

MLIR Compiler Infrastructure Beyond Machine Learning

Alex Zinenko (Brium Inc.)

April 28, 2025

A dark blue diagonal gradient bar that starts from the bottom left and extends towards the top right, covering the lower half of the slide.

What Is a Compiler

compiler

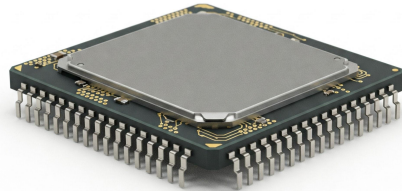
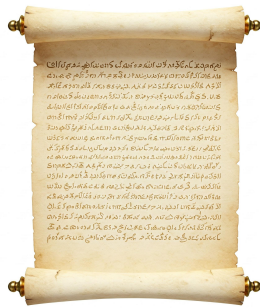
/kəm'praɪlə/

noun

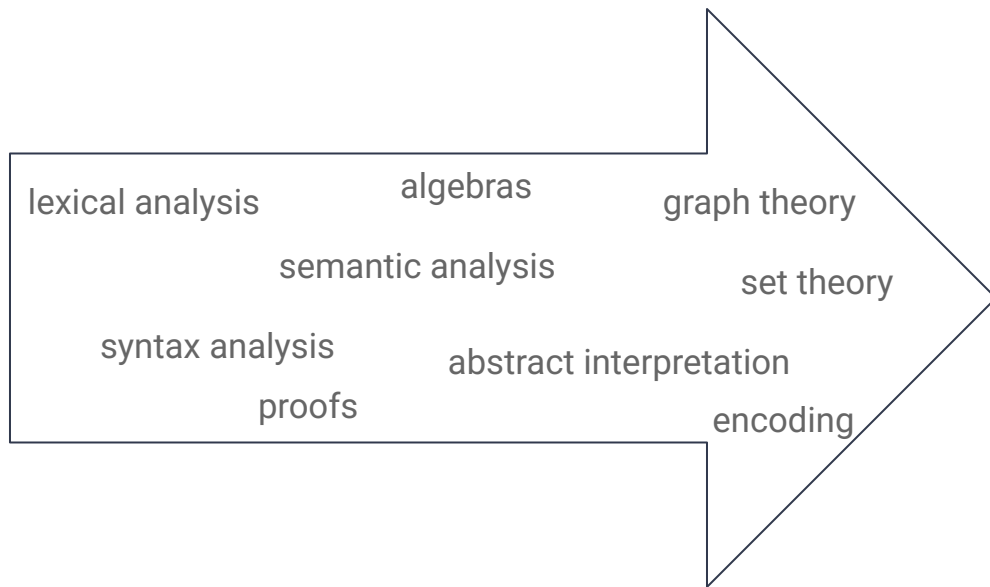
a program that converts instructions into a machine-code or lower-level form so that they can be read and executed by a computer.

"conversion would require more than just running it through a different compiler"

