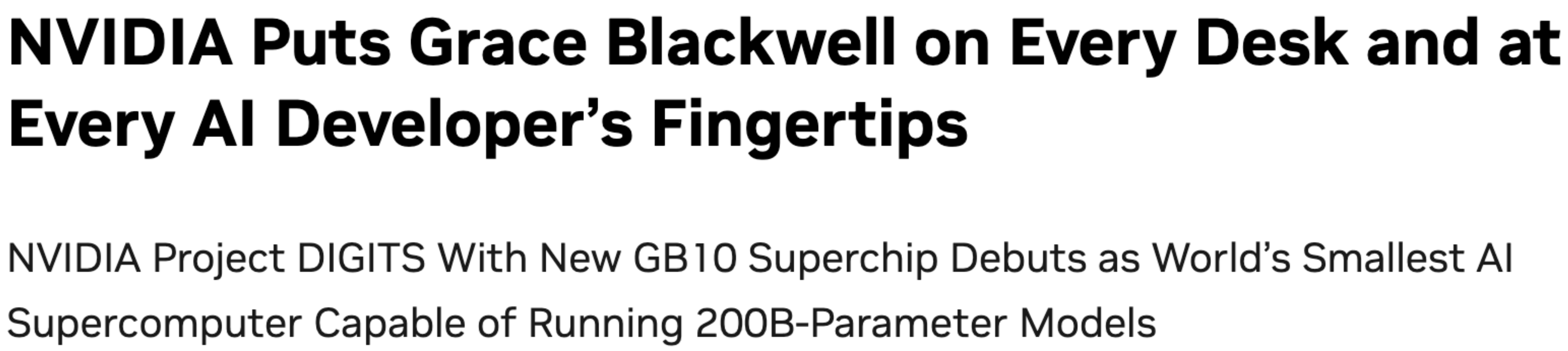
Dec 3, 2024 • 3 min read



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

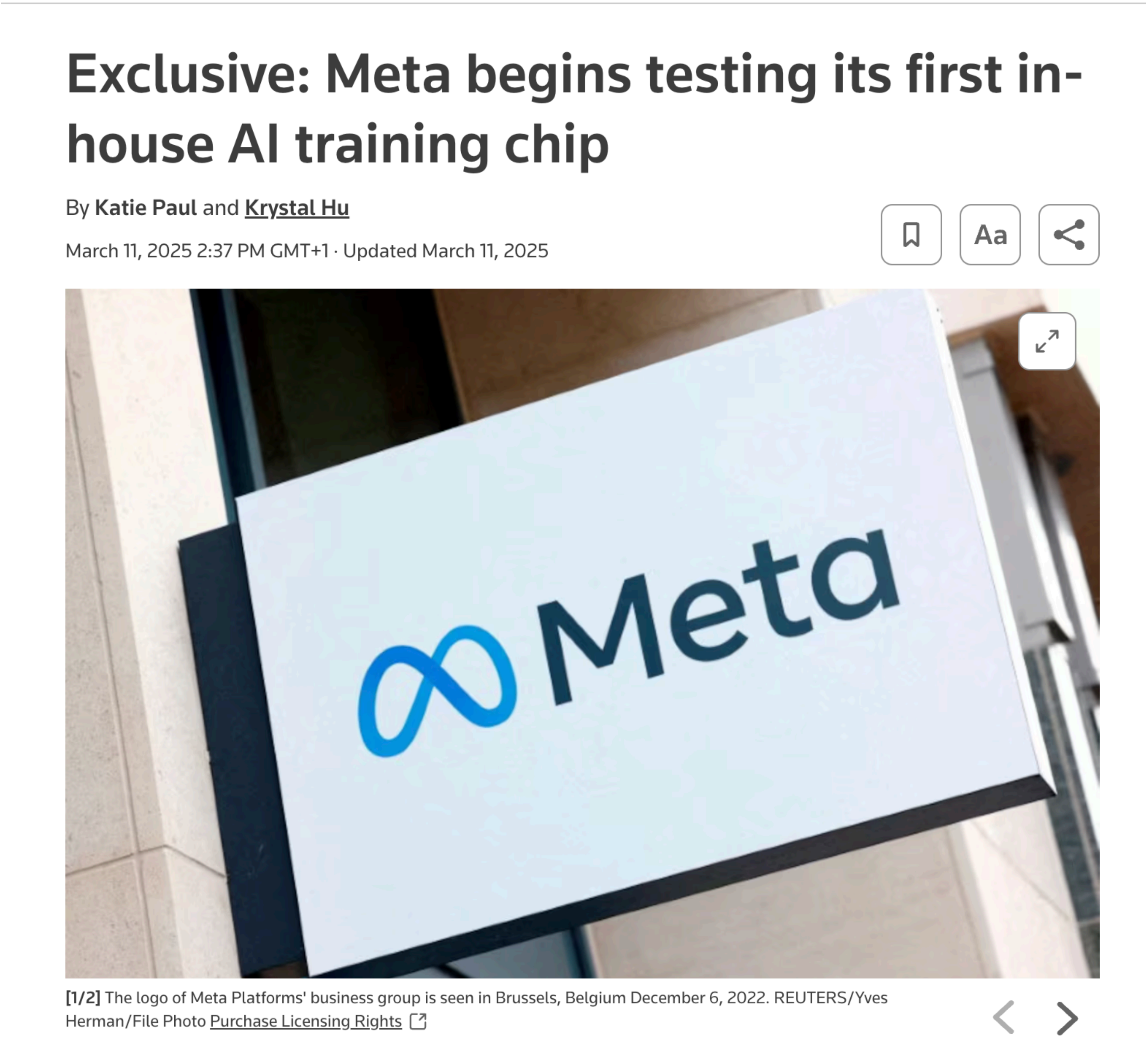
I've checked down the back of the sofa, and I'm not sure I can cover it.



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

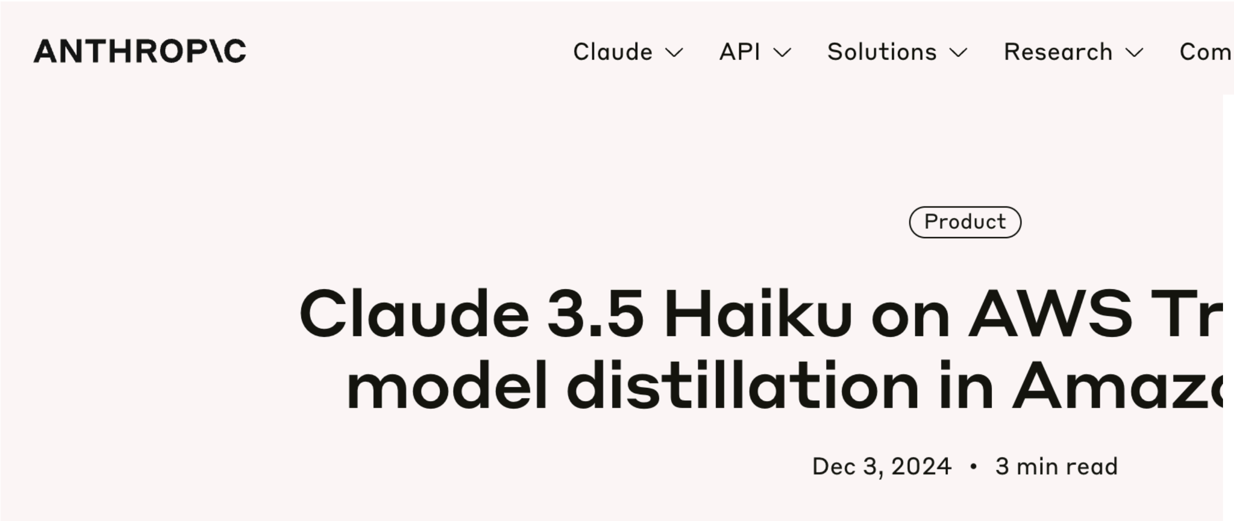


Exclusive: Meta begins testing its first in-house AI training chip

By Katie Paul and Krystal Hu

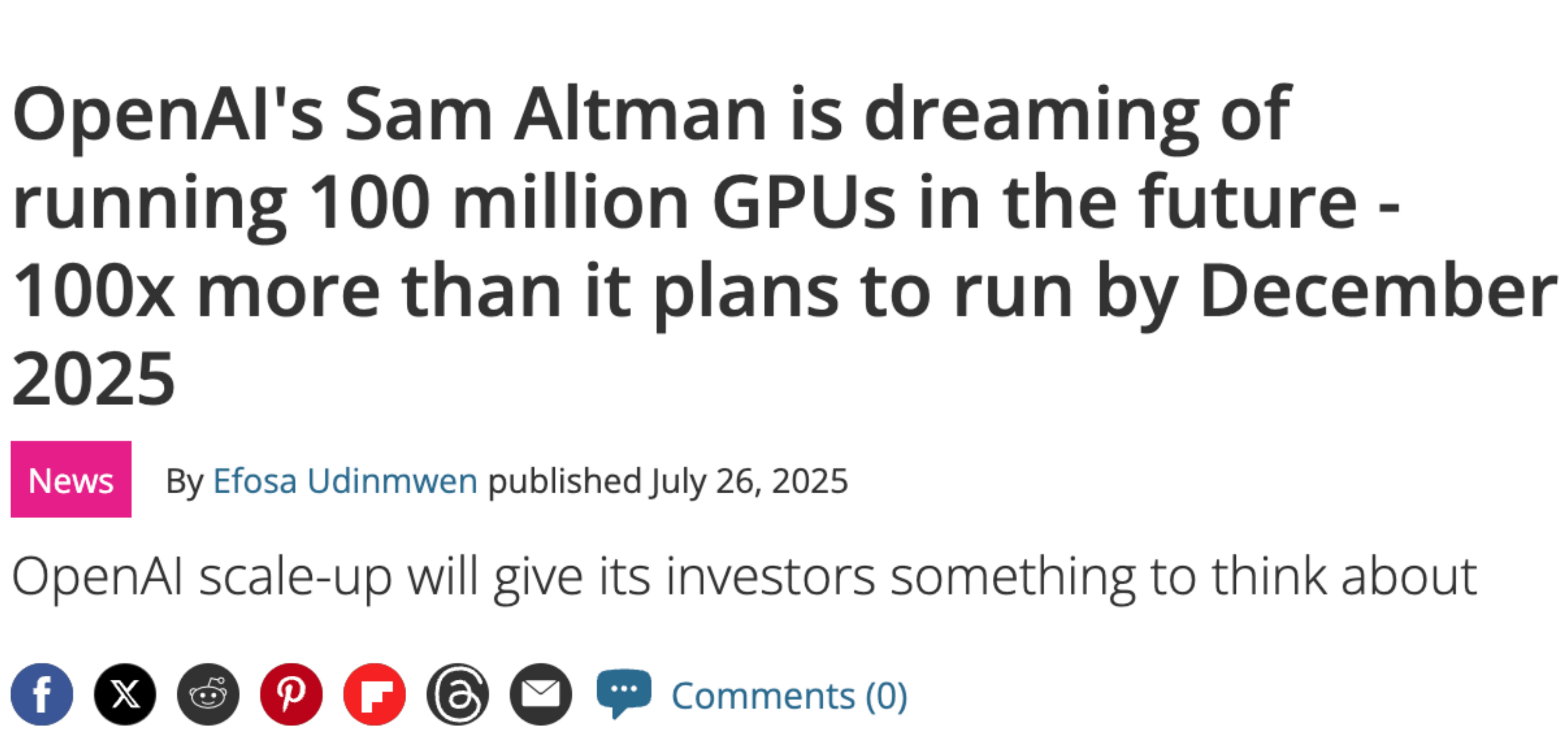
March 11, 2025 2:37 PM GMT+1 · Updated March 11, 2025

[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



Claude 3.5 Haiku on AWS Tr
model distillation in Amaz

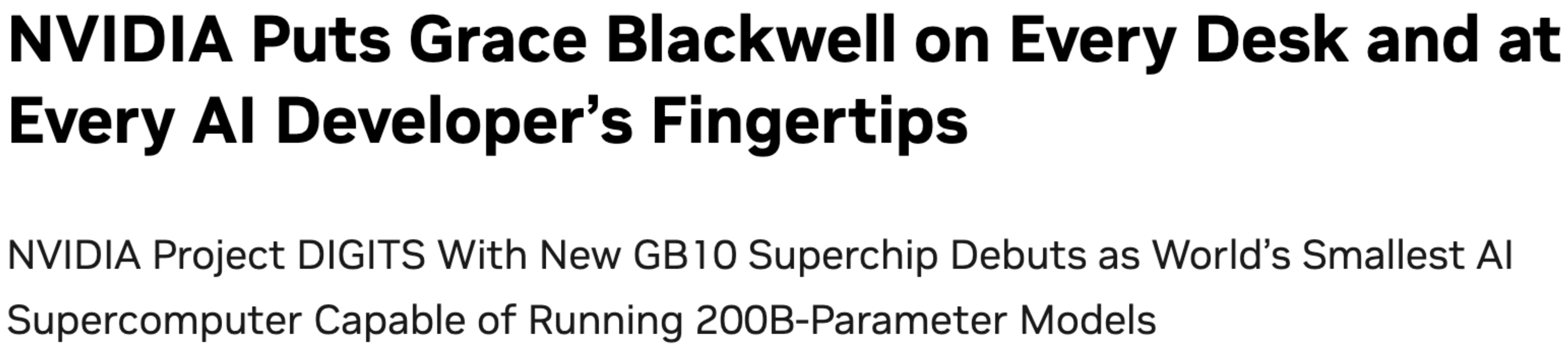
Dec 3, 2024 • 3 min read



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

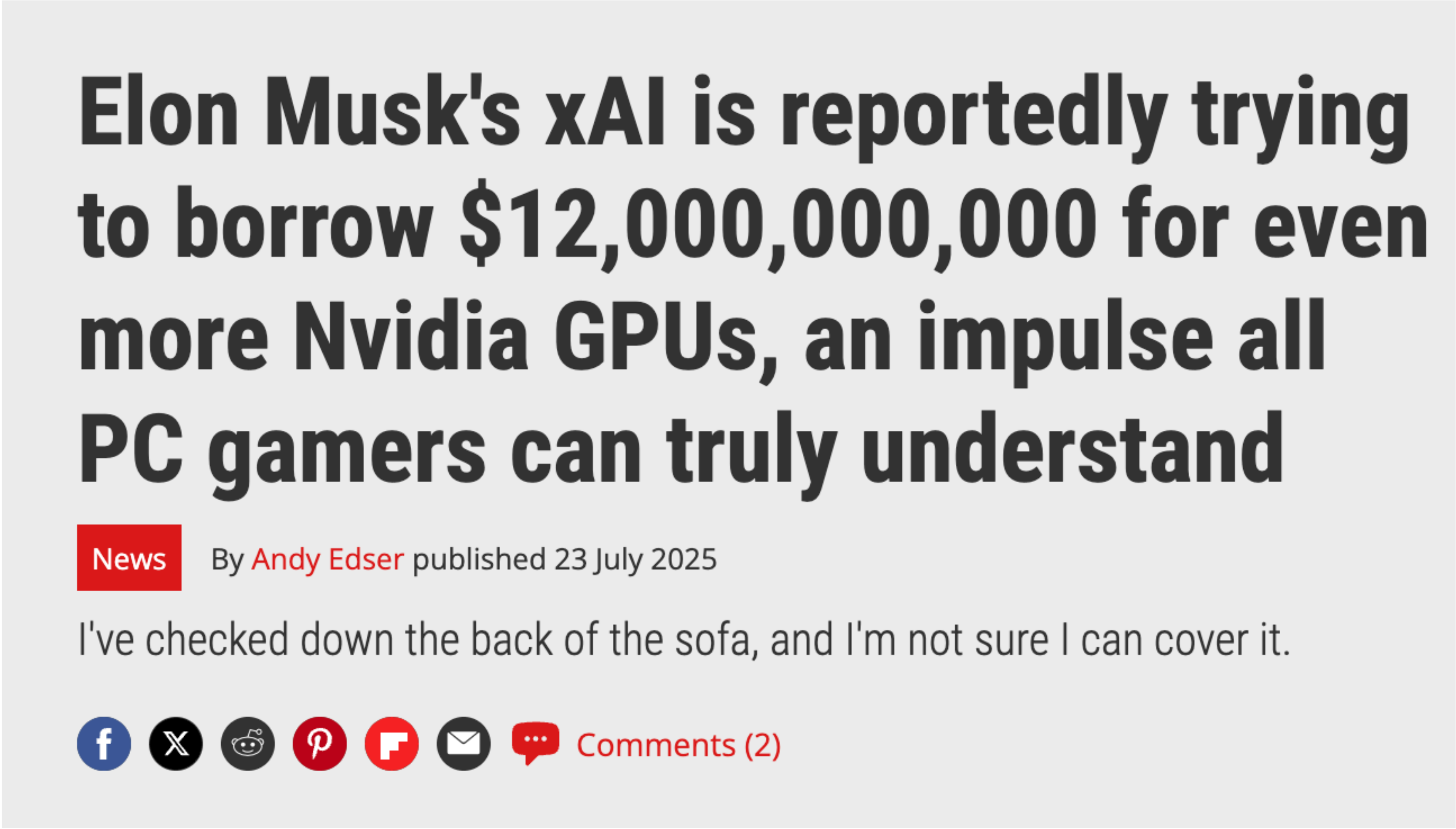
News By Efosa Udinmwen published July 26, 2025

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



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

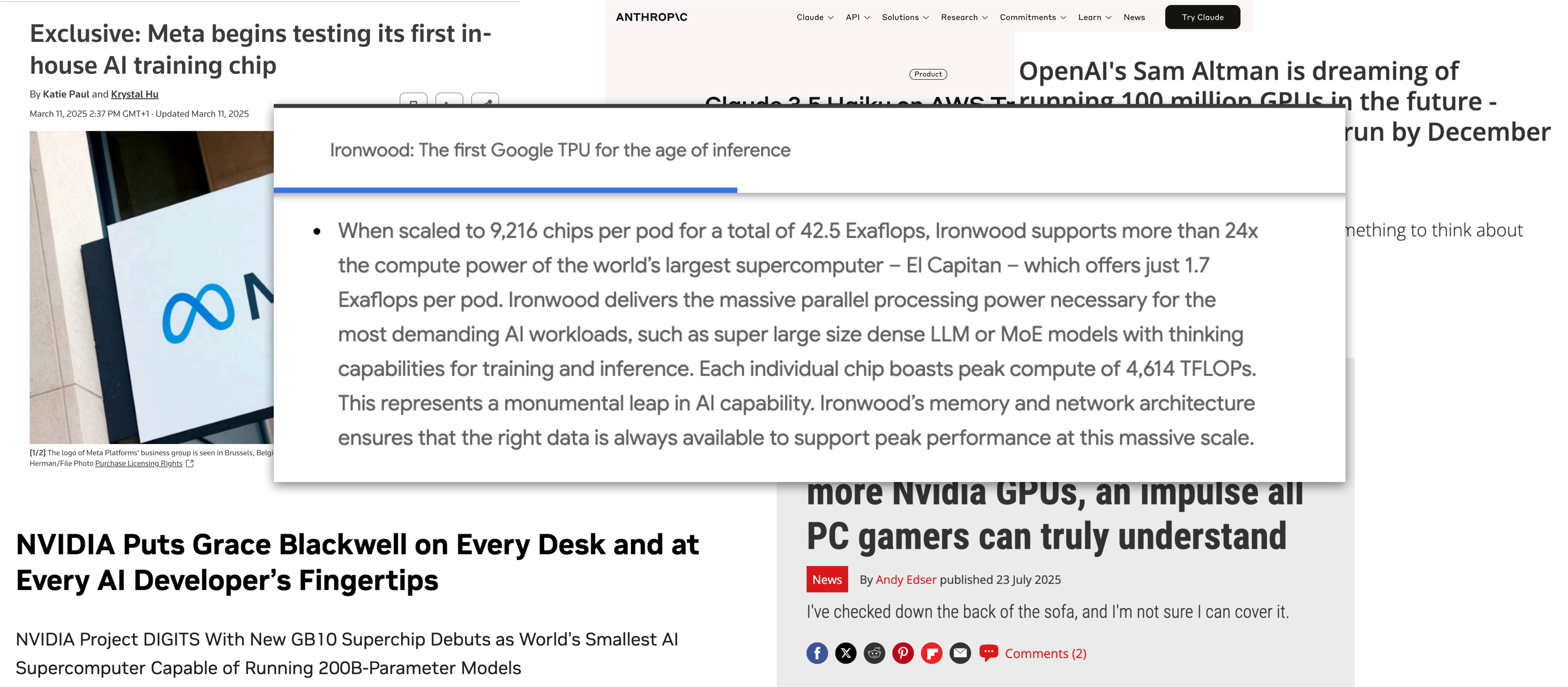


Elon Musk's xAI is reportedly trying to borrow $12,000,000,000 for even more Nvidia GPUs, an impulse all PC gamers can truly understand

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.

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

Claude 3.5 Haiku on AWS Tr

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

mething 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

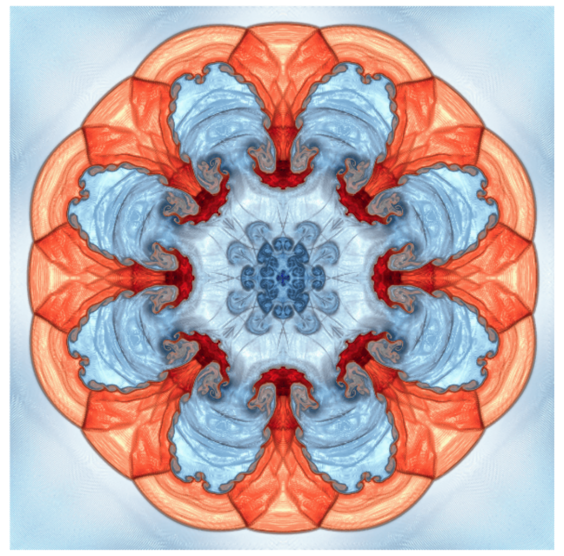
NVIDIA Project DIGITS With New GB10 Superchip Debuts as World's Smallest AI Supercomputer Capable of Running 200B-Parameter Models

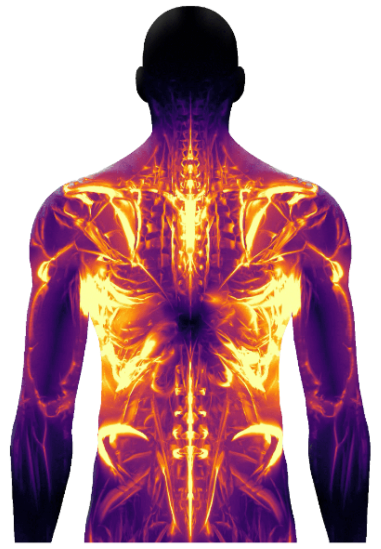
more Nvidia GPUs, an impulse all PC gamers can truly understand

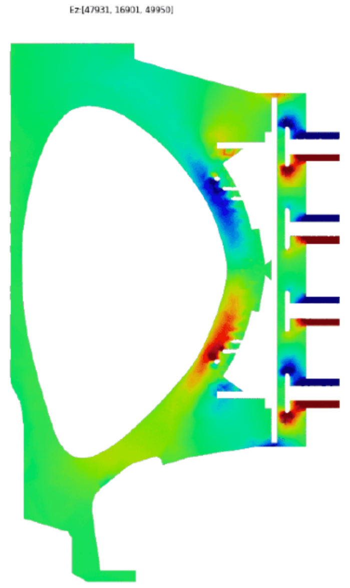
News By Andy Edser published 23 July 2025

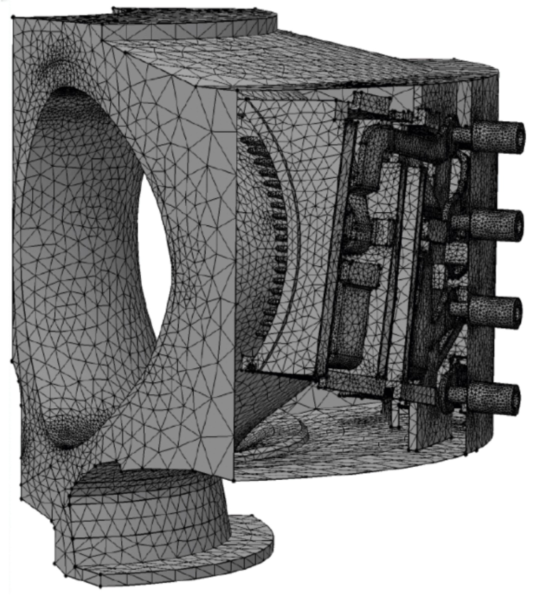
I've checked down the back of the sofa, and I'm not sure I can cover it.

Lingua Franca of Scientific Computing

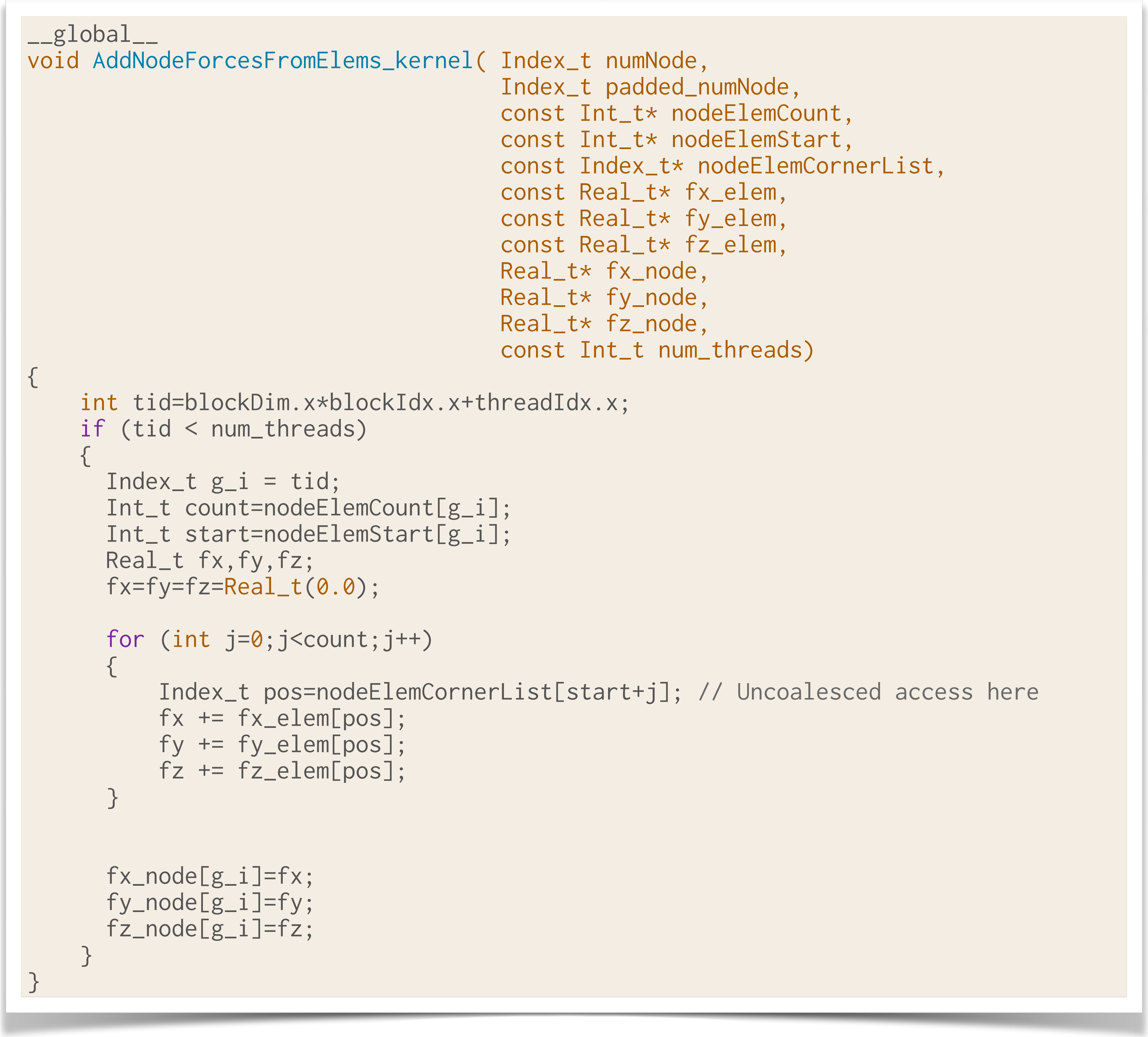






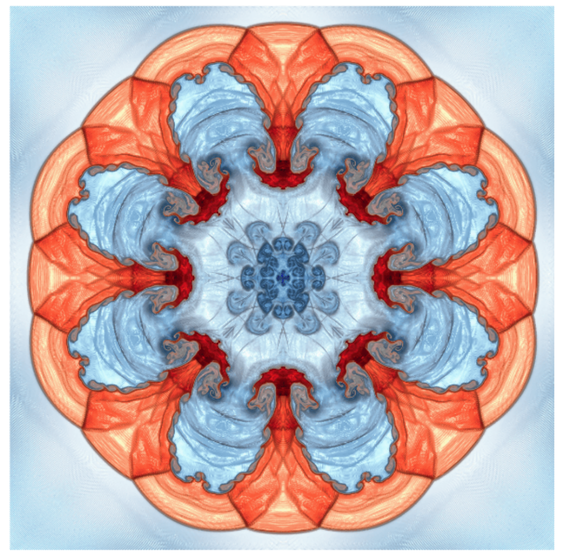


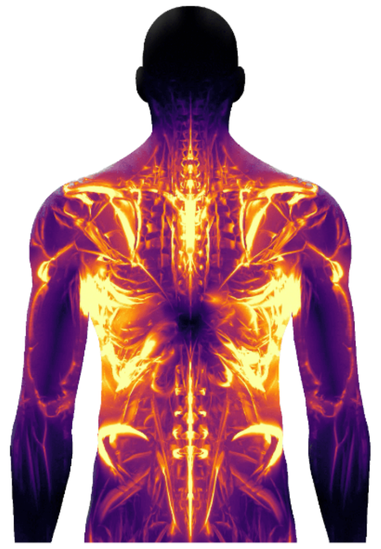
- Scientists do not write TPU* code

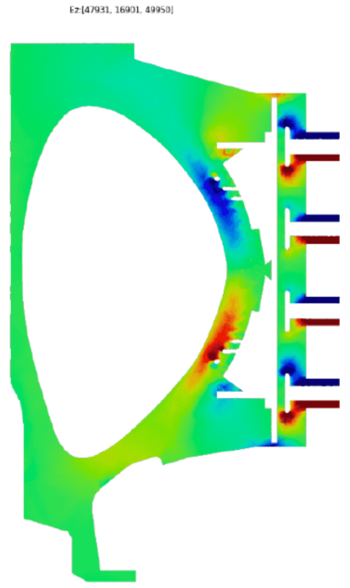


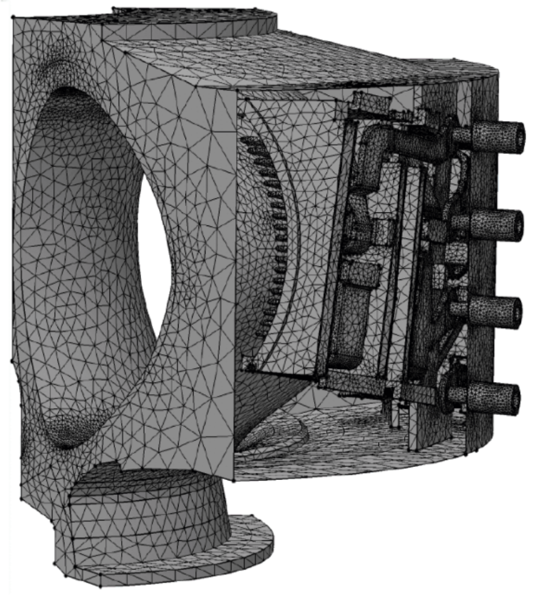
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* 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

