High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs in Polygeist/MLIR

William S. Moses (MIT)
Ivan R. Ivanov (Tokyo Tech)
Jens Domke (Riken)
Toshio Endo (Tokyo Tech)
Johannes Doerfert (ANL)
Oleksandr Zinenko (Google)

Aim: CUDA software on CPU

Optimization & transformation

Use cases of GPU to CPU compilation

Reuse of existing GPU software on CPU-only machines

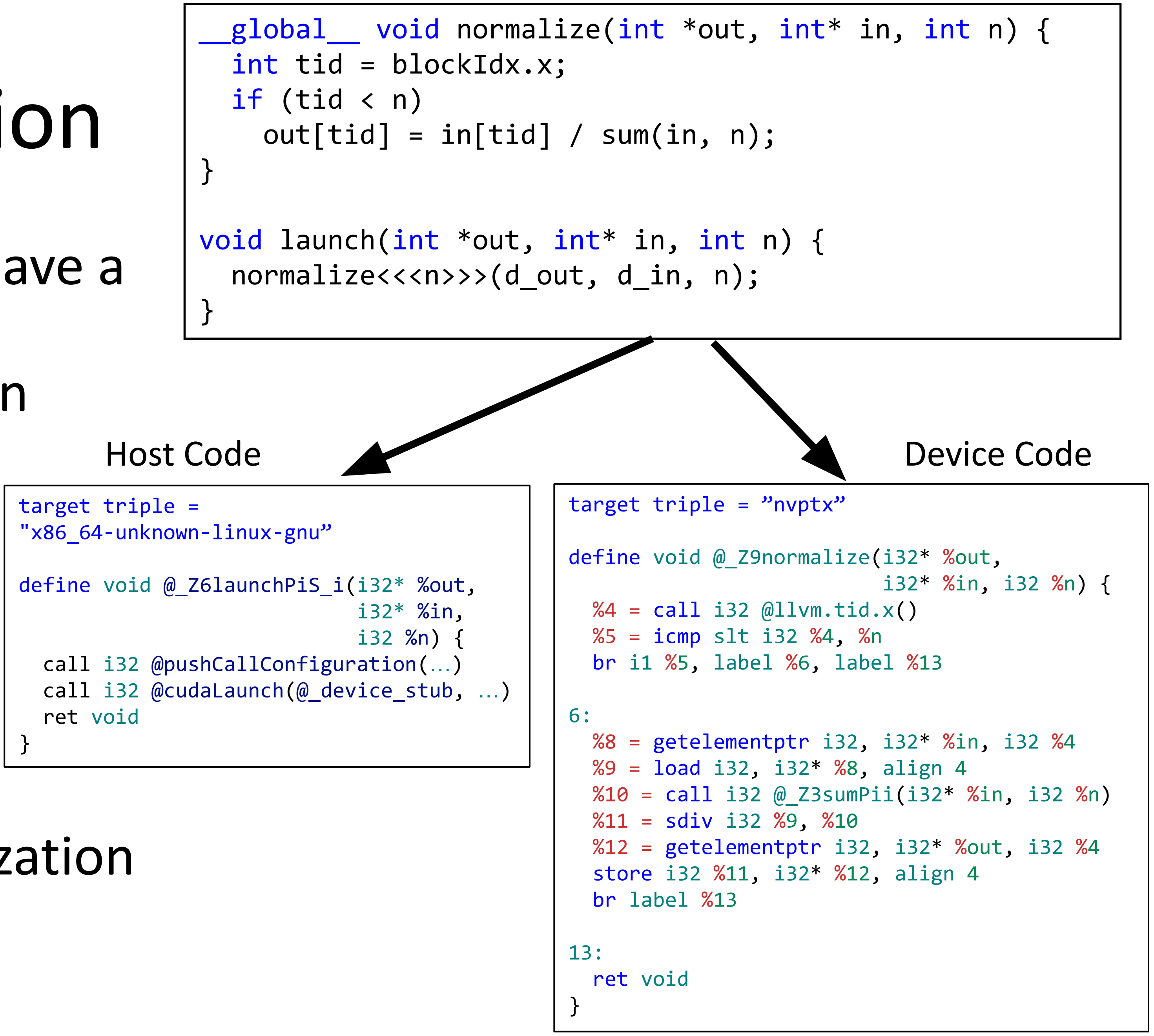
- Lower development/porting cost and time
- Leverage high parallelism

