

Google DeepMind

# from Machine Learning Compilers to Science in the Clouds

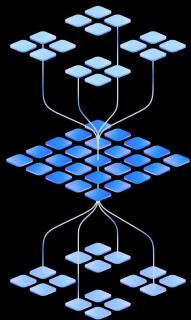
Albert Cohen



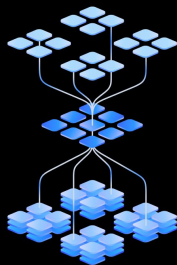


# Automatic Parallelization, Performance Portability: Compilers for ML Models

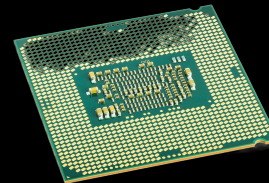
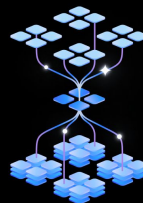
Ultra



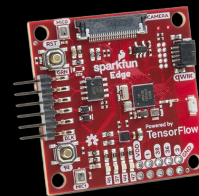
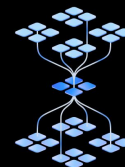
Pro



Flash



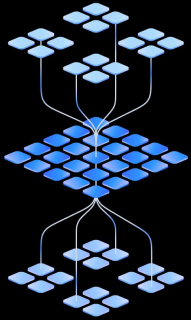
Nano



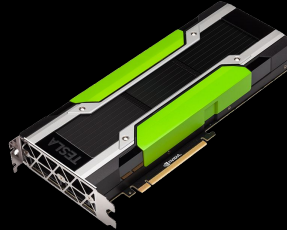
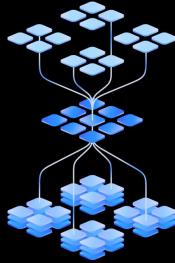


# *What about HPC Performance Portability?*

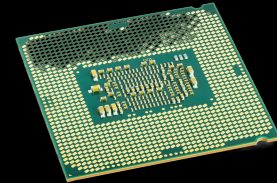
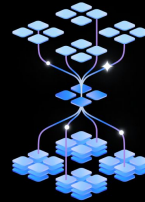
Ultra



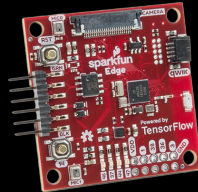
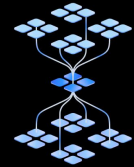
Pro



Flash



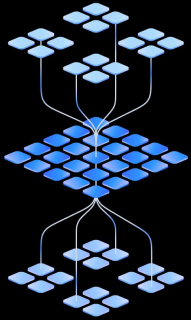
Nano



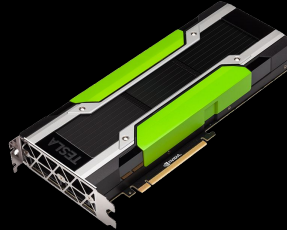
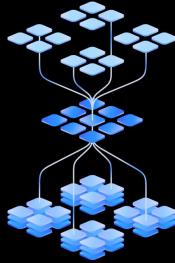


# *What about HPC Performance Portability? On Cloud Systems with HW Accelerators?*

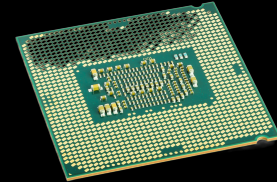
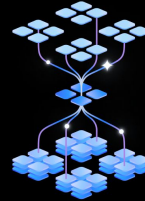
Ultra



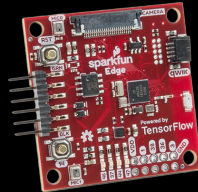
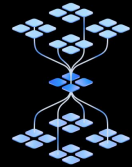
Pro



Flash



Nano



*Time to revisit the role of the compiler!*



# Example: Waves in the Cloud

## Platforms

- NERSC Perlmutter  
1,536 GPU accelerated nodes with  
1 AMD Milan CPU and 4 NVIDIA  
A100 GPUs
- Google Cloud reservation  
1,679 TPU v6e (Trillium)  
1.5 ExaFLOPS (bf16)  
53 TB of HBM  
3.2 TB/s bisection bandwidth

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

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, 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 Montoisson<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>

### Abstract

Ocean and climate models are today limited by compute resources, forcing approximations driven by feasibility rather than theory. They consequently miss important physical processes and decision-relevant regional details. Advances in AI-driven supercomputing — specialized tensor accelerators, AI compiler stacks, and novel distributed systems — offer unprecedented computational power. Yet, scientific applications such as ocean models, often written in Fortran, C++, or Julia and built for traditional HPC, remain largely incompatible with these technologies. This gap hampers performance portability and isolates scientific computing from rapid cloud-based innovation for AI workloads. In this work, we bridge that gap by transpiling a Julia-based ocean model (Oceananigans) using the MLIR compiler infrastructure. This process enables advanced optimizations, deployment on AI hardware (e.g., Google TPUs), and automatic differentiation. Our results demonstrate that cloud-based hardware and software designed for AI workloads can significantly accelerate climate simulations, opening a path for climate modeling to benefit from cutting-edge computational advances.

### 2 Justification for ACM Gordon Bell Prize for Climate Modeling

Automated compiler-based acceleration and retargeting of an ocean model, from GPU-based HPC to TPUs. The model is implemented in Julia with CUDA kernels, while the TPUs only support domain-specific compute graphs. Our demonstration reduces the barrier to entry for scientific computing on cloud systems, which are among today's largest computers.

<sup>†</sup>Correspondence: wsmoses@illinois.edu

Author's Contact Information: 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, 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>,

### 3 Performance Attributes

Performance Attribute	This Submission
Category achievement	scalability
Type of method used	semi-implicit
Results reported on the basis of	whole application except I/O
Precision reported	double precision (GPU) emulated double precision (TPU)
System scale	results measured on full-scale system
Measurement mechanism	timers, FLOP count

### 4 Overview of the Problem

*Climate is governed by planetary fluid dynamics.* This submission focuses on the core of global climate models: simulations of the fluid dynamics of the ocean and atmosphere that dictate the large-scale structure and long-term evolution of Earth's climate. Fluid dynamical processes underpin phenomena like equator-to-pole heat transport, ENSO, the jet stream, air-sea interaction, and the thermohaline circulation, all of which drive variability and set the climate's memory and predictability [13, 19, 36]. Accurately simulating climate requires ocean and atmospheric dynamical cores to solve the governing equations of fluid motion as efficiently and accurately as possible.

*The need for high resolution.* Influential climate processes cover a wide range of interacting spatial scales [18], from planetary (10,000 km), synoptic (1,000 km), tropical cyclones and ocean geostrophic eddies (10 – 200 km), atmospheric mesoscale convective systems and ocean submesoscale processes (1 – 10 km), internal gravity waves, clouds, turbulent diffusion, convective mixing on scales (10 m), and down to the dissipation of kinetic energy (1 mm). Because current global ocean and atmosphere models cannot fully resolve all these small-scale processes, many processes are represented using simplified approximations called parameterizations. However,



Application: **Oceananigans.jl**  
<https://clima.github.io/OceananigansDocumentation>

Simulation of **baroclinic instability** on an Earth-like planet: essential features of ocean and atmosphere interactions

Multiple integrals and solvers:

- implicit vertical diffusion
- hydrostatic pressure anomaly
- vertical velocity
- horizontal velocities
- 5th-order WENO-based advection schemes
- tracers suitable for ultra-high-resolution
- 55-term polynomial approximation to the TEOS10 equation of state for density as a function of oceanic temperature, salinity, and pressure

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

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, 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 Montoisson<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>

## Abstract

Ocean and climate models are today limited by compute resources, forcing approximations driven by feasibility rather than theory. They consequently miss important physical processes and decision-relevant regional details. Advances in AI-driven supercomputing — specialized tensor accelerators, AI compiler stacks, and novel distributed systems — offer unprecedented computational power. Yet, scientific applications such as ocean models, often written in Fortran, C++, or Julia and built for traditional HPC, remain largely incompatible with these technologies. This gap hampers performance portability and isolates scientific computing from rapid cloud-based innovation for AI workloads. In this work, we bridge that gap by transpiling a Julia-based ocean model (Oceananigans) using the MLIR compiler infrastructure. This process enables advanced optimizations, deployment on AI hardware (e.g., Google TPUs), and automatic differentiation. Our results demonstrate that cloud-based hardware and software designed for AI workloads can significantly accelerate climate simulations, opening a path for climate modeling to benefit from cutting-edge computational advances.

## 2 Justification for ACM Gordon Bell Prize for Climate Modeling

Automated compiler-based acceleration and retargeting of an ocean model, from GPU-based HPC to TPUs. The model is implemented in Julia with CUDA kernels, while the TPUs only support domain-specific compute graphs. Our demonstration reduces the barrier to entry for scientific computing on cloud systems, which are among today's largest computers.

