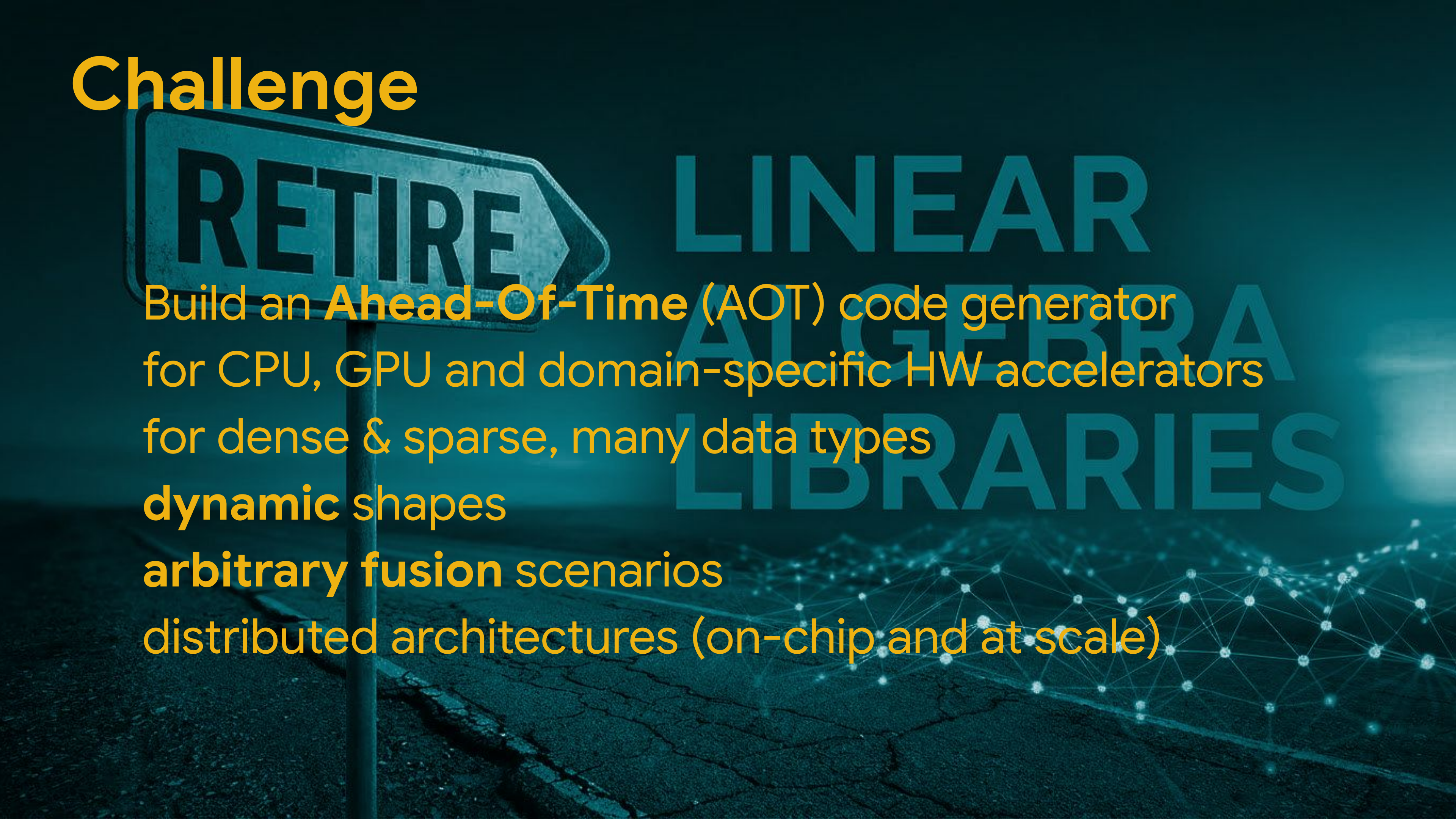
LINEAR ALGEBRA LIBRARIES

… 4 years later

COMPILERS
PERFORMANCE
LIBRARIES, KERNEL
PROGRAMMING

Reality is much more complex, and actually better for compilers!

Challenge



Build an **Ahead-Of-Time** (AOT) code generator
for CPU, GPU and domain-specific HW accelerators
for dense & sparse, many data types
dynamic shapes
arbitrary fusion scenarios
distributed architectures (on-chip and at scale)

MLIR

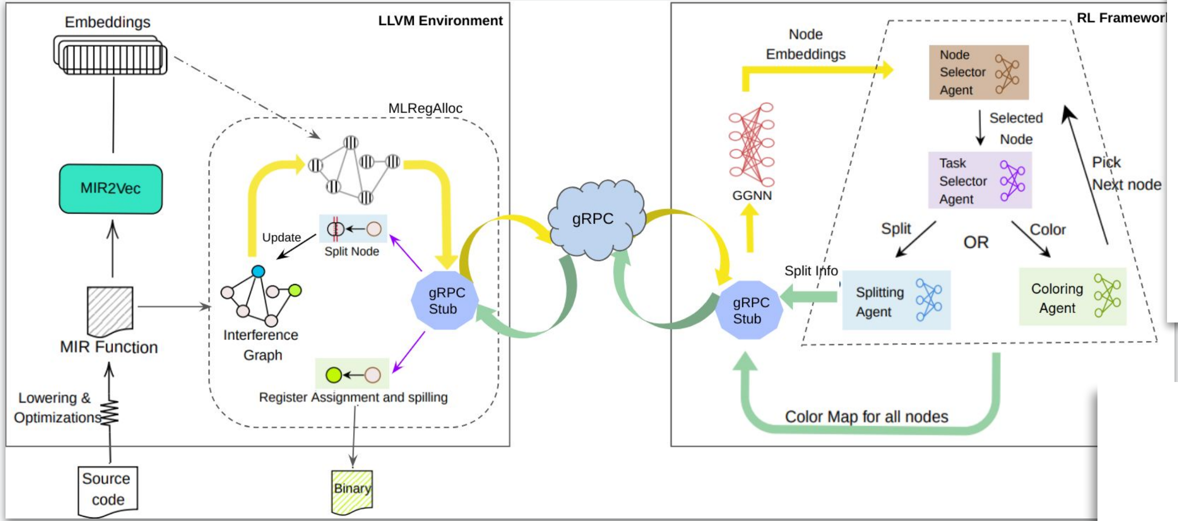
Infrastructure for Compiler Construction

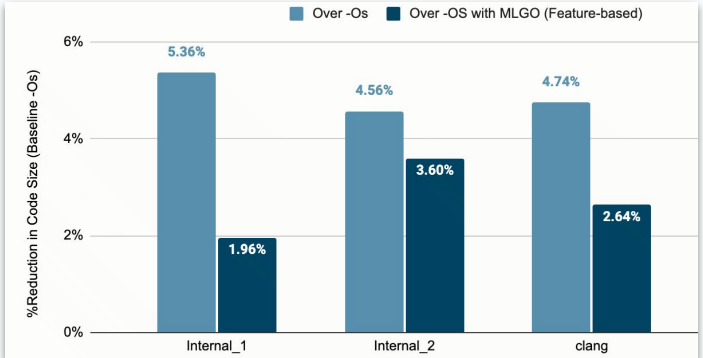
ML for Systems is Everywhere

Databases: The Case for Learned Index Structures
Compilers: MLGO: A Machine Learning Guided Compiler Optimizations Framework
Hardware Design: A graph placement methodology for fast chip design
Accelerator Design: Apollo: Transferable Architecture Exploration
Clustem Management: Autopilot: workload autoscaling at Google
Configuration Tuning: Google Vizier: A Service for Black-Box Optimization
Cache Management: An Imitation Learning Approach for Cache Replacement
Storage Systems: A Bring-Your-Own-Model Approach for ML-Driven Storage Placement in Warehouse-Scale Computers

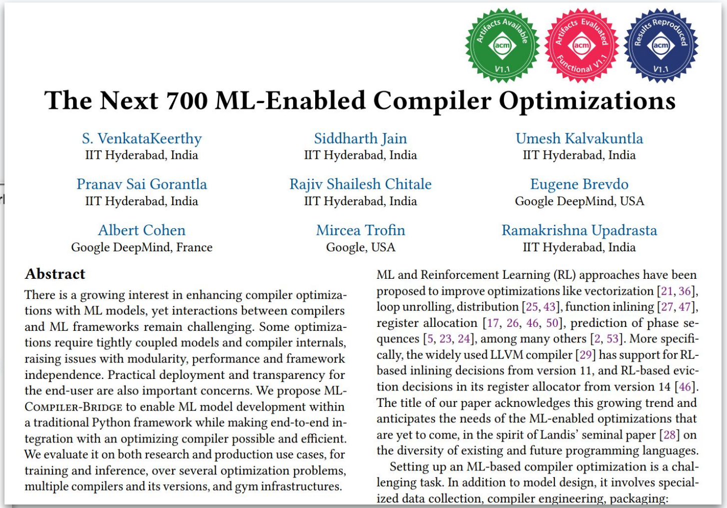
listing selected examples of production systems at Google only...