  e.g. running Pytorch on the Fugaku supercomputer

Debugging

GPU code compilation

- Mainstream compilers do not have a high-level representation of parallelism, making optimization difficult or impossible
- This is accentuated for GPU programs where the kernel is kept in a separate module to allow emission of different assembly and synchronization is treated as a complete optimization barrier.

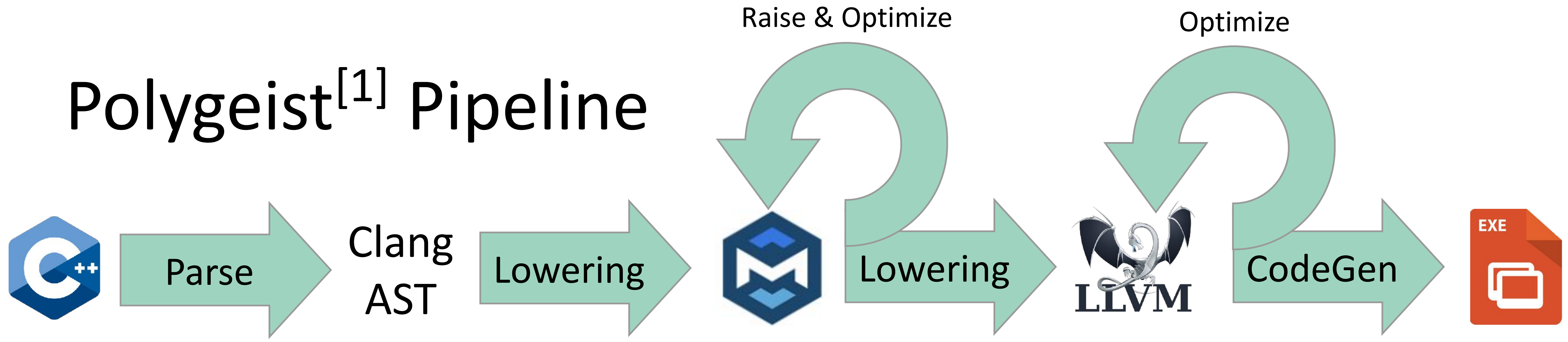


```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
target triple =
"x86_64-unknown-linux-gnu”

define void @_Z6launchPiS_i(i32* %out,
                            i32* %in,
                            i32 %n) {
  call i32 @pushCallConfiguration(…)
  call i32 @cudaLaunch(@_device_stub, …)
  ret void
}
```

```
target triple = ”nvptx”

define void @_Z9normalize(i32* %out,
                          i32* %in, i32 %n) {
  %4 = call i32 @llvm.tid.x()
  %5 = icmp slt i32 %4, %n
  br i1 %5, label %6, label %13

6:
  %8 = getelementptr i32, i32* %in, i32 %4
  %9 = load i32, i32* %8, align 4
  %10 = call i32 @_Z3sumPii(i32* %in, i32 %n)
  %11 = sdiv i32 %9, %10
  %12 = getelementptr i32, i32* %out, i32 %4
  store i32 %11, i32* %12, align 4
  br label %13

13:
  ret void
}
```

Polygeist[1] Pipeline



- Generic C or C++ frontend that generates "standard" and user-defined MLIR
- Raising transformations for raising "standard" MLIR to high-level
- Collection of high-level optimization passes (general mem2reg, parallel optimizations)
- Polyhedral optimization via novel optimizations and integrating prior tools (Pluto, CLooG) into MLIR
- Parallel/GPU optimizations & transformations

[1] Polygeist: Raising C to Polyhedral MLIR; Moses, Chelini, Zhao, and Zinenko. PACT '21.