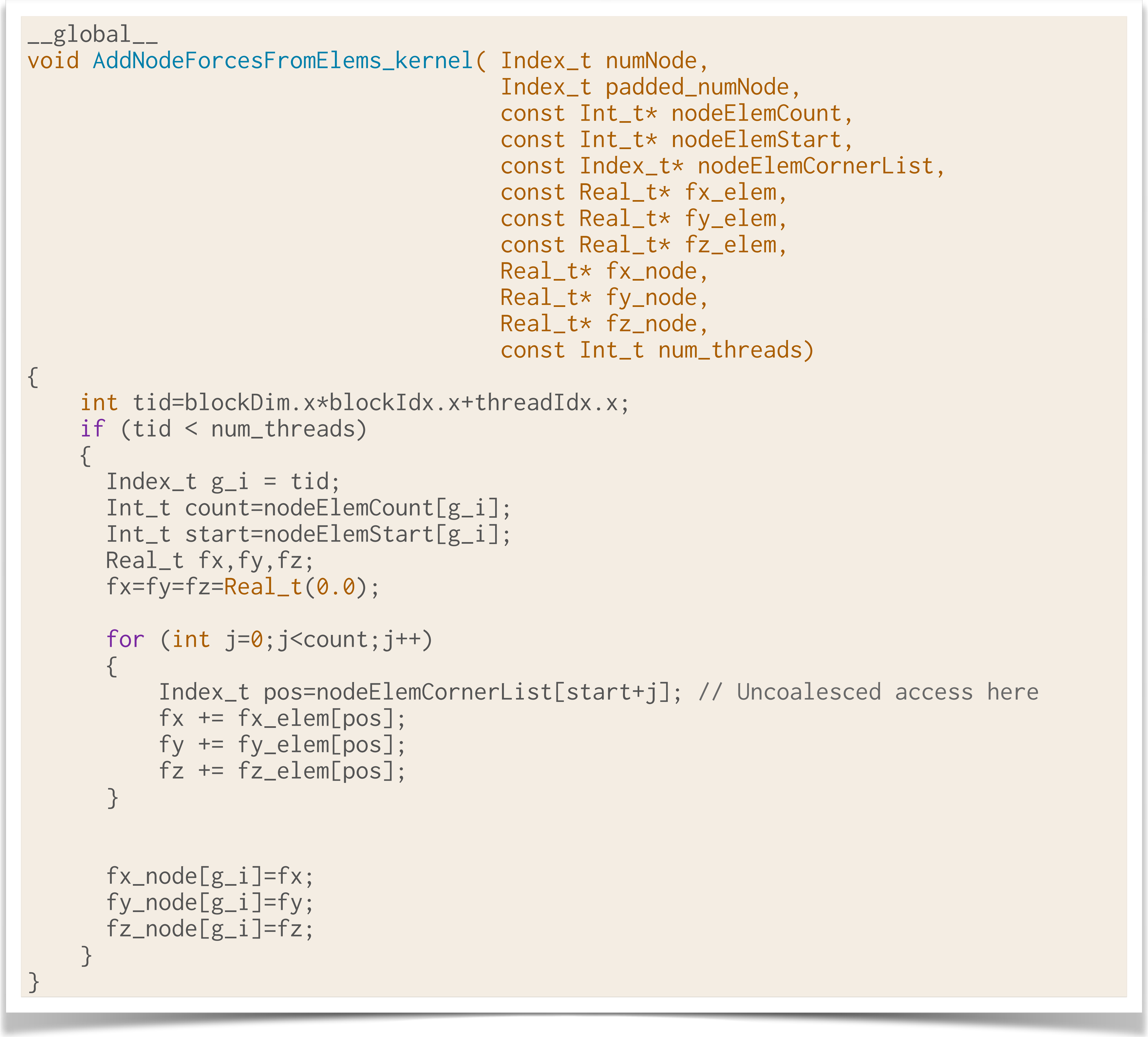






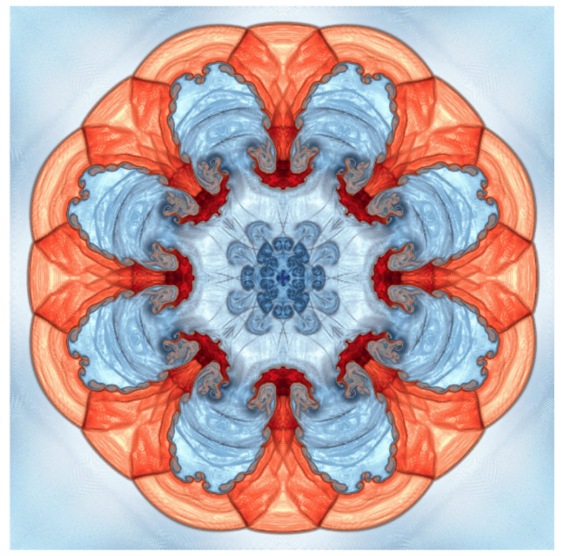


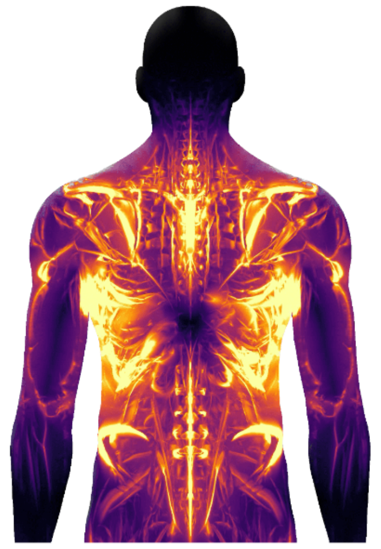
- Scientists do not write TPU* code
    - BIG (MFEM library alone is 737K LOC)

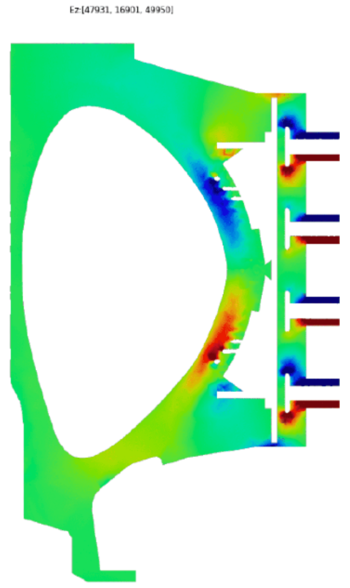


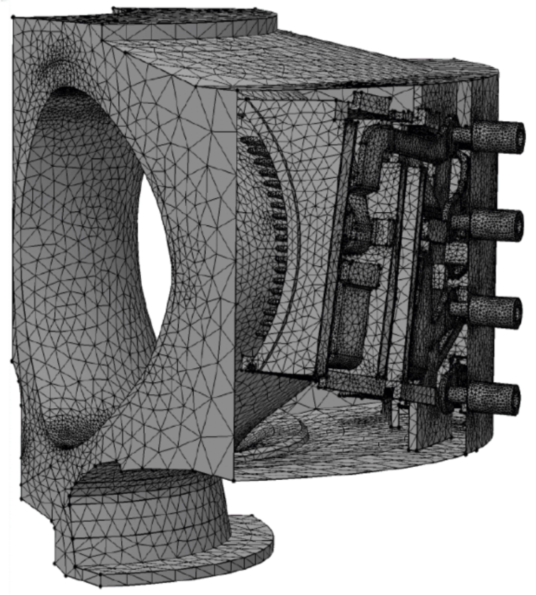
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* 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

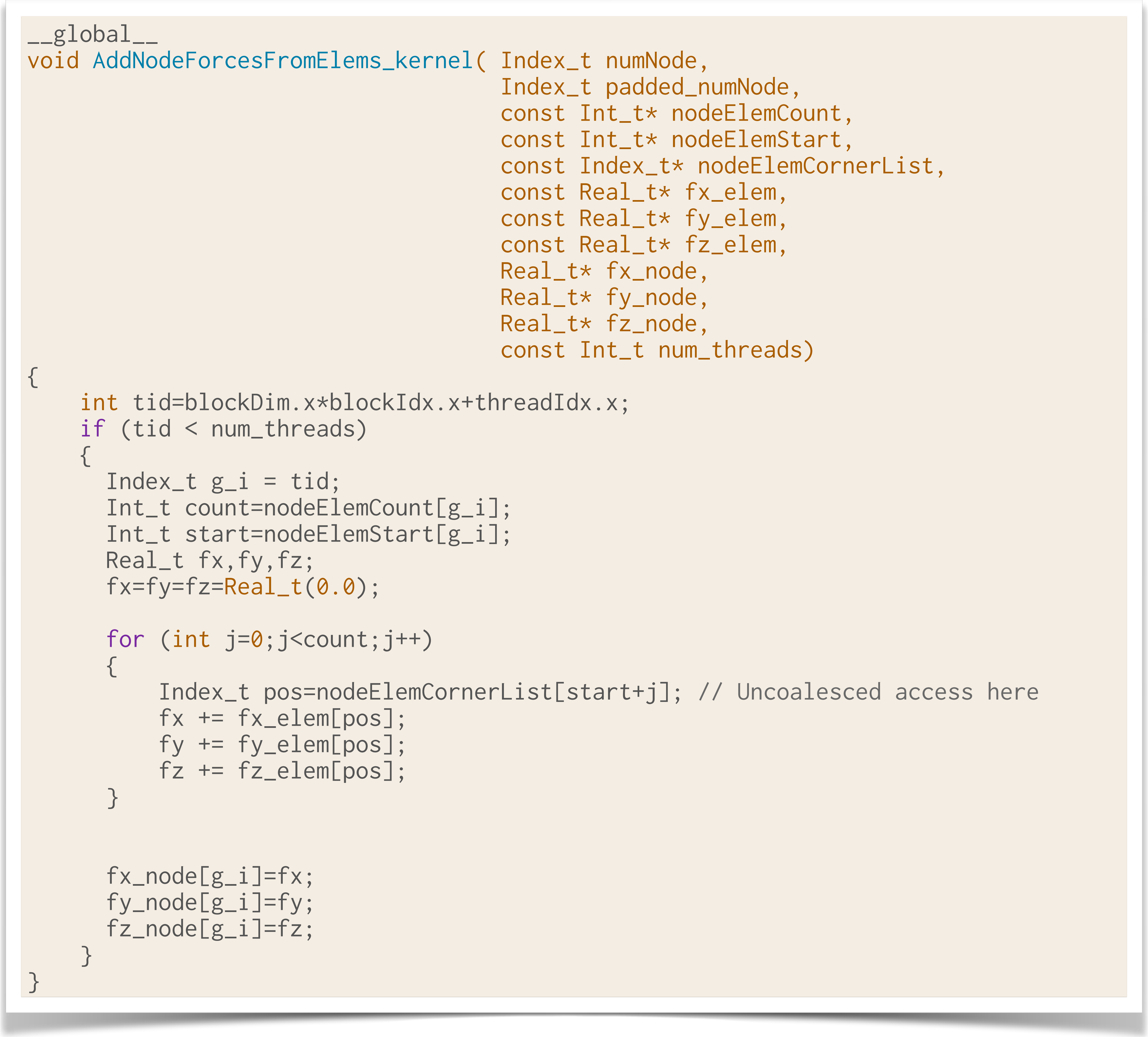






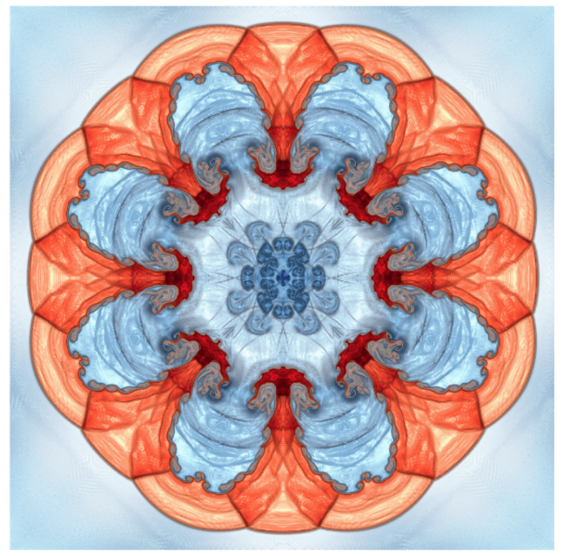


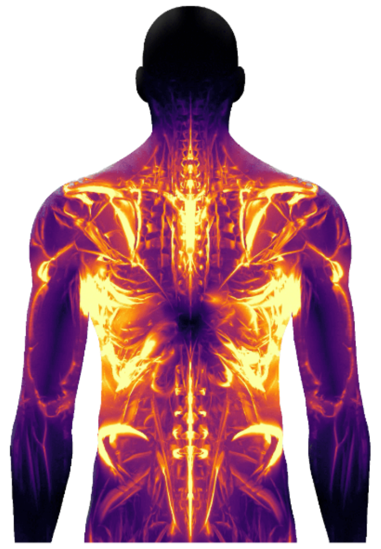
- Scientists do not write TPU* code
    - BIG (MFEM library alone is 737K LOC)
    - Templated

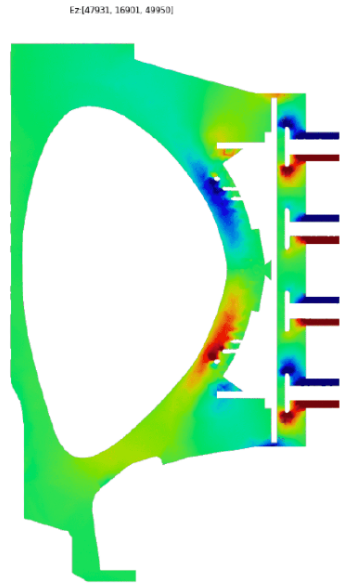


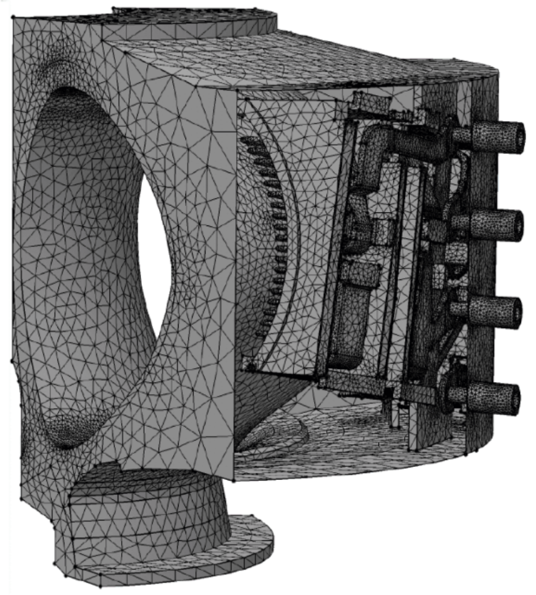
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* 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

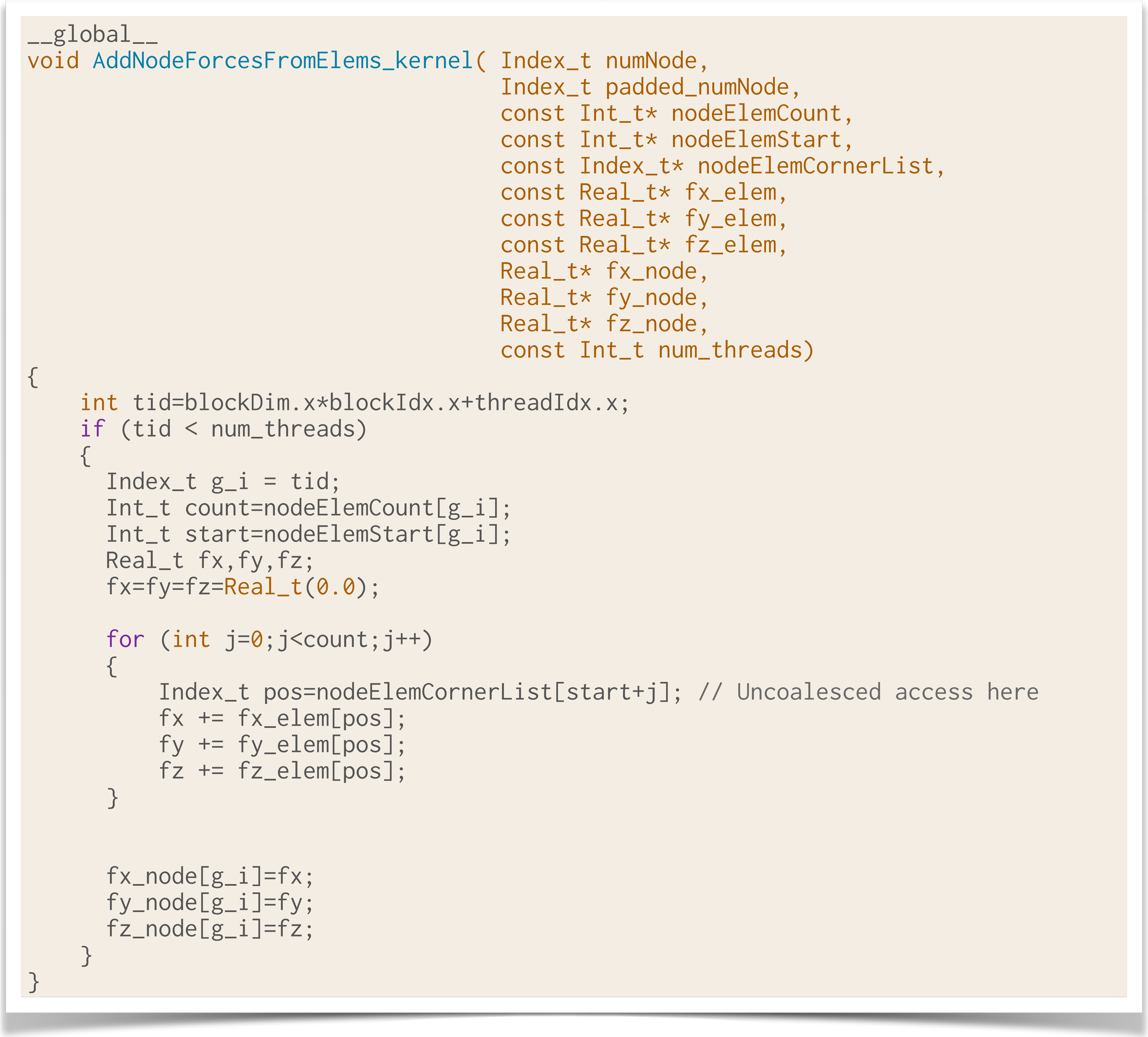






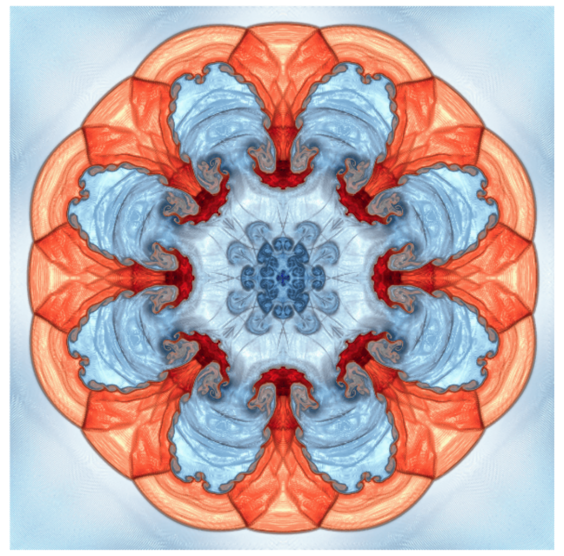


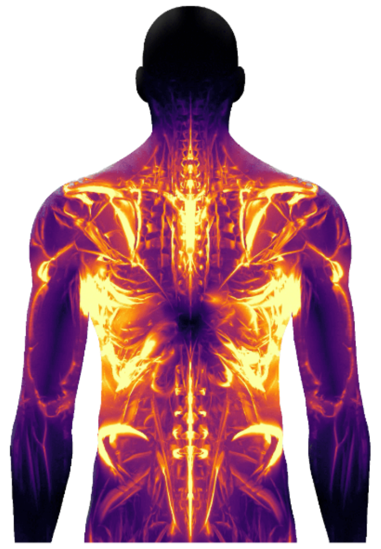
- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)
  - Templated
  - Not in Python

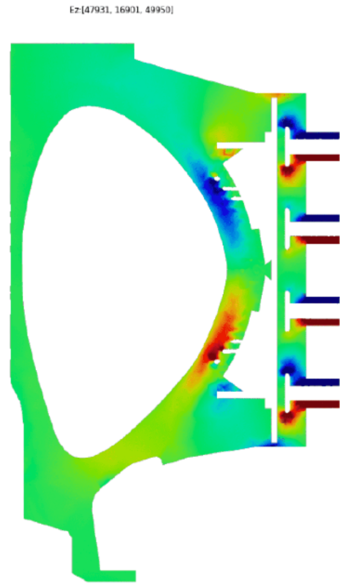


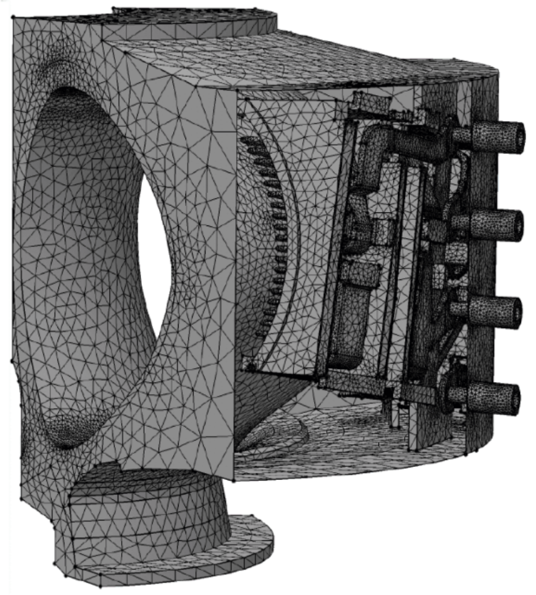
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* 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

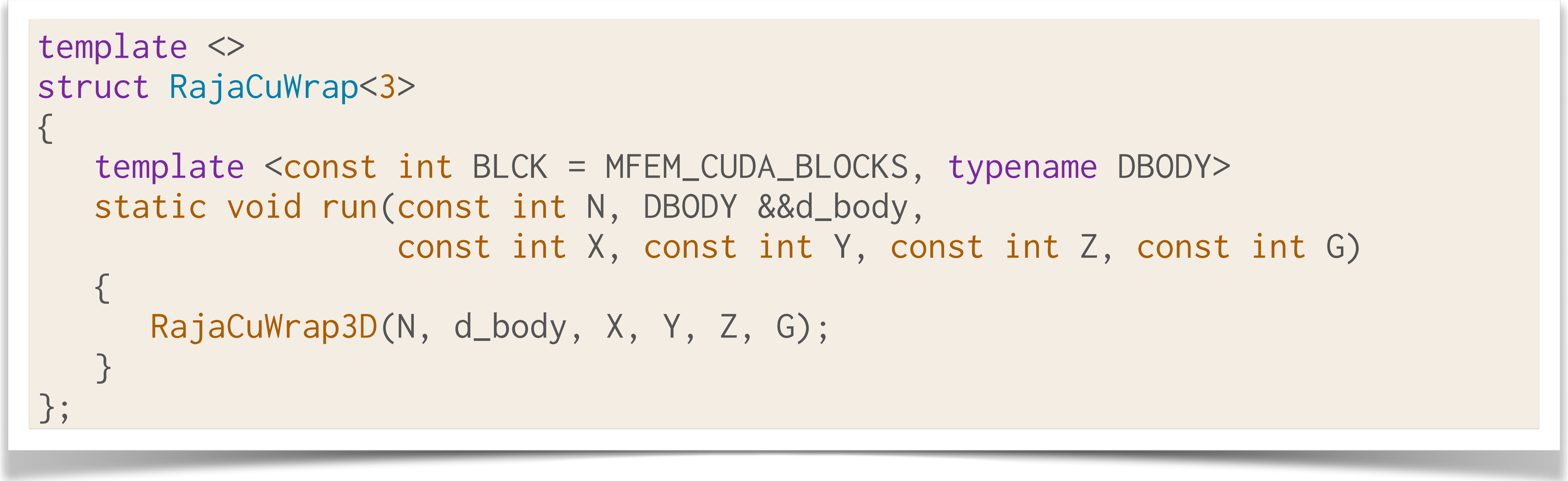




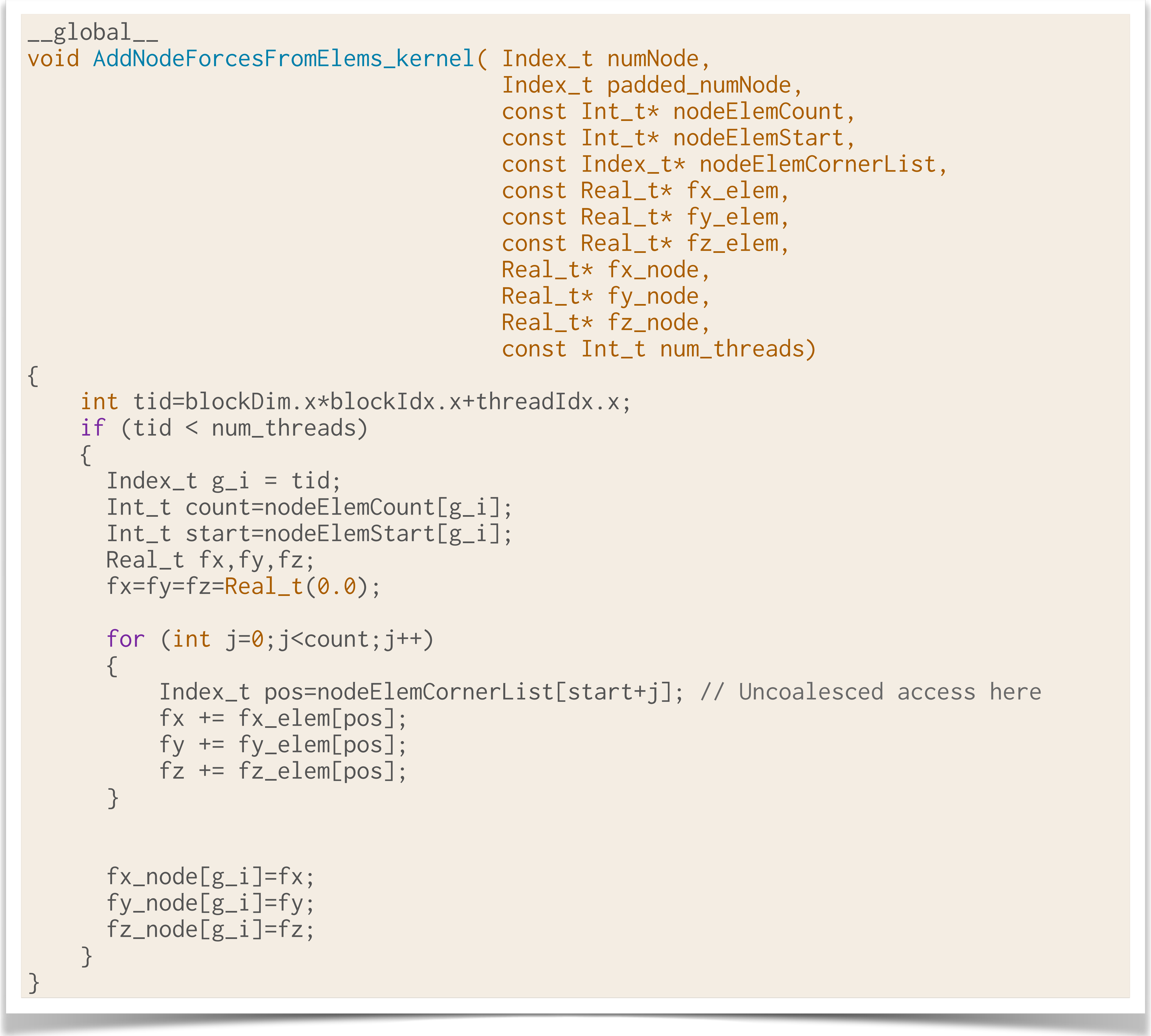




- Scientists do not write TPU* code
  - BIG (MFEM library alone is 737K LOC)
  - Templated
  - Not in Python
  - Sometimes* in CUDA