ML for Compilers





Inlining for size experiment



The Next 700 ML-Enabled Compiler Optimizations

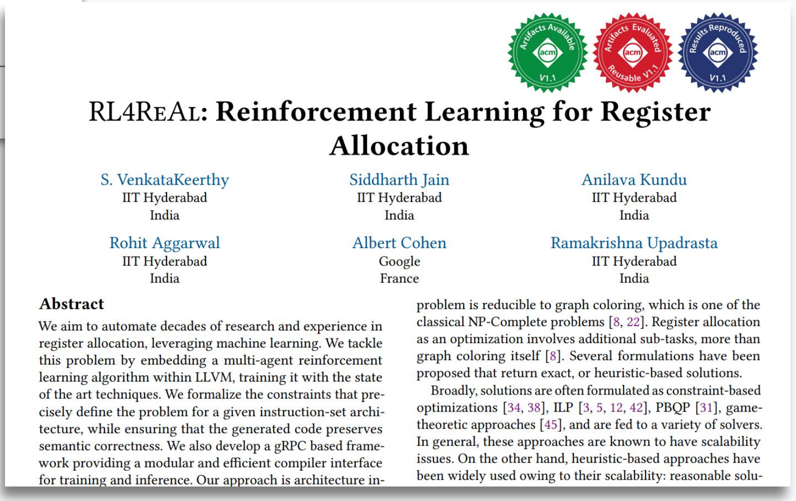
S. VenkataKeerthy, IIT Hyderabad, India
Siddharth Jain, IIT Hyderabad, India
Umesh Kalvakuntla, IIT Hyderabad, India
Pranav Sai Gorantla, IIT Hyderabad, India
Rajiv Shailesh Chitale, IIT Hyderabad, India
Eugene Brevdo, Google DeepMind, USA
Albert Cohen, Google DeepMind, France
Mircea Trofin, Google, USA
Ramakrishna Upadrasta, 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 Upadrasta, IIT Hyderabad, India

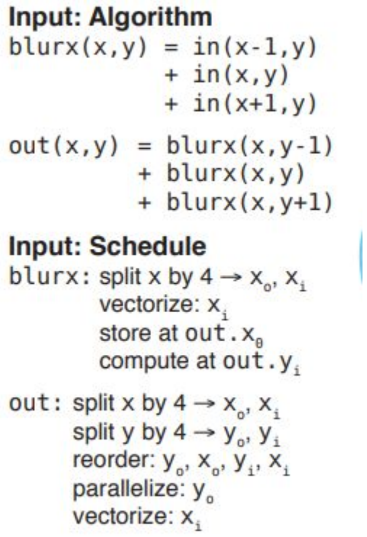
Abstract

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-

Performance Engineering & Compilers Unite! User-Schedulable Languages



Input: Algorithm

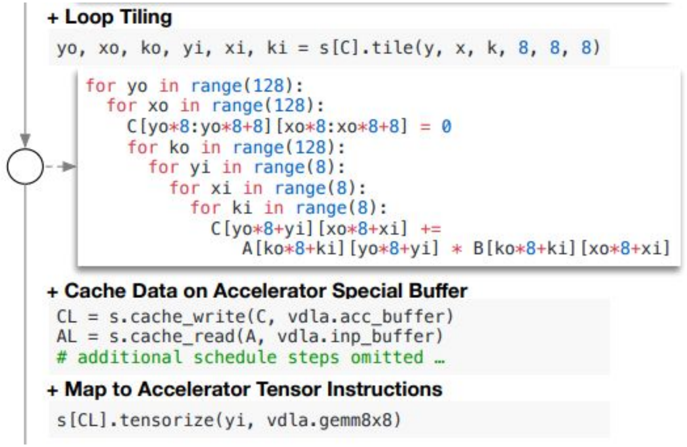
```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)

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

Input: Schedule

blurx: split x by 4 → x_o, x_i
    vectorize: x_i
    store at out.x_θ
    compute at out.y_i

out: split x by 4 → x_o, x_i
    split y by 4 → y_o, y_i
    reorder: y_o, x_o, y_i, x_i
    parallelize: y_o
    vectorize: x_i

Halide (Ragan-Kelley et.al. 2013)



+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
               A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

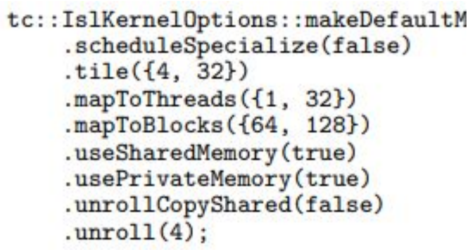
+ Cache Data on Accelerator Special Buffer

```
CL = s.cache_write(C, vdla.acc_buffer)
AL = s.cache_read(A, vdla.inp_buffer)
# additional schedule steps omitted …
```

+ Map to Accelerator Tensor Instructions

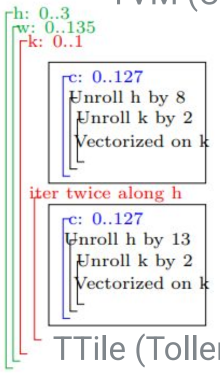
```
s[CL].tensorize(yi, vdla.gemm8x8)
```

TVM (Chen et.al. 2018)



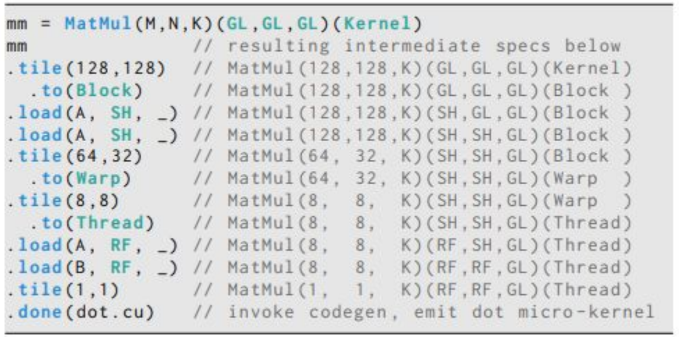
```
tc::IslKernelOptions::makeDefaultM
     .scheduleSpecialize(false)
     .tile({4, 32})
     .mapToThreads({1, 32})
     .mapToBlocks({64, 128})
     .useSharedMemory(true)
     .usePrivateMemory(true)
     .unrollCopyShared(false)
     .unroll(4);
```

TC (Vasilache et.al. 2018)



```
h: 0..3
 w: 0..135
  k: 0..1
     c: 0..127
      Unroll h by 8
       Unroll k by 2
       Vectorized on k
   iter twice along h
     c: 0..127
      Unroll h by 13
       Unroll k by 2
       Vectorized on k
```

TTile (Tollenaere et.al. 2021)

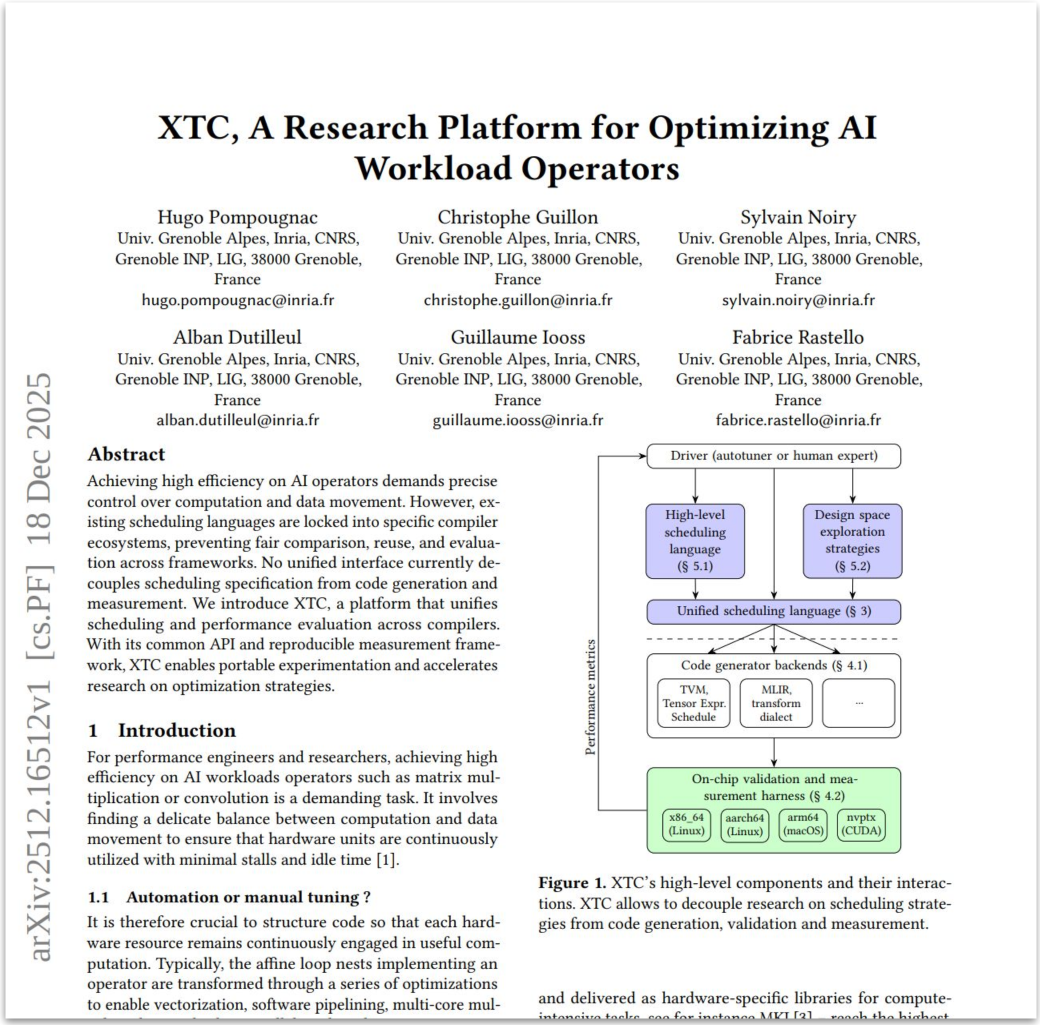


