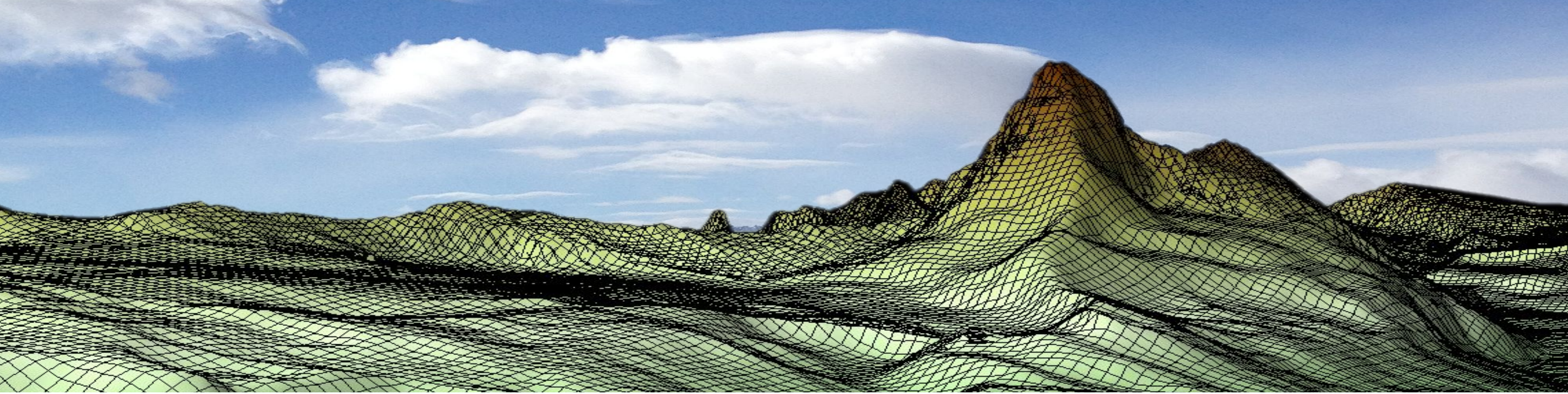


Domain-Specific Multi-IR Rewriting for GPU

Tobias Grosser *with Tobias Gysi, Christoph Mueller, Oleksandre Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Führer, Torsten Hoefler,*

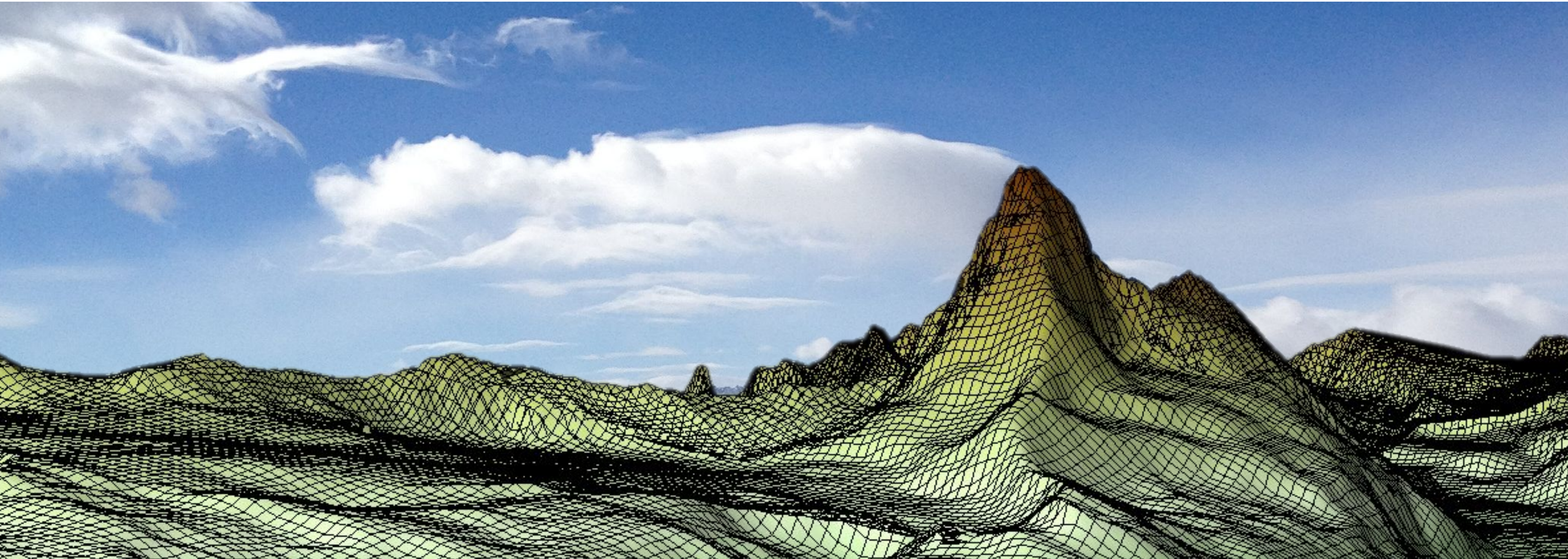
ETH Zurich, Vulcan Inc, University of Edinburgh

The COSMO Atmospheric Model

- Regional model used by 7 national weather services (DE, CH, IT, …)

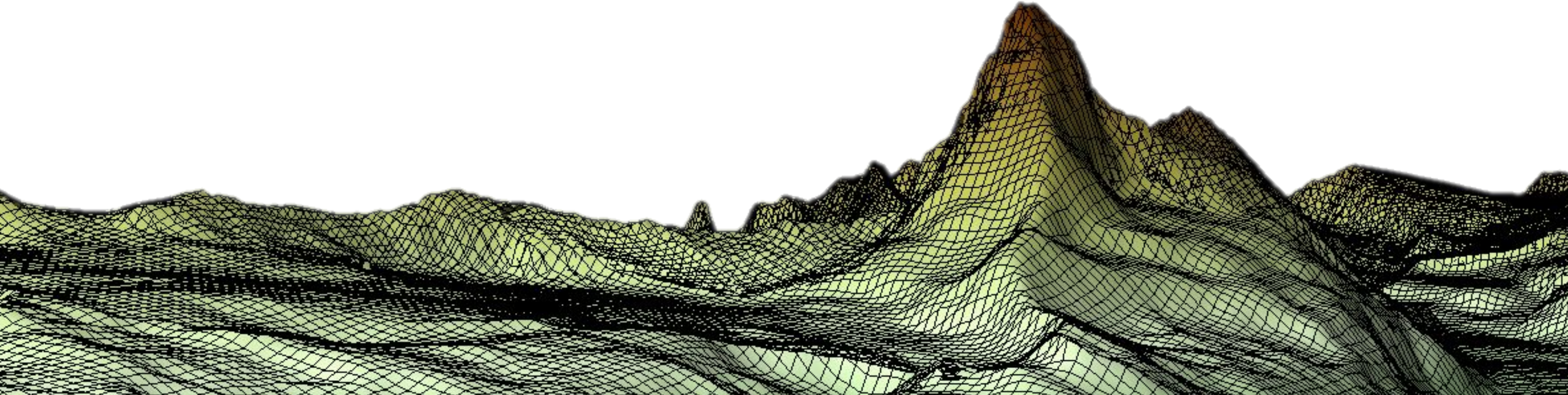
Resolution (35m)

What resolution is needed to predict if there is snow out of the banner cloud at Matterhorn?



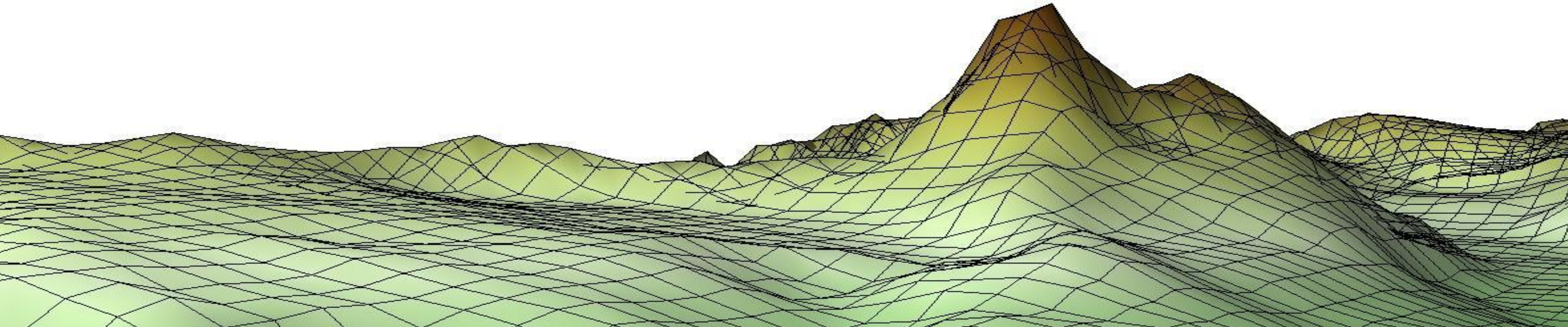
Resolution (35m)

What resolution is needed to predict if there is snow out of the banner cloud at Matterhorn?



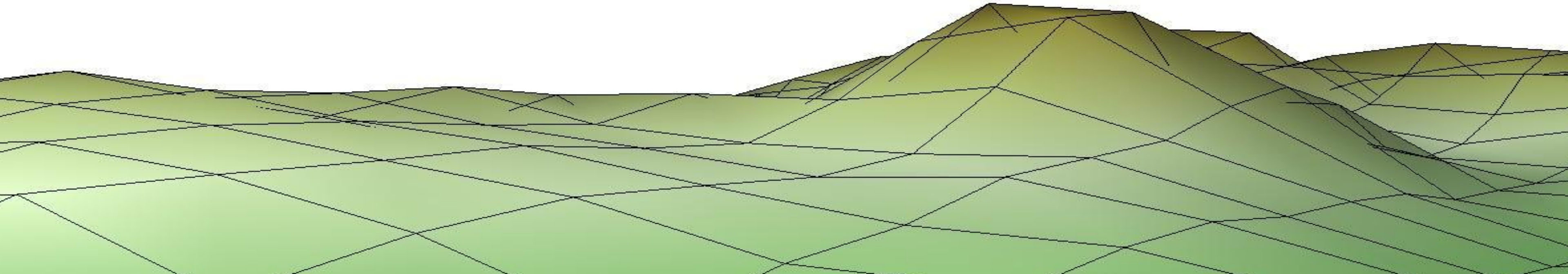
Resolution (140m)

What resolution is needed to predict if there is snow out of the banner cloud at Matterhorn?