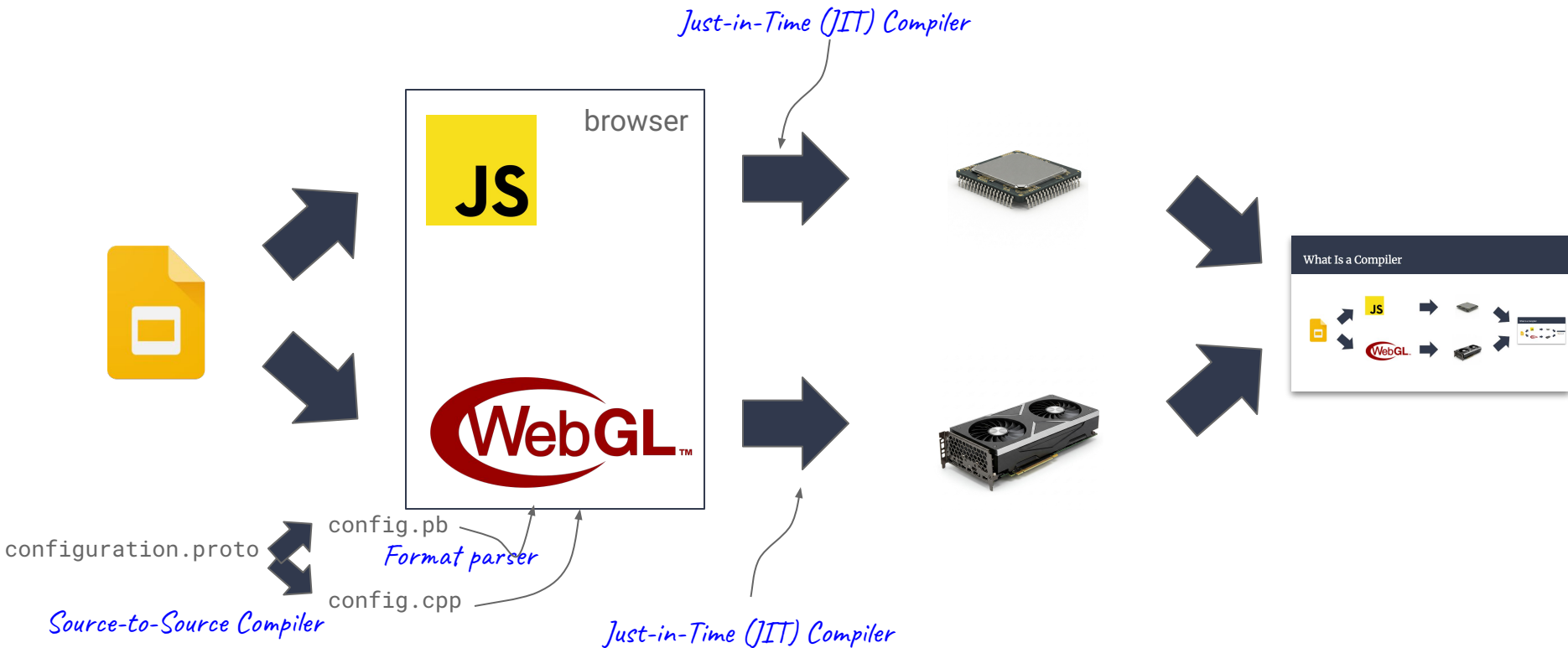
What Is a Compiler



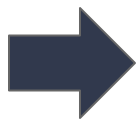
What Is a Compiler



What Is a Compiler



How Does a Compiler Work



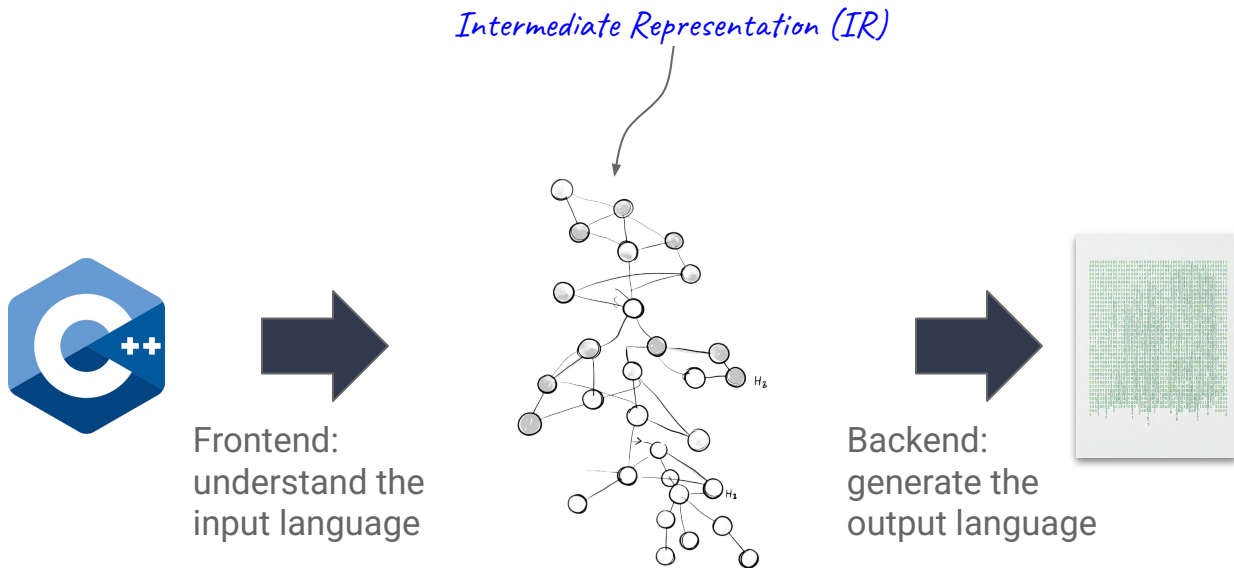
Frontend:
understand the
input language



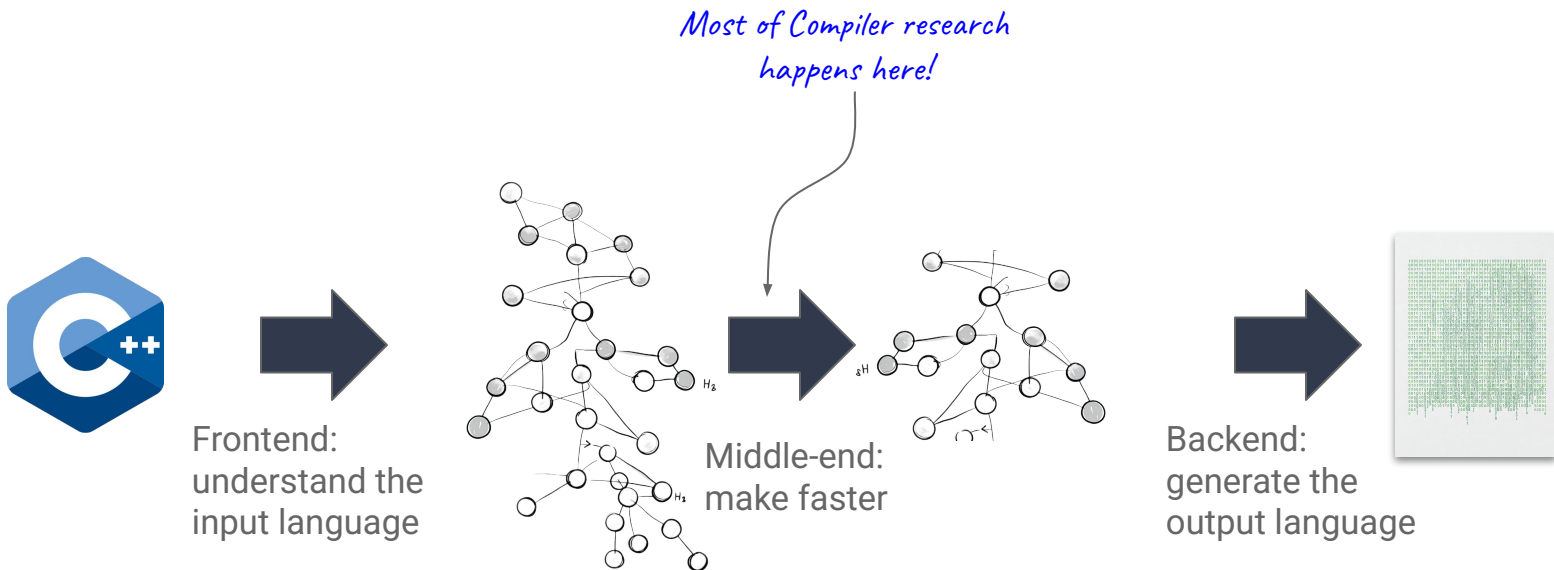
Backend:
generate the
output language



How Does a Compiler Work



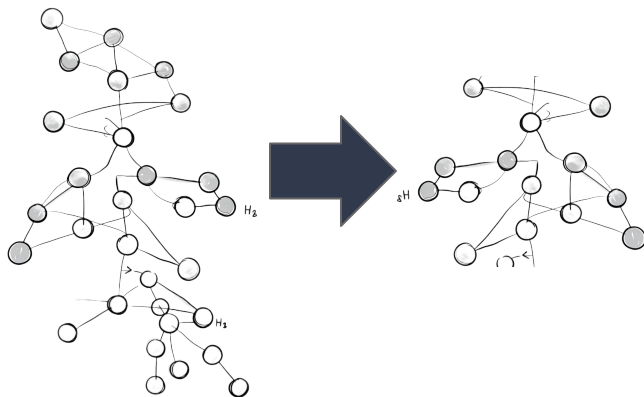
How Does a Compiler Work



How Does a Compiler Work



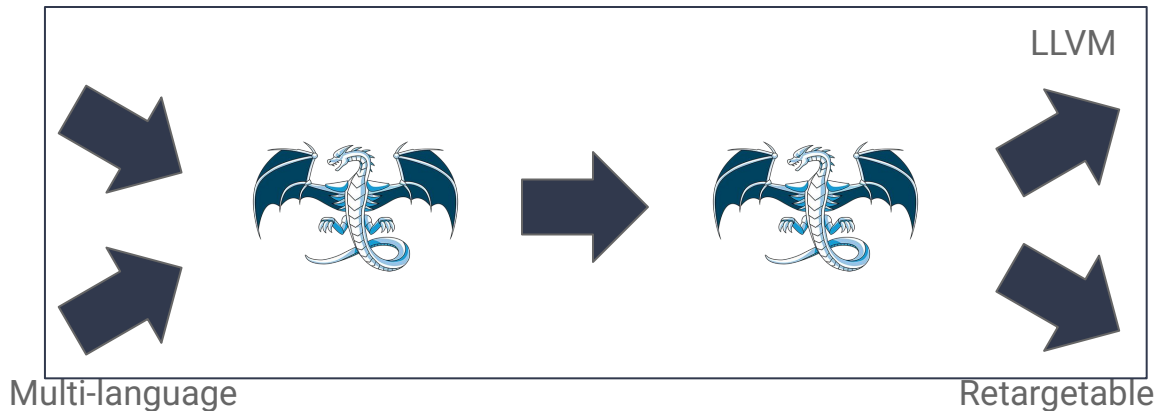
Multi-language



Retargetable



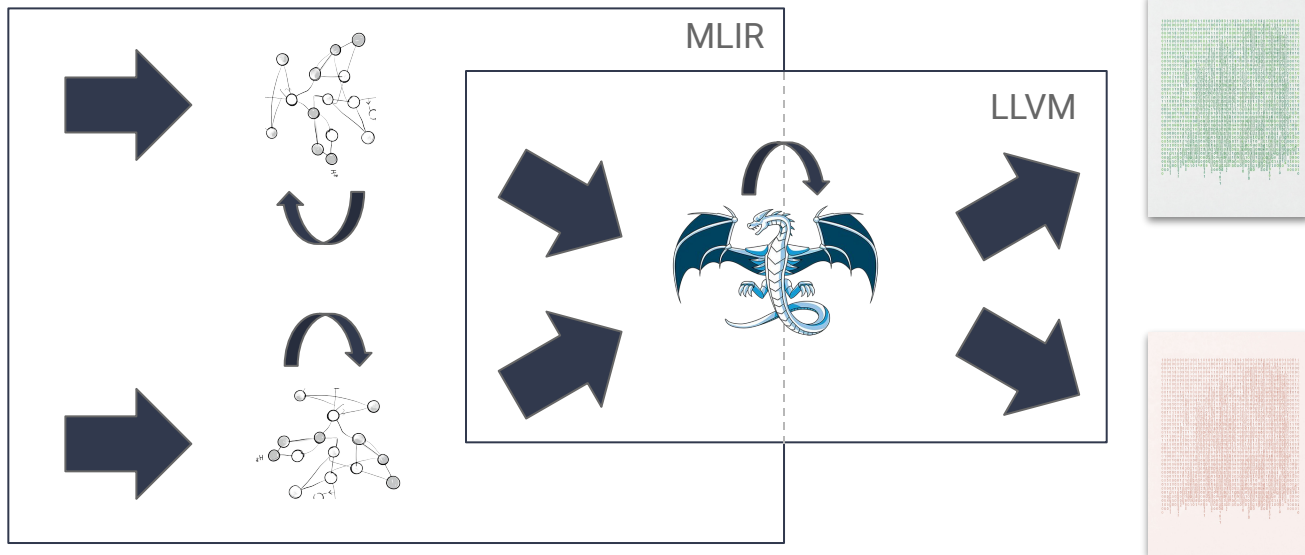
How Does a Compiler Work



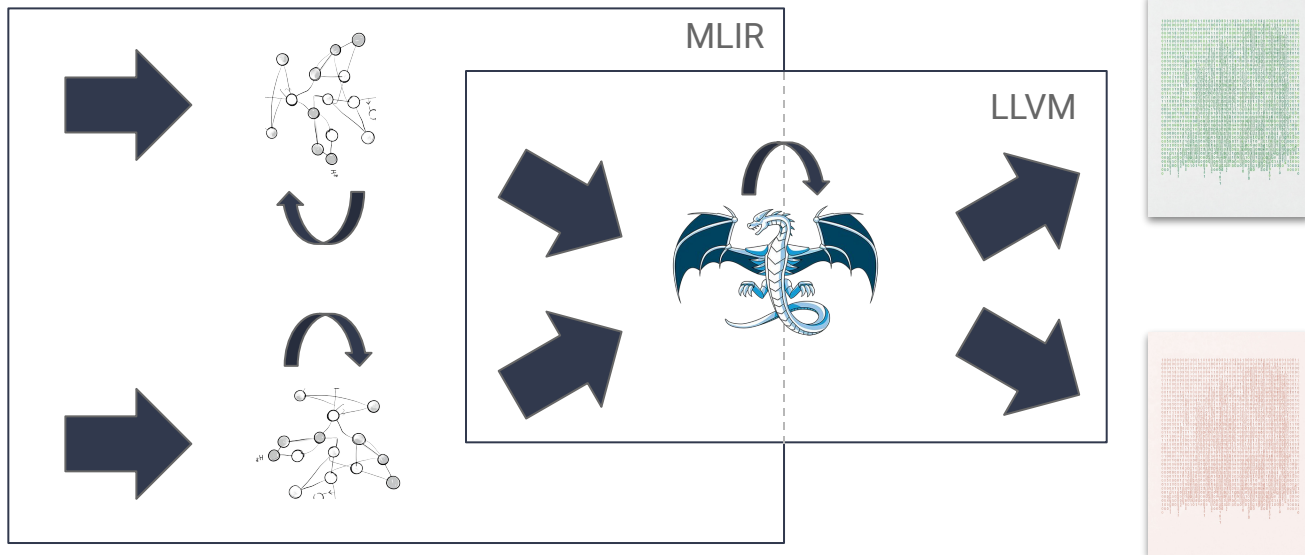
How Does a Compiler Work



≠



How Does a Compiler Work



What Is MLIR

MLIR

/əm əl ɪ ɑːr/

acronym

Multi-Level Intermediate Representation. A unifying software framework for compiler development.

"everybody thought ML for MLIR stood for Machine Learning"

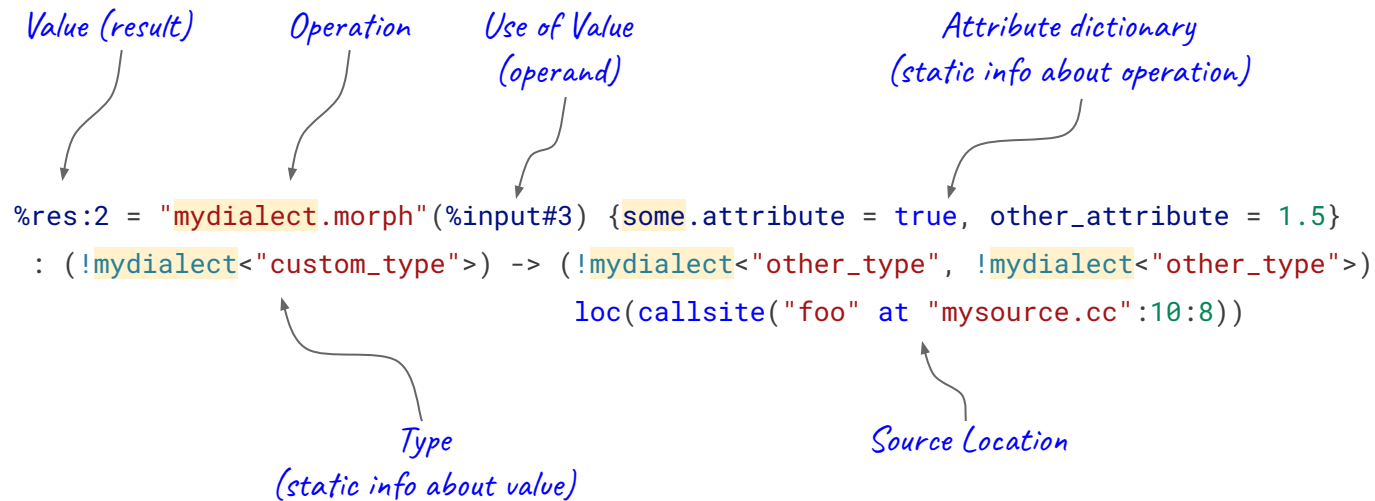
MLIR Structure

The diagram illustrates the structure of an MLIR instruction with the following components and annotations:

- Value (result):** Points to the result identifier `%res:2`.
- Operation:** Points to the operation name `"mydialect.morph"`.
- Use of Value (operand):** Points to the operand `%input#3`.
- Attribute dictionary (static info about operation):** Points to the attribute dictionary `{some.attribute = true, other_attribute = 1.5}`.
- Type (static info about value):** Points to the result type `(!mydialect<"custom_type">)`.
- Source Location:** Points to the source location `loc(callsite("foo" at "mysource.cc":10:8))`.

```
%res:2 = "mydialect.morph"(%input#3) {some.attribute = true, other_attribute = 1.5}
: (!mydialect<"custom_type">) -> (!mydialect<"other_type", !mydialect<"other_type">)
                                loc(callsite("foo" at "mysource.cc":10:8))
```

MLIR Structure



The diagram illustrates the structure of an MLIR instruction with the following components and annotations:

- Value (result):** Points to the variable `%res:2`.
- Operation:** Points to the operation `"mydialect.morph"`.
- Use of Value (operand):** Points to the operand `%input#3`.
- Attribute dictionary (static info about operation):** Points to the dictionary `{some.attribute = true, other_attribute = 1.5}`.
- Type (static info about value):** Points to the type `(!mydialect<"custom_type">)`.
- Source Location:** Points to the source location `loc(callsite("foo" at "mysource.cc":10:8))`.

```
%res:2 = "mydialect.morph"(%input#3) {some.attribute = true, other_attribute = 1.5}
: (!mydialect<"custom_type">) -> (!mydialect<"other_type", !mydialect<"other_type">)
                                loc(callsite("foo" at "mysource.cc":10:8))
```

MLIR Structure

```
%results:2 = "d.operation"(%arg0, %arg1) ({
```

```
  // Regions belong to Ops and can have multiple blocks.
```

Region

```
-> ()
```

```
}) : () -> (!d.type, !d.other_type)
```

MLIR Structure

```
%results:2 = "d.operation"(%arg0, %arg1) ({  
  // Regions belong to Ops and can have multiple blocks.  
  ^block(%argument: !d.type):  
    ^other_block:  
      | "d.terminator"() [^block(%argument : !d.type)] : ()  
-> ()  
}) : () -> (!d.type, !d.other_type)
```

Region

Block

MLIR Structure

```
%results:2 = "d.operation"(%arg0, %arg1) ({  
  // Regions belong to Ops and can have multiple blocks.  
  ^block(%argument: !d.type):  
    %value = "nested.operation"() ({  
      }) : () -> (!d.other_type)  
    "consume.value"(%value) : (!d.other_type) -> ()  
  ^other_block:  
    | "d.terminator"() [^block(%argument : !d.type)] : ()  
-> ()  
}) : () -> (!d.type, !d.other_type)
```

Region

Block

MLIR Structure

```
%results:2 = "d.operation"(%arg0, %arg1) ({  
  // Regions belong to Ops and can have multiple blocks. Region  
  ^block(%argument: !d.type): Block  
    %value = "nested.operation"() ({  
      // Ops can contain nested regions. Region  
      "d.op"() : () -> ()  
    }) : () -> (!d.other_type)  
    "consume.value"(%value) : (!d.other_type) -> ()  
  ^other_block:  
    | "d.terminator"() [^block(%argument : !d.type)] : ()  
-> ()  
}) : () -> (!d.type, !d.other_type)
```

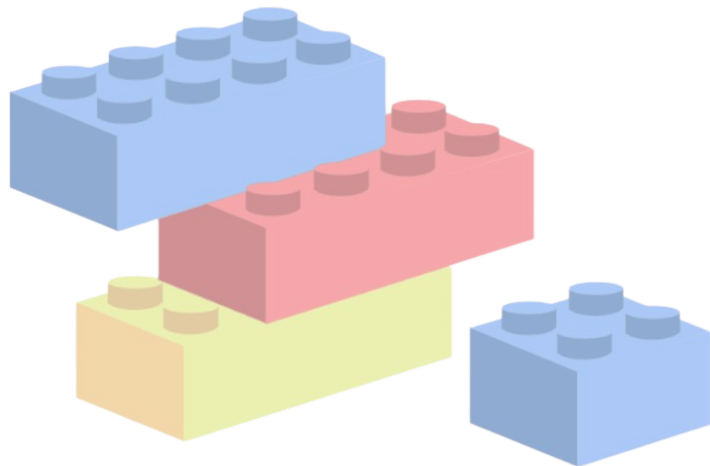
Little Builtin, Everything Customizable

No fixed set of:

- Operations
- Attributes
- Types

Bring your own anything:

- As long as you define and verify semantics
- Group into “dialects”

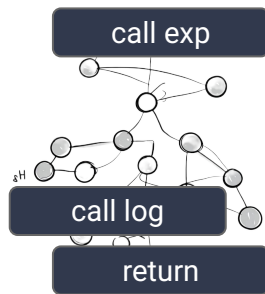
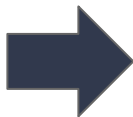


Representation Matters



```
#include <math.h>

double foo(double x) {
    return log(exp(x));
}
```



```
define double @foo(double noundef %0) {
    %2 = tail call double @llvm.exp.f64(double %0)
    %3 = tail call double @llvm.log.f64(double %2)
    ret double %3
}
```



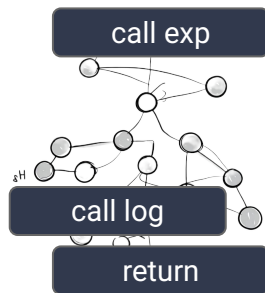
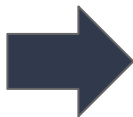
clang -S -emit-llvm -O3

Representation Matters



```
#include <math.h>

double foo(double x) {
    return log(exp(x));
}
```



```
define double @foo(double noundef %0) {
    %2 = tail call double @llvm.exp.f64(double %0)
    %3 = tail call double @llvm.log.f64(double %2)
    ret double %3
}
```



log e^x = x ???