```
mm = MatMul(M,N,K)(GL,GL,GL)(Kernel)
mm                 // resulting intermediate specs below
.tile(128,128)     // MatMul(128,128,K)(GL,GL,GL)(Kernel)
  .to(Block)       // MatMul(128,128,K)(GL,GL,GL)(Block )
.load(A, SH, _)    // MatMul(128,128,K)(SH,GL,GL)(Block )
.load(A, SH, _)    // MatMul(128,128,K)(SH,SH,GL)(Block )
.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)

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

Codegen + Schedules + Retargetable + Performance modeling/feedback



XTC, A Research Platform for Optimizing AI Workload Operators

Hugo Pompougnac
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
hugo.pompougnac@inria.fr

Christophe Guillon
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
christophe.guillon@inria.fr

Sylvain Noiry
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
sylvain.noiry@inria.fr

Alban Dutilleul
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
alban.dutilleul@inria.fr

Guillaume Iooss
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
guillaume.iooss@inria.fr

Fabrice Rastello
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
fabrice.rastello@inria.fr

Abstract

Achieving high efficiency on AI operators demands precise control over computation and data movement. However, existing scheduling languages are locked into specific compiler ecosystems, preventing fair comparison, reuse, and evaluation across frameworks. No unified interface currently decouples scheduling specification from code generation and measurement. We introduce XTC, a platform that unifies scheduling and performance evaluation across compilers. With its common API and reproducible measurement framework, XTC enables portable experimentation and accelerates research on optimization strategies.

1 Introduction

For performance engineers and researchers, achieving high efficiency on AI workloads operators such as matrix multiplication or convolution is a demanding task. It involves finding a delicate balance between computation and data movement to ensure that hardware units are continuously utilized with minimal stalls and idle time [1].

1.1 Automation or manual tuning ?

It is therefore crucial to structure code so that each hardware resource remains continuously engaged in useful computation. Typically, the affine loop nests implementing an operator are transformed through a series of optimizations to enable vectorization, software pipelining, multi-core mul-

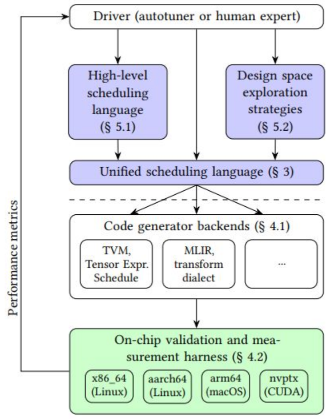

Driver (autotuner or human expert)
High-level scheduling language (§ 5.1)
Design space exploration strategies (§ 5.2)
Unified scheduling language (§ 3)
Code generator backends (§ 4.1)
TVM, Tensor Expr. Schedule
MLIR, transform dialect
...
On-chip validation and measurement harness (§ 4.2)
x86_64 (Linux)
aarch64 (Linux)
arm64 (macOS)
nvptx (CUDA)
Performance metrics

Figure 1. XTC's high-level components and their interactions. XTC allows to decouple research on scheduling strategies from code generation, validation and measurement.

and delivered as hardware-specific libraries for compute-

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

[CGO 2025]

The MLIR Transform Dialect

Your compiler is more powerful than you think

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 p
grained control of general-purpose compilers by
the *Transform dialect*, a controllable IR-based tra
system implemented in MLIR. The Transform di
ers performance engineers to optimize their v
pute loads by composing and reusing existing—
hidden—compiler features without the need t
new passes or even rebuilding the compiler.

We demonstrate in five case studies that th
dialect enables precise, safe composition of co
formations and allows for straightforward inte
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

Tutorial: Controllable Transformations in MLIR

Example: Python (JAX) Schedules

.py

```
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])
  # ...
```

Generates transform IR

.mlir

```
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

.mlir

```
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>
}
```

--apply_transform_script

.mlir

```
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
        // [...]
}
```

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")
```

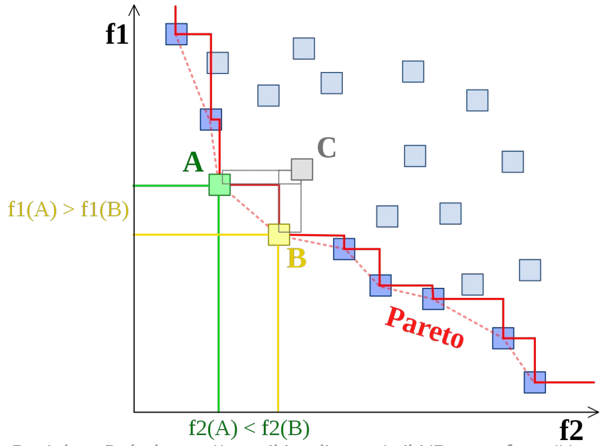
2. Constructing new Passes on-the-fly

```
with handle.apply_patterns():
  structured.ApplyTilingCanonicalizationPatternsOp()
  loop.      ApplyForLoopCanonicalizationPatternsOp()
  transform. ApplyCanonicalizationPatternsOp()
```

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

Proposal: **TLO = Tile-Level Operations**

2D Pareto surface/frontier:
performance vs.
code size vs.
model specialization



By Johan Dréo https://en.wikipedia.org/wiki/Pareto_front#/media/File:Front_pareto.svg

Run-time: bytecode interpretation with generic control flow and dynamic dispatch of TLOs

AOT: synthesis and code generation for 10x-10000x of TLO implementations

TLO
=
dynamic/static interface

Proposal: **TLO = Tile-Level Operations**

- **TLO specification language**
  - **Static and Dynamic** input/output shapes
  - Strides, layouts, data types, etc.
  - Constraints on the above (e.g., ranges of admissible sizes and strides)

Proposal: **TLO = Tile-Level Operations**

- **Bytecode language = framework/platform/domain-specific**
  - **Instantiate a specific bytecode language, using the TLO specification language**
  - Generic control flow & memory management + **instance-specific ops**
  - A TLO graph is dynamically interpreted by default

  - **Focus on making AOT compilation possible**

TLO Challenges?

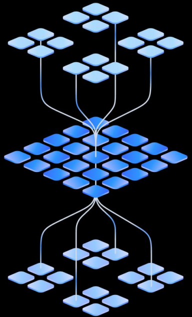
- **Performance**
  - Temporal reuse across TLOs through memory (caches, scratchpad) only: may lose the register-level reuse benefits of finer-grained fusion
  - Bytecode interpretation
  - Code size tradeoffs
- **Dynamic dispatch**
  - Super-fast, from TLO signatures to implementations
  - What about dynamic shapes? fusion?
  - Memoization for loopy bytecode

TLO Challenges?

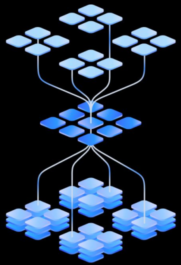
- **Code generation**
  - Risks largely mitigated by existing MLIR-based codegen efforts
    → *natural fit for structured ops, and the transform dialect*
  - Classical autotuning immediately applicable but not required
  - Manual implementation of key ops remains possible (reuse libraries…)
- **Automatic instantiation of target/domain-specific bytecode language**
  - Next generation autotuning required
    - Classification and synthesis for a minimum performance criterion
    - Pareto surface: *performance* vs. *code size* vs. *domain specialization*
  - Deployment into existing frameworks and execution environments

What About Performance Portability?

