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.
Polygeist\textsuperscript{[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

\textsuperscript{[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

```cpp
__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);
}
```

```assembly
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
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.

```c
__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

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

```c
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**