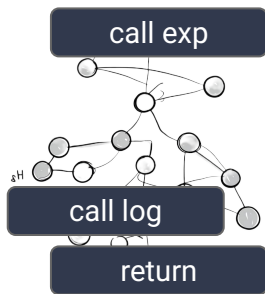
clang -S -emit-llvm -O3

Representation Matters



```
#include <math.h>

double foo(double x) {
    return log(exp(x));
}
```



```
define double @foo(double noundef %0) {
    ret double %0
}
```



```
clang -S -emit-llvm -ffast-math -O3
```

Representation Matters

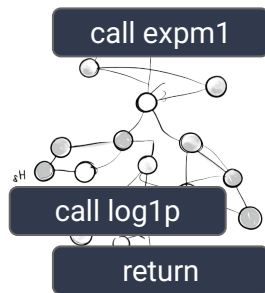


```
#include <math.h>

double foo(double x) {
    return log1p(expm1(x));
}
```

$\log(1+x)$

$e^x - 1$



clang -S -emit-llvm -ffast-math -O3

LLVM IR has intrinsics for e^x and \log , but not for $\expm1$ and $\log1p$



```
define double @foo(double noundef %0) {
    %2 = tail call fast double @expm1(double %0)
    %3 = tail call fast double @log1p(double %2)
    ret double %3
}
```

$\log(1 + e^x - 1) = \log e^x = x$???

Representation Matters

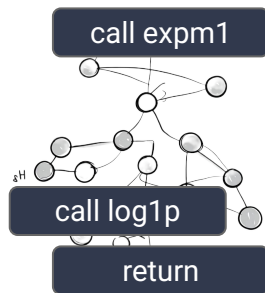


```
#include <math.h>

double foo(double x) {
    return log1p(expm1(x));
}
```

$\log(1+x)$

$e^x - 1$



```
func.func @bar(%0 : f64) -> f64 {
    %1 = math.expm1 %0 fastmath<fast> : f64
    %2 = math.log1p %1 fastmath<fast> : f64
    return %2 : f64
}
```

In MLIR, these live in the Math dialect and are “optional”.



What Can Be Represented

48 dialects

“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'lvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcln' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
OpInterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture (TOSA) Dialect
Transform Dialect

arith

Arithmetic operations

```
arith.addf %a, %b
```

math

Mathematical functions (≈ libm)

```
math.exp %a
```

What Can Be Represented

48 dialects

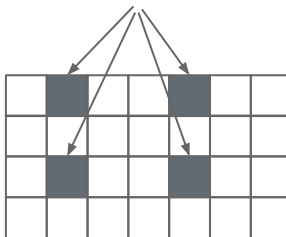
“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcln' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
OpInterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture (TOSA) Dialect
Transform Dialect

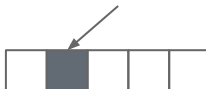
memref

ptr

Multidimensional memory references



Pointers



What Can Be Represented

48 dialects

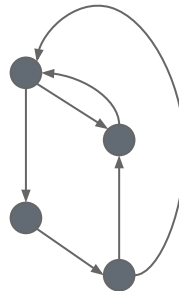
“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'lvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smi' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcll' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
Opinterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture (TOSA) Dialect
Transform Dialect

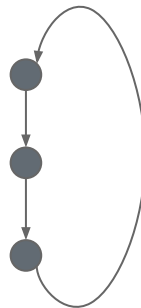
cf

scf

Graph Control flow (goto)



Structured Control flow (loops)



What Can Be Represented

48 dialects

“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vclit' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
Opinterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture
(TOSA) Dialect
Transform Dialect

linalg

Structured Linear Algebra

```
linalg.generic {  
  iterators = [ "parallel", "parallel", "reduction" ],  
  indexing_maps = [  
    affine_map<(i, j, k) -> (i, k),  
    affine_map<(i, j, k) -> (k, j),  
    affine_map<(i, j, k) -> (i, j)  
  ]  
} ins(memref<?x?xf32>, memref<?x?xf32>)  
outs(memref<?x?xf32>) {  
  ^bb0(%a: f32, %b: f32, %c: f32):  
    %0 = arith.mulf %a, %b : f32  
    %1 = arith.addf %0, %c : f32  
  yield %c : f32  
}
```

tensor
tosa

Tensor Arithmetics (ML-style)

tosa.matmul

What Can Be Represented

48 dialects

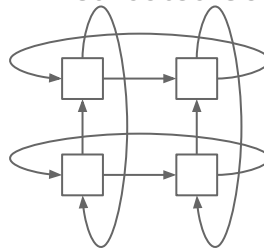
“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcln' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
OpInterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture (TOSA) Dialect
Transform Dialect

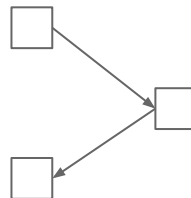
mesh

mpi

Distributed Computation



MPI



What Can Be Represented

48 dialects

“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'lvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcln' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
OpInterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture
(TOSA) Dialect
Transform Dialect

acc

OpenACC “pragmas”

OpenACC

openmp

OpenMP “pragmas”

OpenMP

What Can Be Represented

48 dialects

“upstream”

Dialects
'acc' Dialect
'affine' Dialect
'amdgpu' Dialect
'amx' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dlt' Dialect
'emitc' Dialect
'func' Dialect
'gpu' Dialect
'index' Dialect
'irdl' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'mpi' Dialect
'nvgpu' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdl' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocdl' Dialect
'scf' Dialect
'shape' Dialect
'smn' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcln' Dialect
'vector' Dialect
'x86vector' Dialect
'xegpu' Dialect
Builtin Dialect
Opinterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture (TOSA) Dialect
Transform Dialect

gpu

GPU programming abstraction

```
func.func @no_args(%sz : index) {  
  // Normal (host) function.  
  gpu.launch blocks(%bx, %by, %bz)  
    in (%grid_x = %sz, %grid_y = %sz, %grid_z = %sz)  
    threads(%tx, %ty, %tz)  
    in (%block_x = %sz, %block_y = %sz, %block_z = %sz) {  
    // GPU kernel. Code motion is allowed between the two.  
    gpu.terminator  
  }  
  return  
}
```

nvgpu

CUDA abstraction
(we also have ROCm equivalent)

```
nvgpu.tma.async.load  
nvgpu.mma.sync
```

What Can Be Represented

Dialects

'acc' Dialect
'affine' Dialect
'amdgpv' Dialect
'ams' Dialect
'arith' Dialect
'arm_neon' Dialect
'arm_sve' Dialect
'Armv8ME' Dialect
'async' Dialect
'bufferization' Dialect
'cf' Dialect
'complex' Dialect
'dtr' Dialect
'emile' Dialect
'func' Dialect
'gpv' Dialect
'index' Dialect
'irfi' Dialect
'linalg' Dialect
'llvm' Dialect
'math' Dialect
'memref' Dialect
'mesh' Dialect
'ml_program' Dialect
'nrv' Dialect
'nvvp' Dialect
'nvvm' Dialect
'omp' Dialect
'pdl_interp' Dialect
'pdf' Dialect
'polynomial' Dialect
'ptr' Dialect
'quant' Dialect
'rocd' Dialect
'scf' Dialect
'shape' Dialect
'smf' Dialect
'sparse_tensor' Dialect
'tensor' Dialect
'ub' Dialect
'vcir' Dialect
'vector' Dialect
'y8bvector' Dialect
'xegpv' Dialect
BuiltIn Dialects
OpInterface definitions
SPIR-V Dialect
Tensor Operator Set Architecture
(TOSA) Dialect
Transform Dialect

48 dialects

“upstream”



100s

“downstream”

Users of MLIR

Accera

Accera is a compiler that enables you to experiment with loop optimizations without hand-writing Assembly code. With Accera, these problems and impediments can be addressed in an optimized way. It is available as a Python library and supports cross-

Beaver

Beaver is an MLIR frontend in LLVM features. Beaver provides a simple

BroZuMLIR: A Format

BroZuMLIR applies MLIR to the low-level format's strengths. For example, we

IREE

IREE (pronounced "ree") is a compiler against a HAL, Hardware Abstraction and run ML devices in a variety of

Catalyst

Catalyst is an AOT/IT compiler for full auto-differentiation support

CIRCT: Circuit IR Compiler

The CIRCT project is an (upstream) to the domain of hardware design

Concrete: TFHE Compiler

Concrete is an open-source framework makes writing FHE programs easy

Lingo DB: Revolutionizing

LingoDB is a cutting-edge data processing and analytics engine

DSP-MLIR: A Framework

DSP-MLIR is a framework designed and rewrites patterns that detect DAG

Enzyme: General Autodiff

Enzyme (specifically EnzymeMLIR) types implement or inherit general

MLIR-EmC

MLIR-EmC provides a way to translate Keras and TensorFlow models

Flagg

Flagg is a ground-up implementation of the MLIR-EmC repository.

Mojo

Mojo is a new programming language built of Python syntax with systems

VAST

VAST is a library for program analysis and instrumentation of C/C++ and related languages. VAST provides a

Verona

Project Verona is a research programming language to explore the concept of concurrent ownership. They are

HEIR

HEIR (Homomorphic Encryption Intermediate Representation) is an MLIR-based toolchain developed by

ONNX-MLIR

ONNX-MLIR is a high-performance MLIR-based end-to-end compiler for DL and non-DL computations. It can

OpenMLA

OpenMLA is a community-driven, open source ML compiler ecosystem, using the best of XLA & MLIR.

PlaidML

PlaidML is a tensor compiler that facilitates reusable and performance portable ML models across various

PolyBlocks: An MLIR-based JIT and AOT compiler

PolyBlocks is a high-performance MLIR-based end-to-end compiler for DL and non-DL computations. It can

Polygeist: C/C++ frontend and optimizations for MLIR

Polygeist is a C/C++ frontend for MLIR which preserves high-level structure from programs such as parallelism.

Pylir

Pylir aims to be an optimizing Ahead-of-Time Python Compiler with high language conformance. It uses MLIR

RISE

RISE is a spiritual successor to the LLVM project: "a high-level functional data parallel language with a system

SOPHGO TPU-MLIR

TPU-MLIR is an open-source machine-learning compiler based on MLIR for SOPHGO TPU

Substrait MLIR

Substrait MLIR is an input/output dialect for Substrait, the cross-language serialization format of database

TensorFlow

TensorFlow is used as a Graph Transformation framework and the foundation for building many tools (XLA, TF Lite

Tensorflow MLIR Compiler

Tensorflow MLIR is a project aimed at defining MLIR dialects to abstract compute on Tensorflow AI

TFRT: TensorFlow Runtime

TFRT aims to provide a unified, extensible infrastructure layer for an asynchronous runtime system.

Torch-MLIR

The Torch-MLIR project aims to provide first class compiler support from the PyTorch ecosystem to the MLIR

Triton

Triton is a language and compiler for writing highly efficient custom Deep-Learning primitives. The aim of

VAST: C/C++ frontend for MLIR

VAST is a library for program analysis and instrumentation of C/C++ and related languages. VAST provides a

Verona

Project Verona is a research programming language to explore the concept of concurrent ownership. They are

<https://mlir.lvm.org/users/>

How to Handle Generality: Traits


```
Traits: [Associative, Commutative, Pure, ...  
        SameOperandAndResultType]
```

```
%3 = arith.addi %1, %2
```

```
%4 = matrix.multiply %5, %6
```

```
Traits: [Associative, AntiCommutative, Pure, ...  
        LoopDecomposable]
```

*Transformations reason about
traits, not individual operations*



```
void transformation(Operation *op) {  
    if (op->hasTrait<Associative>()) {  
        Value operand = op->getOperand(0);  
        op->setOperand(0, op->getOperand(1));  
        op->setOperand(1, operand);  
    }  
}
```

How to Handle Generality: Interfaces

Ifaces: [ConditionallyAssociative, ...
ConditionallyCommucative]

Traits: [~~Associative~~, ~~Commutative~~, Pure, ...
SameOperandAndResultType]

bool AddFOp::isAssociative() "override" {
 return getFastMathAttr().isAllowReassoc();
}

%3 = arith.addf %1, %2

%4 = matrix.multiply %5, %6

Traits: [Associative, Pure, ...
LoopDecomposable]

Ifaces: [AntiCommutative]

Value MatrixMultiplyOp::buildInverse(OpBuilder &b, ...) "override" {
 return b.create<MatrixInverseOp>(...).getResult();
}

Why Bother?

Why Bother? Climate Model Domain Compilers

Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation

TOBIAS GYSI and CHRISTOPH MÜLLER, ETH Zurich, Switzerland
OLEKSANDR ZINENKO, Google, France
STEPHAN HERHUT, Google, Germany
EDDIE DAVIS, TOBIAS WICKY, and OLIVER FUHRER, Vulcan Inc, USA
TORSTEN HOEFLER, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, UK

Most compilers have a single core intermediate representation (IR) (e.g., LLVM) sometimes complemented with vaguely defined IR-like data structures. This IR is commonly low-level and close to machine instructions. As a result, optimizations relying on domain-specific information are either not possible or require complex analysis to recover the missing information. In contrast, multi-level rewriting instantiates a hierarchy of dialects (IRs), lowers programs level-by-level, and performs code transformations at the most suitable level. We demonstrate the effectiveness of this approach for the weather and climate domain. In particular, we develop a prototype compiler and design stencil- and GPU-specific dialects based on a set of newly introduced design principles. We find that two domain-specific optimizations (500 lines of code) realized on top of LLVM's extensible MLIR compiler infrastructure suffice to outperform state-of-the-art solutions. In essence, multi-level rewriting promises to herald the age of specialized compilers composed from domain- and target-specific dialects implemented on top of a shared infrastructure.