<sup>†</sup>Correspondence: wsmoses@illinois.edu

Author's Contact Information: 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, 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>,

## 3 Performance Attributes

Performance Attribute	This Submission
Category achievement	scalability
Type of method used	semi-implicit
Results reported on the basis of	whole application except I/O
Precision reported	double precision (GPU) emulated double precision (TPU)
System scale	results measured on full-scale system
Measurement mechanism	timers, FLOP count

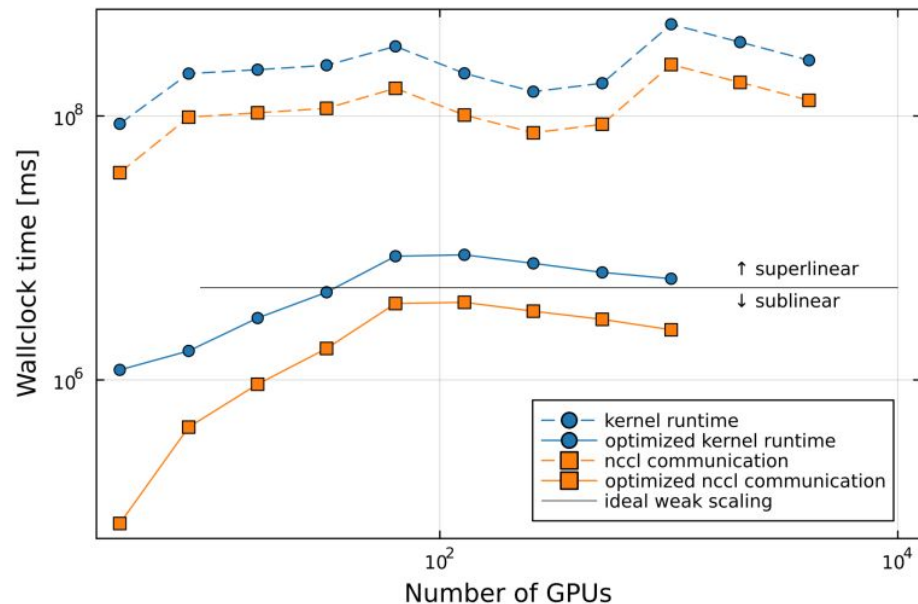
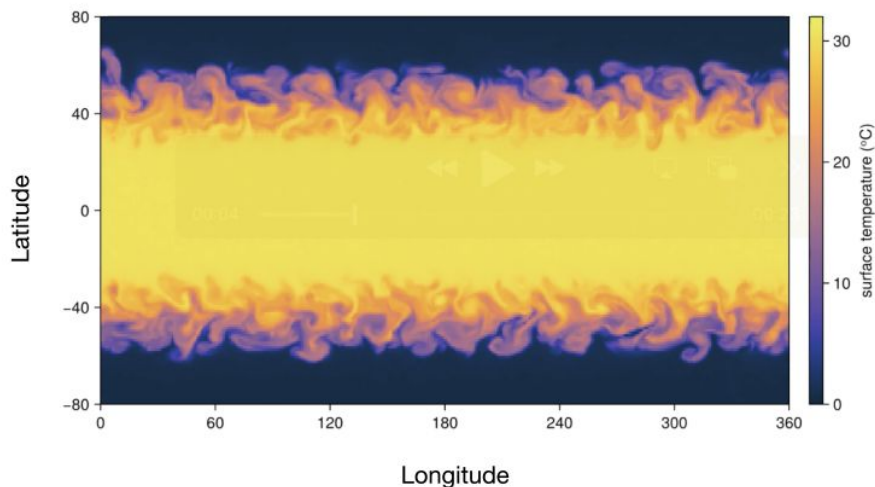
## 4 Overview of the Problem

*Climate is governed by planetary fluid dynamics.* This submission focuses on the core of global climate models: simulations of the fluid dynamics of the ocean and atmosphere that dictate the large-scale structure and long-term evolution of Earth's climate. Fluid dynamical processes underpin phenomena like equator-to-pole heat transport, ENSO, the jet stream, air-sea interaction, and the thermohaline circulation, all of which drive variability and set the climate's memory and predictability [13, 19, 36]. Accurately simulating climate requires ocean and atmospheric dynamical cores to solve the governing equations of fluid motion as efficiently and accurately as possible.

*The need for high resolution.* Influential climate processes cover a wide range of interacting spatial scales [18], from planetary (10,000 km), synoptic (1,000 km), tropical cyclones and ocean geostrophic eddies (10 – 200 km), atmospheric mesoscale convective systems and ocean submesoscale processes (1 – 10 km), internal gravity waves, clouds, turbulent diffusion, convective mixing on scales (10 m), and down to the dissipation of kinetic energy (1 mm). Because current global ocean and atmosphere models cannot fully resolve all these small-scale processes, many processes are represented using simplified approximations called parameterizations. However,

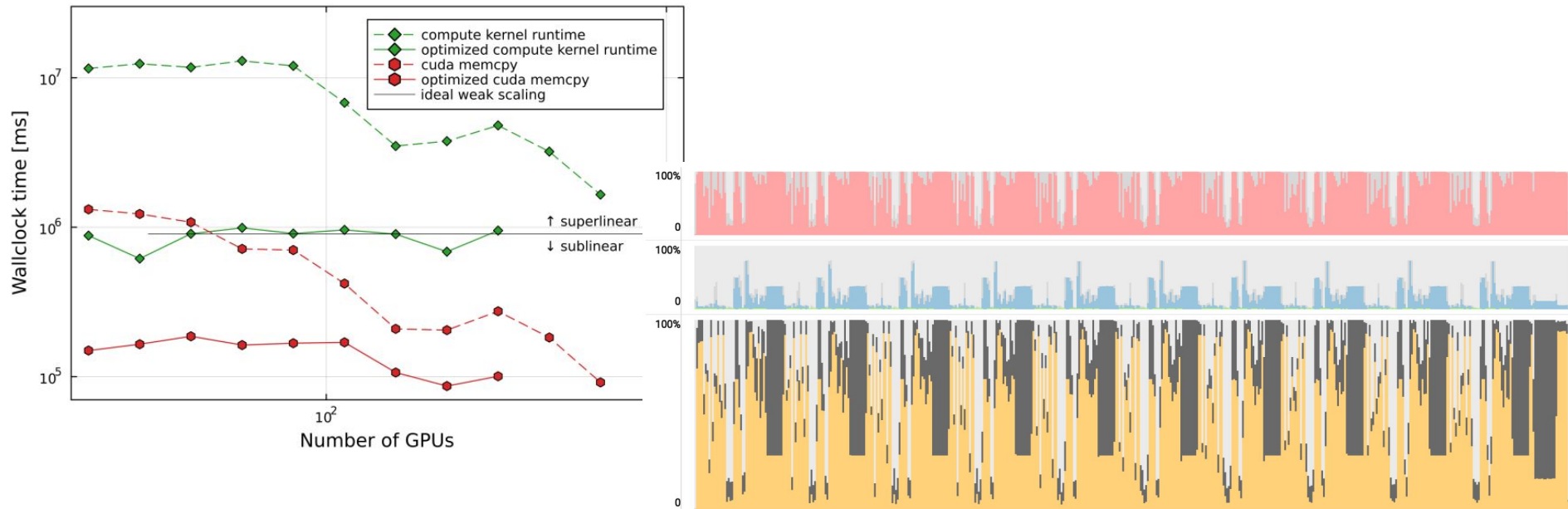


# Weak Scaling Experiments: GPU / Placement and Collectives



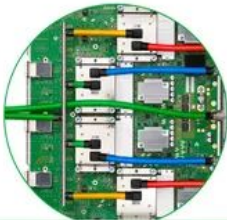




# Weak Scaling Experiments: GPU / Kernels and Host-Device Communication



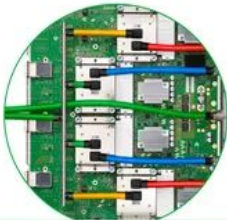




# What about TPUs? And Why?

			
	TPU v4	TPU v5p	Ironwood
	2022	2023	2025
Pod Size (chips)	4096	8960	9216
HBM Bandwidth/ Capacity	32 GB @ 1.2 TBps HBM	95 GB @ 2.8 TBps HBM	192 GB @ 7.4 TBps HBM
Peak Flops per chip	275 TFLOPS	459 TFLOPS	4614 TFLOPS



# What about TPUs? And Why?

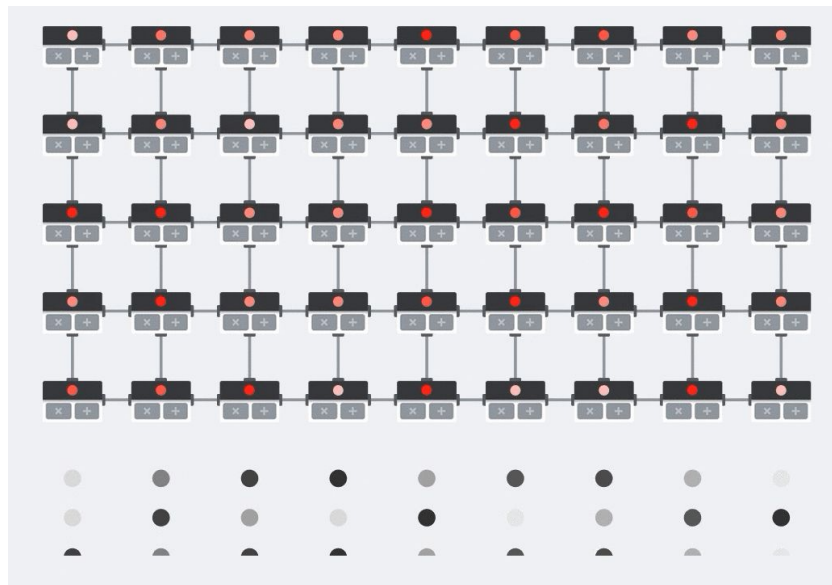
			
	TPU v4	TPU v5p	Ironwood
	2022	2023	2025
Pod Size (chips)	4096	8960	9216
HBM Bandwidth/ Capacity	32 GB @ 1.2 TBps HBM	95 GB @ 2.8 TBps HBM	192 GB @ 7.4 TBps HBM
Peak Flops per chip	275 TFLOPS	459 TFLOPS	4614 TFLOPS

What FLOPS?  
(Osaki emulation)



# What about TPUs? Energy Efficiency and Cost

Systolic matrix multiplication is more energy efficient on TPU than on CPU or GPU





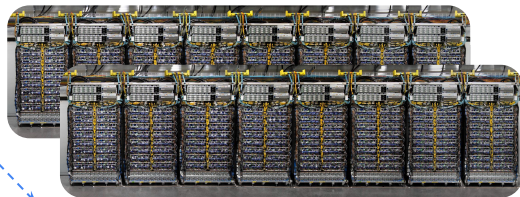
# What about TPUs? Energy Efficiency and Scale