Preserve the parallel structure

- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

$\longrightarrow$

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index

  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]
    %sum = call @_Z3sumPii(%in, %n)
    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

Preserve the parallel structure

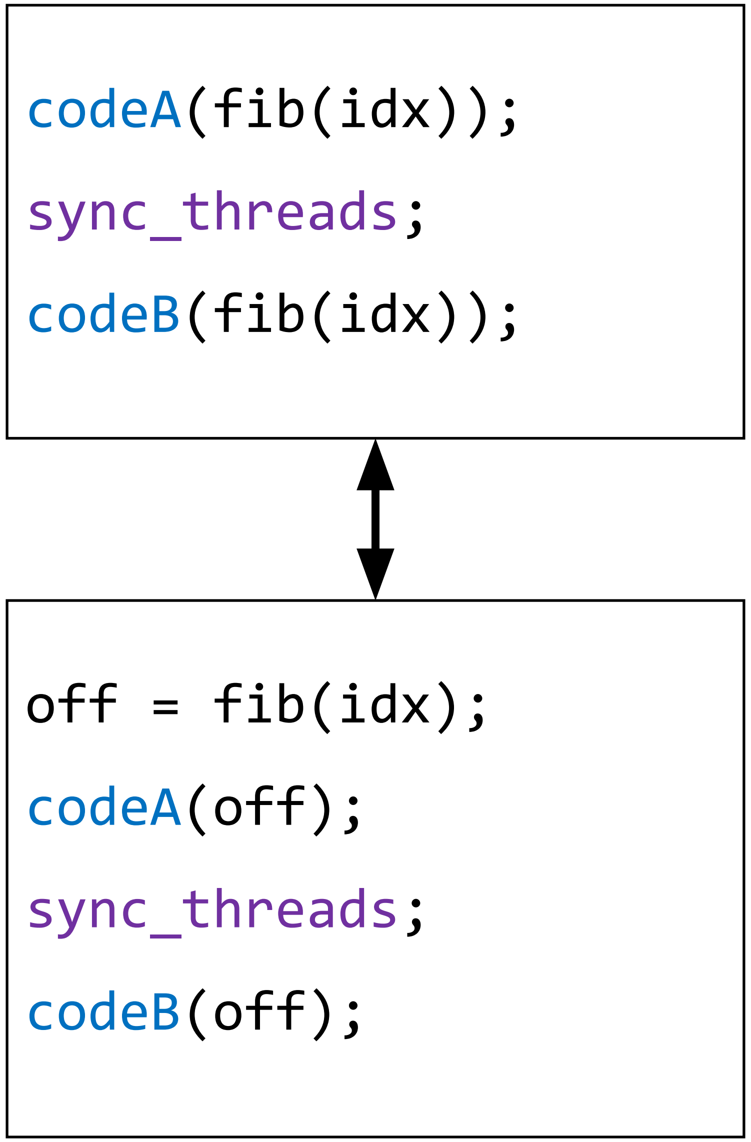
- Maintain GPU parallelism in a form understandable to the compiler
- Enables optimization between caller and kernel
- Enable parallelism-specific optimization

```
__global__ void normalize(int *out, int* in, int n) {
  int tid = blockIdx.x;
  if (tid < n)
    out[tid] = in[tid] / sum(in, n);
}

void launch(int *out, int* in, int n) {
  normalize<<<n>>>(d_out, d_in, n);
}
```

```
func @_Z6launch(%out: memref<?xi32>,
                %in: memref<?xi32>, %n: i32) {
  %c1 = constant 1 : index
  %c0 = constant 0 : index
  %sum = call @_Z3sumPii(%in, %n)
  parallel (%tid) = (%c0) to (%n) step (%c1) {
    %2 = load %in[%tid]

    %4 = divsi %2, %sum : i32
    store %4, %out[%tid]
    yield
  }
  return
}
```

Synchronization via Memory

- Synchronization (sync_threads) ensures all threads within a block finish executing codeA before executing codeB
- The desired synchronization behavior can be reproduced by defining sync_threads to have the union of the memory semantics of the code before and after the sync.
- This prevents code motion of instructions which require the synchronization for correctness, but permits other code motion (e.g. index computation).



```
codeA(fib(idx));
sync_threads;
codeB(fib(idx));
```

```
off = fib(idx);
codeA(off);
sync_threads;
codeB(off);
```