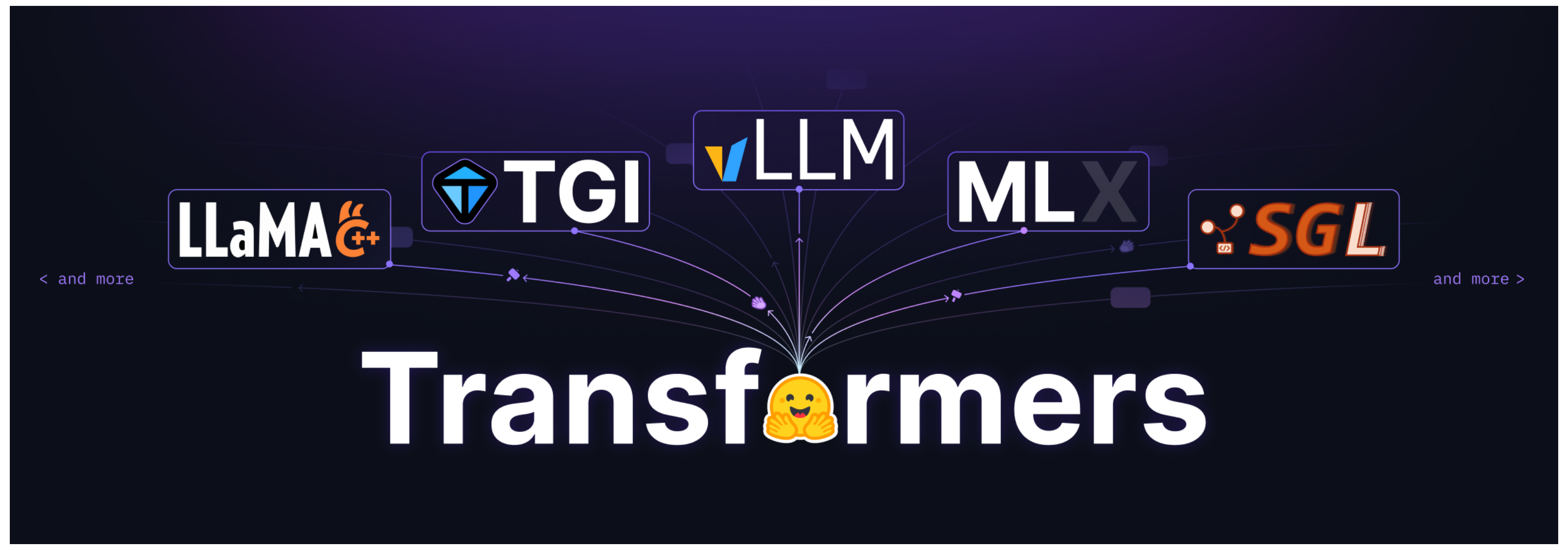
```
template <>
struct RajaCuWrap<3>
{
   template <const int BLCK = MFEM_CUDA_BLOCKS, typename DBODY>
   static void run(const int N, DBODY &&d_body,
                   const int X, const int Y, const int Z, const int G)
   {
      RajaCuWrap3D(N, d_body, X, Y, Z, G);
   }
};
```

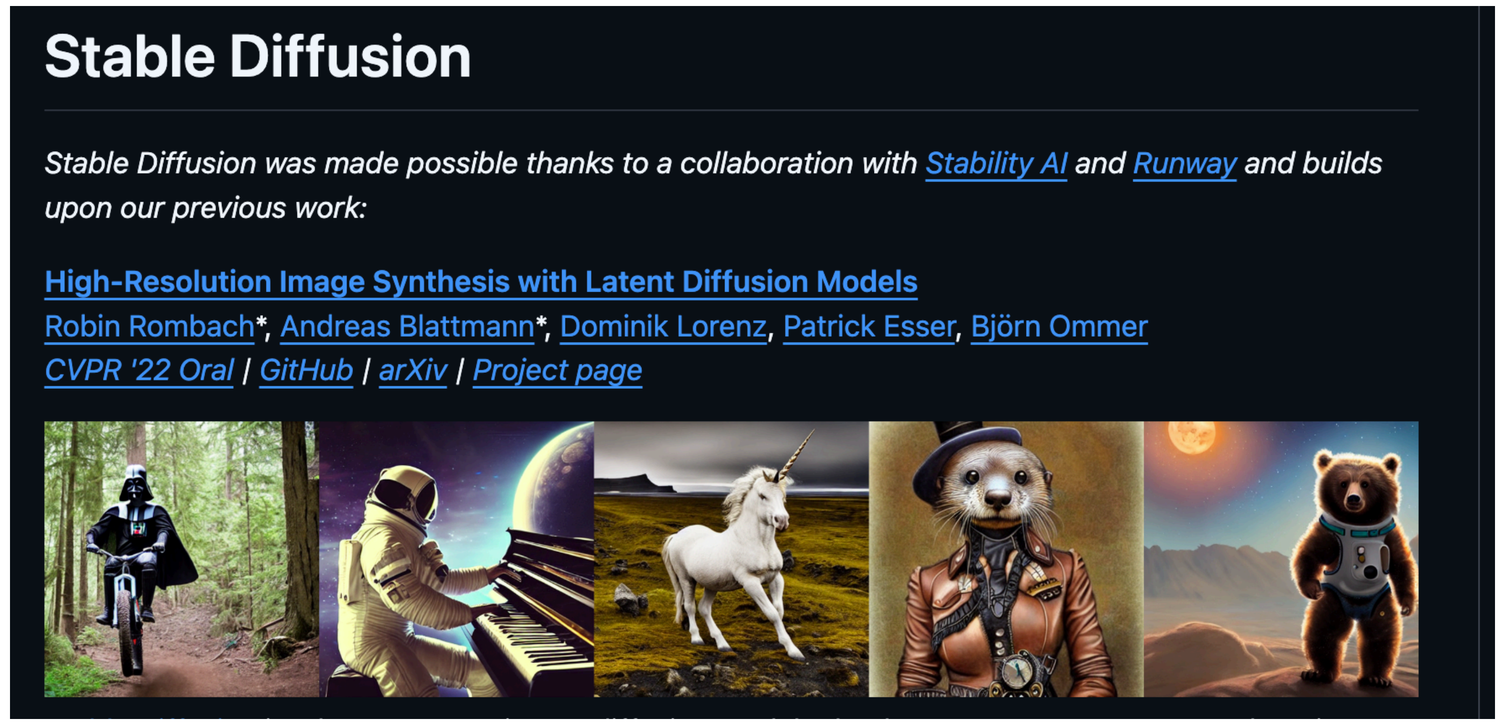


```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                    Index_t padded_numNode,
                                    const Int_t* nodeElemCount,
                                    const Int_t* nodeElemStart,
                                    const Index_t* 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?



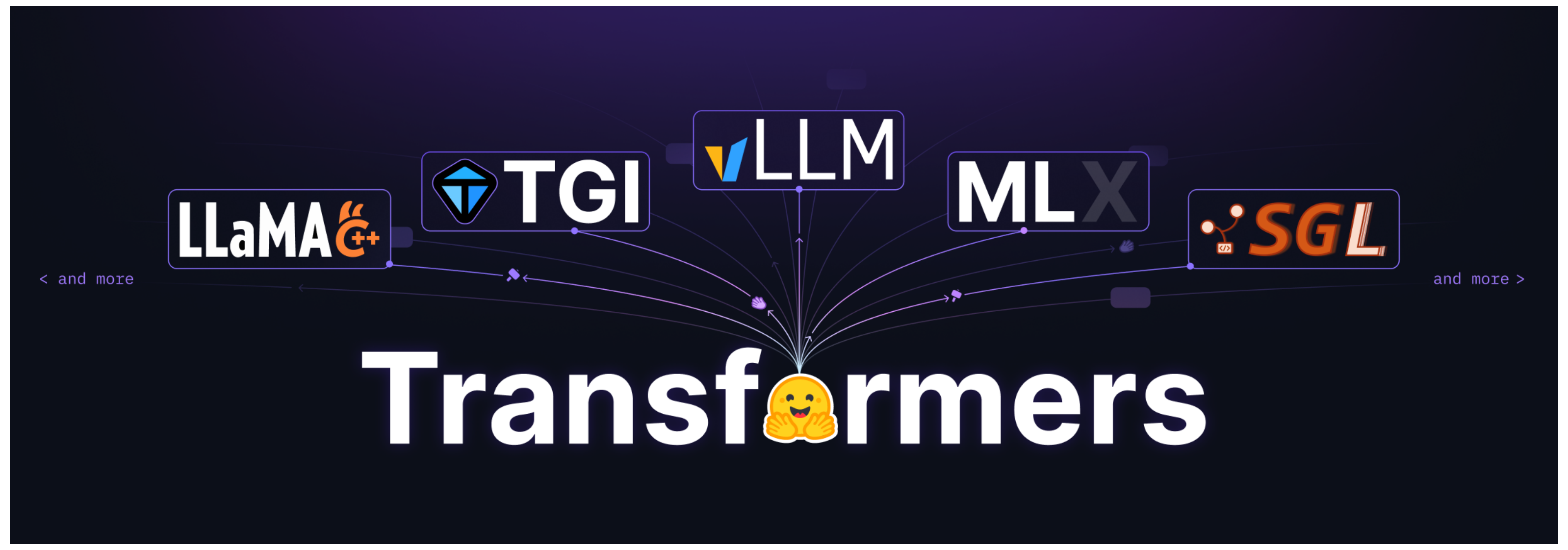


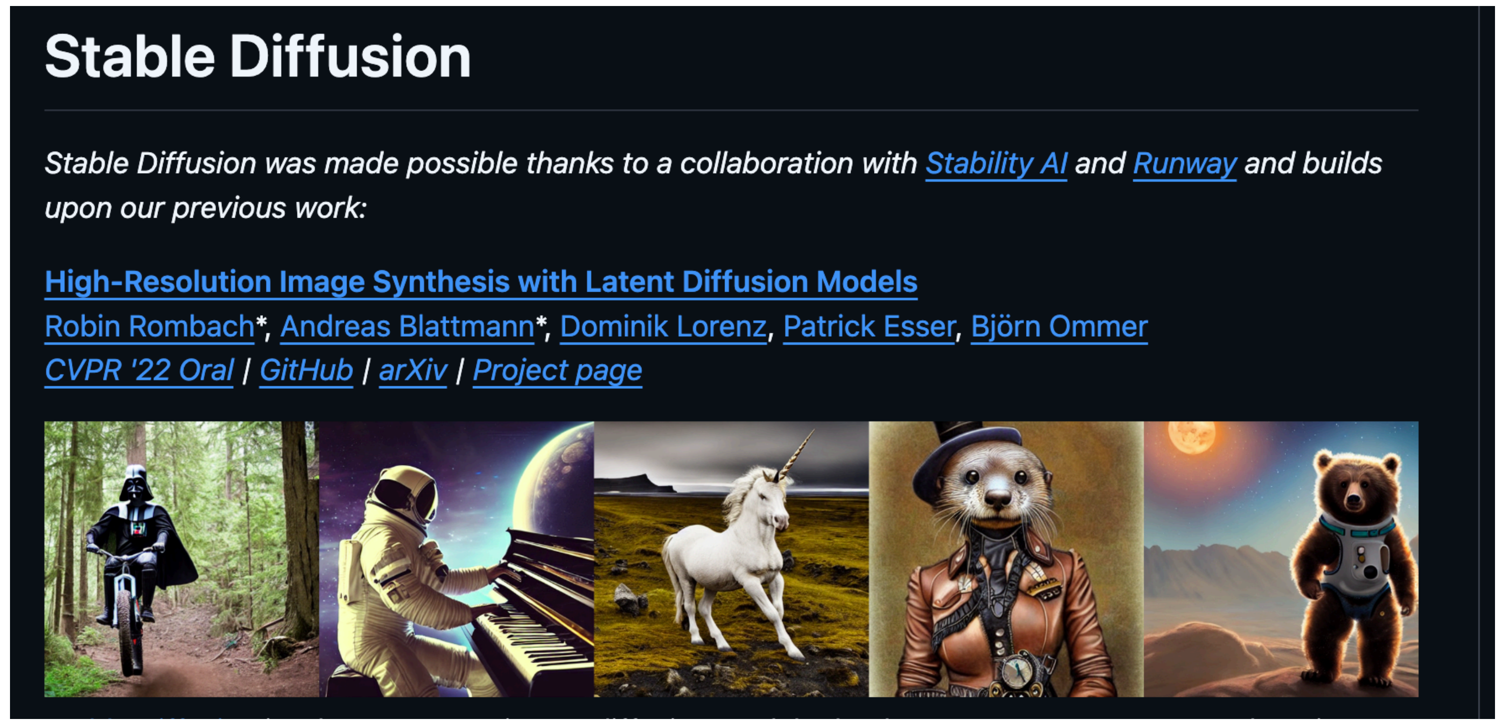
Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with Stability AI and Runway and builds upon our previous work:

High-Resolution Image Synthesis with Latent Diffusion Models
Robin Rombach*, Andreas Blattmann*, Dominik Lorenz, Patrick Esser, Björn Ommer
CVPR '22 Oral | *GitHub* | *arXiv* | *Project page*

How do we write ML Accelerator code now?

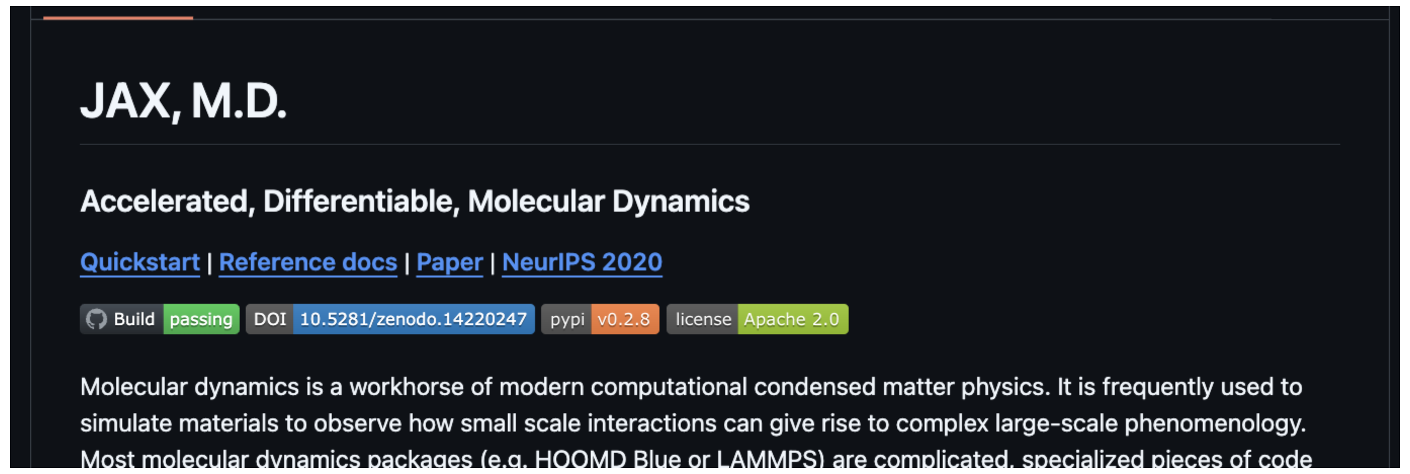




Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with Stability AI and Runway and builds upon our previous work:

High-Resolution Image Synthesis with Latent Diffusion Models
Robin Rombach*, Andreas Blattmann*, Dominik Lorenz, Patrick Esser, Björn Ommer
CVPR '22 Oral | GitHub | arXiv | Project page



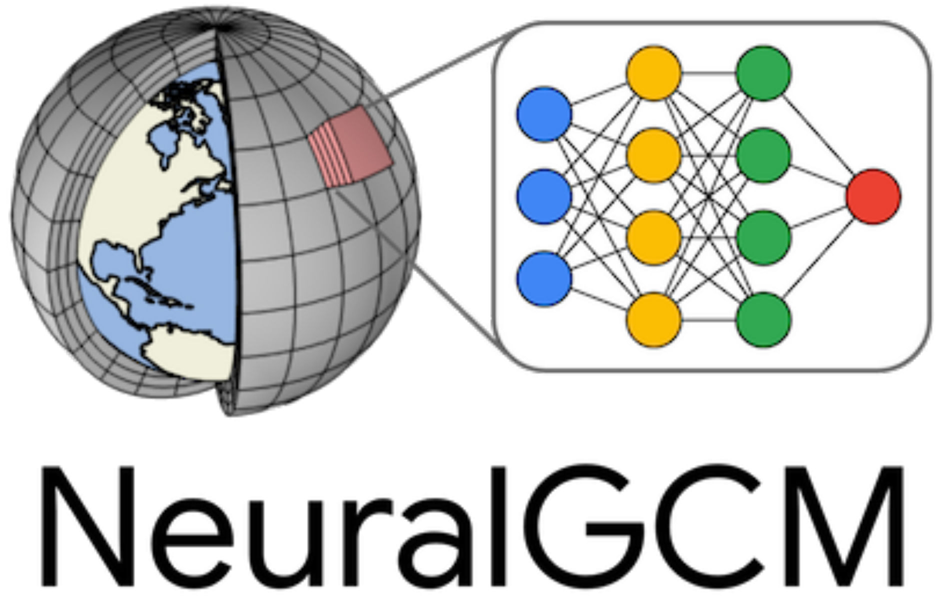
JAX, M.D.

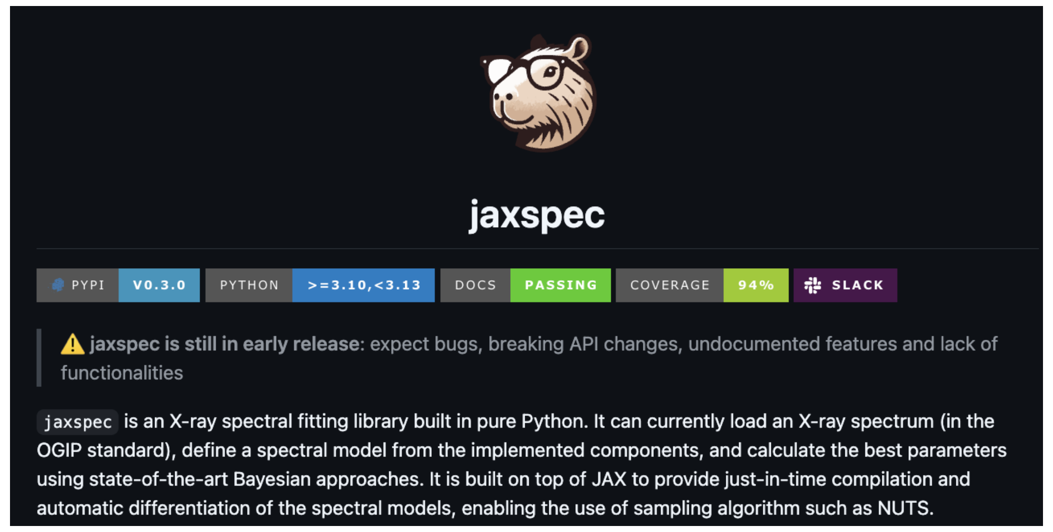
Accelerated, Differentiable, Molecular Dynamics

Quickstart | Reference docs | Paper | NeurIPS 2020

Build passing | DOI 10.5281/zenodo.14220247 | pypi v0.2.8 | license Apache 2.0

Molecular dynamics is a workhorse of modern computational condensed matter physics. It is frequently used to simulate materials to observe how small scale interactions can give rise to complex large-scale phenomenology. Most molecular dynamics packages (e.g. HOOMD Blue or LAMMPS) are complicated, specialized pieces of code





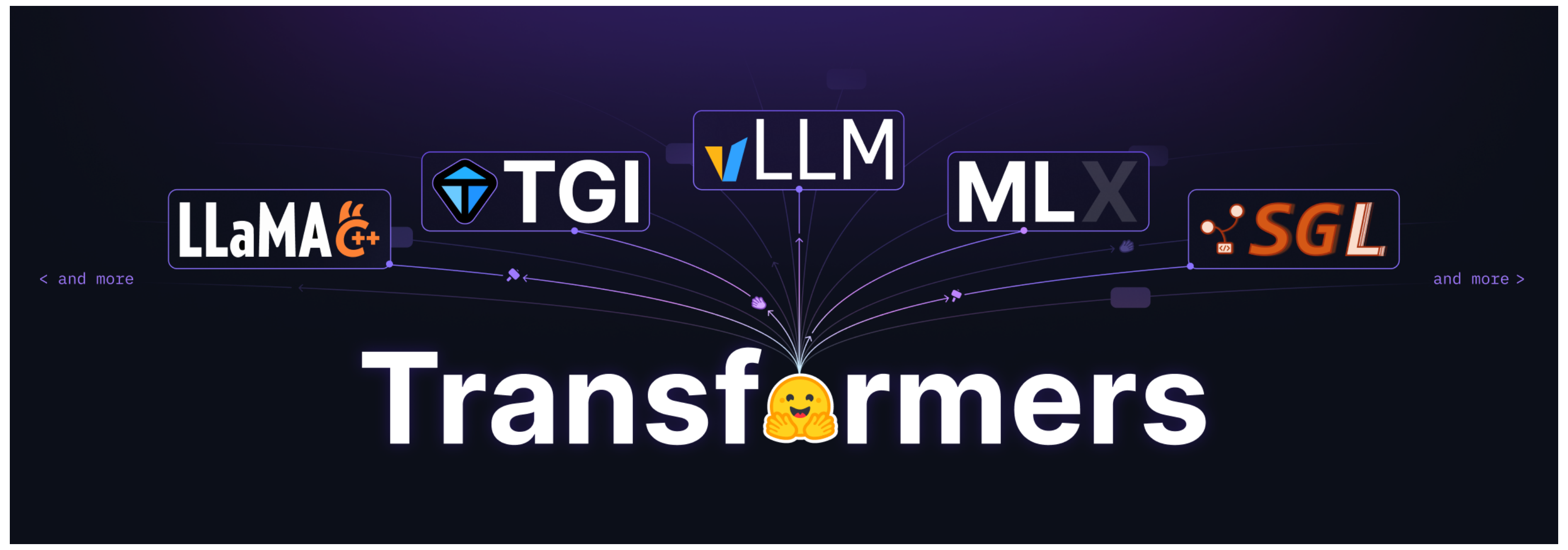
jaxspec

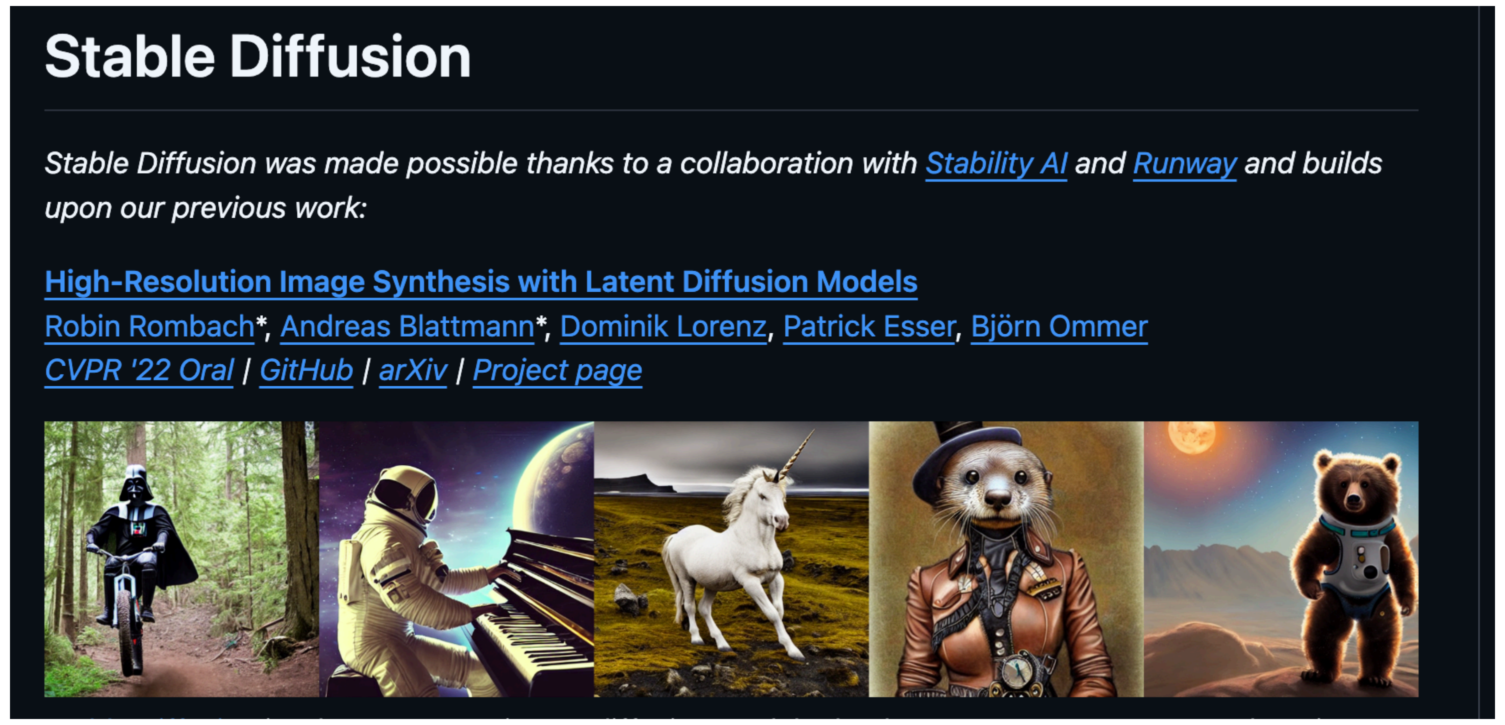
PYPI V0.3.0 | PYTHON >=3.10,<3.13 | DOCS PASSING | COVERAGE 94% | SLACK

> ⚠ **jaxspec is still in early release:** expect bugs, breaking API changes, undocumented features and lack of functionalities

`jaxspec` is an X-ray spectral fitting library built in pure Python. It can currently load an X-ray spectrum (in the OGIP standard), define a spectral model from the implemented components, and calculate the best parameters using state-of-the-art Bayesian approaches. It is built on top of JAX to provide just-in-time compilation and automatic differentiation of the spectral models, enabling the use of sampling algorithm such as NUTS.

How do we write ML Accelerator code now?

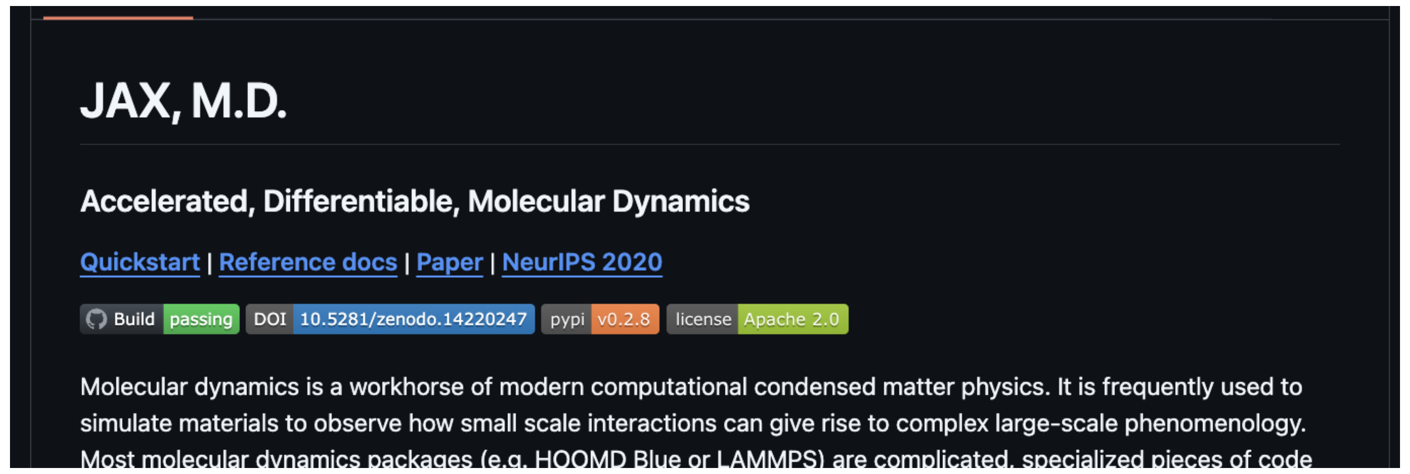




Stable Diffusion

Stable Diffusion was made possible thanks to a collaboration with Stability AI and Runway and builds upon our previous work:

High-Resolution Image Synthesis with Latent Diffusion Models
Robin Rombach*, Andreas Blattmann*, Dominik Lorenz, Patrick Esser, Björn Ommer
CVPR '22 Oral | *GitHub* | *arXiv* | *Project page*



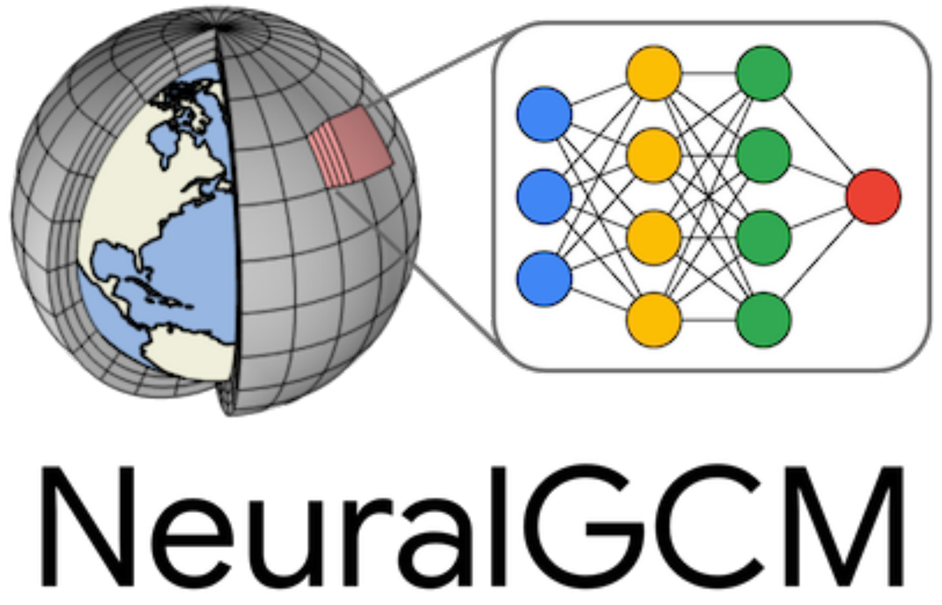
JAX, M.D.

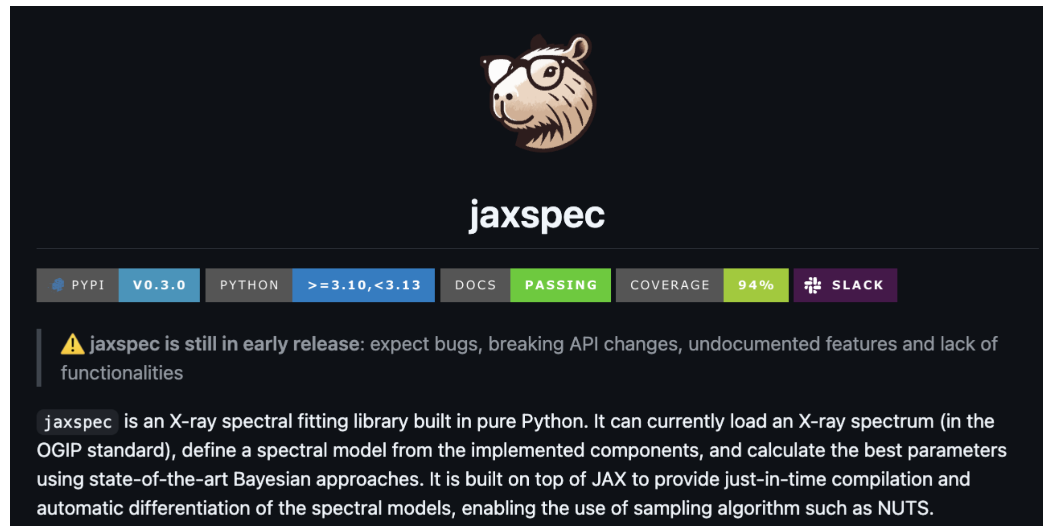
Accelerated, Differentiable, Molecular Dynamics

Quickstart | Reference docs | Paper | NeurIPS 2020

Build passing | DOI 10.5281/zenodo.14220247 | pypi v0.2.8 | license Apache 2.0

Molecular dynamics is a workhorse of modern computational condensed matter physics. It is frequently used to simulate materials to observe how small scale interactions can give rise to complex large-scale phenomenology. Most molecular dynamics packages (e.g. HOOMD Blue or LAMMPS) are complicated, specialized pieces of code





jaxspec

PYPI V0.3.0 | PYTHON >=3.10,<3.13 | DOCS PASSING | COVERAGE 94% | SLACK

> ⚠ **jaxspec is still in early release:** expect bugs, breaking API changes, undocumented features and lack of functionalities

`jaxspec` is an X-ray spectral fitting library built in pure Python. It can currently load an X-ray spectrum (in the OGIP standard), define a spectral model from the implemented components, and calculate the best parameters using state-of-the-art Bayesian approaches. It is built on top of JAX to provide just-in-time compilation and automatic differentiation of the spectral models, enabling the use of sampling algorithm such as NUTS.

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.

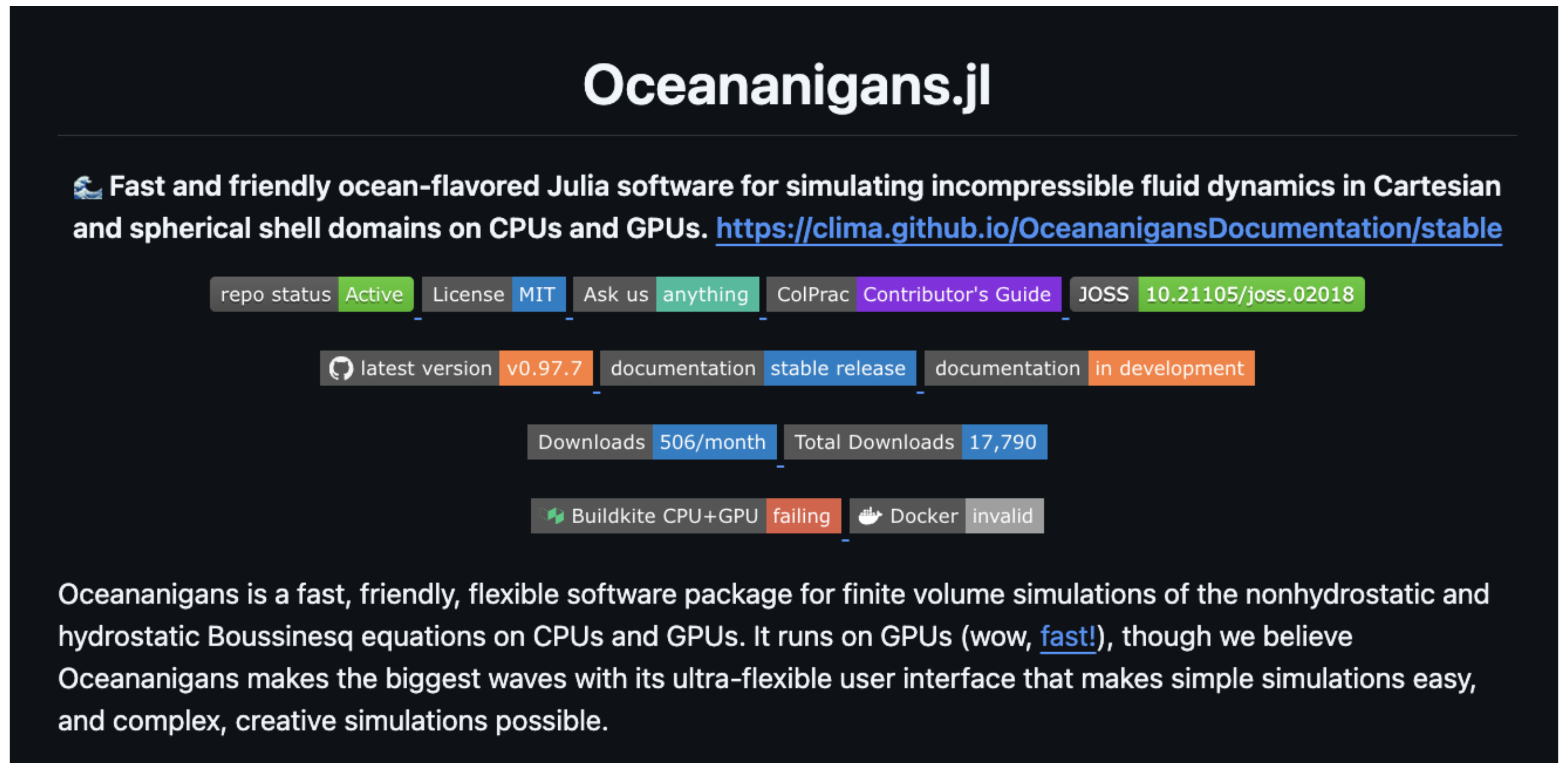
Game-changing results in a broad spectrum of science and engineering application areas.

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
```