## TPU v2

- Domain-specific AI supercomputing
- 256 chips distributed shared memory

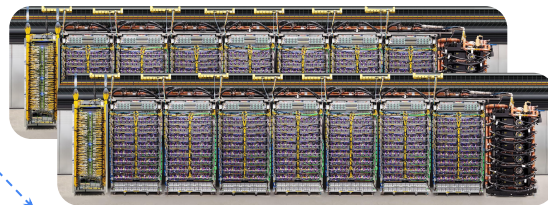
8x



## TPU v4

- Optically reconfigurable 3D Torus
- 4k chips with distributed shared memory

20x



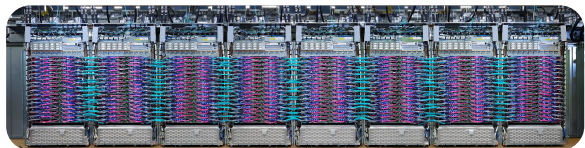
## TPU v5p

- Programmable Sparsecores for embeddings
- 9k chips with distributed shared memory



## TPU v3

- Liquid cooling
- 1k chips distributed shared memory



## TPU v5e

- Efficient and scalable training and serving
- 256 chips, scalable to 10s of k chips



## TPU v6e

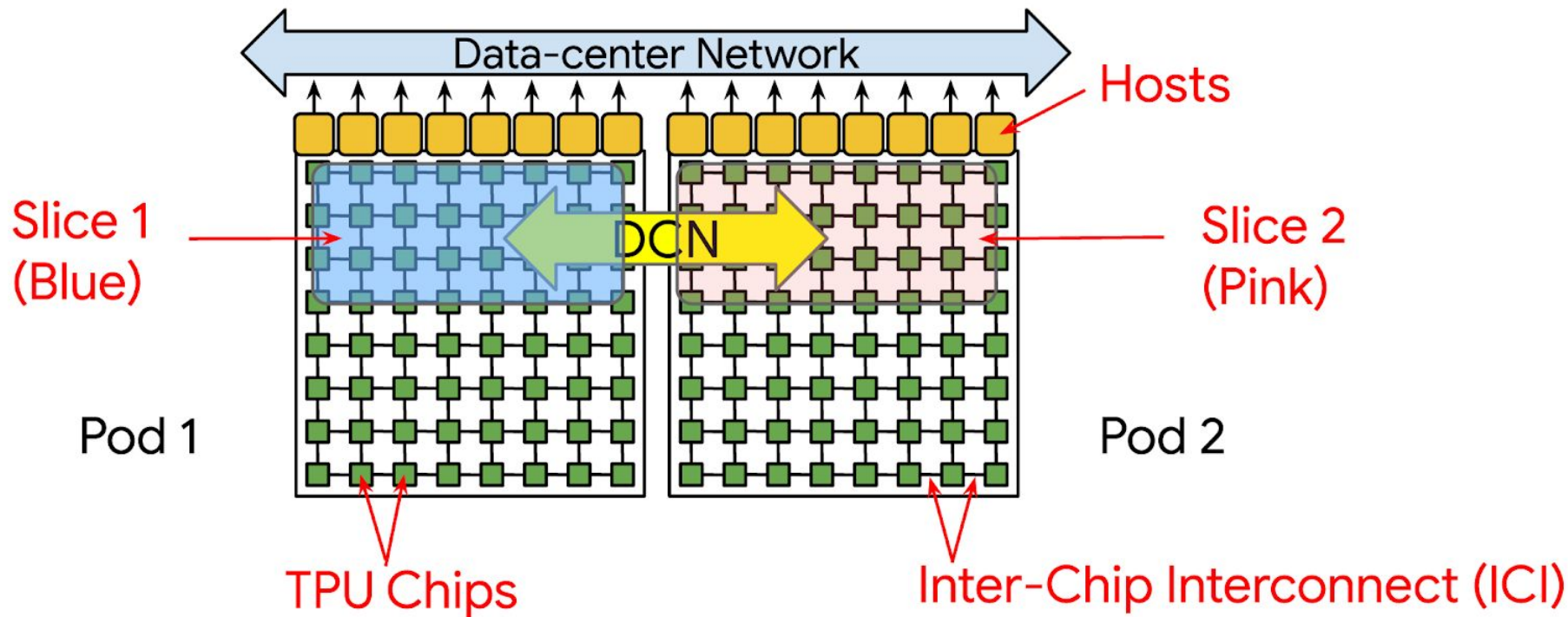
- 67% more energy efficient than 5e
- 256 chips, scalable to 100 k chips





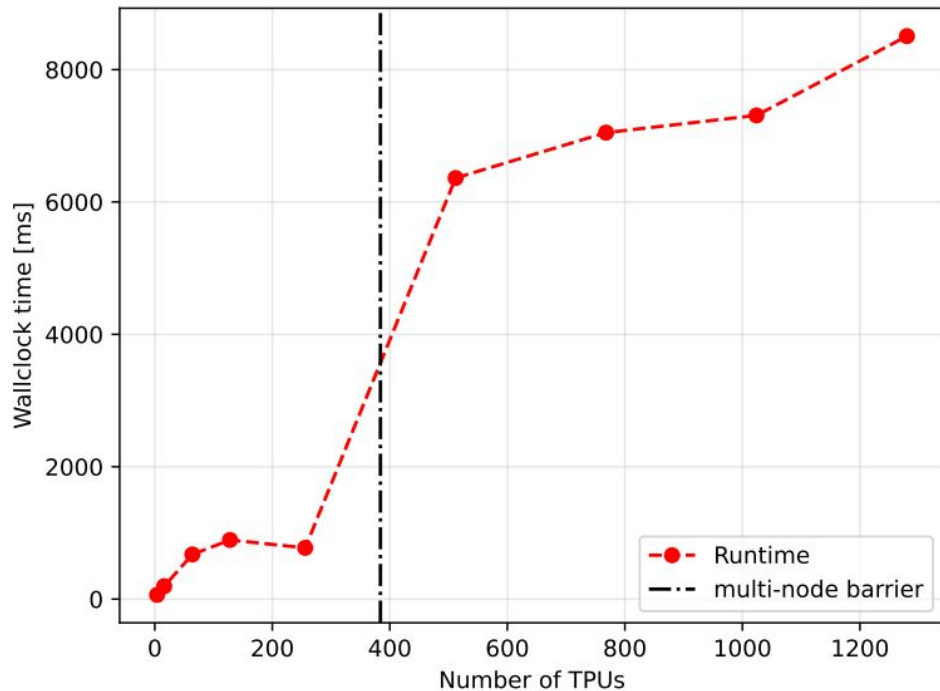
*“Look mom, no MPI!”*

Ad-hoc runtimes and high-level composable abstractions





# Weak Scaling Experiments on TPU



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.**



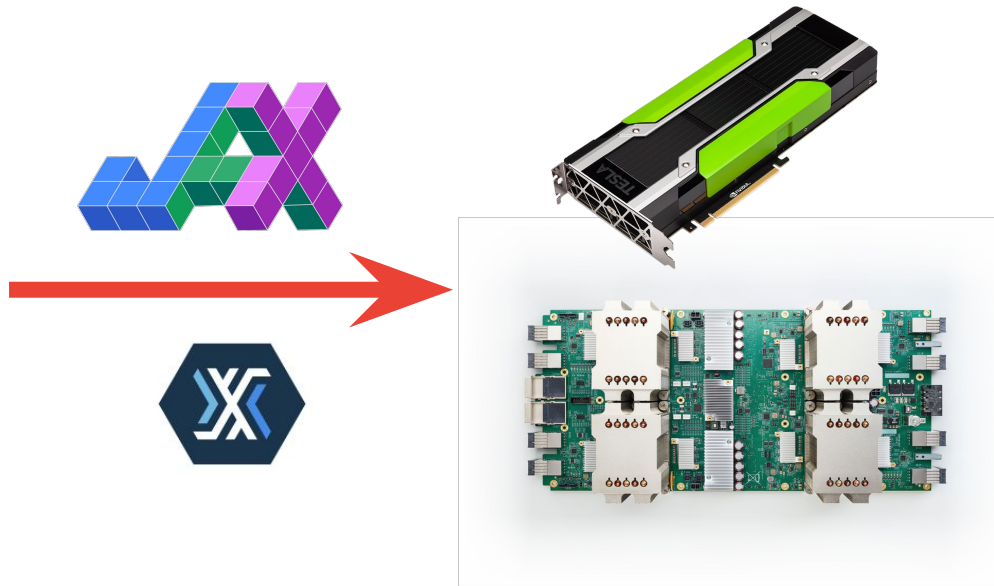
# How Did We Get There?

```
import jax.numpy as np
from jax import jit, grad, vmap

def predict(params, inputs):
    for W, b in params:
        outputs = np.dot(inputs, W) + b
        inputs = np.tanh(outputs)
    return outputs

def loss(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.sum((preds - targets) ** 2)
```

```
gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss)))
```



