Polygeist GPU

By directly emitting MLIR from C/C++, Polygeist preserves parallelism, control-flow, multi-dimensional tensors, and more. Polygeist introduces a new barrier operation to preserve parallel semantics in a backend-agnostic form, enabling Polygeist to handle a variety of parallel input languages, parallel backends, and enable optimizations to apply to all.

![Figure 2](image)

**Figure 2.** Polygeist/MLIR representation of the shared-memory version of the CUDA launch(normalize) code from Fig. 1. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit with parallel for loops across the blocks and threads. Shared memory is placed within the block parallel for, allowing access from any thread in the same block, but not a different block.

Barrier Representation

Representing barriers as a form of memory enables Polygeist to perform GPU-specific optimizations like barrier elimination, shared memory forwarding/elimination, fusion, and more!

```mlir
// Kernel body is available within the calling function, // enabling optimizations across the GPU/CPU boundary. func @launch(//launch): memref<f32>, // host thread index, Nn: int { // Parallel across all blocks in a grid parallel.for(//parallel) for (Nn, Gx, Gy, Nz) | (0, 0, 0) to (grid.x, grid.y, grid.z) { // Shared memory stack allocation per block. %shared_val = memref.alloca<f32>[:]; // Parallel across all threads in a block. parallel.for(//parallel) for (Nn, Hx, Hy, Hz) | (0, 0, 0) to (blk.x, blk.y, blk.z) { // Control-flow is directly preserved. // Host thread is zero. %sync = func.call @sync(//sync, %n: int, %shared_val:.memref<f32>) : void; // Shared memory store. %shared_val = memref.store(%sync, %shared_val:memref<f32>); // Compute results. %res = ... // Store results. %shared_val = memref.store(%shared_val:memref<f32>); } } }
```

![Figure 4](image)

**Figure 4.** Parallel loop splitting around a barrier: the code above the barrier is placed in a separate parallel "for" loop from the code following the barrier. This transformation eliminates the barrier, while preserving the semantics. The min-cut algorithm stores %x in the first loop, and %y in the second loop.

Barrier Lowering

As some systems do not have a GPU-equivalent thread group synchronize, Polygeist efficiently enables execution of barrier-semantics on platforms without a construct through recursive splitting.

Evaluation

To test performance and portability, we used Polygeist to transpose several CUDA GPU benchmarks to efficiently run on the CPU and compare against handwritten CPU (OpenMP) code. On the Rodinia suite, Polygeist achieves a 58% geometric speedup over handwritten OpenMP code. On a PyTorch ResNet-50, Polygeist (with our compatibility MocCUDA layer) outperforms PyTorch’s native CPU backend by 2.7x.

![Figure 5](image)

**Figure 5.** Scaling behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads.