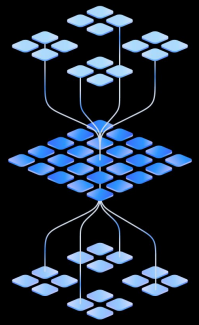
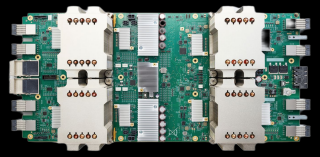
Google DeepMind

Condenser: *noun*, an Apparatus for Compiling Science to the Cloud

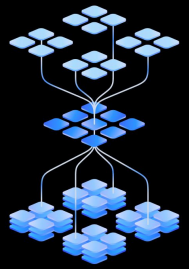
Automatic Parallelization, Performance Portability: Solved Problems for AI Applications?

Ultra

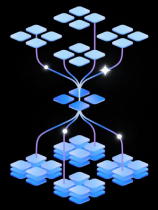


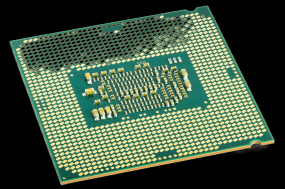


Pro

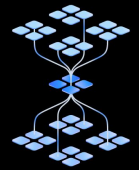


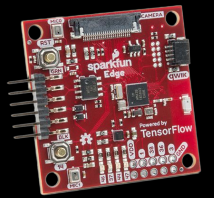
Flash





Nano

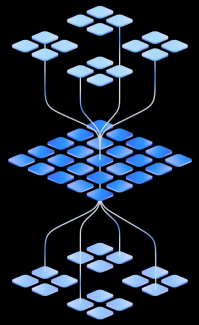


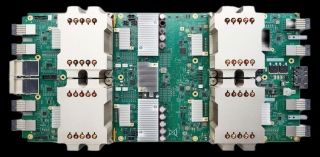


What about HPC Performance Portability?

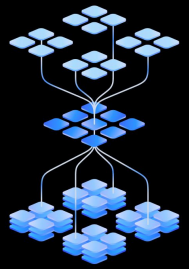
Automatic Parallelization, Performance Portability: Solved Problems for AI Applications?

Ultra

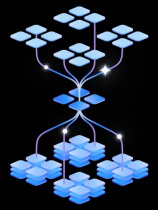


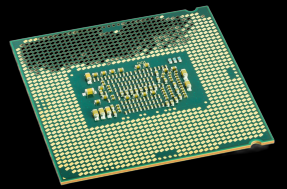


Pro

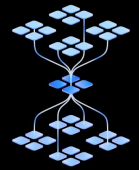


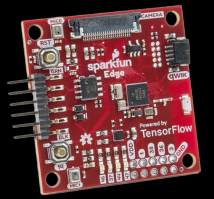
Flash





Nano





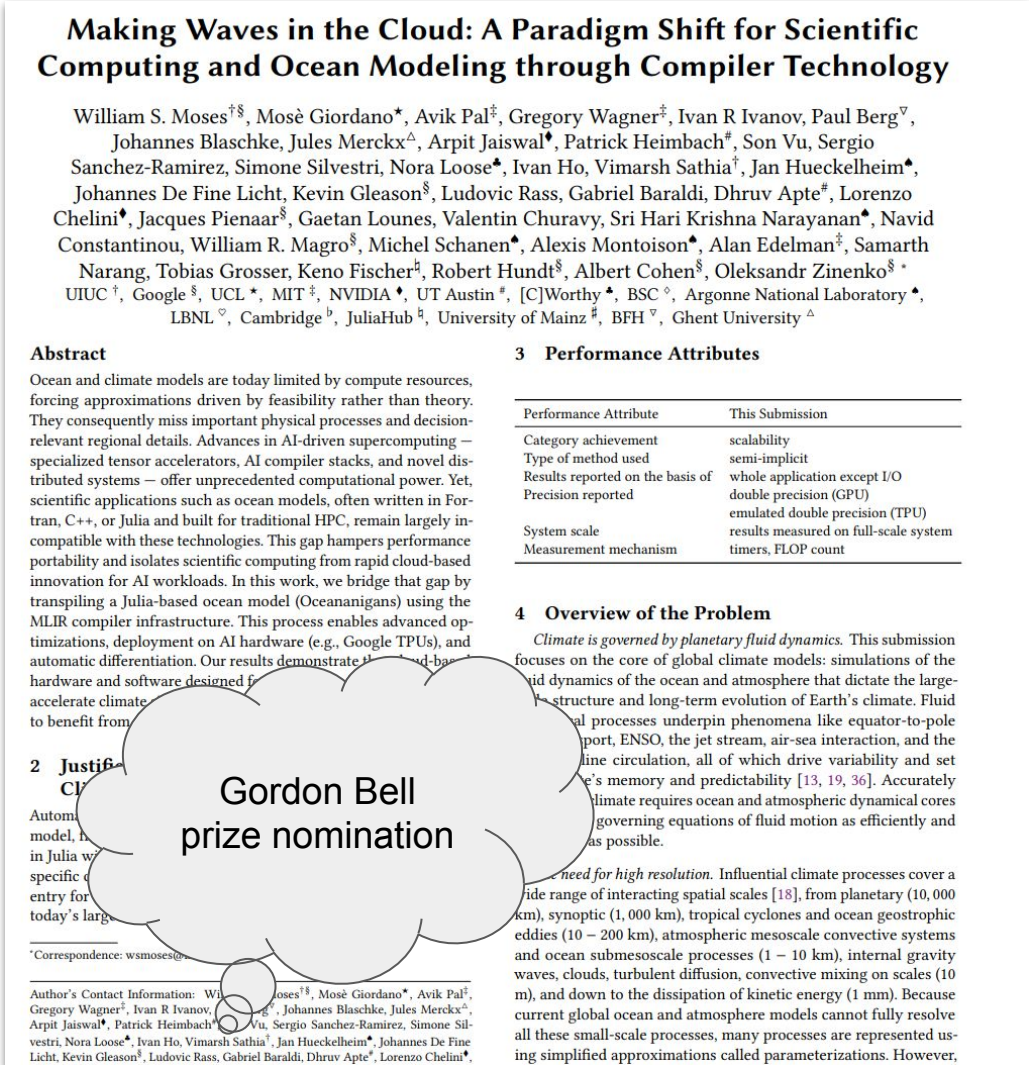
Time to revisit the role of the compiler?

Before the “condenser” we started making waves in the cloud

Platforms

- NERSC Perlmutter
  1536 GPU accelerated nodes with
  1 AMD Milan processor 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 th… hardware and software designed f… accelerate climate… to benefit from…

Gordon Bell prize nomination

2 Justifi… Cl…

Autom… model, h… in Julia w… specific … entry for … today’s larg…

*Correspondence: wsmoses@…

Author’s Contact Information: Wi… …oses[†§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, … …g[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], … …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 … id dynamics of the ocean and atmosphere that dictate the large- … structure and long-term evolution of Earth’s climate. Fluid … al processes underpin phenomena like equator-to-pole … port, ENSO, the jet stream, air-sea interaction, and the … line circulation, all of which drive variability and set … e’s memory and predictability [13, 19, 36]. Accurately … limate requires ocean and atmospheric dynamical cores … governing equations of fluid motion as efficiently and … as possible.