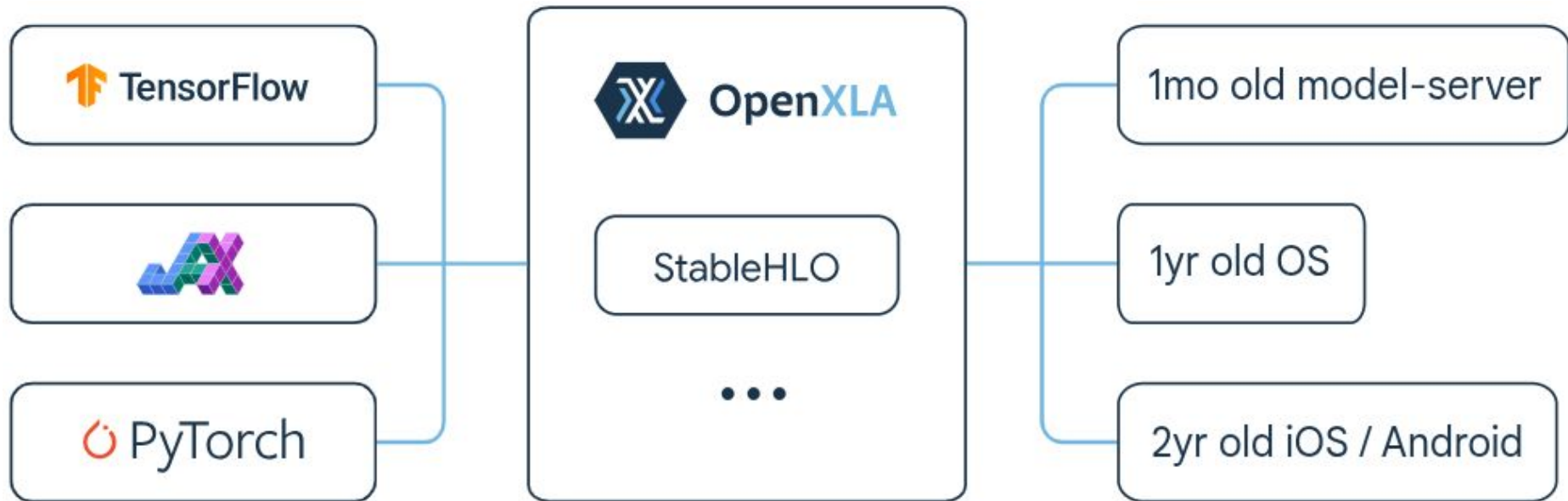
JAX is an extensible system for **composable function transformations** of Python+NumPy code, with **computations staged to XLA**





# StableHLO

<https://openxla.org/stablehlo>





# XLA / HLO / StableHLO

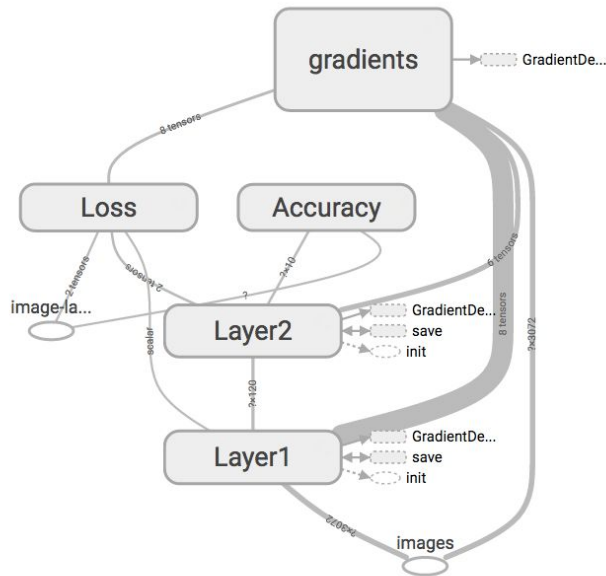


## Folklore: Go Domain-Specific!

→ Domain-Specific Languages (DSLs) expose the right **abstractions** to make **automatic code generation** possible and effective

→ Also **domain-specific accelerators**

HLO: XLA Compute Graph, Static Dataflow





# XLA / HLO / StableHLO



## Folklore: **Go Domain-Specific!**

→ Domain-Specific Languages (DSLs) expose the right **abstractions** to make **automatic code generation** possible and effective

→ Also **domain-specific accelerators**

- **JIT / AOT compiler for linear algebra**
- Multiple backends: CPUs, GPUs, TPUs
- Eliminate dispatch overhead
- Fuse operations:  
avoid round trips to memory
- Specialization, global buffer analysis, vectorization, unrolling, etc.



# What About HPC?

## Folklore: Go Domain-Specific!

→ Domain-Specific Languages (DSLs) expose the right **abstractions** to make **automatic code generation** possible and effective

→ Raising the level of abstraction is harder than riding the abstraction lowering slope



## Progressive Raising in Multi-level IR

Lorenzo Chelini  
TU Eindhoven  
Eindhoven, The Netherlands  
l.chelini@tue.nl

Andi Drebes  
Inria and École Normale Supérieure  
Paris, France  
andi@programmierforen.de

Oleksandr Zinenko  
Google  
Paris, France  
zinenko@google.com

Albert Cohen  
Google  
Paris, France  
albertcohen@google.com

Nicolas Vasilache  
Google  
Zurich, Switzerland  
ntv@google.com

Tobias Grosser  
University of Edinburgh  
Edinburgh, UK  
tobias.grosser@ed.ac.uk

Henk Corporaal  
TU Eindhoven  
Eindhoven, The Netherlands  
h.corporaal@tue.nl

**Abstract**—Multi-level intermediate representations (IR) show great promise for lowering the design costs for domain-specific compilers by providing a reusable, extensible, and non-opinionated framework for expressing domain-specific and high-level abstractions directly in the IR. But, while such frameworks support the progressive lowering of high-level representations to low-level IR, they do not raise in the opposite direction. Thus, the entry point into the compilation pipeline defines the highest level of abstraction for all subsequent transformations, limiting the set of applicable optimizations, in particular for general-purpose languages that are not semantically rich enough to model the required abstractions.

We propose *Progressive Raising*, a complementary approach to the progressive lowering in multi-level IRs that raises from lower to higher-level abstractions to leverage domain-specific transformations for low-level representations. We further introduce *Multi-level Tactics*, our declarative approach for progressive raising, implemented on top of the MLIR framework, and demonstrate the progressive raising from affine loop nests specified in a

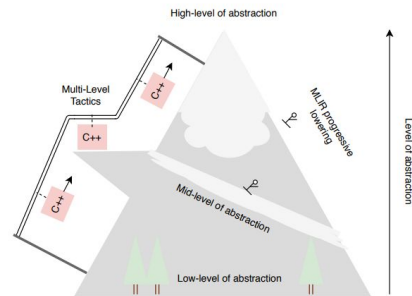


Fig. 1. Multi-level Tactics lifts general-purpose languages to higher-abstraction levels to enable effective domain-specific compilation via progressive lowering.



# What's Wrong?





# What's Wrong?



Lifting, isn't this just as hard as automatic parallelization?  
What about abstraction penalty?  
Control for performance engineers?



# Kernel Programming to the Rescue

## Flurry of GPU acceleration options

CUDA Kernels / OpenCL-C

SYCL

CUTLASS

Triton (PyTorch)

Pallas (JAX)

Turbine (AMD)

Mojo (Modular)

CuTile (Nvidia)

and more coming and going...

## More broadly

*“If high level fails, try lower level”*

Folklore: high-level language



abstraction penalty

Motivations: escape hatch for...

- Performance tricks
- Extra expressiveness  
e.g. ragged or sparse tensors
- Quick experiments



# Performance Engineering & Compilers Unite!

## User-Schedulable Languages

### Input: Algorithm

```
blurx(x,y) = in(x-1,y)
             + in(x,y)
             + in(x+1,y)

out(x,y) = blurx(x,y-1)
           + blurx(x,y)
           + blurx(x,y+1)
```

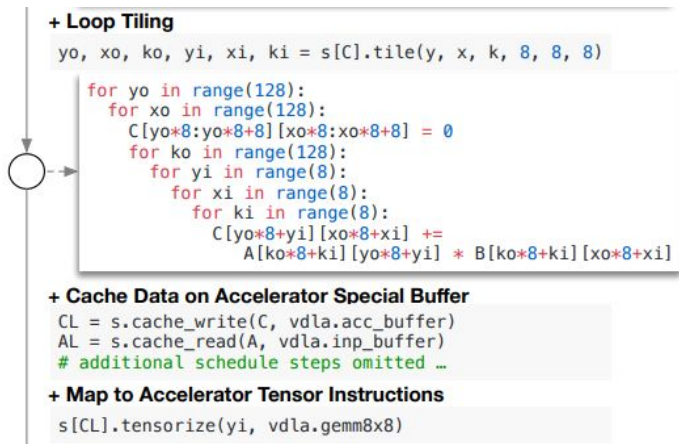
### Input: Schedule

```
blurx: split x by 4 → xo, xi
        vectorize: xi
        store at out.xo
        compute at out.yi
```

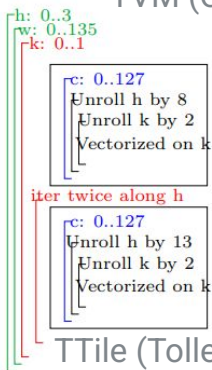
```
out: split x by 4 → xo, xi
      split y by 4 → yo, yi
      reorder: yo, xo, yi, xi
      parallelize: yo
      vectorize: xi
```

Halide (Ragan-Kelley et.al. 2013)

Also rewrite systems with semantic guarantees: Lift, Elevate, Rise



TVM (Chen et.al. 2018)



TTile (Tollenaere et.al. 2021)

```
tc::IslKernelOptions::makeDefaultM
.scheduleSpecialize(false)
.tile({4, 32})
.mapToThreads({1, 32})
.mapToBlocks({64, 128})
.useSharedMemory(true)
.usePrivateMemory(true)
.unrollCopyShared(false)
.unroll(4);
```

