MLIR-RAJA: Bridging AI Models and HPC Performance Portability

Tai-Hsiang Peng, Chun-Lin Huang, Hung-Ming Lai, Wei-Shen Huang, Jenq-Kuen Lee

Department of Computer Science

National Tsing Hua University, Taiwan

Outline

- Motivation: Bridging AI and HPC
- Background: The RAJA Portability Suite
- Proposed MLIR-RAJA Dialect and Pipeline
- Applications: PyTorch MLP and Stencil Computations
- Experimental Results and Performance Analysis
- Future Work and Conclusion

Motivation:
Bridging the Gap between HPC and AI

AI Scientist writes	HPC Engineer needs
model = resnet18()	200+ lines of RAJA C++
output= model(input)	Views, Layouts, Kernel Policy, …
3 lines of Pytorch	Manual rewrite for every operator

- The gap: Every new model or architecture change requires repeating this manual effort.
- Our Solution: Automated compilation via MLIR
  - **MLIR-RAJA Dialect:** Represents RAJA loops and execution policies directly in MLIR, enabling IR-level optimization before code generation.
  - **End-to-End Flow:** An automated pipeline translating AI models directly into optimized RAJA C++.
  - Generated code is portable across all RAJA-supported backends.

PL NTHU LAB

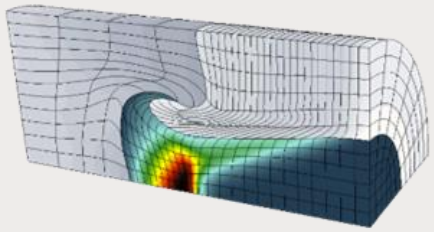
*RAJA is developed by Lawrence Livermore National Laboratory (LLNL).
Reference: David A. Beckingsale, et al. "RAJA: Portable Performance for Large-Scale Scientific Applications." In 2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC), 2019.

Introduction:
The RAJA Portability Suite in HPC

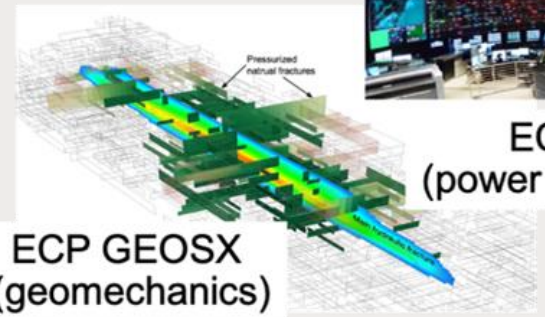
- A Cornerstone of High-Performance Computing:
  - A key technology for the Exascale Computing Project (ECP).
  - Serves as the standard execution layer for next-generation scientific simulations.
  - Proven scalability on world-class supercomputers (e.g., Frontier, Aurora, El Capitan).
- The Portability Suite Ecosystem:
  - RAJA: Manages loop execution and parallelism policies.
  - Umpire: Manages complex memory hierarchies.
  - CHAI: Handles data movement strategies.

RAJA Portability Suite:
Enabling Scientific Applications across Diverse Architectures

ECP apps using RAJA software tools

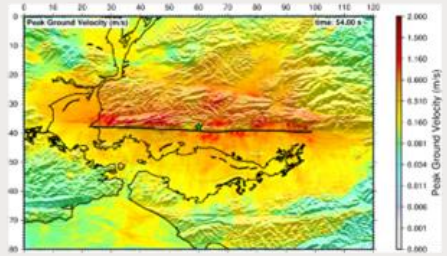


LLNL ECP/ATDM
(high-order ALE hydro)



ECP GEOSX
(geomechanics)

ECP ExaSGD
(power grid optimization)



ECP SW4
(earthquake modeling)

plus others…

RAJA / Umpire / CHAI

Perlmutter (LBL)
AMD Milan CPUs +
NVIDIA Ampere GPUs

Astra (SNL)
ARM architecture

Sierra (LLNL)
IBM P9 CPUs + NVIDIA Volta GPUs

Aurora (ANL)
Intel Xeon CPUs + Xe GPUs

Frontier (ORNL) &
El Capitan (LLNL)
AMD CPUs + GPUs

Source: https://computing.llnl.gov/projects/raja-managing-application-portability-next-generation-platforms

Introduction to RAJA: A C++ Performance Portability Layer for HPC

- Developed by **Lawrence Livermore National Laboratory (LLNL)**.
- An **open-source**, **header-only C++ template library**.
- Enables **Single-Source** application code bases that are portable across platform generations.

```
RAJA::forall<EXEC_POLICY>(range, [=] (int i) {
   A[i] = B[i] + C[i];
});
```

Execution Policy

```
RAJA::forall<seq_exec>(range, [=] (int i) {
   A[i] = B[i] + C[i];
});
```

```
RAJA::forall<omp_parallel_for_exec>(range, [=] (int i) {
   A[i] = B[i] + C[i];
});
```

```
RAJA::forall<simd_exec>(range, [=] (int i) {
   A[i] = B[i] + C[i];
});
```

PL NTHU LAB

Why Emit RAJA C++ from MLIR?