CCS Concepts: • Software and its engineering → Compilers; Domain specific languages; • Applied computing → Earth and atmospheric sciences;

Additional Key Words and Phrases: Weather and climate, stencil computations, intermediate representations

Tobias Gysi also with Google.

Tobias Grosse also with ETH Zurich.

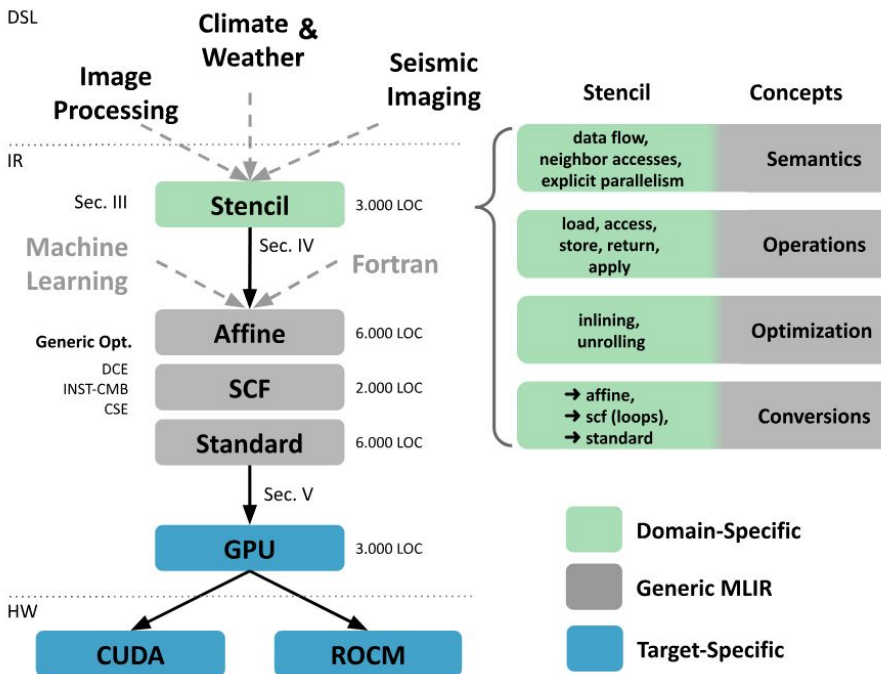
The work done at ETH Zurich has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs.
Authors' addresses: T. Gysi, C. Müller, and T. Hoefer, ETH Zurich, Switzerland; emails: gysi@google.com, christoph.mueller, thorj@inf.ethz.ch; O. Zinenko, Google, France; email: zinenko@google.com; S. Herhut, Google, Germany; email: herhut@google.com; E. Davis, T. Wicky, and O. Fuhrer, Vulcan Inc, USA; emails: (EddieD, TobiasW, OliverF)@vulcan.com; T. Grosse, University of Edinburgh, UK; email: tobias.grosser@ed.ac.uk.



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

© 2021 Copyright held by the owner/authors.
1544-3566/2021/99-ARTS3 \$15.00
<https://doi.org/10.1145/3469030>

ACM Transactions on Architecture and Code Optimization, Vol. 18, No. 4, Article 51. Publication date: September 2021.



<https://dl.acm.org/doi/pdf/10.1145/3469030>

Why Bother? Climate Model Domain Compilers

Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation

TOBIAS GYSI and CHRISTOPH MÜLLER, ETH Zurich, Switzerland
OLEKSANDR ZINENKO, Google, France
STEPHAN HERHUT, Google, Germany
EDDIE DAVIS, TOBIAS WICKY, and OLIVER FUHRER, Vulcan Inc, USA
TORSTEN HOEFLE, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, UK

Most compilers have a single core intermediate representation (IR) (e.g., LLVM) sometimes complemented with vaguely defined IR-like data structures. This IR is commonly low-level and close to machine instructions. As a result, optimizations relying on domain-specific information are either not possible or require complex analysis to recover the missing information. In contrast, multi-level rewriting instantiates a hierarchy of dialects (IRs), lowers programs level-by-level, and performs code transformations at the most suitable level. We demonstrate the effectiveness of this approach for the weather and climate domain. In particular, we develop a prototype compiler and design stencil- and GPU-specific dialects based on a set of newly introduced design principles. We find that two domain-specific optimizations (500 lines of code) realized on top of LLVM's extensible MLIR compiler infrastructure suffice to outperform state-of-the-art solutions. In essence, multi-level rewriting promises to herald the age of specialized compilers composed from domain- and target-specific dialects implemented on top of a shared infrastructure.

CCS Concepts: • Software and its engineering → Compilers; Domain specific languages; • Applied computing → Earth and atmospheric sciences;

Additional Key Words and Phrases: Weather and climate, stencil computations, intermediate representations

Tobias Gysi also with Google.
Tobias Gysi also with ETH Zurich.
The work done at ETH Zurich has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2148016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs.
Authors' addresses: T. Gysi, C. Müller, and T. Hoefler, ETH Zurich, Switzerland; emails: gysi@google.com, [christoph.mueller, ttor]@inf.ethz.ch; O. Zinenko, Google, France; email: zinenko@google.com; S. Herhut, Google, Germany; email: herhut@google.com; E. Davis, T. Wicky, and O. Fuhrer, Vulcan Inc, USA; emails: [EddieD, TobiasW, OliverF]@vulcan.com; T. Grosser, University of Edinburgh, UK; email: tobias.grosser@ed.ac.uk.

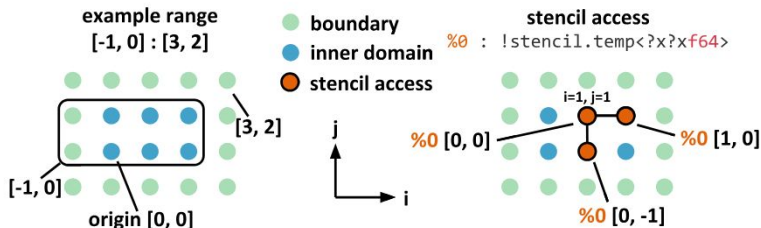


This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/author(s).
1544-3566/2021/99-ARTS1 \$15.00
<https://doi.org/10.1145/3469030>

ACM Transactions on Architecture and Code Optimization, Vol. 18, No. 4, Article 51. Publication date: September 2021.

```
func @sum(%in : !stencil.field<?x?x?xf64>, %out : !stencil.field<?x?x?xf64>) {  
  stencil.assert %in ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64> — define storage shapes  
  stencil.assert %out ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>  
  %0 = stencil.load %in : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>  
  %1 = stencil.apply (%arg0 = %0 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {  
    %2 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64  
    %3 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64  
    %4 = addf %2, %3 : f64 — stencil operator  
    stencil.return %4 : f64  
  }  
  stencil.store %1 to %out ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>  
  return — define output domain  
}
```



<https://dl.acm.org/doi/pdf/10.1145/3469030>

Why Bother? Climate Model Domain Compilers

Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-accelerated Climate Simulation

TOBIAS GYSI and CHRISTOPH MÜLLER, ETH Zurich, Switzerland
OLEKSANDR ZINENKO, Google, France
STEPHAN HERHUT, Google, Germany
EDDIE DAVIS, TOBIAS WICKY, and OLIVER FUHRER, Vulcan Inc, USA
TORSTEN HOEFELER, ETH Zurich, Switzerland
TOBIAS GROSSER, University of Edinburgh, UK

Most compilers have a single core intermediate representation (IR) (e.g., LLVM) sometimes complemented with vaguely defined IR-like data structures. This IR is commonly low-level and close to machine instructions. As a result, optimizations relying on domain-specific information are either not possible or require complex analysis to recover the missing information. In contrast, multi-level rewriting instantiates a hierarchy of dialects (IRs), lowers programs level-by-level, and performs code transformations at the most suitable level. We demonstrate the effectiveness of this approach for the weather and climate domain. In particular, we develop a prototype compiler and design stencil- and GPU-specific dialects based on a set of newly introduced design principles. We find that two domain-specific optimizations (500 lines of code) realized on top of LLVM's extensible MLIR compiler infrastructure suffice to outperform state-of-the-art solutions. In essence, multi-level rewriting promises to herald the age of specialized compilers composed from domain- and target-specific dialects implemented on top of a shared infrastructure.

CCS Concepts: • Software and its engineering → Compilers; Domain specific languages; • Applied computing → Earth and atmospheric sciences;

Additional Key Words and Phrases: Weather and climate, stencil computations, intermediate representations

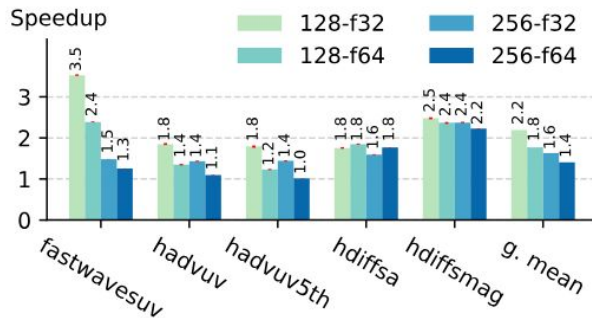
Tobias Gysi also with Google.
Tobias Grosse also with ETH Zurich.
The work done at ETH Zurich has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880), the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016), and ARM Holdings plc and Xilinx Inc in the context of Polly Labs.
Authors' addresses: T. Gysi, C. Müller, and T. Hoefler, ETH Zurich, Switzerland; emails: gysi@google.com, christoph.mueller, thorj@inf.ethz.ch; O. Zinenko, Google, France; email: zinenko@google.com; S. Herhut, Google, Germany; email: herhut@google.com; E. Davis, T. Wicky, and O. Fuhrer, Vulcan Inc, USA; emails: (Eddie), TobiasW, OliverF@vulcan.com; T. Grosser, University of Edinburgh, UK; email: tobias.grosser@ed.ac.uk.



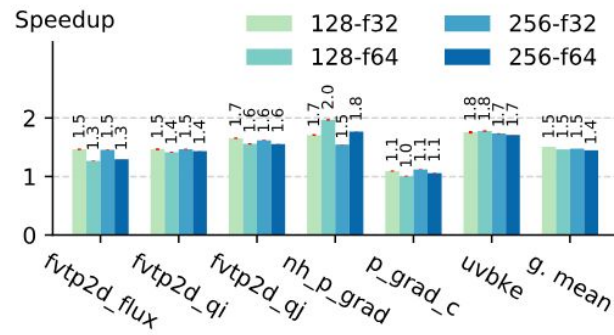
This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2021 Copyright held by the owner/authors.
1544-3566/2021/99-ARTS1 \$15.00
<https://doi.org/10.1145/3469030>

ACM Transactions on Architecture and Code Optimization, Vol. 18, No. 4, Article 51. Publication date: September 2021.



Over STELLA (COSMO)



Over Dawn (FV3)

On V100-SXM2

<https://dl.acm.org/doi/pdf/10.1145/3469030>

Why Bother? CFD Domain Optimization

Code Generation for In-Place Stencils

Mohamed Essadki
ONERA
Chailion, France
mohamed.essadki@onera.fr

Bertrand Michel
ONERA
Chailion, France
bertrand.michel@onera.fr

Bruno Maugars
ONERA
Chailion, France
bruno.maugars@onera.fr

Oleksandr Zinenko
Google
Paris, France
zinenko@google.com

Nicolas Vasilache
Google
Zürich, Switzerland
nv@google.com

Albert Cohen
Google
Paris, France
albertcohen@google.com

Abstract

Numerical simulation often resorts to iterative in-place stencils such as the Gauss-Seidel or Successive Overrelaxation (SOR) methods. Writing high performance implementations of such stencils requires significant effort and time; it also involves non-local transformations beyond the stencil kernel itself. While automated code generation is a mature technology for image processing stencils, convolutions and out-of-place iterative stencils (such as the Jacobi method), the optimization of in-place stencils requires manual craftsmanship. Building on recent advances in tensor compiler construction, we propose the first domain-specific code generator for iterative in-place stencils. Starting from a generic tensor compiler implemented in the MLIR framework, tensor abstractions are incrementally refined and lowered down to parallel, tiled, fused and vectorized code. We used our generator to implement a realistic, implicit solver for structured meshes, and demonstrate results competitive with an industrial computational fluid dynamics framework. We also compare with stand-alone stencil kernels for dense tensors.

CCS Concepts: Software and its engineering → Compilers - Theory of computation → Parallel computing models - Applied computing → Physical sciences and engineering.

Keywords: Computational Fluid Dynamics, Implicit Methods, Gauss-Seidel, SOR, Iterative In-Place Stencils, Domain-Specific Code Generation, MLIR, Vectorization, Tiling.

ACM Reference Format

Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. 2023. Code Generation for In-Place Stencils. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23)*.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner(s).

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada

© 2023 Copyright held by the owner(s).

ACM ISBN 978-1-60959-338-0/23/02.