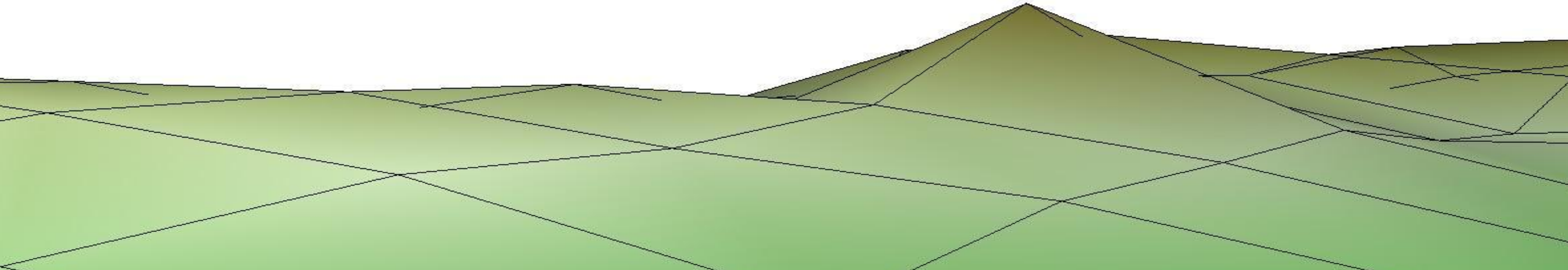
Resolution (560m)

What resolution is needed to predict if there is snow out of the banner cloud at Matterhorn?



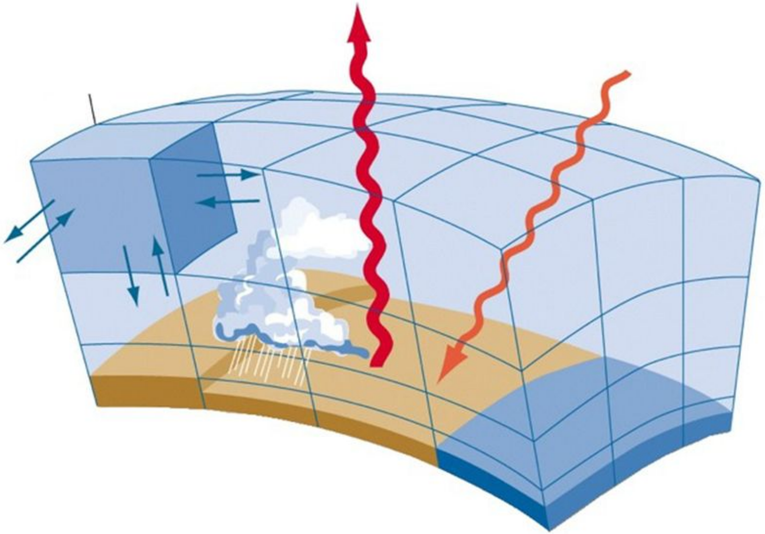
Resolution (1.1km – Weather Forecast Today)

What resolution is needed to predict if there is snow out of the banner cloud at Matterhorn?



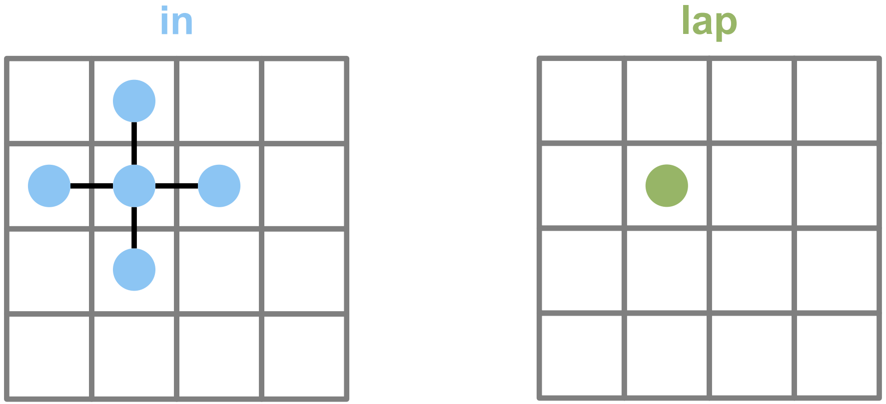
Domain-Science vs Computer-Science

- solve PDE
- finite differences
- structured grid

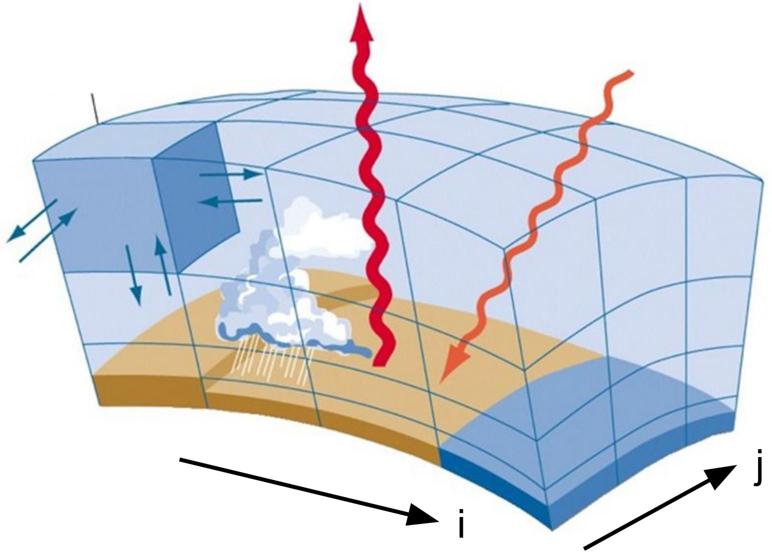


- element-wise computation
- fixed neighborhood

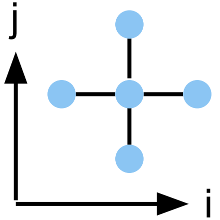
lap(i,j) = -4.0 * in(i,j) +
in(i-1,j) + in(i+1,j) +
in(i,j-1) + in(i,j+1)



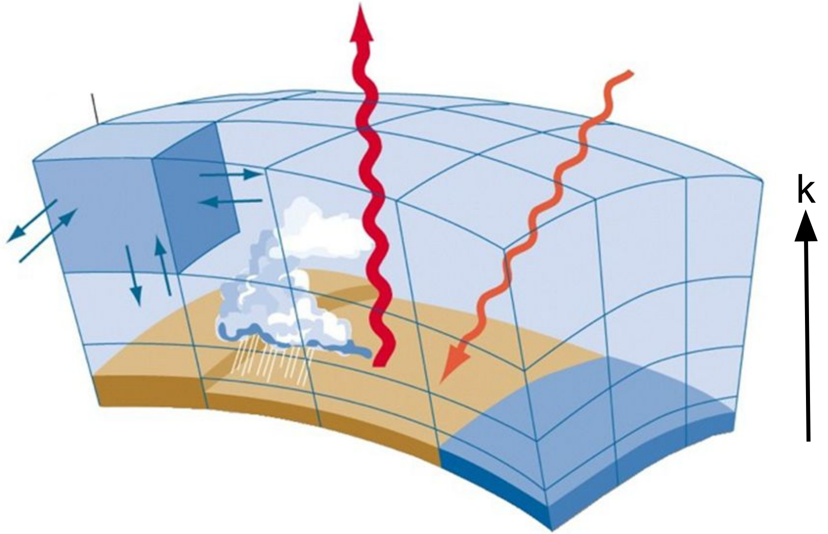
Algorithmic Motifs – Finite Differences



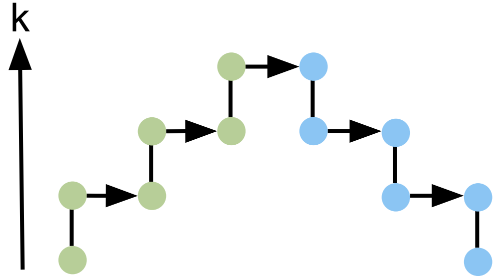
- stencils (no loop carried dependencies)
- mostly horizontal dependencies



Algorithmic Motifs – Tridiagonal Systems



- vertical dependencies
- loop carried dependencies



Global Climate Modeling Challenges

Computational Challenge

Resolution 1 km^2

Surface 500,000,000 km^2 (Switzerland 40,000 km^2)

Duration 100 years (Weather model 2-7 days)

Time-to-Solution 3 months

Software Challenge (COSMO)

Language Fortran

Size 300,000 LoC

Loops thousands

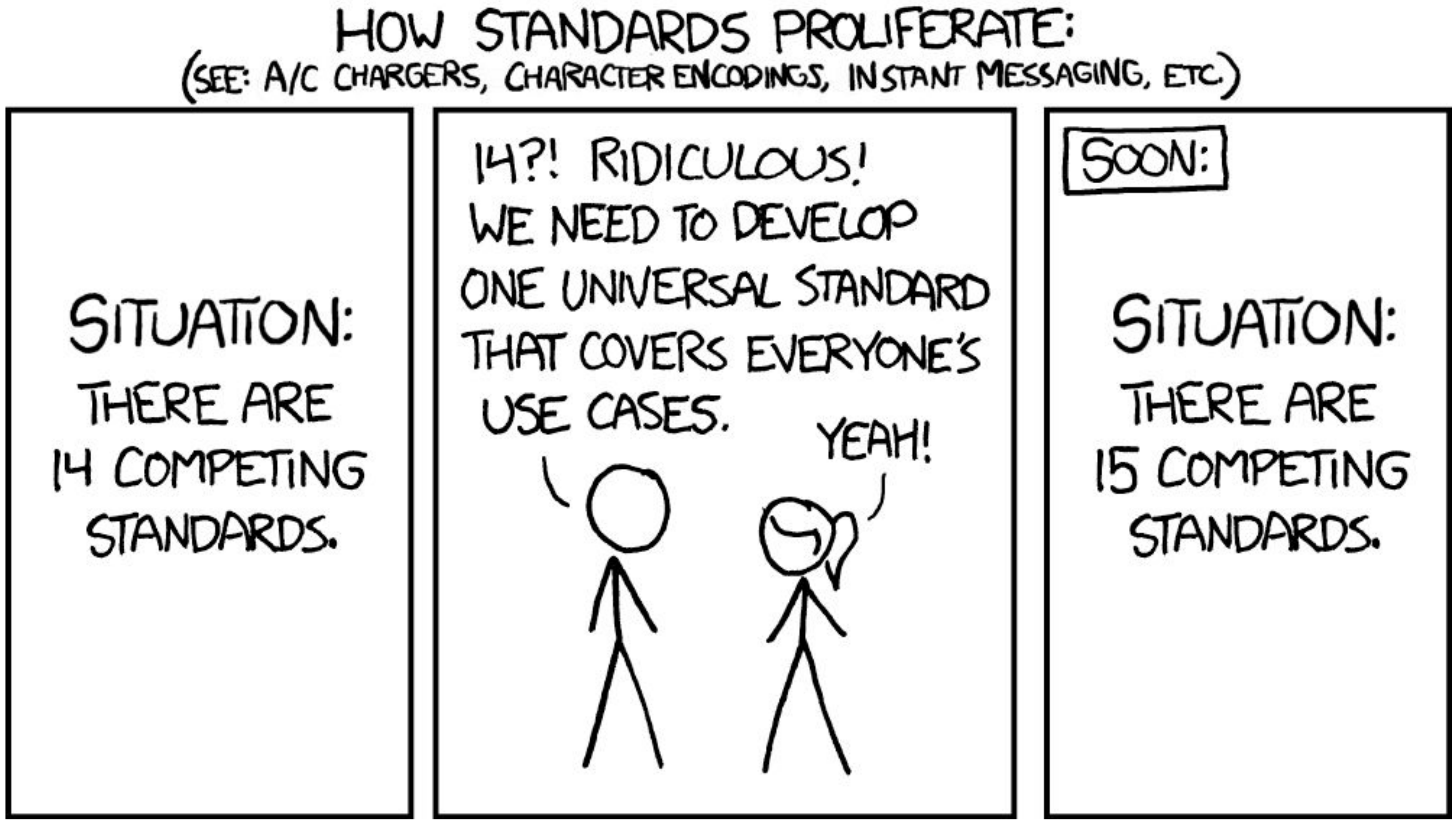
Domains Physics, Stencil (Finite Volume/Difference), General-Purpose, MPI