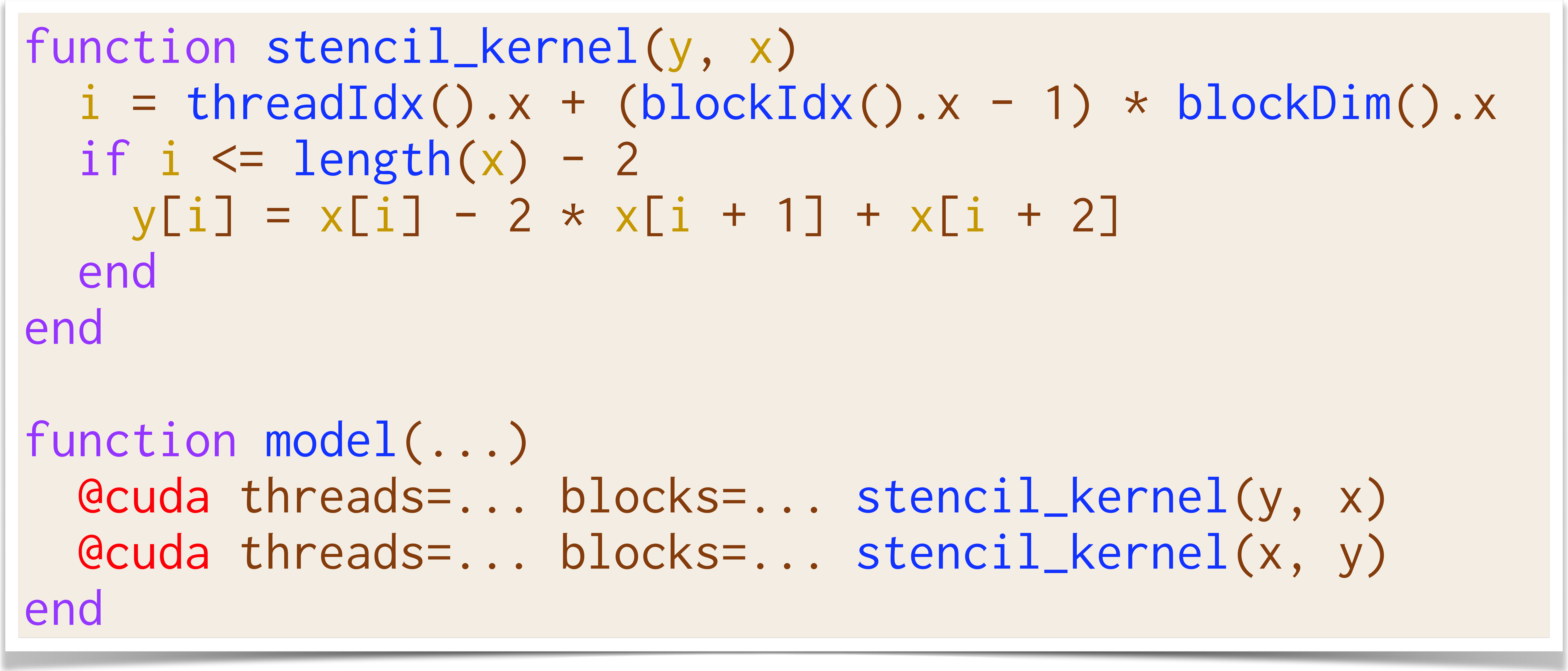
Oceananigans.jl

Fast and friendly ocean-flavored Julia software for simulating incompressible fluid dynamics in Cartesian and spherical shell domains on CPUs and GPUs. https://clima.github.io/OceananigansDocumentation/stable

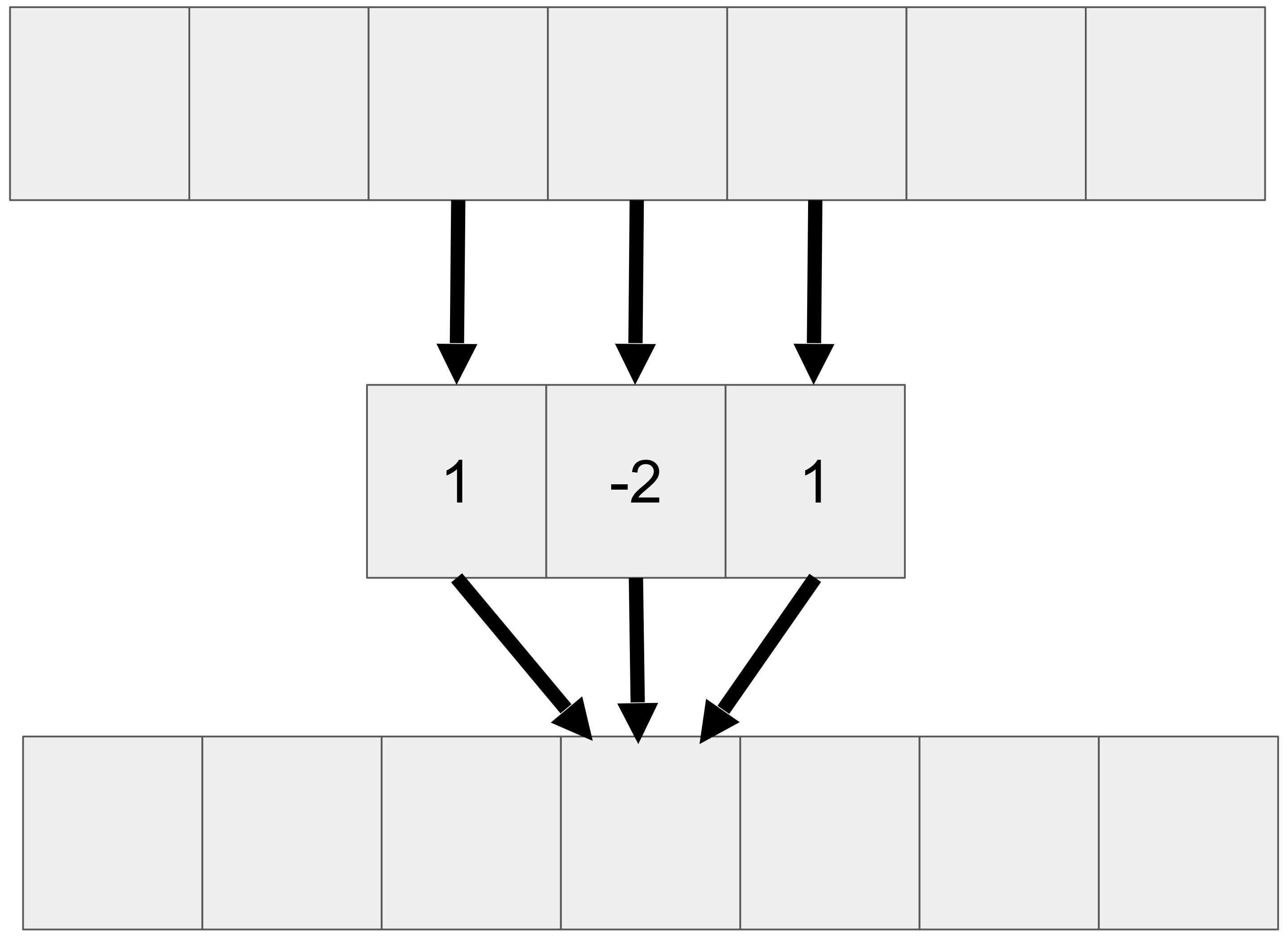
Oceananigans is a fast, friendly, flexible software package for finite volume simulations of the nonhydrostatic and hydrostatic Boussinesq equations on CPUs and GPUs. It runs on GPUs (wow, fast!), though we believe Oceananigans makes the biggest waves with its ultra-flexible user interface that makes simple simulations easy, and complex, creative simulations possible.

> 277 such kernels

Looking More Deeply at Scientific Code



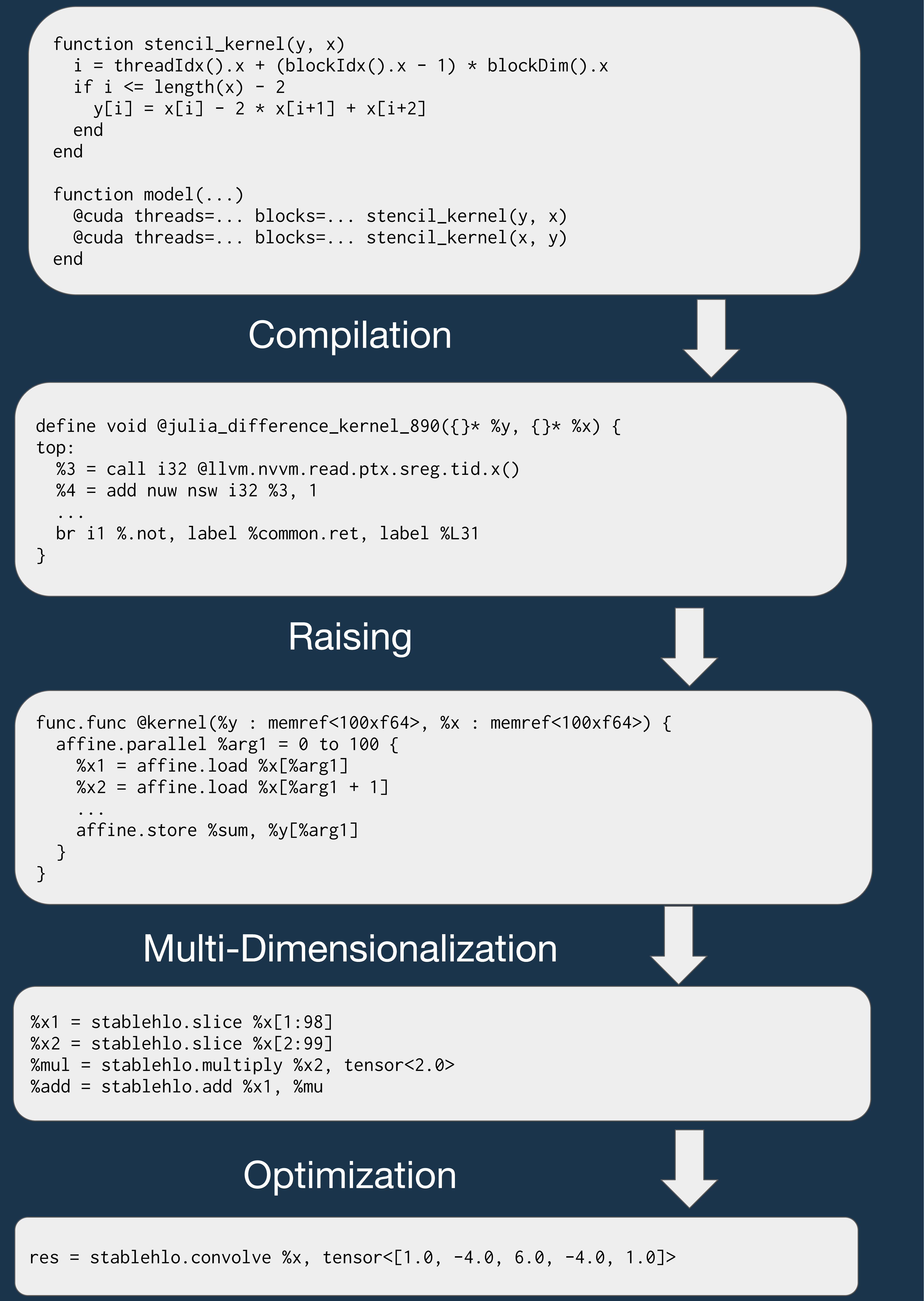
```
function stencil_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
  if i <= length(x) - 2
    y[i] = x[i] - 2 * x[i + 1] + x[i + 2]
  end
end

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

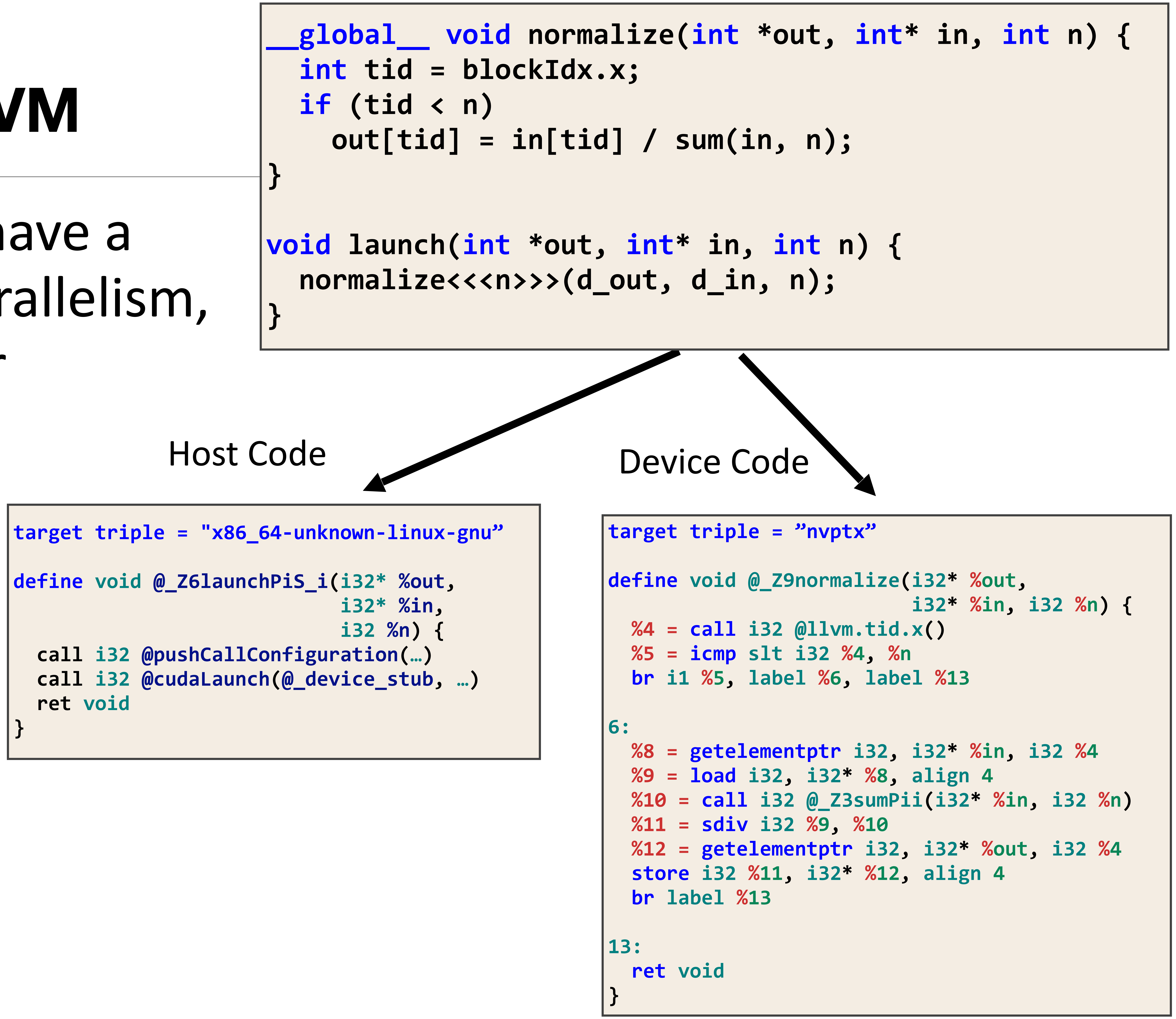
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

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

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

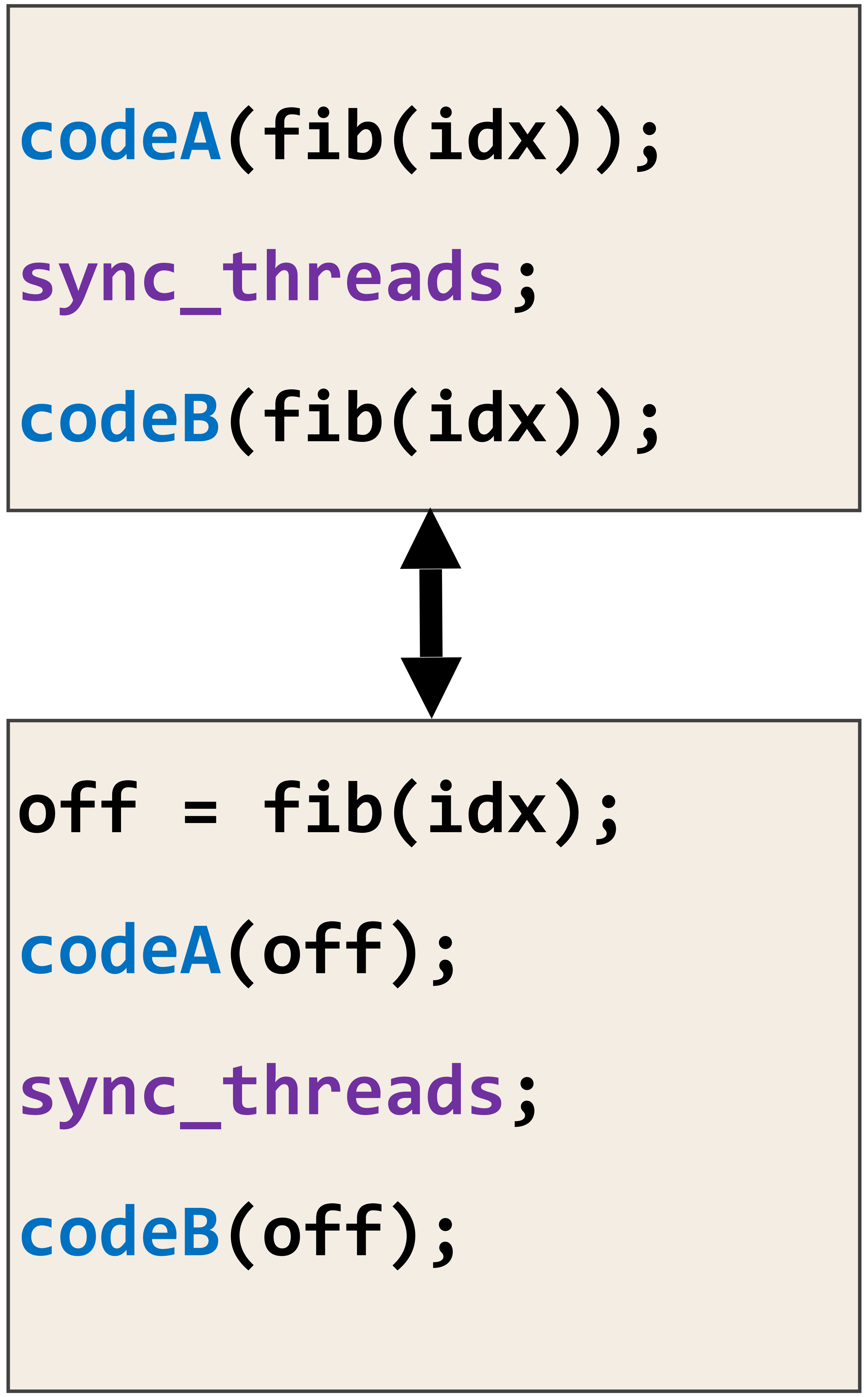
GPU Programming via MLIR

