Parallel Optimizations and Transformations of GPU Kernels Using a High-Level representation in MLIR/Polygeist

Ivan R. Ivanov (Tokyo Tech) William S. Moses (MIT) Jens Domke (RIKEN) Toshio Endo (Tokyo Tech)

Existing Compilation Pipelines

Existing GPU compilers separate host-side and device-side code in separate modules

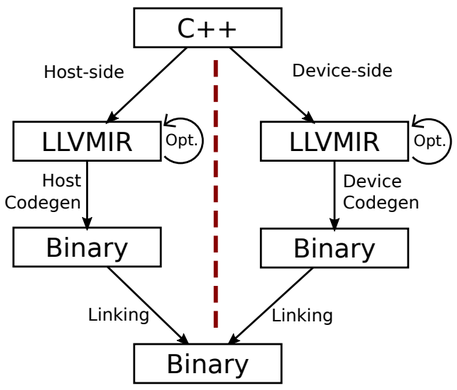

Figure 1. GPU code compilation by clang. Note how host-side and device-side IR never live in a single module and they get optimised and lowered to machine code separately.

Polygeist Compilation Pipeline

Polygeist compiles C++ code to a single-module MLIR representation that can be transformed and optimized.

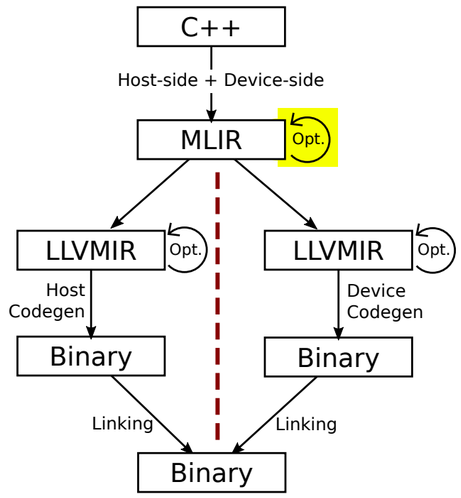

Figure 2. GPU code compilation by Polygeist. Note the ability of Polygeist to do optimizations on the MLIR level where both the host and device-side code exist in a single module.

Device-side Regions

To represent host and device side code within the same module and in a way suitable for optimizations, we introduce the **polygeist.gpu_wrapper** operation which contains code to be executed on the GPU inline with where in host-side code it should be executed.

```
func @f(%out, %in  : memref<?xf32>, %n : i64, %t : i64) {
  // Code contained in this region is to be executed on the device
  polygeist.gpu_wrapper {
    parallel.for (%bx, %by, %bz) = (0, 0, 0)
                 to (grid.x, grid.y, grid.z) {
      parallel.for (%tx, %ty, %tz) = (0, 0, 0)
                   to (blk.x, blk.y, blk.z) {
        %tid = %bx + blk.x * %tx
        %size = %n - 1 - %t
        if %tid < %size {
          %index = %tid + %t + 1
          %a = load %in[%index] : f32
          %b = load %in[ ... ] : f32
          %res = %a / %b
          store %res, %out[%index] : memref<?xf32>
        }
      }
    }
  }
}
```

Figure 3. Using the `gpu_wrapper` operation to specify portions of the code to be executed on the device. This is a simplified MLIR example from the `gaussian` cuda benchmark in Rodinia.

Alternative Code Paths

In some cases, we are able to vary the block sizes of GPU kernels. However, choosing an efficient block size at the MLIR level is difficult due to its multi-level nature and the gradual lowering.

We introduce a new MLIR operation called **polygeist.alternatives** which allows us to generate alternative code paths that achieve the same result - in this case, launching kernels with different block sizes. This allows us to lower each code path to LLVM and then generate device-side binaries before estimating the *cost* of each alternative. This allows us to look at various parameters of the kernels such as theoretical occupancy, register utilisation, register spilling, shared and constant memory usage.

Using this information, we build a cost model that estimates the performance of each kernel and chooses the best option.

```
func @launch(...) {
  polygeist.alternatives {
    polygeist.gpu_wrapper {
      // with block_size = 32
      parallel.for { ... }
    },
    polygeist.gpu_wrapper {
      // with block_size = 64
      parallel.for { ... }
    },
    ...
  }
}
```

(a) Generating alternative code paths with different block sizes.

```
func @launch(...) {
  polygeist.alternatives {
    gpu.launch_func @kernel_0 \
              block_size = 32,
    gpu.launch_func @kernel_1 \
              block_size = 64,
    ...
  }
}
```

(b) After lowering and generating device-side binaries.

kernel	Block Size	Registers used	Stack frame size	Occupancy
@kernel_0	32	1024	0	50%
@kernel_1	64	2048	0	100%
...	...	...	...	...

(c) Collect statistics for the compiled kernels.

```
func @launch(...) {
  gpu.launch_func @kernel_0 \
            block_size = 32,
}
```

(d) The kernel with the best estimated performance is chosen.

Since compile time cost modelling does not always achieve good results, in future work we are planning to explore runtime profiling of kernels and dynamically using the best performing option.

Host-Device Code Motion

We are able to move computation from device side code to the host and do computation only once instead of for each GPU thread. The example given is a simplified MLIR snippet of a real benchmark from Rodinia.

```
func @f(%out, %in  : memref<?xf32>, %n : i64, %t : i64) {
  %h = %t + 1
  %size = %n - %h
  polygeist.gpu_wrapper {
    parallel.for (%bx, %by, %bz) = (0, 0, 0)
                 to (grid.x, grid.y, grid.z) {
      parallel.for (%tx, %ty, %tz) = (0, 0, 0)
                   to (blk.x, blk.y, blk.z) {
        %tid = %bx + blk.x * %tx
        // Hoisted out:
        // %size = %n - 1 - %t
        if %tid < %size {
          // Index computation partially hoisted out:
          // %index = %tid + %t + 1
          %index = %tid + %h
          %a = load %in[%index] : f32
          %b = load %in[ ... ] : f32
          %res = %a / %b
          store %res, %out[%index] : memref<?xf32>
        }
      }
    }
  }
}
```

Figure 5. Hoisting device-region loop invariant code to the host. The original code before the optimization is shown in Figure 5.

Evaluation

We used Polygeist to compile a supported subset of CUDA benchmarks in the Rodinia suite. Evaluation was performed on a dual-socket Intel(R) Xeon(R) Gold 6252 CPU running at 2.10GHz CPUs with 24 cores, 256GB RAM, and an NVIDIA A100 GPU. We currently achieve up to 98% of clang's performance.

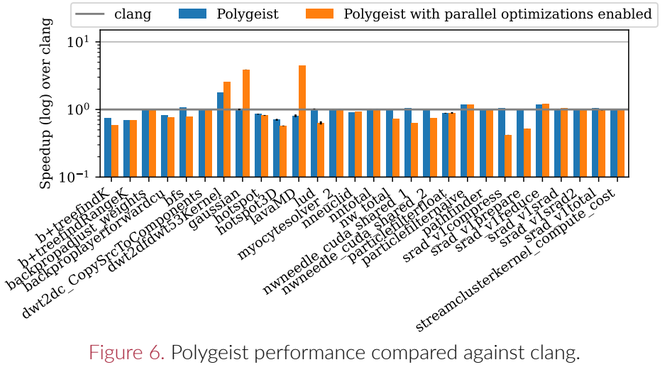

Figure 6. Polygeist performance compared against clang.

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.