TC (Vasilache et.al. 2018)

```
mm = MatMul(M,N,K)(GL, GL, GL)(Kernel)
mm
// resulting intermediate specs below
.tile(128,128) // MatMul(128,128,K)(GL, GL, GL)(Kernel)
.to(Block) // MatMul(128,128,K)(GL, GL, GL)(Block )
.load(A, SH, _) // MatMul(128,128,K)(SH, GL, GL)(Block )
.load(A, SH, _) // MatMul(128,128,K)(SH, SH, GL)(Block )
.tile(64,32) // MatMul(64, 32, K)(SH, SH, GL)(Block )
.to(Warp) // MatMul(64, 32, K)(SH, SH, GL)(Warp )
.tile(8,8) // MatMul(8, 8, K)(SH, SH, GL)(Warp )
.to(Thread) // MatMul(8, 8, K)(SH, SH, GL)(Thread)
.load(A, RF, _) // MatMul(8, 8, K)(RF, SH, GL)(Thread)
.load(B, RF, _) // MatMul(8, 8, K)(RF, RF, GL)(Thread)
.tile(1,1) // MatMul(1, 1, K)(RF, RF, GL)(Thread)
.done(dot.cu) // invoke codegen, emit dot micro-kernel
```

Fireiron (Hagedorn et.al. 2020)



# User-Schedulable Languages... Actually Time-Tested

```
# Avoid spurious versioning
addContext(C1L1,'ITMAX'=9')
addContext(C1L1,'doloop_ub>=ITMAX')
addContext(C1L1,'doloop_ub<=ITMAX')
addContext(C1L1,'N'=500')
addContext(C1L1,'M'=500')
addContext(C1L1,'MMIN>=500')
addContext(C1L1,'MMIN<=M')
addContext(C1L1,'MMIN<=N')
addContext(C1L1,'M<=N')
addContext(C1L1,'M<=N')

# Move and shift calc3 backwards
shift(enclose(C3L1),{'1','0','0'})
shift(enclose(C3L10),{'1','0'})
shift(enclose(C3L11),{'1','0'})
shift(C3L12,{'1'})
shift(C3L13,{'1'})
shift(C3L14,{'1'})
shift(C3L15,{'1'})
shift(C3L16,{'1'})
shift(C3L17,{'1'})
motion(enclose(C3L1),BLOOP)
motion(enclose(C3L10),BLOOP)
motion(enclose(C3L11),BLOOP)
motion(C3L12,BLOOP)
motion(C3L13,BLOOP)
motion(C3L14,BLOOP)
motion(C3L15,BLOOP)
motion(C3L16,BLOOP)
motion(C3L17,BLOOP)

# Peel and shift to enable fusion
peel(enclose(C3L1,2),'3')
peel(enclose(C3L1_2_2), 'N-3')
peel(enclose(C3L1_2_1,1), '3')
peel(enclose(C3L1_2_1_2,1), 'M-3')
peel(enclose(C1L1,2), '2')
peel(enclose(C1L1_2_2), 'N-2')
peel(enclose(C1L1_2_1,1), '2')
peel(enclose(C1L1_2_1_2,1), 'M-2')
peel(enclose(C2L1,2), '1')
peel(enclose(C2L1_2_2), 'N-1')
peel(enclose(C2L1_2_1,1), '3')
peel(enclose(C2L1_2_1_2,1), 'M-3')
shift(enclose(C1L1_2_1_2_1), {'0','1','1'})
shift(enclose(C2L1_2_1_2_1), {'0','2','2'})

# Double fusion of the three nests
motion(enclose(C2L1_2_1_2_1),TARGET_2_1_2_1)
motion(enclose(C1L1_2_1_2_1),C2L1_2_1_2_1)
motion(enclose(C3L1_2_1_2_1),C1L1_2_1_2_1)

# Register blocking and unrolling (factor 2)
time_stripmine(enclose(C3L1_2_1_2_1,2),2,2)
time_stripmine(enclose(C3L1_2_1_2_1,1),4,2)
interchange(enclose(C3L1_2_1_2_1,2))
time_peel(enclose(C3L1_2_1_2_1,3),4,'2')
time_peel(enclose(C3L1_2_1_2_1_2,3),4,'N-2')
time_peel(enclose(C3L1_2_1_2_1_2_1,1),5,'2')
time_peel(enclose(C3L1_2_1_2_1_2_1_2,1),5,'M-2')
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,2))
fullunroll(enclose(C3L1_2_1_2_1_2_1_2_1,1))
```

URUK (Girbal et.al. 2006)

**Distribution** Distribute loop at depth  $L$  over the statements  $D$ , with statement  $s_p$  going into  $r_p$ <sup>th</sup> loop.

Requirements:  $\forall s_p, s_q \ s_p \in D \wedge s_q \in D \Rightarrow \text{loop}(f_p^L) \wedge L \leq \text{csl}(s_p, s_q)$

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $[f_p^1, \dots, f_p^{L-1}]$ , syntactic( $r_p$ ),  $f_p^L, \dots, f_p^n$

**Statement Reordering** Reorder statements  $D$  at level  $L$  so that new position of statement  $s_p$  is  $r_p$ .

Requirements:  $\forall s_p, s_q \ s_p \in D \wedge s_q \in D \Rightarrow \text{syntactic}(f_p^L) \wedge L \leq \text{csl}(s_p, s_q) + 1 \wedge$

$(L \leq \text{csl}(s_p, s_q) \Leftrightarrow r_p = r_q)$

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $[f_p^1, \dots, f_p^{L-1}]$ , syntactic( $r_p$ ),  $f_p^{L+1}, \dots, f_p^n$

**Fusion** Fuse the loops at level  $L$  for the statements  $D$  with statement  $s_p$  going into the  $r_p$ <sup>th</sup> loop.

Requirements:  $\forall s_p, s_q \ s_p \in D \wedge s_q \in D \Rightarrow \text{syntactic}(f_p^{L-1}) \wedge \text{loop}(f_p^L) \wedge L - 2 \leq \text{csl}(s_p, s_q) + 2 \wedge$   
 $(L - 2 < \text{csl}(s_p, s_q) + 2 \Rightarrow r_p = r_q)$

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $[f_p^1, \dots, f_p^{L-2}]$ , syntactic( $r_p$ ),  $f_p^{L-1}, f_p^L, f_p^{L+1}, \dots, f_p^n$

**Unimodular Transformation** Apply a  $k \times k$  unimodular transformation  $U$  to a perfectly nested loop containing statements  $D$  at depth  $L \dots L+k$ . Note: Unimodular transformations include loop interchange, skewing and reversal [Ban90, WL91b].

Requirements:  $\forall i, s_p, s_q \ s_p \in D \wedge s_q \in D \wedge L \leq i \leq L+k \Rightarrow \text{loop}(f_p^i) \wedge L+k \leq \text{csl}(s_p, s_q)$

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $[f_p^1, \dots, f_p^{L-1}]$ ,  $U[f_p^L, \dots, f_p^{L+k}]^\top$ ,  $f_p^{L+k+1}, \dots, f_p^n$

**Strip-mining** Strip-mine the loop at level  $L$  for statements  $D$  with block size  $B$

Requirements:  $\forall s_p, s_q \ s_p \in D \wedge s_q \in D \Rightarrow \text{loop}(f_p^L) \wedge L \leq \text{csl}(s_p, s_q) \wedge B$  is a known integer constant

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $[f_p^1, \dots, f_p^{L-1}]$ ,  $B(f_p^L \text{ div } B)$ ,  $f_p^{L+1}, \dots, f_p^n$

**Index Set Splitting** Split the index set of statements  $D$  using condition  $C$

Requirements:  $C$  is affine expression of symbolic constants and indexes common to statements  $D$ .

Transformation:  $\forall s_p \in D$ , replace  $T_p$  by  $(T_p \mid C) \cup (T_p \mid \neg C)$

Omega (Kelly & Pugh, 1991)

Common ancestor: the Alpha language for high-level synthesis

1992-... Le Verge, Quinton, Rajopadhye, Risset, Wilde...



# Schedules as Pragmas

## A Language for the Compact Representation of Multiple Program Versions [LCPC 2005]

Sebastien Donadio<sup>1,2</sup>, James Brodman<sup>4</sup>, Thomas Roeder<sup>5</sup>, Kamen Yotov<sup>5</sup>,  
Denis Barthou<sup>2</sup>, Albert Cohen<sup>3</sup>, María Jesús Garzarán<sup>4</sup>, David Padua<sup>4</sup>,  
and Keshav Pingali<sup>5</sup>

<sup>1</sup> BULL SA

<sup>2</sup> University of Versailles St-Quentin-en-Yvelines

<sup>3</sup> INRIA Futurs

<sup>4</sup> University of Illinois at Urbana-Champaign

<sup>5</sup> Cornell University

**Abstract.** As processor complexity increases compilers tend to deliver suboptimal performance. Library generators such as ATLAS, FFTW and SPIRALz overcome this issue by empirically searching in the space of possible program versions for the one that performs the best. Empirical search can also be applied by programmers, but because they lack a tool to automate the process, programmers need to manually re-write the application in terms of several parameters whose best value will be determined by the empirical search in the target machine.

In this paper, we present the design of an annotation language, meant to be used either as an intermediate representation within library generators or directly by the programmer. This language that we call *X* represents parameterized programs in a compact and natural way. It provides an powerful optimization framework for high performance computing.

## Xlanguage

### Description of Xlanguage

- begin/end directives:

```
#pragma xlang begin
for (i=.; i<.; i++) {
  ..
}
```

```
#pragma xlang end
```

They surround the outermost loop where all transformations will apply. All loops must be normalized with an affectation `i=.`, a condition of the kind `i<.` and an increment either like `i++` or `i+=.`.

- transformations: all loops are identified by their loop counter. All transformations can be put right after the outermost loop and they will be applied in sequence.

```
#pragma xlang transform interchange(i,j) interchange loops i and j. Loops i and j must be perfectly nested (either i in j or j in i).
```

```
#pragma xlang transform unroll(i,n) makes a partial unroll of loop i, with unroll factor n (n>0)
```

```
#pragma xlang transform fission(i) fissions loop i. Several fissions are possible if the loop has more than 2 statements. All solutions are explored.
```

```
#pragma xlang transform fusion(i,j) fusions loops i and j that must be in sequence. Lower and upper bounds must be the same (syntactically), and the increment as well.
```

```
#pragma xlang transform stripmine(i,ii,n) strip mines loop i (with factor n) and creates a new inner loop ii.
```

```
#pragma xlang transform tile(i,ii,n) same as strip mine, but the new loop ii is created at the innermost position of the loopnest.
```

```
#pragma xlang transform [ transformation1, ... ] applies one of the transformations of the list (this is an OR). Transformations can be one of the previous ones (unroll(i,n), fission(i,j), fusion(i,j),...).
```

```
#pragma xlang transform nop does nothing. Useful for OR construct
```

- parameters

```
#pragma xlang parameter X [val1,val2,...] defines X with possible values val1,val2...
```

### Documentation

Description of Xlanguage can be found in the paper. This version has only a subset of the features.

