High-Performance GPU-to-CPU Transpilation and Optimization via Polygeist/MLIR

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

Existing Compilation Pipelines

Existing compilers *lose* information about programs when compiled. This hinders any subsequent optimizations which may require this information including **parallel optimizations**, **polyhedral optimizations**, and **domain-specific optimizations**.

```
__device__ float sum(float* data, int n) { ... }
__global__ void normalize(float* out, float* in, int n) {
  int tid = blockIdx.x + blockDim.x * threadIdx.x;
  // Optimization: Compute the sum once per block.
  // __shared__ int val;
  // if (threadIdx.x == 0) val = sum(in, n);
  // __syncthreads;
  float val = sum(in, n);
  if (tid < n) out[tid] = in[tid] / val;
}
void launch(int* d_out, int* d_in, int n) {
  normalize<<<(n+31)/32, 32>>>(d_out, d_in, n);
}
```

Figure 1. A sample CUDA program `normalize`, which normalizes a vector and the CPU function `launch` launching the kernel. Each GPU threads calls `sum`, resulting in $O(N^2)$ work. Using shared memory (commented) reduces the work to $O(N^2/B)$ at extra resource cost. Computing `sum` before the kernel reduces work to $O(N)$.

Polygeist GPU

By directly emiting 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.

```
// Kernel body is available within the calling function,
// enabling optimizations across the GPU/CPU boundary.
func @launch(%d_out : memref<?xf32>, %d_in : memref<?xf32>, %n : i64) {
  // Parallel for across all blocks in a grid.
  parallel.for (%gx, %gy, %gz) = (0, 0, 0) to (grid.x, grid.y, grid.z) {
    // Shared memory = stack allocation in a block.
    %shared_val = memref.alloca : memref<f32>
    // Parallel for across all threads in a block.
    parallel.for (%tx, %ty, %tz) = (0, 0, 0) to (blk.x, blk.y, blk.z) {
      // Control-flow is directly preserved.
      if %tx == 0 {
        %sum = func.call @sum(%d_in, %n)
        memref.store %sum, %shared_val[] : memref<f32>
      }
      // Syncronization via explicit operation.
      polygeist.barrier(%tx, %ty, %tz)
      %tid = %gx + grid.x * %tx
      if %tid < %n {
        %res = ...
        store %res, %d_out[%tid] : memref<?xf32>
      }
    }
  }
}
```

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!

```
__global__ void bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];
  if ( tx == 0 ) node[ty] = input[index_in] ;
  // Unnecessary Barrier #1
  __syncthreads();
  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];
  __syncthreads();

  // Unnecessary Load #1
  weights[ty][tx] = weights[ty][tx] * node[ty];
  __syncthreads();

  for ( int i = 1 ; i <= log2(HEIGHT) ; i++){
    if( ty % pow(2, i) == 0 )
      weights[ty][tx] += weights[ty+pow(2, i-1)][tx];
    __syncthreads();
  }

  hidden[index] = weights[ty][tx];
  // Unnecessary Barrier #2
  __syncthreads();

  if ( tx == 0 ) out[by * hid + ty] = weights[tx][ty];
}
```

Figure 3. A CUDA kernel from the Rodinia that contains unnecessary synchronization.

Polygeist can also perform transformations that entirely eliminate barriers to enable backends without barrier support (like CPUs) to efficiently execute parallel programs from other models.

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.

```
parallel %i = 0 to 10 {
  %x = load data[%i]
  %y = load data[2 * %i]
  %a = fmul %x, %x
  %b = fmul %y, %y
  %c = fsub %x, y
  barrier
  call @use(%a, %b, %c)
  ...
}
```

```
%x_cache = memref<10xf32>
%y_cache = memref<10xf32>
parallel %i = 0 to 10 {
  %x = load data[%i]
  %y = load data[2 * %i]
  store %x, %x_cache[%i]
  store %y, %y_cache[%i]
}
parallel %i = 0 to 10 {
  %x = load %x_cache[%i]
  %y = load %y_cache[%i]
  %a = fmul %x, %y
  %b = fsub %y, %z
  call @use(%a, %b)
  ...
}
```

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` and `%y`, which are then used to recompute `%a`, `%b`, and `%c` in the second loop.

Evaluation

To test **performance and portability**, we use Polygeist to transpile several CUDA GPU benchmarks to efficiently run on the CPU and compare against hand-written CPU (OpenMP) code. On the Rodinia suite, Polygeist achieves a **58% geomean speedup** over handwritten OpenMP code. On a PyTorch Resnet-50, Polygeist (with our compatability MocCUDA layer) **outperforms PyTorch's native CPU backend by** $2.7\times$.

