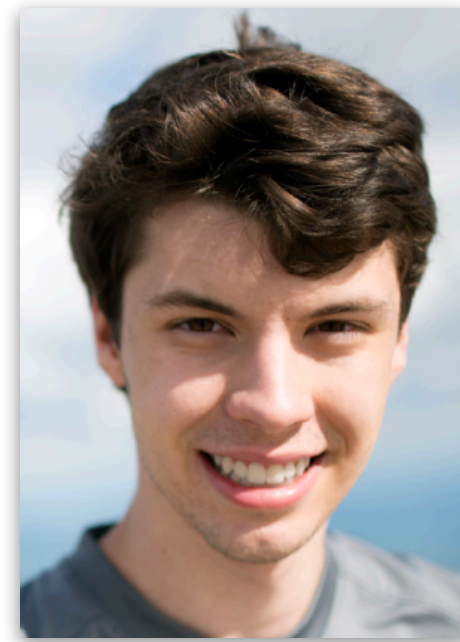


# Making Waves in the Cloud: A Paradigm Shift for Scientific Computing through Compiler Technology



William S. Moses

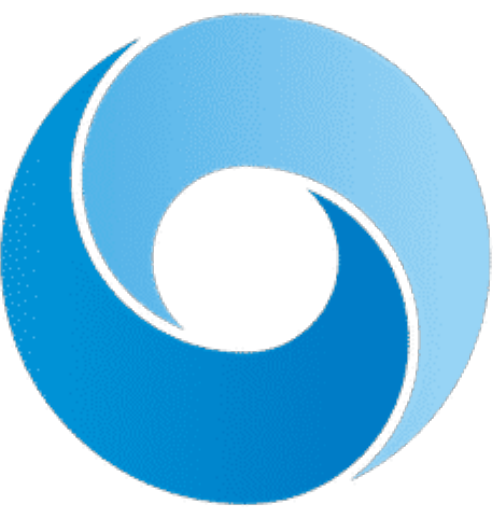
[wsmoses@illinois.edu](mailto:wsmoses@illinois.edu)

Dagstuhl SPE

Aug 21, 2025



UNIVERSITY OF  
**ILLINOIS**  
URBANA-CHAMPAIGN



William S. Moses<sup>†§</sup>, Mosè Giordano<sup>★</sup>, Avik Pal<sup>‡</sup>, Gregory Wagner<sup>‡</sup>, Ivan R Ivanov, Paul Berg<sup>∇</sup>,  
 Johannes Blaschke, Jules Merckx<sup>△</sup>, Arpit Jaiswal<sup>◆</sup>, Patrick Heimbach<sup>#</sup>, Son Vu, Sergio  
 Sanchez-Ramirez<sup>◇</sup>, Simone Silvestri, Nora Loose<sup>♣</sup>, Ivan Ho, Vimarsh Sathia<sup>†</sup>, Jan Hueckelheim<sup>♣</sup>,  
 Johannes De Fine Licht, Kevin Gleason<sup>§</sup>, Ludovic Rass, Gabriel Baraldi, Dhruv Apte<sup>#</sup>, Lorenzo  
 Chelini<sup>◆</sup>, Jacques Pienaar<sup>§</sup>, Gaetan Lounes, Valentin Churavy, Sri Hari Krishna Narayanan<sup>♣</sup>, Navid  
 Constantinou, William R. Magro<sup>§</sup>, Michel Schanen<sup>♣</sup>, Alexis Montoison<sup>♣</sup>, Alan Edelman<sup>‡</sup>, Samarth  
 Narang, Tobias Grosser, Keno Fischer<sup>‡</sup>, Robert Hundt<sup>§</sup>, Albert Cohen<sup>§</sup>, Oleksandr Zinenko<sup>§ \*</sup>  
 UIUC <sup>†</sup>, Google <sup>§</sup>, UCL <sup>★</sup>, MIT <sup>‡</sup>, NVIDIA <sup>◆</sup>, UT Austin <sup>#</sup>, [C]Worthy <sup>♣</sup>, BSC <sup>◇</sup>, Argonne National Laboratory <sup>♣</sup>,  
 LBNL <sup>♡</sup>, Cambridge <sup>‡</sup>, JuliaHub <sup>‡</sup>, University of Mainz <sup>#</sup>, BFH <sup>∇</sup>, Ghent University <sup>△</sup>

# 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

Aa



[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

ANTHROPIC

Claude ▾ API ▾ Solutions ▾ Research ▾ Commitments ▾ Learn ▾ News

Try Claude

Product

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

Dec 3, 2024 • 3 min read

# 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

ANTHROPIC

Claude ▾API ▾Solutions ▾Research ▾Commitments ▾Learn ▾News

Try Claude

Product

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

Dec 3, 2024 • 3 min read

## 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, 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

ANTHROPIC

[Claude](#) [API](#) [Solutions](#) [Research](#) [Commitments](#) [Learn](#) [News](#)

Try Claude

Product

## Claude 3.5 Haiku on AWS Tr

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 Udinmwun](#) published July 26, 2025

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



## 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, Belgium. Photo by Herman/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

ANTHROPIC

Claude ▾ API ▾ Solutions ▾ Research ▾ Commitments ▾ Learn ▾ News

Try Claude

Product

Claude 3.5 Haiku on AWS Tru

### 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.

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

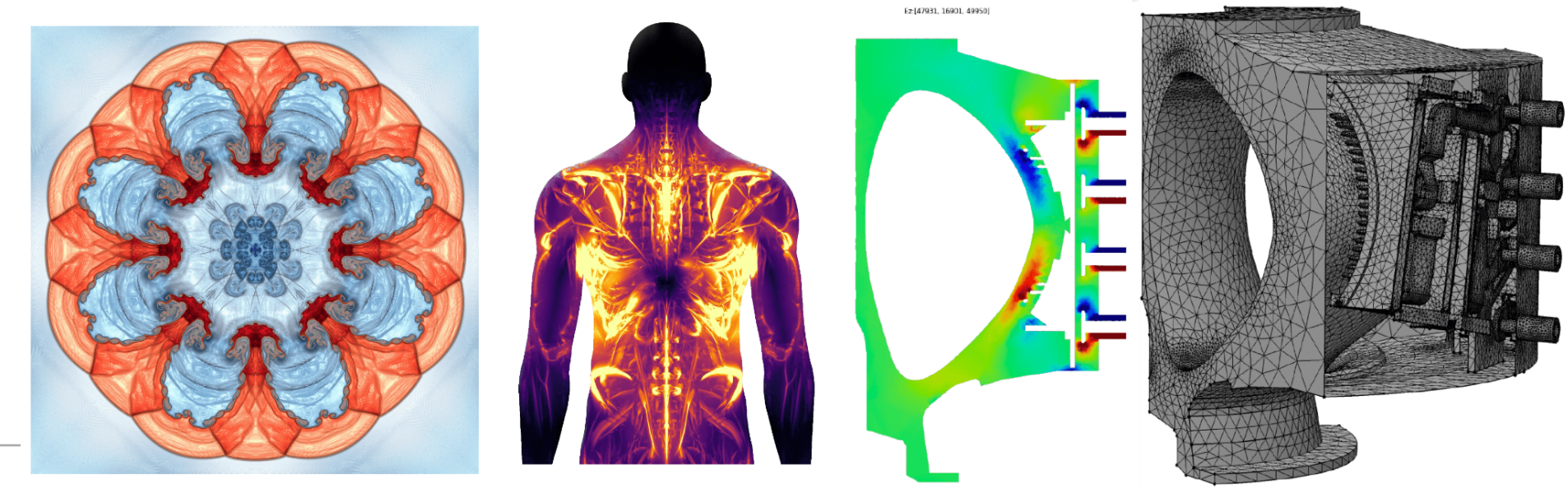
News By [Andy Edser](#) published 23 July 2025

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

[f](#) [x](#) [r](#) [p](#) [m](#) [Comments \(2\)](#)



# Lingua Franca of Scientific Computing



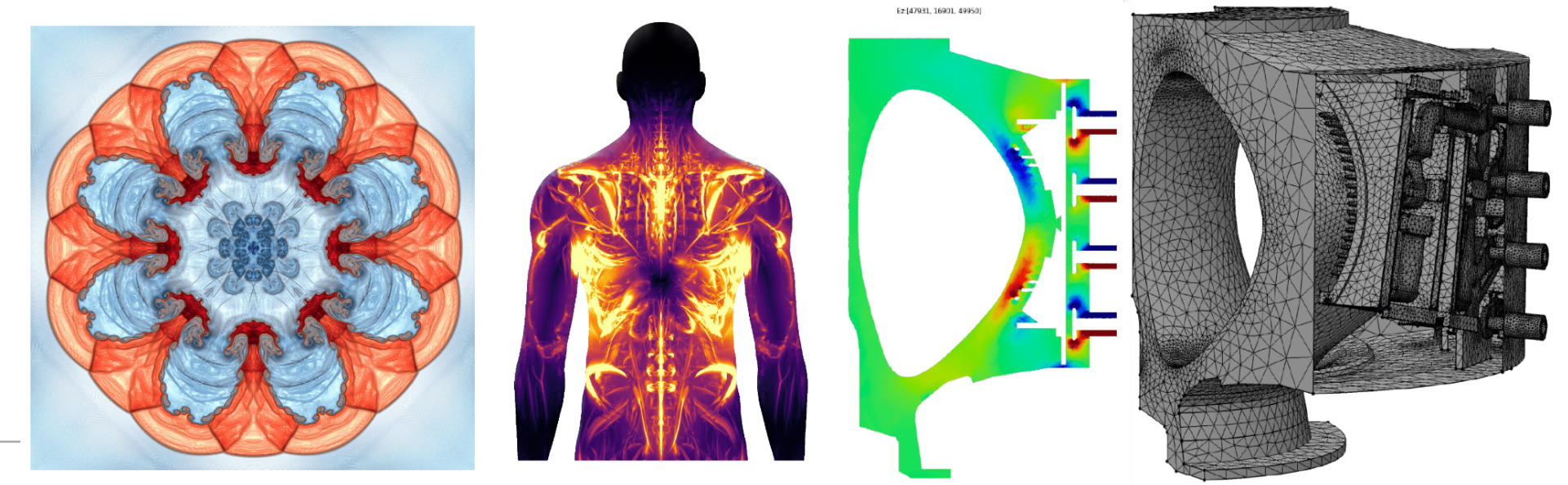
- Scientists do not write TPU\* code

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

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

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

# Lingua Franca of Scientific Computing

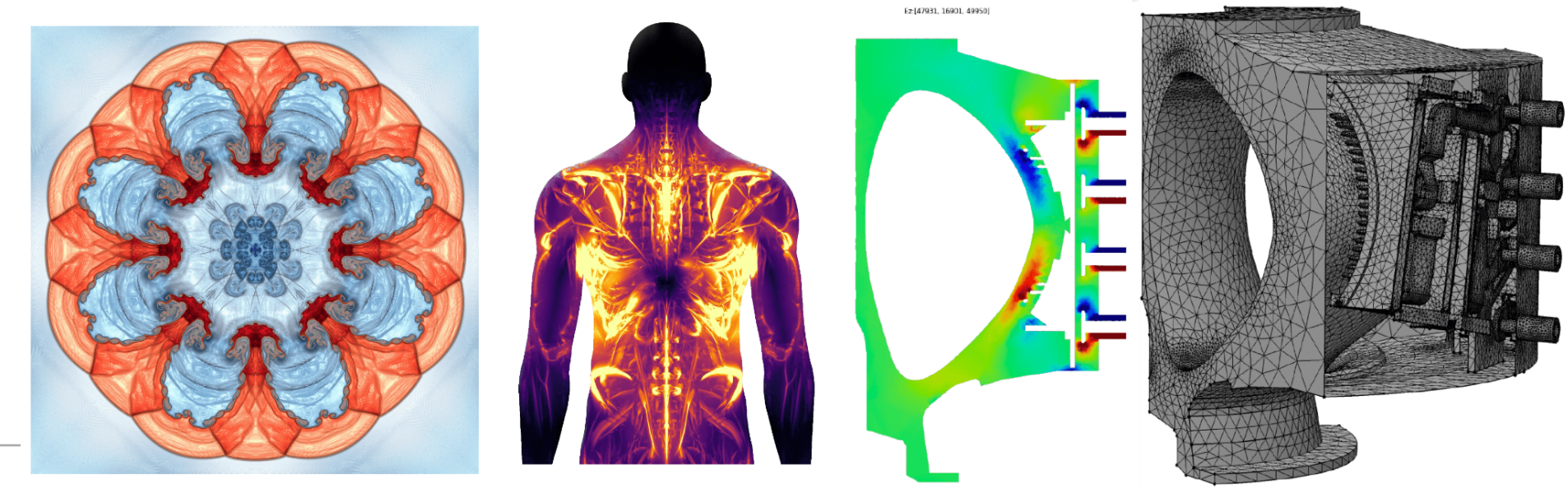


- Scientists do not write TPU\* code
  - BIG (MFEM library alone is 737K LOC)

```
__global__  
void AddNodeForcesFromElems_kernel( Index_t numNode,  
                                     Index_t padded_numNode,  
                                     const Int_t* nodeElemCount,  
                                     const Int_t* nodeElemStart,  
                                     const Index_t* nodeElemCornerList,  
                                     const Real_t* fx_elem,  
                                     const Real_t* fy_elem,  
                                     const Real_t* fz_elem,  
                                     Real_t* fx_node,  
                                     Real_t* fy_node,  
                                     Real_t* fz_node,  
                                     const Int_t num_threads)  
{  
    int tid=blockDim.x*blockIdx.x+threadIdx.x;  
    if (tid < num_threads)  
    {  
        Index_t g_i = tid;  
        Int_t count=nodeElemCount[g_i];  
        Int_t start=nodeElemStart[g_i];  
        Real_t fx,fy,fz;  
        fx=fy=fz=Real_t(0.0);  
  
        for (int j=0;j<count;j++)  
        {  
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here  
            fx += fx_elem[pos];  
            fy += fy_elem[pos];  
            fz += fz_elem[pos];  
        }  
  
        fx_node[g_i]=fx;  
        fy_node[g_i]=fy;  
        fz_node[g_i]=fz;  
    }  
}
```



# Lingua Franca of Scientific Computing



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

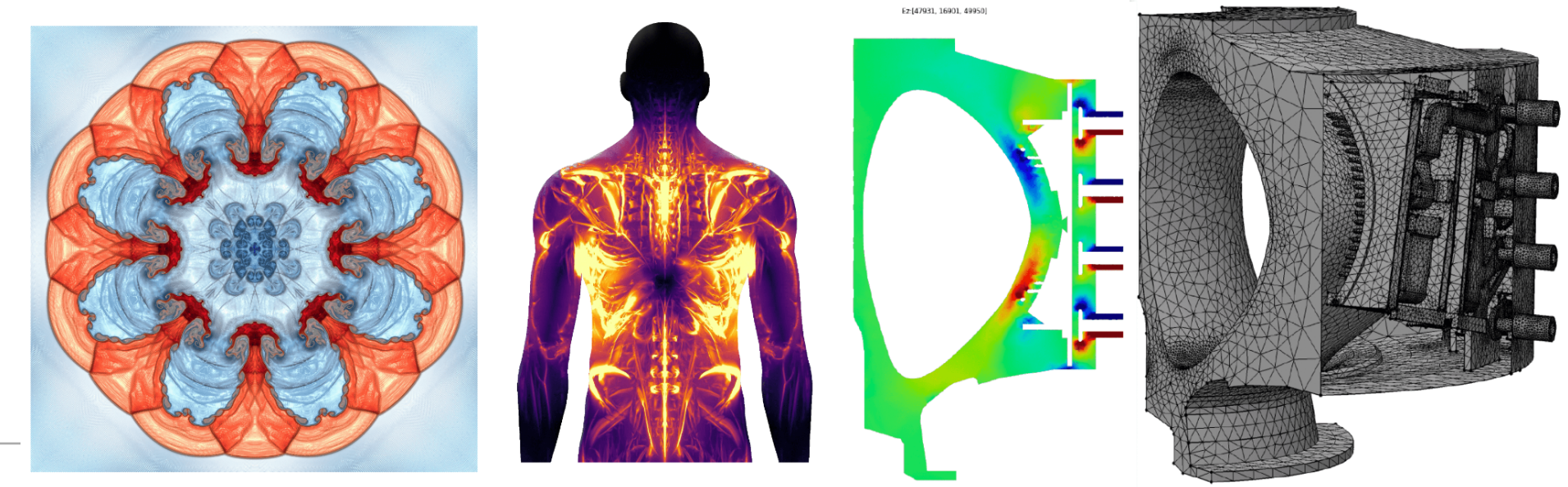
```
__global__
void AddNodeForcesFromElems_kernel( Index_t numNode,
                                     Index_t padded_numNode,
                                     const Int_t* nodeElemCount,
                                     const Int_t* nodeElemStart,
                                     const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

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

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



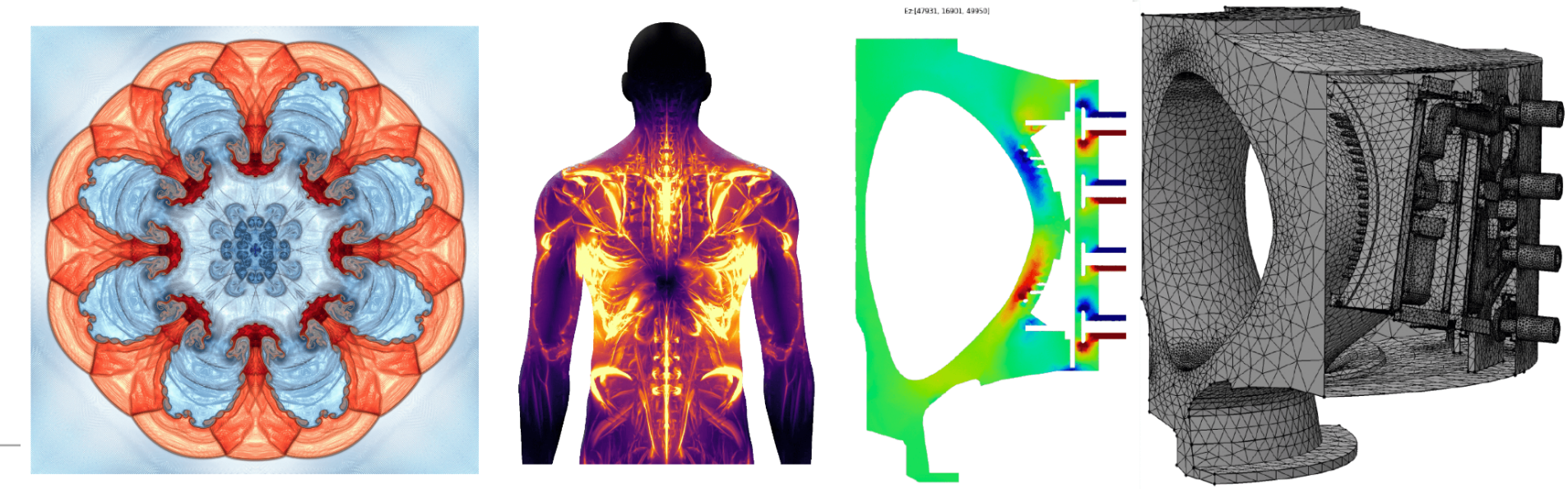
# Lingua Franca of Scientific Computing



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