<https://doi.org/10.1145/357990.358006>

February 25 – March 1, 2023, Montréal, QC, Canada. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/357990.358006>

1 Introduction

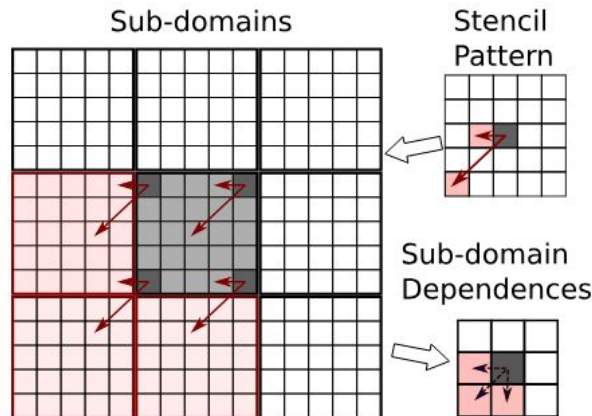
We are interested in the parallelization and optimization of Computational Fluid Dynamics (CFD) applications, and more specifically implicit finite-volume numerical methods to solve differential equations. This consists in discretizing the space domain into small cells representing the conservative fields of the simulation (mass density, momentum, energy, etc., where the volume value of each field is averaged over a given cell). Then, at every step of the simulation, a solver based on the implicit method (for faster convergence and scalability) proceeds by rewriting the differential equations in the form of a large and sparse linear system

$$A \cdot x = b \quad (1)$$

where A is a square matrix of size $m \times m$, x and b are two column vectors of the same size m , and x contains the numerical solution of the physical fields. An implicit CFD solver can typically be split in two main phases:

1. first compute the vector b , iterating over the faces of the cells to compute a numerical flux [34, 36] which can be considered as a function of the two solutions in adjacent cells separated by a common face;
2. then, rather than explicitly updating the fields in the cell, solve the linear system using an iterative method like Successive Overrelaxation (SOR), a variant of the Gauss-Seidel method [12, 42].

It remains an open problem to design and implement a domain-specific code generator for implicit finite-volume solvers using state-of-the-art methods like SOR. Unlike the Jacobi iterative method and all stencil codes occurring in image processing and neural networks, SOR is an in-place stencil computation, carrying an internal data dependence over the space domain. Because of these internal dependencies, parallelization and vectorization of in-place stencils require a wavefront schedule. This incurs additional control flow and indexing overheads and higher complexity in modeling locality-enhancing transformations such as tiling (cache blocking) and fusion. It is important to optimize such in-place stencils, since in typical scenarios Gauss-Seidel and



<https://research.google/pubs/code-generation-for-data-dependent-stencils/>

Why Bother? CFD Domain Optimization

Code Generation for In-Place Stencils

Mohamed Essadki
ONERA
Chailion, France
mohamed.essadki@onera.fr

Bertrand Michel
ONERA
Chailion, France
bertrand.michel@onera.fr

Bruno Maugars
ONERA
Chailion, France
bruno.maugars@onera.fr

Oleksandr Zinenko
Google
Paris, France
zinenko@google.com

Nicolas Vasilache
Google
Zürich, Switzerland
nv@google.com

Albert Cohen
Google
Paris, France
albertcohen@google.com

Abstract

Numerical simulation often resorts to iterative in-place stencils such as the Gauss-Seidel or Successive Overrelaxation (SOR) methods. Writing high performance implementations of such stencils requires significant effort and time; it also involves non-local transformations beyond the stencil kernel itself. While automated code generation is a mature technology for image processing stencils, convolutions and out-of-place iterative stencils (such as the Jacobi method), the optimization of in-place stencils requires manual craftsmanship. Building on recent advances in tensor compiler construction, we propose the first domain-specific code generator for iterative in-place stencils. Starting from a generic tensor compiler implemented in the MLIR framework, tensor abstractions are incrementally refined and lowered down to parallel, tiled, fused and vectorized code. We used our generator to implement a realistic, implicit solver for structured meshes, and demonstrate results competitive with an industrial computational fluid dynamics framework. We also compare with stand-alone stencil kernels for dense tensors.

CCS Concepts: Software and its engineering → Compilers - Theory of computation → Parallel computing models; Applied computing → Physical sciences and engineering.

Keywords: Computational Fluid Dynamics, Implicit Methods, Gauss-Seidel, SOR, Iterative In-place Stencils, Domain-Specific Code Generation, MLIR, Vectorization, Tiling.

ACM Reference Format

Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. 2023. Code Generation for In-Place Stencils. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23)*.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner(s).

CGO '23, February 25 – March 1, 2023, Montreal, QC, Canada.

© 2023 Copyright held by the owner(s).

ACM ISBN 978-1-145-35799-0/23/0006.

<https://doi.org/10.1145/357990.358006>

February 25 – March 1, 2023, Montreal, QC, Canada. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/357990.358006>

1 Introduction

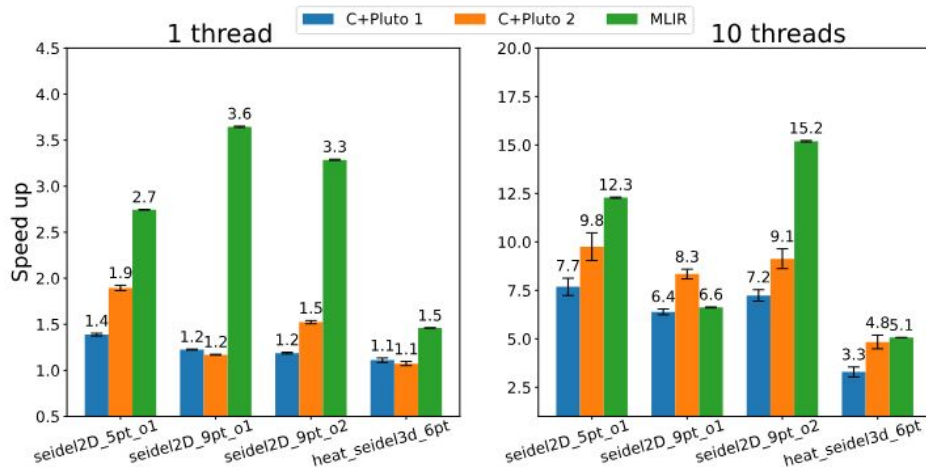
We are interested in the parallelization and optimization of Computational Fluid Dynamics (CFD) applications, and more specifically implicit finite-volume numerical methods to solve differential equations. This consists in discretizing the space domain into small cells representing the conservative fields of the simulation (mass density, momentum, energy, etc., where the volume value of each field is averaged over a given cell). Then, at every step of the simulation, a solver based on the implicit method (for faster convergence and scalability) proceeds by rewriting the differential equations in the form of a large and sparse linear system

$$A \cdot x = b \quad (1)$$

where A is a square matrix of size $m \times m$, x and b are two column vectors of the same size m , and x contains the numerical solution of the physical fields. An implicit CFD solver can typically be split in two main phases:

1. first compute the vector b , iterating over the faces of the cells to compute a numerical flux [34, 36] which can be considered as a function of the two solutions in adjacent cells separated by a common face;
2. then, rather than explicitly updating the fields in the cell, solve the linear system using an iterative method like Successive Overrelaxation (SOR), a variant of the Gauss-Seidel method [12, 42].

It remains an open problem to design and implement a domain-specific code generator for implicit finite-volume solvers using state-of-the-art methods like SOR. Unlike the Jacobi iterative method and all stencil codes occurring in image processing and neural networks, SOR is an in-place stencil computation, carrying an internal data dependence over the space domain. Because of these internal dependencies, parallelization and vectorization of in-place stencils require a wavefront schedule. This incurs additional control flow and indexing overheads and higher complexity in modeling locality-enhancing transformations such as tiling (cache blocking) and fusion. It is important to optimize such in-place stencils, since in typical scenarios Gauss-Seidel and



Speedup over Pluto compiler
on 2x Intel Xeon 6152 CPUs (NUMA)

<https://research.google/pubs/code-generation-for-data-dependent-stencils/>

Why Bother? CFD Domain Optimization

Code Generation for In-Place Stencils

Mohamed Essadki
ONERA
Châtillon, France
mohamed.essadki@onera.fr

Bertrand Michel
ONERA
Châtillon, France
bertrand.michel@onera.fr

Bruno Maugars
ONERA
Châtillon, France
bruno.maugars@onera.fr

Oleksandr Zinenko
Google
Paris, France
zinenko@google.com

Nicolas Vasilache
Google
Zürich, Switzerland
nv@google.com

Albert Cohen
Google
Paris, France
albertcohen@google.com

Abstract

Numerical simulation often resorts to iterative in-place stencils such as the Gauss-Seidel or Successive Overrelaxation (SOR) methods. Writing high performance implementations of such stencils requires significant effort and time; it also involves non-local transformations beyond the stencil kernel itself. While automated code generation is a mature technology for image processing stencils, convolutions and out-of-place iterative stencils (such as the Jacobi method), the optimization of in-place stencils requires manual craftsmanship. Building on recent advances in tensor compiler construction, we propose the first domain-specific code generator for iterative in-place stencils. Starting from a generic tensor compiler implemented in the MLIR framework, tensor abstractions are incrementally refined and lowered down to parallel, tiled, fused and vectorized code. We used our generator to implement a realistic, implicit solver for structured meshes, and demonstrate results competitive with an industrial computational fluid dynamics framework. We also compare with stand-alone stencil kernels for dense tensors.

CCS Concepts: Software and its engineering → Compilers - Theory of computation → Parallel computing models - Applied computing → Physical sciences and engineering.

Keywords: Computational Fluid Dynamics, Implicit Methods, Gauss-Seidel, SOR, Iterative In-Place Stencils, Domain-Specific Code Generation, MLIR, Vectorization, Tiling.

ACM Reference Format:

Mohamed Essadki, Bertrand Michel, Bruno Maugars, Oleksandr Zinenko, Nicolas Vasilache, and Albert Cohen. 2023. Code Generation for In-Place Stencils. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization (CGO '23)*.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner(s).

CGO '23, February 25 – March 1, 2023, Montréal, QC, Canada.

© 2023 Copyright held by the owner(s).

ACM ISBN 978-1-60959-338-0/23/02.

<https://doi.org/10.1145/357990.358006>

February 25 – March 1, 2023, Montréal, QC, Canada. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/357990.358006>

1 Introduction

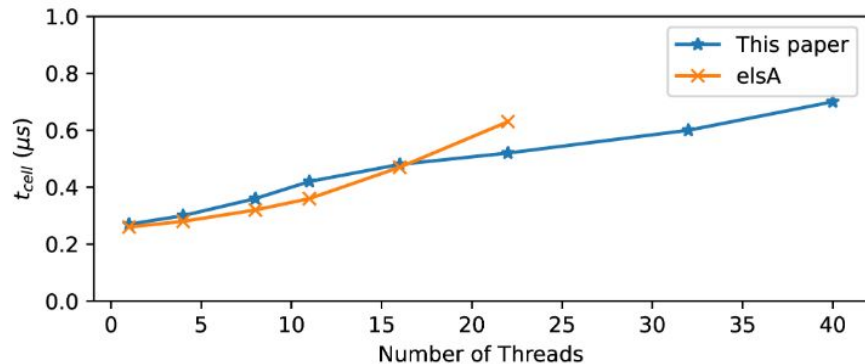
We are interested in the parallelization and optimization of Computational Fluid Dynamics (CFD) applications, and more specifically implicit finite-volume numerical methods to solve differential equations. This consists in discretizing the space domain into small cells representing the conservative fields of the simulation (mass density, momentum, energy, etc., where the volume value of each field is averaged over a given cell). Then, at every step of the simulation, a solver based on the implicit method (for faster convergence and scalability) proceeds by rewriting the differential equations in the form of a large and sparse linear system

$$A \cdot x = b \quad (1)$$

where A is a square matrix of size $m \times m$, x and b are two column vectors of the same size m , and x contains the numerical solution of the physical fields. An implicit CFD solver can typically be split in two main phases:

1. first compute the vector b , iterating over the faces of the cells to compute a numerical flux [34, 36] which can be considered as a function of the two solutions in adjacent cells separated by a common face;
2. then, rather than explicitly updating the fields in the cell, solve the linear system using an iterative method like Successive Overrelaxation (SOR), a variant of the Gauss-Seidel method [12, 42].