- Human-readable, maintainable output
  - Generated code is standard RAJA C++ — not a black box.
  - HPC developers can inspect, modify, and integrate into existing codebases.

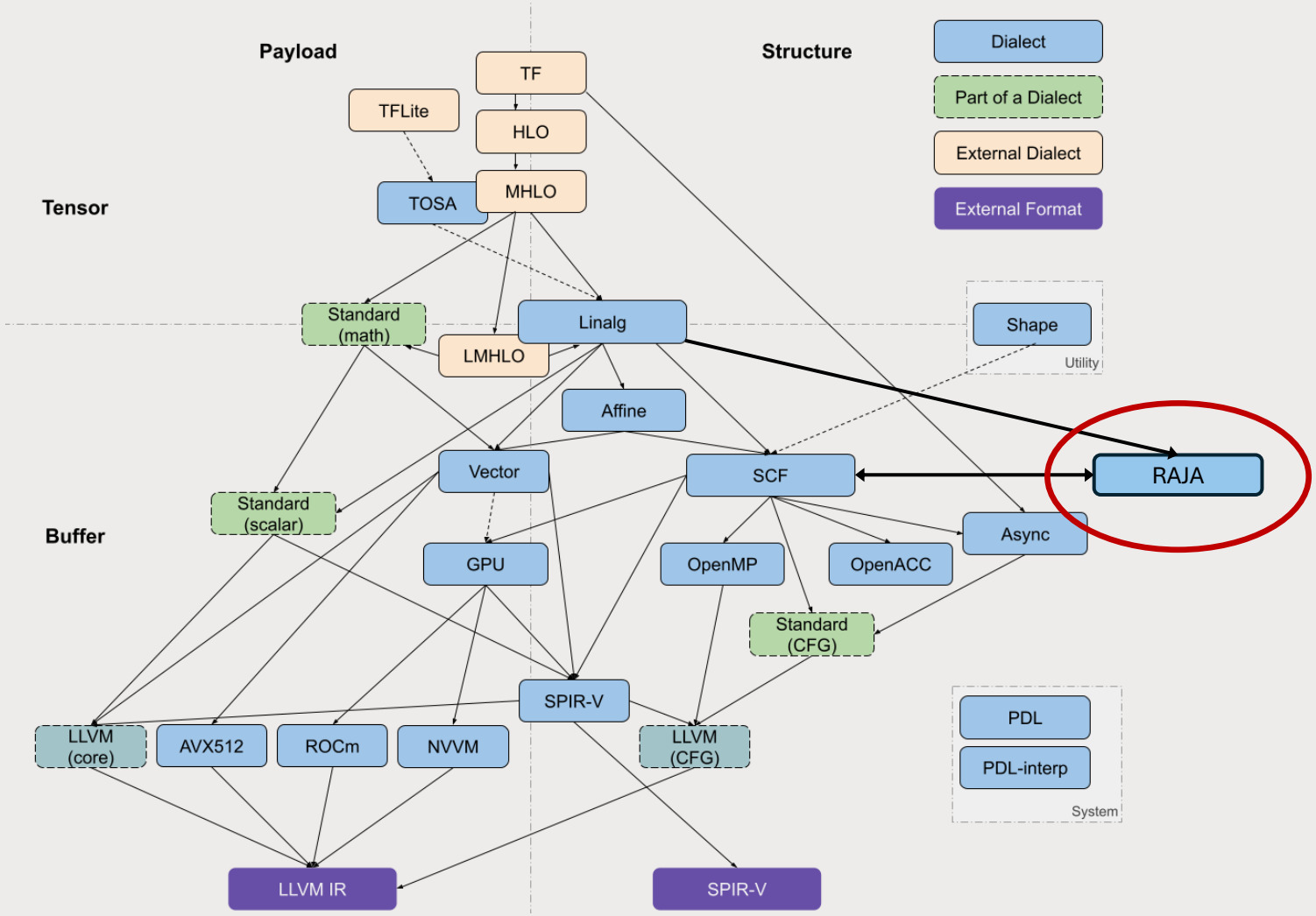
- MLIR-level optimization before code generation.
  - Apply tiling, loop fusion, and transformations at the IR level.
  - Optimizations that are difficult to express in C++ templates.

Why Emit RAJA C++ from MLIR? (cont'd)

- RAJA's portability: one codegen, multiple backends.
  - Same generated code targets CPU, OpenMP, CUDA, HIP, SYCL — just change the execution policy.
  - Avoids writing separate MLIR backends (NVVM, ROCm, SPIR-V…) for each platform.

- Broad frontend compatibility via MLIR ecosystem.
  - Any framework that lowers to Linalg / SCF can feed into this pipeline.
  - Supports PyTorch, ONNX, TensorFlow, and more.

The Abstraction Level of Our Proposed RAJA Dialect at MLIR



Our work!

The RAJA Dialect Design

- Goal: Represent RAJA constructs structurally in MLIR.
- Key Operation: raja.forall
  - Carries execution policy.
  - Defines loop bounds and induction variables.