```
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);
```

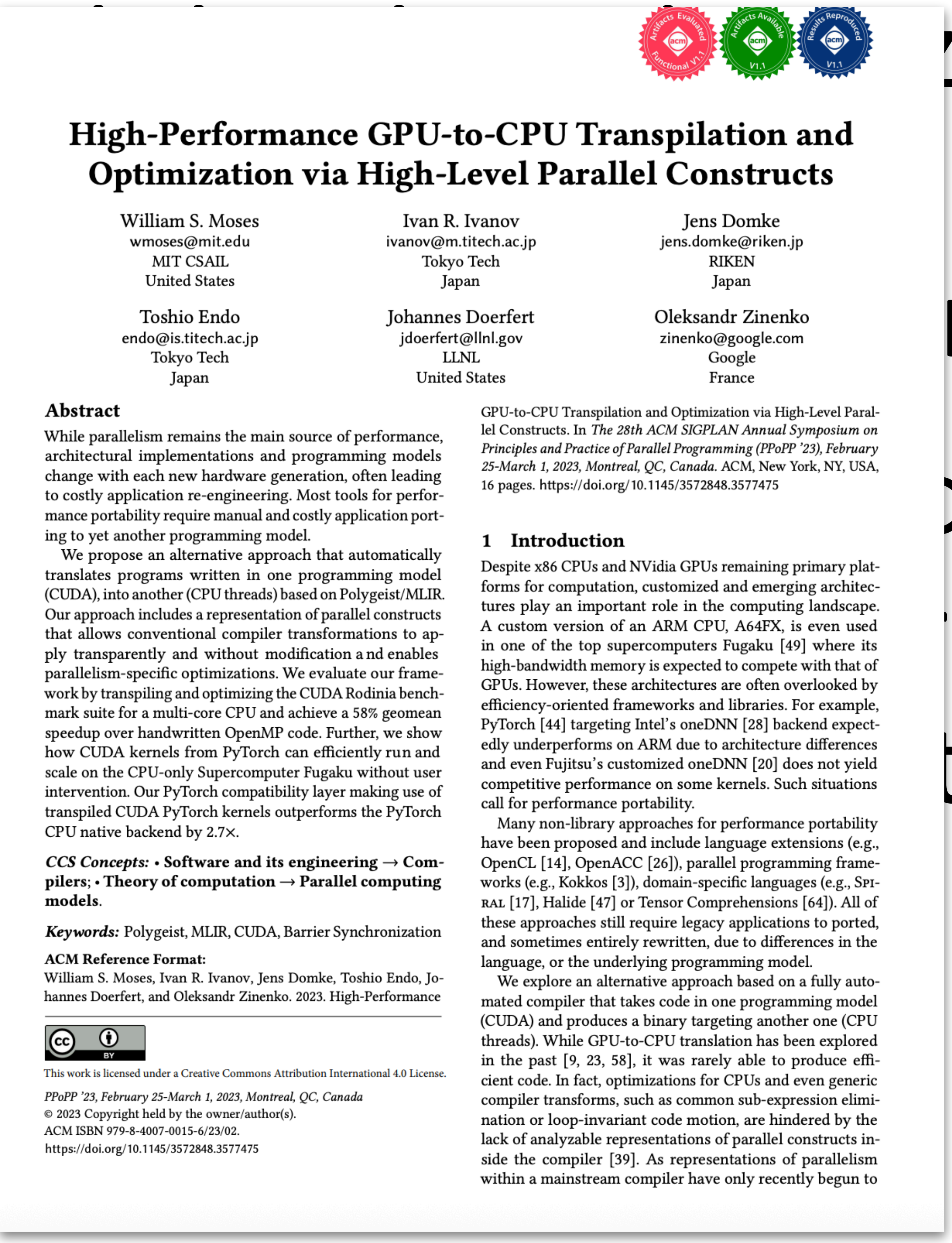
Synchronization via Memory

- 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

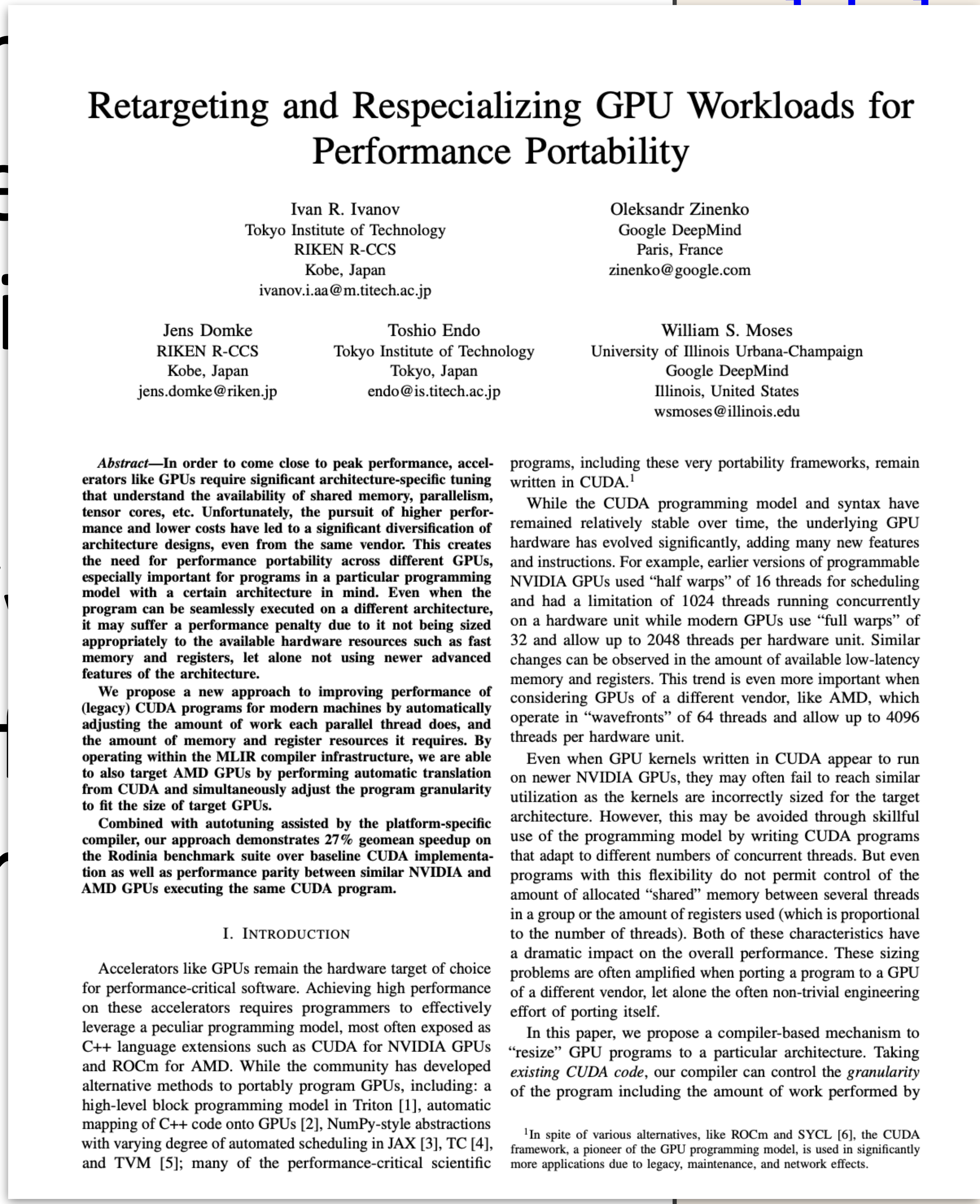
PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577475

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

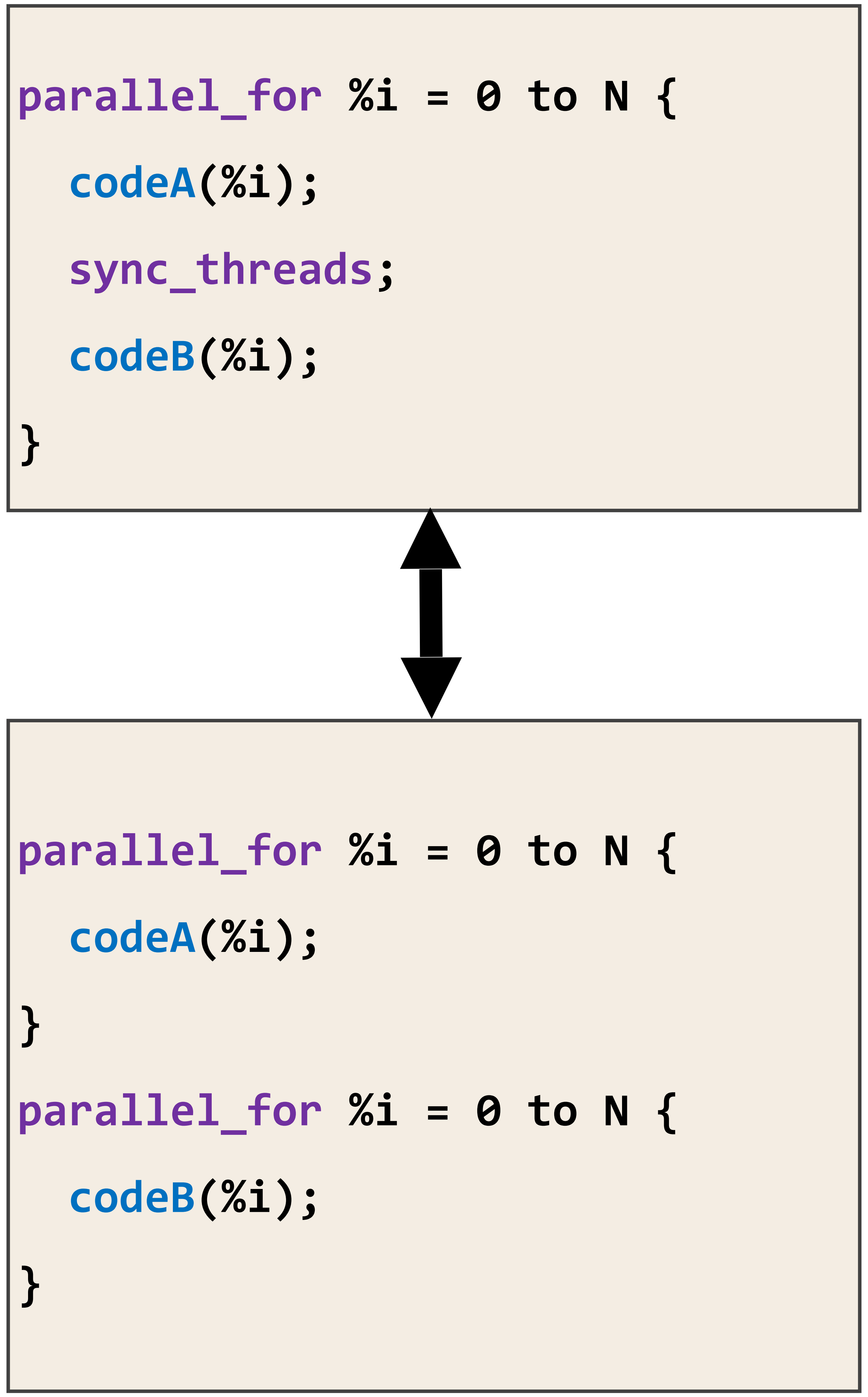
[1] In spite of various alternatives, like ROCm and SYCL [6], the CUDA framework, a pioneer of the GPU programming model, is used in significantly more applications due to legacy, maintenance, and network effects.

```
               _ 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

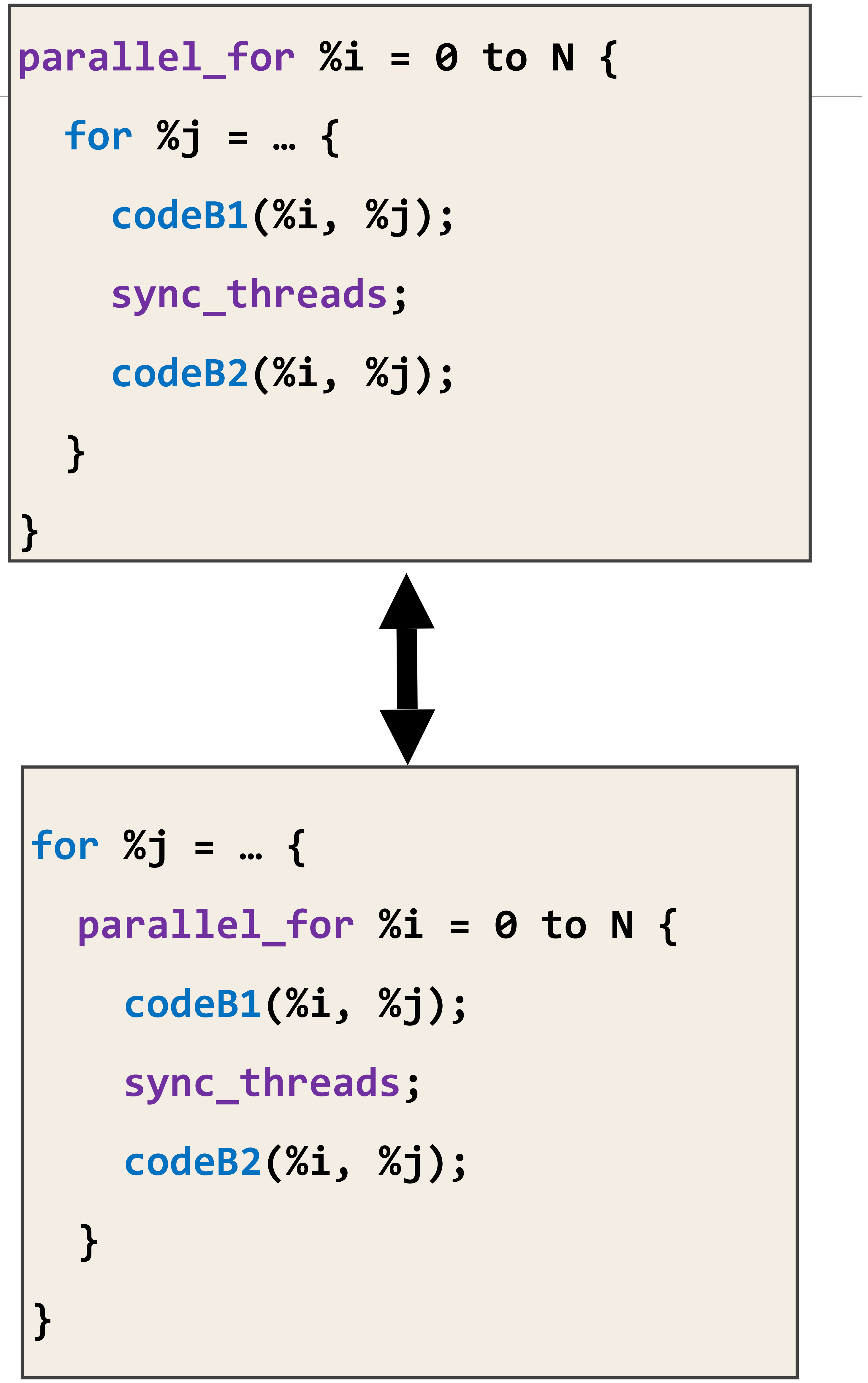


```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  codeB(%i);
}
```

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

Synchronization via Memory

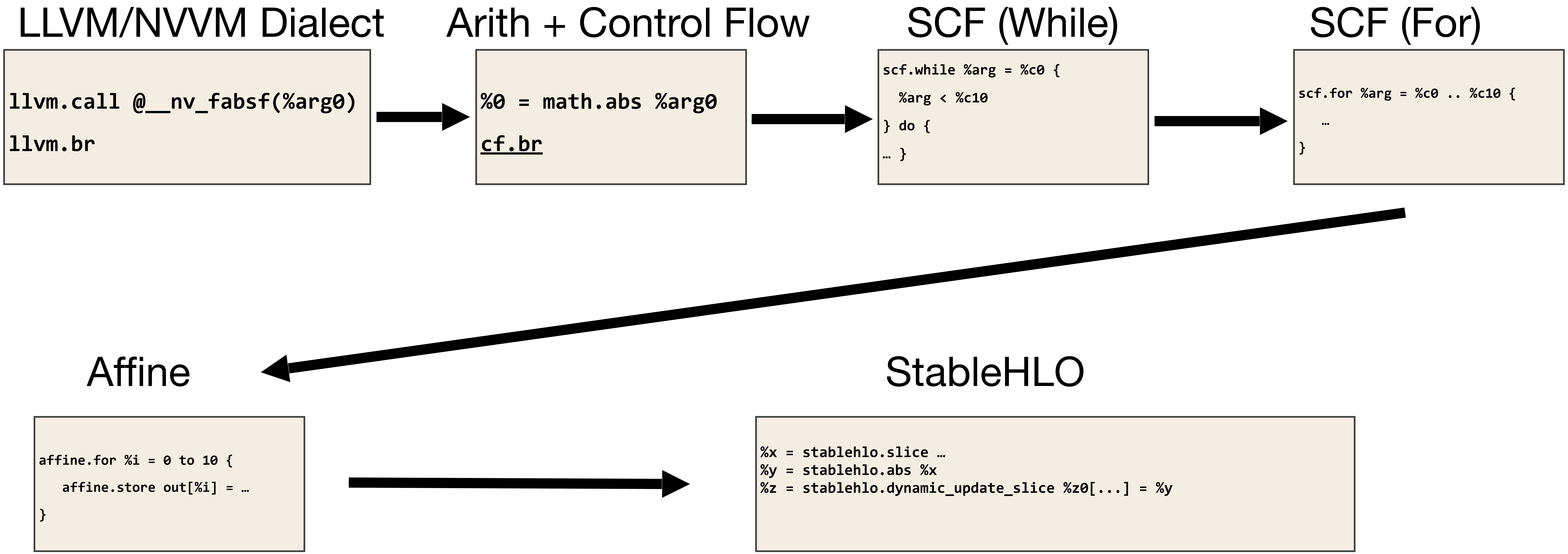
- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. historically OpenMP, now TPUs)
- Some backends do not have block synchronization
- Lower a top-level synchronization by distributing the parallel for loop around the sync,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

LLVM to StableHLO



LLVM/NVVM Dialect

```
llvm.call @__nv_fabsf(%arg0)
llvm.br
```

Arith + Control Flow

```
%0 = math.abs %arg0
cf.br
```

SCF (While)

```
scf.while %arg = %c0 {
  %arg < %c10
} do {
… }
```

SCF (For)

```
scf.for %arg = %c0 .. %c10 {
   …
}
```

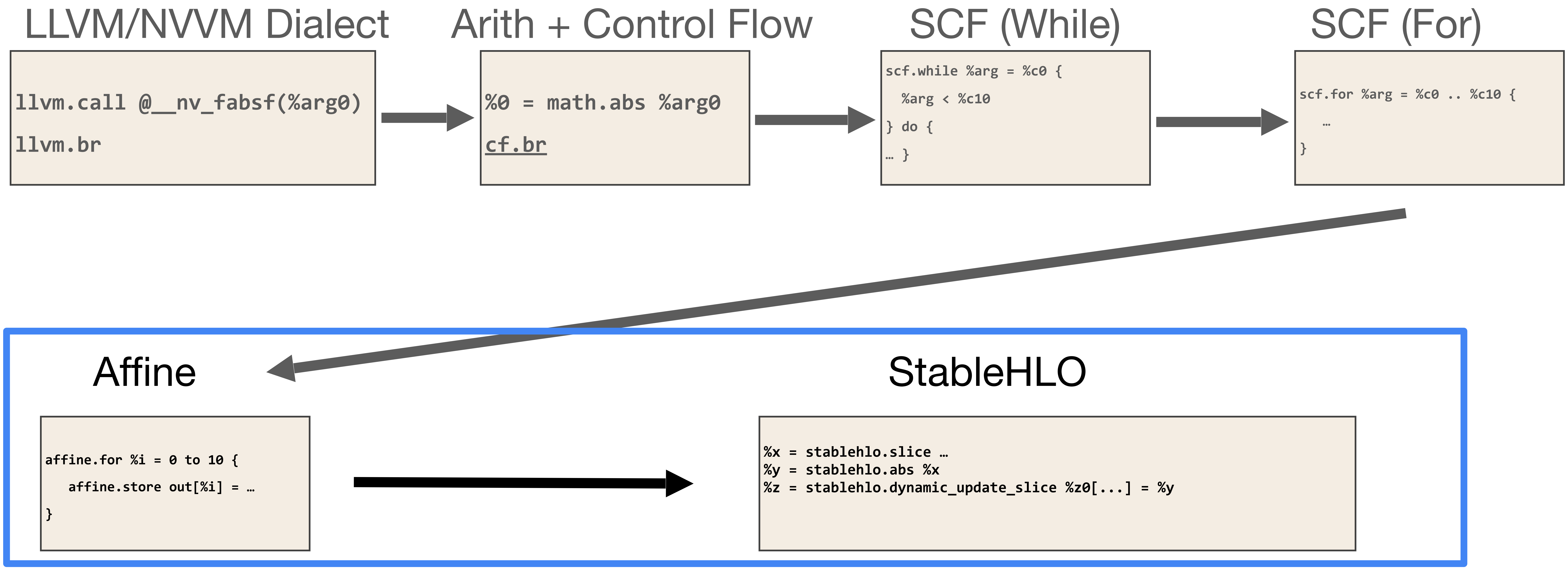
Affine

```
affine.for %i = 0 to 10 {
   affine.store out[%i] = …
}
```

StableHLO

```
%x = stablehlo.slice …
%y = stablehlo.abs %x
%z = stablehlo.dynamic_update_slice %z0[...] = %y
```

LLVM to StableHLO



LLVM/NVVM Dialect

```
llvm.call @__nv_fabsf(%arg0)
llvm.br
```

Arith + Control Flow

```
%0 = math.abs %arg0
cf.br
```

SCF (While)

```
scf.while %arg = %c0 {
  %arg < %c10
} do {
… }
```

SCF (For)

```
scf.for %arg = %c0 .. %c10 {
  …
}
```

Affine

```
affine.for %i = 0 to 10 {
   affine.store out[%i] = …
}
```

StableHLO

```
%x = stablehlo.slice …
%y = stablehlo.abs %x
%z = stablehlo.dynamic_update_slice %z0[...] = %y
```

Affine to StableHLO

- Represent ***permissive, device-agnostic parallelism***
  - Legal to re-order and interchange instructions
  - One execution (lock-step), runs all of A1, then all of A2, etc
  - Lets us form efficient tensor (stablehlo) versions of kernels

```
parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){
   %A1 = load x[%tx, %ty, %tz]
   %A2 = sin(%A1)
   store y[%tx, %ty, %tz] = %A2
   …
}
```

Affine to StableHLO

- Represent ***permissive, device-agnostic parallelism***
  - Legal to re-order and interchange instructions
  - One execution (lock-step), runs all of A1, then all of A2, etc
  - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]

parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){

   %A2 = sin(%A1)

   store y[%tx, %ty, %tz] = %A2

   …
}
```

Affine to StableHLO

- Represent ***permissive, device-agnostic parallelism***
  - Legal to re-order and interchange instructions
  - One execution (lock-step), runs all of A1, then all of A2, etc
  - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]

%A2 = stablehlo.sine %A1

parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){

   store y[%tx, %ty, %tz] = %A2

   …
}
```

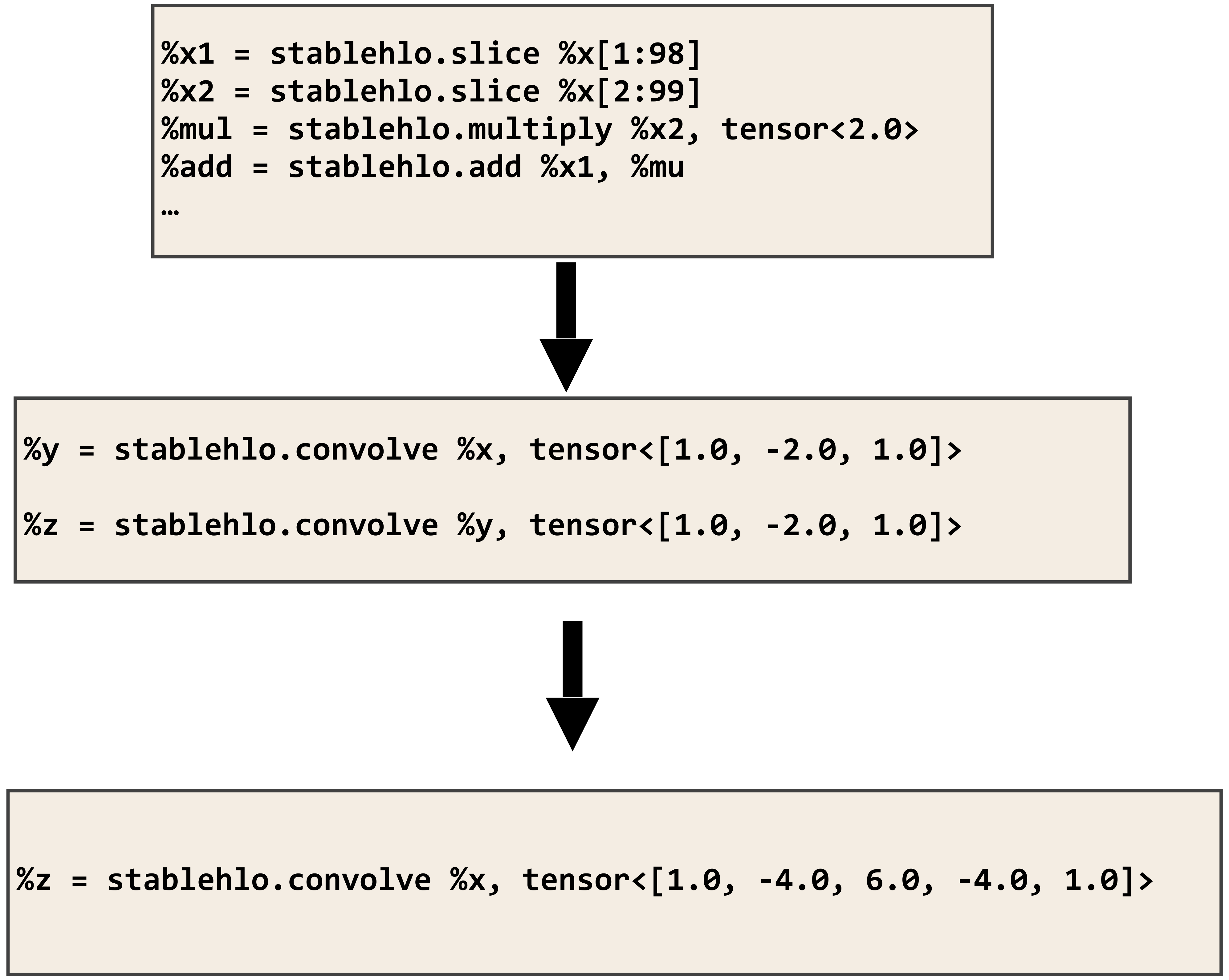
Affine to StableHLO

- Represent ***permissive, device-agnostic parallelism***
  - Legal to re-order and interchange instructions
  - One execution (lock-step), runs all of A1, then all of A2, etc
  - Lets us form efficient tensor (stablehlo) versions of kernels

```
%A1 = stablehlo.slice %x[0:5, 0:7, 0:9]

%A2 = stablehlo.sine %A1

%Y2 = stablehlo.dynamic_update_slice
                     %Y[0:5, 0:7, 0:9], %A2

parallel.for (%tx, %ty, %tz) = (0,0,0) to (5,7,9){
   …
}
```

StableHLO … to better StableHLO

- The direct vectorization of the code works, but may not be efficient.
- We will lost the convolution!
- Perform tensor-level optimizations on stablehlo to recover and optimize higher-level structures



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

```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>

%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```

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

StableHLO … to better StableHLO

- The direct vectorization of the code works, but may not be efficient.
- We will lost the convolution!
- F el optimizations c over and c el structures

Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers

ARYA VOHRA*, University of Chicago, USA
LEO SEOJUN LEE*, University of Oxford, UK
JAKUB BACHURSKI, University of Cambridge, UK
OLEKSANDR ZINENKO, Brium, France
PHITCHAYA MANGPO PHOTHILIMTHANA, OpenAI, USA
ALBERT COHEN, Google, France
WILLIAM S. MOSES, UIUC, USA

Machine learning (ML) compilers rely on graph-level transformations to enhance the runtime performance of ML models. However, performing local transformations on individual operations can create effects far beyond the location of the rewrite. In particular, a local rewrite can change the profitability or legality of hard-to-predict downstream transformations, particularly regarding data layout, parallelization, fine-grained scheduling, and memory management. As a result, program transformations are often driven by manually-tuned compiler heuristics, which are quickly rendered obsolete by new hardware and model architectures.

Instead of hand-written local heuristics, we propose the use of equality saturation. We replace such heuristics with a more robust *global* performance model, which accounts for downstream transformations. Equality saturation addresses the challenge of local optimizations inadvertently constraining or negating the benefits of subsequent transformations, thereby providing a solution that is inherently adaptable to newer workloads. While this approach still requires a global performance model to evaluate the profitability of transformations, it holds significant promise for increased automation and adaptability.

This paper addresses challenges in applying equality saturation on real-world ML compute graphs and state-of-the-art hardware. By doing so, we present an improved method for discovering effective compositions of graph optimizations. We study different cost modeling approaches to deal with fusion and layout optimization, and tackle scalability issues that arise from considering a very wide range of algebraic optimizations. We design an equality saturation pass for the XLA compiler, with an implementation in C++ and Rust. We demonstrate an average speedup of 3.45% over XLA's optimization flow across our benchmark suite on various CPU and GPU platforms, with a maximum speedup of 56.26% for NasRNN on CPU.

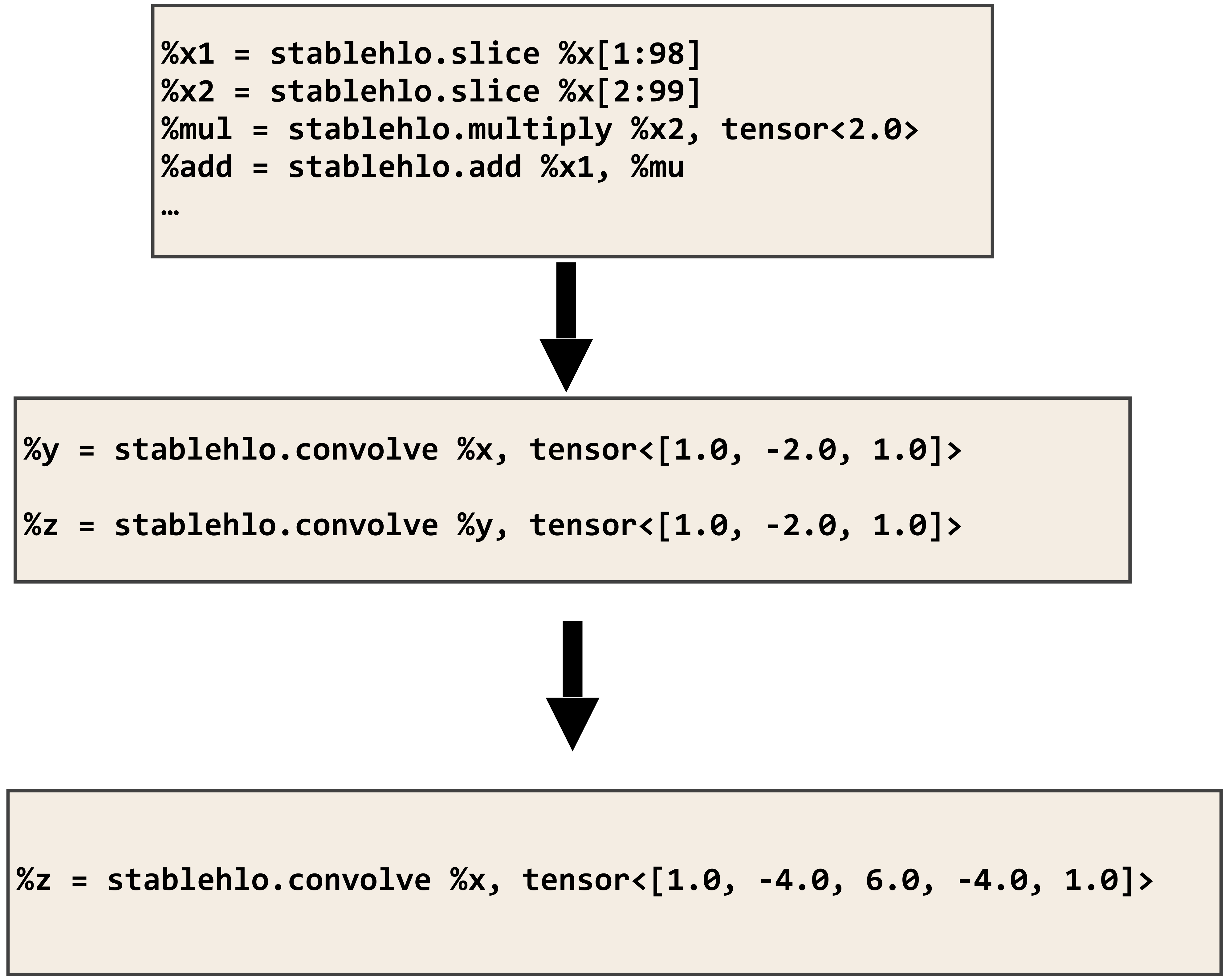
ACM Reference Format:
Arya Vohra, Leo Seojun Lee, Jakub Bachurski, Oleksandr Zinenko, Phitchaya Mangpo Phothilimthana, Albert Cohen, and William S. Moses. 2025. Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers. 1, 1 (August 2025), 28 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

*These authors contributed equally.

Authors' addresses: Arya Vohra, aryavohra@uchicago.edu, University of Chicago, USA; Leo Seojun Lee, seojun.lee@oriel.ox.ac.uk, University of Oxford, UK; Jakub Bachurski, kbachurski@gmail.com, University of Cambridge, UK; Oleksandr Zinenko, alex@brium.ai, Brium, France; Phitchaya Mangpo Phothilimthana, , OpenAI, USA; Albert Cohen, albertcohen@google.com, Google, France; William S. Moses, wsmoses@illinois.edu, UIUC, USA.

XXXX-XXXX/2025/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

56% speedup on JaX ML workloads

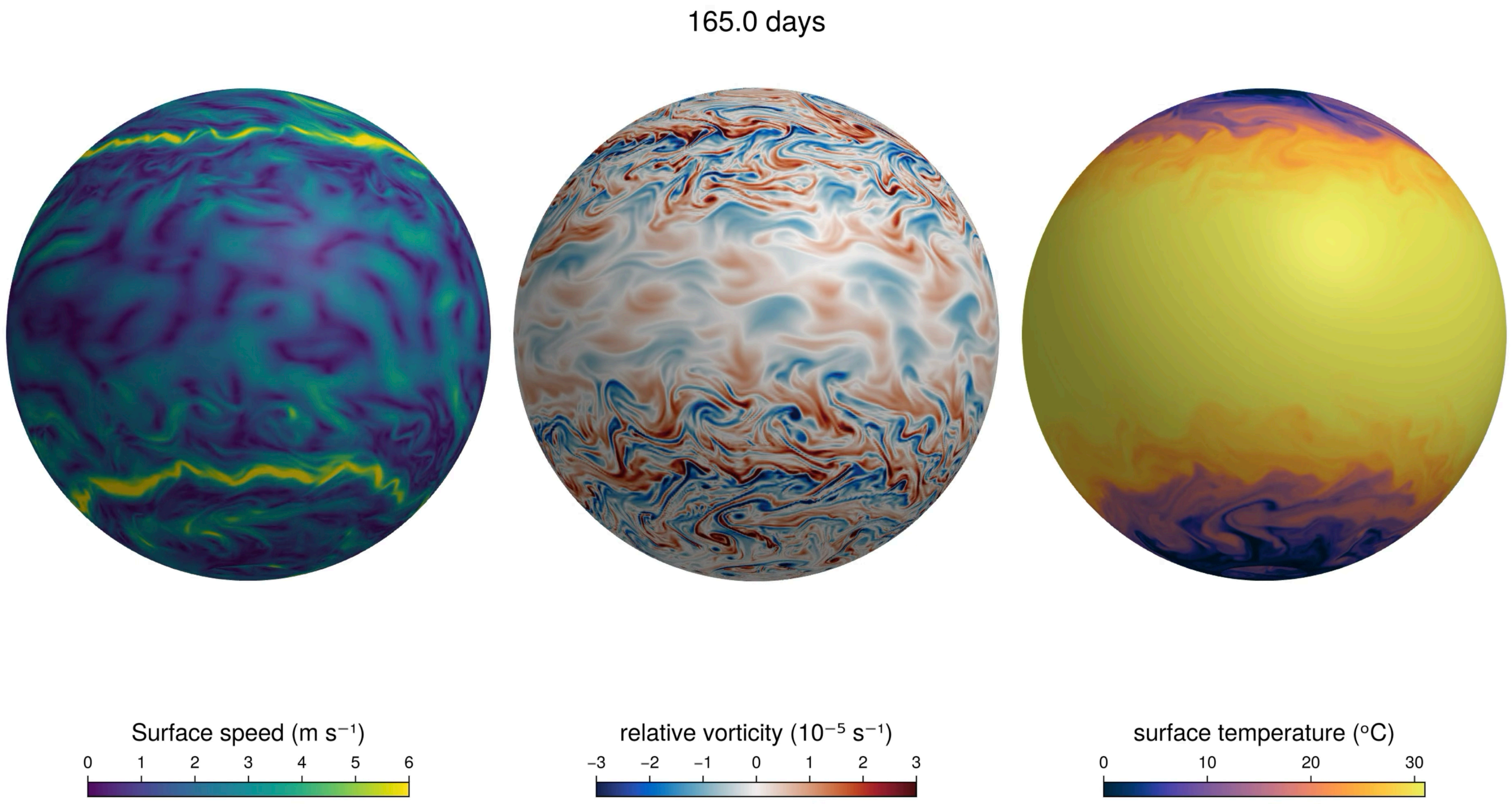


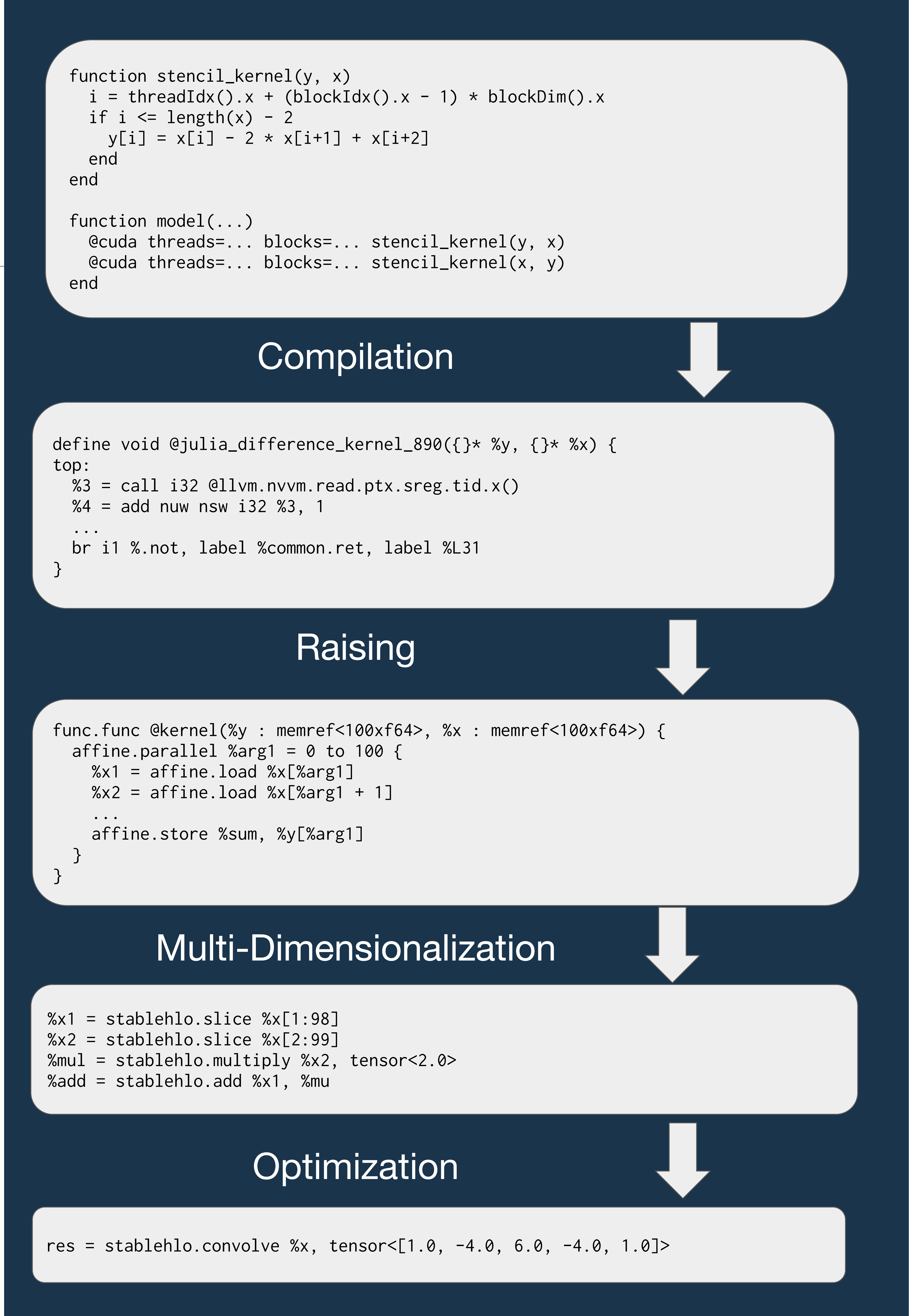
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
…
```

```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>

%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```