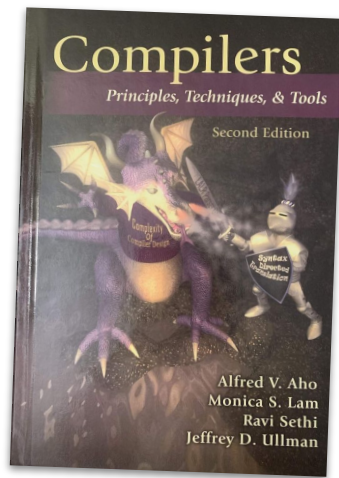
It remains an open problem to design and implement a domain-specific code generator for implicit finite-volume solvers using state-of-the-art methods like SOR. Unlike the Jacobi iterative method and all stencil codes occurring in image processing and neural networks, SOR is an in-place stencil computation, carrying an internal data dependence over the space domain. Because of these internal dependencies, parallelization and vectorization of in-place stencils require a wavefront schedule. This incurs additional control flow and indexing overheads and higher complexity in modeling locality-enhancing transformations such as tiling (cache blocking) and fusion. It is important to optimize such in-place stencils, since in typical scenarios Gauss-Seidel and



Comparable to elsA (ONERA)
Keeps using memory instead of MPI

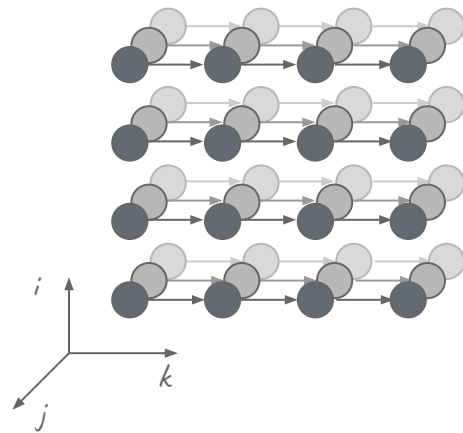
<https://research.google/pubs/code-generation-for-data-dependent-stencils/>

Why Bother? Programs as Mathematical Objects



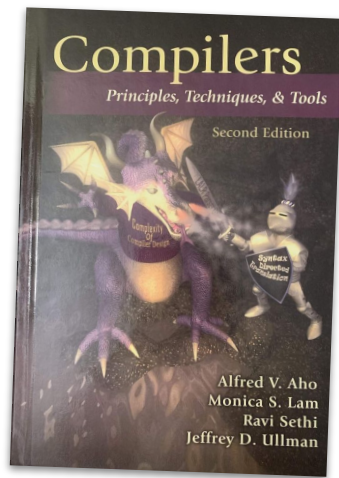
* textbook material

```
for i in range(N):  
  for j in range(M):  
    for k in range(P):  
      C[i][j] += A[i][k] + B[k][j]
```



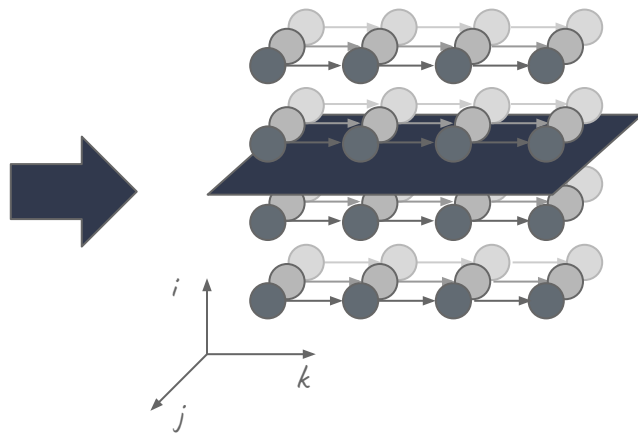
$$\{(i, j, k) : i, j, k \in \mathbb{Z}, 0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq k < P\}$$

Why Bother? Programs as Mathematical Objects



* textbook material

```
for i in range(N):  
  for j in range(M):  
    for k in range(P):  
      C[i][j] += A[i][k] + B[k][j]
```



$$\{(i, j, k) : i, j, k \in \mathbb{Z}, 0 \leq i < N \wedge 0 \leq j < M \wedge 0 \leq k < P\}$$

Why Bother? Programs as Mathematical Objects

Polygeist: Raising C to Polyhedral MLIR

William S. Moses*
MIT CSAIL
Cambridge, MA, USA
wsmoses@mit.edu

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

Ruijie Zhao*
Imperial College London
London, UK
ruijie.zhao15@imperial.ac.uk

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

Abstract—We present Polygeist, a new compilation flow that connects the MLIR compiler infrastructure to existing edge polyhedral optimization tools. It consists of a C and C++ frontend capable of converting a broad range of existing codes into MLIR suitable for polyhedral transformation and a bi-directional conversion between MLIR and OpenSCoP exchange format. The Polygeist/MLIR intermediate representation featuring high-level (affine) loop constructs and n-D arrays embedded into a single static assignment (SSA) substrate enables an unprecedented combination of SSA-based and polyhedral optimizations. We illustrate this by proposing and implementing two extra transformations: statement splitting and reduction parallelization. Our evaluation demonstrates that Polygeist outperforms an average both an LLVM IR-level optimizer (Poly) and a source-to-source state-of-the-art polyhedral compiler (Pluto) when exercised on the Polybench¹ benchmark suite in sequential (2.5x vs 1.4x, 2.34x) and parallel mode (9.47x vs 1.26x, 7.54x) thanks to the new representation and transformations.

1. INTRODUCTION

Improving the efficiency of computation has always been one of the prime goals of computing. Program performance can be improved significantly by reaping the benefits of parallelism, temporal and spatial locality, and other performance sources. Relevant program transformations are particularly useful and challenging when targeting modern multicore CPUs and GPUs with deep memory hierarchies and parallelism, and are often performed automatically by optimizing compilers.

The polyhedral model enables precise analyses and a relatively easy specification of transformations (loop restructuring, automatic parallelization, etc.) that take advantage of hardware performance sources. As a result, there is growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of compute-intensive programs [1], [2], [3], and for programming accelerator architectures [4], [5], [6]. Consequently, the compiler community has focused on building tools that identify and optimize parts of the program that can be represented within the polyhedral model (commonly referred to as static-control parts, or SCoPs). Such tools tend to fall into two categories.

Compiler-based tools like Poly [7] and Graphite [8] detect and transform SCoPs in compiler intermediate representations (IRs). While this offers seamless integration with rest of the compiler, the lack of high-level structure and information hinders the tool's ability to perform analyses and transformations. This structure needs to be recovered from optimized IR, often

imperfectly or at a significant cost [8]. Moreover, common compiler optimizations such as LCM may interfere with the process [10]. Finally, low-level IRs often lack constructs for, e.g., parallelism or reductions, produced by the transformation, which makes the flow more complex.

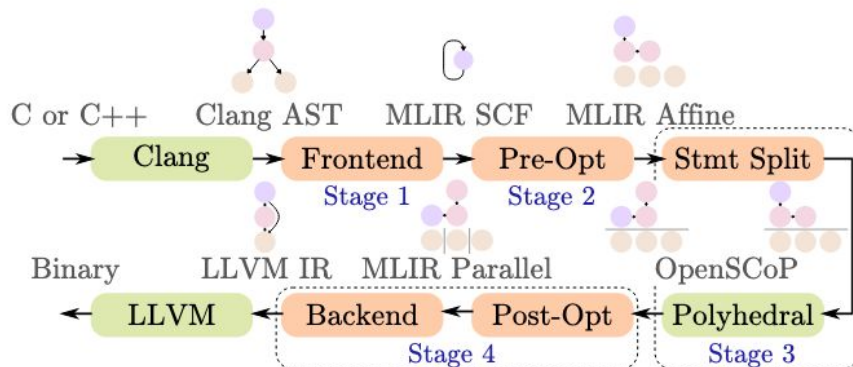
Source-to-source compilers such as Pluto [11], PoCC [12] and rrc [5] operate directly on C or C++ code. While this can effectively leverage the high-level information from source code, the effectiveness of such tools is often reduced by the lack of enabling optimizations such as those converting hazardous memory loads into single-assignment virtual registers. Furthermore, the transformation results must be expressed in C, which is known to be complex [13], [14] and is also missing constructs for, e.g., reduction loops or register values not backed by memory storage.

This paper proposes and evaluates the benefits of a polyhedral compilation flow, Polygeist (Figure 1), that can leverage both the high-level structure available in source code and the fine-grained control of compiler optimization provided by low-level IRs. It builds on the recent MLIR compiler infrastructure that allows the interplay of multiple abstraction levels within the same representation, during the same transformations [15]. Intermediate MLIR abstractions, or *dialects*, include high-level constructs such as loops, parallel and reduction patterns; low-level representations fully covering LLVM IR [16]; and a polyhedral-inspired representation featuring loops and memory accesses annotated with affine expressions. Moreover, by combining the best of source-level and IR-level tools in an end-to-end polyhedral flow, Polygeist preserves high-level information and leverages them to perform new or improved

* Equal contribution.



Fig. 1. The Polygeist compilation flow consists of 4 stages. The forward (green) Clang AST to emit MLIR SCF dialect (Section II-B), which is then processed by a polyhedral scheduler (Section II-C) before post-optimization and parallelization (Section II-D). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.



Connecting C and C++, and any MLIR loops, to pre-existing Polyhedral optimization tools

<https://ieeexplore.ieee.org/abstract/document/9563011>

Why Bother? Programs as Mathematical Objects

Polygeist: Raising C to Polyhedral MLIR

William S. Moses*
MIT CSAIL
Cambridge, MA, USA
wsmoses@mit.edu

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

Ruijie Zhao*
Imperial College London
London, UK
ruijie.zhao15@imperial.ac.uk

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

Abstract—We present Polygeist, a new compilation flow that connects the MLIR compiler infrastructure to cutting-edge polyhedral optimization tools. It consists of a C and C++ frontend capable of converting a broad range of existing codes into MLIR suitable for polyhedral transformation and a bi-directional conversion between MLIR and OpenMP exchange format. The Polygeist/MLIR intermediate representation featuring high-level (affine) loop constructs and n-D arrays embedded into a single static assignment (SSA) substrate enables an unprecedented combination of SSA-based and polyhedral optimizations. We illustrate this by proposing and implementing two extra transformations: statement splitting and reduction parallelization. Our evaluation demonstrates that Polygeist outperforms on average both an LLVM IR-level optimizer (Polly) and a source-to-source state-of-the-art polyhedral compiler (Pluto) when exercised on the Polybench¹ benchmark suite in sequential (2.5x vs 1.4x), the Polybench² benchmark suite in sequential (2.5x vs 1.4x), and parallel mode (9.4x vs 3.2x, 7.5x) thanks to the new representation and transformations.

1. INTRODUCTION

Improving the efficiency of computation has always been one of the prime goals of computing. Program performance can be improved significantly by tapping the benefits of parallelism, temporal and spatial locality, and other performance sources. Relevant program transformations are particularly tedious and challenging when targeting modern multicore CPUs and GPUs with deep memory hierarchies and parallelism, and are often performed automatically by optimizing compilers.

The polyhedral model enables precise analyses and a relatively easy specification of transformations (loop restructuring, automatic parallelization, etc.) that take advantage of hardware performance sources. As a result, there is growing evidence that the polyhedral model is one of the best frameworks for efficient transformation of compute-intensive programs [1], [2], [3], and for programming accelerator architectures [4], [5], [6]. Consequently, the compiler community has focused on building tools that identify and optimize parts of the program that can be represented within the polyhedral model (commonly referred to as static-control parts, or SCAPs). Such tools tend to fall into two categories.

Compiler-based tools like Polly [7] and Graphite [8] detect and transform SCAPs in compiler intermediate representations (IRs). While this offers seamless integration with rest of the compiler, the lack of high-level structure and information hinders the tool's ability to perform analyses and transformations. This structure needs to be recovered from optimized IR, often

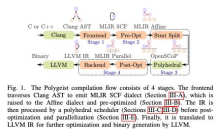
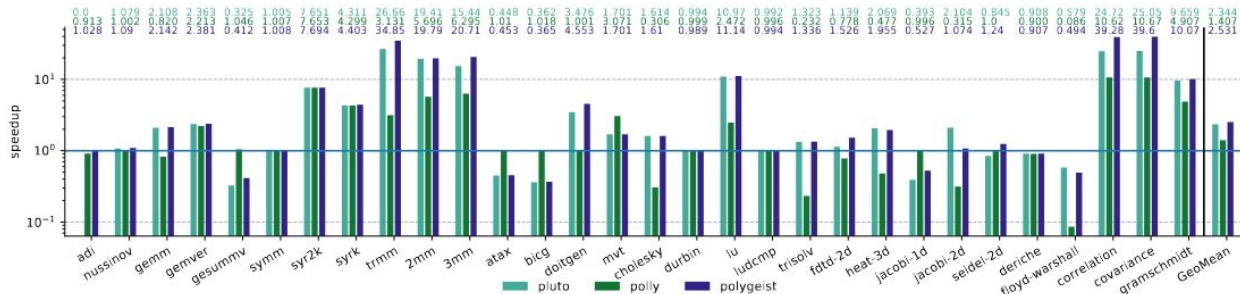


Fig. 1. The Polygeist compilation flow consists of a Clang AST, the lowered LLVM IR, the lowered MLIR IR, and the lowered Polygeist IR. The IR is then processed by a polyhedral scheduler (Polygeist-IR) before post-optimization and parallelization (Section III-D). Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.

imperfectly or at a significant cost [8]. Moreover, common compiler optimizations such as LCM may interfere with the process [10]. Finally, low-level IRs often lack constructs for, e.g., parallelism or reductions, produced by the transformation, which makes the flow more complex.

Source-to-source compilers such as Pluto [11], PoCC [12] and rcc [13] operate directly on C or C++ code. While this can effectively leverage the high-level information from source code, the effectiveness of such tools is often reduced by the lack of enabling optimizations such as those converting hardware memory loads into single-assignment virtual registers. Furthermore, the transformation results must be expressed in C, which is known to be complex [13], [14] and is also missing constructs for, e.g., reduction loops or register values not backed by memory storage.

This paper proposes and evaluates the benefits of a polyhedral compilation flow, Polygeist (Figure 1), that can leverage both the high-level structure available in source code and the fine-grained control of compiler optimization provided by low-level IRs. It builds on the recent MLIR compiler infrastructure that allows the interplay of multiple abstraction levels within the same representation, during the same transformations [15]. Intermediate MLIR abstractions, or *dialects*, include high-level constructs such as loops, parallel and reduction patterns; low-level representations fully covering LLVM IR [16]; and a polyhedral-inspired representation featuring loops and memory accesses annotated with affine expressions. Moreover, by combining the best of source-level and IR-level tools in an end-to-end polyhedral flow, Polygeist preserves high-level information and leverages them to perform new or improved



Faster than pre-existing Polyhedral optimization tools
thanks to higher-level abstraction

<https://ieeexplore.ieee.org/abstract/document/9563011>

* Equal contribution.