Execution Policy	Backend
seq_exec	Sequential CPU
simd_exec	SIMD vectorization
omp_parallel_for_exec	OpenMP parallel

```
// Sequential
raja.forall { #raja.policy<seq_exec> }
  %i = %c0 to %c256 type "index" {
    // loop body
}

 // OpenMP parallel — same code, different policy
raja.forall { #raja.policy<omp_parallel_for_exec> }
  %i = %c0 to %c256 type "index" {
    // loop body
}
```

RAJA Dialect

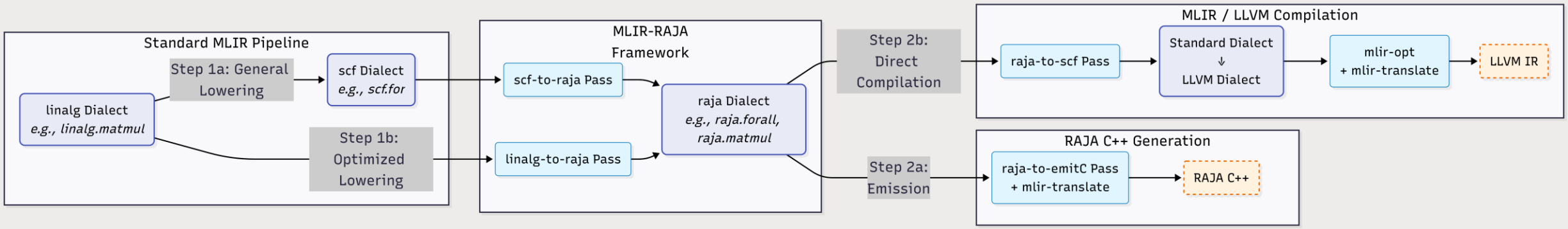
```
// Sequential
RAJA::forall<RAJA::seq_exec>(
  RAJA::RangeSegment(0, 256), [=](size_t i) {
    // loop body
});

// OpenMP parallel — same code, different policy
RAJA::forall<RAJA::omp_parallel_for_exec>(
  RAJA::RangeSegment(0, 256), [=](size_t i) {
    // loop body
});
```

RAJA C++

Our MLIR-RAJA Pipeline

- Step 1: Lower Linalg / SCF operations to the RAJA Dialect with execution policy assignment.
- Step 2: Lower the RAJA Dialect to target backends (RAJA C++ or LLVM IR).



The MLIR-RAJA Pipeline

Step1a: General Lowering (The SCF Path)

- Target:
  - linalg.generic
  - linalg.conv_2d_...

```
linalg.generic {indexing_maps = [#map, #map, #map], iterator_types = ["parallel", "parallel"]}
ins(%arg0, %arg1 : memref<4x5xf32>, memref<4x5xf32>) outs(%arg2 : memref<4x5xf32>) {
^bb0(%in: f32, %in_0: f32, %out: f32):
  %0 = arith.addf %in, %in_0 : f32
  linalg.yield %0 : f32
}
```

Linalg Dialect

- Conversion Passes:
  - Lower Linalg operations to SCF loops.
  - Apply custom the scf-to-raja pass.
    - Maps SCF loops to raja.forall operations.

RAJA Dialect general lowering path

```
raja.forall { #raja.policy<seq_exec> } %arg3 = %c0 to %c4 type "index"{
  raja.forall { #raja.policy<seq_exec> } %arg4 = %c0 to %c5 type "index"{
    %0 = memref.load %arg0[%arg3, %arg4] : memref<4x5xf32>
    %1 = memref.load %arg1[%arg3, %arg4] : memref<4x5xf32>
    %2 = arith.addf %0, %1 : f32
    memref.store %2, %arg2[%arg3, %arg4] : memref<4x5xf32>
    raja.yield
  }
  raja.yield
}
```

RAJA Dialect

- Assigns RAJA Execution Policy.

PL NTHU LAB

Step1b: Optimized Lowering (The RAJA Abstraction-Aware Lowering Path)

- Target:
  - linalg.matmul
- General lowering often misses optimization opportunities.
- Preserves Semantics:
  - Retains the matrix multiplication structure.
  - Prevents early conversion into scalar loops.
- Enables Optimizations:
  - Allows structure-aware techniques like **Tiling** and **Loop Transformation**.
  - Generates High-Efficiency RAJA C++ with better cache locality.

```
%0 = linalg.matmul ins(%arg0, %arg1 : tensor<4096x4096xf32>, tensor<4096x4096xf32>)
     outs(%arg2 : tensor<4096x4096xf32>) -> tensor<4096x4096xf32>
                    Linalg Dialect
```

RAJA Abstraction-Aware lowering path

```
%0 = raja.matmul ins(%arg0, %arg1 : tensor<4096x4096xf32>, tensor<4096x4096xf32>)
     outs(%arg2 : tensor<4096x4096xf32>) -> tensor<4096x4096xf32>
                    RAJA Dialect
```

PL NTHU LAB

Step 2a: Emission to RAJA C++

- Utilizes EmitC verbatim to model C++ structure within MLIR.
  - Maps raja.forall to generated RAJA loop templates and lambda syntax.
  - Maps raja.matmul to high performance RAJA C++ kernel.