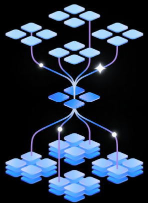
Ultra



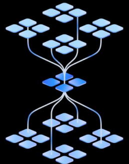
Pro

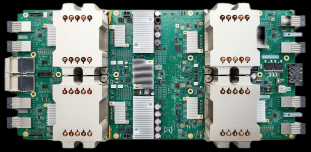


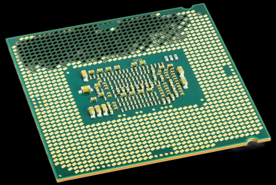
Flash

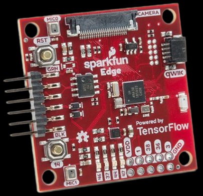


Nano







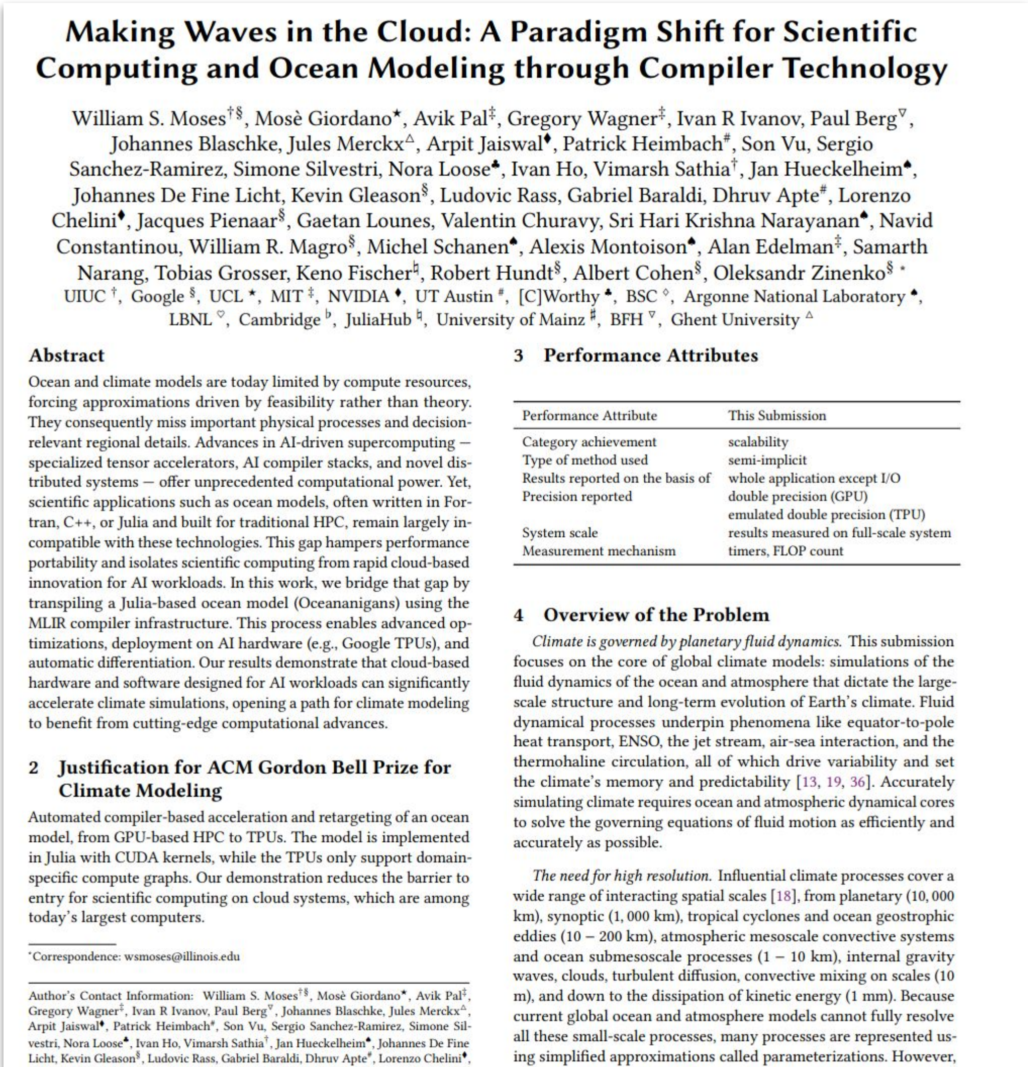


Example: Waves in the Cloud

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

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.

[*]Correspondence: wsmoses@illinois.edu

Author's Contact Information: William S. Moses[†§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, Paul Berg[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], Son Vu, Sergio Sanchez-Ramirez, Simone Silvestri, Nora Loose[♣], Ivan Ho, Vimarsh Sathia[†], Jan Hueckelheim[♠], Johannes De Fine Licht, Kevin Gleason[§], Ludovic Rass, Gabriel Baraldi, Dhruv Apte[#], Lorenzo Chelini[♦],

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,

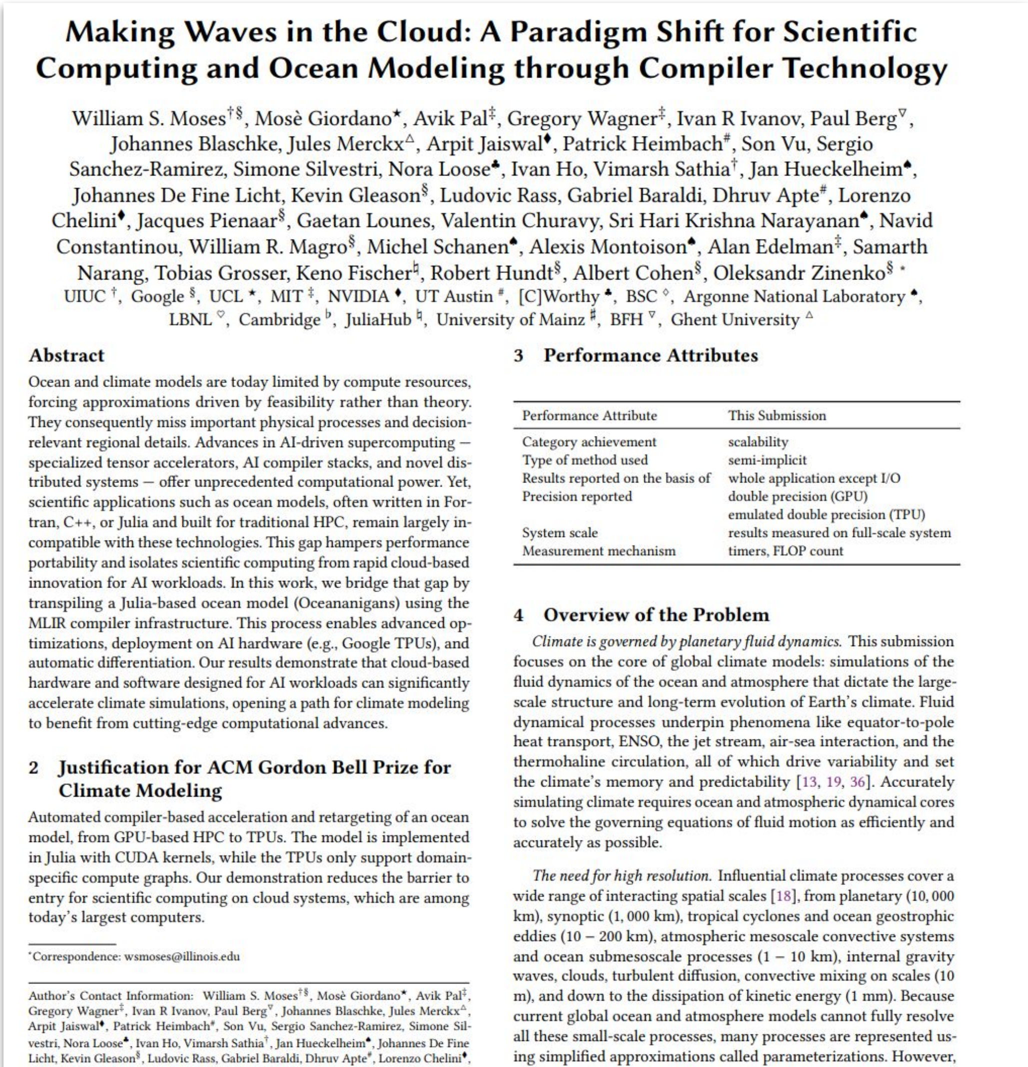
Application: **Oceananigans.jl**

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

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

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.