Hardware Challenge

Today’s hardware has insufficient memory bandwidth

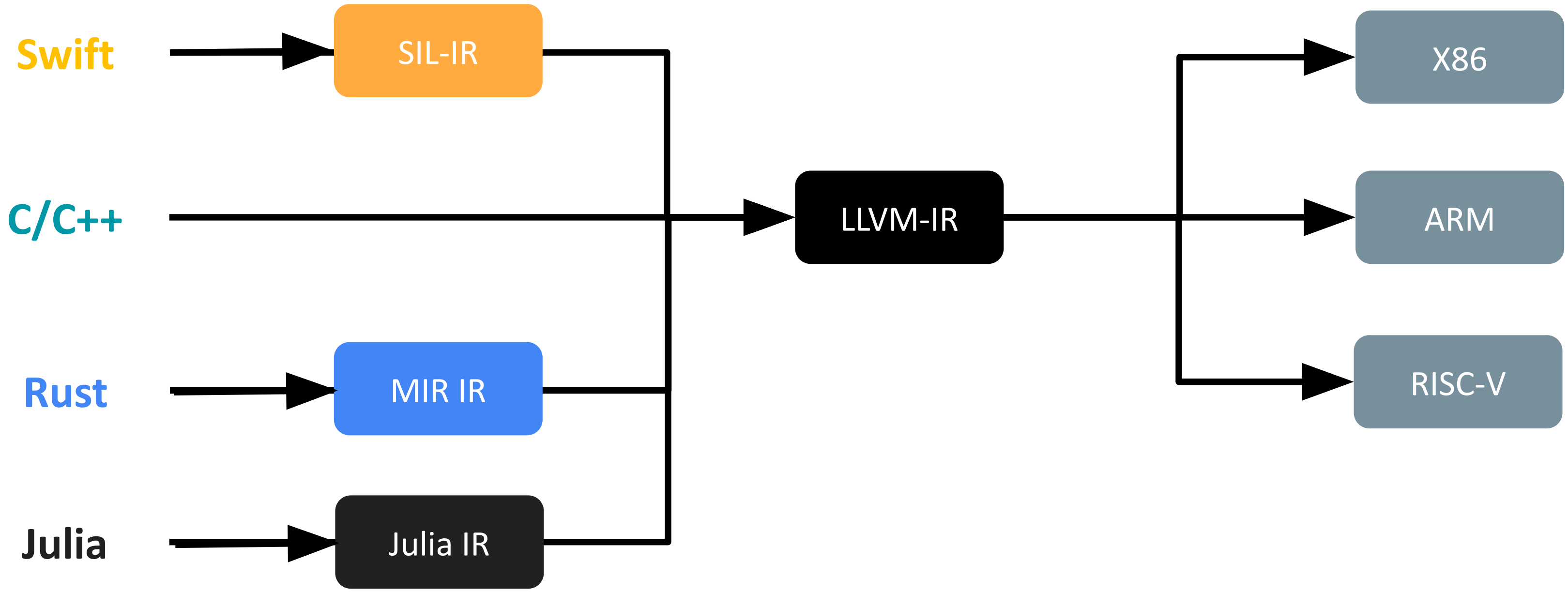
Community Challenge

Large Fortran based Ecosystem, DSL and Non-DSL code, HPC engineer wants control, …

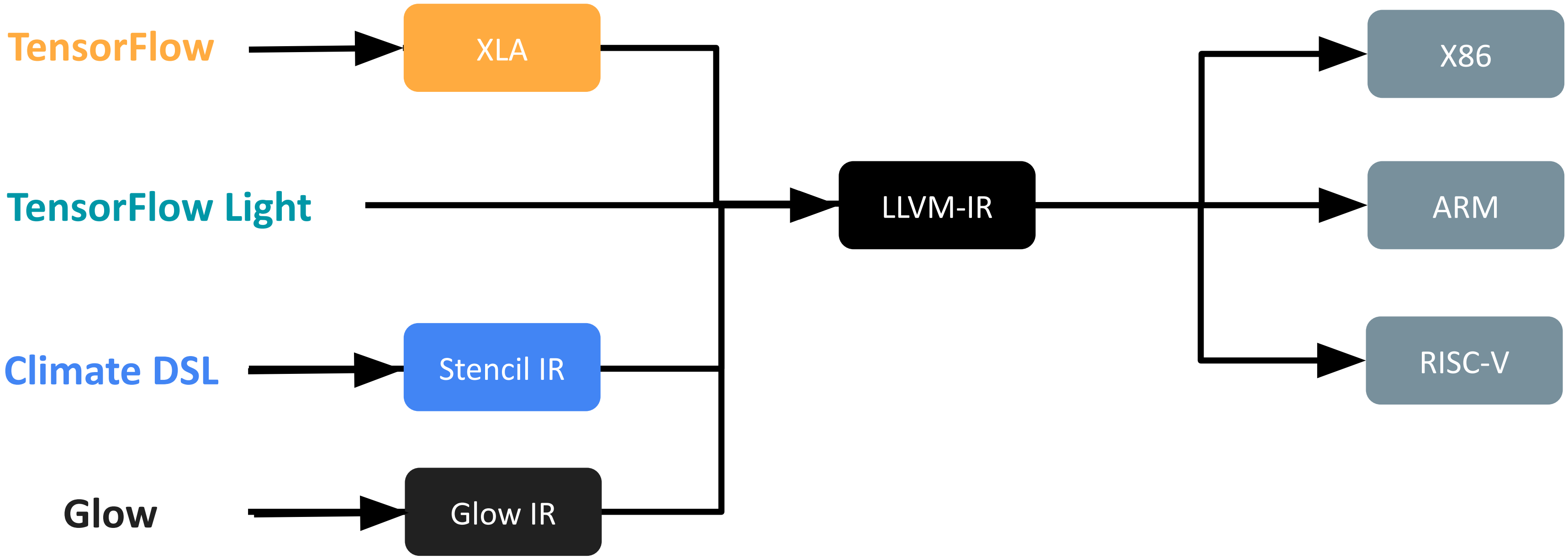

HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC.)
SITUATION:
THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION:
THERE ARE 15 COMPETING STANDARDS.


ice at record
nd shrinkin
TensorFlow
Save our
Planet

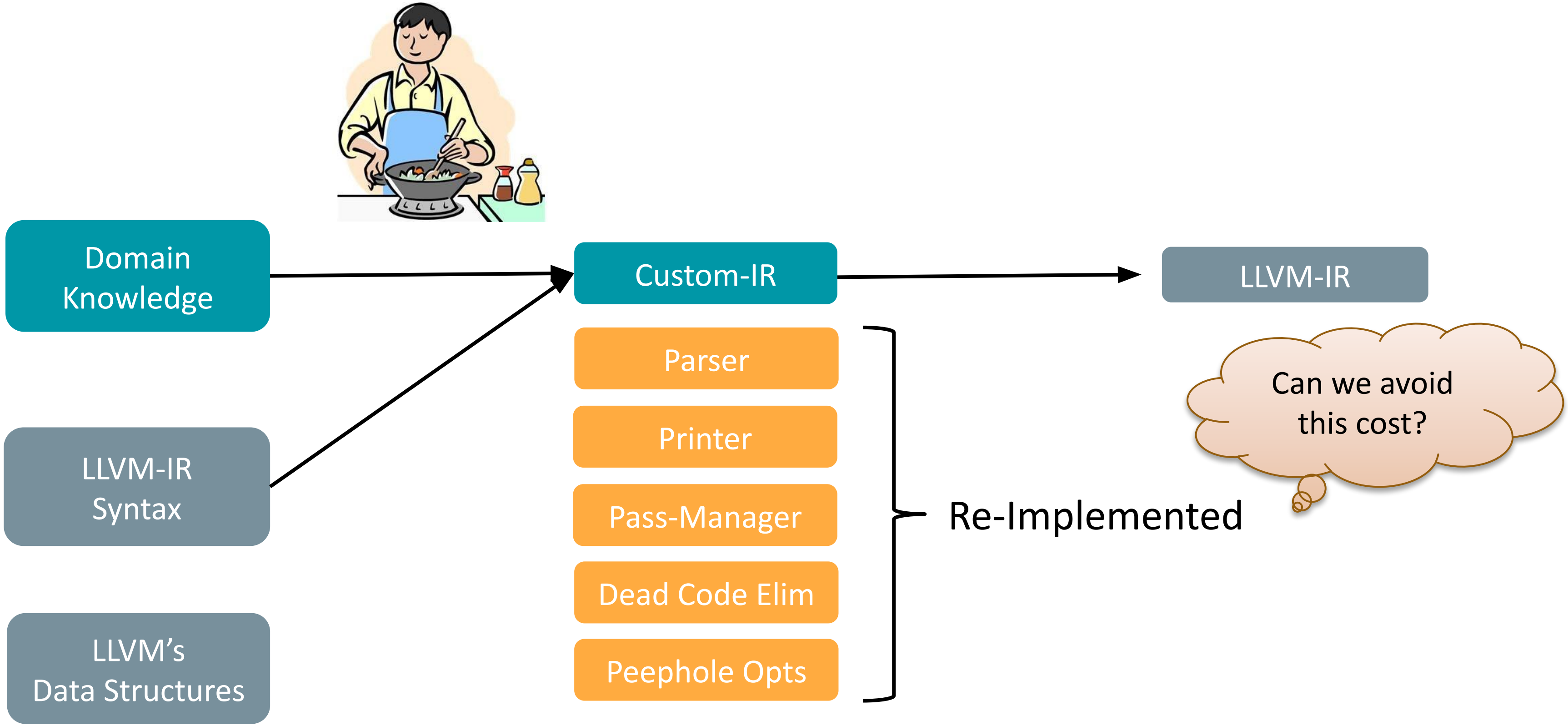
Compiler Pipelines



Compiler Pipelines for Deep Learning



How to build a Modern High-Level Compiler IR



LLVM IR

Function

Argument

```
define void @func(i64 %n, double* %A) {

entry:
  br label %for.cond

for.cond:
  %i.0 = phi i64 [ 0, %entry ], [ %add, %for.body ]
  %cmp = icmp slt i64 %i.0, %n
  br i1 %cmp, label %for.body, label %for.end

for.body:
  %arrayidx = getelementptr inbounds double, double* %A, i64 %i.0
  store double 2.100000e+01, double* %arrayidx
  %add = add nuw nsw i64 %i.0, 1
  br label %for.cond, !llvm.loop !7

for.end:
  ret void
}
```

Basic Block

PHI-Node

Terminator

Operation

SIL - The Swift Compiler IR

```
sil @fibonacci: $(Swift.Int) -> () {
entry(%limi: $Swift.Int):
  %lim = struct_extract %limi: $Swift.Int, #Int.value
  %print = function_ref @print: $(Swift.Int) -> ()
  %a0 = integer_literal $Builtin.Int64, 0
  %b0 = integer_literal $Builtin.Int64, 1
  br loop(%a0: $Builtin.Int64, %b0: $Builtin.Int64)

loop(%a: $Builtin.Int64, %b: $Builtin.Int64):
  %lt = builtin "icmp_lt_Int64"(%b: $Builtin.Int64, %lim: $Builtin.Int64): $Builtin.Int1
 cond_br %lt: $Builtin.Int1, body, exit

body:
  %b1 = struct $Swift.Int (%b: $Builtin.Int64)
  apply %print(%b1) : $(Swift.Int) -> ()
  %c = builtin "add_Int64"(%a: $Builtin.Int64, %b: $Builtin.Int64): $Builtin.Int64
  br loop(%b: $Builtin.Int64, %c: $Builtin.Int64)

exit:
  %unit = tuple ()
  return %unit: $()
}
```

Can you understand it?