```
%z = 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

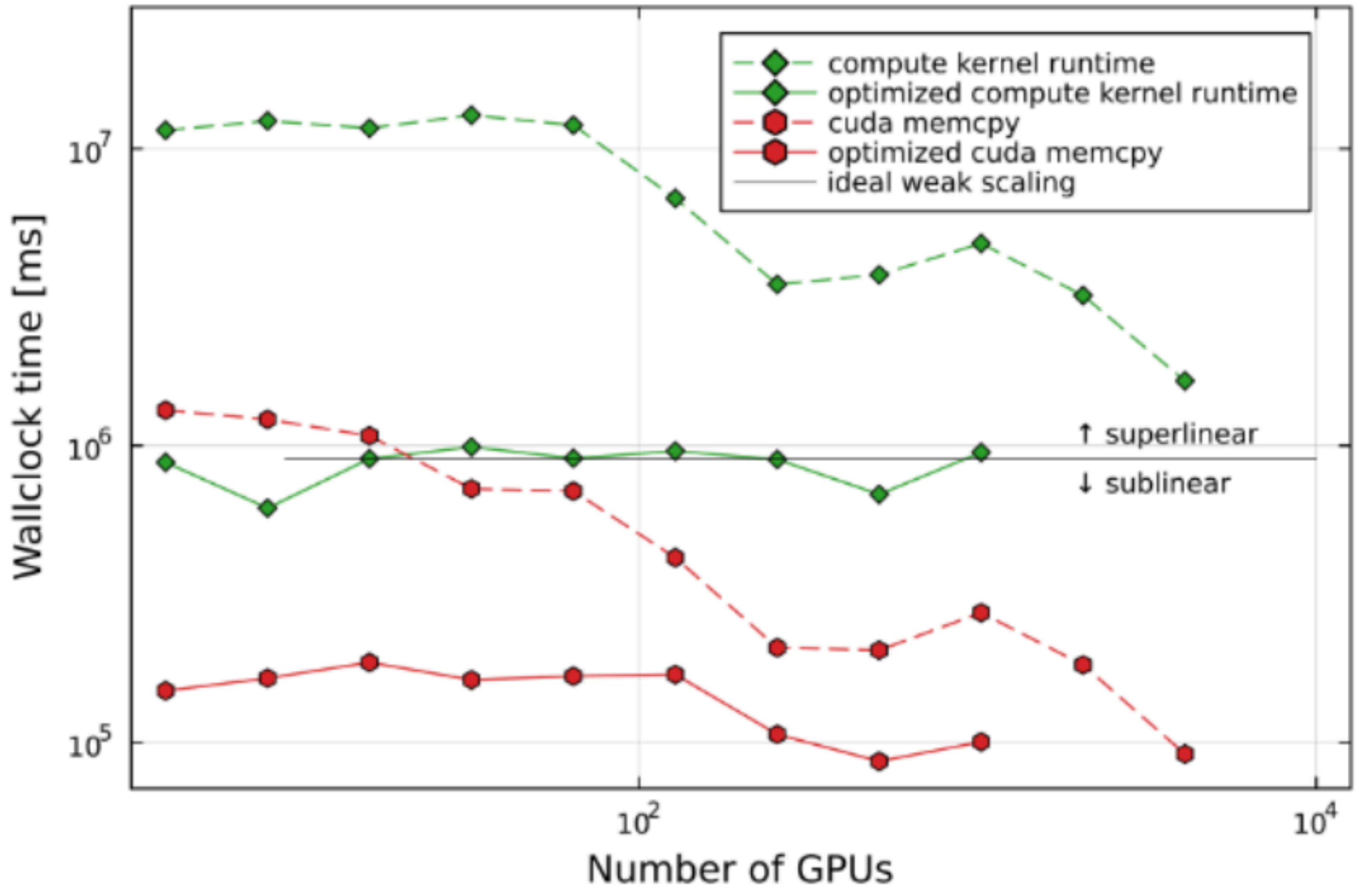
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

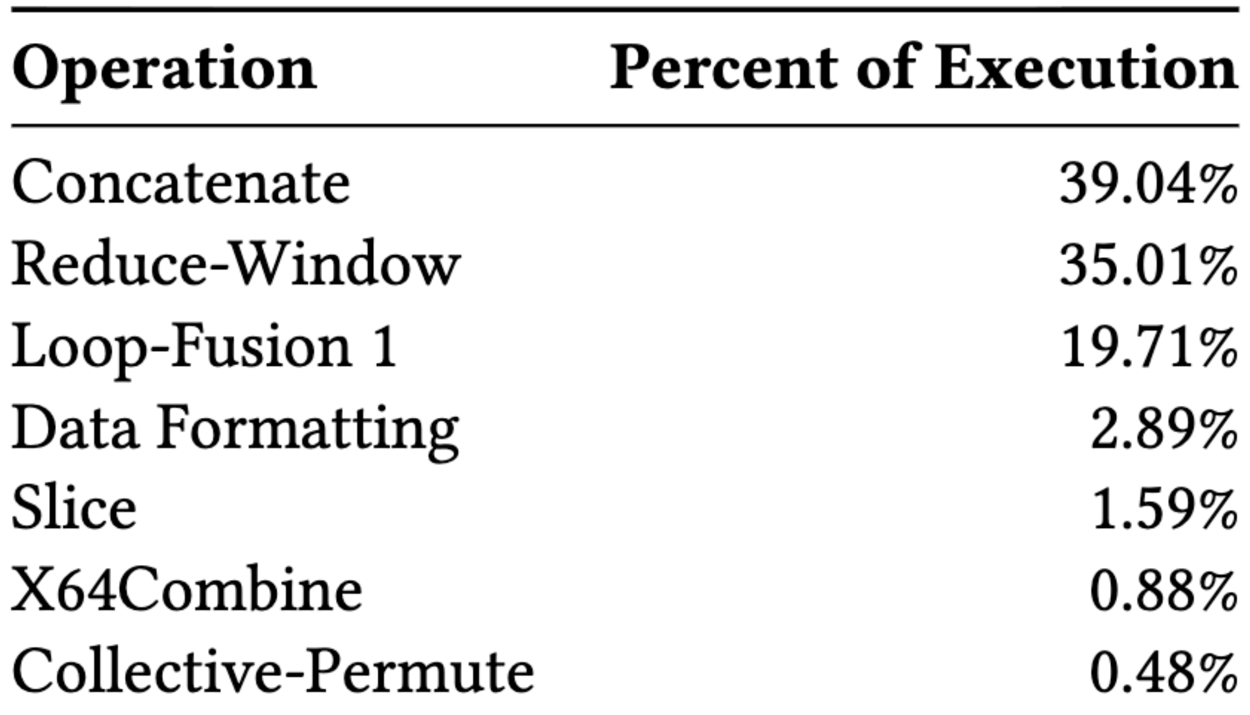
Optimization

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

Primal Raising Performance Results

- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
  - Perlmutter (1536 nodes x 4 NVIDIA A100 GPUs)
  - 1,679 Google TPUs v6e (918 TFLOPS each)
- Communication optimizations were key
- Good Single-Node Perf (CPU)
  - Vanilla Model: 272.0seconds
  - Tensor Optims: 11.5seconds





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%

Table 1: Breakdown of TPU execution time by operation type, on a single node 4-TPU machine.

How Does Raising & Tensor Transformations Impact AD?

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

William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hückelheim †,
Sri Hari Krishna Narayanan †, Michel Schanen †, Johannes Doerfert†
{wmoses,vchuravy}@mit.edu,ludger.paehler@tum.de,{jhuckelheim,snarayan,mschanen,jdoerfert}@anl.gov
MIT CSAIL *, Technical University of Munich §, Argonne National Laboratory†

ABSTRACT

Computing derivatives is key to many algorithms in scientific computing and machine learning such as optimization, uncertainty quantification, and stability analysis. Enzyme is a LLVM compiler plugin that performs reverse-mode automatic differentiation (AD) and thus generates high performance gradients of programs in languages including C/C++, Fortran, Julia, and Rust. Prior to this work, Enzyme and other AD tools were not capable of generating gradients of GPU kernels. Our paper presents a combination of novel techniques that make Enzyme the first fully automatic reverse-mode AD tool to generate gradients of GPU kernels. Since unlike other tools Enzyme performs automatic differentiation within a general-purpose compiler, we are able to introduce several novel GPU and AD-specific optimizations. To show the generality and efficiency of our approach, we compute gradients of five GPU-based HPC applications, executed on NVIDIA and AMD GPUs. All benchmarks run within an order of magnitude of the original program's execution time. Without GPU and AD-specific optimizations, gradients of GPU kernels either fail to run from a lack of resources or have infeasible overhead. Finally, we demonstrate that increasir
the problem size by either increasing the number of threads
increasing the work per thread, does not substantially impact tl
overhead from differentiation.

CCS CONCEPTS

• Mathematics of computing → Automatic differentiatio
• Software and its engineering → Source code generation;
Theory of computation → *Parallel computing models*; *Shar*
memory algorithms; • Computing methodologies → *Machi*
learning.

KEYWORDS

Automatic Differentiation, AD, CUDA, ROCm, GPU, LLVM, HP(

ACM Reference Format:
William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hüc
elheim †,, Sri Hari Krishna Narayanan †, Michel Schanen †, Johann
Doerfert†. 2021. Reverse-Mode Automatic Differentiation and Optimiz
tion of GPU Kernels via Enzyme. In *The International Conference for Hi*

Publication rights licensed to ACM. ACM acknowledges that this contribution w
authored or co-authored by an employee, contractor or affiliate of the United Stat
government. As such, the Government retains a nonexclusive, royalty-free right
publish or reproduce this article, or to allow others to do so, for Government purpos
only.
SC '21, November 14-19, 2021, St. Louis, MO, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM
ACM ISBN 978-1-4503-8442-1/21/11...$15.00
https://doi.org/10.1145/3458817.3476165

```
void init(double* ar, int N, double val) {
  parallel_for(int i=0; i<N ; i++)
    // Concurrent reads of val
    ar[i] = val;
}

double gradient_init(double* ar, double* d_ar,
                     int N, double val) {
  double d_val = 0.0;
  parallel_for(int i=0; i<N ; i++)
    ar[i] = val;
  parallel_for(int i=0; i<N ; i++) {
    // Concurrent writes to d_val
    d_val += d_ar[i];                ↯ race ↯
    d_ar[i] = 0.0;
  }
  return d_val;
}
```

Figure 1: A parallel initialize function (top) with a naive reverse mode AD gradient function (bottom) that does not

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

William S. Moses
MIT CSAIL
Cambridge, MA
wmoses@mit.edu

Sri Hari Krishna Narayanan
Argonne National Laboratory
Lemont, IL
snarayan@anl.gov

Ludger Paehler
Technical University of Munich
Munich, Germany
ludger.paehler@tum.de

Valentin Churavy
MIT CSAIL
Cambridge, MA
vchuravy@mit.edu

Michel Schanen
Argonne National Laboratory
Lemont, IL
mschanen@anl.gov

Jan Hückelheim
Argonne National Laboratory
Lemont, IL
jhuckelheim@anl.gov

Johannes Doerfert
Argonne National Laboratory
Lemont, IL
jdoerfert@anl.gov

Paul Hovland
Argonne National Laboratory
Lemont, IL
hovland@mcs.anl.gov

***Abstract*—Derivatives are key to numerous science, engineering, and machine learning applications. While existing tools generate derivatives of programs in a single language, modern parallel applications combine a set of frameworks and languages to leverage available performance and function in an evolving hardware landscape.**

We propose a scheme for differentiating arbitrary DAG-based parallelism that preserves scalability and efficiency, implemented into the LLVM-based Enzyme automatic differentiation framework. By integrating with a full-fledged compiler backend, Enzyme can differentiate numerous parallel frameworks and directly control code generation. Combined with its ability to differentiate any LLVM-based language, this flexibility permits Enzyme to leverage the compiler tool chain for parallel and differentiation-specific optimizations.

We differentiate nine distinct versions of the LULESH and miniBUDE applications, written in different programming lan-
ks (OpenMP, MPI,
imilar scalability to
54 threads or nodes,
6.8× **on C++ and**

PI, OpenMP, Tasks,
arallel, distributed,

lern applications in
arning [1], gradient-
s [4], and computer
AD) is a method
s of mathematical
ams. AD is able to
e precision, unlike

f frameworks, has
ale computing and
nly involves using
e Message Passing
ed parallelism, or
nemory parallelism,
RAJA [9].

In addition to being difficult to create any derivatives of parallel programs, it is desirable to preserve the original program's parallelism for the accumulation of derivatives. This is not always straightforward, particularly in the so-called reverse-mode AD or the closely related back-propagation [10]–[15], which will be briefly explained in Section III.

This paper demonstrates how using a common low-level compiler infrastructure to synthesize adjoints of parallel codes enables differentiation across a wide variety of parallel models and source languages. To this end, we extend the Enzyme automatic differentiation framework [16], which already supports synthesizing adjoints of GPU kernels [17] to arbitrary parallel frameworks representable as a directed acyclic graph (DAG) of dependencies. To showcase the generality of our approach, we differentiate MPI (distributed parallelism), OpenMP (multicore parallelism), Julia Tasks (multicore parallelism within a JIT), and describe how additional frameworks can be supported by simply marking the parallelism.

By enabling support for the underlying programming models within the compiler, we are able to differentiate any parallel framework built on top of them such as RAJA (running atop OpenMP and MPI) and MPI.jl (Julia bindings for MPI). Moreover, we demonstrate that differentiating low-level parallelism concepts such as shared and thread-local memory automatically yields support for higher-level primitives such as reductions or `firstprivate` variables. Finally, we showcase how jointly supporting these parallelism models in one tool naturally enables differentiation of hybrid parallel programs, and that deep integration of AD into the compiler enables performance optimizations usually only available in domain-specific/functional programming languages. Overall, our paper makes the following contributions:

- An extension to the theory of reverse-mode differentiation of single-static-assignment (SSA) intermediate representations to handle parallel execution of instructions, and thus differentiation of parallel languages and constructs that lower to such a representation.
- A demonstration of how implementing this model within the

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

William S. Moses
MIT CSAIL
`wmoses@mit.edu`

Valentin Churavy
MIT CSAIL
`vchuravy@mit.edu`

Abstract

Applying differentiable programming techniques and machine learning algorithms to foreign programs requires developers to either rewrite their code in a machine learning framework, or otherwise provide derivatives of the foreign code. This paper presents Enzyme[1], a high-performance automatic differentiation (AD) compiler plugin for the LLVM compiler framework capable of synthesizing gradients of statically analyzable programs expressed in the LLVM intermediate representation (IR). Enzyme synthesizes gradients for programs written in any language whose compiler targets LLVM IR including C, C++, Fortran, Julia, Rust, Swift, MLIR, etc., thereby providing native AD capabilities in these languages. Unlike traditional source-to-source and operator-overloading tools, Enzyme performs AD on optimized IR. On a machine-learning focused benchmark suite including Microsoft's ADBench, AD on optimized IR achieves a geometric mean speedup of 4.2 times over AD on IR before optimization allowing Enzyme to achieve state-of-the-art performance. Packaging Enzyme for PyTorch and TensorFlow provides convenient access to gradients of foreign code with state-of-the-art performance, enabling foreign code to be directly incorporated into existing machine learning workflows.

1 Introduction

Machine learning (ML) frameworks such as PyTorch [48] and TensorFlow [1] have become widespread as the primary workhorses of the modern ML community. Computing gradients necessary for algorithms such as backpropagation [32], Bayesian inference, uncertainty quantification [60], and probabilistic programming [16] requires all of the code being differentiated to be written in these frameworks. This is problematic for applying ML to new domains as existing tools like physics simulators [23, 10, 17, 18, 35], game engines, and climate models [58] are not written in the domain specific languages (DSL's) of ML frameworks. The rewriting required has been identified as the quintessential challenge of applying ML to scientific computing [4]. As stated by Rackauckas [50] "this is [the key challenge of scientific ML] because, if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train."

To remedy this issue, the trend has been to either create new DSL's [35, 17, 43] that make the rewriting process easier or to add differentiation as a first-class construct in programming languages [44, 9, 61, 37]. This results in efficient gradients, but still requires rewriting in either the DSL or the differentiable programming language. Developers may want to use code foreign to a ML framework to either re-use existing tools or write loss functions in a language with an easier abstraction for their use case. While there exist reverse-mode automatic differentiation (AD) frameworks for various languages, using them automatically on foreign code for an ML framework is difficult as they still require rewriting and have limited support for cross-language AD and libraries[61, 33, 30, 36]. The two primary approaches to computing gradients are as follows.

[1] Code and documentation at `https://github.com/wsmoses/Enzyme` and `https://enzyme.mit.edu`.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

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

William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hückelheim †,
Sri Hari Krishna Narayanan †, Michel Schanen †, Johannes Doerfert†
{wmoses,vchuravy}@mit.edu,ludger.paehler@tum.de,{jhuckelheim,snarayan,mschanen,jdoerfert}@anl.gov
MIT CSAIL *, Technical University of Munich §, Argonne National Laboratory†

ABSTRACT

Computing derivatives is key to many algorithms in scientific computing and machine learning such as optimization, uncertainty quantification, and stability analysis. Enzyme is a LLVM compiler plugin that performs reverse-mode automatic differentiation (AD) and thus generates high performance gradients of programs in languages including C/C++, Fortran, Julia, and Rust. Prior to this work, Enzyme and other AD tools were not capable of generating gradients of GPU kernels. Our paper presents a combination of novel techniques that make Enzyme the first fully automatic reverse-mode AD tool to generate gradients of GPU kernels. Since unlike other tools Enzyme performs automatic differentiation within a general-purpose compiler, we are able to introduce several novel GPU and AD-specific optimizations. To show the generality and efficiency of our approach, we compute gradients of five GPU-based HPC applications, executed on NVIDIA and AMD GPUs. All benchmarks run within an order of magnitude of the original program's execution time. Without GPU and AD-specific optimizations, gradients of GPU kernels either fail to run from a lack of resources or have infeasible overhead. Finally, we demonstrate that increasin
the problem size by either increasing the number of threads
increasing the work per thread, does not substantially impact th
overhead from differentiation.

CCS CONCEPTS

• **Mathematics of computing** → **Automatic differentiatio**
• **Software and its engineering** → **Source code generation**
Theory of computation → *Parallel computing models*; *Shar*
memory algorithms; • **Computing methodologies** → *Machi*
learning.

KEYWORDS

Automatic Differentiation, AD, CUDA, ROCm, GPU, LLVM, HP

ACM Reference Format:

William S. Moses*, Valentin Churavy *, Ludger Paehler§, Jan Hüc
elheim †,, Sri Hari Krishna Narayanan †, Michel Schanen †, Johann
Doerfert†. 2021. Reverse-Mode Automatic Differentiation and Optimiz
tion of GPU Kernels via Enzyme. In *The International Conference for Hi*

Publication rights licensed to ACM. ACM acknowledges that this contribution w
authored or co-authored by an employee, contractor or affiliate of the United Stat
government. As such, the Government retains a nonexclusive, royalty-free right
publish or reproduce this article, or to allow others to do so, for Government purpos
only.
SC '21, November 14-19, 2021, St. Louis, MO, USA
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM
ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476165

```
void init(double* ar, int N, double val) {
  parallel_for(int i=0; i<N ; i++)
    // Concurrent reads of val
    ar[i] = val;
}

double gradient_init(double* ar, double* d_ar,
                     int N, double val) {
  double d_val = 0.0;
  parallel_for(int i=0; i<N ; i++)
    ar[i] = val;
  parallel_for(int i=0; i<N ; i++) {
    // Concurrent writes to d_val
    d_val += d_ar[i];                    ⚡ race ⚡
    d_ar[i] = 0.0;
  }
  return d_val;
}
```

Figure 1: A parallel initialize function (top) with a naive reverse mode AD gradient function (bottom) that does not

Scalable Automatic Differentiation of Multiple Parallel Paradigms through Compiler Augmentation

William S. Moses *MIT CSAIL* Cambridge, MA wmoses@mit.edu	Sri Hari Krishna Narayanan *Argonne National Laboratory* Lemont, IL snarayan@anl.gov	Ludger Paehler *Technical University of Munich* Munich, Germany ludger.paehler@tum.de	Valentin Churavy *MIT CSAIL* Cambridge, MA vchuravy@mit.edu
Michel Schanen *Argonne National Laboratory* Lemont, IL mschanen@anl.gov	Jan Hückelheim *Argonne National Laboratory* Lemont, IL jhuckelheim@anl.gov	Johannes Doerfert *Argonne National Laboratory* Lemont, IL jdoerfert@anl.gov	Paul Hovland *Argonne National Laboratory* Lemont, IL hovland@mcs.anl.gov

***Abstract*—Derivatives are key to numerous science, engineering, and machine learning applications. While existing tools generate derivatives of programs in a single language, modern parallel applications combine a set of frameworks and languages to leverage available performance and function in an evolving hardware landscape.**

We propose a scheme for differentiating arbitrary DAG-based parallelism that preserves scalability and efficiency, implemented into the LLVM-based Enzyme automatic differentiation framework. By integrating with a full-fledged compiler backend, Enzyme can differentiate numerous parallel frameworks and directly control code generation. Combined with its ability to differentiate any LLVM-based language, this flexibility permits Enzyme to leverage the compiler tool chain for parallel and differentiation-specific optimizations.

We differentiate nine distinct versions of the LULESH and miniBUDE applications, written in different programming lan-
ks (OpenMP, MPI,
imilar scalability to
54 threads or nodes,
6.8× **on C++ and**

PI, OpenMP, Tasks,
arallel, distributed,

lern applications in
arning [1], gradient-
s [4], and computer
AD) is a method
s of mathematical
ams. AD is able to
e precision, unlike

f frameworks, has
ale computing and
only involves using
e Message Passing
ed parallelism, or
nemory parallelism,
RAJA [9].

In addition to being difficult to create any derivatives of parallel programs, it is desirable to preserve the original program's parallelism for the accumulation of derivatives. This is not always straightforward, particularly in the so-called reverse-mode AD or the closely related back-propagation [10]–[15], which will be briefly explained in Section III.

This paper demonstrates how using a common low-level compiler infrastructure to synthesize adjoints of parallel codes enables differentiation across a wide variety of parallel models and source languages. To this end, we extend the Enzyme automatic differentiation framework [16], which already supports synthesizing adjoints of GPU kernels [17] to arbitrary parallel frameworks representable as a directed acyclic graph (DAG) of dependencies. To showcase the generality of our approach, we differentiate MPI (distributed parallelism), OpenMP (multicore parallelism), Julia Tasks (multicore parallelism within a JIT), and describe how additional frameworks can be supported by simply marking the parallelism.