Why Bother? Raise the Abstraction Level

Progressive Raising in Multi-level IR

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

Aodi Drebes
Google
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-optimized 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 IR 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 general-purpose language to high-level linear algebra operations. Our raising paths leverage subsequent high-level domain-specific transformations with significant performance improvements.

Index Terms—MLIR, progressive raising, multi-level intermediate representations

1. INTRODUCTION

The increasing complexity of hardware resulting from the ongoing trend for heterogeneous systems has made it difficult for general-purpose compilers to generate efficient code automatically [1]. One of the main issues is the mismatch between the low level of abstraction at which general-purpose compilers operate and the various high-level abstractions for computation required by today's applications [2]. Although high-level programming languages allow for the specification of high-level operations, this information is often not captured by the low-level intermediate representation (IR) of general-purpose compilers or lost early in the compilation process during lowering [3].

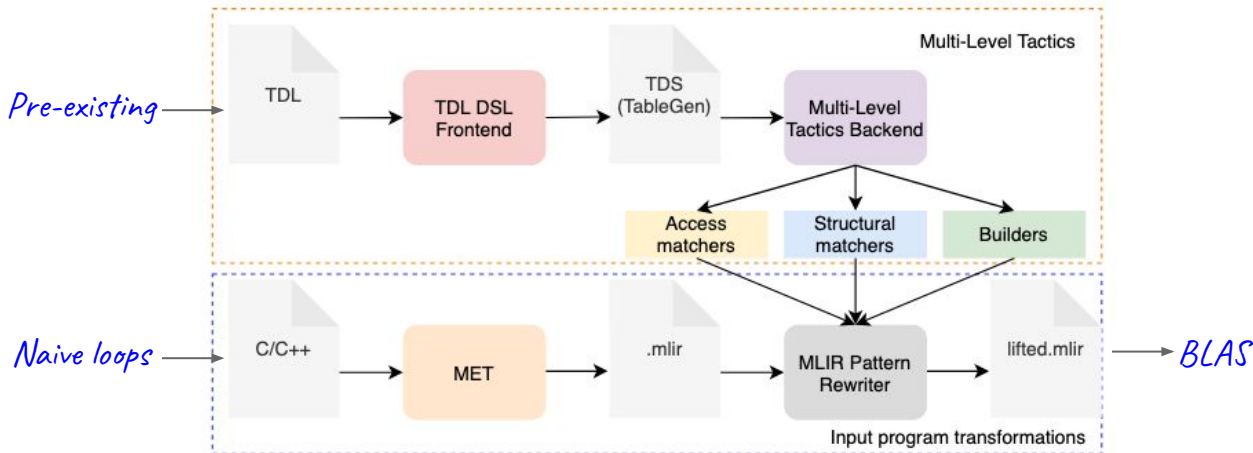
Domain-specific languages (DSLs) and compilers attempt to capture and explicitly preserve high-level information throughout the compilation process and have been employed successfully to generate efficient code for modern hardware [4], [5]. However, such languages commit to a limited set of isolated

abstractions and domain-specific optimizations, resulting in poor interoperability, limited reusability of software components, and few opportunities for inter-domain optimizations [6].

Multi-level intermediate representations explicitly allow for the co-existence of multiple abstractions within the same compilation framework with interoperable representations, breaking the isolation between domains and enabling comprehensive optimizations. During compilation, the source program's high-level representation is progressively optimized and transformed to lower-level abstractions, until reaching a low-level, general-purpose representation for code generation [7].

Multi-level frameworks solve many issues of DSLs, but the optimizations in progressive lowering compilation scheme crucially rely on the adequate initial representation of the source program. If the initial representation is below the required level of abstraction for a given optimization, the optimization simply fails to apply. However, providing an adequate high-level input representation may not always be possible. General-purpose languages not being semantically rich enough to preserve the right level of information enter the lowering pipeline at a very low level, thus precluding most, if not all, domain-specific

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



Why Bother? Raise the Abstraction Level

Progressive Raising in Multi-level IR

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

Aodi Drebes
TU Eindhoven
Paris, France
andi@programmierforum.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-optimized 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 IR 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 general-purpose language to high-level linear algebra operations. Our raising paths leverage subsequent high-level domain-specific transformations with significant performance improvements.

Index Terms—MLIR, progressive raising, multi-level intermediate representations

1. INTRODUCTION

The increasing complexity of hardware resulting from the ongoing trend for heterogeneous systems has made it difficult for general-purpose compilers to generate efficient code automatically [1]. One of the main issues is the mismatch between the low level of abstraction at which general-purpose compilers operate and the various high-level abstractions for computation required by today's applications [2]. Although high-level programming languages allow for the specification of high-level operations, this information is often not captured by the low-level intermediate representation (IR) of general-purpose compilers or lost early in the compilation process during lowering [3].

Domain-specific languages (DSLs) and compilers attempt to capture and explicitly preserve high-level information throughout the compilation process and have been employed successfully to generate efficient code for modern hardware [4], [5]. However, such languages commit to a limited set of isolated

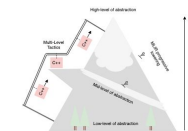
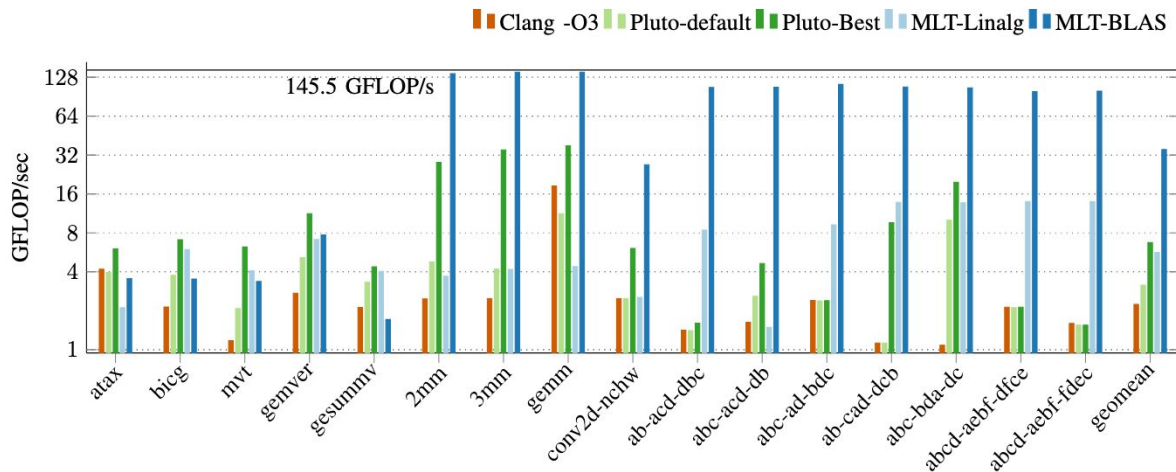


Fig. 1: Multi-Level Tactics lifts general-purpose languages to higher-abstraction levels to enable effective domain-specific compilation via progressive lowering.

abstractions and domain-specific optimizations, resulting in poor interoperability, limited reusability of software components, and few opportunities for inter-domain optimizations [6].

Multi-level intermediate representations explicitly allow for the co-existence of multiple abstractions within the same compilation framework with interoperable representations, breaking the isolation between domains and enabling comprehensive optimizations. During compilation, the source program's high-level representation is progressively optimized and transformed to lower-level abstractions, until reaching a low-level, general-purpose representation for code generation [7].

Multi-level frameworks solve many issues of DSLs, but the optimizations in progressive lowering compilation scheme crucially rely on the adequate initial representation of the source program. If the initial representation is below the required level of abstraction for a given optimization, the optimization simply fails to apply. However, providing an adequate high-level input representation may not always be possible. General-purpose languages not being semantically rich enough to preserve the right level of information enter the lowering pipeline at a very low level, thus precluding most, if not all, domain-specific



Can target BLAS, which is (obviously) faster than any automatically compiled code. On Intel i9-9900K.

<https://ieeexplore.ieee.org/abstract/document/9370332>

Why Bother? Modernize and Port Legacy Code

Retargeting and Respecializing GPU Workloads for Performance Portability

Ivan Ivanov
Tokyo Institute of Technology
RIKEN R-CCS
Kobe, Japan
ivanov.i.aa@riken.ac.jp

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

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

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

William S. Moses
University of Illinois Urbana-Champaign
Google DeepMind
Illinois, United States
wsmoses@illinois.edu

Abstract—In order to come close to peak performance, accelerators like GPUs require significant architecture-specific tuning that understands the availability of shared memory, parallelism, tensor cores, etc. Unfortunately, the pursuit of higher performance and lower costs have led to a significant diversification of architectures, even from the same vendor. This creates the need for performance portability across different GPUs, especially important for programs in a particular programming model with a certain architecture in mind. Even when the program can be seamlessly executed on a different architecture, it may suffer a performance penalty due to it not being sized appropriately to the available hardware resources such as fast memory and registers, let alone not using newer advanced features of the architecture.

We propose a new approach to improving performance of legacy CUDA programs for modern machines by automatically adjusting the amount of work each parallel thread does, and the amount of memory and register resources it requires. By operating within the MLIR compiler infrastructure, we are able to also target AMD GPUs by performing automatic translation from CUDA and simultaneously adjust the program granularity to fit the size of target GPUs.

Combined with autotuning assisted by the platform-specific compiler, our approach demonstrates 37% geometric speedup on the Rodinia benchmark suite over baseline CUDA implementations as well as performance parity between similar NVIDIA and AMD GPUs executing the same CUDA program.

1. INTRODUCTION

Accelerators like GPUs remain the hardware target of choice for performance-critical software. Achieving high performance on these accelerators requires programmers to effectively leverage a peculiar programming model, most often expressed as C++ language extensions such as CUDA for NVIDIA GPUs and ROCm for AMD. While the community has developed alternative methods to portably program GPUs, including a high-level block programming model in Triton [1], automatic mapping of C++ code onto GPUs [2], manually-writable abstractions with varying degree of automated scheduling in JAX [3], TC [4], and TVM [5], many of the performance-critical scientific

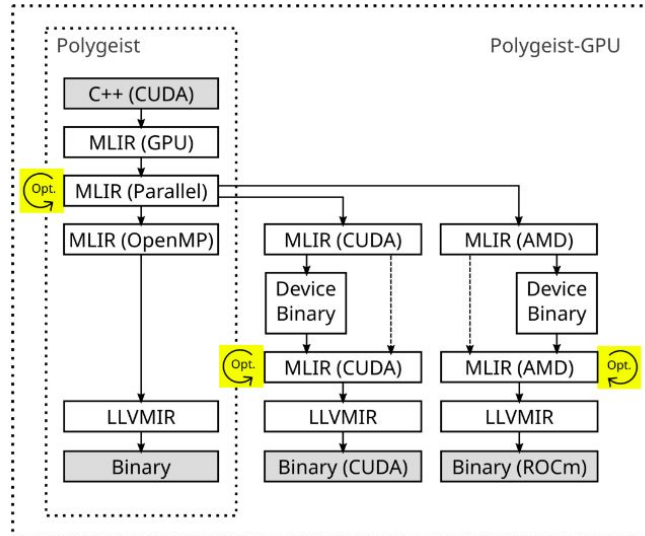
programs, including these very portability frameworks, remain written in CUDA [6].

While the CUDA programming model and syntax have remained relatively stable over time, the underlying GPU hardware has evolved significantly, adding many new features and instructions. For example, earlier versions of programmable NVIDIA GPUs used “full warps” of 16 threads for scheduling and had a limitation of 1024 threads running concurrently on a hardware unit while modern GPUs use “full warps” of 32 and allow up to 2048 threads per hardware unit. Similar changes can be observed in the amount of available low-latency memory and registers. This trend is even more important when considering GPUs of a different vendor, like AMD, which operate in “wavefronts” of 64 threads and allow up to 4096 threads per hardware unit.

Even when GPU kernels written in CUDA appear to run on newer NVIDIA GPUs, they may often fail to reach similar utilization as the kernels are incorrectly sized for the target architecture. However, this may be avoided through skilful use of the programming model by writing CUDA programs that adapt to different numbers of concurrent threads. But even programs with this flexibility do not permit control of the amount of allocated “shared” memory between several threads in a group or the amount of registers used (which is proportional to the number of threads). Both of these characteristics have a dramatic impact on the overall performance. These sizing problems are often amplified when porting a program to a GPU of a different vendor, let alone the often non-trivial engineering effort of porting itself.

In this paper, we propose a compile-based mechanism to “resize” GPU programs to a particular architecture. Taking existing CUDA code, our compiler can control the granularity of the program including the amount of work performed by

¹An type of various alternatives, like the BCC and SVCL [6] the CUDA framework, a power of the GPU programming model, is used in significantly more applications due to legacy maintenance, and network effects.



Why Bother? Modernize and Port Legacy Code

Retargeting and Respecializing GPU Workloads for Performance Portability

Ivan R. Ivanov
Tokyo Institute of Technology
RIKEN R-CCS
Kobe, Japan
ivanov.i.aa@riken.ac.jp

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

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

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

William S. Moses
University of Illinois Urbana-Champaign
Google DeepMind
Illinois, United States
wsamos@illinois.edu

Abstract—In order to come close to peak performance, accelerators the GPUs require significant architecture-specific tuning that understand the availability of shared memory, parallelism, tensor cores, etc. Unfortunately, the pursuit of higher performance and lower costs have led to a significant diversification of architectures designs, even from the same vendor. This creates the need for performance portability across different GPUs, especially important for programs in a particular programming model with a certain architecture in mind. Even when the program can be seamlessly executed on a different architecture, it may suffer a performance penalty due to it not being sized appropriately to the available hardware resources such as fast memory and registers, let alone not using newer advanced features of the architecture.