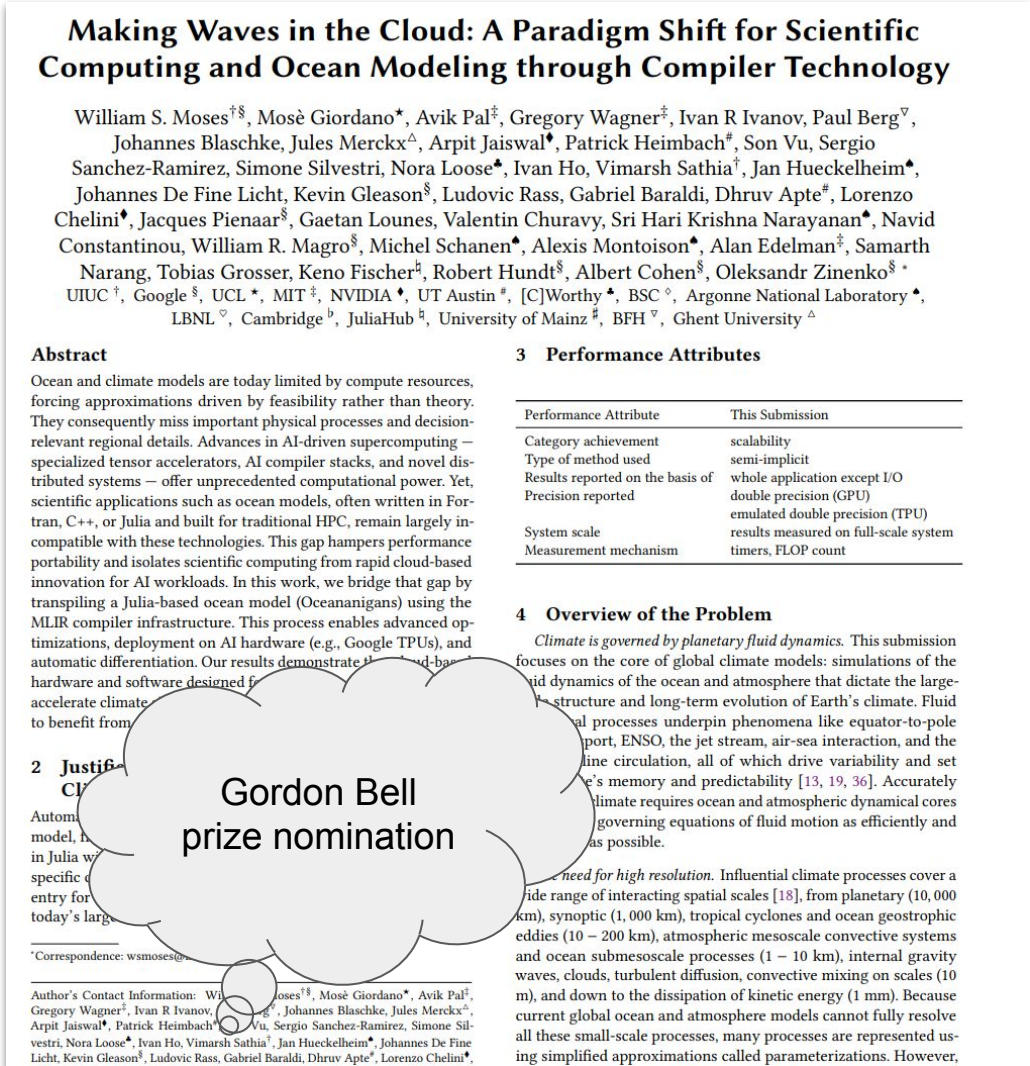
… need for high resolution. Influential climate processes cover a … ide range of interacting spatial scales [18], from planetary (10, 000 km), synoptic (1, 000 km), tropical cyclones and ocean geostrophic eddies (10 – 200 km), atmospheric mesoscale convective systems and ocean submesoscale processes (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

Simple 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 th… hardware and software designed f… accelerate climate… to benefit from…

2 Justifi… Cl…

Autom… model, h… in Julia w… specific … entry for … today's larg…

*Correspondence: wsmoses@…

Author's Contact Information: Wi… …oses[†§], Mosè Giordano[★], Avik Pal[‡], Gregory Wagner[‡], Ivan R Ivanov, … g[▽], Johannes Blaschke, Jules Merckx[△], Arpit Jaiswal[♦], Patrick Heimbach[#], … 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

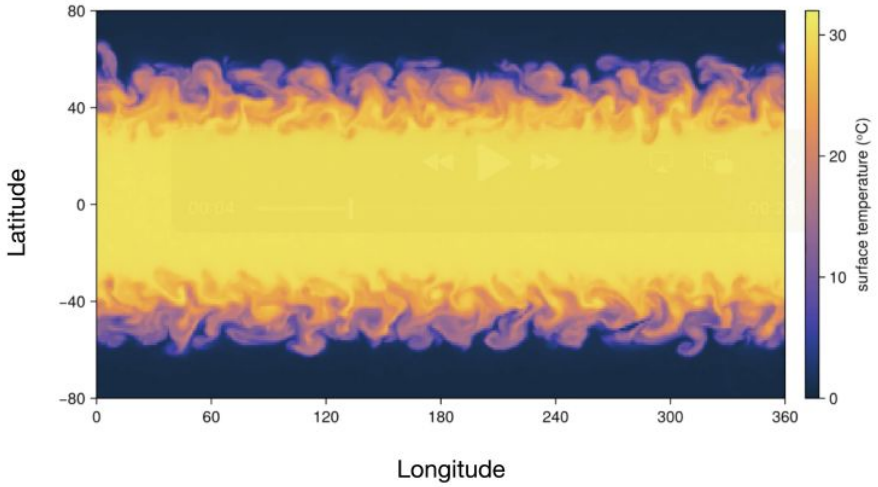
4 Overview of the Problem

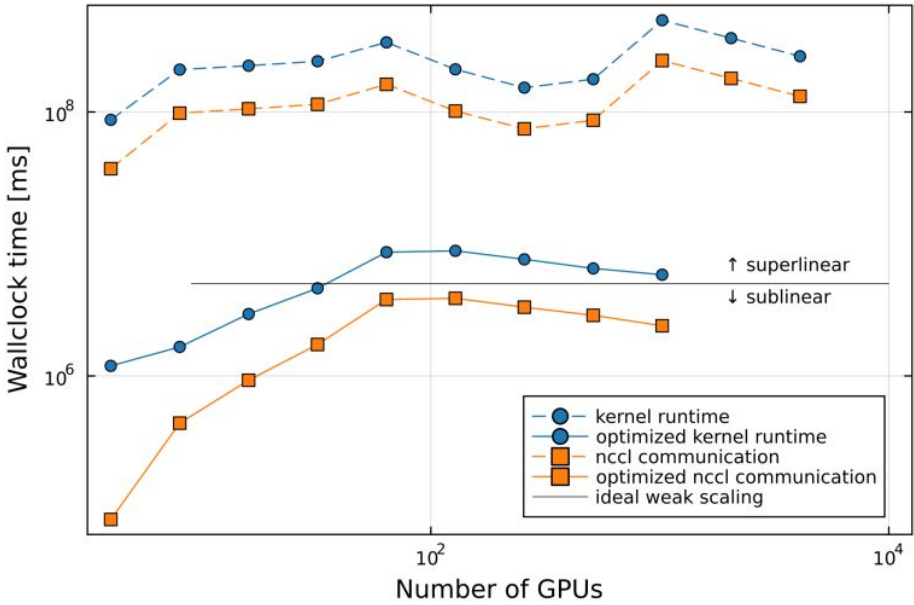
Climate is governed by planetary fluid dynamics. This submission focuses on the core of global climate models: simulations of the … id dynamics of the ocean and atmosphere that dictate the large- … structure and long-term evolution of Earth's climate. Fluid … al processes underpin phenomena like equator-to-pole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