By enabling support for the underlying programming models within the compiler, we are able to differentiate any parallel framework built on top of them such as RAJA (running atop OpenMP and MPI) and MPI.jl (Julia bindings for MPI). Moreover, we demonstrate that differentiating low-level parallelism concepts such as shared and thread-local memory automatically yields support for higher-level primitives such as reductions or `firstprivate` variables. Finally, we showcase how jointly supporting these parallelism models in one tool naturally enables differentiation of hybrid parallel programs, and that deep integration of AD into the compiler enables performance optimizations usually only available in domain-specific/functional programming languages. Overall, our paper makes the following contributions:

- An extension to the theory of reverse-mode differentiation of single-static-assignment (SSA) intermediate representations to handle parallel execution of instructions, and thus differentiation of parallel languages and constructs that lower to such a representation.
- A demonstration of how implementing this model within the

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

William S. Moses
MIT CSAIL
`wmoses@mit.edu`

Valentin Churavy
MIT CSAIL
`vchuravy@mit.edu`

Abstract

Applying differentiable programming techniques and machine learning algorithms to foreign programs requires developers to either rewrite their code in a machine learning framework, or otherwise provide derivatives of the foreign code. This paper presents Enzyme[1], a high-performance automatic differentiation (AD) compiler plugin for the LLVM compiler framework capable of synthesizing gradients of statically analyzable programs expressed in the LLVM intermediate representation (IR). Enzyme synthesizes gradients for programs written in any language whose compiler targets LLVM IR including C, C++, Fortran, Julia, Rust, Swift, MLIR, etc., thereby providing native AD capabilities in these languages. Unlike traditional source-to-source and operator-overloading tools, Enzyme performs AD on optimized IR. On a machine-learning focused benchmark suite including Microsoft's ADBench, AD on optimized IR achieves a geometric mean speedup of 4.2 times over AD on IR before optimization allowing Enzyme to achieve state-of-the-art performance. Packaging Enzyme for PyTorch and TensorFlow provides convenient access to gradients of foreign code with state-of-the-art performance, enabling foreign code to be directly incorporated into existing machine learning workflows.

1 Introduction

Machine learning (ML) frameworks such as PyTorch [48] and TensorFlow [1] have become widespread as the primary workhorses of the modern ML community. Computing gradients necessary for algorithms such as backpropagation [32], Bayesian inference, uncertainty quantification [60], and probabilistic programming [16] requires all of the code being differentiated to be written in these frameworks. This is problematic for applying ML to new domains as existing tools like physics simulators [23, 10, 17, 18, 35], game engines, and climate models [58] are not written in the domain specific languages (DSL's) of ML frameworks. The rewriting required has been identified as the quintessential challenge of applying ML to scientific computing [4]. As stated by Rackauckas [50] "this is [the key challenge of scientific ML] because, if there is just one part of your loss function that isn't AD-compatible, then the whole network won't train."

To remedy this issue, the trend has been to either create new DSL's [35, 17, 43] that make the rewriting process easier or to add differentiation as a first-class construct in programming languages [44, 9, 61, 37]. This results in efficient gradients, but still requires rewriting in either the DSL or the differentiable programming language. Developers may want to use code foreign to a ML framework to either re-use existing tools or write loss functions in a language with an easier abstraction for their use case. While there exist reverse-mode automatic differentiation (AD) frameworks for various languages, using them automatically on foreign code for an ML framework is difficult as they still require rewriting and have limited support for cross-language AD and libraries[61, 33, 30, 36]. The two primary approaches to computing gradients are as follows.

[1] Code and documentation at `https://github.com/wsmoses/Enzyme` and `https://enzyme.mit.edu`.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

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)
```

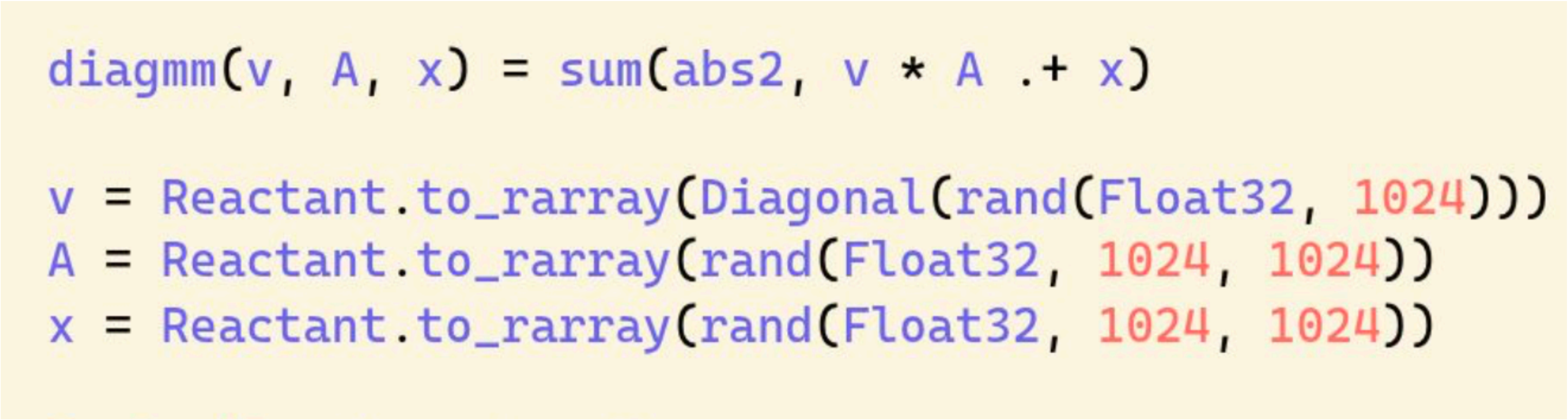
Often require program context

```
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
```

Linear Algebra + AD

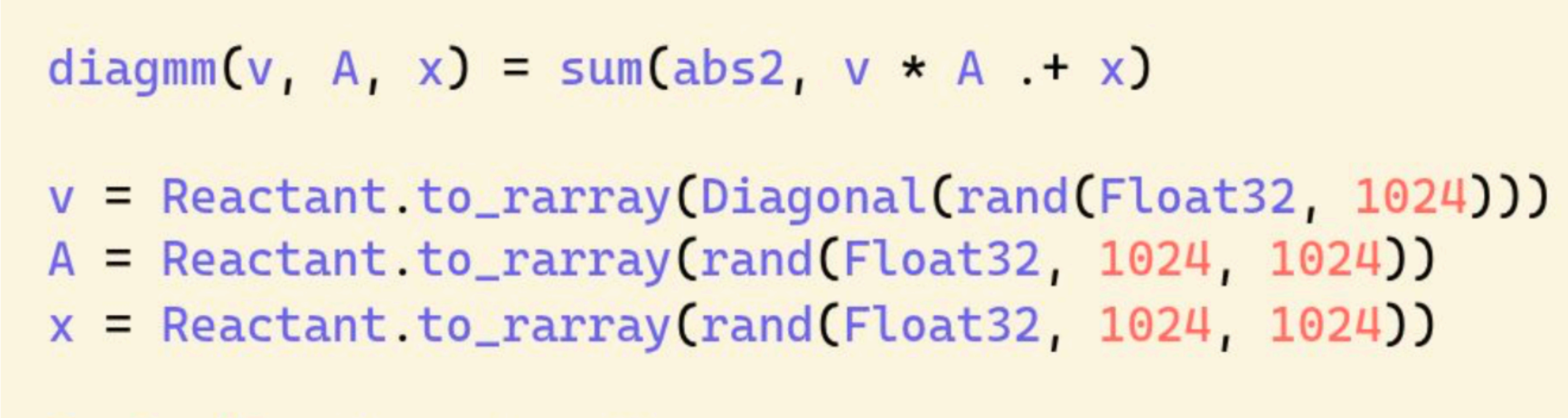
- Consider a simple code which performs a matmul and add on a Diagonal matrix



```
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))
```

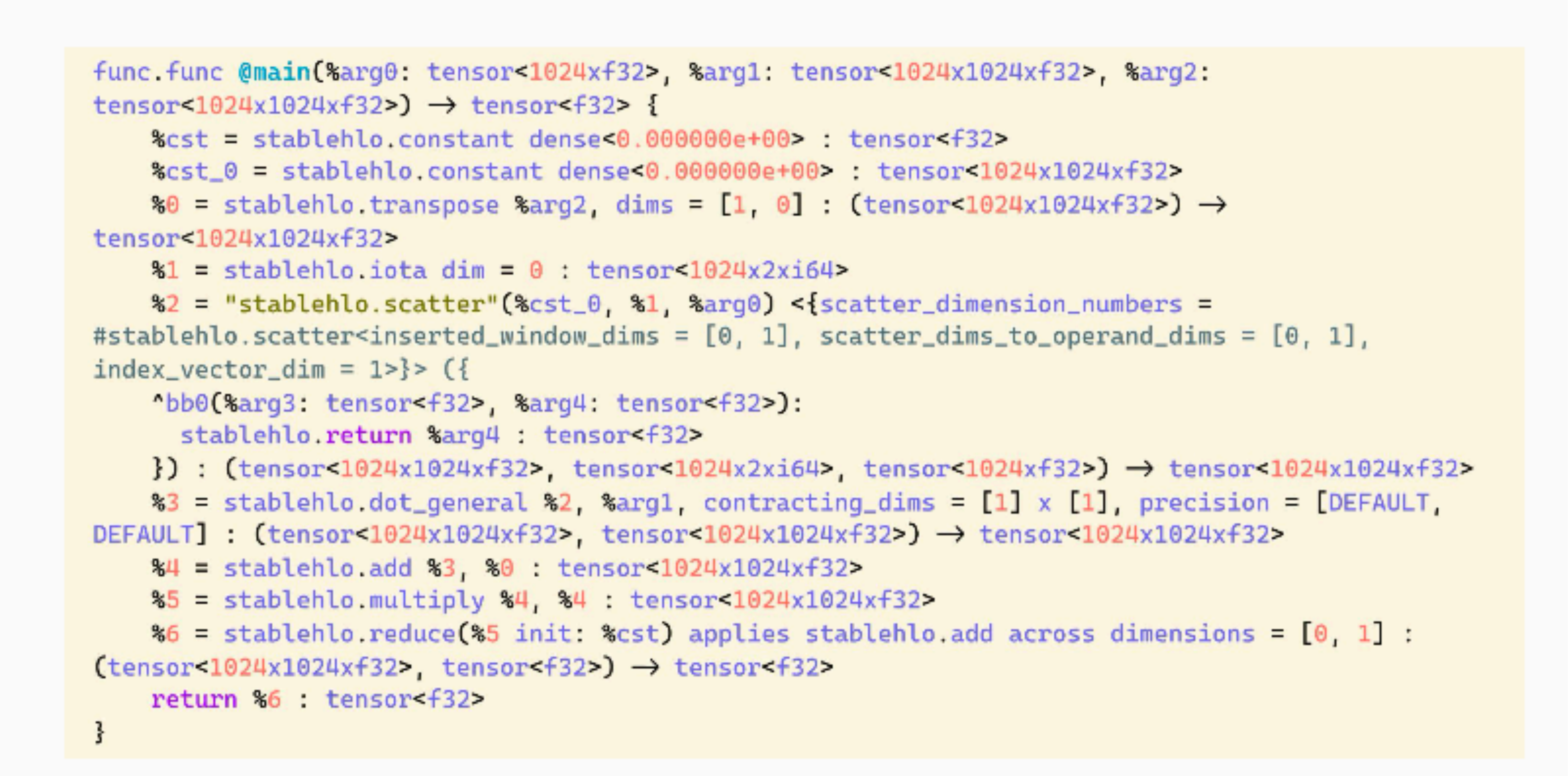
Linear Algebra + AD

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



```julia
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



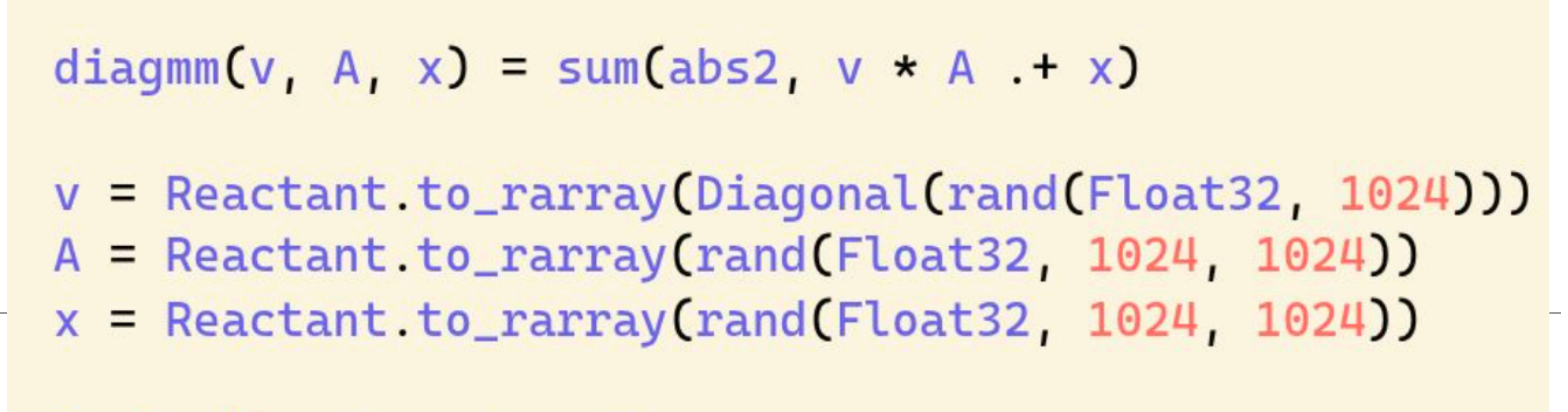
```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) → tensor<f32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %cst_0 = stablehlo.constant dense<0.000000e+00> : tensor<1024x1024xf32>
    %0 = stablehlo.transpose %arg2, dims = [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

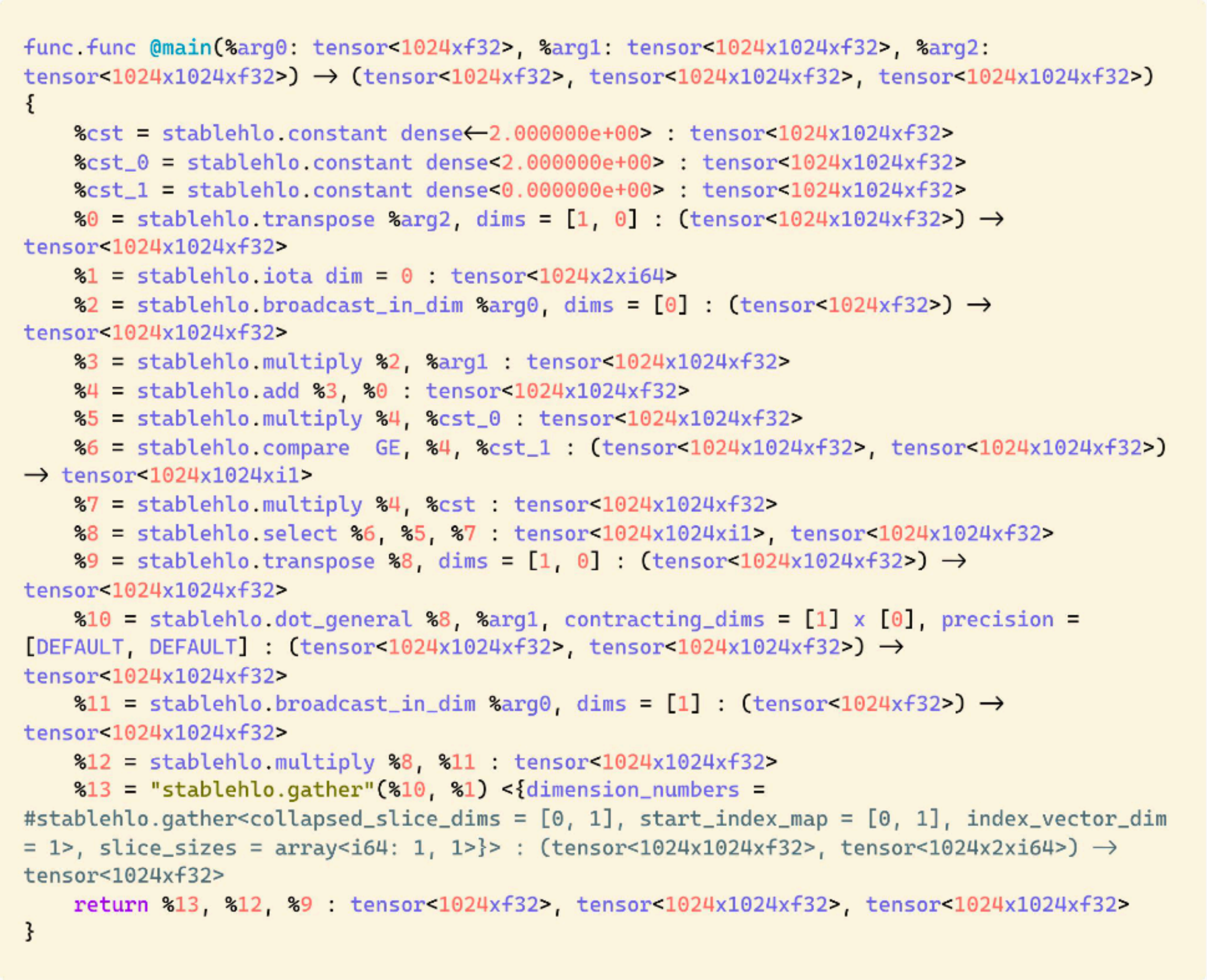
- Consider a simple code which performs a matmul and add on a Diagonal matrix

- 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))
```



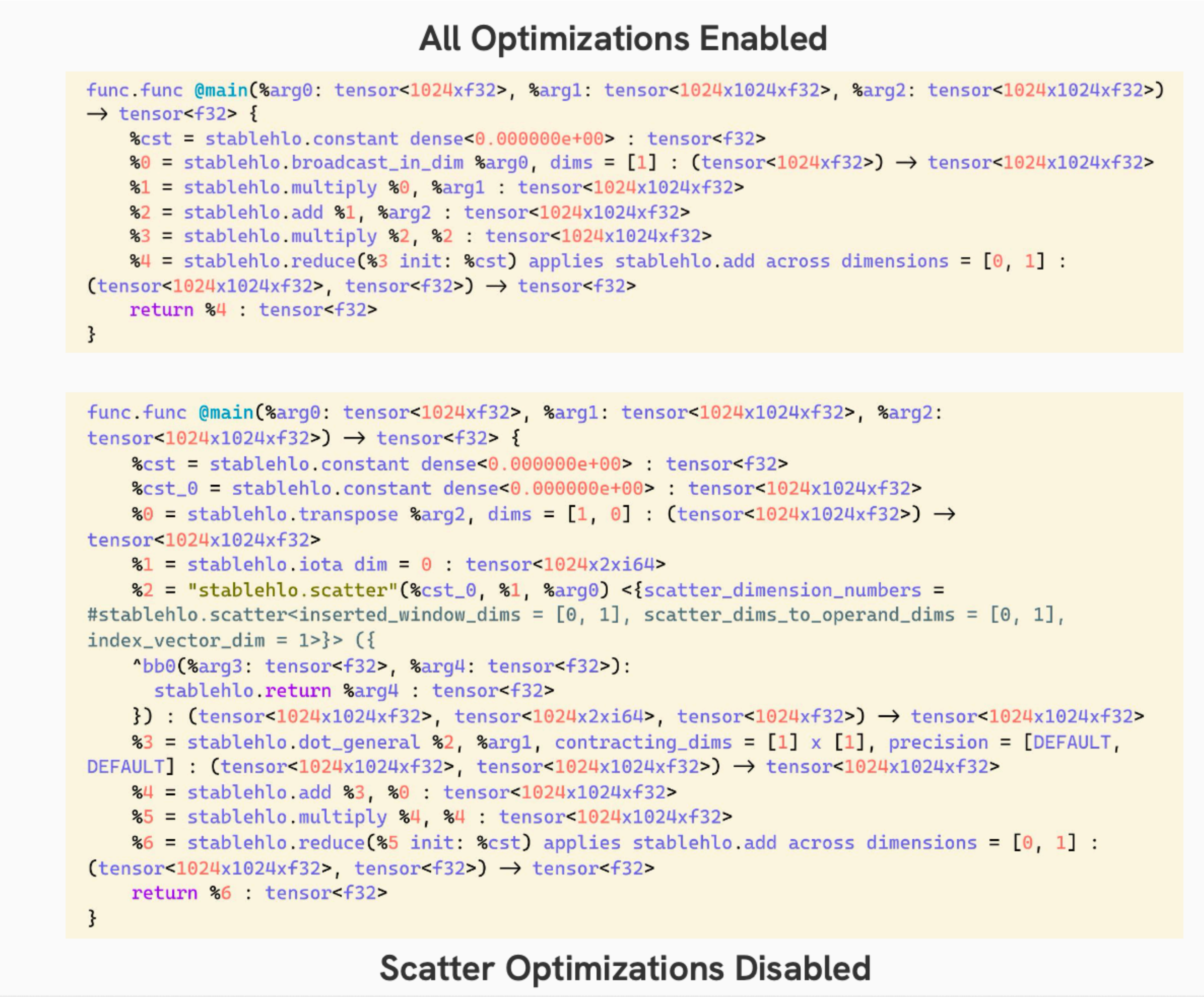
```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2:
tensor<1024x1024xf32>) → (tensor<1024xf32>, tensor<1024x1024xf32>, tensor<1024x1024xf32>)
{
    %cst = stablehlo.constant dense←2.000000e+00> : tensor<1024x1024xf32>
    %cst_0 = stablehlo.constant dense<2.000000e+00> : 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!



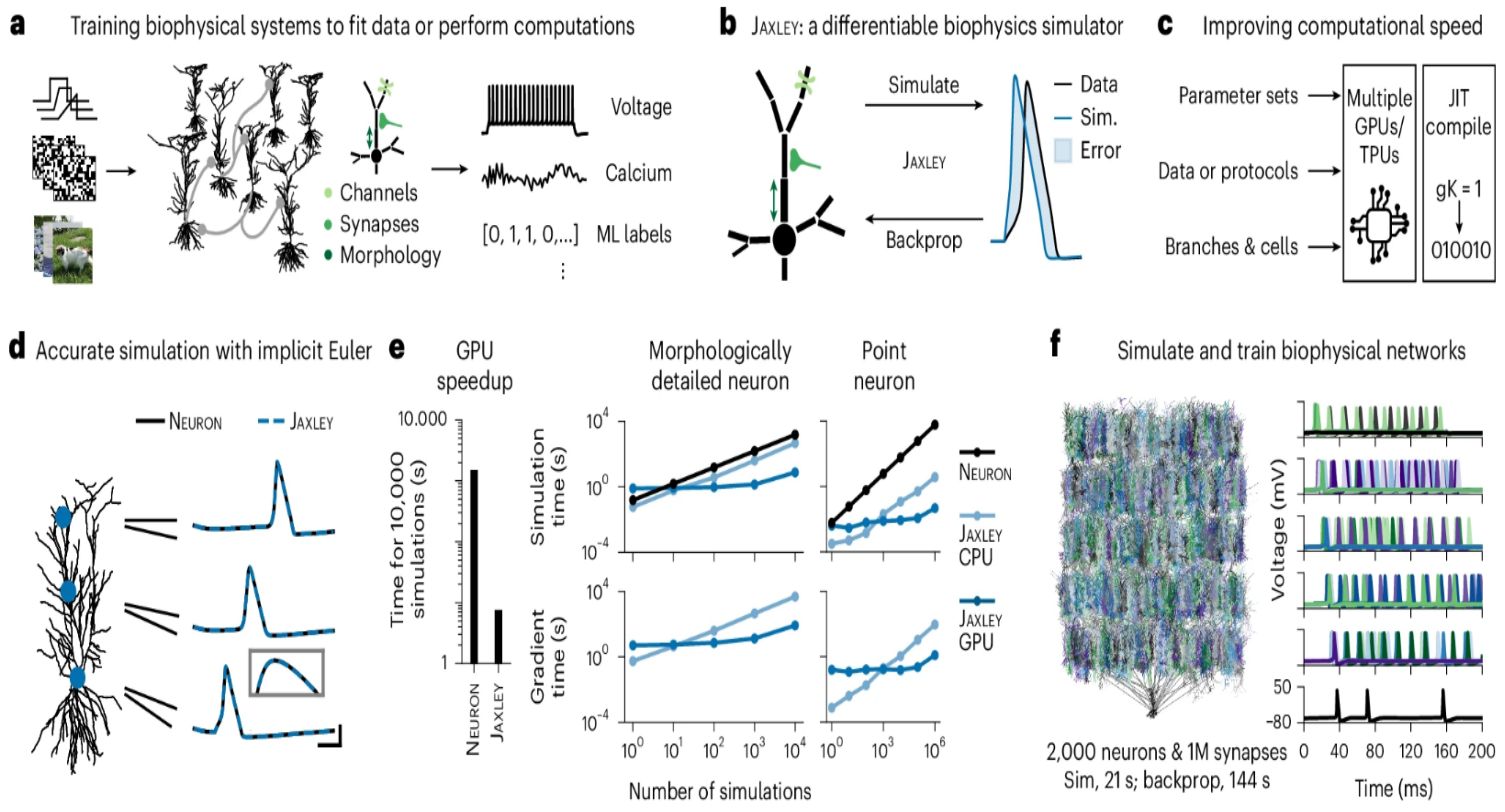
All Optimizations Enabled

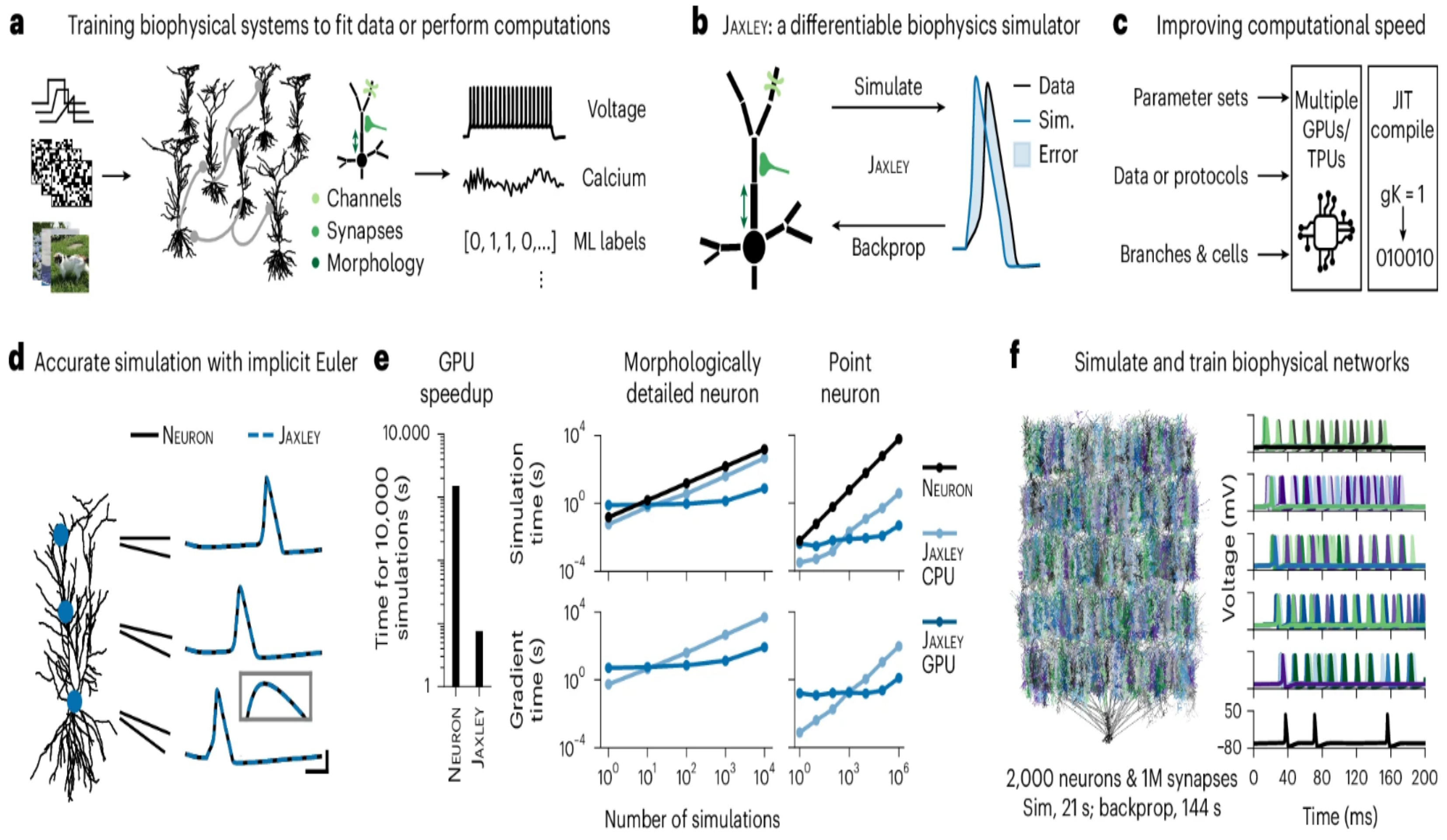
```
func.func @main(%arg0: tensor<1024xf32>, %arg1: tensor<1024x1024xf32>, %arg2: tensor<1024x1024xf32>)
→ tensor<f32> {
    %cst = stablehlo.constant dense<0.000000e+00> : tensor<f32>
    %0 = stablehlo.broadcast_in_dim %arg0, dims = [1] : (tensor<1024xf32>) → tensor<1024x1024xf32>
    %1 = stablehlo.multiply %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