**Combining Experimental Search and Optimization.** Julien Jaeger and Denis Barthou. In *ACM/IEEE Int. Symp. on Machine Learning Approaches to Automatic Scheduling*, January 2009. [ [bib](#) | [pdf](#) ]

**Loop Optimization using Adaptive Scheduling.** Barthou, Sebastien Donadio, Alexandre Cohen, and Keshav Pingali. In *ACM/IEEE Int. Symp. on Machine Learning Approaches to Automatic Scheduling*, San Jose, California, March 2007.

**Iterative Compilation with Kernel Search.** Alexandre Duchateau, William Jalby, and William J. Rorh. In *Compilers for Parallel Computing*, pages 173-189, New Orleans, Louisiana, 2006.

**A Language for the Compact Representation of Multiple Program Versions.** Sebastien Donadio, James Brodman, Kamen Yotov, Denis Barthou, Albert Cohen, Maria Garzarán, David Padua, and Keshav Pingali. In *Compilers for Parallel Computing*, pages 136-151, New Orleans, Louisiana, 2006. Verlag. [ [bib](#) | [pdf](#) ]



# ML for Schedule Automation



## The Next 700 ML-Enabled Compiler Optimizations

S. VenkataKeerthy  
IIT Hyderabad, India

Siddharth Jain  
IIT Hyderabad, India

Umesh Kalvakuntla  
IIT Hyderabad, India

Pranav Sai Gorantla  
IIT Hyderabad, India

Rajiv Shailesh Chitale  
IIT Hyderabad, India

Eugene Brevdo  
Google DeepMind, USA

Albert Cohen  
Google DeepMind, France

Mircea Trofin  
Google, USA

Ramakrishna Upadrastra  
IIT Hyderabad, India

### Abstract

There is a growing interest in enhancing compiler optimizations with ML models, yet interactions between compilers and ML frameworks remain challenging. Some optimizations require tightly coupled models and compiler internals, raising issues with modularity, performance and framework independence. Practical deployment and transparency for the end-user are also important concerns. We propose ML-COMPILER-BRIDGE to enable ML model development within a traditional Python framework while making end-to-end integration with an optimizing compiler possible and efficient. We evaluate it on both research and production use cases, for training and inference, over several optimization problems, multiple compilers and its versions, and gym infrastructures.

ML and Reinforcement Learning (RL) approaches have been proposed to improve optimizations like vectorization [21, 36], loop unrolling, distribution [25, 43], function inlining [27, 47], register allocation [17, 26, 46, 50], prediction of phase sequences [5, 23, 24], among many others [2, 53]. More specifically, the widely used LLVM compiler [29] has support for RL-based inlining decisions from version 11, and RL-based eviction decisions in its register allocator from version 14 [46]. The title of our paper acknowledges this growing trend and anticipates the needs of the ML-enabled optimizations that are yet to come, in the spirit of Landis' seminal paper [28] on the diversity of existing and future programming languages. Setting up an ML-based compiler optimization is a challenging task. In addition to model design, it involves specialized data collection, compiler engineering, packaging,



## RL4REAL: Reinforcement Learning for Register Allocation

S. VenkataKeerthy  
IIT Hyderabad  
India

Siddharth Jain  
IIT Hyderabad  
India

Anilava Kundu  
IIT Hyderabad  
India

Rohit Aggarwal  
IIT Hyderabad  
India

Albert Cohen  
Google  
France

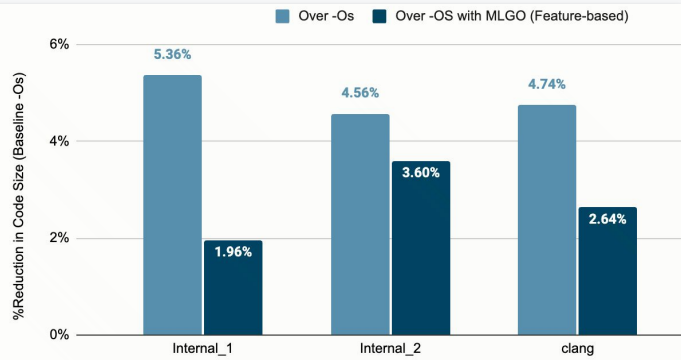
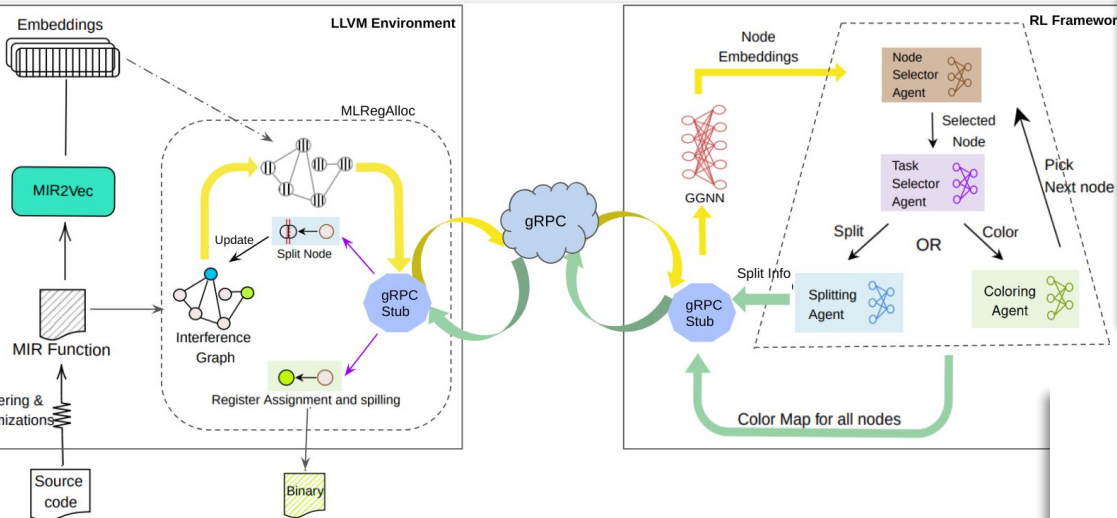
Ramakrishna Upadrastra  
IIT Hyderabad  
India

### Abstract

We aim to automate decades of research and experience in register allocation, leveraging machine learning. We tackle this problem by embedding a multi-agent reinforcement learning algorithm within LLVM, training it with the state of the art techniques. We formalize the constraints that precisely define the problem for a given instruction-set architecture, while ensuring that the generated code preserves semantic correctness. We also develop a gRPC based framework providing a modular and efficient compiler interface for training and inference. Our approach is architecture in-

problem is reducible to graph coloring, which is one of the classical NP-Complete problems [8, 22]. Register allocation as an optimization involves additional sub-tasks, more than graph coloring itself [8]. Several formulations have been proposed that return exact, or heuristic-based solutions.

Broadly, solutions are often formulated as constraint-based optimizations [34, 38], ILP [3, 5, 12, 42], PBQP [31], game-theoretic approaches [45], and are fed to a variety of solvers. In general, these approaches are known to have scalability issues. On the other hand, heuristic-based approaches have been widely used owing to their scalability: reasonable solu-



Inlining for size experiment





**MLIR**

**Infrastructure for Compiler Construction**



# Controllable Compiler Optimizations

- **Algorithm/Model level**

Python schedules, Lücke et al.

- Expose codegen building blocks to performance engineers
- Reuse schedules across models/layers and targets

- **IR-level**

MLIR [transform dialect](#) to construct

“custom codegen flows”, [tutorial](#), [recording](#)

## The MLIR Transform Dialect

Your compiler is more powerful than you think

[CGO 2025]

Martin Lücke, U. Edinburgh

Oleksander Zinenko, Google DeepMind

Albert Cohen, Google DeepMind

William Moses, Google DeepMind and UIUC

Michel Steuwer, TU Berlin

### Abstract

To take full advantage of a specific hardware target, performance engineers need to gain control on compilers in order to leverage their domain knowledge about the program and hardware. Yet, modern compilers are poorly controlled, usually by configuring a sequence of coarse-grained monolithic black-box passes, or by means of predefined compiler annotations/pragmas. These can be effective, but often do not let users precisely optimize their varying compute loads. As a consequence, performance engineers have to resort to implementing custom passes for a specific optimization heuristic, requiring compiler engineering expert knowledge.

In this paper, we present a technique that provides fine-grained control of general-purpose compilers by the *Transform dialect*, a controllable IR-based transformation system implemented in MLIR. The Transform dialect empowers performance engineers to optimize their varying compute loads by composing and reusing existing—often hidden—compiler features without the need to implement new passes or even rebuilding the compiler.

We demonstrate in five case studies that the dialect enables precise, safe composition of compiler transformations and allows for straightforward integration with state-of-the-art search methods.

and to perform specific optimizations parameterized by their corresponding flags—e.g. apply *loop invariant code motion* on all loops. However, this coarse level of control is increasingly insufficient to optimize programs for today’s heterogeneous hardware that require precise optimization decisions. Pragmas, or compiler annotations in the source code, provide finer grained control—e.g. vectorization or unrolling hints. These are effective but their implementation requires in-depth and non-modular changes to the compiler, hence their restriction to specific cases anticipated by compiler engineers.