Similarities with LLVM-IR?

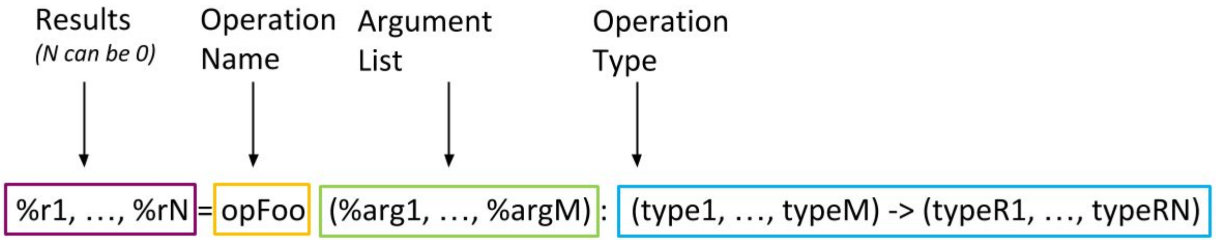
Comparing existing Compiler IRs

Differences

- Types
  - names
  - Semantics
- Operations
  - names
  - Semantics
- Control flow constructs

Similarities

- Units
  - Functions
  - Basic Blocks
  - Operations
- Operations
  - Take arguments
  - Return (multiple) results
- Some kind of control flow
- Existence of types




```
%val = get_random() : () -> i64
%sum = add (%val, %val) : (i64, i64) -> i64
print(%sum) : (i64) -> ()
```

MLIR:
Abstraction Sharing Across Communities

An MLIR Operation

Results *(N can be 0)* | Operation Name | Argument List | Operation Type

```
%r1, …, %rN = opFoo (%arg1, …, %argM) : (type1, …, typeM) -> (typeR1, …, typeRN)
```

```
%val = get_random() : () -> i64
%sum = add (%val, %val) : (i64, i64) -> i64
print(%sum) : (i64) -> ()
```

An MLIR Block

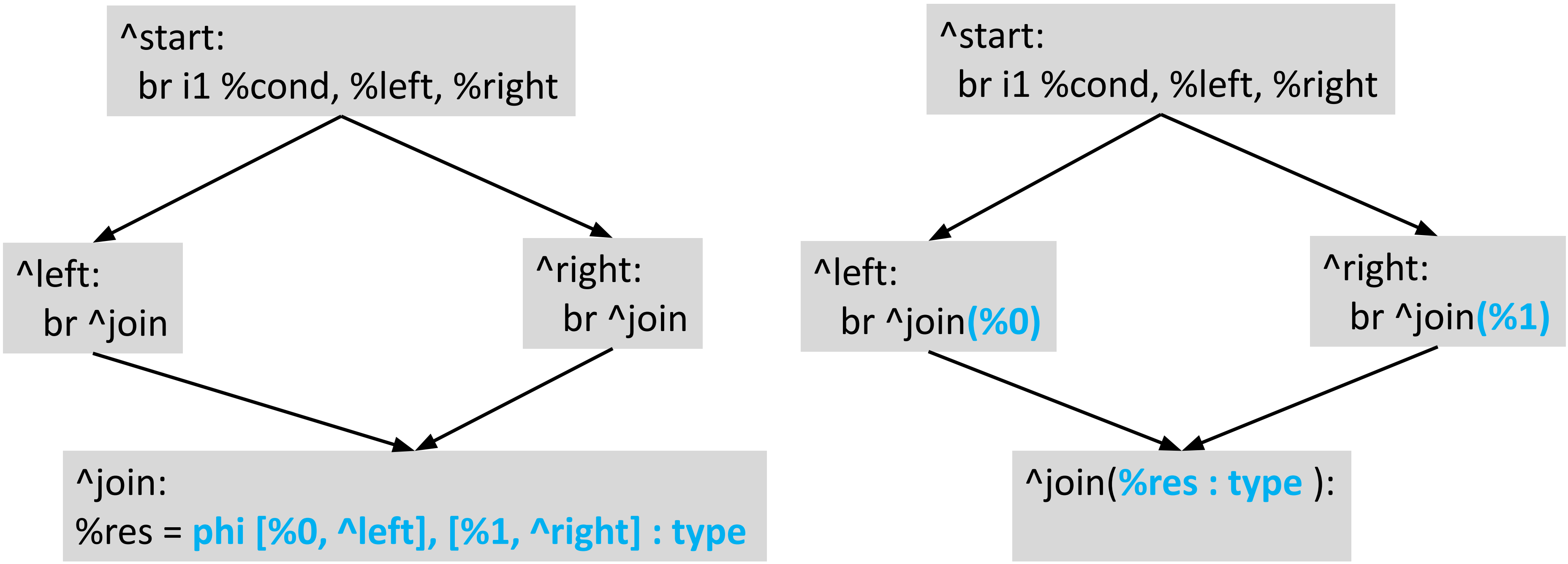
Block Name

Instruction List

```
^block:
%r1_1, …, %r1_N = opA  (%arg1, …, %argM) : type
opB  (%r1, %arg2) : type
%r3 = opB  () : type
termOp(%r3) : type
```

Terminator Operation

Block Arguments instead of PHI-Nodes



Functions

Function Name

Argument List

Return Types

```
func @foo (%arg1: !type1, ..., %argN : !typeN) -> (retType1, ..., retTypeM)
{

  <list of blocks>

}
```

Dialects

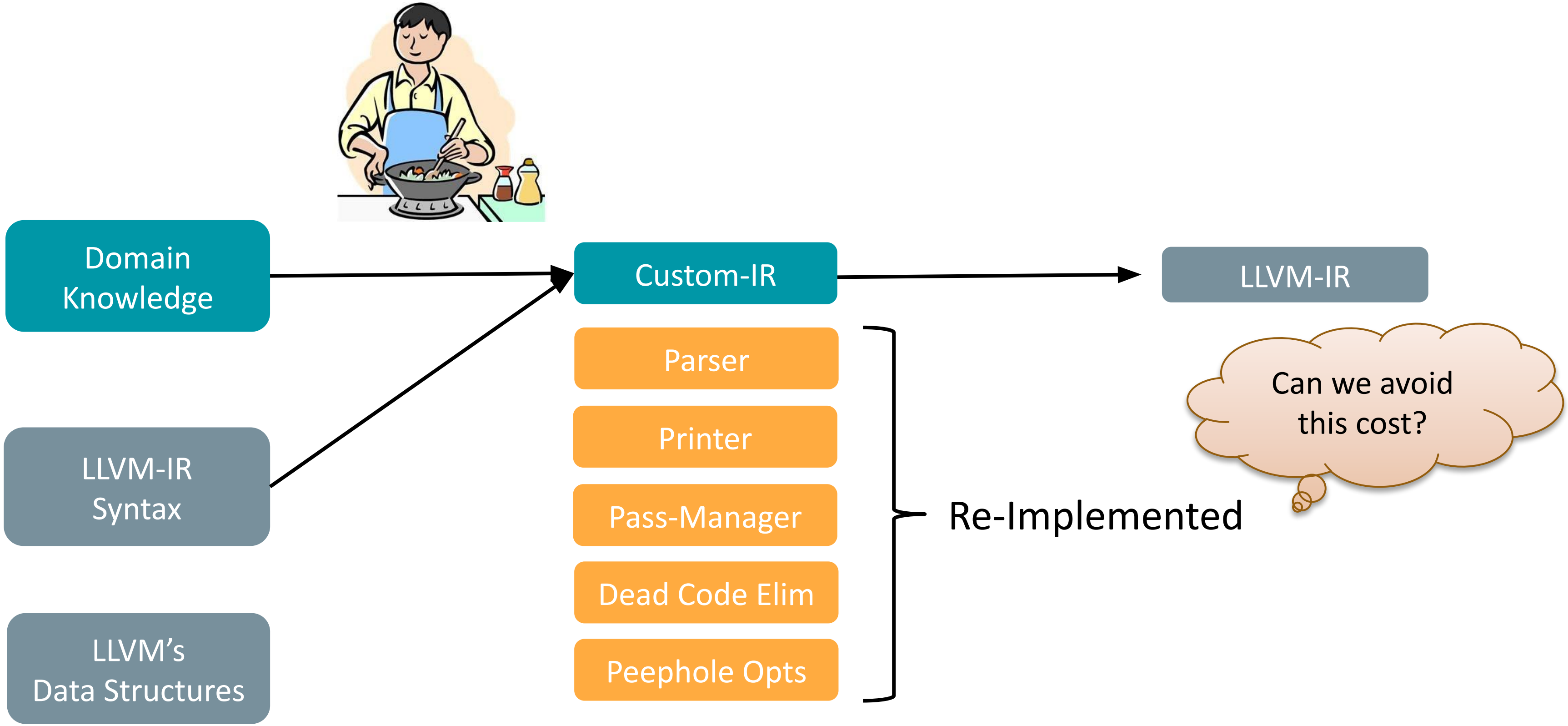
```
%r1, …, %rN = dialectA.opFoo   (%arg1, …, %argM) :
                  (!dialectB<type1>, …, !dialectC<typeM>) -> (!dialectA<typeR>)
```