- Uses mlir-translate to generate the final .cpp file.
  - Includes all necessary RAJA headers.

```
raja.forall { #raja.policy<seq_exec> }
%arg = %c0 to %c5 type "index"{
  // computation
}
```

RAJA Dialect

```
verbatim "RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment({}, {}), [=](size_t i0) {{"
args %c0 to %c5 : !emitc.size_t, !emitc.size_t
  %arg5 = "emitc.constant"() <{value = #emitc.opaque<"i0">}> : () -> !emitc.size_t
  // computation
verbatim "});“
```

EmitC Dialect Code Segment

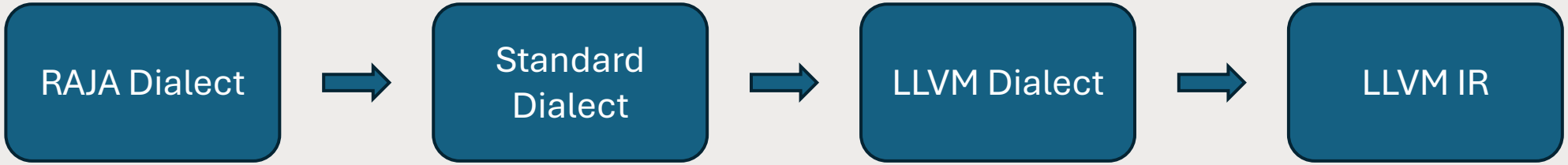
mlir-translate

```
RAJA::forall<RAJA::seq_exec>(
RAJA::RangeSegment(c0, c5), [=](size_t i0) {
  size_t arg5 = i0;
  // computation
});
```

RAJA C++ Code Segment

Step 2b: Direct Compilation to LLVM IR

- Lowering flow:




```
raja.forall { #raja.policy<seq_exec> } %arg3 = %c0 to %c4 type "index"{
  raja.forall { #raja.policy<seq_exec> } %arg4 = %c0 to %c5 type "index"{
    %0 = memref.load %arg0[%arg3, %arg4] : memref<4x5xf32>
    %1 = memref.load %arg1[%arg3, %arg4] : memref<4x5xf32>
    %2 = arith.addf %0, %1 : f32
    memref.store %2, %arg2[%arg3, %arg4] : memref<4x5xf32>
    raja.yield
  }
  raja.yield
}
```

RAJA Dialect

mlir-opt

+

mlir-translate

```
43:   ; Outer loop
  %44 = phi i64 [ %68, %67 ], [ 0, %21 ]
  %45 = icmp slt i64 %44, 4
  br i1 %45, label %46, label %69
46:
  br label %47
47:   ; Inner loop
  %48 = phi i64 [ %66, %50 ], [ 0, %46 ]
  %49 = icmp slt i64 %48, 5
  br i1 %49, label %50, label %67
50:
  ; computation (load / add / store)
  %66 = add i64 %48, 1
  br label %47
67:
  %68 = add i64 %44, 1
  br label %43
```

LLVM IR Code Segment

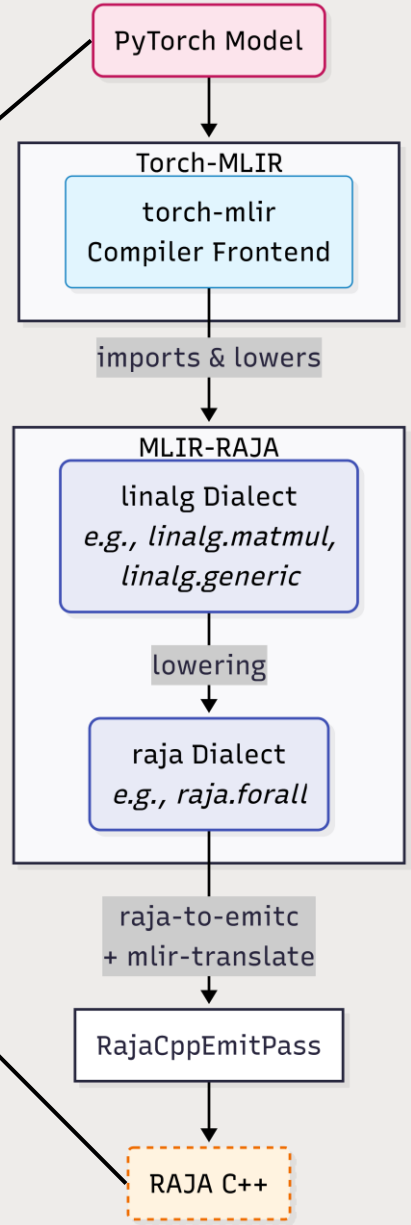
Application: PyTorch MLP Model

- Model: PyTorch SimpleMLP

```
self.fc1 = nn.Linear(8, 16, bias=False)
self.relu = nn.ReLU()
self.fc2 = nn.Linear(16, 4, bias=False)
```
Pytorch

- No tiling applied due to small matrix sizes.
- Embedded weights are not yet supported; weights are passed as function arguments.