… need for high resolution. Influential climate processes cover a … ide 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,

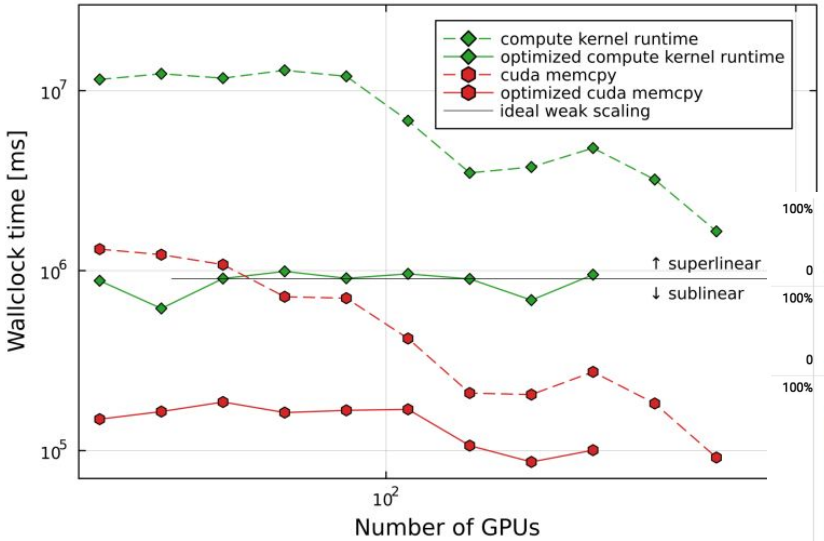
Gordon Bell prize nomination

Weak Scaling Experiments: GPU / Placement and Collectives





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



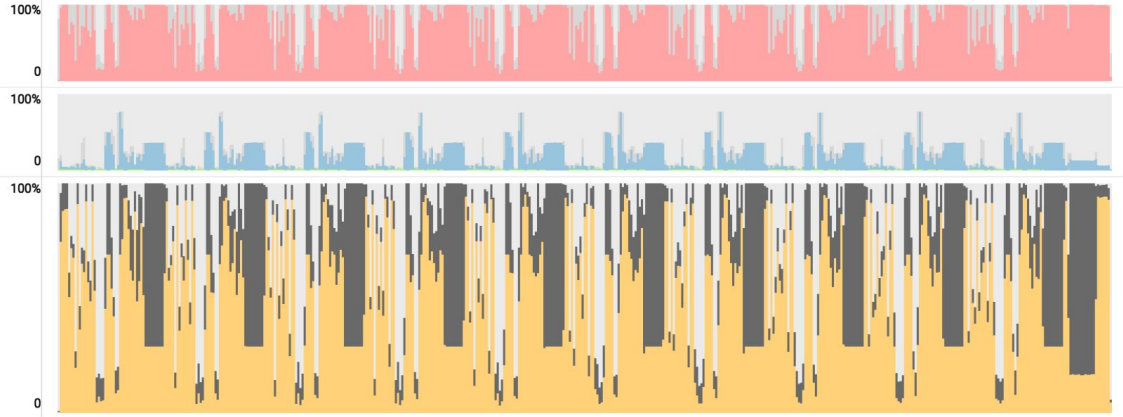
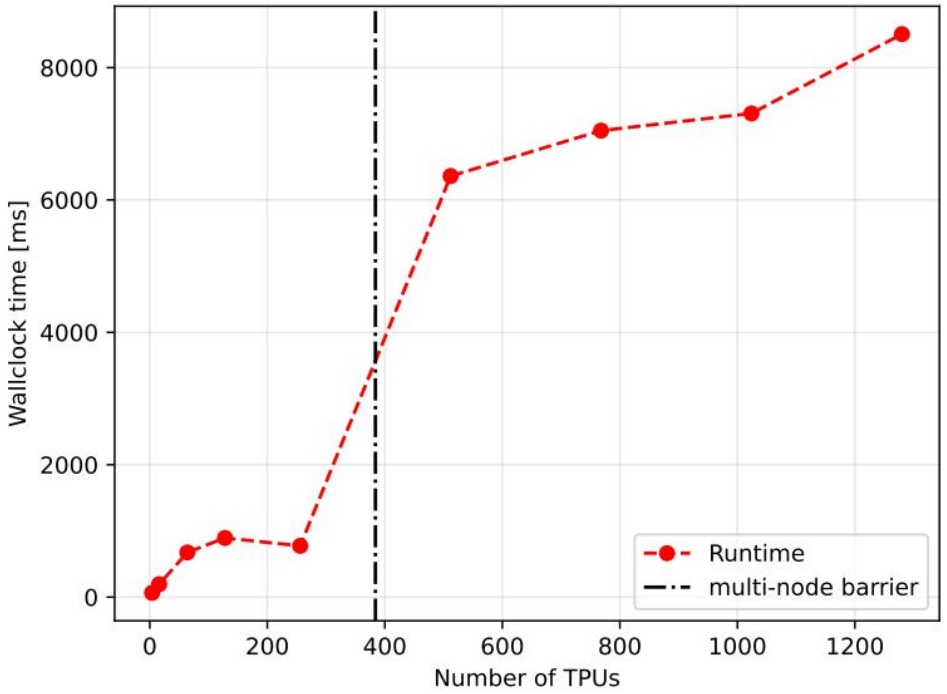


Figure 7: GPU utilization during 10 iterations of the benchmark problem on one Perlmutter GPU node (4 Nvidia A100s). Data collected with Nsight Systems. *top:* Percentage of SMs active, *middle:* Percentage saturation of active SMs, *bottom:* SM Warp Occupancy (yellow shows the active warps, gray shows inactive warps). During these 10 iterations on average 50.6% of compute warps were utilized, while an average of 26.6% compute warps were unallocated on active SMs.

Weak Scaling Experiments: 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%

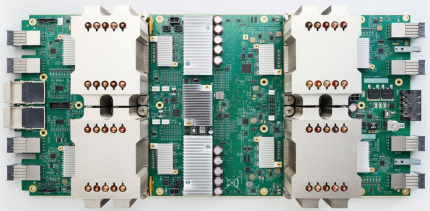
Table 1: Breakdown of TPU execution time by operation type, on a single node 4-TPU machine.

Q1: What missing interoperability layers (software, standard, or abstraction) would most accelerate convergence between traditional HPC linear algebra workflows and today's extreme-scale AI workloads?

Let's take a look.