Terminator Operations

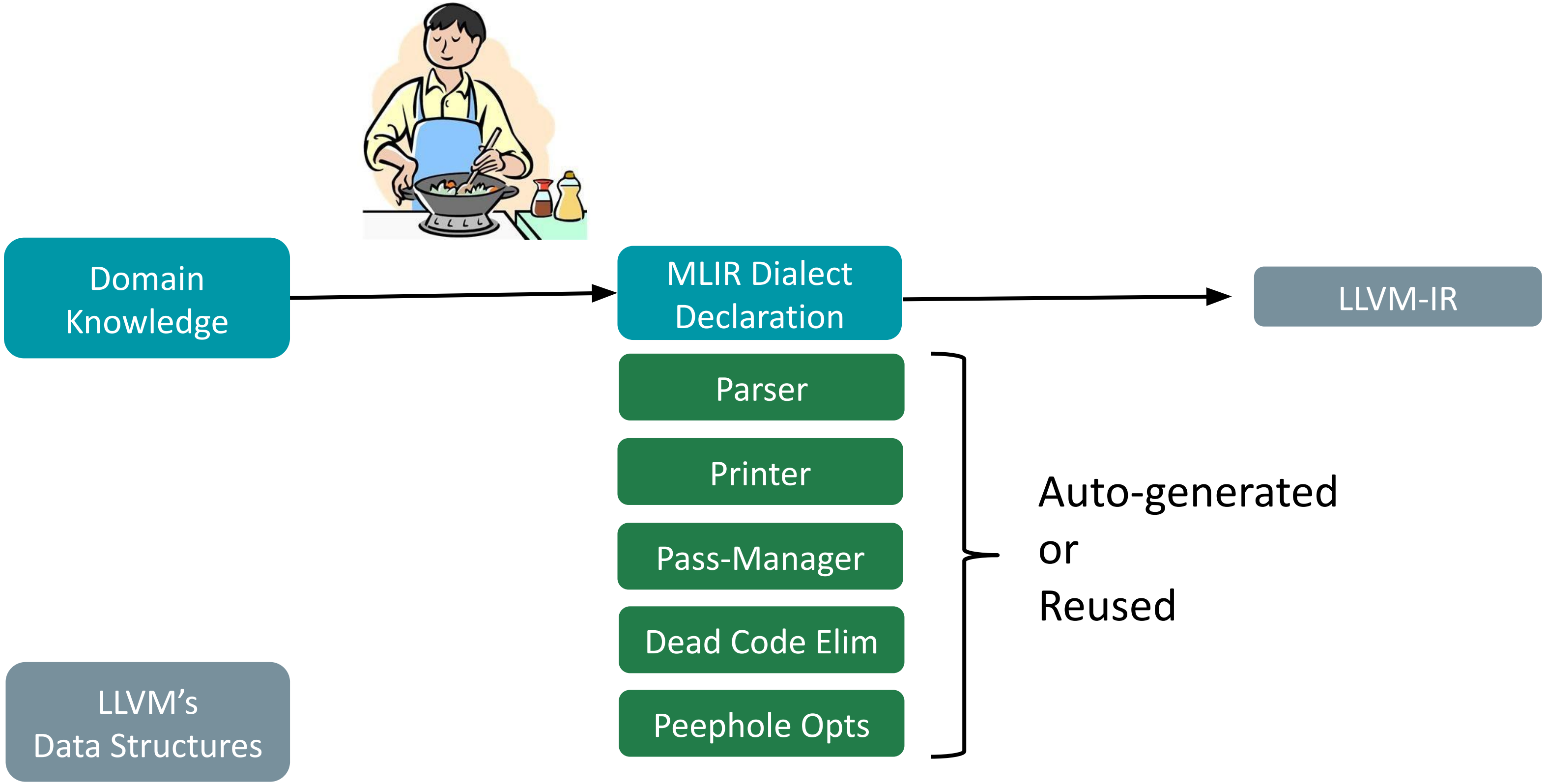
Special Operations at the end of each Block

- **Take an (optional) list of blocks and their block arguments**
- **Each block must be terminated by a terminator operation**
- **Allows custom definition of terminator instructions**