[*]Correspondence: wsmoses@illinois.edu

Author's Contact Information: William S. Moses[†§], Mosè Giordano[⋆], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, Paul Berg[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], Son Vu, Sergio Sanchez-Ramirez, Simone Silvestri, Nora Loose[♣], Ivan Ho, Vimarsh Sathia[†], Jan Hueckelheim[♠], Johannes De Fine Licht, Kevin Gleason[§], Ludovic Rass, Gabriel Baraldi, Dhruv Apte[#], Lorenzo Chelini[♦],

3 Performance Attributes

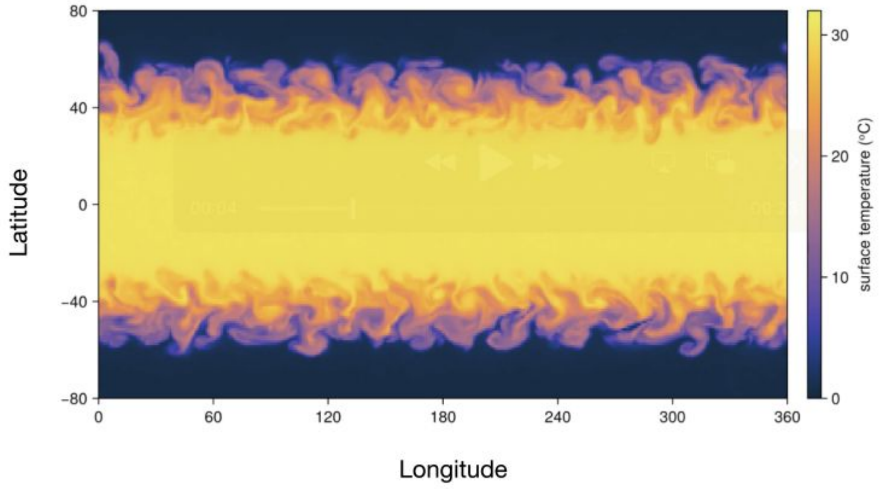
Performance Attribute	This Submission
Category achievement	scalability
Type of method used	semi-implicit
Results reported on the basis of	whole application except I/O
Precision reported	double precision (GPU)
	emulated double precision (TPU)
System scale	results measured on full-scale system
Measurement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

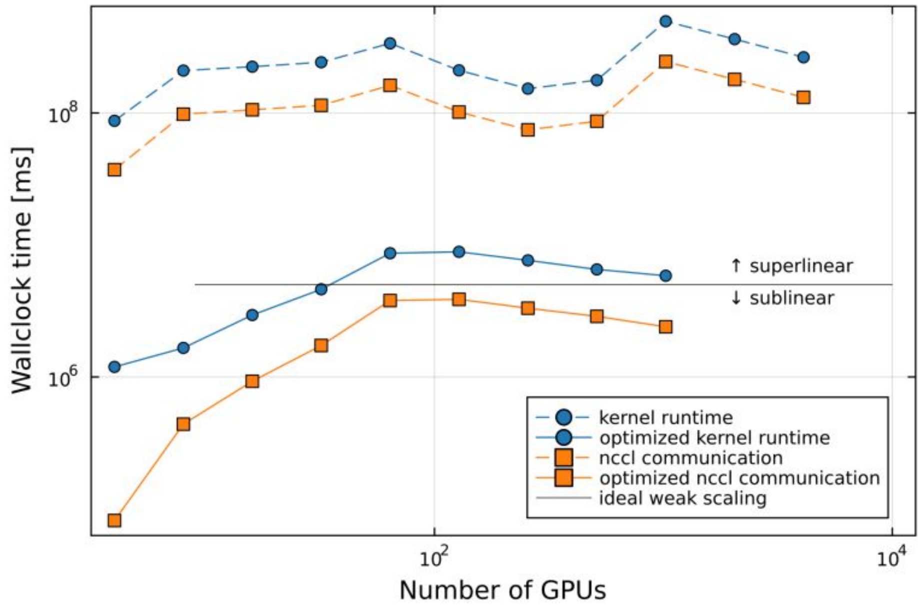
4 Overview of the Problem

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





What about TPUs? And Why?

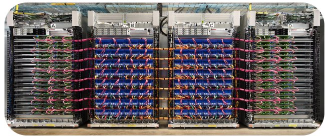
	TPU v4	TPU v5p	Ironwood
	2022	2023	2025
Pod Size (chips)	4096	8960	9216
HBM Bandwidth/ Capacity	32 GB @ 1.2 TBps HBM	95 GB @ 2.8 TBps HBM	192 GB @ 7.4 TBps HBM
Peak Flops per chip	275 TFLOPS	459 TFLOPS	4614 TFLOPS

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)

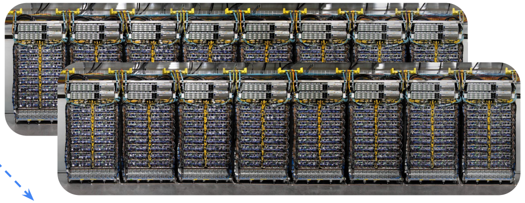
What about TPUs? Energy Efficiency and Scale



TPU v2

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

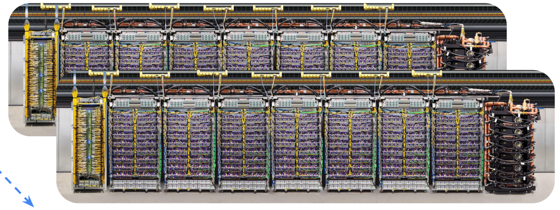
8x



TPU v4

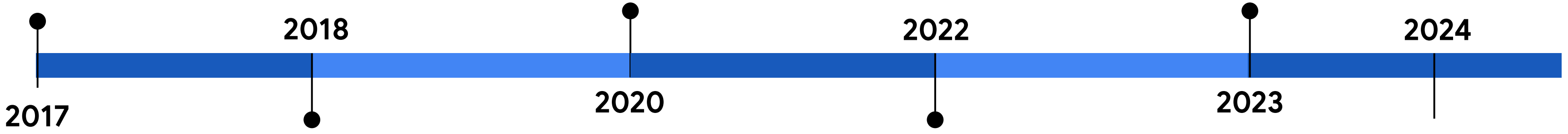
- Optically reconfigurable 3D Torus
- 4k chips with distributed shared memory

20x



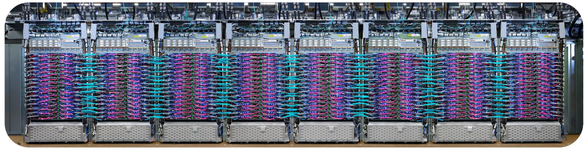
TPU v5p

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



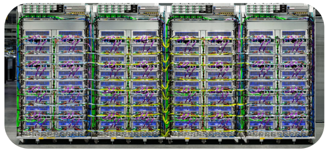
TPU v3

- Liquid cooling
- 1k chips distributed shared memory



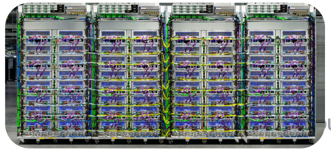
TPU v5e

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

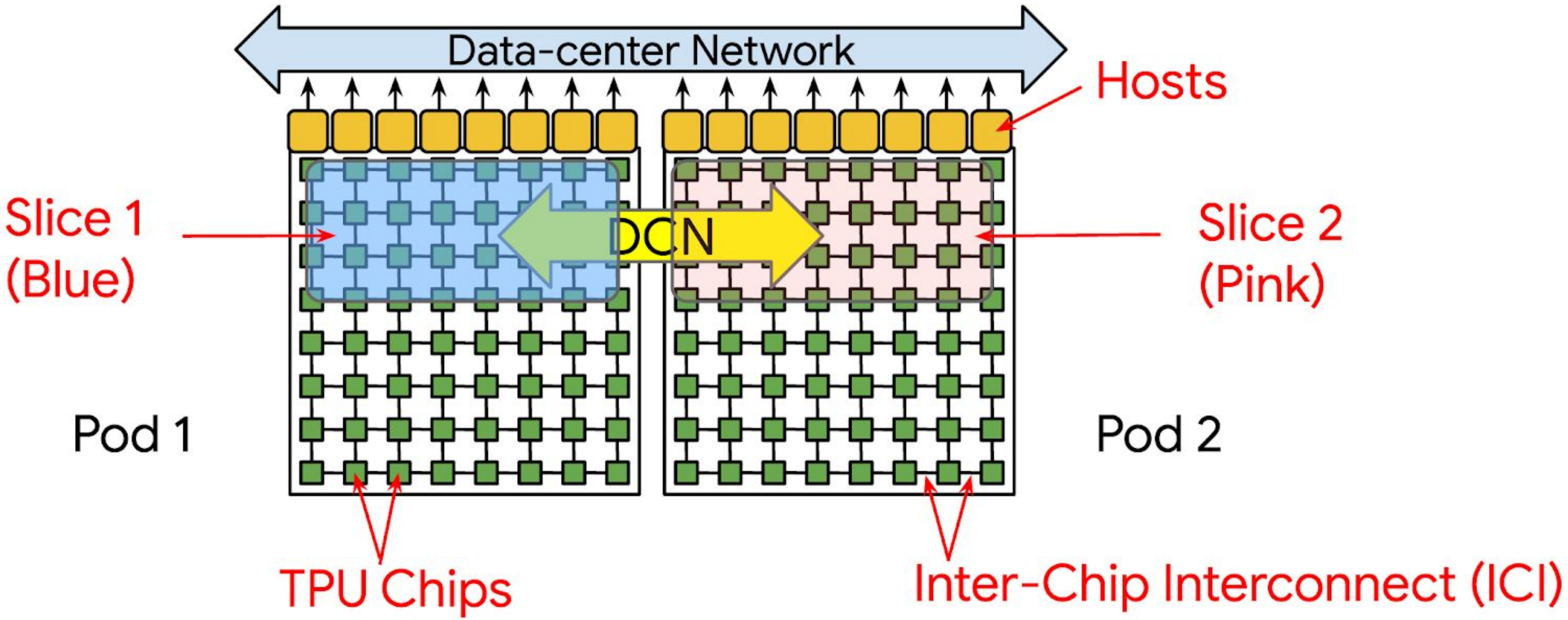


TPU v6e

- 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



https://cloud.google.com/blog/products/compute/using-cloud-tpu-multislice-to-scale-ai-workloads