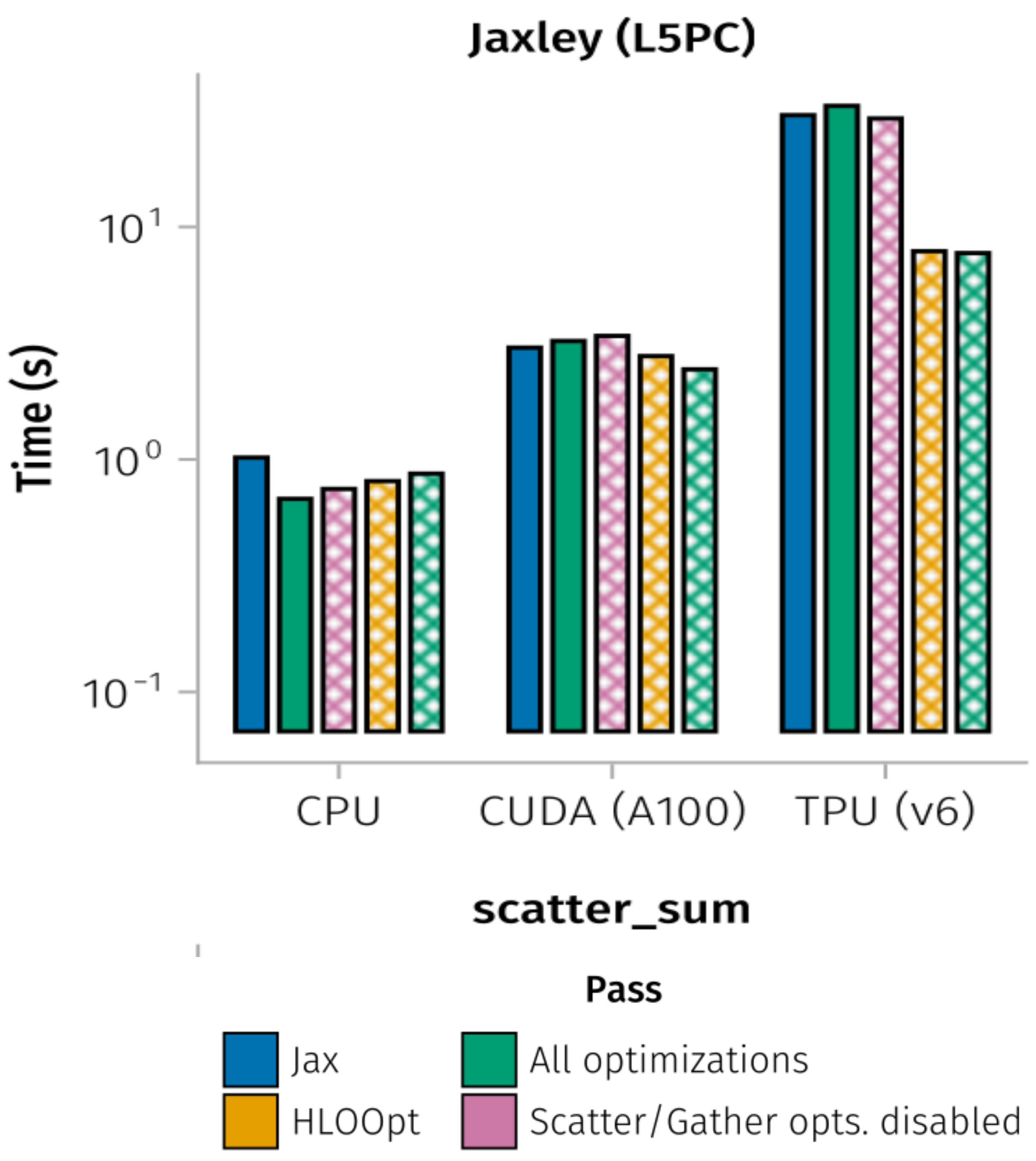
Work Reduction Benchmark: Jaxley

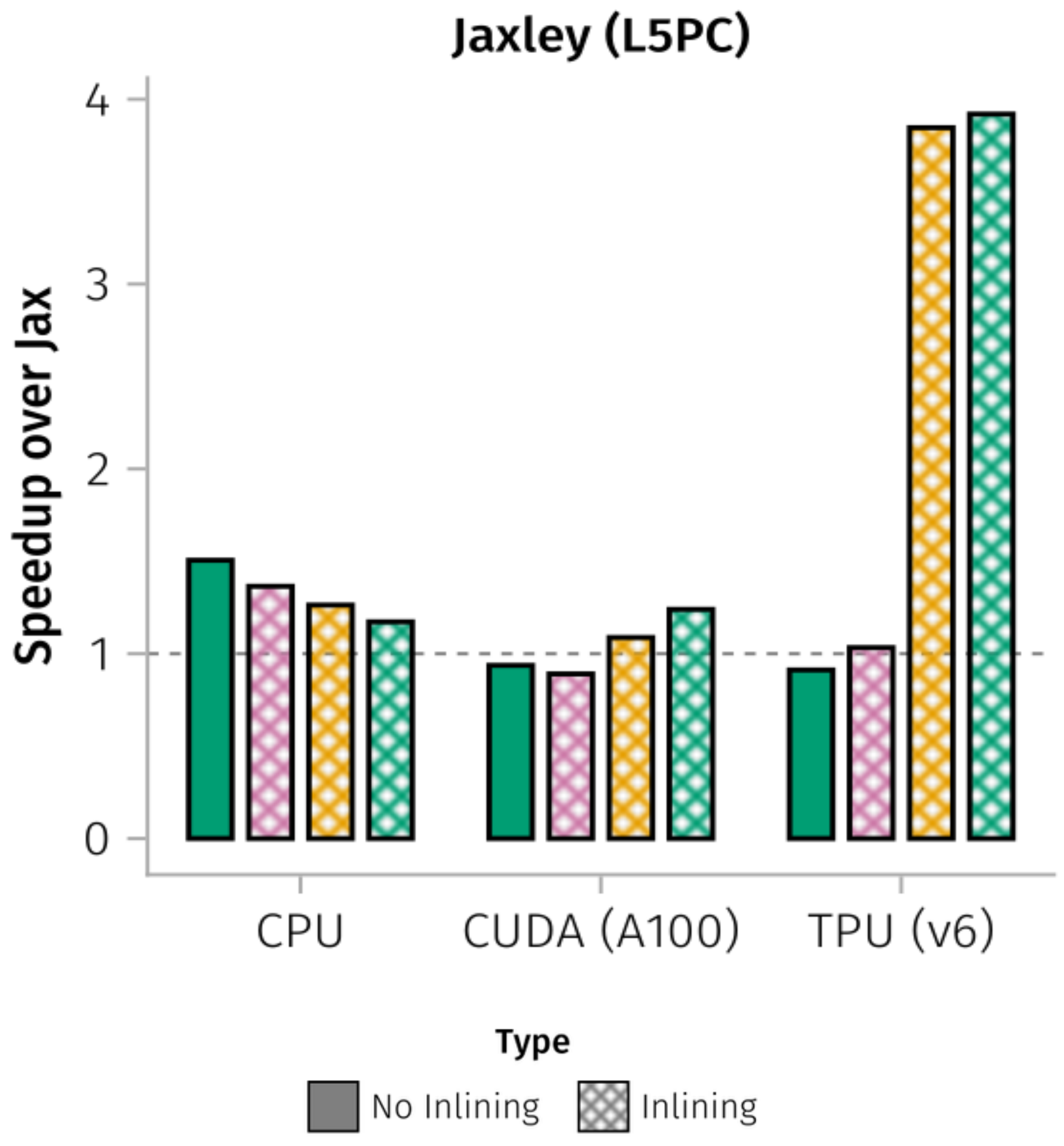

a Training biophysical systems to fit data or perform computations
Channels
Synapses
Morphology
Voltage
Calcium
[0, 1, 1, 0,...] ML labels
b Jaxley: a differentiable biophysics simulator
Simulate
Jaxley
Backprop
Data
Sim.
Error
c Improving computational speed
Parameter sets
Data or protocols
Branches & cells
Multiple GPUs/ TPUs
JIT compile
gK = 1
010010
d Accurate simulation with implicit Euler
Neuron
Jaxley
e GPU speedup
Time for 10,000 simulations (s)
10,000
1
Neuron
Jaxley
Morphologically detailed neuron
Point neuron
Simulation time (s)
Gradient time (s)
Number of simulations
Neuron
Jaxley CPU
Jaxley GPU
f Simulate and train biophysical networks
Voltage (mV)
50
−80
Time (ms)
0 40 80 120 160 200
2,000 neurons & 1M synapses
Sim, 21 s; backprop, 144 s



a Training biophysical systems to fit data or perform computations
Channels
Synapses
Morphology
Voltage
Calcium
[0, 1, 1, 0,...] ML labels
b Jaxley: a differentiable biophysics simulator
Simulate
Jaxley
Backprop
Data
Sim.
Error
c Improving computational speed
Parameter sets
Data or protocols
Branches & cells
Multiple GPUs/ TPUs
JIT compile
gK = 1
010010
d Accurate simulation with implicit Euler
Neuron
Jaxley
e GPU speedup
Time for 10,000 simulations (s)
10,000
1
Neuron
Jaxley
Morphologically detailed neuron
Point neuron
Simulation time (s)
Gradient time (s)
Number of simulations
Neuron
Jaxley CPU
Jaxley GPU
f Simulate and train biophysical networks
Voltage (mV)
50
−80
Time (ms)
0 40 80 120 160 200
2,000 neurons & 1M synapses
Sim, 21 s; backprop, 144 s

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

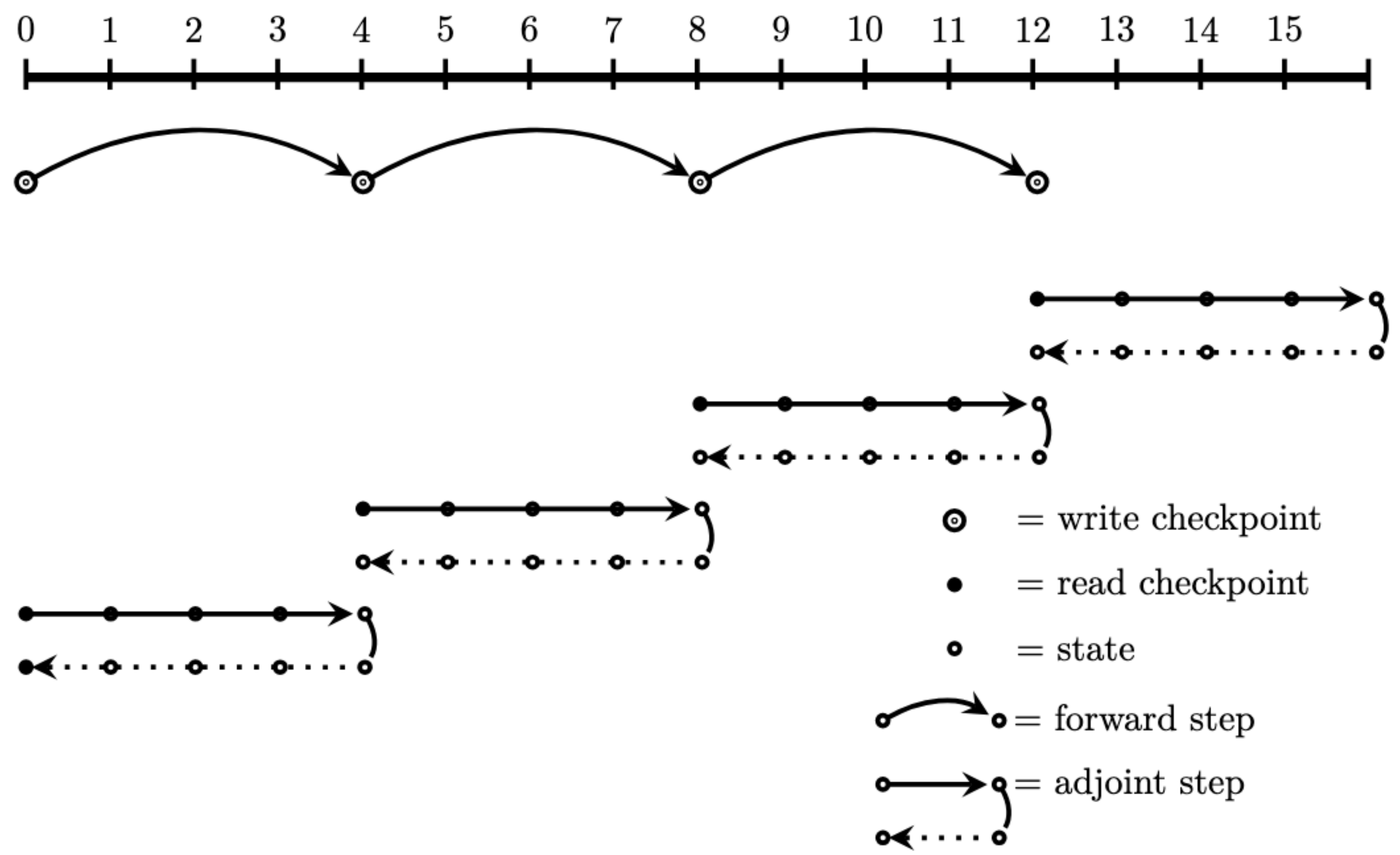




[1] Deistler, Michael, et al. "Jaxley: differentiable simulation enables large-scale training of detailed biophysical models of neural dynamics." Nature Methods (2025): 1-9.

Checkpointing

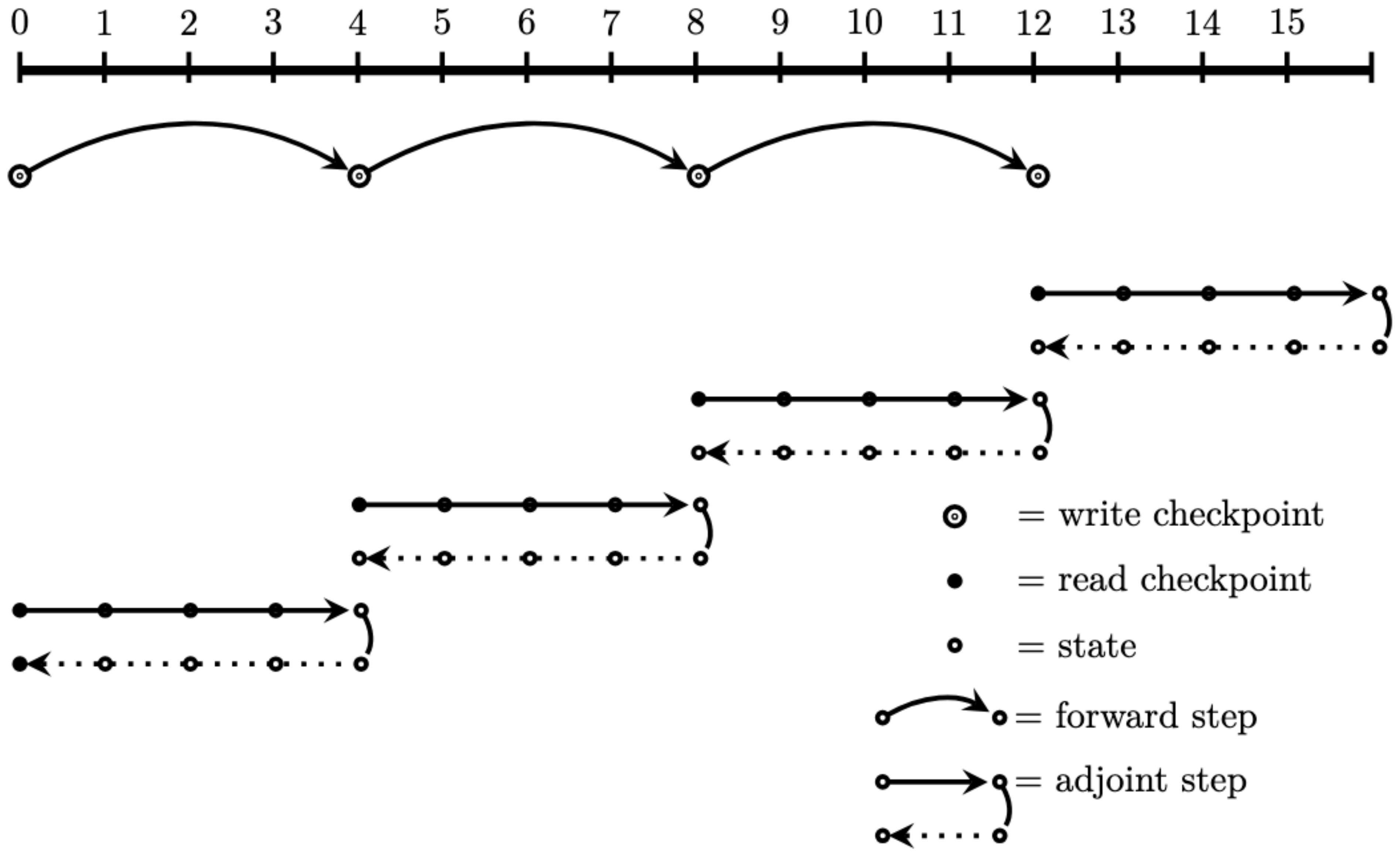
- Checkpointing is a technique for trading off memory and compute time in the derivative



```
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

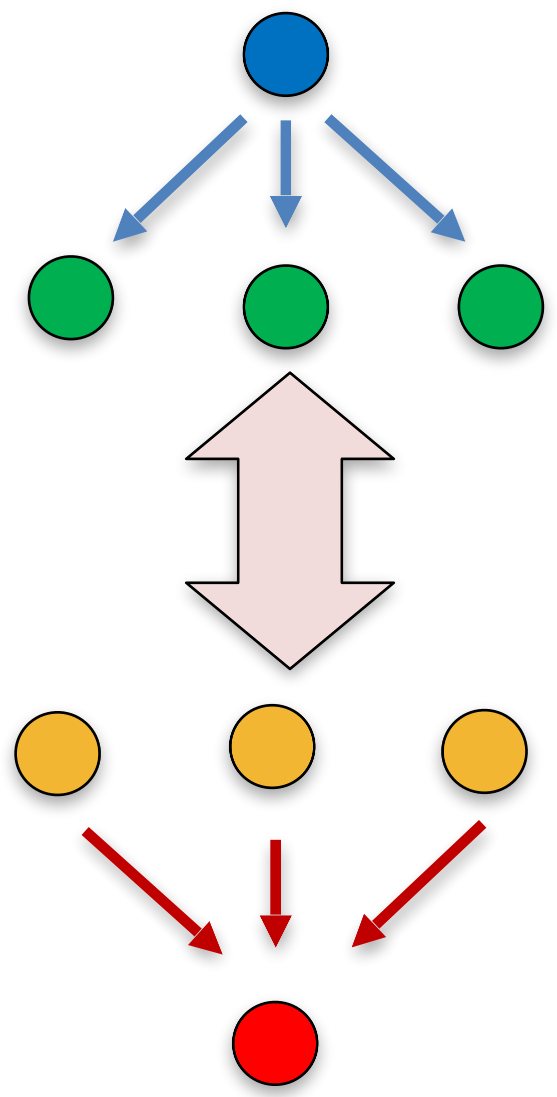
```
x = tensor<100x100xf32>
for i = 0:steps {
   x[0,  :] = 0
   x[end,:] = 0
   y = foo(x, y)
}
```

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

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

Communication + AD

Differentiation changes how we want to parallelize code

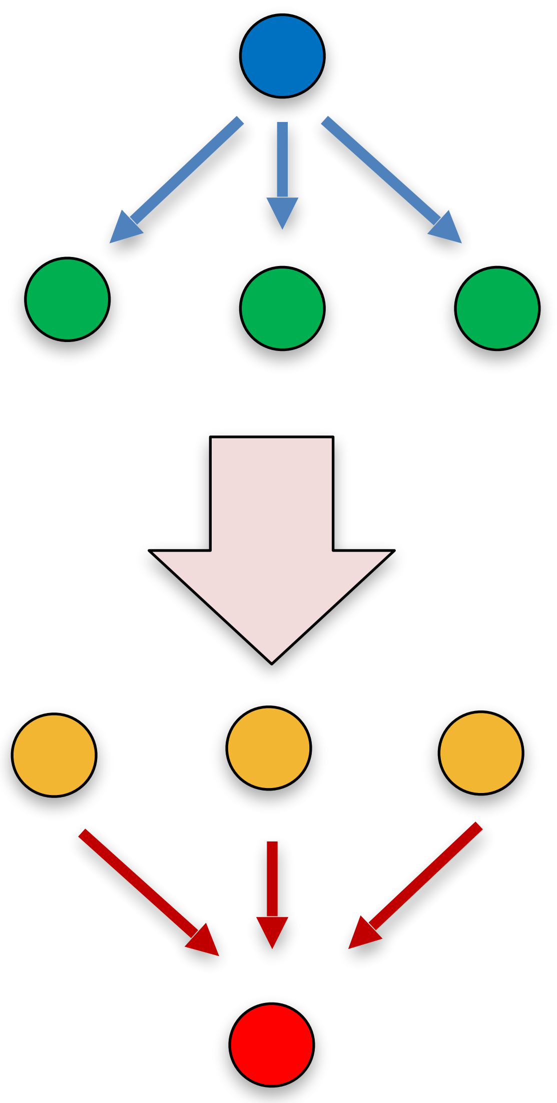
- Scatters <-> Gathers



Communication + AD

Differentiation changes how we want to parallelize code

- Scatters <-> Gathers
- Can create **race conditions**



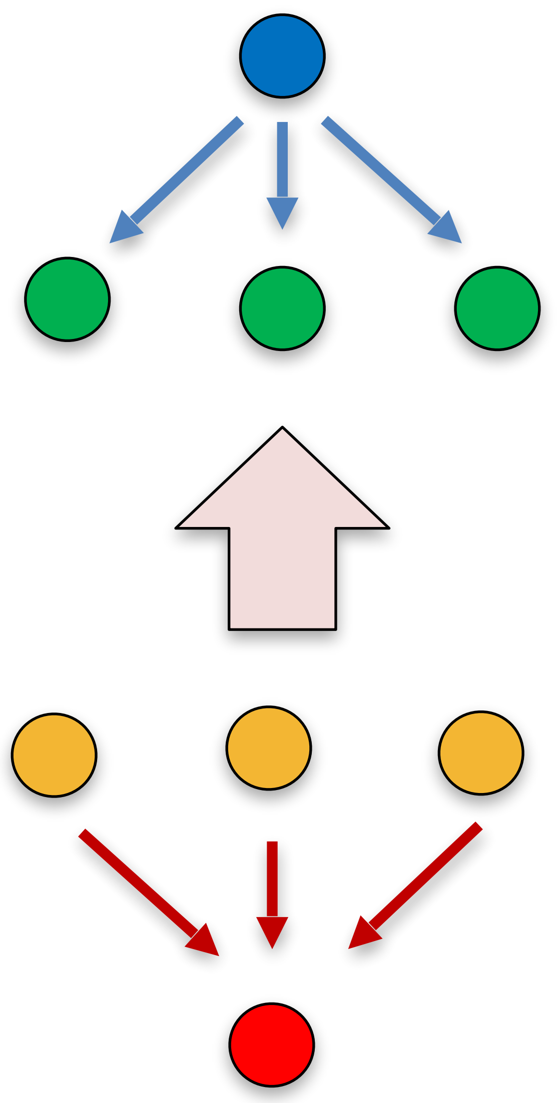
```
void set(double* ar, double val) {
  pfor(int i=0; i<n; i++) {
     ar[i] = val;
  }
  …
}
```

```
void grad_set(double* ar, double* d_ar) {
  double d_val = 0;
  pfor (int i=0; i<n; i++) {
     d_val += d_ar[i];
  }
  …
```

Communication + AD

Differentiation changes how we want to parallelize code

- Scatters <-> Gathers
- Can create **race conditions**
- Serial Primal => Parallel Derivative

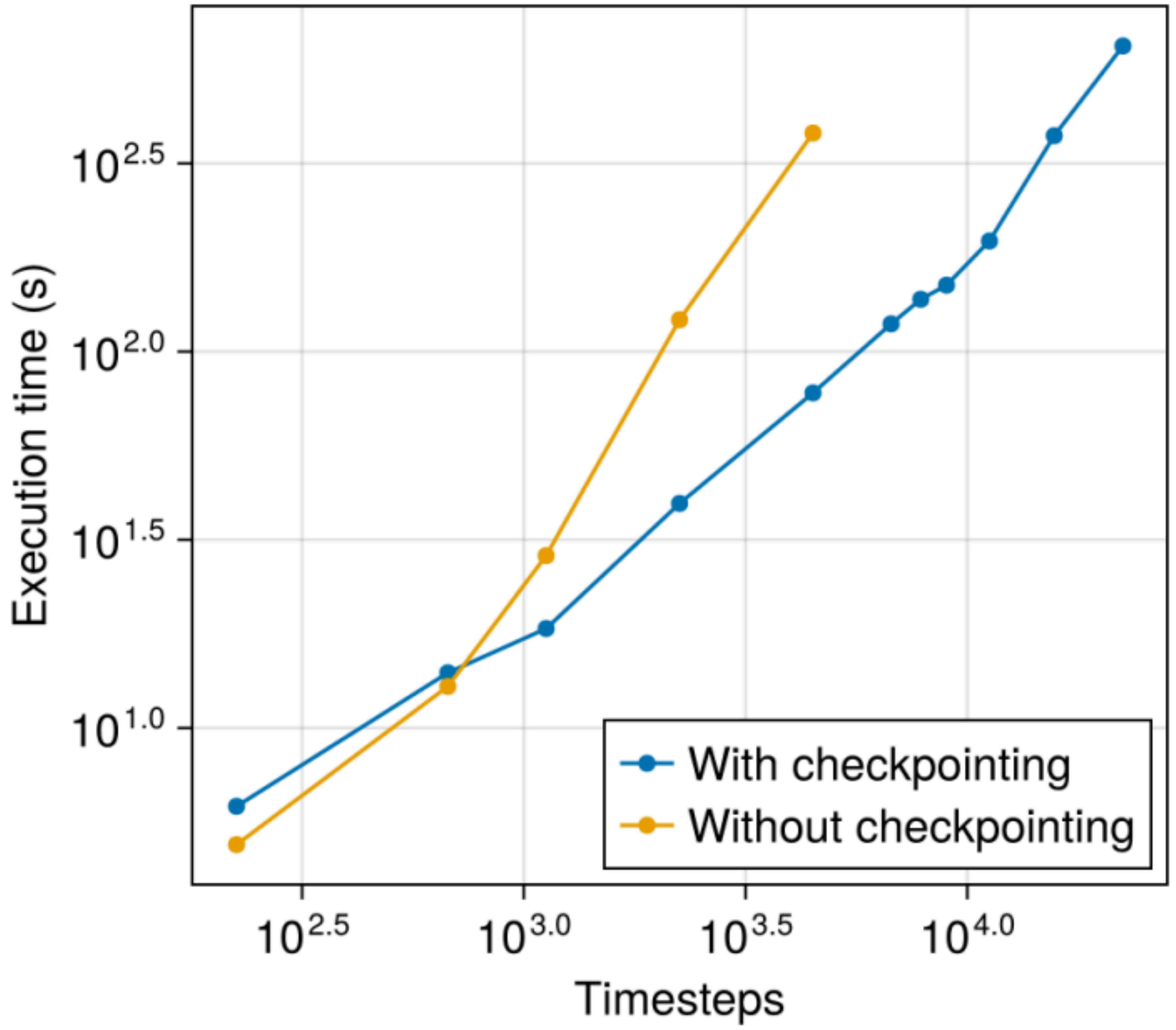


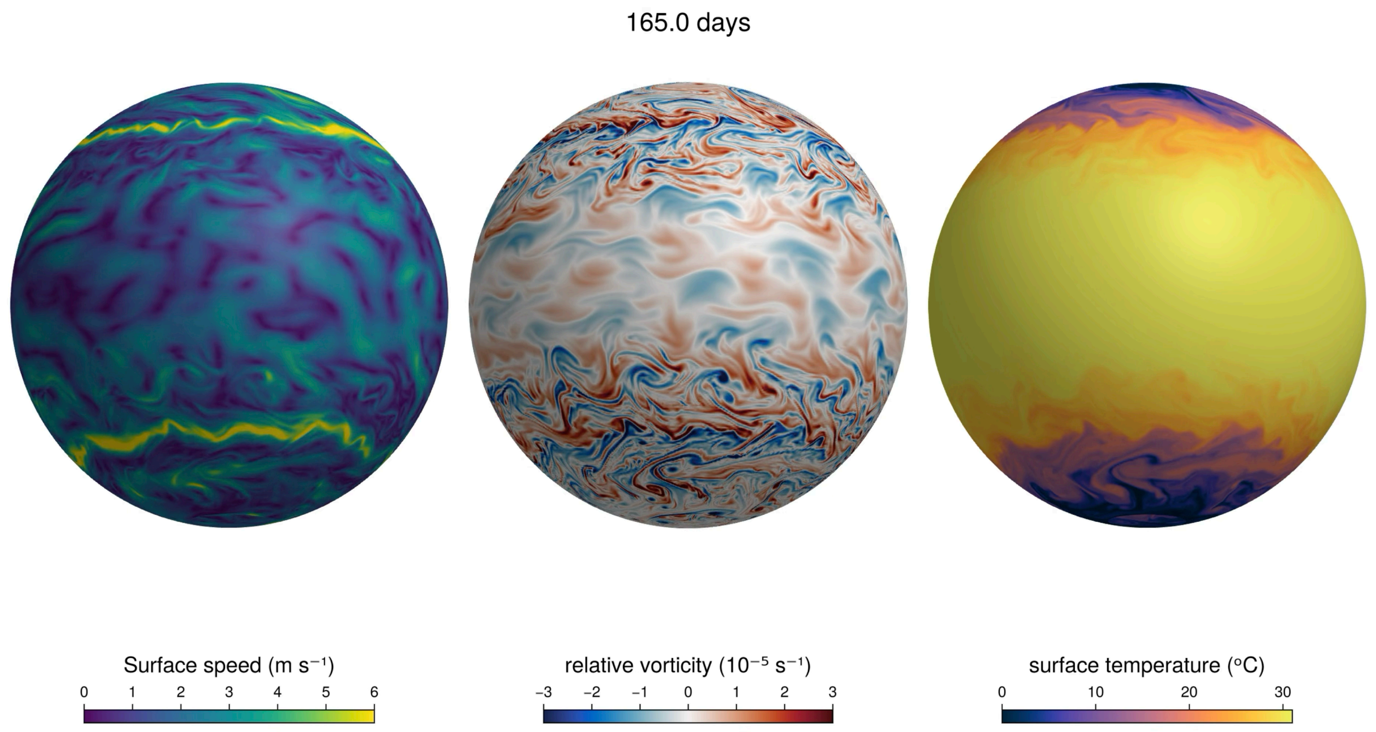
```
double sum(double* x) {
  double S = 0.0;
  for (int i = 0; i < N; i++) {
    S += x[i] * x[i];
  }
  return S;
}
```

```
void grad_sum(double* x, double* d_x,
              double d_S) {
  pfor (int i = 0; i < N; i++) {
    d_x[i] += 2.0 * x[i] * d_S;
  }
}
```

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!





Takeaways

- Compilers Make Differentiation Fast and Easy to use
  - Key to this is interaction with Optimization
- Executing on accelerators historically require rewriting entire workflows
- Raising enables existing workflows to execute on (distributed accelerators)
- EnzymeMLIR enables preserving and optimizing high-level structure and optimizations, whose impact is compounded on such accelerators
- All open source (GitHub.com/EnzymeAD/Enzyme ; GitHub.com/EnzymeAD/Enzyme-JaX ; GitHub.com/EnzymeAD/Reactant.jl)