```
__global__  
void AddNodeForcesFromElems_kernel( Index_t numNode,  
                                     Index_t padded_numNode,  
                                     const Int_t* nodeElemCount,  
                                     const Int_t* nodeElemStart,  
                                     const Index_t* nodeElemCornerList,  
                                     const Real_t* fx_elem,  
                                     const Real_t* fy_elem,  
                                     const Real_t* fz_elem,  
                                     Real_t* fx_node,  
                                     Real_t* fy_node,  
                                     Real_t* fz_node,  
                                     const Int_t num_threads)  
{  
    int tid=blockDim.x*blockIdx.x+threadIdx.x;  
    if (tid < num_threads)  
    {  
        Index_t g_i = tid;  
        Int_t count=nodeElemCount[g_i];  
        Int_t start=nodeElemStart[g_i];  
        Real_t fx,fy,fz;  
        fx=fy=fz=Real_t(0.0);  
  
        for (int j=0;j<count;j++)  
        {  
            Index_t pos=nodeElemCornerList[start+j]; // Uncoalesced access here  
            fx += fx_elem[pos];  
            fy += fy_elem[pos];  
            fz += fz_elem[pos];  
        }  
  
        fx_node[g_i]=fx;  
        fy_node[g_i]=fy;  
        fz_node[g_i]=fz;  
    }  
}
```

# Lingua Franca of Scientific Computing



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

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

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

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

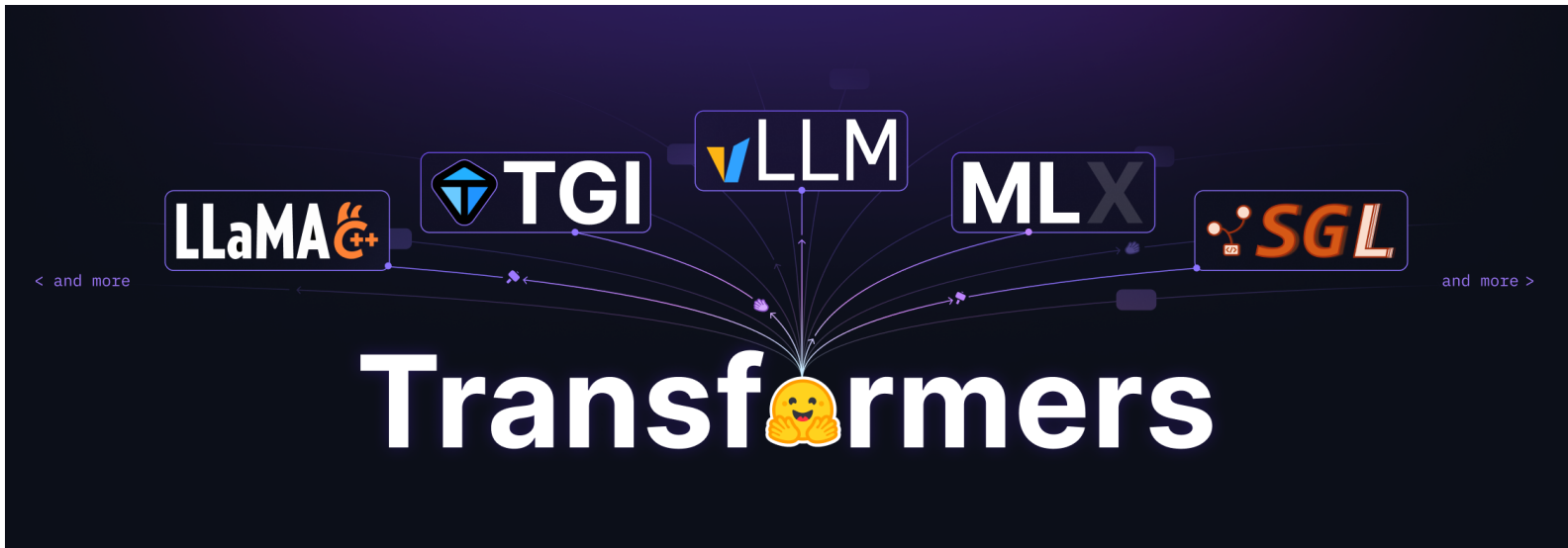
        fx_node[g_i]=fx;
        fy_node[g_i]=fy;
        fz_node[g_i]=fz;
    }
}
```

# How do we write ML Accelerator code now?

---



# How do we write ML Accelerator code now?



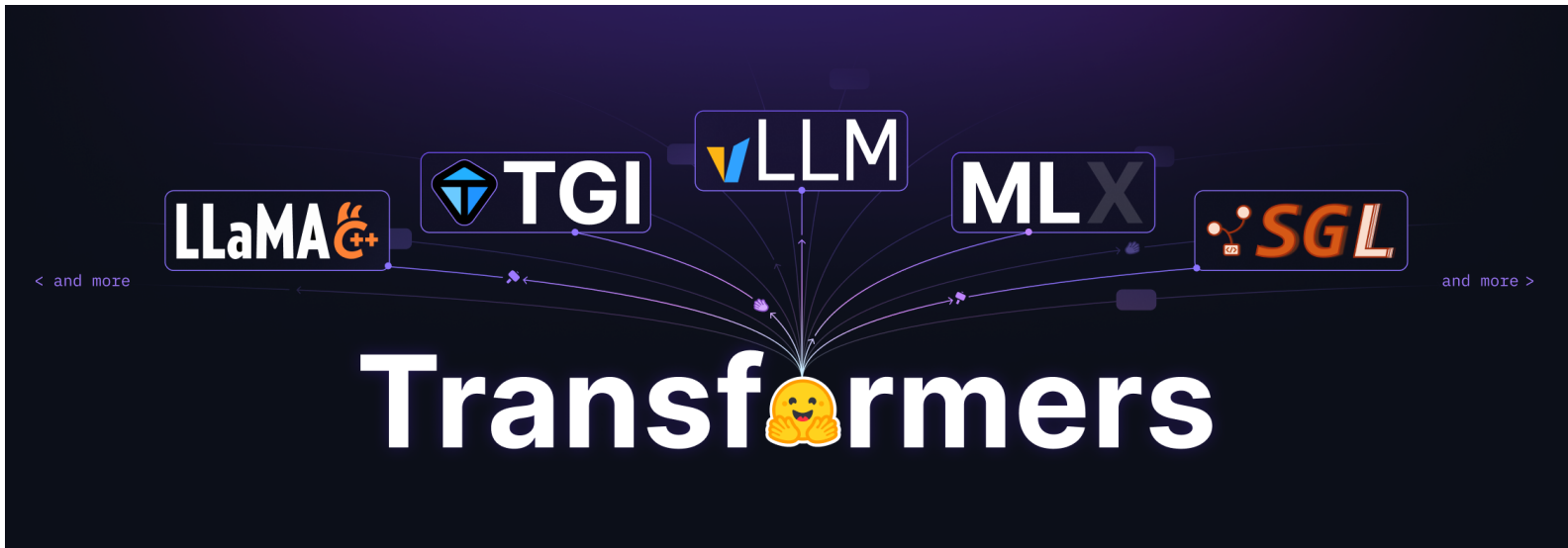
## Stable Diffusion

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](#)