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)
```

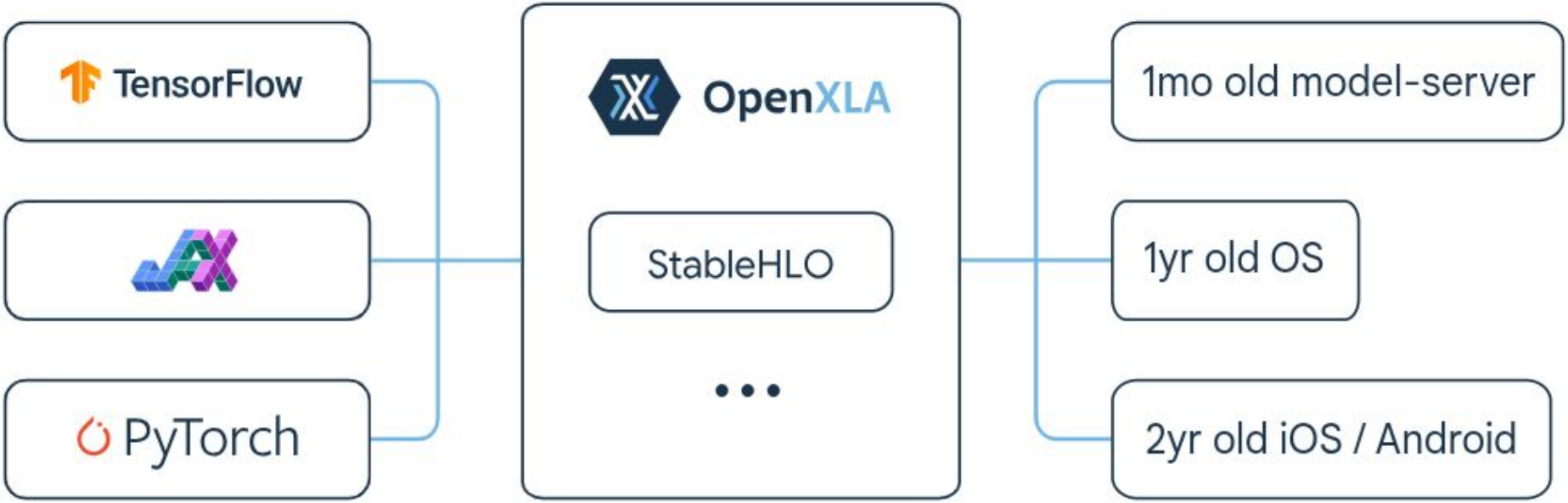