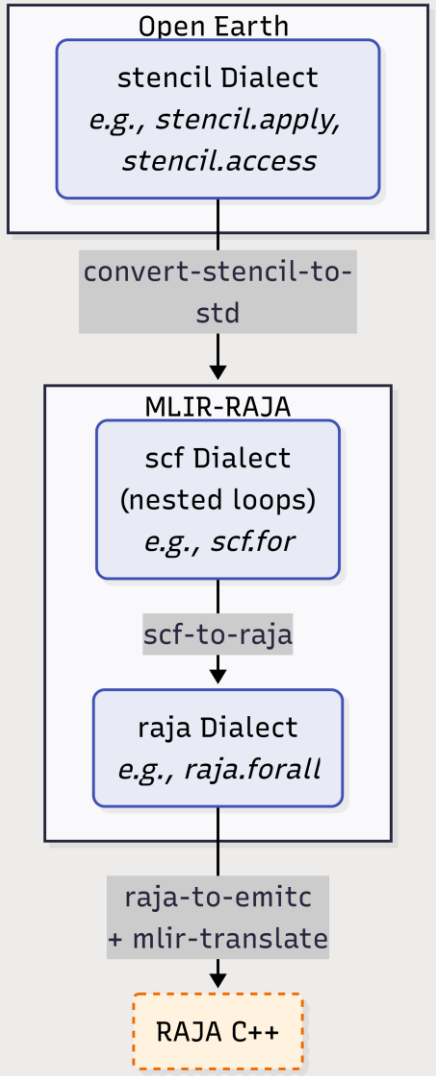
```
// Matmul 1: input[1,8] x W1[8,16] → hidden[1,16]
RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(0, 16), [=](size_t i1) {
  RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(0, 8), [=](size_t i2) {
    // Matmul computation
}); });
// ReLU
RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(v9, v11), [=](size_t i3) {
  //ReLU computation
});
// Matmul 2: hidden[1,16] x W2[16,4] → output[1,4]
RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(v9, v6), [=](size_t i5) {
  RAJA::forall<RAJA::seq_exec>(RAJA::RangeSegment(v9, v11), [=](size_t i6) {
    // Matmul computation
}); });
```
RAJA C++



Application: Climate Simulation Stencil via Open Earth Compiler

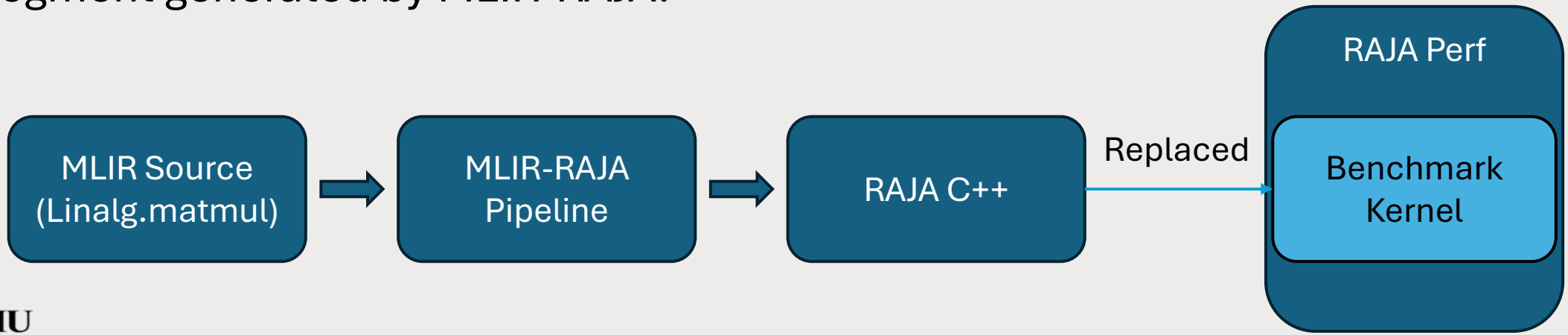
- Open Earth Compiler (OEC): An MLIR-based compiler for weather and climate simulations.
- It defines a stencil dialect for structured grid computations commonly used in atmospheric modeling.

→ Any MLIR dialect that lowers to SCF can feed into MLIR-RAJA — not limited to AI models.