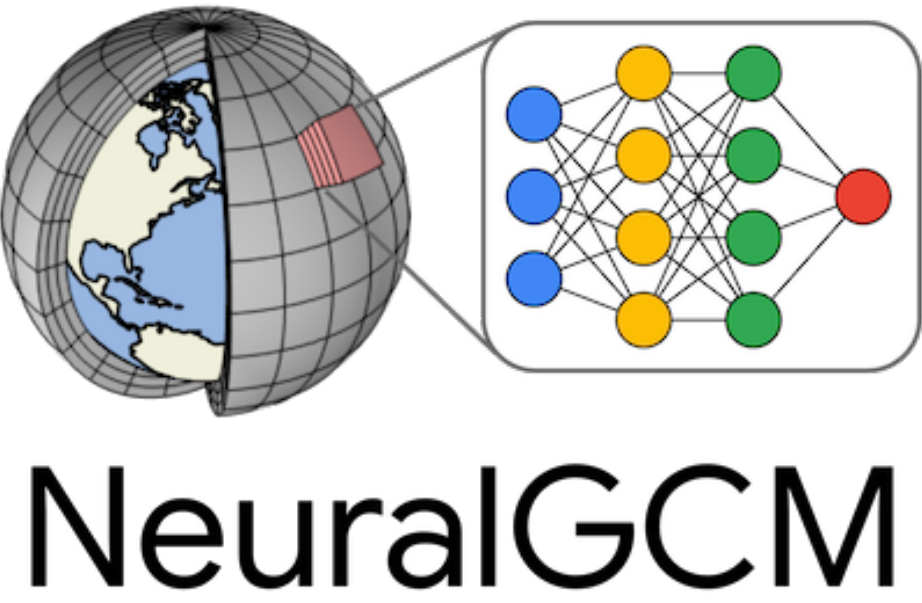
### JAX, M.D.


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

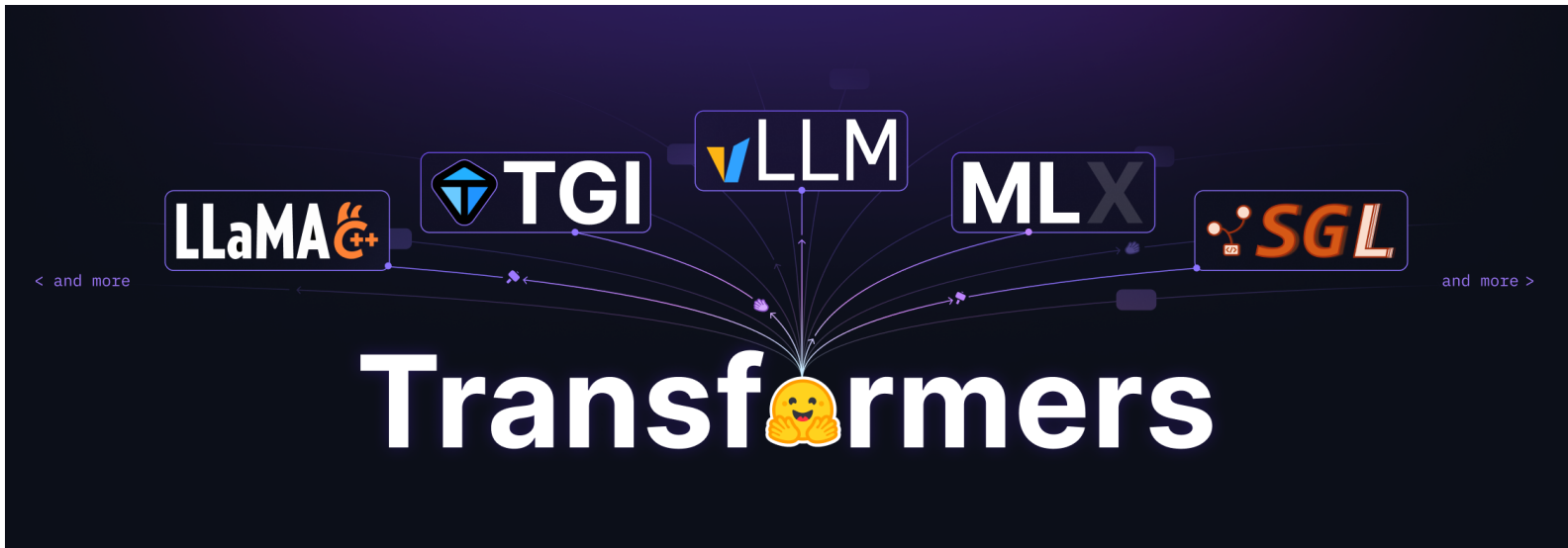
 PYPI [v0.3.0](#)  PYTHON [>=3.10,<3.13](#)  DOCS  PASSING  COVERAGE [94%](#)  SLACK

⚠ 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](#)



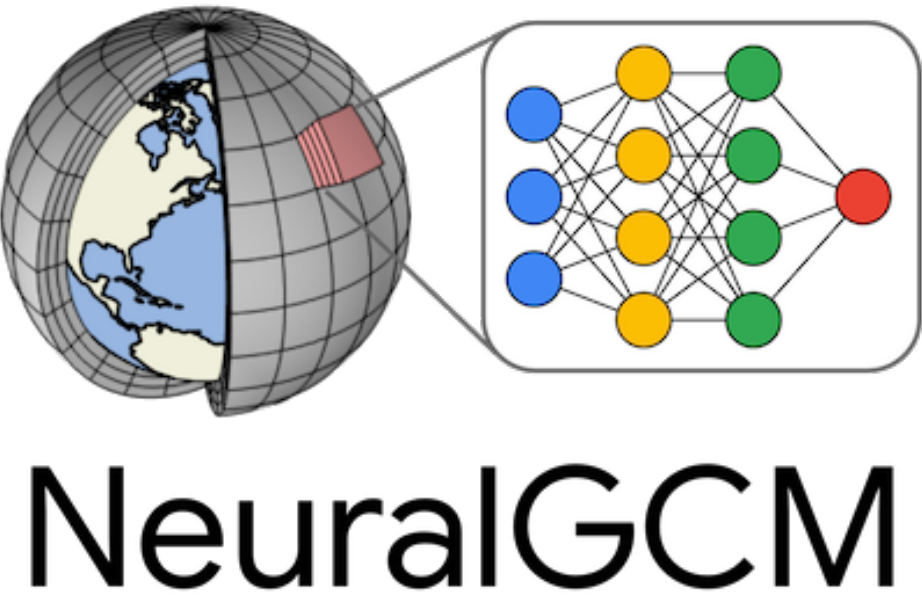
### JAX, M.D.