```
gradient_fun = jit(grad(loss))
perexample_grads = jit(vmap(grad(loss), (None, 0)))
```

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

StableHLO

https://openxla.org/stablehlo



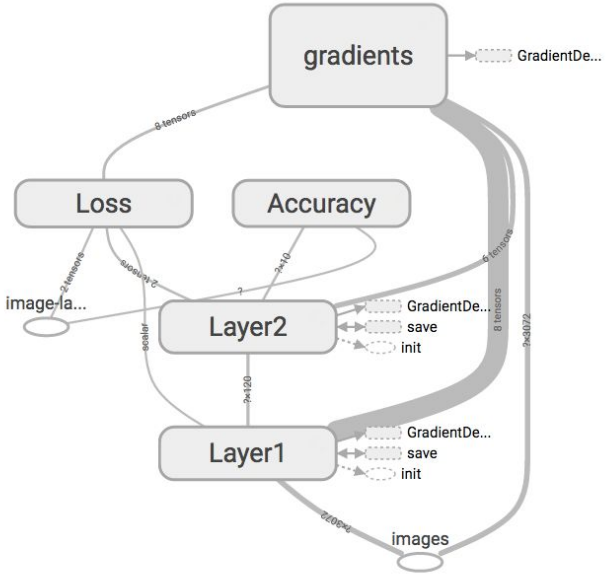
XLA / HLO / StableHLO

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**

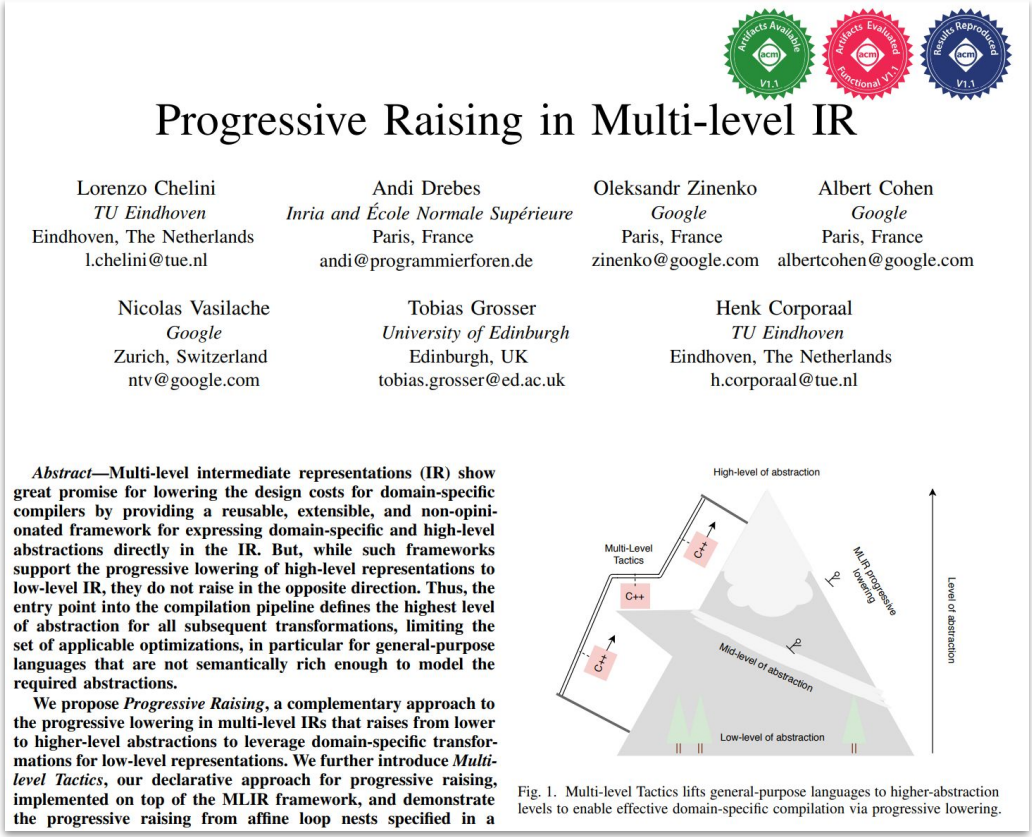
- **JIT / AOT compiler for linear algebra**
- Multiple backends: CPUs, GPUs, TPUs
- Eliminate dispatch overhead
- Fuse operations:
  avoid round trips to memory
- Specialization, global buffer analysis, vectorization, unrolling, etc.

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 the abstraction penalty? Performance predictability? Control for performance engineers?

HPC Systems are (also) Stuck in a Rut!

https://commons.wikimedia.org/wiki/File:Stuck_in_a_Rut.jpg

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

Abstract

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.

We explain how the evolution of hardware accelerators favors compiler back ends that hyper-optimize large monolithic kernels, show how this reliance on high-performance but inflexible kernels reinforces the dominant style of programming model, and argue these programming abstractions lack expressiveness, maintainability, and modularity; all of which hinders research progress.

We conclude by noting promising directions in the field, and advocate steps to advance progress towards high-performance general purpose numerical computing systems on modern accelerators.

ACM Reference Format:
Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *Workshop on Hot Topics in Operating Systems (HotOS '19), May 13–15, 2019, Bertinoro, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3317550.3321441

1 Compiling for modern accelerators

We became interested in this paper's subject when trying to improve an implementation of Capsule networks [1] to scale up to larger datasets. Capsule networks are an exciting machine learning research idea where scalar-valued "neurons" are replaced by small matrices, allowing them to capture more complex relationships. Capsules may or may not be the "next big thing" in machine learning, but they serve as a representative example of a disruptive ML research idea.

Although our convolutional Capsule model requires around 4 times fewer floating point operations (FLOPS)

HotOS '19, May 13–15, 2019, Bertinoro, Italy

ACM ISBN 978-1-4503-6727-1/19/05.
https://doi.org/10.1145/3317550.3321441

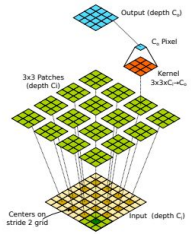


Figure 1. Conv2D operation with 3×3 kernel, stride=2

with 16 times fewer training parameters than the convolutional neural network (CNN) we were comparing it to, implementations in both TensorFlow[2] and PyTorch[3] were much slower and ran out of memory with much smaller models. We wanted to understand why.

1.1 New ideas often require new primitives

We won't discuss the full details of Capsule networks in this paper[1], but for our purposes it is sufficient to consider a simplified form of the inner loop, which is similar to the computation in a traditional CNN layer but operating on 4×4 matrices rather than scalars.

A basic building block of current machine learning frameworks is the strided 2D convolution. Most frameworks provide a primitive operation that accepts N input images of size $H \times W$, where each pixel has a "depth" of C_i channels[2]. Informally, for a "kernel size" K=3 and "stride" S=2, conv2d computes a weighted sum of overlapping 3×3 patches of pixels centered at every other (x, y) coordinate, to produce N smaller images with pixel depth C_o (Figure 1). Mathematically, this can be expressed as follows:

$$\forall n, x, y, c_o : O^{n,c_o}_{x,y} = \sum_{k_x} \sum_{k_y} \sum_{c_i} I^{n,c_i}_{sx+k_x,sy+k_y} \cdot K^{c_i,c_o}_{k_x,k_y} \quad (1)$$

where $\cdot$ denotes scalar multiplication, and O, I, and K are all 4-dimensional arrays of scalars. The resulting code is little more than 7 nested loops around a multiply-accumulate operation, but array layout, vectorization,

[1] For an excellent tutorial on Capsule networks see [4].
[2] Section 4 discusses why these dimensions are used.

https://doi.org/10.1145/3317550.3321441

Kernel Programming to the Rescue

Flurry of GPU acceleration options

CUDA Kernels / OpenCL-C
SYCL
CUTLASS
Triton (PyTorch, JAX)
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

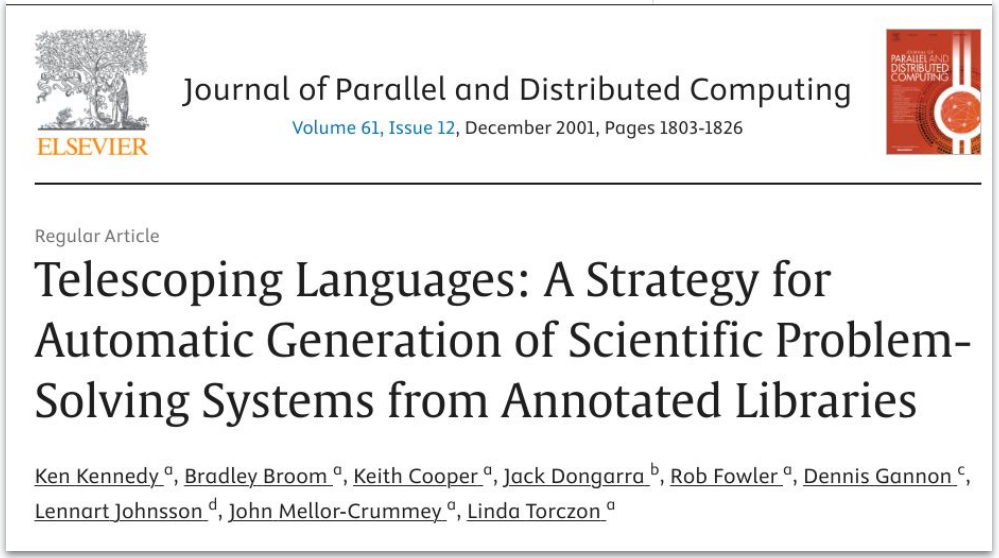
Motivations: escape hatch for...

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

Unify Kernel Programming with Compiler Automation!

Why? Hardware Optionality

- Compiler = HW-enabling differentiator
  ⟶ best-effort
- Kernel languages = siloed HW / SW stacks
  ⟶ all or nothing



ELSEVIER

Journal of Parallel and Distributed Computing

Volume 61, Issue 12, December 2001, Pages 1803-1826

Regular Article

Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries

Ken Kennedy [a], Bradley Broom [a], Keith Cooper [a], Jack Dongarra [b], Rob Fowler [a], Dennis Gannon [c], Lennart Johnsson [d], John Mellor-Crummey [a], Linda Torczon [a]

Staging and Partial Evaluation

Domain-specific generators

Multi-stage programming

(macro-expansion, quasi-quotation)

Late binding of kernel implementations
Optional cross-stage persistence

[SCP 2006]

In search of a program generator to implement generic transformations for high-performance computing

Albert Cohen[a,*], Sébastien Donadio[b], Maria-Jesus Garzaran[c], Christoph Herrmann[d], Oleg Kiselyov[e], David Padua[c]

[a] *ALCHEMY group, INRIA Futurs, Orsay, France*
[b] *PRiSM, University of Versailles, France*
[c] *DCS, University of Illinois at Urbana-Champaign, IL, USA*
[d] *FMI, University of Passau, Germany*
[e] *FNMOC, Monterey, CA, USA*

Received 1 January 2005; received in revised form 1 June 2005; accepted 1 October 2005

Abstract

The quality of compiler-optimized code for high-performance applications is far behind what optimization and domain experts can achieve by hand. Although it may seem surprising at first glance, the performance gap has been widening over time, due to the tremendous complexity increase in microprocessor and memory architectures, and to the rising level of abstraction of popular programming languages and styles. This paper explores in-between solutions, neither fully automatic nor fully manual ways to adapt a computationally intensive application to the target architecture. By mimicking complex sequences of transformations useful to optimize real codes, we show that generative programming is a practical means to implement architecture-aware optimizations for high-performance applications.

This work explores the promises of generative programming languages and techniques for the high-performance computing expert. We show that complex, architecture-specific optimizations *can* be implemented in a type-safe, purely generative framework. Peak performance is achievable through the careful combination of a high-level, multi-stage evaluation language – MetaOCaml – with low-level code generation techniques. Nevertheless, our results also show that generative approaches for high-performance computing do not come without technical caveats and implementation barriers concerning productivity and reuse. We describe these difficulties and identify ways to hide or overcome them, from abstract syntaxes to heterogeneous generators of code generators, combining high-level and type-safe multi-stage programming with a back-end generator of imperative code.

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_0
       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)

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

+ 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 …
```