Often specific parts of a program dominate the overall runtime and are worth optimizing precisely or offloading to





# Example: Python (JAX) Schedules

```
def schedule (module: OpHandle) -> None:
    matmul    = module.match_ops (linalg.BatchMatmulOp)
    fill      = module.match_ops (linalg.FillOp)
    for_all   = matmul.tile_to_forall (tile_sizes=[64, 64, 1])
    fill.fuse_into (for_all)
    for_all2 = matmul.tile_to_forall (tile_sizes=[4, 32, 1])
    # ...
```

.py

Generates transform IR

```
transform.sequence (%module: !transform.op<module>) {
    %matmul = transform.match_op name "linalg.batch_matmul" in %module
    // [...]
    %forall, %tiled = transform.tile_to_forall_op %matmul tile_sizes [64, 64, 1]
    // [...]
    %fused, %containing = transform.fuse_into_containing_op %forall
    // [...]
    %forall0, %tiled0 = transform.tile_to_forall_op %tiled tile_sizes [4, 32, 1]
    // [...]
```

Inject

```
func.func public @batch_matmul(%arg0: tensor<128x80x32xf32>,
                                %arg1: tensor<128x32x320xf32>)->
    (tensor<128x80x320xf32>) {

    // prepare output
    %0 = tensor.empty() : tensor<128x80x320xf32>
    %cst = arith.constant 0.0 : f32
    %1 = linalg.fill ins(%cst) outs(%0)
    %2 = linalg.batch_matmul ins(%arg0, %arg1) outs(%1)
    return %2 : tensor<128x80x320xf32>

}
```

.mlir

--apply\_transform\_script

```
func.func public @batch_matmul(%arg0: tensor<128x80x32xf32>,
                                %arg1: tensor<128x32x320xf32>)->
    (tensor<128x80x320xf32>) {
    %0 = tensor.empty() : tensor<128x80x320xf32>
    %cst = arith.constant 0.0 : f32
    scf.forall (64, 64, 1) {
        %1 = linalg.fill
        scf.forall (4, 32, 1) {
            %2 = linalg.batch_matmul
            // [...]
        }
    }
}
```

.mlir



# The Schedule is the Compiler

## 1. Schedule completely drives the compiler

```
def schedule(module: OpHandle) -> None:
    # [...]
    # lower to llvm is actually:
    module.convert_linalg_to_loops_pass()
    module.convert_scf_to_cf_pass()
    module.lower_affine_pass()
    module.convert_vector_to_llvm_pass()
    module.convert_math_to_llvm_pass()
    module.finalize_memref_to_llvm_conversion_pass()
    module.func_to_llvm_pass()
    module.reconcile_unrealized_casts_pass()
```



Every pass can be initiated through this interface

```
module.run_pass("MyPassName")
```

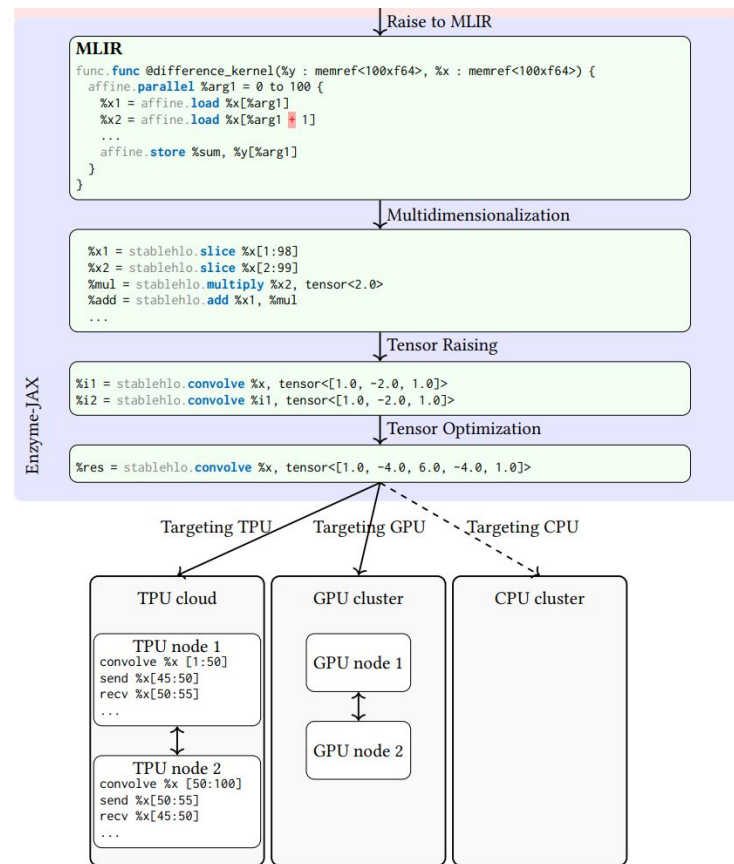
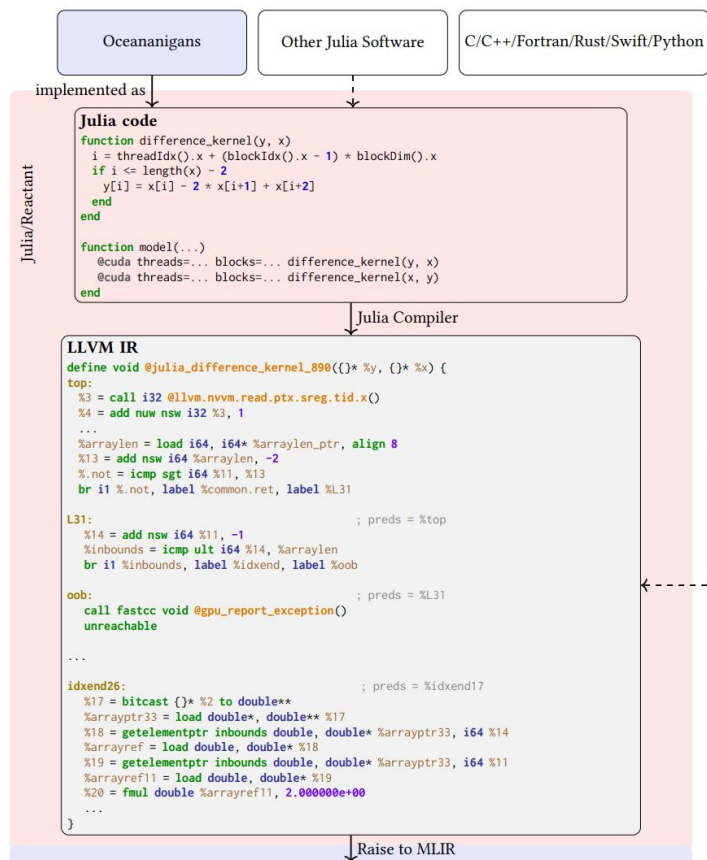
## 2. Constructing new Passes on-the-fly

```
with handle.apply_patterns():
    structured.ApplyTilingCanonicalizationPatternsOf()
    loop. ApplyForLoopCanonicalizationPatternsOf()
    transform ApplyCanonicalizationPatternsOf()
```

- Not possible with any ML compiler until now
- Combination of patterns does not have to be known statically