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) ECP by the Numbers

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.

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) ECP by the Numbers

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.

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



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



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



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



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



# Looking More Deeply at Scientific Code

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

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

> 277 such kernels

**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>

repo status **Active** License **MIT** Ask us **anything** ColPrac **Contributor's Guide** JOSS **10.21105/joss.02018**

latest version **v0.97.7** documentation **stable release** documentation **in development**

Downloads **506/month** Total Downloads **17,790**

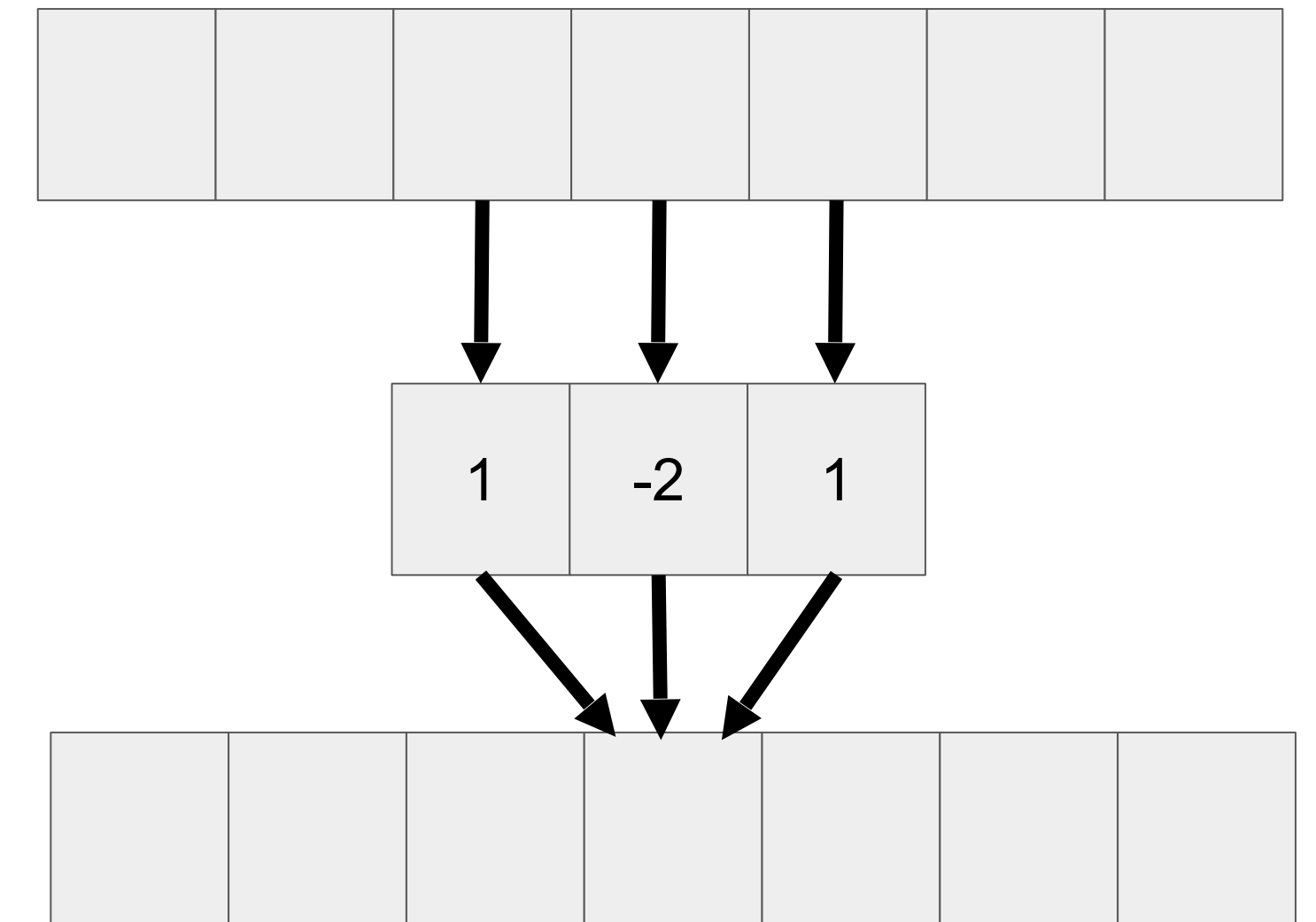
Buildkite CPU+GPU **failing** Docker **invalid**

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.

# 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

Device 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  
}
```

```
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  
}
```



# 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  
}
```

# 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)  
      ...  
    }  
  }  
}
```

# 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 bpnnp_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

- Here on a real code, we observe a 27% speedup on real code, 2.7x on PyTorch cross compilation!



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

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

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

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

Johannes Doerfert  
jdoerfert@lnl.gov  
LLNL  
United States

Jens Domke  
jens.domke@riken.jp  
RIKEN  
Japan

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 and 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% geometric 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 2.7x.

**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>



This work is licensed under a Creative Commons Attribution International 4.0 License.  
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>

### 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

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

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

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

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% geometric 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.<sup>1</sup>

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

```
__void bpnnp_layerforward(...) {  
    __float node[HEIGHT];  
    __float weights[HEIGHT][WIDTH];
```

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

**Necessary Barrier #1**  
**of the read/writes below the sync**  
**ights, hidden)**  
**intersect with the read/writes above the sync**  
**de, input)**  
**threads();**

**Necessary Store #1**  
**[ty][tx] = hidden[index];**

```
__syncthreads();
```

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

# 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);  
}
```



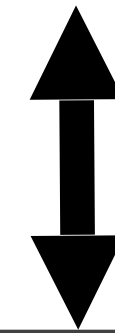
```
parallel_for %i = 0 to N {  
  codeA(%i);  
}  
parallel_for %i = 0 to N {  
  codeB(%i);  
}
```