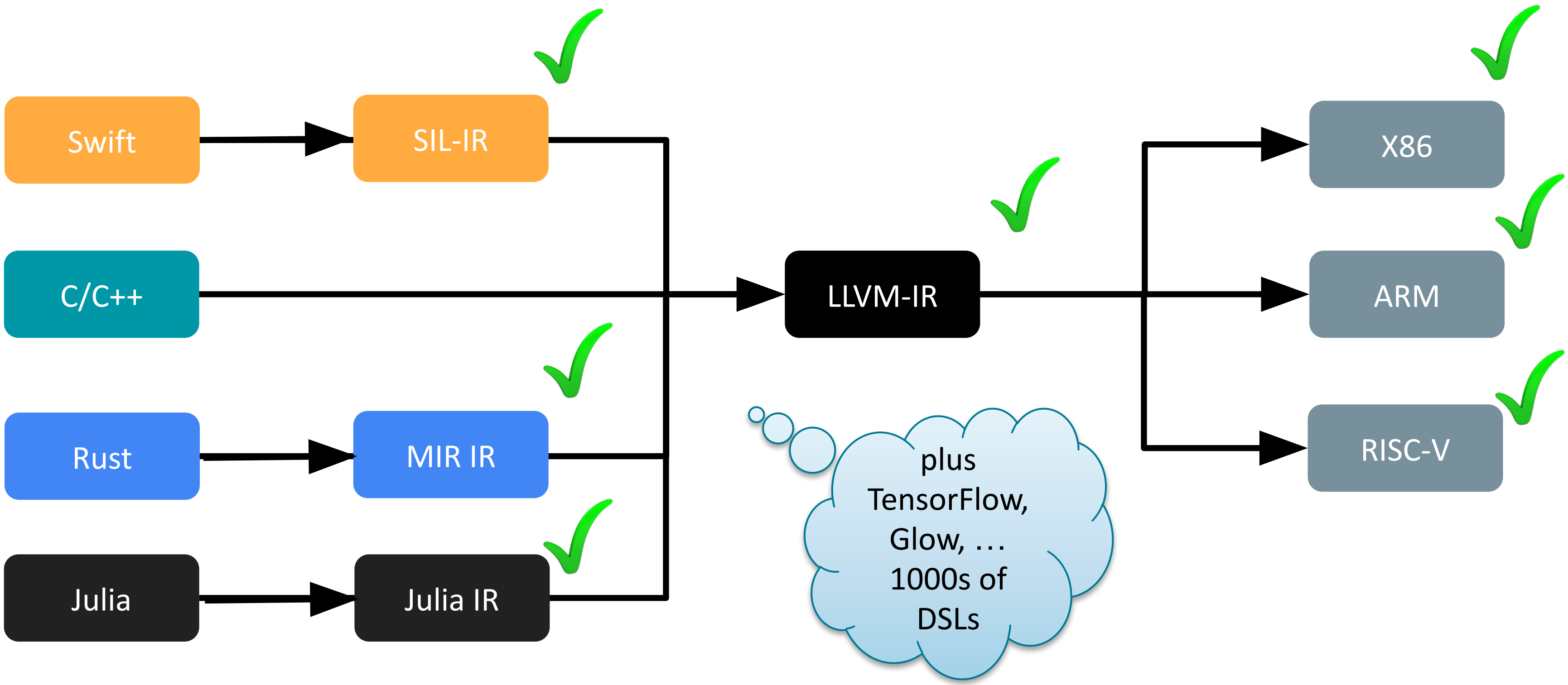
Building a Modern Compiler IR



Building a Modern Compiler IR – using MLIR

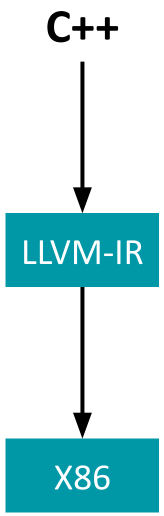


LLVM-Quality Everywhere

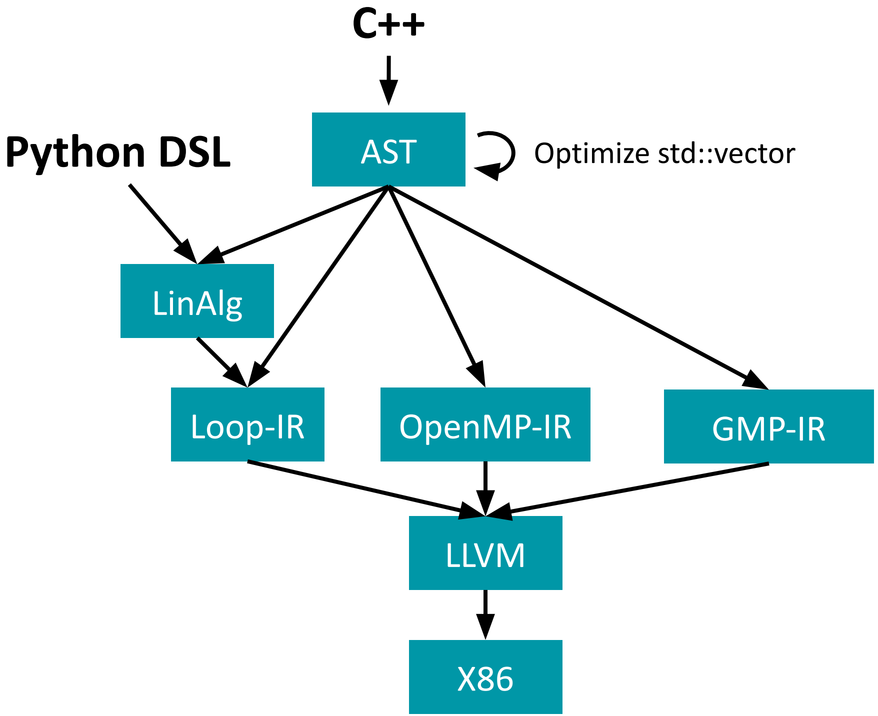


New Paradigm: Multi Abstraction Rewriting

Traditional Compilation

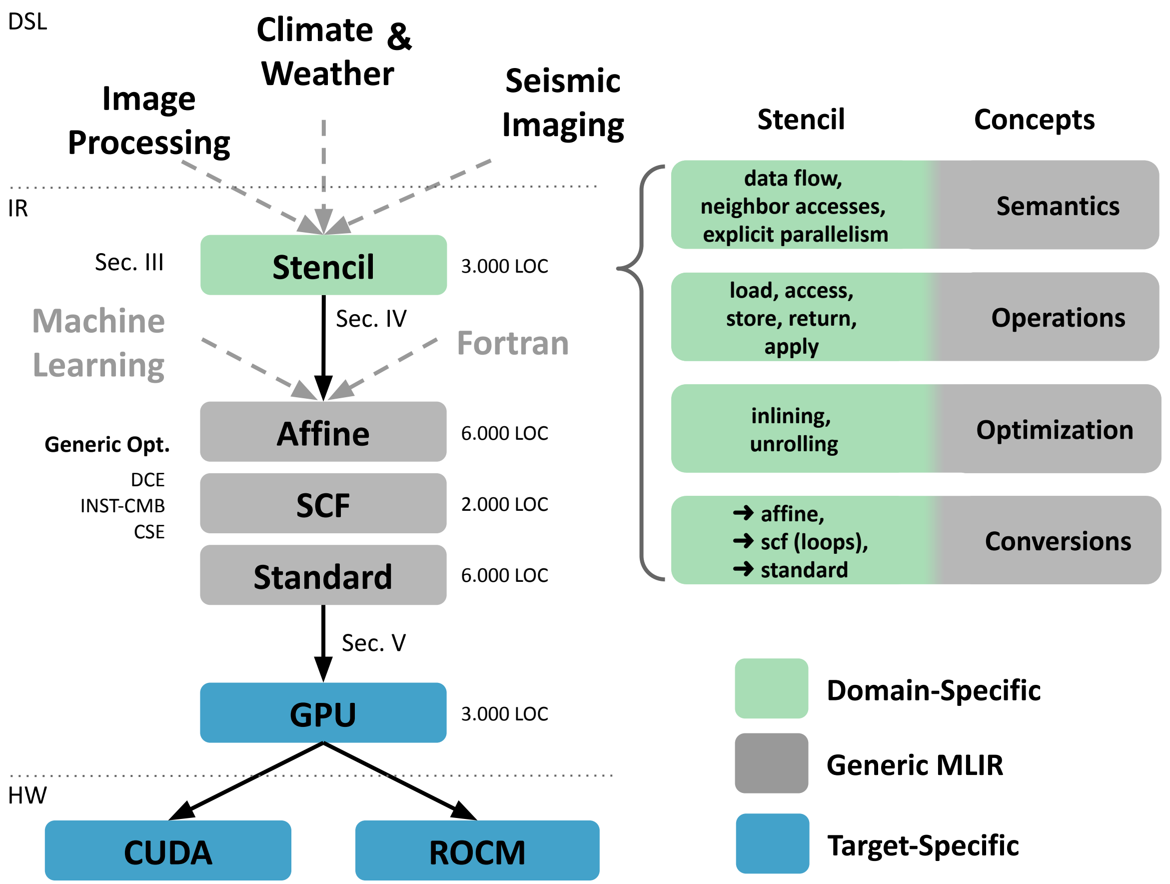


MLIR



The Open Earth Compiler

Architecture



The MLIR Stencil Dialect

Stencil Dialect

```
func @sum(%in : !stencil.field<?x?x?xf64>, %out : !stencil.field<?x?x?xf64>) {

  %0 = stencil.cast %in ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>
  %1 = stencil.cast %out ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>

  %2 = stencil.load %0 : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>

  %3 = stencil.apply (%arg0 = %2 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
      %5 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %6 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

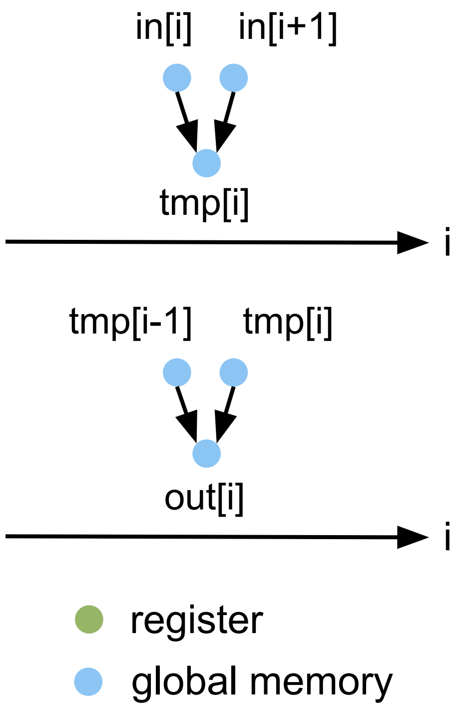
static offsets

stencil operator

stencil inlining and stencil unrolling

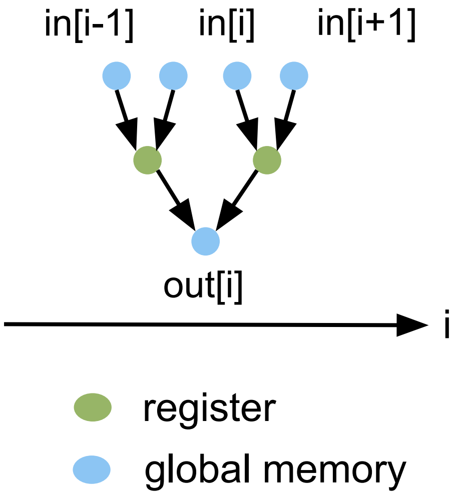
Stencil Inlining

```
for(int i = IB; i < IE; i++)
  tmp[i] = in[i] + in[i+1];

for(int i = IB; i < IE; i++)
  out[i] = tmp[i] + tmp[i-1];
```



Stencil Inlining

```
for(int i = IB; i < IE; i++)
  tmp[i] = in[i] + in[i+1];

for(int i = IB; i < IE; i++)
  out[i] = tmp[i] + tmp[i-1];
```




```
for(int i = IB; i < IE; i++)
  out[i] =
    (in[i] + in[i+1]) +
    (in[i-1] + in[i]);
```

Lowering Patterns - FuncOp

```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = stencil.cast %in ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>
  %1 = stencil.cast %out ([-4, -4, -4]:[68, 68, 68]) : !stencil.field<?x?x?xf64>

  %2 = stencil.load %0 : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>

  %3 = stencil.apply (%arg0 = %2 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
      %5 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %6 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

Lowering Patterns - CastOp

```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = memref_cast %arg0 : memref<?x?x?xf64> to memref<72x72x72xf64>
  %1 = memref_cast %arg1 : memref<?x?x?xf64> to memref<72x72x72xf64>

  %2 = stencil.load %0 : (!stencil.field<?x?x?xf64>) -> !stencil.temp<?x?x?xf64>

  %3 = stencil.apply (%arg0 = %2 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
      %5 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %6 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

Lowering Patterns - LoadOp

```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = memref_cast %arg0 : memref<?x?x?xf64> to memref<72x72x72xf64>
  %1 = memref_cast %arg1 : memref<?x?x?xf64> to memref<72x72x72xf64>

  %2 = subview %0[4,4,3][64,64,66][1,1,1] : memref<72x72x72xf64> to memref<64x64x66xf64, #map0>

  %3 = stencil.apply (%arg0 = %2 : !stencil.temp<?x?x?xf64>) -> !stencil.temp<?x?x?xf64> {
      %5 = stencil.access %arg0[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %6 = stencil.access %arg0[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

Lowering Patterns - ApplyOp

```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = memref_cast %arg0 : memref<?x?x?xf64> to memref<72x72x72xf64>
  %1 = memref_cast %arg1 : memref<?x?x?xf64> to memref<72x72x72xf64>

  %2 = subview %0[4,4,3][64,64,66][1,1,1] : memref<72x72x72xf64> to memref<64x64x66xf64, #map0>

  %c0 = constant 0 : index
  %c64 = constant 64 : index
  %c1 = constant 1 : index
  scf.parallel (%i, %j, %k) = (%c0, %c0, %c0) to (%c64, %c64, %c64) step (%c1, %c1, %c1) {
      %5 = stencil.access %2[1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %6 = stencil.access %2[-1, 0, 0] : (!stencil.temp<?x?x?xf64>) -> f64
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
      scf.yield
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

Lowering Patterns - AccessOp

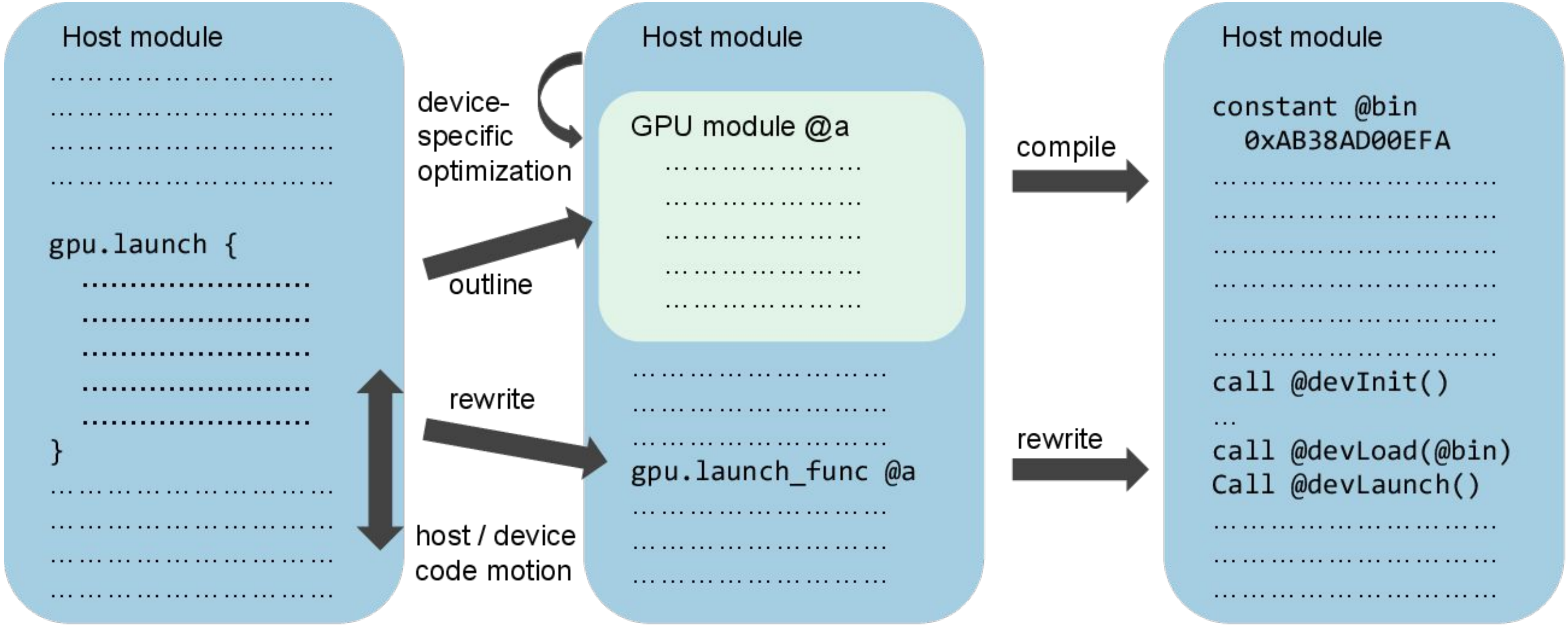
```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = memref_cast %arg0 : memref<?x?x?xf64> to memref<72x72x72xf64>
  %1 = memref_cast %arg1 : memref<?x?x?xf64> to memref<72x72x72xf64>

  %2 = subview %0[4,4,3][64,64,66][1,1,1] : memref<72x72x72xf64> to memref<64x64x66xf64, #map0>

  %c0 = constant 0 : index
  %c64 = constant 64 : index
  %c1 = constant 1 : index
  scf.parallel (%i, %j, %k) = (%c0, %c0, %c0) to (%c64, %c64, %c64) step (%c1, %c1, %c1) {
      %4 = affine.apply #map2(%i)
      %5 = load %2[%k, %j, %4] : memref<64x64x66xf64, #map0>
      %6 = load %2[%k, %j, %i] : memref<64x64x66xf64, #map0>
      %7 = addf %5, %6 : f64
      stencil.return %7 : f64
      scf.yield
  }

  stencil.store %3 to %1 ([0, 0, 0]:[64, 64, 64]) : !stencil.temp<?x?x?xf64> to !stencil.field<?x?x?xf64>
  return
}
```

Lowering Patterns - StoreOp/ReturnOp

```
func @sum(%arg0: memref<?x?x?xf64>, %arg1: memref<?x?x?xf64>) {

  %0 = memref_cast %arg0 : memref<?x?x?xf64> to memref<72x72x72xf64>
  %1 = memref_cast %arg1 : memref<?x?x?xf64> to memref<72x72x72xf64>

  %2 = subview %0[4,4,3][64,64,66][1,1,1] : memref<72x72x72xf64> to memref<64x64x66xf64, #map0>
  %3 = subview %1[4,4,4][64,64,64][1,1,1] : memref<72x72x72xf64> to memref<64x64x64xf64, #map1>