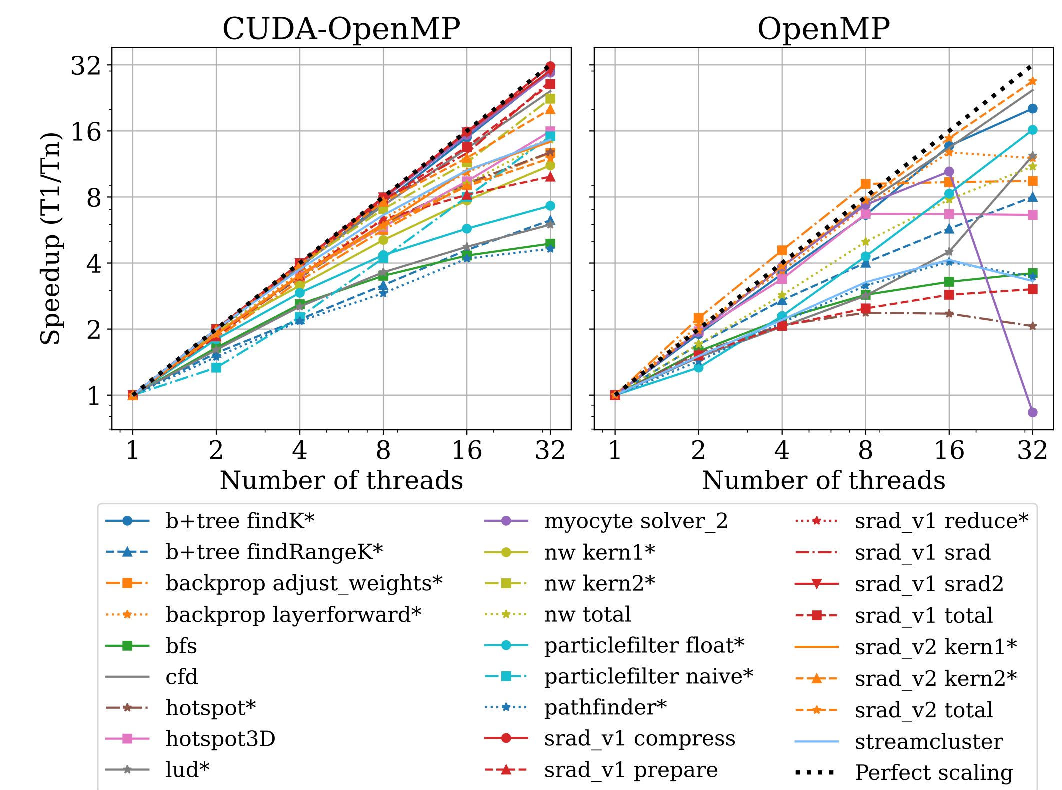

Figure 5. Scaling behavior behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads.

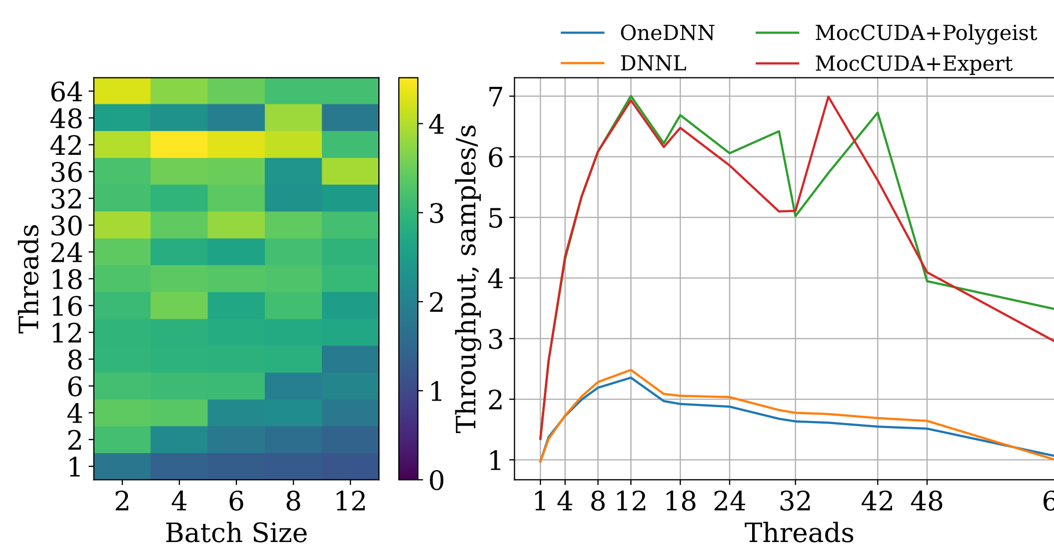

Figure 6. ResNet50 training on Fugaku node. Left: heatmap of relative throughput increase of "MocCUDA+Polygeist" over *tuned* oneDNN, higher is better. Right: throughput across batch sizes.

References

[1] W. S. Moses *et al.*, arXiv preprint arXiv:2207.00257 (2022).

[2] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, Polygeist: Raising C to polyhedral MLIR, in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, 2021, Association for Computing Machinery.