Synchronization via Memory

- High-level synchronization representation enables new optimizations, like sync elimination.
- A synchronize instruction is not needed if the set of read/writes before the sync don’t conflict with the read/writes after the sync.

```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];

  if ( tx == 0 )
    node[ty] = input[index_in] ;

  // Unnecessary Barrier #1
  // None of the read/writes below the sync
  //  (weights, hidden)
  // intersect with the read/writes above the sync
  //  (node, input)
  __syncthreads();

  weights[ty][tx] = hidden[index];

  __syncthreads();

  …

}
```

GPU Transpilation

- A unified representation of parallelism enables programs in one parallel architecture (e.g. CUDA) to be compiled to another (e.g. CPU/OpenMP)
- Most CPU backends do not have an equivalent block synchronization
- Efficiently lower a top-level synchronization by distributing the parallel for loop around the sync, and interchanging control flow

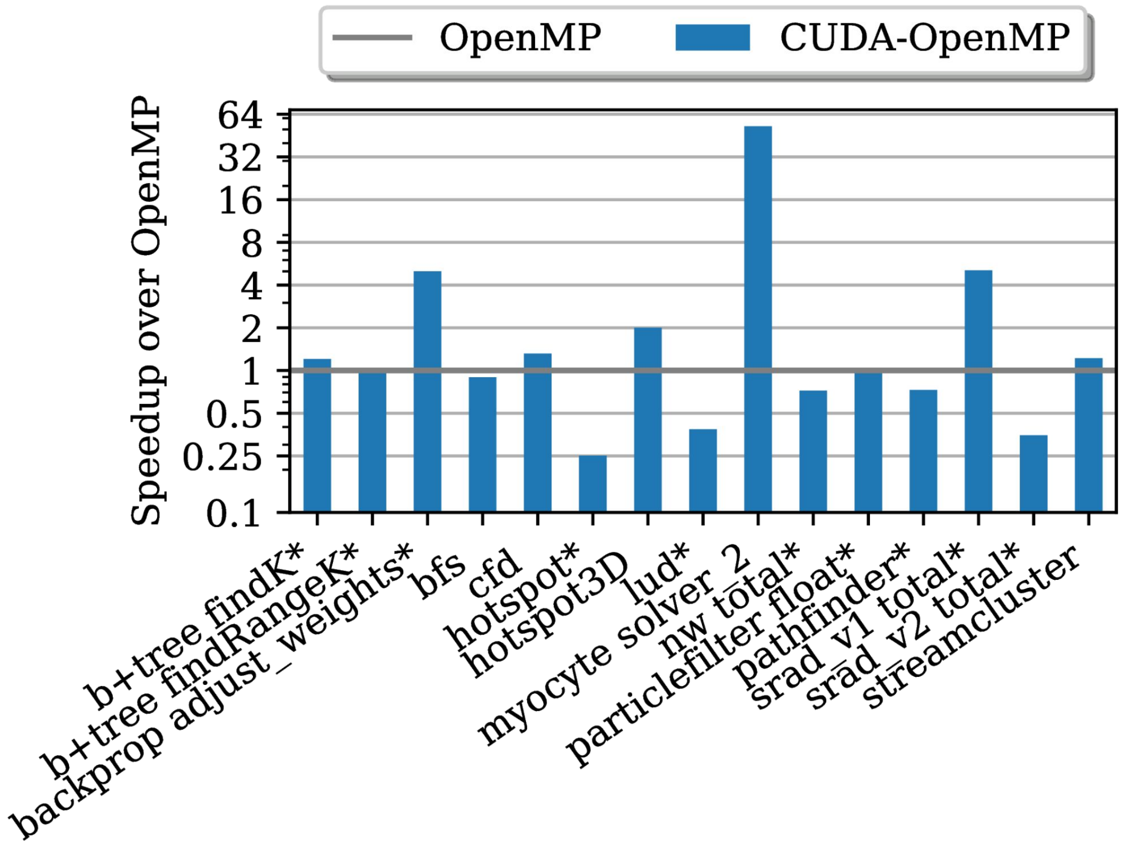
```
parallel_for %i = 0 to N {
  codeA(%i);
  sync_threads;
  codeB(%i);
}
```

$\longrightarrow$

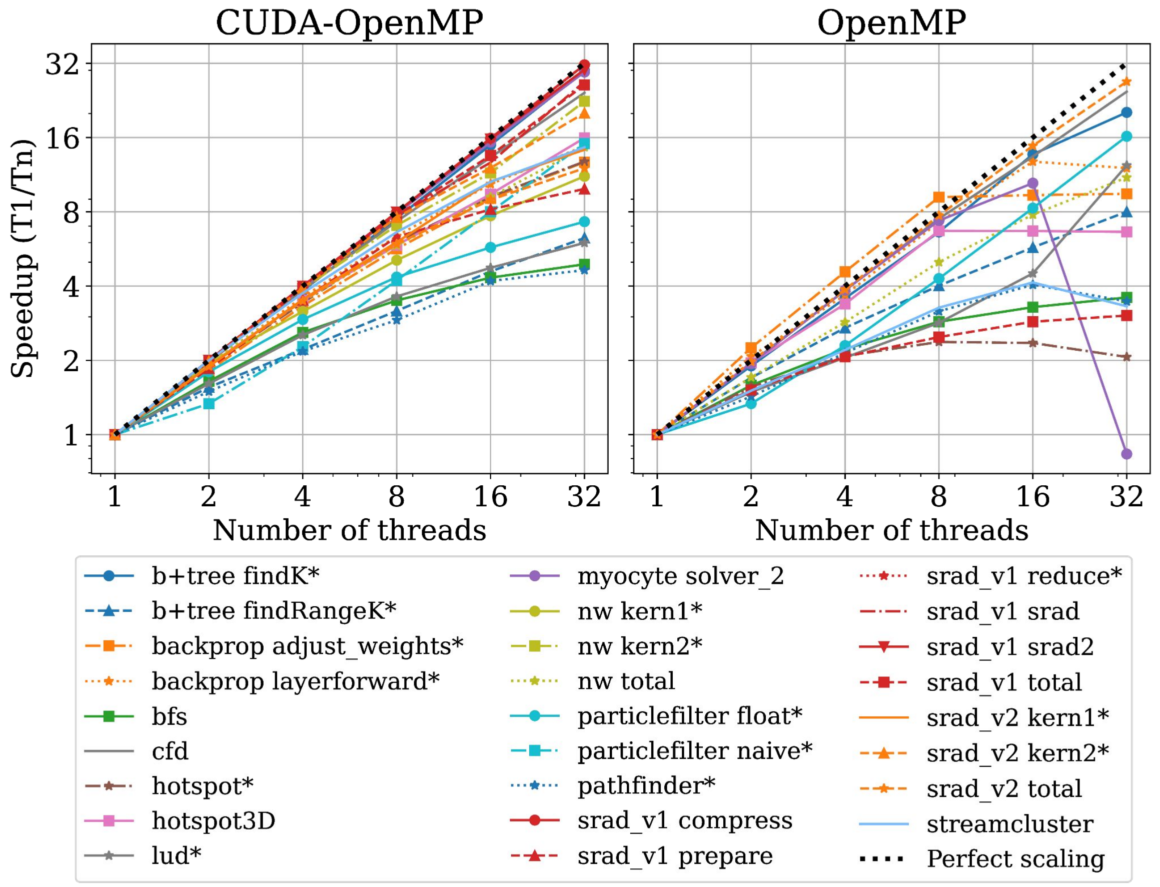
```
parallel_for %i = 0 to N {
  codeA(%i);
}
parallel_for %i = 0 to N {
  codeB(%i);
}
```

Evaluation 1: Rodinia benchmark suite

Better performance over hand-written OpenMP! (1.7x)