  %c0 = constant 0 : index
  %c64 = constant 64 : index
  %c1 = constant 1 : index
  scf.parallel (%i, %j, %k) = (%c0, %c0, %c0) to (%c64, %c64, %c64) step (%c1, %c1, %c1) {
      %4 = affine.apply #map2(%i)
      %5 = load %2[%k, %j, %4] : memref<64x64x66xf64, #map0>
      %6 = load %2[%k, %j, %i] : memref<64x64x66xf64, #map0>
      %7 = addf %5, %6 : f64
      store %7, %3[%k, %i, %j] : memref<64x64x64xf64, #map1>
      scf.yield
  }

  return
}
```

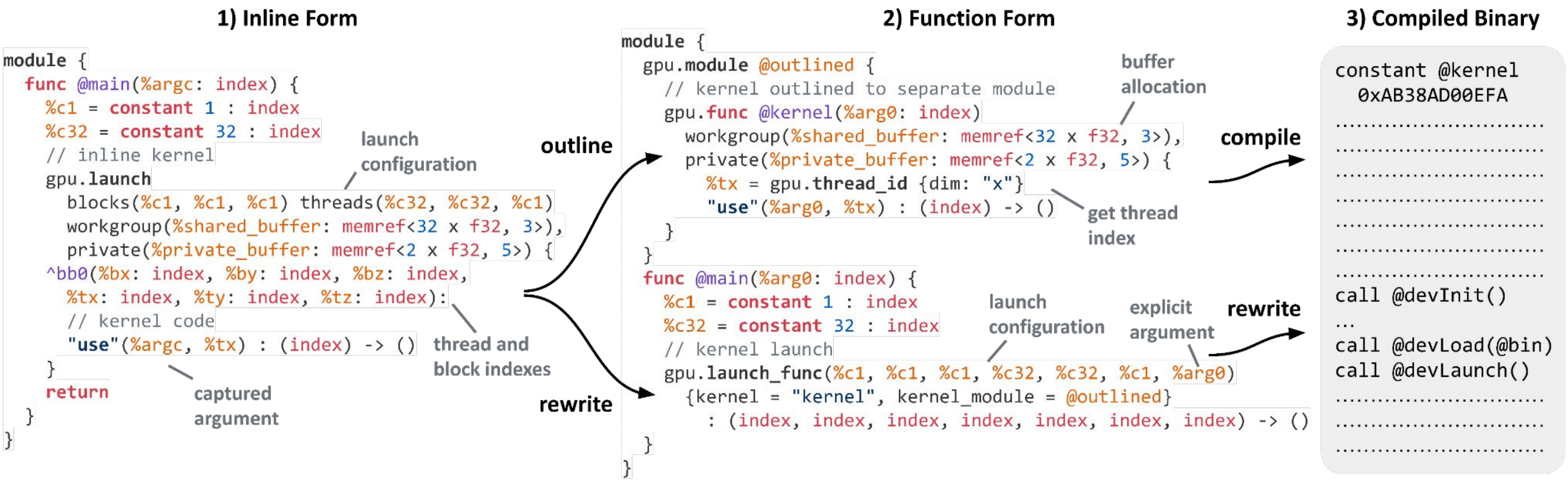
The MLIR GPU Dialect

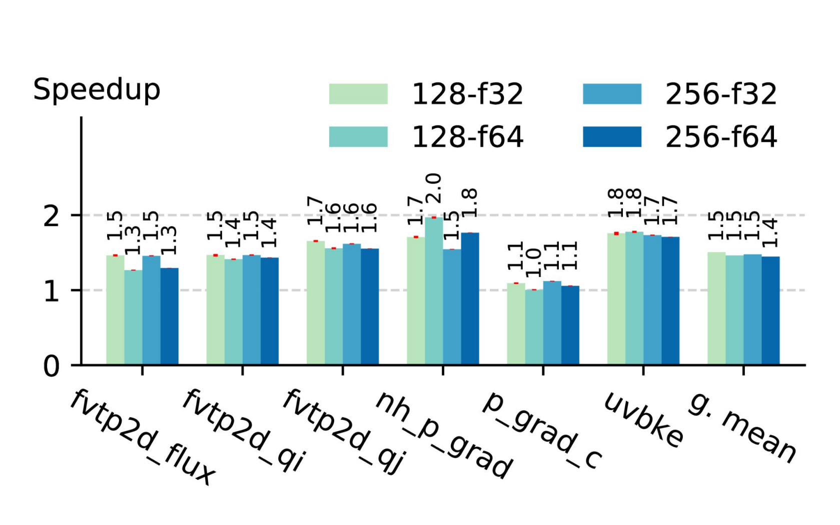
GPU Dialect



target AMD and NVIDIA GPUs

Extracting GPU Kernel Launches into Separate Modules




Speedup
128-f32
128-f64
256-f32
256-f64
0
1
2
1.5 1.3 1.5 1.3
1.5 1.4 1.5 1.4
1.7 1.6 1.6 1.6
1.7 2.0 1.5 1.8
1.1 1.0 1.1 1.1
1.8 1.8 1.7 1.7
1.5 1.5 1.5 1.4
fvtp2d_flux
fvtp2d_qi
fvtp2d_qj
nh_p_grad
p_grad_c
uvbke
g. mean

Evaluation

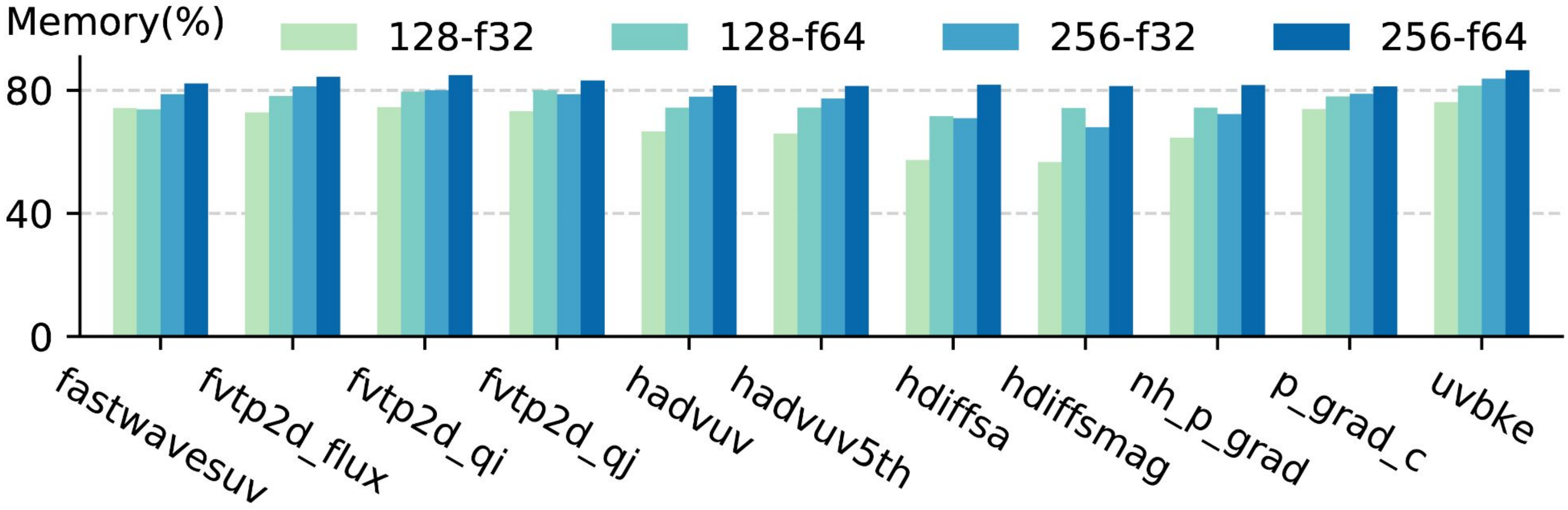
Experimental Setup

NVIDIA Tesla
V100-SMX2

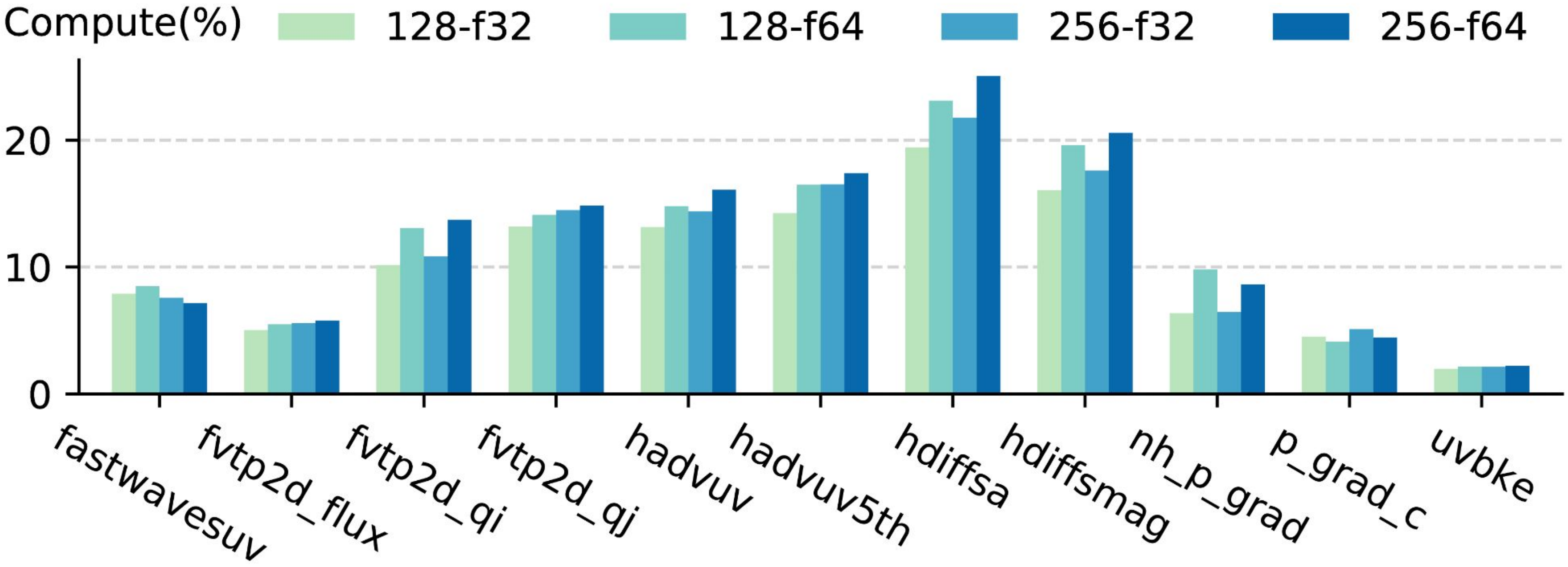
900 GB/s Bandwidth

- COSMO | Europe
    - Regional numerical weather model
    - Kernels
        - Fastwaves Sound Wave Forward Integration
        - Horizontal-Advection (2x)
        - Horizontal-Diffusion (2x)
- SV3 | US
    - Dynamical core of CM4 an GEOS-5 global climate models
    - Kernels
        - Monotone Finite Volume Advection Operators (3x)
        - 3D Pressure Gradients (2x)