Kernel programming to the rescue

Flurry of GPU acceleration options

CUDA Kernels / OpenCL-C
SYCL
Kokkos
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

$$\Downarrow$$

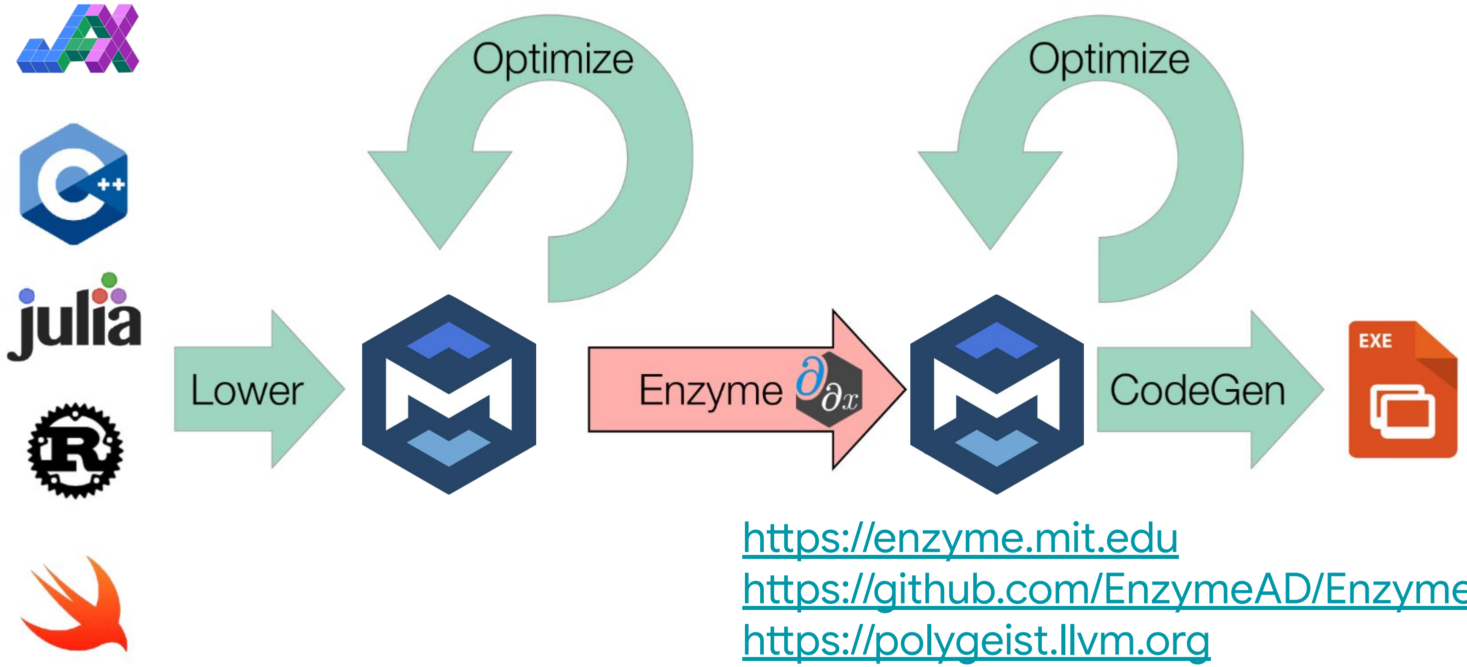
abstraction penalty

Motivations: escape hatch for...

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

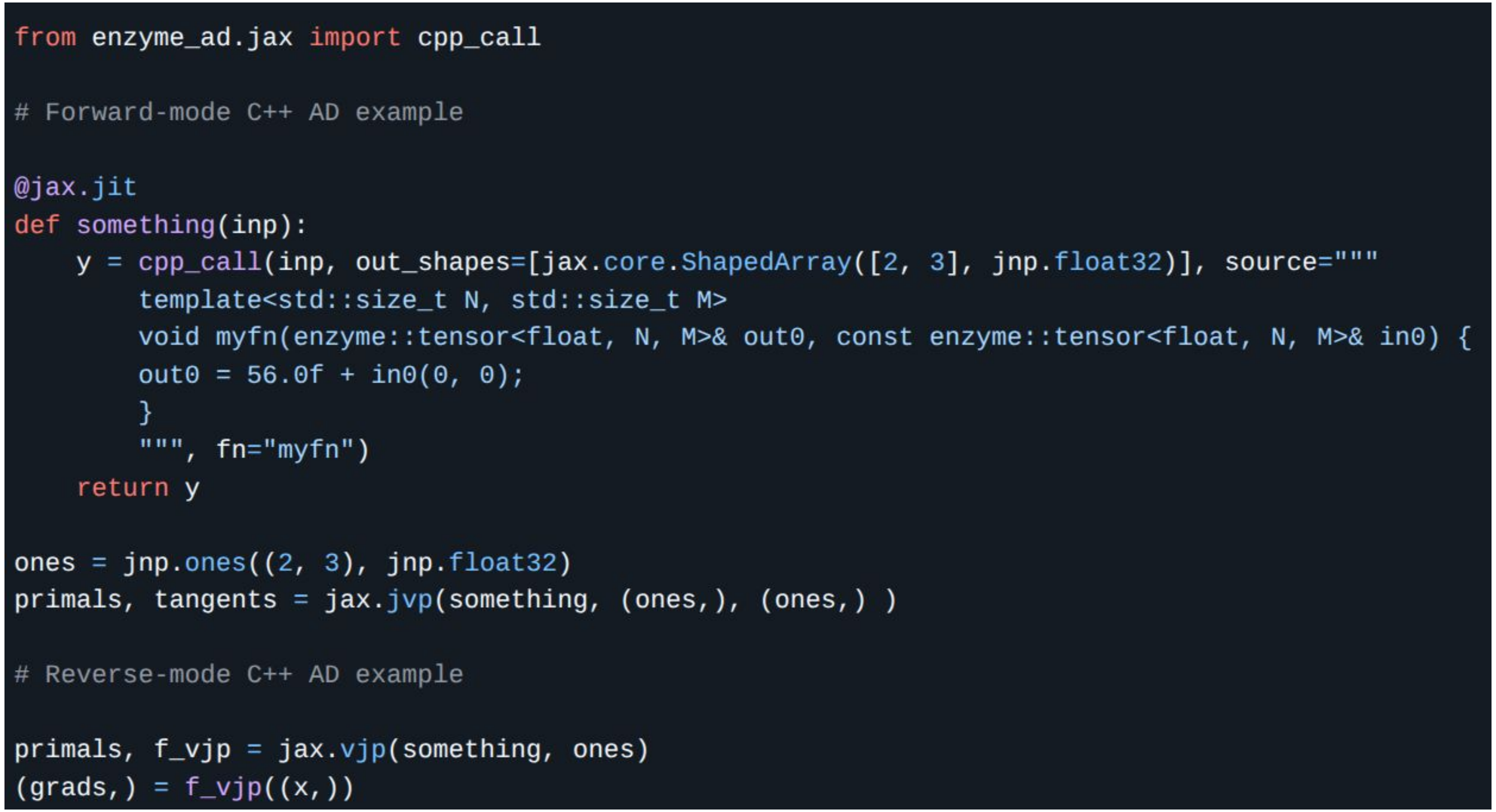
We can do better than kernel programming...