We propose a new approach to improving performance of legacy CUDA programs for modern machines by automatically adjusting the amount of work each parallel thread does, and the amount of memory and register resources it requires. By operating within the MLIR compiler infrastructure, we are able to also target AMD GPUs by performing automatic translation from CUDA and simultaneously adjust the program granularity to fit the size of target GPUs.

Combined with autotuning assisted by the platform-specific compiler, our approach demonstrates 27% geometric speedup on the Rodas benchmark suite over baseline CUDA implementations as well as performance parity between similar NVIDIA and AMD GPUs executing the same CUDA program.

1. INTRODUCTION

Accelerators like GPUs remain the hardware target of choice for performance-critical software. Achieving high performance on these accelerators requires programmers to effectively leverage a peculiar programming model, most often exposed as C++ language extensions such as CUDA for NVIDIA GPUs and ROCm for AMD. While the community has developed alternative methods to portably program GPUs, including a high-level block programming model in Triton [1], automatic mapping of C++ code onto GPUs [2], manually-written abstractions with varying degree of automated scheduling in JAX [3], TC [4], and TVM [5], many of the performance-critical scientific

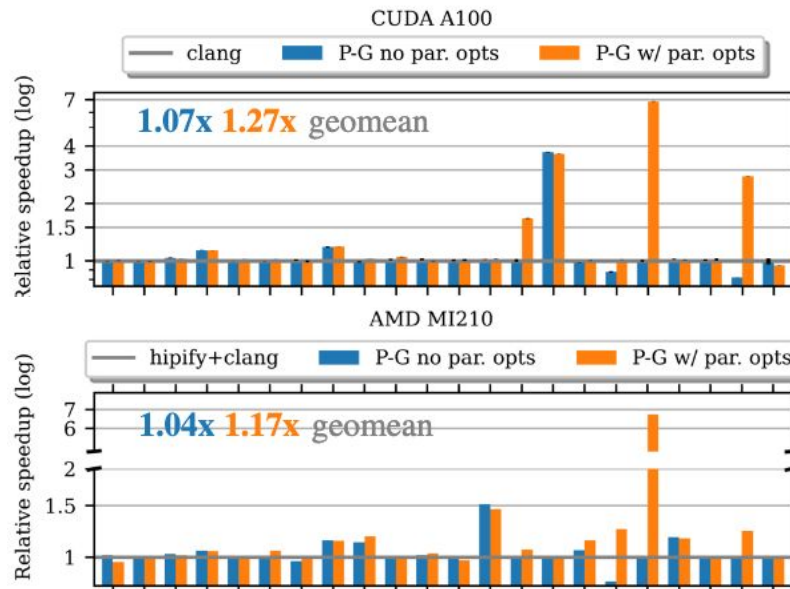
programs, including these very portability frameworks, remain written in CUDA[6].

While the CUDA programming model and syntax have remained relatively stable over time, the underlying GPU hardware has evolved significantly, adding many new features and instructions. For example, earlier versions of programmable NVIDIA GPUs used “full warps” of 16 threads for scheduling and had a limitation of 1024 threads running concurrently on a hardware unit while modern GPUs use “full warps” of 32 and allow up to 2048 threads per hardware unit. Similar changes can be observed in the amount of available low-latency memory and registers. This trend is even more important when considering GPUs of a different vendor, like AMD, which operate in “wavefronts” of 64 threads and allow up to 4096 threads per hardware unit.

Even when GPU kernels written in CUDA appear to run on newer NVIDIA GPUs, they may often fail to reach similar utilization as the kernels are incorrectly sized for the target architecture. However, this may be avoided through skilful use of the programming model by writing CUDA programs that adapt to different numbers of concurrent threads. But even programs with this flexibility do not permit control of the amount of allocated “shared” memory between several threads in a group or the amount of registers used (which is proportional to the number of threads). Both of these characteristics have a dramatic impact on the overall performance. These sizing problems are often amplified when porting a program to a GPU of a different vendor, let alone the often non-trivial engineering effort of porting itself.

In this paper, we propose a compile-based mechanism to “resize” GPU programs to a particular architecture. Taking existing CUDA code, our compiler can control the granularity of the program including the amount of work performed by


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



Older CUDA code is made faster (better shmem use)
And also runs on AMD transparently!

<https://ieeexplore.ieee.org/abstract/document/10444828>

How to Start Using It

 **MLIR**
Multi-Level IR Compiler Framework

Community > Debugging Tips FAQ Source > Bugs Logo Assets Youtube Channel

[Home](#)
[Governance](#)
[Users of MLIR](#)
[MLIR Related Publications](#)
[Talks](#)
[Deprecations & Current Refactoring](#)
[Getting Started](#) +
[Code Documentation](#) +

Getting Started

Don't miss the MLIR Tutorial! [slides](#) - [recording](#) - [online step-by-step](#)

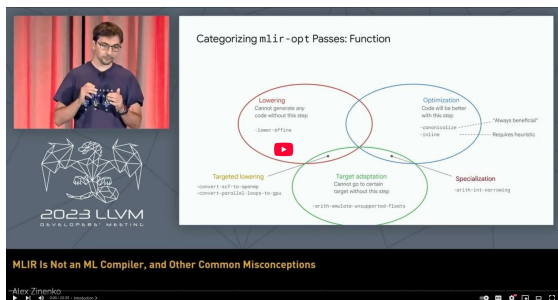
Please refer to the [LLVM Getting Started](#) in general to build LLVM. Below are quick instructions to build MLIR with LLVM.

The following instructions for compiling and testing MLIR assume that you have `git`, `ninja`, and a working C++ toolchain (see [LLVM requirements](#)).

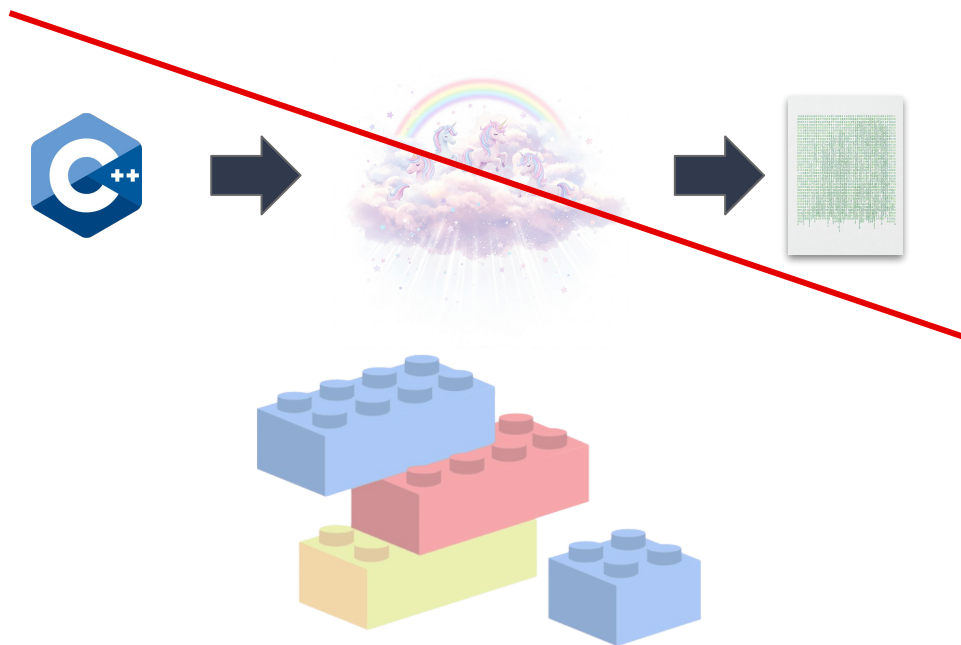
As a starter, you may try [the tutorial](#) on building a compiler for a Toy language.

https://mlir.llvm.org/getting_started/

How to Start Using It



<https://www.youtube.com/watch?v=IXAp6ZAWyBY>



Little Builtin, Everything Customizable

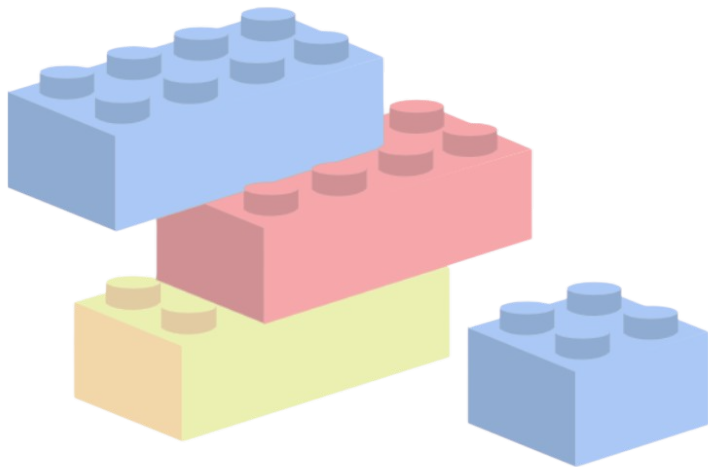
No fixed set of:

- Operations
- Attributes
- Types

Define your abstraction
Reuse existing when possible

Bring your own anything:

- As long as you define and verify semantics
- Group into “dialects”



How to Start Using It

<https://mlir.llvm.org/>

Contact these people

Code Generation for In-Place Stencils

Mehrdad Eshghi
ONERA
Chadler, France
mehrad.eshghi@onera.fr

Thomas Magagnoli
ONERA
Chadler, France
thomas.magagnoli@onera.fr

Nikolaus Vasilakos
ONERA
Chadler, France
nikolaus.vasilakos@onera.fr

Albert Cohen
ONERA
Chadler, France
albert.cohen@onera.fr

Abstract
Stencil computation often occurs in a pipeline where the first stage is a stencil computation on a 2D or 3D grid, followed by a reduction operation. This paper presents a new stencil computation framework that leverages the power of GPUs to accelerate the stencil computation. The framework is designed to be flexible and easy to use, and it can be integrated into existing stencil computation frameworks.

Progressive Raising in Multi-Level IR

Lorenzo Chiodi
ONERA
Chadler, France
lorenzo.chiodi@onera.fr

Albert Cohen
ONERA
Chadler, France
albert.cohen@onera.fr

Abstract
This paper presents a new progressive raising framework for multi-level IR. The framework is designed to be flexible and easy to use, and it can be integrated into existing multi-level IR frameworks.

Abstract Interpretation of Stencil Computations

Mehrdad Eshghi
ONERA
Chadler, France
mehrad.eshghi@onera.fr

Thomas Magagnoli
ONERA
Chadler, France
thomas.magagnoli@onera.fr

Nikolaus Vasilakos
ONERA
Chadler, France
nikolaus.vasilakos@onera.fr

Abstract
This paper presents a new abstract interpretation framework for stencil computations. The framework is designed to be flexible and easy to use, and it can be integrated into existing abstract interpretation frameworks.

Retargeting and Respecializing GPU Workloads for Performance Portability

Thomas Magagnoli
ONERA
Chadler, France
thomas.magagnoli@onera.fr

Nikolaus Vasilakos
ONERA
Chadler, France
nikolaus.vasilakos@onera.fr

Abstract
This paper presents a new retargeting and respecializing framework for GPU workloads. The framework is designed to be flexible and easy to use, and it can be integrated into existing GPU workload frameworks.

Polygeist: Raising C to Polyhedral MLIR

William S. Moses
MIT
Cambridge, MA
moses@mit.edu

Lorenzo Chiodi
ONERA
Chadler, France
lorenzo.chiodi@onera.fr

Abstract
This paper presents a new Polygeist framework for raising C code to Polyhedral MLIR. The framework is designed to be flexible and easy to use, and it can be integrated into existing Polyhedral MLIR frameworks.

The MLIR project aims to provide a framework for defining intermediate representation (IR). Feel free to use this category for any MLIR-related discussion!

Announcements
This category contains important announcements about MLIR, in particular our weekly meetings.

Newsletter
We will try to publish a weekly (or bi-weekly) newsletter about MLIR, trying to achieve what [Haskell](#)...

Tensor Compiler
This category is reserved to discuss the development of the Tensor Compiler component in MLIR.

Deprecation & Important Refactoring
This category collects recent news of MLIR that are deprecated and other announcement about lang.

TOSA
This category is reserved for coordination and discussion about development related to the TOSA dialect and tool.

Topic

Topic	Replies	Views	Activity
Tosa Mu1/32 shift incorrect result #TOSA	0	5	0%
MLIR Open Meeting: Tensor Compiler WG, 2025-04-29 #Announcements	1	45	7%
Bug in 'OperationEquivalence' (breaks '-cse' on 'trialg.index') #Bug	43	1,16	7%
MLIR Area Team Meeting / Office Hours #Announcements	6	146	9%
Mandatory data layout in the LLVM dialect #Bug	27	387	18%
MLIR Area Team Meeting Minutes - 2025-04-24 #Announcements	6	37	23%

<https://llvm.discourse.group/c/mlir/31>

<https://discord.gg/xS7Z362>

Open Meeting: Thursdays, 18:00 CEST