# 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)

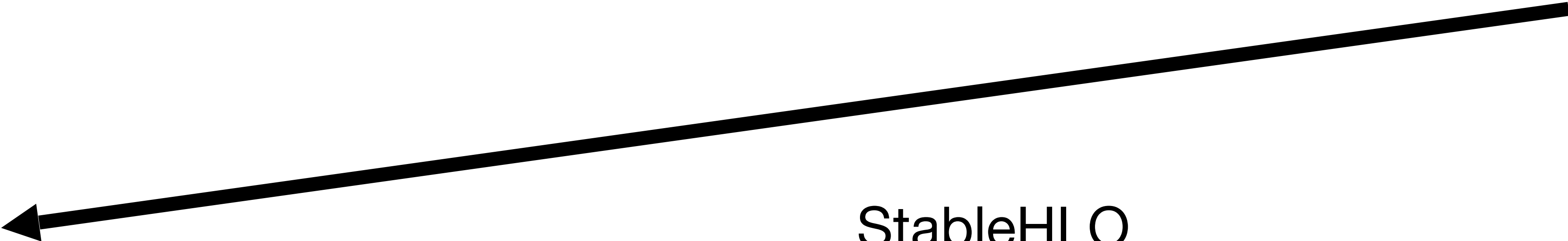
```
scf.for %arg = %c0 .. %c10 {
  ...
}
```

Affine

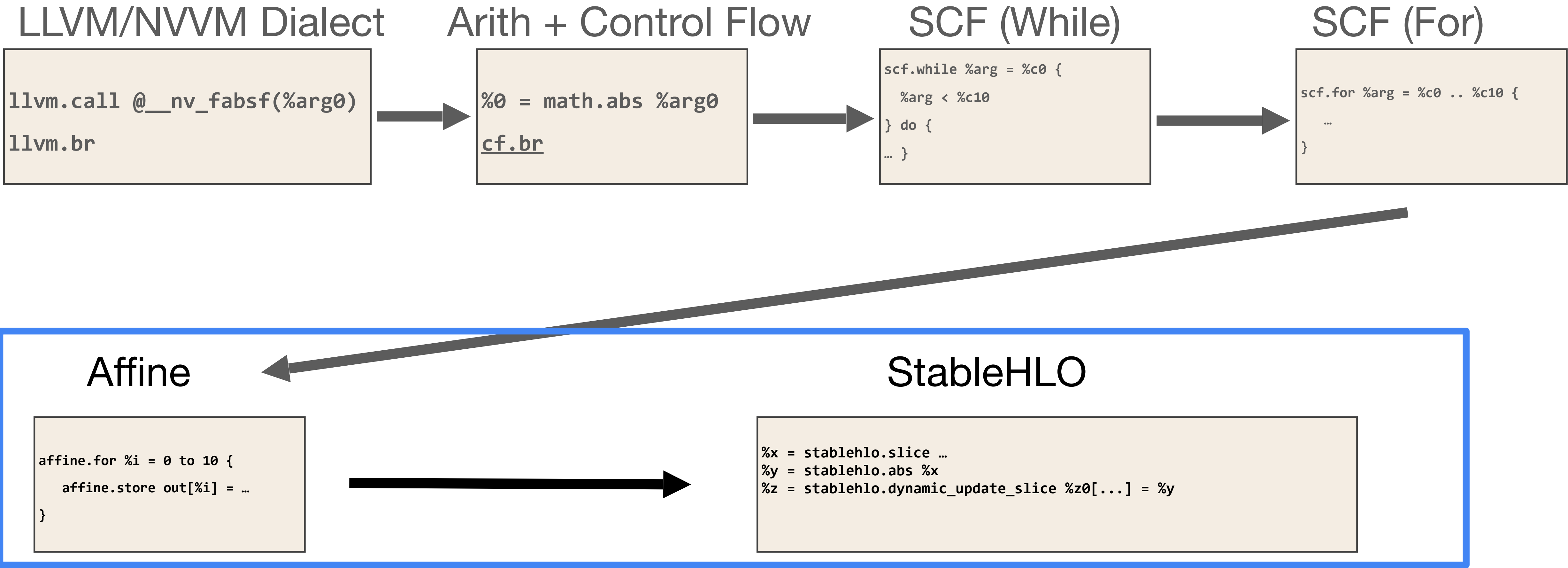
```
affine.for %i = 0 to 10 {
  affine.store out[%i] = ...
}
```

StableHLO

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



# LLVM to StableHLO



# 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 lose the convolution!

- 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.

# 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 3.45x over XLA’s optimization flow across our benchmark suite on various CPU and GPU platforms, with a maximum speedup of 56.26x 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/nnnnnnnn.nnnnnnn>

\*These authors contributed equally.

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions.acm.org](http://permissions.acm.org).

© 2025 Association for Computing Machinery.  
XXXX-XXXX/2025/8-ART \$15.00  
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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

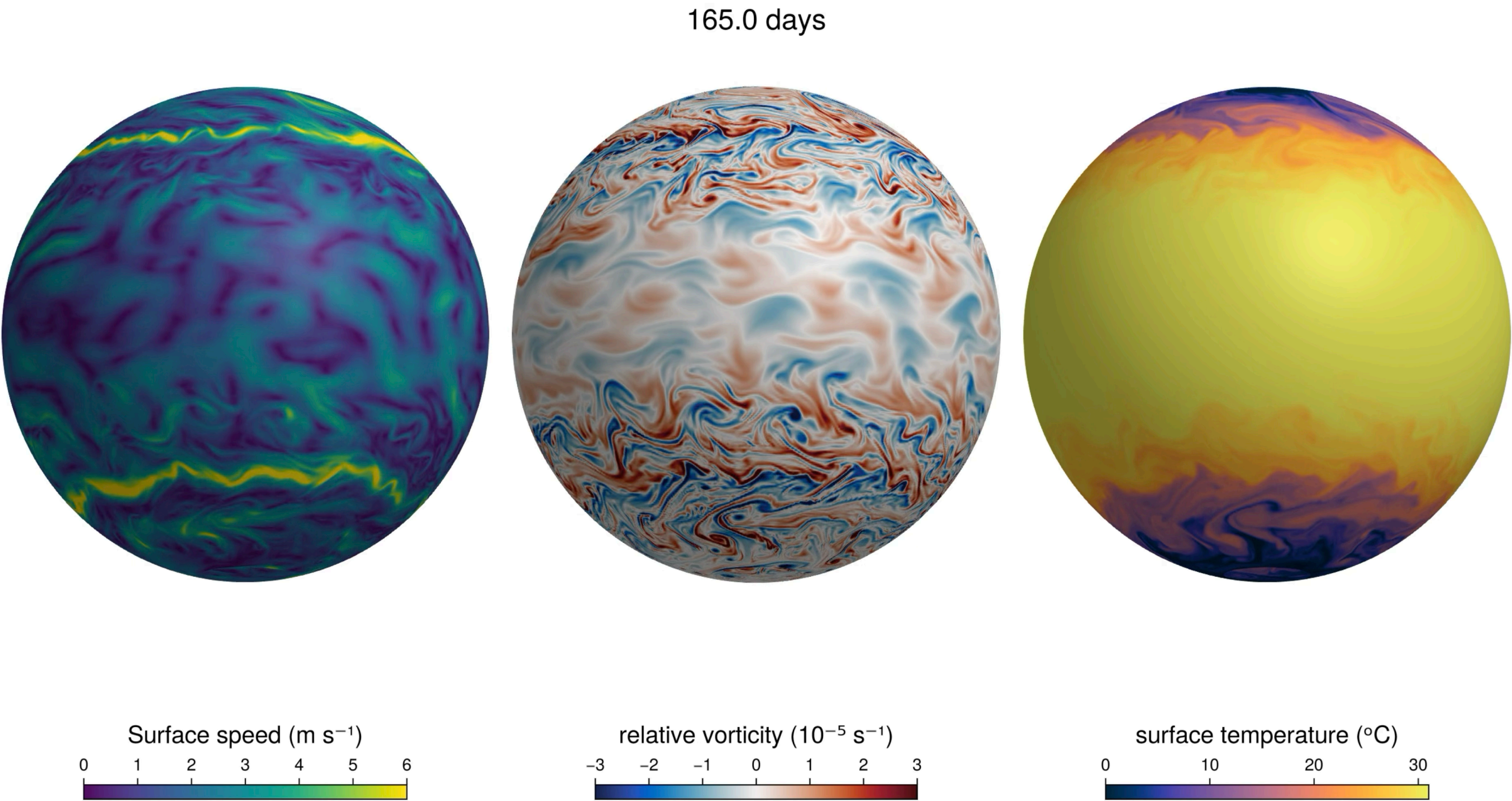
```
%y = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>
%z = stablehlo.convolve %y, tensor<[1.0, -2.0, 1.0]>
```