Enzyme Framework: AutoDiff for LLVM/MLIR
Billy Moses (UIUC / Google)



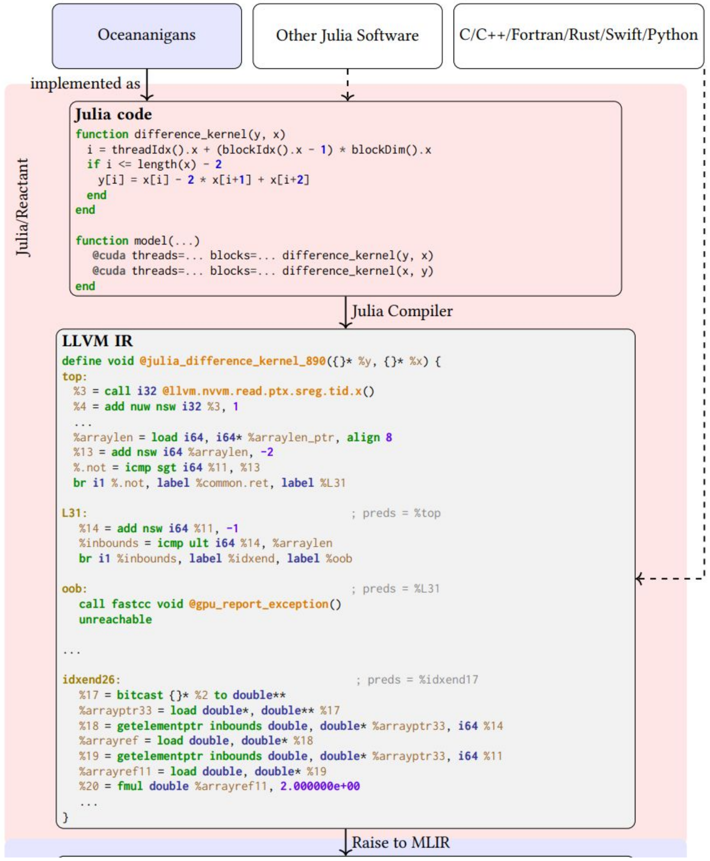
https://enzyme.mit.edu
https://github.com/EnzymeAD/Enzyme-JAX
https://polygeist.llvm.org

Enzyme-JAX
Also for C++, CUDA, Julia, Fortran, Rust



```python
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,))
```

Science → LLVM → MLIR → Heterogeneous Platform

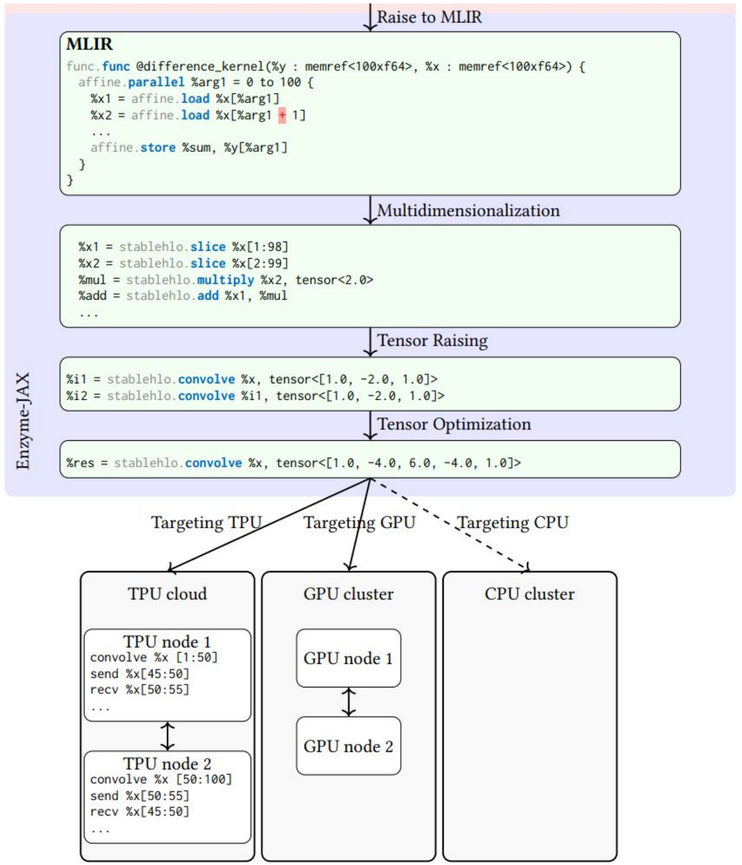


Julia code

```
function difference_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=... difference_kernel(y, x)
   @cuda threads=... blocks=... difference_kernel(x, y)
end
```

LLVM IR

```
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
  ...
  %arraylen = load i64, i64* %arraylen_ptr, align 8
  %13 = add nsw i64 %arraylen, -2
  %.not = icmp sgt i64 %11, %13
  br i1 %.not, label %common.ret, label %L31

L31:                                              ; preds = %top
   %14 = add nsw i64 %11, -1
   %inbounds = icmp ult i64 %14, %arraylen
   br i1 %inbounds, label %idxend, label %oob

oob:                                              ; preds = %L31
   call fastcc void @gpu_report_exception()
   unreachable

...

idxend26:                                          ; preds = %idxend17
   %17 = bitcast {}* %2 to double**
   %arrayptr33 = load double*, double** %17
   %18 = getelementptr inbounds double, double* %arrayptr33, i64 %14
   %arrayref = load double, double* %18
   %19 = getelementptr inbounds double, double* %arrayptr33, i64 %11
   %arrayref11 = load double, double* %19
   %20 = fmul double %arrayref11, 2.000000e+00
   ...
}
```



MLIR

```
func.func @difference_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]
  }
}
```

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

```
%i1 = stablehlo.convolve %x, tensor<[1.0, -2.0, 1.0]>
%i2 = stablehlo.convolve %i1, tensor<[1.0, -2.0, 1.0]>
```

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

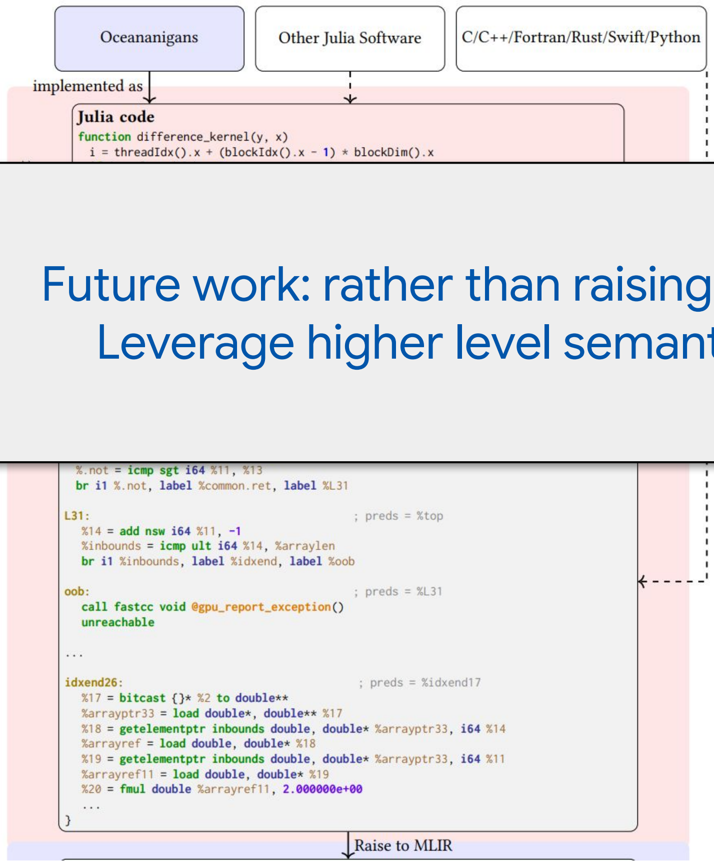
TPU node 1

```
convolve %x [1:50]
send %x[45:50]
recv %x[50:55]
...
```

TPU node 2

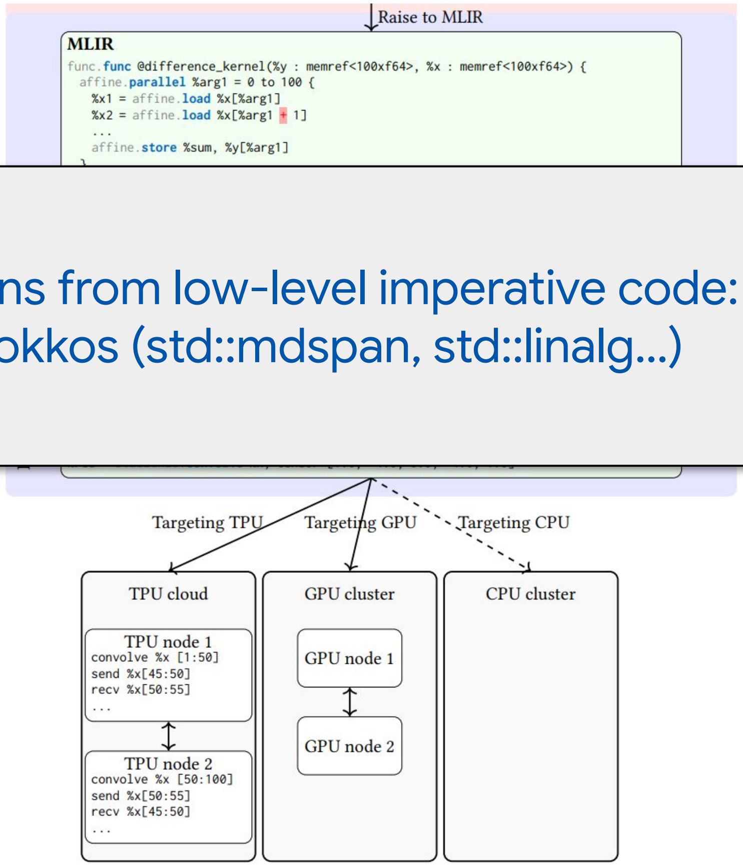
```
convolve %x [50:100]
send %x[50:55]
recv %x[45:50]
...
```

