Bridge Over Troubled Water: Offloading OpenMP Regions to AI Compiler via StableHLO/XLA

Institute of SCIENCE TOKYO

Muyao Xiao [1] Ivan R. Ivanov [1,2] Jens Domke [2] Toshio Endo [1]

[1] Institute of Science Tokyo [2] RIKEN Center for Computational Science (R-CCS)

RIKEN R-CCS

Problem Formulation: The Gap

To meet the intense computational demands of AI, manufacturers have developed specialized hardware, such as NVIDIA's Tensor Cores, to accelerate matrix operations.

However, despite sharing workload characteristics with AI, scientific computing has yet to fully leverage these advancements (or requires painful manual work and expertise).

This research aims to bridge the gap between Fortran-based scientific computing and AI-specialized hardware by introducing a novel compilation path that utilizes the XLA compiler.

Research Scope: The `workdistribute` Directive

Fortran natively supports array operations as first-class language constructs. For example, given arrays X, Y, Z of the same size n, and a scalar a. AXPY operation could be expressed in:

```
Z = a * X + Y
```

Listing 1. array like syntax

Traditionally, achieving parallelism required explicit do-loops rewriting within OpenMP:

```
!$omp target teams distribute parallel do
do i = 1, n
  Z(i) = a * X(i) + Y(i)
end do
```

Listing 2. OpenMP syntax

However, OpenMP 6.0 standard introduced the `workdistribute` directive, enabling automatic parallelism for array syntax[1]. Notice the following code is exactly the same with Listing 1 except for the OpenMP directives:

```
!$omp target teams workdistribute
Z = a * X + Y
!$omp end target teams workdistribute
```

Listing 3. workdistribute syntax

Taking Advantage of AI Compiler: The Era of Configurable Compiler Infrastructure

The modular design of the LLVM/MLIR framework [2], which powers the flang compiler, provides the flexibility to decouple the frontend from the backend.

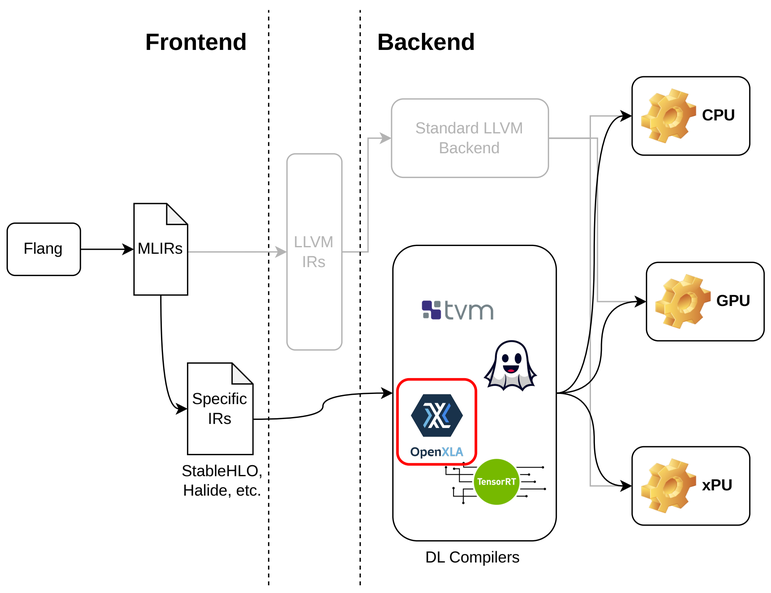

Figure 1. The Big Picture

This allows us to replace traditional backends with specialized AI compilers that already offer highly optimized code generation for GPUs and other accelerators.

System Design: Intercept Modern Flang Compilation and OpenMP Runtime

Our custom LLVM/MLIR pass, WorkdistributeJITPass, extracts workdistribute regions IR. They are then transformed into StableHLO and compiled JIT into a kernel binary for the target accelerator:

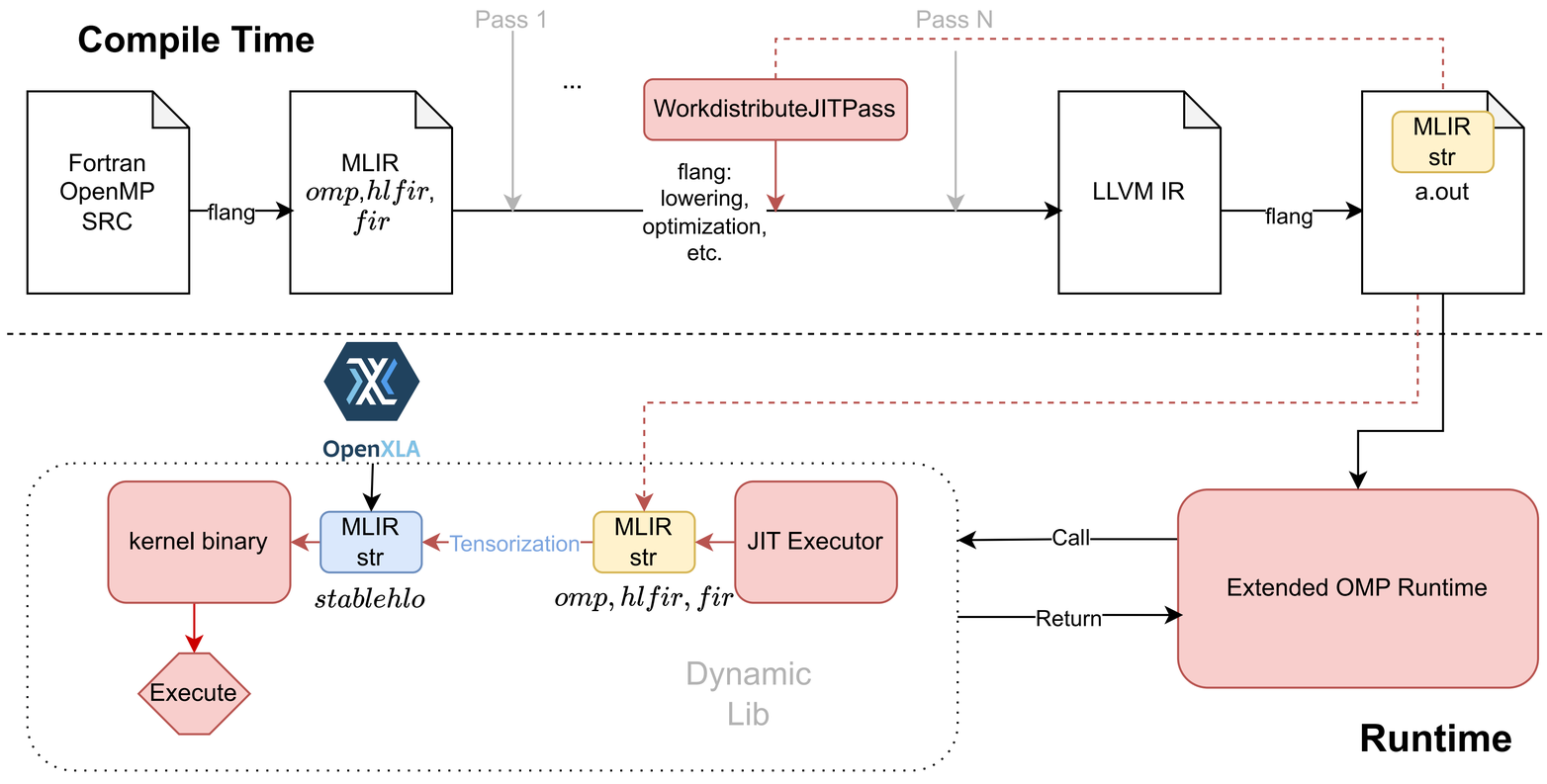

Figure 2. The Compilation Architecture

Tensorization: Translate from Fortran IRs to StableHLO

Memory related operations in Fortran IRs like `load, designate, apply` are transformed to inputs and outputs in StableHLO, leaving only compuations of tensors.