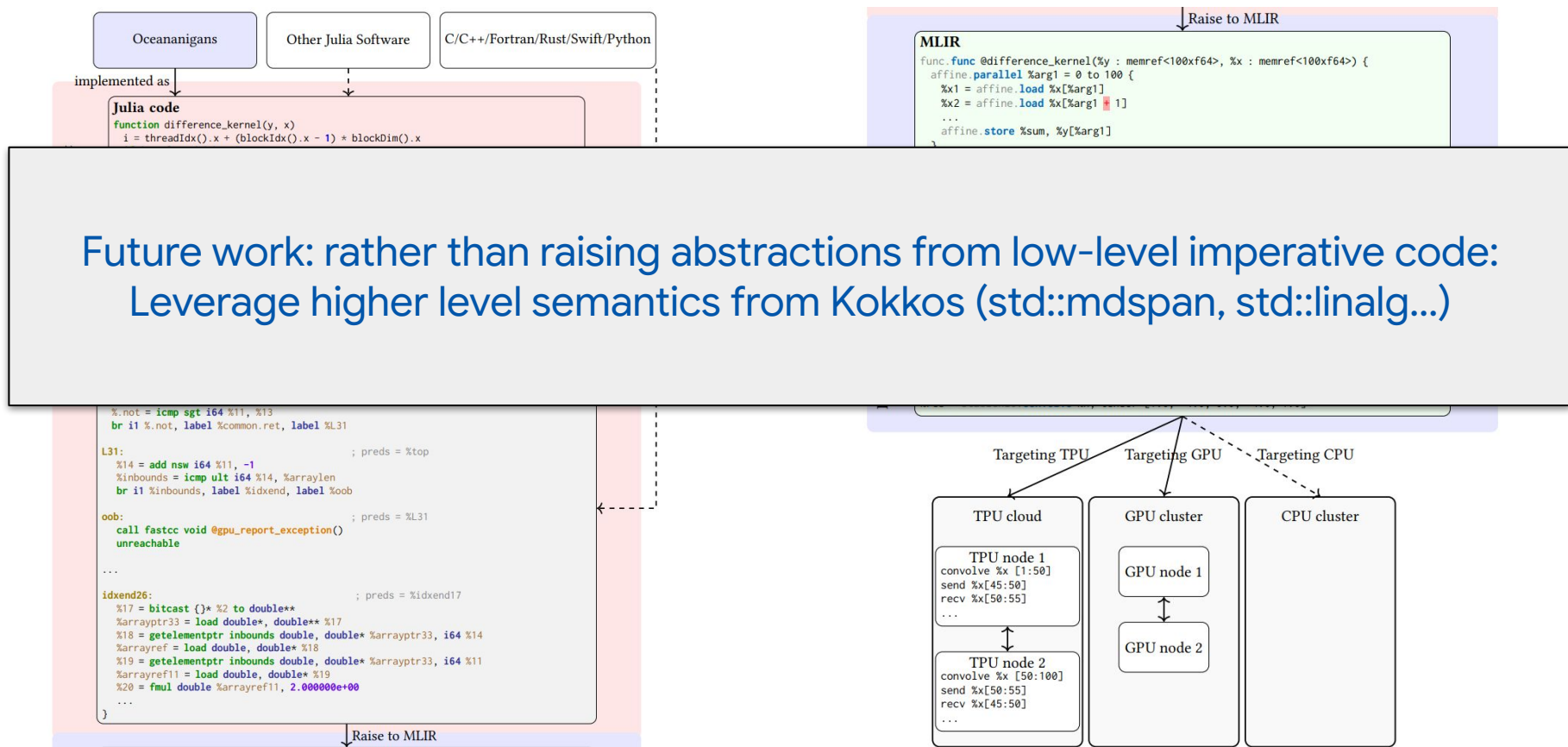


# Science → LLVM → MLIR → Heterogeneous Platform





# Science → LLVM → MLIR → Heterogeneous Platform





# Distribution and Mapping

## Sharded Matrix Multiplication

```
mesh = Sharding.Mesh(  
    reshape(Reactant.devices(), :, 4), (:x, :y)  
)  
sharding = Sharding.NamedSharding(mesh, (:x, :y))  
  
x = Reactant.to_rarray(rand(Float32, 8, 4); sharding)  
y = Reactant.to_rarray(rand(Float32, 4, 8); sharding)  
  
@jit x * y
```

Lower to MLIR

## MLIR Pre-Sharding Propagation

```
module @"reactant_*" attributes {mhlo.num_partitions = 8 : i64, mhlo.num_replicas = 1 :  
  i64} {  
  ↪ i64 {  
    sdy.mesh @mesh = <["x"=2, "y"=4]>  
    func.func @main(%arg0: tensor<4x8xf32> {sdy.sharding = #sdy.sharding@mesh, [{"y"}],  
      ↪ {"x"}]>}, %arg1: tensor<8x4xf32> {sdy.sharding = #sdy.sharding@mesh, [{"y"}],  
      ↪ {"x"}]>}) -> tensor<8x8xf32> {  
      %0 = stablehlo.dot_general %arg1, %arg0, contracting_dims = [1] x [0], precision =  
      ↪ [DEFAULT, DEFAULT] : (tensor<8x4xf32>, tensor<4x8xf32>) -> tensor<8x8xf32>  
      return %0 : tensor<8x8xf32>  
    }  
  }
```

Propagate Sharding

## MLIR Post-Sharding Propagation

```
module @"reactant_*" attributes {mhlo.num_partitions = 8 : i64, mhlo.num_replicas = 1 :  
  i64} {  
  ↪ i64 {  
    func.func @main(%arg0: tensor<4x8xf32> {mhlo.sharding =  
      ↪ "{devices=[4,2]<=[2,4]T(1,0)}"}}, %arg1: tensor<8x4xf32> {mhlo.sharding =  
      ↪ "{devices=[4,2]<=[2,4]T(1,0)}"}}) -> (tensor<8x8xf32> {mhlo.sharding =  
      ↪ "{devices=[4,2]<=[2,4]T(1,0)}"}}) {  
      %0 = stablehlo.dot_general %arg1, %arg0, contracting_dims = [1] x [0], precision =  
      ↪ [DEFAULT, DEFAULT] {mhlo.sharding = "{devices=[4,2]<=[2,4]T(1,0)}"} :  
      ↪ (tensor<8x4xf32>, tensor<4x8xf32>) -> tensor<8x8xf32>  
      return %0 : tensor<8x8xf32>  
    }  
  }
```

# Compute Graphs Are More Expressive Than You Think

## Listing 1 Reactant code for compiling Julia functions

using Reactant

```
a = Reactant.to_rarray(ones(10))  
b = Reactant.to_rarray(ones(10))
```

```
sinsum_add(x, y) = sum(sin.(x) .+ y)  
f = @compile sinsum_add(a, b)
```

```
# one can now run the program  
f(a, b)
```

## Listing 2 Compiled MLIR from Julia code

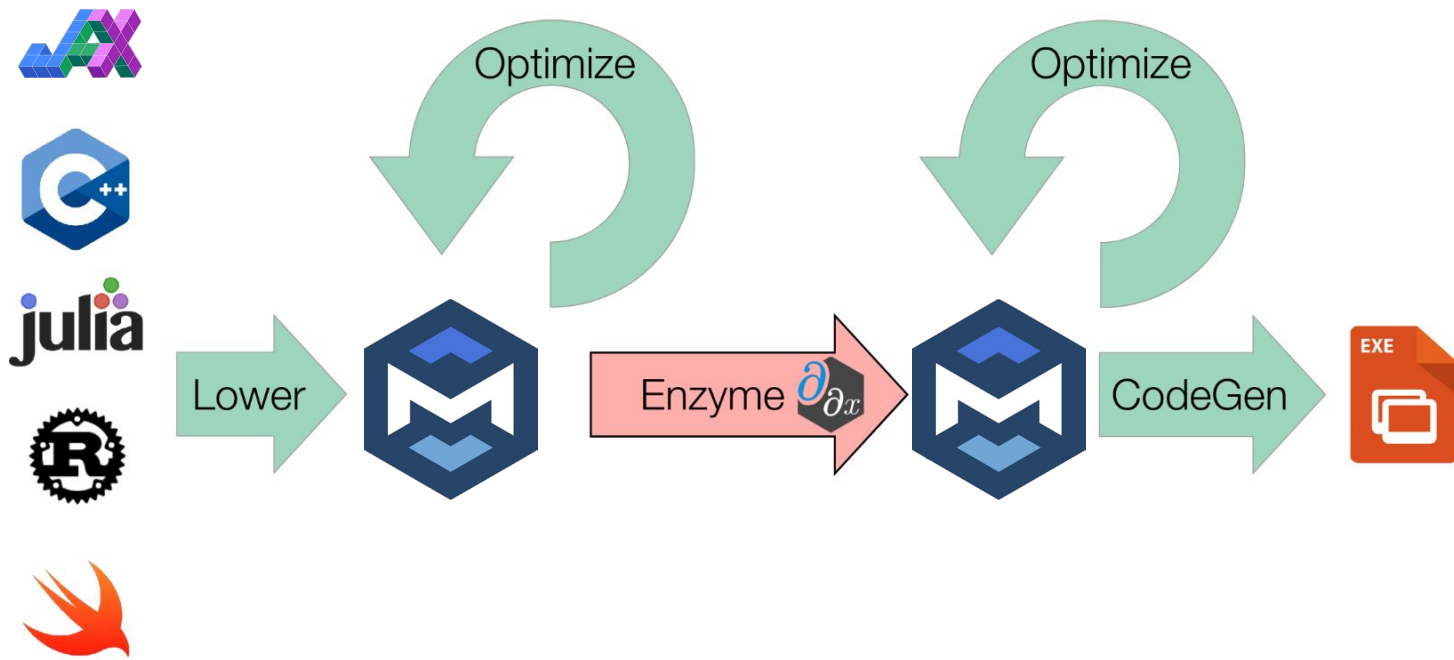
```
module @reactant_sinsum_add attributes {mhlo.num_partitions  
  ↪ = 1 : i64, mhlo.num_replicas = 1 : i64} {  
  func.func @main(%arg0: tensor<10xf64>, %arg1:  
    ↪ tensor<10xf64>) -> tensor<f64> {  
    %cst = stablehlo.constant dense<0.0> : tensor<f64>  
    %0 = stablehlo.sine %arg0 : tensor<10xf64>  
    %1 = stablehlo.add %0, %arg1 : tensor<10xf64>  
    %2 = stablehlo.reduce(%1 init: %cst) applies  
    ↪ stablehlo.add across dimensions = [0] :  
    ↪ (tensor<10xf64>, tensor<f64>) -> tensor<f64>  
    return %2 : tensor<f64>  
  }
```





# Enzyme Framework: AutoDiff for LLVM/MLIR

Billy Moses (UIUC / Google)







# Enzyme-JAX

Also for C++, CUDA, Julia, Fortran, Rust

```
from enzyme_ad.jax import cpp_call

# Forward-mode C++ AD example

@jax.jit
def something(inp):
    y = cpp_call(inp, out_shapes=[jax.core.ShapedArray([2, 3], jnp.float32)], source="""
        template<std::size_t N, std::size_t M>
        void myfn(enzyme::tensor<float, N, M>& out0, const enzyme::tensor<float, N, M>& in0) {
            out0 = 56.0f + in0(0, 0);
        }
        """, fn="myfn")
    return y

ones = jnp.ones((2, 3), jnp.float32)
primals, tangents = jax.jvp(something, (ones,), (ones,))

# Reverse-mode C++ AD example

primals, f_vjp = jax.vjp(something, ones)
(grads,) = f_vjp((x,))
```



# Join the Enzyme-JAX adventure!

```
from enzyme_ad.jax import cpp_call

# Forward-mode C++ AD example

@jax.jit
def something(inp):
    y = cpp_call(inp, out_shapes=[jax.core.ShapedArray([2, 3], jnp.float32)], source="""
        template<std::size_t N, std::size_t M>
        void myfn(enzyme::tensor<float, N, M>& out0, const enzyme::tensor<float, N, M>& in0) {
            out0 = 56.0f + in0(0, 0);
        }
        """, fn="myfn")
    return y

ones = jnp.ones((2, 3), jnp.float32)
primals, tangents = jax.jvp(something, (ones,), (ones,))

# Reverse-mode C++ AD example

primals, f_vjp = jax.vjp(something, ones)
(grads,) = f_vjp((x,))
```

<https://enzyme.mit.edu>

<https://github.com/EnzymeAD/Enzyme-JAX>

<https://polygeist.llvm.org>