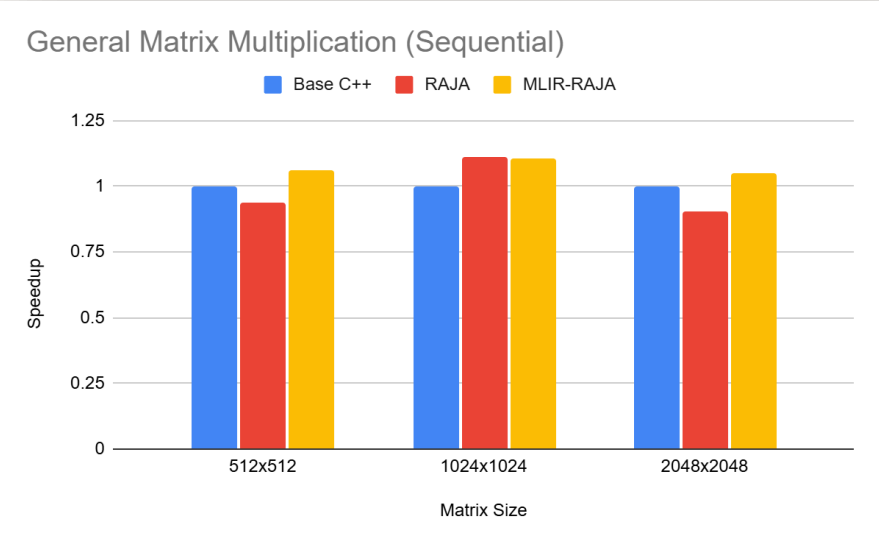
Experimental Setup for RAJA C++

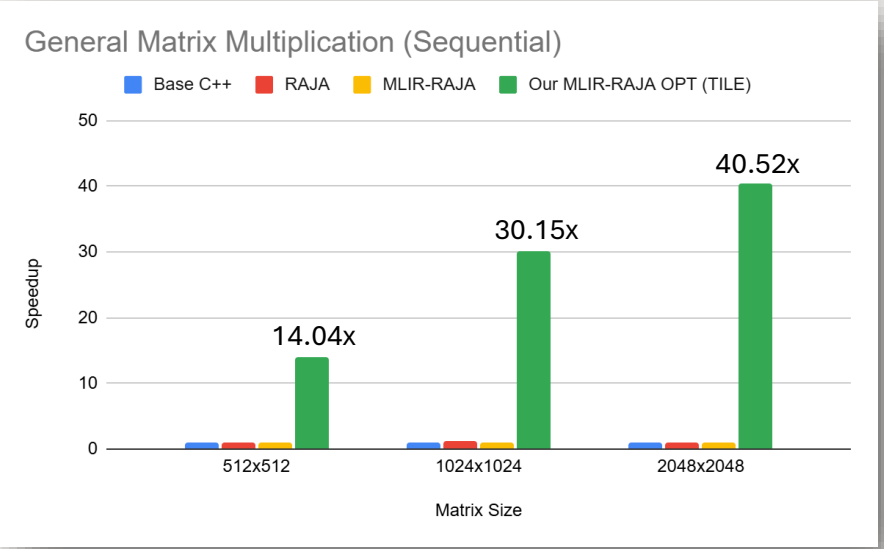
- Hardware Environment:
  AMD Ryzen 9 9950X 16-Core Processor (x86_64)
- Benchmark Framework:
  RAJAPerf Suite (Kernel: General Matrix Multiplication)
- The original benchmark kernel in RAJAPerf was replaced with the code segment generated by MLIR-RAJA.



Experimental Results for RAJA C++ (Sequential)

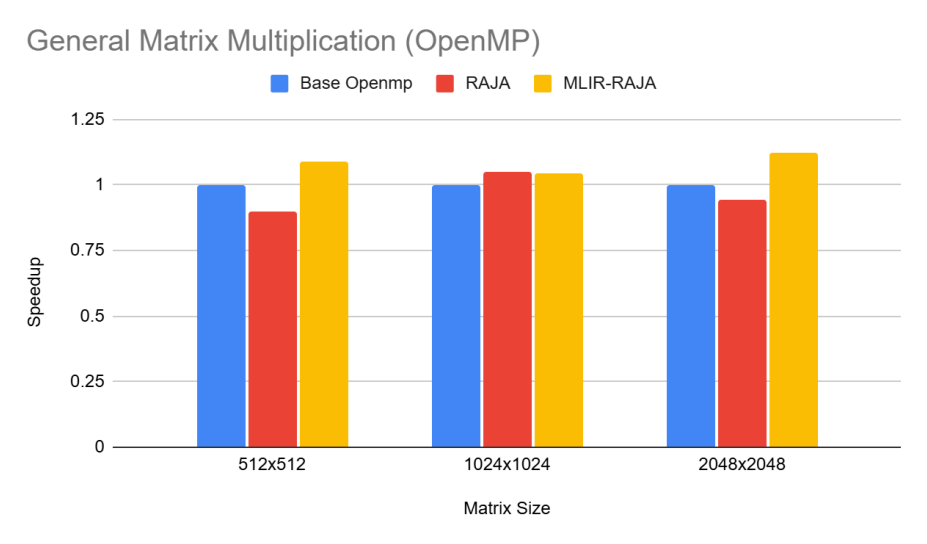
- Evaluated Variant:
  - **Base C++ (Baseline):** Baseline sequential C++ implementation.
  - **RAJA:** Hand-written RAJAC++ implementation.
  - **MLIR-RAJA:** Code auto-generated by our **General SCF Path** (Step 1a).
  - **Our MLIR-RAJA OPT (TILE):** Code auto-generated by our **RAJA Abstraction-Aware Lowering Path** (Step 1b).