        - Preprocessing for Kinetic Energy Computation

High Memory Bandwidth Utilization

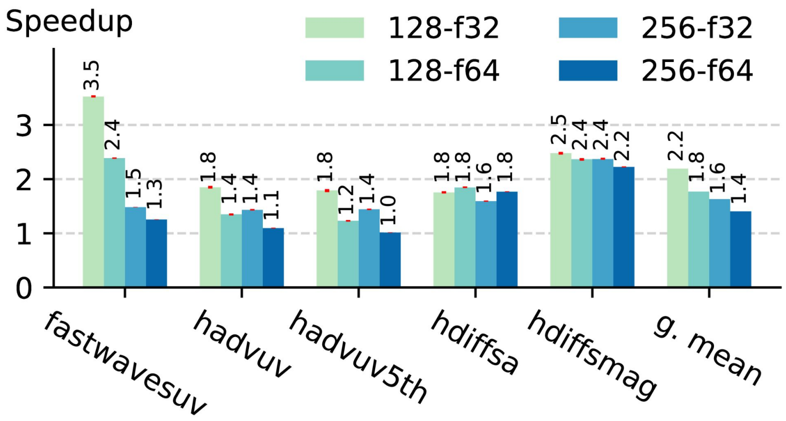


Reasonable Compute Utilization

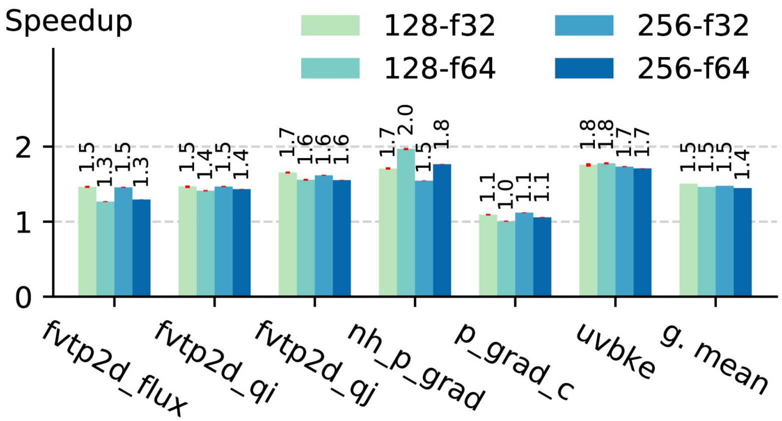


The OEC is faster than previous Climate DSLs

Europe | COSMO (GridTools)



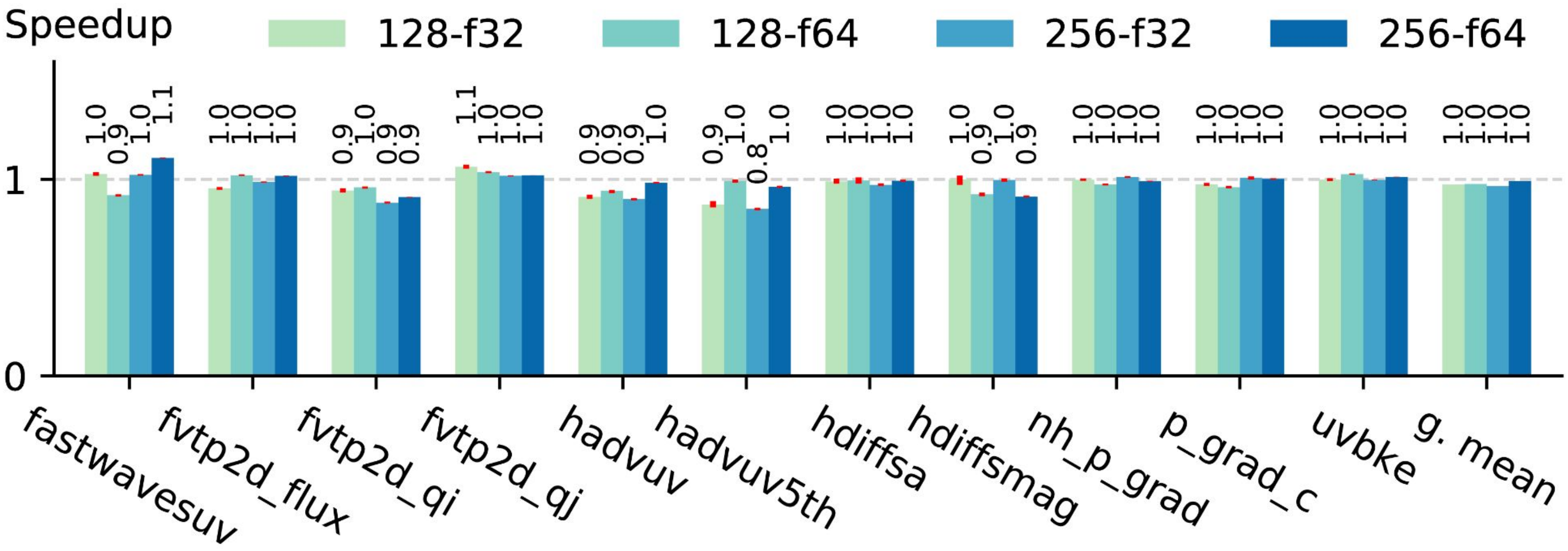
US | FV3 (Dawn)



The OEC is on-par with the fastest Stencils DSLs

Halide Image Processing DSL

Halide introduces rounding errors for performance!

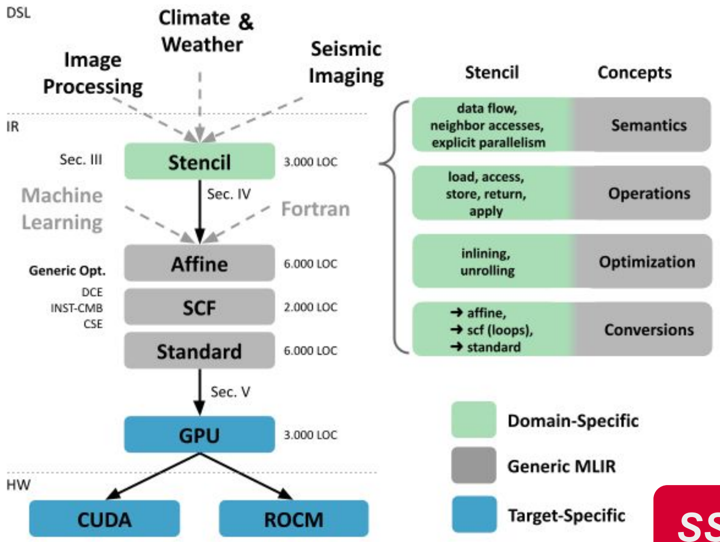


Conclusion

Conclusion

Open-Source Community

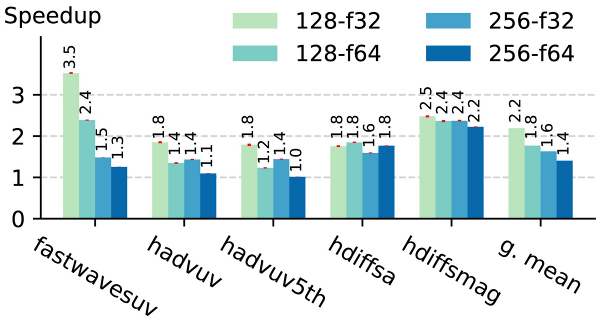
A Modular Climate Compiler



SSA Stencil Dialect

SSA GPU Dialect

High Performance



xDSL

What’s Next?

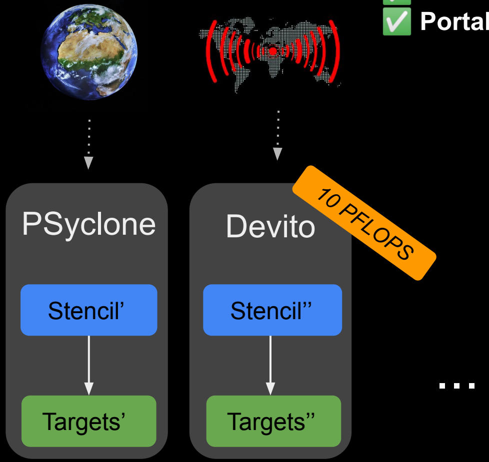
Efficient Cross-Domain DSL Development for Exascale

Tobias Grosser, Nick Brown, Amrey Krause, Michel Steuwer (U. Edinburgh), Gerard Gorman, Paul Kelly (Imperial)

XDSL4X

Today: Monolithic DSLs

✅ Performance
✅ Productivity
✅ Portability



Technical Challenges

❌ Composability
❌ Interoperability
❌ Code Reuse
❌ Longevity

Societal Challenges

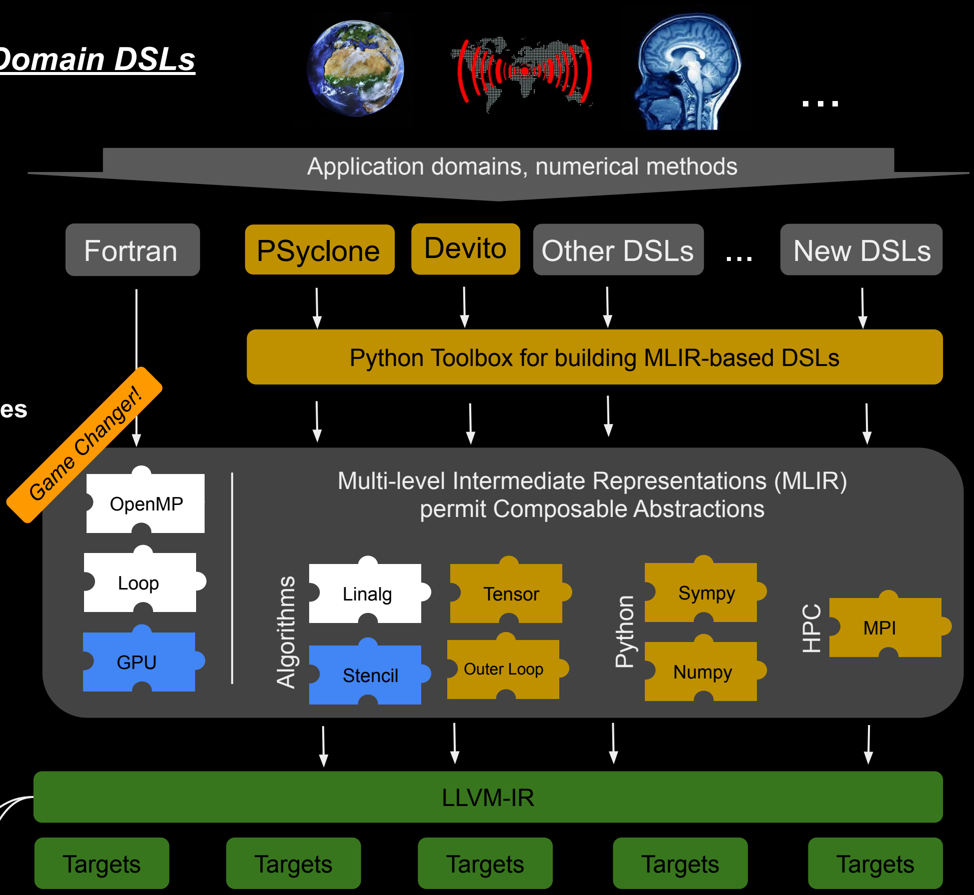
❌ *Disjoint Communities*
❌ *Lack of Knowledge Transfer*

Our Future: Cross-Domain DSLs

✅ Performance
✅ Productivity
✅ Portability

✅ Composability
✅ Interoperability
✅ Code Reuse
✅ Longevity

✅ Connected Communities
✅ Knowledge Transfer



Collaborative Support by Academics and Industry via LLVM Community

Debuggers

Profilers