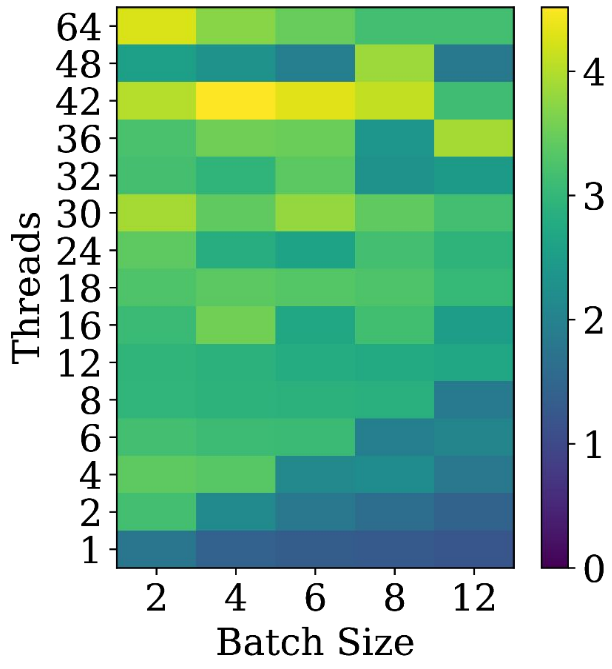


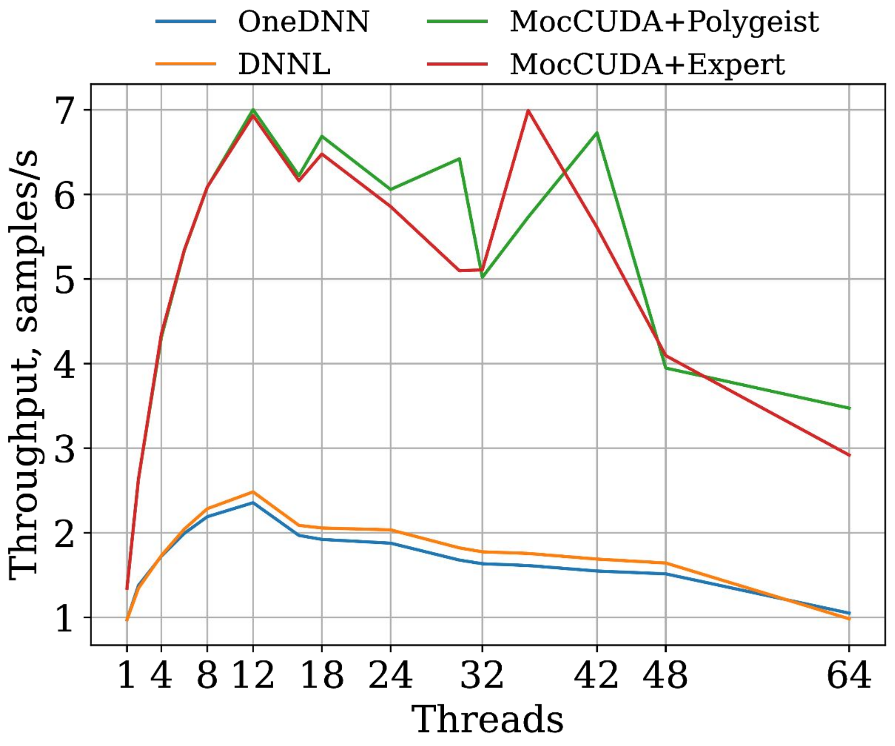
Better scaling!



Evaluation 2: Pytorch on fugaku

2.7x performance improvement over Pytorch CPU backend





Conclusion

Automatic transpilation with performance improvement!

- Compiling CUDA to a parallel representation in MLIR
- Parallel/GPU optimizations
  - Unnecessary synchronization removal
  - Code motion across parallel regions and synchronization
  - Unnecessary memory copy removal

Implemented in Polygeist:

LLVM incubator project, open sourced on Github, see https://polygeist.mit.edu

Longer talk on Polygeist at:

- MLIR Summit - **Tomorrow**
- SC22 - **14 November**