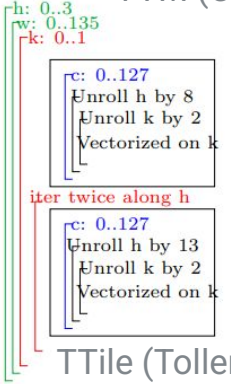
+ Map to Accelerator Tensor Instructions

```
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)



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)

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,'MNMIN>=500')
addContext(C1L1,'MNMIN<=M')
addContext(C1L1,'MNMIN<=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_2,2))
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 $r_p{}^{\text{th}}$ 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 $[f_p^1, \ldots, f_p^{(L-1)}, \text{syntactic}(r_p), f_p^L, \ldots, f_p^n]$

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

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

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

Fusion Fuse the loops at level L for the statements D with statement s_p going into the $r_p{}^{\text{th}}$ loop.

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

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

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

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

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

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

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

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

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

[LCPC 2005]

A Language for the Compact Representation of Multiple Program Versions

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

[1] BULL SA
[2] University of Versailles St-Quentin-en-Yvelines
[3] INRIA Futurs
[4] University of Illinois at Urbana-Champaign
[5] 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.

labri.fr/perso/barthou/xc.html

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=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 fo
version has only a subset of the fea

Combining Experimental Search
Optimization. Julien Jaeger and D
Machine Learning Approaches to A
January 2009. [bib | .pdf]

Loop Optimization using Adapti
Barthou, Sebastien Donadio, Alexa
Jalby. In *ACM/IEEE Int. Symp. on*
San Jose, California, March 2007.

Iterative Compilation with Kern
Alexandre Duchateau, William Jal
and Compilers for Parallel Compu
Science, pages 173-189, New Orle

A Language for the Compact Re
Sebastien Donadio, James Brodma
Albert Cohen, Maria Garzaran, Da
Languages and Compilers for Para
Computer Science, pages 136-151,
Verlag. [bib | .pdf]

The Compiler Has More Interfaces Than You Think

A Polynomial Spilling Heuristic: Layered Allocation

[CGO 2013]

Boubacar Diouf Albert Cohen Fabrice Rastello

Processor Virtualization and Split Compilation for Heterogeneous Multicore Embedded Systems

[DAC 2010]

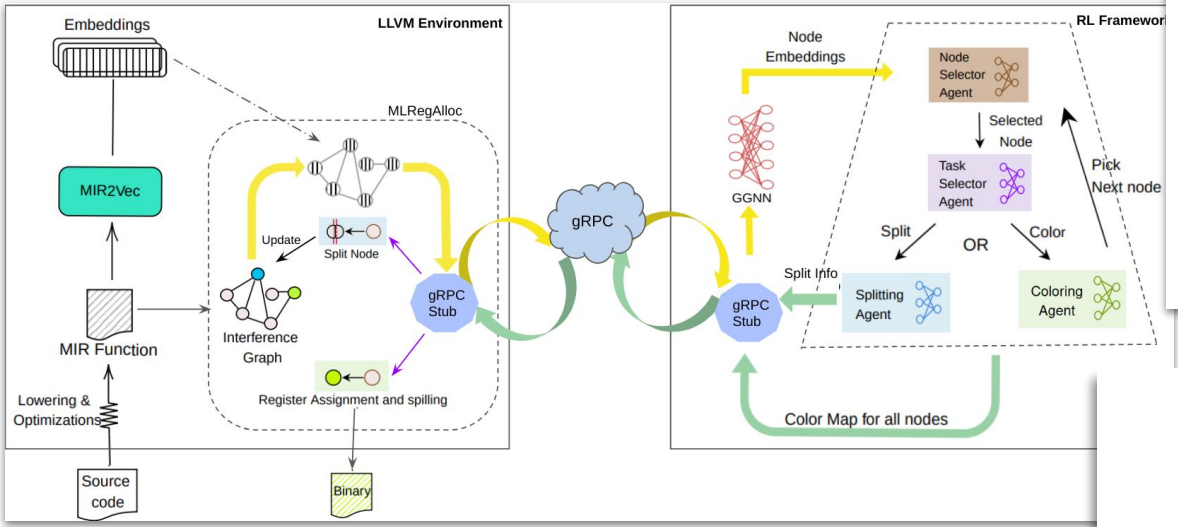
Albert Cohen Erven Rohou

Vapor SIMD: Auto-Vectorize Once, Run Everywhere

[CGO 2011]

Dorit Nuzman*, Sergei Dyshel*, Erven Rohou†, Ira Rosen*,
Kevin Williams†, David Yuste†, Albert Cohen‡, and Ayal Zaks*
*IBM Haifa Research Lab, Haifa, Israel – HiPEAC member, Email: (dorit,sergeid,irar,zaks)@il.ibm.com

ML + Compiler Construction



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 independent and 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-

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 p
grained control of general-purpose compilers by
the *Transform dialect*, a controllable IR-based tra
system implemented in MLIR. The Transform dia
ers performance engineers to optimize their v
pute loads by composing and reusing existing—l
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

2023 EURO LLVM DEVELOPERS' MEETING

Tutorial: Controllable Transformations in MLIR

Example: Python (JAX) Schedules

.py