Tensorization

```
omp.workdistribute {
  %a = fir.load %148#0: f32
  %tmp = hlfir.elemental %149: %Xshape->expr<10xf32> {
  ^bb0(%i: index):
    %154 = hlfir.designate %X[%i]: ref<f32>
    %155 = fir.load %154: f32
    %156 = arith.mulf %151, %a: f32
    hlfir.yield_element %156: f32
  }
  %153 = hlfir.elemental %149: %Yshape->expr<10xf32> {
    ...
    %157 = arith.addf %154, %156: f32
    ...
  }
  hlfir.assign %153 to %Z: expr<10xf32>, ref<10xf32>
  ...
}
```

Lowering

Listing 4. AXPY function before *Tensorization*: omp hlfir fir

```
func.func @kernel(%Z: tensor<f32>, %a:
  tensor<f32>, %X: tensor<10xf32>, %Y:
tensor<10xf32>) {
  %0 = stablehlo.broadcast_in_dim %a:
(tensor<f32>) -> tensor<10xf32>
  %1 = stablehlo.multiply %X, %0:
tensor<10xf32>
  %2 = stablehlo.add %1, %Y: tensor<10
xf32>
  return %2, %a, %X, %Y: tensor<10xf32
>, tensor<f32>, tensor<10xf32>, tensor
<10xf32>
}
```

Listing 5. AXPY function after *Tensorization*: stableHLO

Preliminary Benchmark

We evaluated the above AXPY operations kernel execution time comparing **traditional Fortran OpenMP kernels** and **XLA kernels compiled from handwritten StableHLO** on an Nvidia A100 GPU:

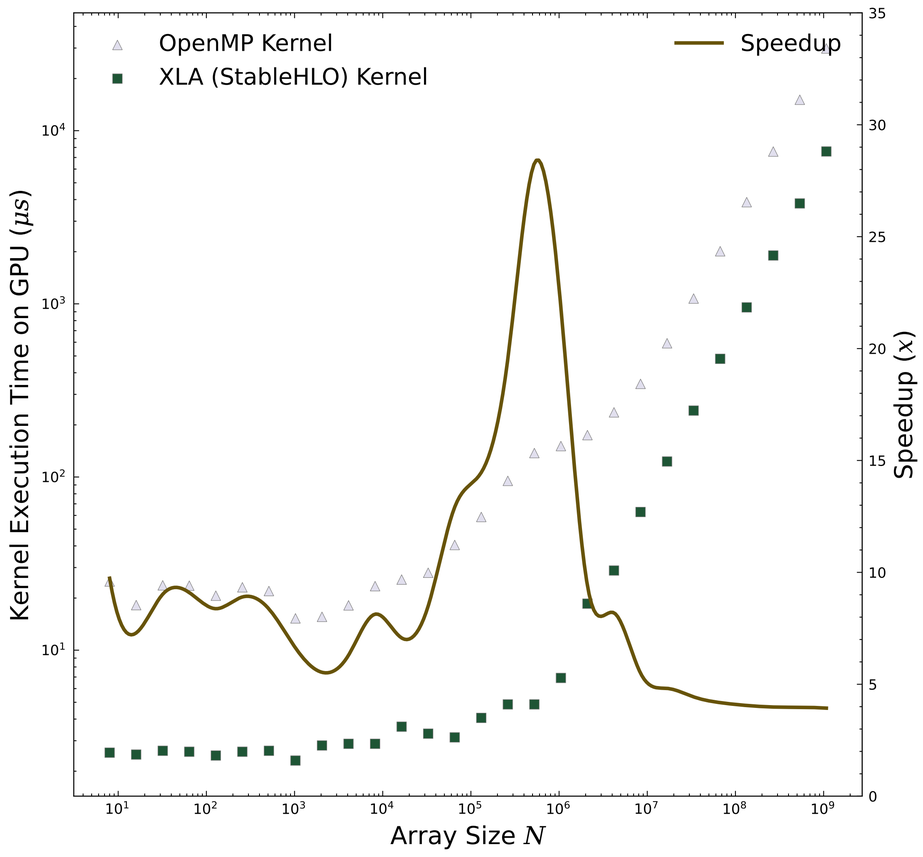

Figure 3. OpenMP Kernel vs. XLA kernel Peformance Comparison

Summary and Future Research

XLA kernels demonstrate significant speedup over the native OpenMP kernels, though total performance includes runtime overheads not fully captured by GPU profiling.

Our other Matrix multiplication tests also confirm the automatic generation of NVIDIA CUTLASS tensor instructions.

Future work includes further system integration and evaluation, mitigating JIT overhead and broadening syntax support.

[1] Ivanov et al. Automatic Parallelization and OpenMP Offloading of Fortran Array Notation. Cham: Springer Nature Switzerland, 2024. ISBN: 978-3-031-72567-8.

[2] Lattner and Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. CGO '04. Palo Alto, California: IEEE Computer Society, 2004. ISBN: 0769521029.