Science → LLVM → MLIR → Heterogeneous Platform



```
function difference_kernel(y, x)
  i = threadIdx().x + (blockIdx().x - 1) * blockDim().x
```

```
  %.not = icmp sgt i64 %11, %13
  br i1 %.not, label %common.ret, label %L31

L31:                                          ; preds = %top
   %14 = add nsw i64 %11, -1
   %inbounds = icmp ult i64 %14, %arraylen
   br i1 %inbounds, label %idxend, label %oob

oob:                                          ; preds = %L31
   call fastcc void @gpu_report_exception()
   unreachable

...

idxend26:                                     ; preds = %idxend17
   %17 = bitcast {}* %2 to double**
   %arrayptr33 = load double*, double** %17
   %18 = getelementptr inbounds double, double* %arrayptr33, i64 %14
   %arrayref = load double, double* %18
   %19 = getelementptr inbounds double, double* %arrayptr33, i64 %11
   %arrayref11 = load double, double* %19
   %20 = fmul double %arrayref11, 2.000000e+00
   ...
}
```



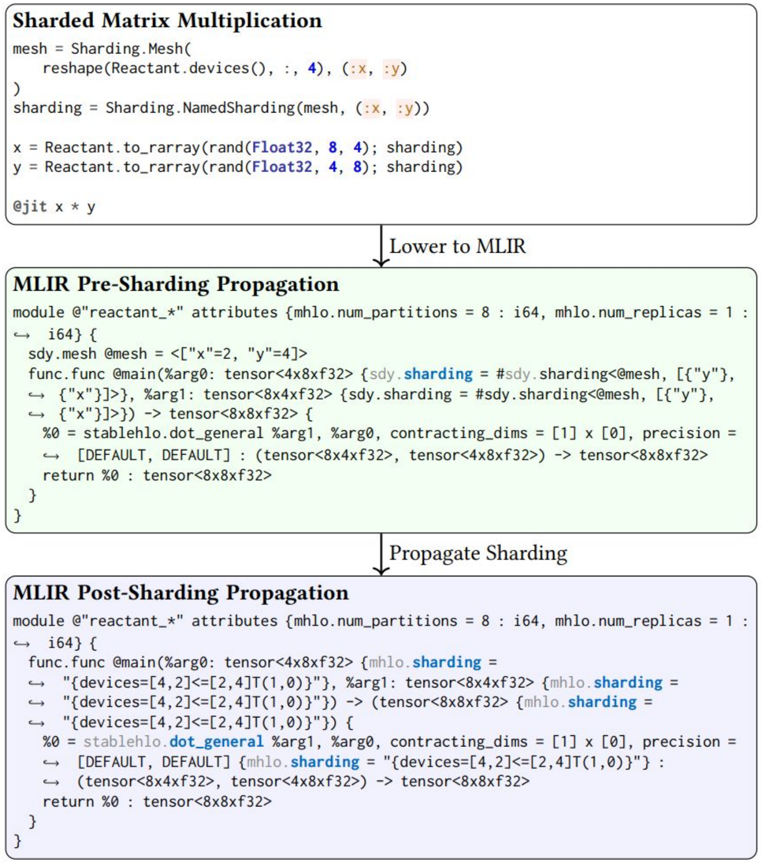
```
func.func @difference_kernel(%y : memref<100xf64>, %x : memref<100xf64>) {
  affine.parallel %arg1 = 0 to 100 {
    %x1 = affine.load %x[%arg1]
    %x2 = affine.load %x[%arg1 + 1]
    ...
    affine.store %sum, %y[%arg1]
```

```
convolve %x [1:50]
send %x[45:50]
recv %x[50:55]
...
```

```
convolve %x [50:100]
send %x[50:55]
recv %x[45:50]
...
```

Future work: rather than raising abstractions from low-level imperative code: Leverage higher level semantics from Kokkos (std::mdspan, std::linalg...)

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} {
  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} {
  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

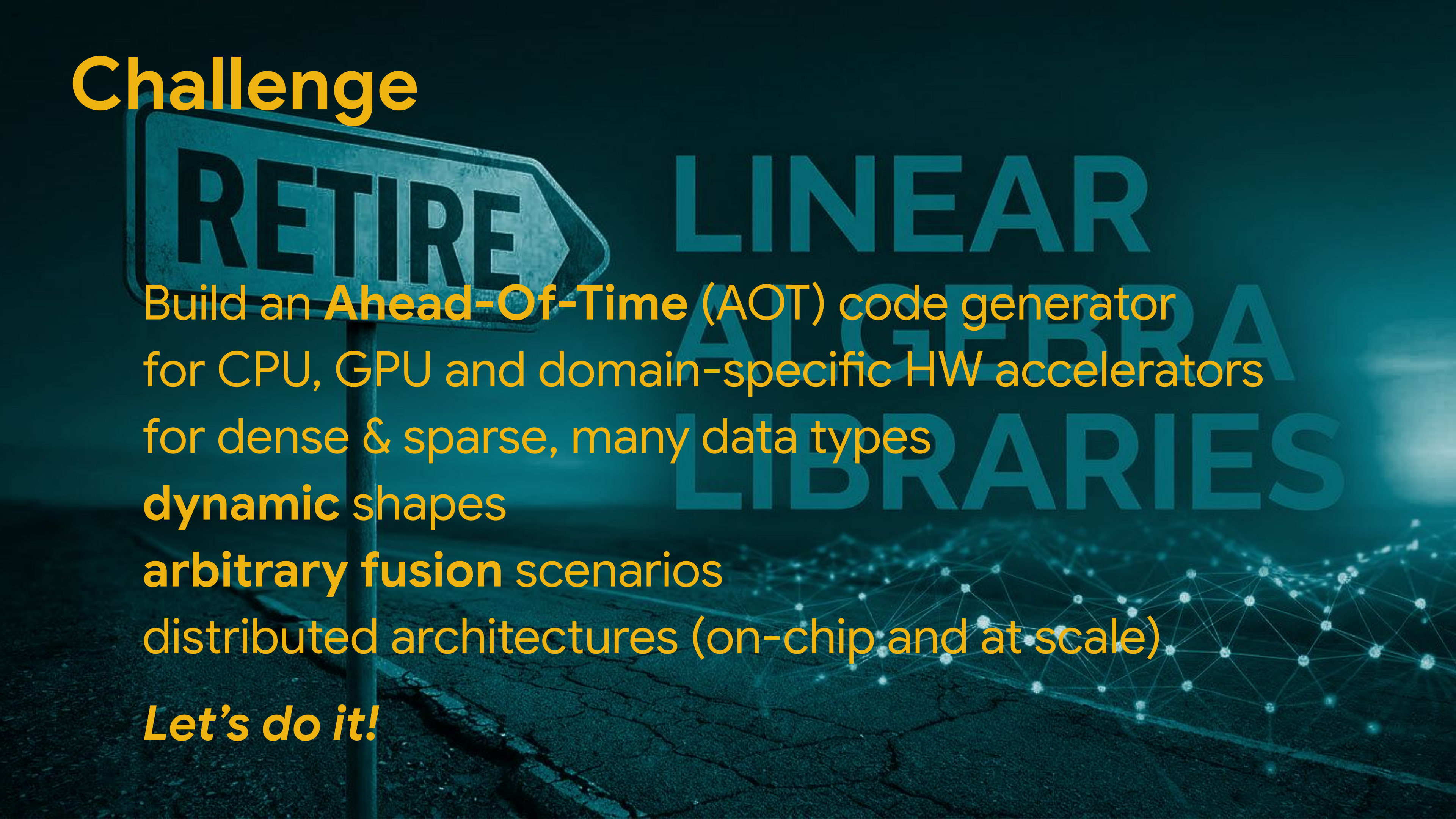
Listing 1 Reactant code for compiling Julia functions

```
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>
  }
}
```

Challenge



Build an **Ahead-Of-Time** (AOT) code generator
for CPU, GPU and domain-specific HW accelerators
for dense & sparse, many data types
dynamic shapes
arbitrary fusion scenarios
distributed architectures (on-chip and at scale)

Let's do it!