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

# 56% speedup on JaX ML workloads



# CUDA to Accelerator IR (StableHLO)



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

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

Compilation

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

Raising

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

Multi-Dimensionalization

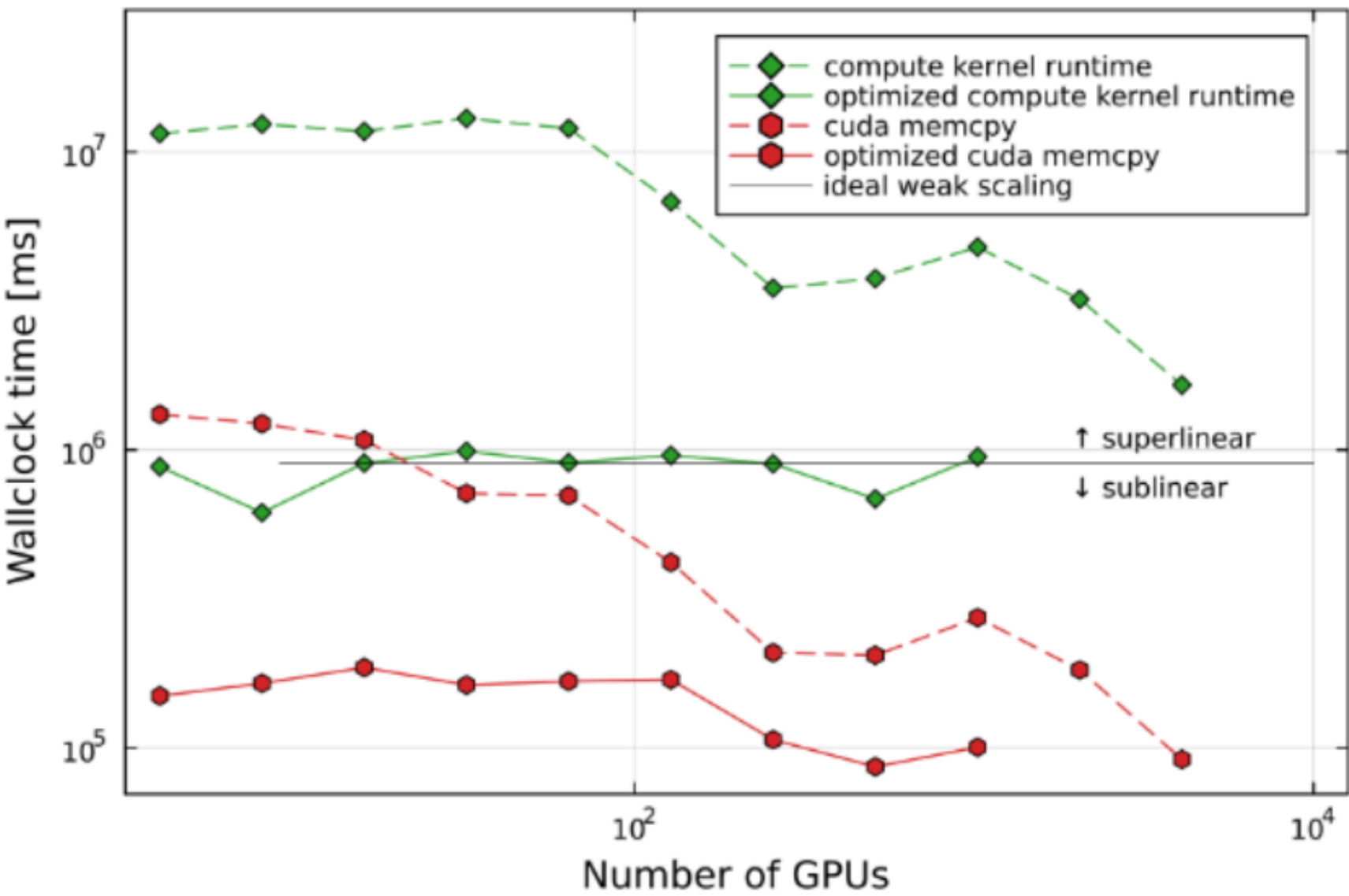
```
%x1 = stablehlo.slice %x[1:98]
%x2 = stablehlo.slice %x[2:99]
%mul = stablehlo.multiply %x2, tensor<2.0>
%add = stablehlo.add %x1, %mu
```

Optimization

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

# Performance Results

- Successfully ran single-node Oceanangians.jl on thousands of distributed accelerators
  - 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 Optimis: 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.



# Conclusions

- Computing hardware is increasingly moving to domain-specific accelerators, leaving existing scientific workloads in the dust
- New tool to extract the existing accelerator-friendly tensor operators written in existing parallel code and run them on distributed accelerators
- Opens the door for moving workloads to where you want to run them, without needing to re-engineer them
- Works generically on LLVM code, with explicit frontends for C++ ([github.com/EnzymeAD/Reactant](https://github.com/EnzymeAD/Reactant)) and Julia ([github.com/EnzymeAD/Reactant.jl](https://github.com/EnzymeAD/Reactant.jl))