```python
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

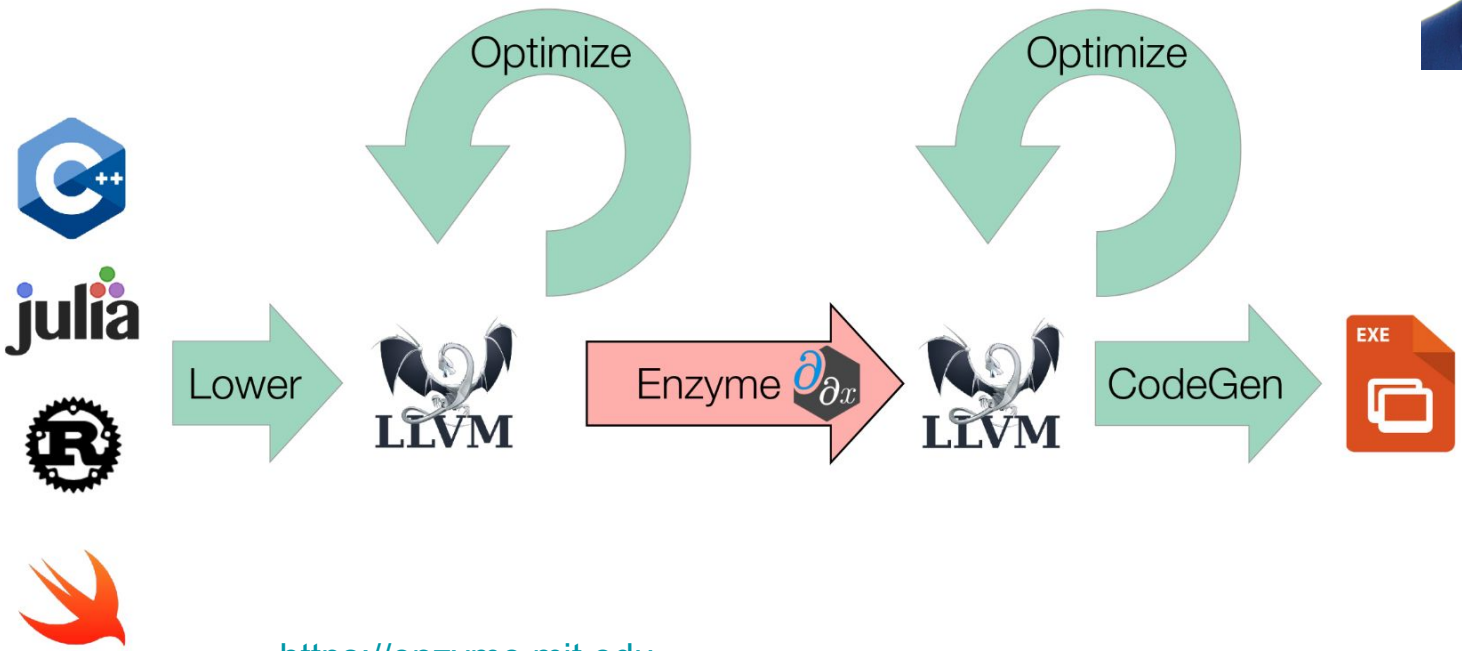
```
  module.run_pass("MyPassName")
```

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

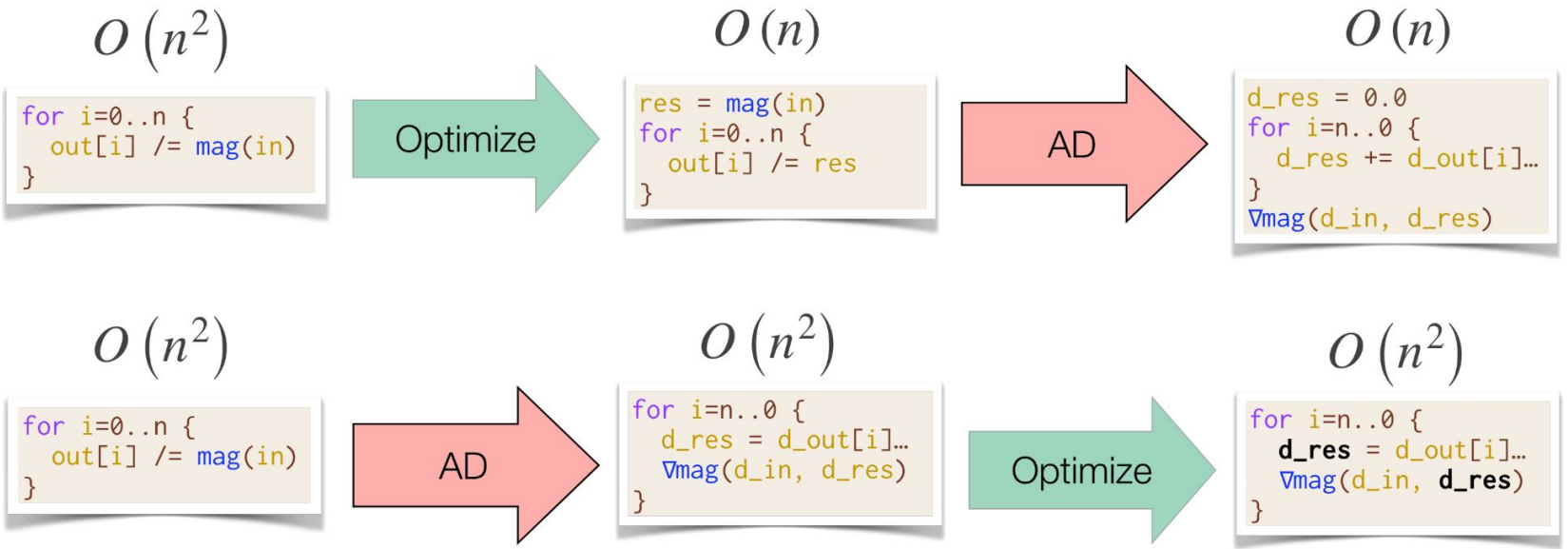
Enzyme Framework
Billy Moses (UIUC / Google)