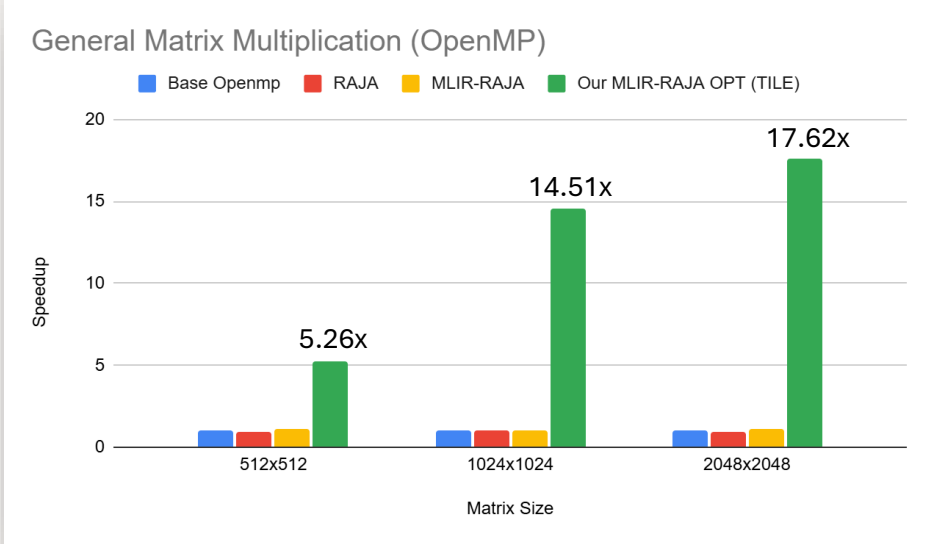




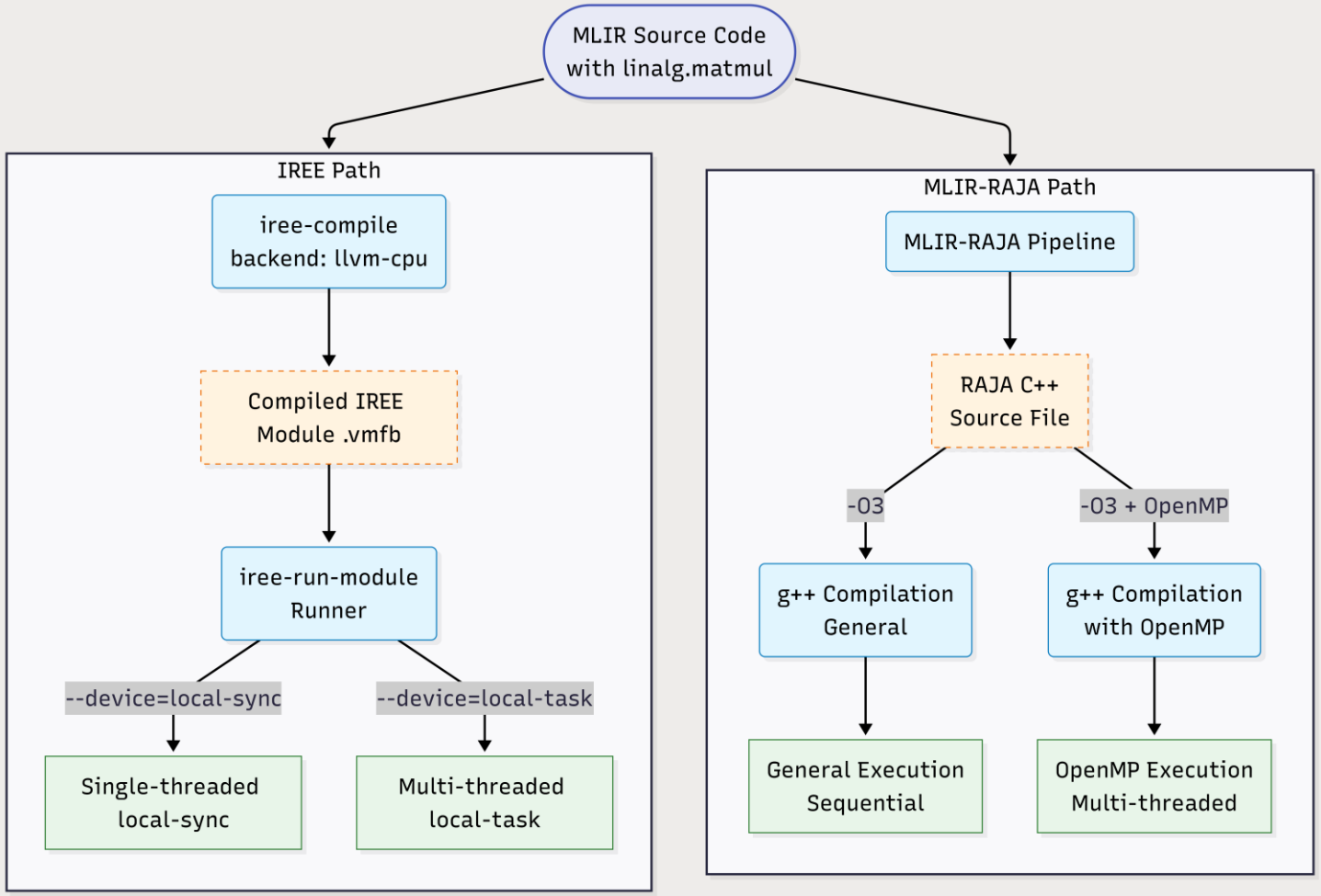
Experimental Results for RAJA C++ (OpenMP)

- Evaluated Variant:
  - **Base OpenMP (Baseline):** Baseline OpenMP implementation.
  - **RAJA:** Hand-written RAJA C++ implementation.
  - **MLIR-RAJA:** Code auto-generated by our **General SCF Path** (Step 1a).
  - **Our MLIR-RAJA OPT (TILE):** Code auto-generated by our **RAJA Abstraction-Aware Lowering Path** (Step 1b).



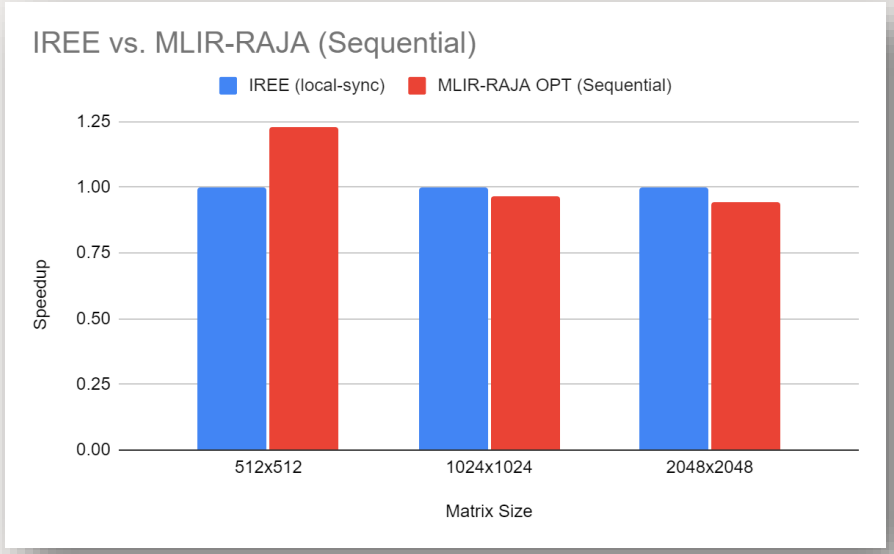


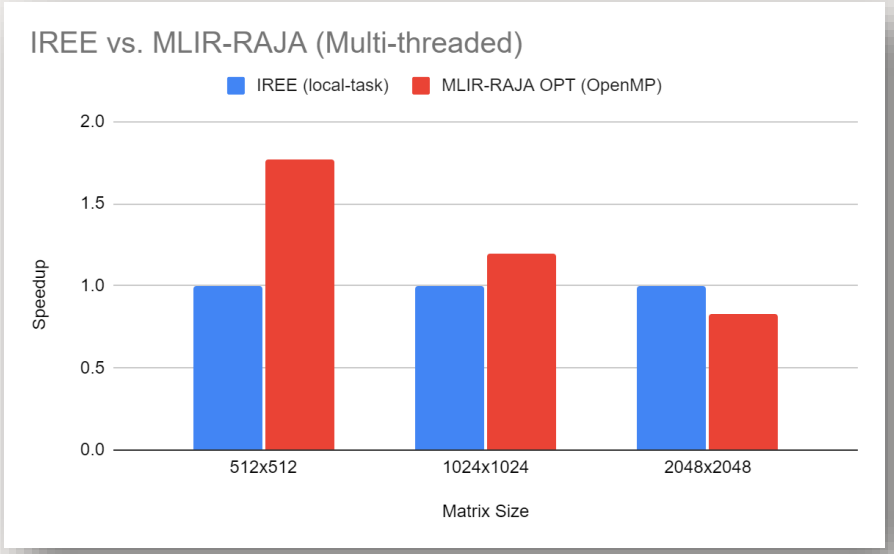
Comparison with IREE — Experimental Setup



Comparison with IREE — Results

- MLIR-RAJA generates competitive performance with IREE.
- At small sizes, MLIR-RAJA wins due to lower runtime overhead.
- At large sizes, IREE's multi-threaded runtime has an advantage.





Current Limitations

- Fixed execution policy assignment:
  - The scf-to-raja pass currently assigns seq_exec to all loops by default.
  - Other execution policies require manual policy modification.
- CPU-only execution:
  - Currently supports sequential and OpenMP CPU backends.
  - No GPU execution policies (CUDA, HIP, SYCL) support yet.
- Static shapes only:
  - All tensor dimensions must be known at compile time.
  - Dynamic shapes are not yet handled in the MLIR-RAJA pipeline.

Future Work

- Automatic execution policy selection:
  - Analyze loop characteristics to determine optimal policy assignment.
  - Extend supported policies to CUDA, HIP, and SYCL backends.
- Auto-tuning integration:
  - Parameterized tile size selection based on target architecture.
  - Explore MLIR transform dialect for automated performance tuning.
- Full RAJA Portability Suite integration
  - Umpire: memory management across CPU and GPU memory hierarchies.
  - CHAI: automatic data movement between host and device.

PL NTHU LAB

Conclusion

- Defined a **RAJA Dialect** to embed HPC semantics into MLIR.
- Built an end-to-end flow from AI Models and scientific computations to RAJA C++.
- Achieved improvements compared to baseline C++:
  - Sequential: **14.04x** improvement with tiling.
  - OpenMP: **5.26x** improvement with tiling.
- Generated code is competitive with IREE while producing human-readable, portable RAJA C++.

Thank you for listening.