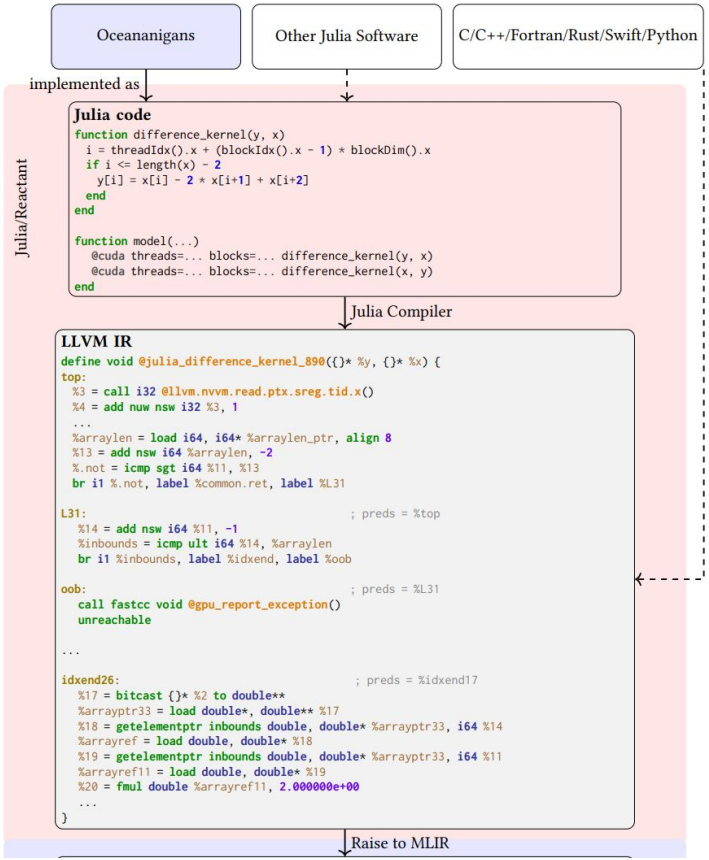
https://enzyme.mit.edu

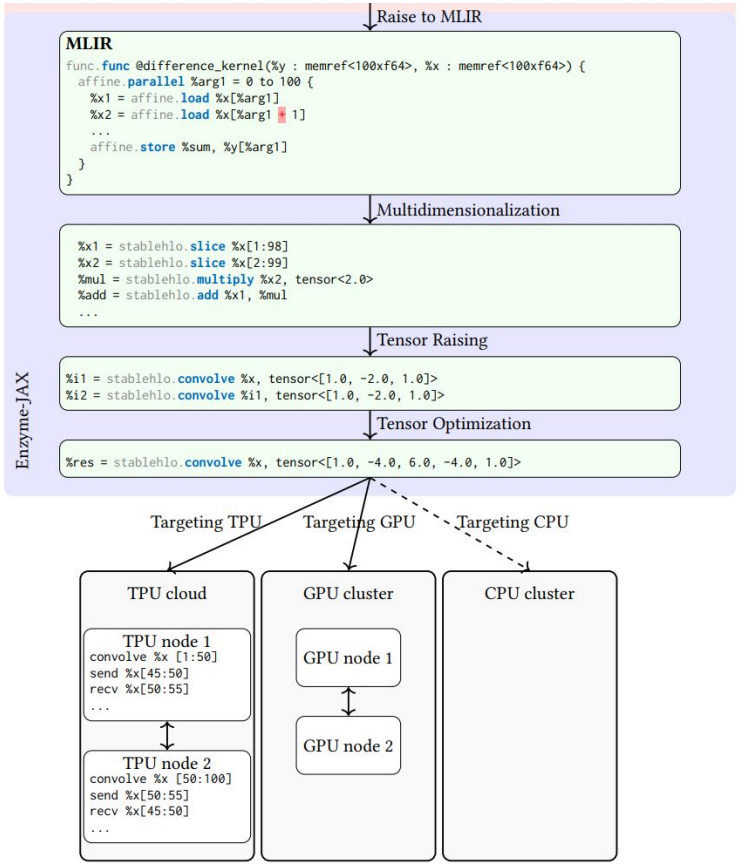
And now Also on MLIR, JAX, Julia! https://github.com/EnzymeAD/Enzyme-JAX

Enzyme Autodiff: Everything, Everywhere, All at Once

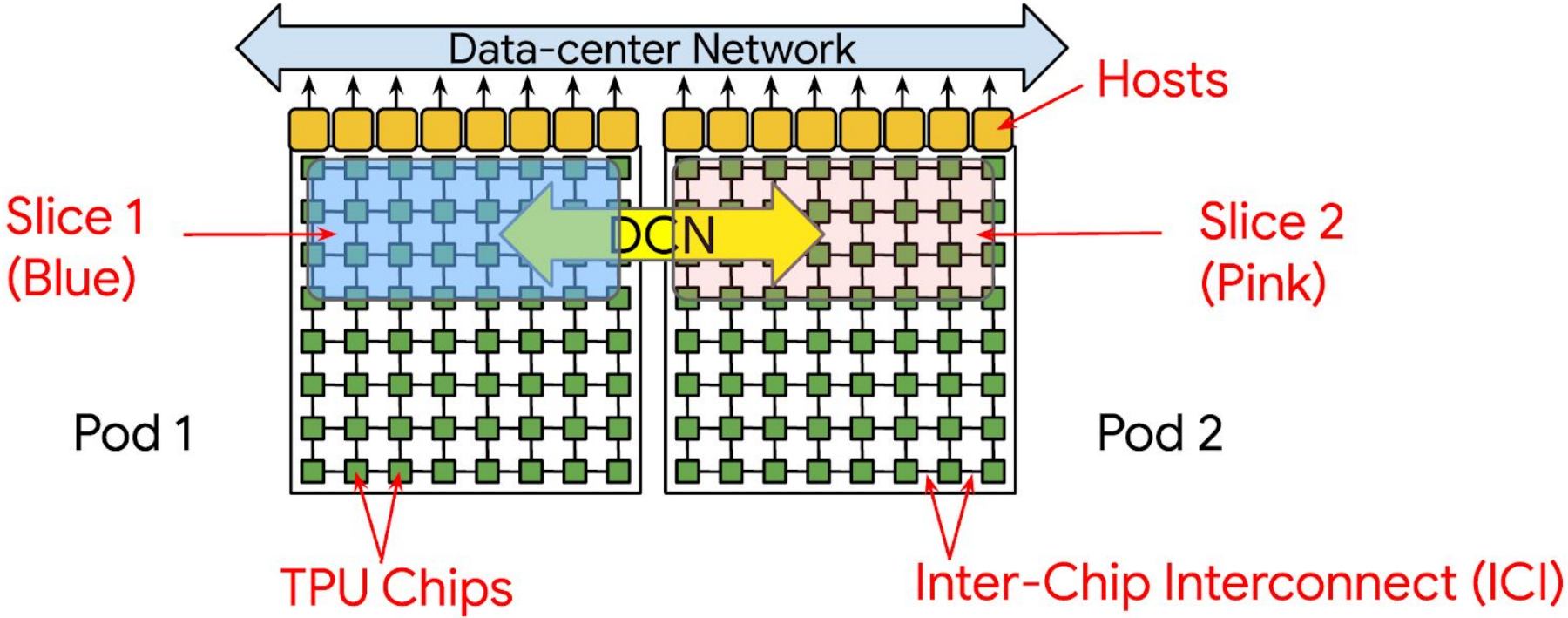


Computational Science → LLVM → MLIR → Heterogeneous Platform





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

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>
  }
}
```

HLO Is More Expressive Than You Think

Listing 1 Reactant code for compiling Julia functions

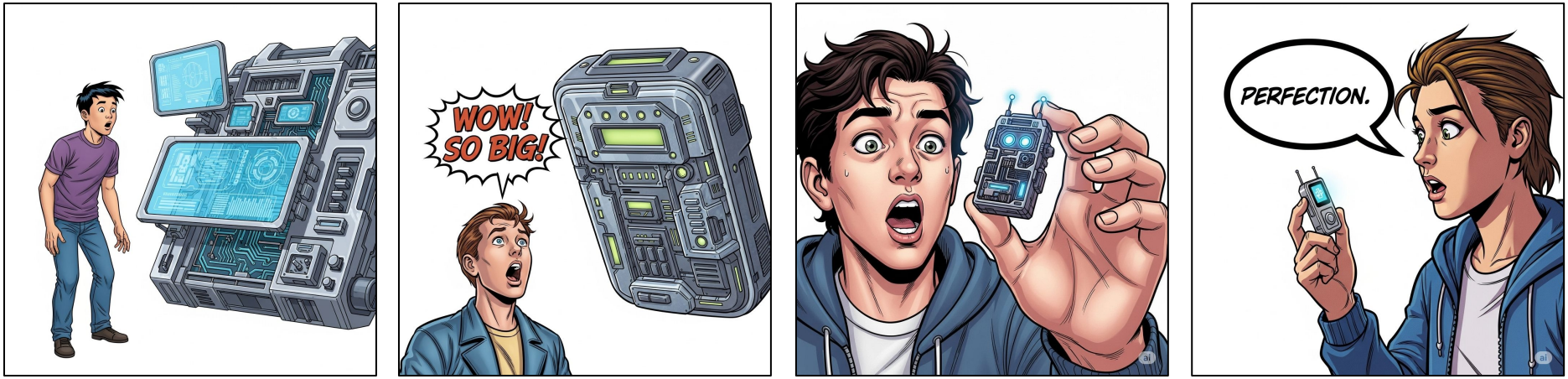
```
using Reactant

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>
  }
}
```

Q2: Looking ahead to 2030, do you expect the principal bottleneck for extreme-scale AI to be data, algorithms, resilience or energy, and how does that prediction shape your research priorities today?

The principal bottleneck is “smaller is better”



Generate a comic storyboard based on the "smaller is better" meme, with a white background.

Q3: Given the different developments in “computer architecture for AI” and “computational science”, do you think we'll see a convergence or divergence of roadmaps?

Convergence:

Divergence: ***commodity AI vs. supercomputing AI***