High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs

William S. Moses
wmoses@mit.edu
MIT CSAIL
United States

Ivan R. Ivanov
ivanov@m.titech.ac.jp
Tokyo Tech
Japan

Jens Domke
jens.domke@riken.jp
RIKEN
Japan

Toshio Endo
endo@is.titech.ac.jp
Tokyo Tech
Japan

Johannes Doerfert
jdoerfert@llnl.gov
LLNL
United States

Oleksandr Zinenko
zinenko@google.com
Google
France

Abstract

While parallelism remains the main source of performance, architectural implementations and programming models change with each new hardware generation, often leading to costly application re-engineering. Most tools for performance portability require manual and costly application porting to yet another programming model.

We propose an alternative approach that automatically translates programs written in one programming model (CUDA), into another (CPU threads) based on Polygeist/MLIR. Our approach includes a representation of parallel constructs that allows conventional compiler transformations to apply transparently and without modification a nd enables parallelism-specific optimizations. We evaluate our framework by transpiling and optimizing the CUDA Rodinia benchmark suite for a multi-core CPU and achieve a 58% geomean speedup over handwritten OpenMP code. Further, we show how CUDA kernels from PyTorch can efficiently run and scale on the CPU-only Supercomputer Fugaku without user intervention. Our PyTorch compatibility layer making use of transpiled CUDA PyTorch kernels outperforms the PyTorch CPU native backend by 2.7×.

CCS Concepts: • **Software and its engineering → Compilers**; • **Theory of computation → Parallel computing models**.

Keywords: Polygeist, MLIR, CUDA, Barrier Synchronization

ACM Reference Format:
William S. Moses, Ivan R. Ivanov, Jens Domke, Toshio Endo, Johannes Doerfert, and Oleksandr Zinenko. 2023. High-Performance GPU-to-CPU Transpilation and Optimization via High-Level Parallel Constructs. In *The 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming (PPoPP '23), February 25-March 1, 2023, Montreal, QC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3572848.3577475

This work is licensed under a Creative Commons Attribution International 4.0 License.

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0015-6/23/02.
https://doi.org/10.1145/3572848.3577475

1 Introduction

Despite x86 CPUs and NVidia GPUs remaining primary platforms for computation, customized and emerging architectures play an important role in the computing landscape. A custom version of an ARM CPU, A64FX, is even used in one of the top supercomputers Fugaku [49] where its high-bandwidth memory is expected to compete with that of GPUs. However, these architectures are often overlooked by efficiency-oriented frameworks and libraries. For example, PyTorch [44] targeting Intel's oneDNN [28] backend expectedly underperforms on ARM due to architecture differences and even Fujitsu's customized oneDNN [20] does not yield competitive performance on some kernels. Such situations call for performance portability.

Many non-library approaches for performance portability have been proposed and include language extensions (e.g., OpenCL [14], OpenACC [26]), parallel programming frameworks (e.g., Kokkos [3]), domain-specific languages (e.g., SPIRAL [17], Halide [47] or Tensor Comprehensions [64]). All of these approaches still require legacy applications to ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model.

We explore an alternative approach based on a fully automated compiler that takes code in one programming model (CUDA) and produces a binary targeting another one (CPU threads). While GPU-to-CPU translation has been explored in the past [9, 23, 58], it was rarely able to produce efficient code. In fact, optimizations for CPUs and even generic compiler transforms, such as common sub-expression elimination or loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [39]. As representations of parallelism within a mainstream compiler have only recently begun to

be explored [10, 12, 32, 50, 55], existing transformations are limited and tend to apply to simple CPU codes only.

We propose a compiler model for most common GPU constructs: multi-level parallelism, level-wide synchronization, and level-local memory. In contrast to source and AST-level approaches, which operate before the optimization pipeline, and existing compiler approaches, which model synchronization as a black-box optimization barrier, we model synchronization from memory semantics. This allows synchronization-based code to interoperate with existing optimizations and enables novel parallel-specific optimizations.

Our model is implemented using MLIR [34] and LLVM [33] and leverages MLIR's nested-module approach for GPU [21]. We extended the Polygeist [40] C/C++ frontend to support CUDA and to produce MLIR which preserves high-level parallel structure. Our prototype compiler is capable of compiling PyTorch CUDA kernels, as well as other compute-intensive benchmarks, to any CPU architecture supported by LLVM. In addition to transformations accounting for the differences in the execution model, we also exploit parallelism on the CPU via OpenMP. Finally, our MocCUDA PyTorch integration allows us to compile and execute CUDA kernels in absence of a GPU while substituting unsupported calls.

We evaluate our compiler on Rodinia CUDA benchmarks [5] and PyTorch CUDA kernels. When targeting a commodity CPU, our OpenMP-accelerated CUDA code yields comparable performance with the reference OpenMP implementations from the Rodinia suite, as well as improved scalability. When using our framework to run PyTorch on the CPU-only Fugaku Supercomputer, we achieve roughly twice the images processed per second of a Resnet-50 [25] training run compared to existing PyTorch CPU backends.

Overall, our paper makes the following contributions:

- A common high-level and platform-agnostic representation of SIMT-style parallelism backed by a semantic definition of barrier synchronization that ensures correctness through memory semantics, and thus transparent application of existing optimizations.
- Novel parallel-specific optimizations which can exploit our high-level parallel semantics to optimize programs.
- An extension to the Polygeist C/C++ MLIR frontend capable of directly mapping GPU and CPU parallel constructs into our high-level parallelism primitives.
- An end-to-end transpilation[1] of CUDA to CPU for a subset of the Rodinia [5] benchmark suite and the internal CUDA kernels in PyTorch [44] necessary to run Resnet-50 on the CPU-only Fugaku supercomputer.

[1]We use the term *transpilation* to refer to taking a program in one programming model and emitting code for another, similar to source-to-source CUDA-to-C transpilers though now on IR. This procedure also *cross-compiles* the code. which refers to emitting non-native instructions.

2 Background

Mainstream compilers like Clang and GCC lack a unified high-level representation of parallelism. Compiling parallel constructs in frameworks like CUDA, OpenMP, or SYCL, forces the body of a parallel region to exist within a separate (closure) function which is invoked by a parallel runtime. Concepts such as thread index or synchronization are then represented separately, often through opaque intrinsic calls. As the compiler historically lacked information about parallelism and effects of the involved runtimes, any parallel construct also inadvertently acted as a barrier to optimization. While there have been attempts [10, 12, 32, 39, 50, 55, 61] in recent years to improve representations for CPU parallel constructs, accelerator programming comes with additional challenges. The unique programming model and complex memory hierarchy have left high-level representations of GPU parallelism within mainstream compilers under-explored.

```
__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 $O(N)$.

2.1 GPU Compilation

Consider the CUDA program in Fig. 1, which normalizes a vector. When compiled using Clang, the GPU program is a separate compilation unit. This prevents any optimization between the GPU kernel and the CPU calling code. In the case of Fig. 1, the total work of the program in a traditional compiler is $O(N^2)$, due to the $O(N)$ call to `sum` being performed for each thread. However, if the call to `sum` is performed only once prior to the kernel call, e.g., by performing loop-invariant code motion (LICM), the work would reduce to $O(N)$. A less effective variant of this optimization could reduce the work to $O(\frac{N^2}{B})$ through the use of shared memory. MLIR provides a nested-module representation for GPU programs that supports host/device code motion [21], but parallel code motion has not been implemented. In GPU to CPU code motion, LICM out of a parallel loop is always legal as any former device memory is also available on the host.

```
// Kernel launch is available within the calling
// function, enabling optimizations across the
// GPU/CPU boundary.
func @launch(%h_out : memref<?xf32>,
             %h_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 equivalent of `launch`/`normalize` code from Fig. 1. The kernel call is available directly in the host code which calls it. The parallelism is 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.

2.2 MLIR Infrastructure

MLIR is a recent compiler infrastructure designed for reuse and extensibility [34]. Rather than providing a predefined set of instructions and types, MLIR operates on collections of *dialects* containing interoperable user-defined operations, attributes and types. Operations are a generalization of IR instructions that can be arbitrarily complex, in particular, contain regions with more IR thus creating a nested representation. Operations define and use values that obey single static assignment (SSA) [7]. For example, MLIR dialects may model entire instruction sets such as NVVM (virtual IR for NVidia GPUs), other IRs such as LLVM IR [33], control flow such as loops, parallel programming models such as OpenMP and OpenACC, machine learning graphs, etc.

MLIR supports GPU thanks to the eponymous dialect, which defines the high-level SIMT programming model, host/device communication, and a set of platform-specific dialects: NVVM (CUDA), ROCDL (ROCm) and SPIR-V. MLIR's approach to GPU programming benefits from a *unified* code representation. Since an MLIR module may contain other modules, the "host" translation unit may embed the "device" translation unit as IR rather than file reference or binary blob. This approach provides host/device optimization opportunities unavailable to other compilers, in particular to move code between host and device [21].

```
__global__ f() {
  codeA();
  barrier();
  codeB();
}
```

```
__global__ f() {      // 0<=t.x< blockDim.x
  A[threadIdx.x] = ...; //  W A[i]: i==t.x
  barrier();            // RW A[i]: i!=t.x
  ... = A[threadIdx.x]; // R  A[i]: i==t.x
}
```

Figure 3. Left: A program containing a barrier between two arbitrary instructions. **Right:** Barrier semantics can be refined memory addresses accessed by operations above/below it in all threads *except* the current one.

2.3 Polygeist

Polygeist is a C/ C++ frontend for MLIR based on Clang [40]. It is capable of translating a broad range of C++ programs into a mix of MLIR dialects that preserve elements of the high-level structure of the program. Specifically, Polygeist preserves structured control flow (loops and conditionals) as MLIR SCF dialect operations and simplifies analyses by preserving multi-dimensional array constructs whenever possible by relying on the MLIR's multi-dimensional memory reference (memref) type. Finally, Poylgeist is able to identify parts of the program suitable for polyhedral optimization [16] and represent them using the Affine dialect.

3 Approach

We extended the Polygeist compiler [40] to directly emit parallel MLIR from CUDA. This leverages the unified CPU/GPU representation to allow the optimizer to understand host/device execution, and to enable optimization across kernel boundary. The use of existing MLIR's first-class parallel constructs (`scf.parallel`, `affine.parallel`) enables us to target existing CPU and GPU backends. Finally, MLIR's extensible operation set allows us to define custom instructions, with relevant properties and custom optimizations.

We define the representation of a GPU kernel launch as follows (illustrated in Fig. 2):

- A 3D parallel for-loop over all blocks in the grid.
- A stack allocation for any shared memory, scoped to be unique per block.
- A 3D parallel for-loop over all threads in a block.
- A custom Polygeist barrier operation that provides equivalent semantics to a CUDA synchronization.

This procedure enables us to represent any GPU program in a form that preserves the desired semantics. It is fully understood by the compiler and is thus amenable to compiler optimization. Moreover, by representing GPU programs with general parallelism, allocation, and synchronization constructs, we are not only able to optimize the original program, but also retarget it for a different architecture.

3.1 Barrier Semantics

A CUDA `__syncthreads` function guarantees that all threads in a block have finished executing all instructions prior to

the function call, before any threads executes any instruction after the call. Traditionally, compilers represent such functions as opaque optimization barriers that could touch all memory, and forbid any transformation involving them.

In our system, we chose to represent thread-level synchronization through a new `polygeist.barrier` operation. Unlike other approaches, `polygeist.barrier` (hence referred to as simply `barrier`) aims to only prevent transformations that would change externally visible behavior. Rather than disallowing any code motion across a `barrier`, we can successfully achieve the desired semantics by defining `barrier` to have specific memory properties, represented as a collection of memory locations (including unknown), and memory effect type (read, write, allocate, free), as is standard within MLIR. Consider the simple program in Fig. 3(left). The impact of the synchronization can only be observed if `codeA` and `codeB` access the same memory. Moreover, if both only read the same memory location, the synchronization is also unnecessary. We can enumerate the remaining cases: (1) `codeA` writes, `codeB` loads; (2) `codeA` loads, `codeB` writes; (3) `codeA` writes, `codeB` writes.

The barrier having the write behavior of `codeA` would ensure correctness of (1): the load in `codeB` could not be hoisted above the barrier, as it would appear to read a different value. Symmetrically, the barrier having the write behavior of `codeB` ensures the correctness of (2). Thus, the union of the writing behaviors of `codeA` and `codeB` is sufficient to prevent illegal movement of loads across the barrier.

However, this does not prevent writes from being moved. For example, `codeB` could be duplicated above the barrier in (3), and it would appear to have the same final memory state since the extraneous write before the barrier would never be read. Thus, we also define the barrier to have the reading behavior of `codeA` and `codeB`.

This model can be extended to include memory effects of all operations in the parallel loop which may have been executed before, or after, a given barrier. On a control flow graph with explicit branches, this requires exploring the operations within predecessors or successors, respectively. However, operating on MLIR's structured control flow level, with explicit operations for loops and conditionals, simplifies the analysis. Furthermore, if more than one barrier is present in the same block, it is unnecessary to look past it.

Given a sufficiently expressive side effect model, the memory semantics of the barrier can be further expanded. While barriers enforce ordering reads/writes to the same location from *different* threads, the natural execution order is sufficient within one thread. Therefore, barriers need not capture the memory effects of operations where the address is an *injective function* of the thread identifier. We implement the refinement for *affine* forms of access expressions leveraging the polyhedral framework in MLIR/Polygiest. For each memory access, we define an integer relation between a set of possible thread id values and the set of accessed array subscripts,

```
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.

$\mathcal{R} : T \to A$. We then compose direct and inverse relations for relevant operations to obtain a relation between thread indices accessing the same subscript, $\mathcal{D} = \mathcal{R}^{-1} \circ \mathcal{R} : T \to T'$. Finally, we subtract the identity relation $\mathcal{D} \setminus \mathcal{I} : T \to T'$. If non-empty, $\mathcal{D} \neq \emptyset$, different threads may access the same address and the barrier is required. Given a non-affine access or non-static control flow, we conservatively assume an access of the entire array dimension. In practice, this is rarely necessary on GPU code, whose loops typically have parametric/static bounds. Aliasing guarantees must be checked when more than one base address is involved.

Consider the code in Fig. 3(right). Since the sets of accessed addresses do not overlap, $\mathcal{A}_o \cap \mathcal{A}_b = \emptyset$, code motion across the barrier is allowed. In contrast, if the load or store to `A` were offset by 1, the barrier would be necessary as the data loaded after the barrier would be stored by a different thread.

3.2 Barrier Lowering

To enable GPU programs to run on a CPU, we must efficiently emulate the synchronization behavior of GPU programs. Whereas the memory semantics in Section 3.1 enable us to preserve the correctness of barriers during optimization, this section discusses how to implement the barrier on a CPU.

CPU architectures have no notion of thread blocks, nor the barrier instruction which waits on this conceptual grouping of threads. Instead, we use regular CPU threads and work sharing to distribute the thread-block loop iterations across them. Conceptually, this differs from the GPU execution model in which threads execute one iteration each. Work sharing requires each thread to execute multiple iterations sequentially, making it impossible to synchronize in the middle of iterations, but only at the end of the loop.

To address this, we developed a new barrier elimination technique for our MLIR representation. Our approach is

```
parallel for %id=0 to N {
  for %j = 5 to 0 {
    if (%id < 2^%j)
      A[%id] += \
          A[%id + 2^%j]
    barrier
  }
}
```

```
for %j = 5 to 0 {
  parallel for %id=0 to N {
    if (%id < 2^%j)
      A[%id]+=A[%id + 2^%j]
    barrier
  }
}
```

Figure 5. ***Left:*** A shared memory addition, which consists of a kernel call which contains for loop with a barrier inside. ***Right:*** Same but with the barrier directly in the parallel loop after a parallel/serial loop interchange.

```
parallel for %i=0 to N {
  do {
    run(%i)
    barrier
  } while(condition())
}
```

```
%helper = alloca memref<i1>
scf.do {
  parallel for %i=0 to N {
    run(%i)
    barrier
    %c = condition()
    if %i == 0 {
      store %c, %helper[]
    }
  }
  %c = load %helper[]
} while(%c)
```

Figure 6. Parallel interchange around a `while` loop. As the `condition()` function call must be executed on each thread to preserve correctness, a helper variable is used which holds the value of the call on the first thread.

an extension of loop fission (see Section 7) combining two transformations: *parallel loop splitting* and *interchange*.

3.2.1 Parallel Loop Splitting. Suppose a barrier has the kernel function (or, in our representation, parallel `for` loop) as its direct parent. It can be eliminated by splitting the loop around the barrier into two parallel `for` loops that run the code before and after the barrier, respectively. If the code before the barrier created SSA values that were used after it, these must be either stored or recomputed in the second parallel loop. We use the technique similar to one in [41] to determine the minimum amount of data that needs to be stored. Specifically, we create a graph of all SSA values. We then mark each value definition that cannot be recomputed (e.g. loads from overwritten memory) before the `barrier` as source, and values used after the `barrier` as sinks. We derive the minimum amount of data needing to be stored by performing a minimum branch cut on this graph.

3.2.2 Parallel Loop Interchange. Not all barrier operations have a parallel `for` as their immediate parent, some may be nested in other control flow operations. We created a model that specifies what instructions may run in parallel. With the sole exception of `barrier`, our representation does not require any specific ordering or concurrency to the program. Therefore it is legal (though potentially a reduction in parallelism) to add additional barriers. We can use this property to implement barrier lowering for control flow.

```
__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 7. An example CUDA kernel from the Rodinia backprop test that contains unnecessary synchronization and unnecessary use of shared memory.

Consider a control-flow construct `C` containing a barrier and nested in a parallel `for`. Adding barriers immediately around `C` will result in parallel loop splitting directly above and below `C`. As a result, the operations above and below `C` will be separated into their own parallel `for` and `C` will be the sole operation in the middle loop. We can then apply one of the following techniques to interchange `C` with the parallel `for`, thus making the `barrier`'s parent a parallel `for`.

Consider the case of a serial `for` loop containing a barrier, Fig. 5. This pattern is common in GPU code, e.g., to implement a reduction across threads [24]. As `barrier` must wait for all threads, each thread must execute the same number of `barriers`. Therefore, the number of iterations of the inner loop is the same for all threads, allowing for loop interchange.

While an `if` statement can be considered a loop with zero or one iteration, directly interchanging it with the surrounding parallel `for` when necessary is more efficient.

Whereas `for` loops in MLIR have a fixed trip count, `while` loops support dynamic exit conditions, like in Fig. 6. Since correctness requires executing `condition()` in every thread, a direct interchange would not be legal. However, GPU synchronization semantics require the trip count to be the same in all threads. Therefore, one can still perform an interchange using a helper variable to store the result of the condition.

This illustrates one of the advantages of building off of MLIR/Polygeist. By preserving high level program structures, we can use more efficient patterns to remove barriers.

4 Parallel Optimization

The high-level representation of both parallelism and GPU programs provided by Polygeist/MLIR enables a variety of optimizations. These include general optimizations that would

apply to any parallel program as well as specific optimizations in the context of GPU to CPU conversion.

4.1 Barrier Elimination & Motion

As GPU-style barriers have to be specially transformed to support CPU architectures, eliminating or simplifying any barriers can have dramatic effects. Moreover, even when running GPU code on the GPU, barrier elimination is highly useful as any synchronization reduces parallelism. Much of the infrastructure for barrier elimination/simplification comes directly from its memory behavior defined in Section 3.1. Let $M_B^{\uparrow}$ ($M_B^{\downarrow}$) be the union of memory effects before (after) a barrier `B` until the edge of the parallel region. Let $M_B^{\bullet\dagger}$ be the subset of $M_B^{\bullet}$ with effects until the first barrier rather than region edge. Given a barrier `B`, if there are no memory effects to the same location across the barrier other than a read-after-read (RAR), i.e. $M_B^{\uparrow\dagger} \cap M_B^{\downarrow} = \emptyset$, `B` has its behavior subsumed by the previous barrier. Symmetrically $M_B^{\uparrow} \cap M_B^{\downarrow\dagger} = \emptyset$ means the barrier is subsumed by the following one. A specific case of a removable barrier is one that has no memory effects at all.

For example, consider the code in Fig. 7, which comes from the `backprop` Rodinia benchmark [5]. The first and last `__syncthreads` instructions are unnecessary. This can be proven from our memory-based barrier elimination algorithm above as follows. For the first barrier, $M^{\uparrow}$ (going all the way to the start) contains only a write to `node` and a read from `input`. $M^{\downarrow\dagger}$ (going to the second `__syncthreads`) contains a write to `weights` and a read from `hidden`. None of these conflict if, given the calling context, the pointers are known not to alias. Thus, it is safe to eliminate the barrier.

The same memory analysis can also be applied to perform barrier motion. One simply needs to place a fictitious barrier at the intended location and check if the previous memory analysis would deduce that the current barrier is unnecessary, thereby permitting barrier motion.

4.2 Memory-to-register promotion across barriers

One of the goals of defining `barrier`'s semantics from its memory behavior is to enable memory optimizations to operate correctly and effectively in code that contains barriers. As described in Section 3.1, barriers have the memory behavior of the code above and below them with the notable exception of an access from the current thread. This hole is important as it enables memory-to-register promotion (mem2reg) to operate on thread-local memory such as local variables. This optimization can replace slow memory reads with fast registers. For example, consider again the code in Fig. 7. Consider the load and store to `weights[ty][tx]` labeled "Unnecessary Store #1" and "Unnecessary Load #1", and the sync in between the two. The only value that can be loaded at that point is the same value which was stored earlier, a register containing the value loaded from `hidden`. As

```
omp.parallel {
  omp.wsloop %i= 1 to 10 {
    codeA(%i)
  }
}
omp.parallel {
  omp.wsloop %i= 1 to 10 {
    codeA(%i)
  }
}
```

```
omp.parallel {
  omp.wsloop %i=1 to 10 {
    codeA(%i)
  }
  omp.barrier
  omp.wsloop %i=1 to 10 {
    codeA(%i)
  }
}
```

Figure 8. Example of OpenMP parallel region fusion. Fuse two adjacent OpenMP parallel regions by inserting a barrier to allow the threads to be initialized once instead of twice.

that same location is overwritten before anyone else could read from `weights`, the first store also can be safely eliminated once the load is removed. During mem2reg, Polygeist can derive this forwarding property, since the hole in the memory properties described in Section 3.1 allows it to deduce that the barrier operation does not overwrite the store for the current thread. As a result, traditional load and store forwarding correctly operates on the barrier code.

4.3 Parallel loop-invariant code motion

The traditional loop-invariant code motion optimization aims to move an instruction `I` outside serial "for" loops, reducing the number of times `I` is executed. If `I` may access memory, or has other side effects, in addition to checking that the operands of `I` are themselves loop invariant, the compiler must check that no other code within the "for" loop conflicts with the memory access performed by `I`.

On present compilers, while it is possible to apply loop-invariant code motion to serial for loops within GPU kernels, it is not possible to apply loop-invariant code motion to hoist instructions outside of a kernel call. This is in part due to the fact that GPU kernels are kept in a separate module from the CPU code which calls them, as well as a lack of understanding of parallelism (see Fig. 1).

Counter-intuitively, with the right semantics we can apply loop-invariant code motion to parallel for loops even if we would not be able to apply it to an equivalent serial loop. We will rely on the fact that semantics of our program permits us to arbitrarily interleave iterations of a parallel "for" loop as long as we maintain the orderings required by barriers. As such, it is legal, though not necessarily fast, to run the program in lock-step. In other words, if a parallel for loop had 10 instructions, each thread can execute instruction 1 before any thread executed instruction 2, and so on. As a consequence, it is now legal to hoist an instruction so long as its operands are invariant and no *prior* instruction in the parallel for loop conflicts with `I`.

4.4 Block Parallelism Optimizations

OpenMP is our primary target for parallel execution on the CPU. It implements parallel "for" loops as two constructs.

```
for (i=0; i<N; i++) {
  #pragma omp parallel for
  for (j=0; j<10; j++) {
    body(i, j);
  }
}
```

```
#pragma omp parallel
for (i=0; i<N; i++) {
  #pragma omp for
  for (j=0; j<10; j++) {
    body(i, j);
  }
  #pragma omp barrier
}
```

Figure 9. Example of OpenMP parallel region hoisting. This can be seen as an extension of parallel region fusion across "regions" corresponding to each iteration of the outer loop.

First, the loop is outlined into a function which is called once per thread, representing OpenMP's "parallel" construct. Then, within the outlined function, the iteration space is distributed across threads, representing OpenMP's "worksharing loop" construct. OpenMP also has a "barrier" construct, but with semantics *different* than that of a GPU barrier.

When multiple parallel loops are executed in a row, e.g., following the barrier lowering from Section 3.2, the overhead of thread management can be reduced by fusing adjacent OpenMP "parallel" constructs [11] *without* fusing the worksharing loops (see Fig. 8), thus not undoing the barrier lowering. This can be extended to moving the OpenMP parallel region outside the surrounding "for" in Fig. 9, initializing threads once rather than N times. Applying these to control flow constructs enables all of the "for" loops generated by performing parallel loop fission on a block to have their OpenMP "parallel" (but not work sharing loops) fused.

As GPU programs tend to be written with high parallelism in mind, the parallelism provided by the different blocks may already saturate the number of available cores alone. If there is no use of shared memory, the block and thread parallelism can be collapsed into a single OpenMP parallel `for`, which will evenly divide the total iteration space in a single parallel region. However, if there is shared memory, our tool will generate nested parallel regions to represent the shared memory allocation. In this case, the additional overhead from the nested OpenMP parallel regions may outweigh the potential added parallelism. In addition, parallelizing the inner loops may lead to adverse memory effects such as false sharing, further penalizing performance [63, 65]. As such, we also support an optimization for serializing any nested OpenMP parallel regions. Performing such serialization may leverage memory locality to improve performance.

5 MocCUDA: Integration into PyTorch

One of our goals is to support execution of originally GPU codes on a CPU-only supercomputer such as Fugaku [49]. We focus on PyTorch [44] that has not been ported to the A64FX architecture and therefore uses naive fallback CPU kernels. Observing that CPUs with high-bandwidth memory are likely to benefit from GPU-style optimization, we implement MocCUDA, a mock GPU backend for PyTorch that redirects the calls to CUDA runtime and libraries to our

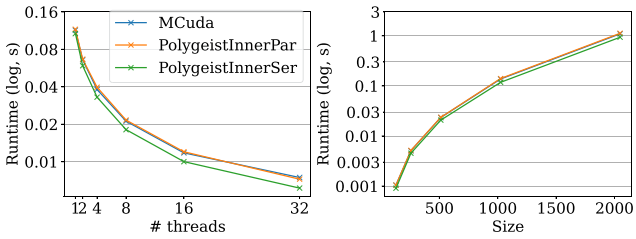

Figure 10. PolygeistInnerPar performs similarly to MCUDA; PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similaly to MCUDA, whereas PolygeistInnerPar keeps both the blocks and threads parallel. Left: Average runtime as a function of thread count (averaging over matrix sizes). Right: Average runtime as a function of matrix size (averaging over thread counts).

implementations or A64FX-specific math libraries [20]. We collect statistics of library calls and may optionally substitute them with CPU versions transpiled by Polygeist.

6 Evaluation

We demonstrate the advantages and applicability of our approach on two well-known GPU benchmark suites: a subset of the GPU Rodinia benchmark suite [5] and a PyTorch implementation of a Resnet-50 neural network. These benchmarks were chosen to 1) provide a rough performance comparison of our GPU to CPU compilation on a benchmark suite (Rodinia) that has hand-coded CPU versions and 2) demonstrate a successful end-to-end integration of our system into a useful and real application (PyTorch Resnet-50) on Supercomputer Fugaku, which does not have any GPUs. Additionally, we compare the performance of our approach to the existing MCUDA [58] tool on a CUDA matrix multiplication.

For Rodinia, we compare our translated CUDA to CPU code against OpenMP versions of the benchmarks, where they exist, as well as a run on a GPU. For the PyTorch Resnet-50, we compare against the "native" and oneDNN backends.

Polygeist[2] was compiled using LLVM 15 (git `00a1258`). For the PyTorch Resnet-50, we compile Pytorch v1.4.0 using NVidia's CUDA 11.6 SDK for Arm[3], LLVM 13, and Fujitsu's SSL2 v1.2.34 library. For the baseline PyTorch measuremets, we use Fujitsu's pre-installed PyTorch (v1.5.0).

We evaluate the Rodinia and matrix multiplication tests on an AWS c6i.metal instance (dual-socket Intel Xeon Platinum 8375C CPU at 2.9 GHz with 32 cores each and 256 GB RAM) running Ubuntu 20.04. Measurements were performed on the first socket, with hyperthreading and turbo boost disabled. Each number is the median of at least 5 repetitions.

[2]MocCUDA and Polygeist are available at https://gitlab.com/domke/MocCUDA and https://github.com/llvm/Polygeist.

[3]Even though we will run PyTorch on a GPU-less system, we must compile PyTorch on a CUDA-enabled system to ensure the correct code is emitted. We also prevented inlining of three Pytorch functions.

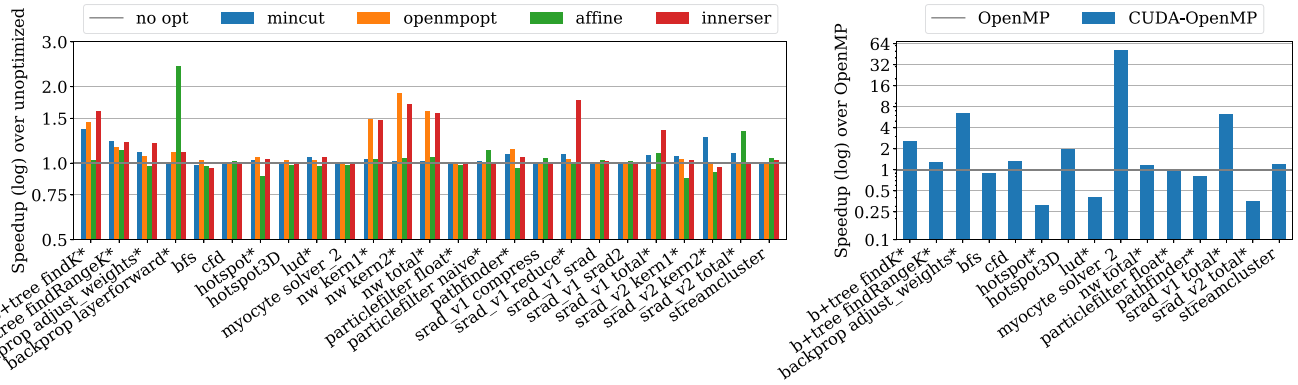

Figure 11. Left: Relative speedup (higher is better) applying parallel optimizations, proposed in Section 4, over our flow without optimization. Right: Speedup of transpiled CUDA-to-OpenMP compared against native OpenMP code (when available) running with 32 threads. Asterisks denote barriers within the benchmark.

6.1 Comparison to MCUDA

First, we compare with the previous work in MCUDA [58]. MCUDA is an AST-level tool which produces new CPU C/C++ as an output and uses loop fission to handle synchronization. As a source-to-source tool, MCUDA only handles a fraction of the input language, making it unable to run on Rodinia programs. Instead, we compare the runtimes of a matrix multiplication kernel across a range of threads (1–24) and matrix sizes (128×128 – 2048×2048) in Fig. 10. Polygeist with all optimization excluding serialization of the inner loop (PolygeistInnerPar) produces code within 1.3% of MCUDA on average. PolygeistInnerPar has a 1.5% slowdown on 1 thread, and 3.2% speedup on 32 threads. This behavior is caused by OpenMP overhead in handling nested parallel constructs. In fact, MCUDA only parallelizes the outermost loop. When Polygeist also serializes the inner loops (PolygeistInnerSer), it achieves an overall 14.9% speedup over MCUDA, with a 4.5% speedup on 1 thread and 21.7% speedup on 32 threads.

6.2 Use case 1: Rodinia Benchmarks

We benchmarked the 14 benchmarks that are currently supported by Polygeist, and had a nontrivial runtime.[4] We verified correctness by comparing the program outputs produced by compiling with `nvcc` and executed on a GPU, and compiled by our flow and executed on a CPU. We also employed the use of CPU-based parallel and undefined behavior analysis tools, which via our tool, allowed us to successfully diagnose and repair one race bug and several undefined memory bugs in the original CUDA code. We inserted timing measurements across kernels and/or computational portions of the code that include kernels, in some cases multiple per benchmark. Where possible, we time equivalent portions of the OpenMP versions of the same benchmarks.

We compare the Rodinia CUDA benchmarks compiled for the CPU with the Rodinia OpenMP verions of the benchmark in Fig. 11(right). While there is some variation from benchmark to benchmark, overall our approach is on par with the hand-coded versions of the benchmarks, and even nets a 58% geomean performance improvement, when the inner serialization optimization is enabled. Without inner serialization, we still see a geomean speedup of 34%. The speedup for `myocyte` is largely due to fewer instruction and data cache misses on the transcompiled code, which comes from optimizations which specialize the (parallel) to kernel call context, as well as the CUDA version employing fewer branches. The speedup for `backprop` is partially due to parallel optimizations (see Fig. 11(left)) and partially due to the CUDA code being implemented with a linear array, as required by CUDA, instead of the double-pointer used in the OpenMP code. The `srad_v1` benchmark benefits from a shared memory reduction in addition to parallel optimizations which eliminate most barriers and shared memory. In contrast, `hotspot` and `pathfinder` see a slowdown compared against native OpenMP code, due to duplicated computation in order to reduce synchronization and make better use of plentiful GPU parallelism. The slowdown for the transpiled CUDA version of `lud` is due to being written with a transposed loop ordering in contrast to the OpenMP code.

We test the scaling properties of our approach by comparing transpiled CUDA with native OpenMP kernels in Fig. 12. Transpiled CUDA codes generally scale much better than the native OpenMP versions. As most CUDA programs are written with thousands of threads in mind, this indicates that our framework was able to preserve that parallelism as the GPU-specific constructs were being rewritten for CPU-compatible equivalents. On 32 threads without inner serialization, transpiled CUDA codes had a geomean speedup of 16.1× across all tests. As OpenMP versions of benchmarks do not exist

[4]The `hybridsort`, `kmeans`, `leukocyte`, `mummergpu` `huffman` and `heartwall` use unsupported C++ or CUDA features within Polygeist (virtual functions and texture memory). The `lavaMD` and `dwt2d` benchmarks use *ill-formed* C++ with undefined behavior due to reading from uninitialized memory. The `nn` and `gaussian` tests ran in ≤ 0.005 seconds.

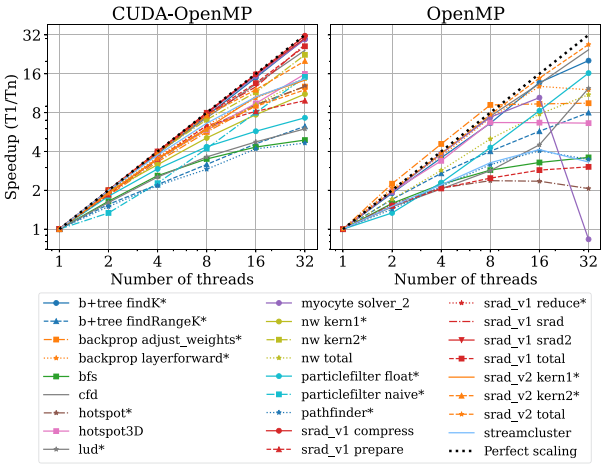

Figure 12. Scaling behavior behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads. Not all Rodinia CUDA kernels have OpenMP versions.

for all tests, if we consider only CUDA codes for which there exists an OpenMP version, we find a geomean speedup of 14.0×, whereas OpenMP has only a speedup of 7.1×. Serializing the inner loop slightly reduces scalability, but still results in improved scalability over OpenMP, finding a geomean speedup of 14.9× over all tests with inner serialization enabled, and a 12.5× speedup on codes with OpenMP versions. That is, most of the speedup is due to transpliation and barrier optimization as illustrated in Fig. 14(right). Inner lop serialization was observed to be beneficial in presence of multiple outer loops for which the OpenMP model triggers barrier synchronization repeatedly after the inner loop.

We perform an ablation analysis to show how individual optimizations impact performance. The "mincut" series in Fig. 11(left) shows performance with the optimization outlined in Section 3.2.1. This is only relevant for benchmarks containing barriers (marked by an asterisk in the Figure). When applicable, mincut provides a 5.8% geomean speedup. The "openmpopt" series in Fig. 11(left) demonstrates the impact of OpenMP region merging and similar optimizations and results in a 10.5% geomean speedup. The "affine" series in Fig. 11(left) shows the result of raising control flow to their affine variants and enabling simple serial and parallel loop optimizations (such as loop unrolling and re-indexing). While this produces a geomean speedup of 5.4% across the board, it results in a 2.4× speedup for the backprop layerforward test as it results in a loop containing synchronization being fully unrolled and reduced to `if` statements.

6.3 Use case 2: Pytorch/Resnet50 Test

To evaluate the PyTorch Resnet-50, we execute a full node-parallel training run on one TofuD unit of the Fugaku FX1000 supercomputer, comparing against the native PyTorch CPU backend and the optimized oneDNN backend, as available.

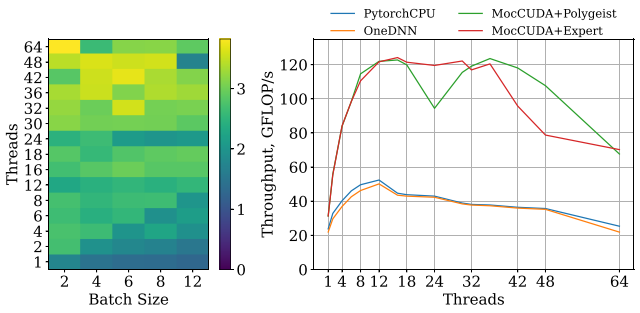

Figure 13. ResNet50 training on Fugaku node. Left: heatmap of relative throughput increase of "MocCUDA+Polygeist" over Fujitsu-*tuned* oneDNN, higher is better. Right: geomean throughput across batch sizes; "MocCUDA+Expert" uses an expert-written OpenMP kernel; "MocCUDA+Polygeist" uses the generated kernel, and PytorchCPU is Pytorch's native OpenMP backend.

We *replaced* the functions related to computing log-likelihood with Polygeist-transpiled functions as their CUDA kernels use barriers and their CPU versions contain naive implementations, and dispatched other calls to relevant libraries.

We ran multiple forward and back propagation passes of Resnet-50 on 224×224 ImageNet in a data-parallel fashion. We employ Horovod's synthetic benchmarking script [52]. We build Horovod v0.19.5 with CUDA, LLVM, and Fujitsu's MPI library to enable multi-node, distributed deep learning on top of Pytorch. We assign one MPI rank per A64FX core memory group (CMG), emulating up to 4 GPUs per node, and scale the test from one node (2 ranks) to 12 nodes (48 ranks) in one TofuD unit (smallest 2×3×2 torus) while keeping the number of OpenMP threads fixed at 12 to accommodate one thread per core. We use Pytorch v1.4.0 for our approach, while the other backends depend on Pytorch v1.5.0.

Performance was measured in GFLOP/s by using perf, and Benchmarker [13], which sets up the neural network and test data and executes the layer. We run with batch sizes 1–228 on 1–64 threads, averaging across epochs, and we compare the different backends for batch sizes 1–12 where all backends ran successfully.

Peak performance for MocCUDA was achieved at batch size 168 with 42 threads at 943 GFLOP/s, which amounts to 14% of the theoretical peak of the A64FX processor [19]. MocCUDA systematically outperforms Fujitsu's tuned oneDNN across batch sizes and thread counts, yielding up to 4.5× throughput increase (geomean 2.7×, min 1.2×) as shown in Fig. 13. MocCUDA with expert-written kernels is comparable to MocCUDA with Polygeist-generated kernels. Furthermore, the throughput of MocCUDA keeps increasing with the number of threads provided a sufficiently large batch size as shown in Fig. 14(left). For batch size 24, it plateaus at 24 threads while for batch size 168, it peaks at 42 threads.

The improvement can be explained by a combination of the PyTorch CPU design and performance characteristics

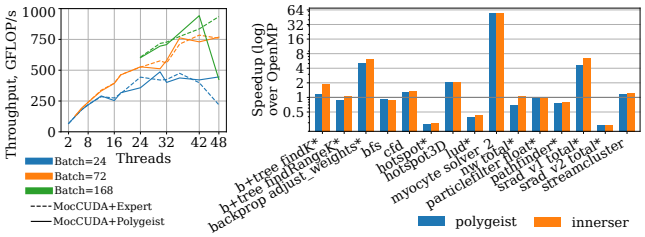

Figure 14. Left: ResNet throughput continues to scale for large batch sizes; large batches time out with few threads. Right: inner loop serialization contributes up to 30% speedup while most comes from barrier optimizations.

of oneDNN. As Intel's oneDNN [28] does not account for HBM available on A64FX, it uses cache-friendly direct convolutions instead of GEMM-based convolutions, less efficient in presence of HBM for Arm CPUs. While the custom fork of oneDNN tuned by Fujitsu [20], improves upon Intel oneDNN's performance (though by a geomean of 6%), it still leaves room for performance improvements.

This demonstrates that our approach is capable of automatically deriving efficient versions of deep learning kernels (and potentially other applications) from their CUDA versions, thus addressing the limitations of missing or inefficient kernels for CPUs with high-bandwidth memory without the need for reverse or re-engineering the application.

7 Related Work

7.1 GPU to CPU Synchronization

One of the first tools for emulating GPUs on a CPU was provided directly by NVidia for debugging purposes and emulated each thread on the GPU with a distinct CPU thread. While functional, the large gap in the number of available threads makes the emulation inefficient.

MCUDA [58] (2008) performs an AST transformation of C GPU code to generate new C CPU code that calls a thread-independent parallel `for` routine. MCUDA pioneered the use of "deep fission" to handle synchronization, which splits parallel loops and other constructs at synchronization points in order to eliminate them. This fission technique is also applied in other tools: Ocelot [9] (2010), a binary-translation tool that parses PTX assembly into LLVM and just-in-time compiles kernel functions; POCL [29] (2015), a Clang/LLVM compiler pass for OpenCL; COX [23] (2021), another LLVM transformation pass for translation of CUDA that uses fission, and handles warp-level primitives; and even this work. While the intuition behind the fission approach is similar to that used here, we apply fission inside of a high-level compiler, rather than either source or a low-level IR. As demonstrated in Section 3.1, performing fission on structured programs enables more efficient code transformations. While applying fission at a source-level misses the opportunity to run optimizations before fission (like barrier elimination) and applying fission at a low-level requires attempting to reconstruct the high-level structure, operating within MLIR allows us to both apply optimization and preserve high-level structure. Moreover, source-level tools tend to be quite fragile as they must re-implement parsing and semantics or the target language (e.g. C++), and as a result only operate on a limited subset of the input language, requiring re-engineering effort to replace unsupported constructs (like pointer arithmetic).

Another approach uses continuation-passing to handle synchronization by creating state machine of all synchronization points (e.g. "microthreading") [57] (2010). Karrenberg and Hack [30] (2012) propose a continuation-passing approach in LLVM that includes an algorithm for detecting and reducing divergence in the control-flow-graph. Follow-up work minimizes live values to reduce memory traffic [37].

VGPU [45] (2021) is similar to NVidia's virtual GPU, except using C++ thread and fence. Shared memory, implemented as a single global, is expanded by the number of blocks.

Prior work that operates at the low-level LLVM IR extends significant effort to reconstruct high-level constructs, such as loops and kernel configurations, required for either efficient fission or continuation passing. For example, POCL [29] runs canonicalizations and loop transformations to rewrite the control flow graph and attempt to recognize it as a specific form that can be handled. Prior work that operates at source/AST level (e.g. MCUDA), beyond still needing to recognize GPU-level concepts, cannot benefit from optimizations that simplify the code resulting in easier control flow.

In contrast, by operating on MLIR's mix-of-abstractions, we are able to simultaneously preserve source-level structure and perform program transformations such as loop unrolling or LICM that can, e.g., remove nested synchronization.

7.2 Parallel Portablity/IR, & OpenMP Optimizations

Several tools define new abstractions in the host language that are amenable to CPU or GPU execution. Examples include ISPC [46], RAJA [2], Kokkos [15], or MapCG [27] (limited to map-reduce code) in C++, Loo.py [31] in Python, and KernelAbstractions.jl [6] in Julia. These approaches provide performance portability for any new code written with them. However, existing code must be rewritten in said framework and may not compose with other frameworks/languages.

Several pieces of prior art discuss parallel intermediate representations, such as Tapir [50] for representing Cilk [18] in LLVM; OpenMPIR [56] for representing OpenMP in LLVM, PPIR [51] for pattern trees, and the MLIR OpenMP Dialect; as well as SDf3 [59] for visually representing concurrency as a control-flow graph. These works primarily focus on the *representation* for their particular style of parallelism (e.g. OpenMP tasks in OpenMPIR), which does not include GPU-style barriers, rather than on parallel *transformations* (such as barrier elimination) or optimizations, with the exception of consistency/race checks or automatic parallelization [38, 42].

The use of OpenMP parallel region expansion is known to be beneficial [11]. Clang/LLVM optionally supports the transformation in a weaker form [36].

7.3 Barriers

Several pieces of prior work explored the semantics of barrier or synchronization instructions, including in relation to GPUs. Work has been done to verify the correctness of barriers [1]. [54] experimentally evaluates the forward progress, fairness models of various GPU vendors. [53] implements a GPU barrier that applies across work-groups, as opposed to just within a work group. [60] add Java memory barriers to programs to ensure weak and sequential consistency semantics. They find that without synchronization and delay set analysis, introducing consistency semantics has an average 26.5× slowdown, whereas when using these analyses to insert fewer synchronizations can achieve a 10% and 26% slowdown for weak and sequential consistency, respectively.

Barrier elimination was implemented in the SUIF compiler for SPMD with shared memory [62] and for software-distributed memory [22]. This relies on a purpose-built communication analysis across the barrier whereas our method leverages the memory effects of the barrier itself. On the other hand, it supports synchronization minimization, such as replacing a barrier with nearest-neighbor communication, which our flow currently does not. Several pieces of work have proposed code generation techniques or code transformations aimed at minimizing the amount of synchronization within SPMD programs [8] or imperfect loops [43]. These approaches are applied to a sequential program, or one without synchronization at all, while our approach is applied parallel CUDA programs.

Synchronization minimization was explored within the polyhedral framework [35]. PolyAST supported analysis and transformation of programs with OpenMP directives [4]. While our flow may benefit from the polyhedral representation, it may operate without it and supports a significantly larger set of input programs. Razanajato et.al. leveraged the framework to generate different OpenMP parallelism constructs [48], which is complementary to our code generation.

8 Conclusion

By extending Polygeist/MLIR, we developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs. A key component of our framework is the development of a high-level barrier operation, key to representing GPU programs, whose semantics can be fully defined by its memory behavior. Unlike prior representations of parallel barriers, our semantics enable direct integration of barriers within optimization. As efficacy validation, we demonstrated GPU to CPU transpilation of a subset of the Rodinia benchmark suite on a commodity CPU and transpile Resnet-50 from the PyTorch CUDA source to run on A64FX CPU. The Rodinia benchmarks achieve a 58% geomean speedup of the transpiled GPU code over handwritten OpenMP versions. Similarly, we observe a $\approx 2\times$ speedup of transpiled kernels over the native PyTorch CPU backend.

Currently, the transpiled GPU code keeps the same schedule when run on the CPU, except for the innermost loop serialization that improves performance. A fruitful avenue of future work may perform advanced rescheduling the code to better take advantage of CPU-style memory hierarchies.

Acknowledgment

Thanks to Valentin Churavy of MIT and Albert Cohen of Google for discussions about transformations within MLIR. Thanks to Douglas Kogut, Jiahao Li, and Bojan Serafimov for discussions about parallel optimizations in compilers. Many thanks to Cosmin Oancea for helping with the final version.

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323, in part by Los Alamos National Laboratories grant 531711, and in part by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. Johannes Doerfert was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative, and in part by the Lawrence Livermore National Security, LLC ("LLNS") via MPO No. B642066, LLNL-CONF-843530. This work was supported in part by the Japan Society for the Promotion of Science KAKENHI Grant Number 19H04119 and by the Japanese New Energy and Industrial Technology Development Organization (NEDO).

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government, or Lawrence Livermore National Security, LLC neither of whom nor any of their employees make any endorsements, express or implied warranties or representations or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of the information contained herein. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

References

[1] Alexander Aiken and David Gay. 1998. Barrier Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 342–354.

https://doi.org/10.1145/268946.268974

[2] David Beckingsale, Richard Hornung, Tom Scogland, and Arturo Vargas. 2019. Performance Portable C++ Programming with RAJA. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming* (Washington, District of Columbia) *(PPoPP '19)*. Association for Computing Machinery, New York, NY, USA, 455–456. https://doi.org/10.1145/3293883.3302577

[3] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216. https://doi.org/10.1016/j.jpdc.2014.07.003 Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[4] Prasanth Chatarasi, Jun Shirako, and Vivek Sarkar. 2015. Polyhedral Optimizations of Explicitly Parallel Programs. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. 213–226. https://doi.org/10.1109/PACT.2015.44

[5] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[6] Valentin Churavy, Dilum Aluthge, Lucas C Wilcox, Simon Byrne, Maciej Waruszewski, Ali Ramadhan, Meredith, Simeon Schaub, James Schloss, Julian Samaroo, Jake Bolewski, Charles Kawczynski, Jeremy E Kozdon, Jinguo Liu, Oliver Schulz, Oscar, Páll Haraldsson, Takafumi Arakaki, and Tim Besard. 2022. JuliaGPU/KernelAbstractions.jl: v0.8.0. https://doi.org/10.5281/zenodo.6324344

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

[8] Alain Darte and Robert Schreiber. 2005. A Linear-Time Algorithm for Optimal Barrier Placement. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 26–35. https://doi.org/10.1145/1065944.1065949

[9] Gregory Diamos, Andrew Kerr, Sudhakar Yalamanchili, and Nathan Clark. 2010. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 353–364.

[10] Johannes Doerfert, Jose Manuel Monsalve Diaz, and Hal Finkel. 2019. The TRegion Interface and Compiler Optimizations for OpenMP Target Regions. In *OpenMP: Conquering the Full Hardware Spectrum - 15th International Workshop on OpenMP, IWOMP 2019, Auckland, New Zealand, September 11-13, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11718)*, Xing Fan, Bronis R. de Supinski, Oliver Sinnen, and Nasser Giacaman (Eds.). Springer, 153–167. https://doi.org/10.1007/978-3-030-28596-8_11

[11] Johannes Doerfert and Hal Finkel. 2018. Compiler Optimizations for OpenMP. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 113–127.

[12] Johannes Doerfert and Hal Finkel. 2018. Compiler Optimizations for Parallel Programs. In *Languages and Compilers for Parallel Computing - 31st International Workshop, LCPC 2018, Salt Lake City, UT, USA, October 9-11, 2018, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 11882)*, Mary W. Hall and Hari Sundar (Eds.). Springer, 112–119. https://doi.org/10.1007/978-3-030-34627-0_9

[13] Aleksandr Drozd. 2021. Benchmarker. Online GitHub repository: https://github.com/undertherain/benchmarker/, commit `e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d`.

[14] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.* 38, 8 (2012), 391–407. https://doi.org/10.1016/j.parco.2011.10.002

[15] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing* 74, 12 (2014), 3202–3216.

[16] Paul Feautrier and Christian Lengauer. 2011. Polyhedron Model. *Encyclopedia of parallel computing* (2011), 1581–1592.

[17] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M. Veras, Daniele G. Spampinato, Jeremy R. Johnson, Markus Püschel, James C. Hoe, and José M. F. Moura. 2018. SPIRAL: Extreme Performance Portability. *Proc. IEEE* 106, 11 (2018), 1935–1968. https://doi.org/10.1109/JPROC.2018.2873289

[18] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. https://doi.org/10.1145/277650.277725

[19] Fujitsu. 2021. https://www.fujitsu.com/downloads/SUPER/a64fx/a64fx_datasheet_en.pdf

[20] Fujitsu. 2022. https://github.com/fujitsu/dnnl_aarch64

[21] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. 2021. Domain-Specific Multi-Level IR Rewriting for GPU: The Open Earth Compiler for GPU-Accelerated Climate Simulation. *ACM Trans. Archit. Code Optim.* 18, 4, Article 51 (sep 2021), 23 pages. https://doi.org/10.1145/3469030

[22] Hwansoo Han, Chau-Wen Tseng, and Pete Keleher. 1998. Eliminating barrier synchronization for compiler-parallelized codes on software DSMs. *International journal of parallel programming* 26, 5 (1998), 591–612.

[23] Ruobing Han, Jaewon Lee, Jaewoong Sim, and Hyesoon Kim. 2022. COX: CUDA on X86 by Exposing Warp-Level Functions to CPUs. *ACM Trans. Archit. Code Optim.* (jul 2022). https://doi.org/10.1145/3554736

[24] Mark Harris et al. 2007. Optimizing parallel reduction in CUDA. *Nvidia developer technology* 2, 4 (2007), 70.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[26] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis. 2014. Achieving Portability and Performance through OpenACC. In *2014 First Workshop on Accelerator Programming using Directives*. 19–26. https://doi.org/10.1109/WACCPD.2014.10

[27] Chuntao Hong, Dehao Chen, Wenguang Chen, Weimin Zheng, and Haibo Lin. 2010. MapCG: Writing Parallel Program Portable between CPU and GPU. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* (Vienna, Austria) *(PACT '10)*. Association for Computing Machinery, New York, NY, USA, 217–226. https://doi.org/10.1145/1854273.1854303

[28] Intel. 2022. OneAPI Deep Neural Network Library (OneDNN). https://github.com/oneapi-src/oneDNN

[29] Pekka Jääskeläinen, Carlos Sánchez de La Lama, Erik Schnetter, Kalle Raiskila, Jarmo Takala, and Heikki Berg. 2015. pocl: A performance-portable OpenCL implementation. *International Journal of Parallel Programming* 43, 5 (2015), 752–785.

[30] Ralf Karrenberg and Sebastian Hack. 2012. Improving performance of OpenCL on CPUs. In *Compiler Construction*, Michael O'Boyle (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[31] Andreas Klöckner. 2014. Loo.Py: Transformation-Based Code Generation for GPUs and CPUs. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming (ARRAY'14)* (Edinburgh, United Kingdom). Association for Computing Machinery, New York, NY, USA, 82–87. https://doi.org/10.1145/2627373.2627387

[32] Maria Kotsifakou, Prakalp Srivastava, Matthew D. Sinclair, Rakesh Komuravelli, Vikram Adve, and Sarita Adve. 2018. HPVM: Heterogeneous parallel virtual machine. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 68–80. https://doi.org/10.1145/3178487.3178493

[33] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 75–86. https://doi.org/10.1109/CGO.2004.1281665

[34] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[35] Amy W. Lim and Monica S. Lam. 1997. Maximizing Parallelism and Minimizing Synchronization with Affine Transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 201–214. https://doi.org/10.1145/263699.263719

[36] LLVM Contributors. 2021. OpenMP-aware optimizations. Online: https://openmp.llvm.org/optimizations/OpenMPOpt.html.

[37] Simon Moll, Johannes Doerfert, and Sebastian Hack. 2016. Input Space Splitting for OpenCL. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 251–260. https://doi.org/10.1145/2892208.2892217

[38] Sungdo Moon and Mary W Hall. 1999. Evaluation of predicated array data-flow analysis for automatic parallelization. *ACM SIGPLAN Notices* 34, 8 (1999), 84–95.

[39] William Steven Moses. 2017. *How should compilers represent fork-join parallelism?* Master's thesis. Massachusetts Institute of Technology.

[40] William S. Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zinenko. 2021. Polygeist: Raising C to Polyhedral MLIR. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 45–59. https://doi.org/10.1109/PACT52795.2021.00011

[41] William S. Moses, Valentin Churavy, Ludger Paehler, Jan Hückelheim, Sri Hari Krishna Narayanan, Michel Schanen, and Johannes Doerfert. 2021. Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (St. Louis, Missouri) *(SC '21)*. Association for Computing Machinery, New York, NY, USA, Article 61, 16 pages. https://doi.org/10.1145/3458817.3476165

[42] Cosmin E Oancea and Lawrence Rauchwerger. 2012. Logical inference techniques for loop parallelization. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. 509–520.

[43] M. O'Boyle and E. Stohr. 2002. Compile time barrier synchronization minimization. *IEEE Transactions on Parallel and Distributed Systems* 13, 6 (2002), 529–543. https://doi.org/10.1109/TPDS.2002.1011394

[44] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[45] Atmn Patel, Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2021. A Virtual GPU as Developer-Friendly OpenMP Offload Target. In *50th International Conference on Parallel Processing Workshop* (Lemont, IL, USA) *(ICPP Workshops '21)*. Association for Computing Machinery, New York, NY, USA, Article 24, 7 pages. https://doi.org/10.1145/3458744.3473356

[46] Matt Pharr and William R Mark. 2012. ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13.

[47] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[48] Harenome Razanajato, Cidric Bastoul, and Vincent Loechner. 2017. Lifting Barriers Using Parallel Polyhedral Regions. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*. 338–347. https://doi.org/10.1109/HiPC.2017.00046

[49] Mitsuhisa Sato, Yutaka Ishikawa, Hirofumi Tomita, Yuetsu Kodama, Tetsuya Odajima, Miwako Tsuji, Hisashi Yashiro, Masaki Aoki, Naoyuki Shida, Ikuo Miyoshi, Kouichi Hirai, Atsushi Furuya, Akira Asato, Kuniki Morita, and Toshiyuki Shimizu. 2020. Co-Design for A64FX Manycore Processor and "Fugaku". In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00051

[50] Tao B Schardl, William S Moses, and Charles E Leiserson. 2019. Tapir: Embedding recursive fork-join parallelism into LLVM's intermediate representation. *ACM Transactions on Parallel Computing (TOPC)* 6, 4 (2019), 1–33.

[51] Adrian Schmitz, Julian Miller, Lukas Trümper, and Matthias S Müller. 2021. PPIR: Parallel Pattern Intermediate Representation. In *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 30–40.

[52] Alexander Sergeev and Mike Del Balso. 2018. Horovod: fast and easy distributed deep learning in TensorFlow. https://doi.org/10.48550/ARXIV.1802.05799

[53] Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, and Zvonimir Rakamarić. 2016. Portable Inter-Workgroup Barrier Synchronisation for GPUs. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Amsterdam, Netherlands) *(OOPSLA 2016)*. Association for Computing Machinery, New York, NY, USA, 39–58. https://doi.org/10.1145/2983990.2984032

[54] Tyler Sorensen, Lucas F Salvador, Harmit Raval, Hugues Evrard, John Wickerson, Margaret Martonosi, and Alastair F Donaldson. 2021. Specifying and testing GPU workgroup progress models. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–30.

[55] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing OpenMP Tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (Denver, CO, USA) *(LLVM-HPC'17)*. Association for Computing Machinery, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/3148173.3148186

[56] George Stelle, William S. Moses, Stephen L. Olivier, and Patrick McCormick. 2017. OpenMPIR: Implementing OpenMP Tasks with Tapir. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC* (Denver, CO, USA). ACM, New York, NY, USA, Article 3, 12 pages. https://doi.org/10.1145/3148173.3148186

[57] John A. Stratton, Vinod Grover, Jaydeep Marathe, Bastiaan Aarts, Mike Murphy, Ziang Hu, and Wen-mei W. Hwu. 2010. Efficient Compilation of Fine-Grained SPMD-Threaded Programs for Multicore CPUs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization* (Toronto, Ontario, Canada) *(CGO '10)*. Association for Computing Machinery, New York, NY, USA, 111–119. https://doi.org/10.1145/1772954.1772971

[58] John A. Stratton, Sam S. Stone, and Wen-mei W. Hwu. 2008. MCUDA: An Efficient Implementation of CUDA Kernels for Multi-core CPUs. In *Languages and Compilers for Parallel Computing*, José Nelson Amaral (Ed.). Vol. 5335. Springer, Berlin, Heidelberg, 16–30. https://doi.org/10.1007/978-3-540-89740-8_2 Series Title: Lecture Notes in Computer Science.

[59] Sander Stuijk, Marc Geilen, and Twan Basten. 2006. Sdf^ 3: Sdf for free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*. IEEE, IEEE, 276–278.

[60] Zehra Sura, Xing Fang, Chi-Leung Wong, Samuel P. Midkiff, Jaejin Lee, and David Padua. 2005. Compiler Techniques for High Performance Sequentially Consistent Java Programs. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 2–13. https://doi.org/10.1145/1065944.1065947

[61] Xinmin Tian, Hideki Saito, Ernesto Su, Jin Lin, Satish Guggilla, Diego Caballero, Matt Masten, Andrew Savonichev, Michael Rice, Elena Demikhovsky, Ayal Zaks, Gil Rapaport, Abhinav Gaba, Vasileios Porpodas, and Eric N. Garcia. 2017. LLVM Compiler Implementation for Explicit Parallelization and SIMD Vectorization. In *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. ACM, Denver, CO, USA, 4:1–4:11. https://doi.org/10.1145/3148173.3148191

[62] Chau-Wen Tseng. 1995. Compiler optimizations for eliminating barrier synchronization. *ACM SIGPLAN Notices* 30, 8 (1995), 144–155.

[63] Nicolas Vasilache, Benoit Meister, Muthu Baskaran, and Richard Lethin. 2012. Joint scheduling and layout optimization to enable multi-level vectorization. *IMPACT* 12 (2012).

[64] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (oct 2019), 26 pages. https://doi.org/10.1145/3355606

[65] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the Conflicting Demands of Parallelism and Temporal/Spatial Locality in Affine Scheduling. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 3–13. https://doi.org/10.1145/3178372.3179507

A Artifact

The evaluation of our results consists of three parts:

- A performance comparison of a CUDA matrix multiply code on CPU, as transpiled by our pipeline (Polygeist) and an existing tool (MCuda).
- An evaluation of CUDA benchmarks from the Rodinia suite on CPU, as transpiled by our pipeline (Polygeist), and a comparison to the native CPU versions of the same benchmarks when available.
- An evaluation of the GPU kernels within PyTorch being replaced with CPU versions of the kernels.

Obtaining the code. A meta repository containing dockerfiles and source for our set up is available at https://github.com/wsmoses/PolygeistGPU-Docker with DOI 10.5281/zenodo.7508499. The remainder of this section will describe the individual components of our system.

Code for our tool is available at https://github.com/llvm/Polygeist, commit `4a232df859` and is obtained as follows:

```
$ cd $HOME && git clone https://github.com/llvm/Polygeist
$ cd Polygeist
$ git checkout 4a232df859
$ git submodule update --init --recursive
```

This repository contains submodules for a corresponding version of LLVM/MLIR/Clang, which is automatically checked out by the previous command.

A fork of the Rodinia benchmark suite with scripts for timing as well as the matrix multiplication tests are found at https://github.com/ivanradanov/rodinia at commit `025fa7dc`.

```
$ cd $HOME && git clone https://github.com/ivanradanov/rodinia
$ cd rodinia
$ git checkout 025fa7dc
```

The Cpucuda runtime dependency of the matrix multiplication comparison found at https://github.com/ivanradanov/cpucuda_runtime at commit `265fe49`:

```
$ cd $HOME
$ git clone https://github.com/ivanradanov/cpucuda_runtime
$ cd cpucuda_runtime
$ git checkout 265fe49
```

The MocCUDA layer for PyTorch integration can be found at https://gitlab.com/domke/MocCUDA at commit `5a3955d`:

```
$ cd $HOME && git clone https://gitlab.com/domke/MocCUDA
$ cd MocCUDA
$ git checkout 5a3955d
```

To evaluate the artifact, we offer three options.

1. The first option is an AMI or Amazon Machine Image under image ID `ami-016572c3eb2ab565a`. You may then launch the instance and then skip the rest of this section that involves downloading or building the experiments.
2. The second option is to build the tools and experiments from source and is outlined below.
3. The third option is to use a Docker container. The docker container contains pre-built versions of the relevant binaries, a collection of all the benchmarks, and so on. As such any benchmark downloading or building of Polygeist/LLVM described here can be skipped, however the instructions on how to run the scripts are the same. The source for the Docker images is available at https://github.com/wsmoses/PolygeistGPU-Docker and can be run by executing the following command:

```
# Rodinia and MCUDA
$ docker run -i -t ivanradanov/polygeistgpu /bin/bash
# MocCUDA on x86_64
$ docker run -i -t ivanradanov/moccuda /bin/bash
```

We begin by installing build dependencies (C++ compiler, cmake, ninja). This can be done on Ubuntu 20.04 with the following command:

```
$ sudo apt-get install -y cmake \
  gcc g++ ninja-build
```

llvm-project. We now need to build the LLVM compiler toolchain. To install LLVM, please follow the following steps:

```
$ cd $HOME/Polygeist
$ mkdir mlir-build && cd mlir-build
$ cmake ../llvm-project/llvm -GNinja \
 -DCMAKE_BUILD_TYPE=Release \
 -DLLVM_ENABLE_PROJECTS="mlir;clang;openmp" \
 -DLLVM_TARGETS_TO_BUILD="X86"
# This may take a while
$ ninja
```

Polygeist. We now must build Polygeist based off of the LLVM version we just built.

```
$ cd $HOME/Polygeist
$ export MLIR_BUILD=`pwd`/mlir-build
$ mkdir build
$ cd build
$ cmake .. -GNinja \
 -DCMAKE_BUILD_TYPE=Release \
 -DMLIR_DIR=$MLIR_BUILD/lib/cmake/mlir \
 -DClang_DIR=$MLIR_BUILD/lib/cmake/clang
$ ninja
# cgeist will now be available at
#   $HOME/Polygeist/build/bin/mlir-clang
```

Cpucuda runtime. To build and install this dependency one can follow these steps:

```
$ cd $HOME/cpucuda_runtime
$ mkdir build
$ cd build
$ cmake .. -DCUDA_PATH=/usr/local/cuda \
 -DCMAKE_CXX_COMPILER=clang++ \
 -DCMAKE_C_COMPILER=clang
$ make
$ cp src/libcpucudart.a $HOME/rodinia/mcuda-test/mcuda/libcpucuda.a
```

Disabling/Enabling Hyperthreading. We recommend disabling hyperthreading, and provide two scripts for this purpose, assuming a dual-socket 32-core machine.

```
$ cd $HOME/rodinia/scripts
$ ./disable.sh
```

Benchmark Configuration. The Rodinia and MCUDA benchmarks use configuration files in `rodinia/common/` to specify Polygeist, Clang/LLVM, and other installations. The config files for the machine we used are `ubuntu.polygeist.host.make.config` for the CUDA versions of benchmarks and `ubuntu.polygeist-clang.openmp.host.make.config` for openmp versions. The structure of the filename must be kept the same, with the `ubuntu` substring representing the machine's hostname. One must set five variables for the first file and two for the second. `POLYGEIST_DIR` should denote the build directory of Polygeist. `POLYGEIST_LLVM_DIR`

should denote the build directory of LLVM. CUDA_PATH should denote a valid CUDA path. CPUCUDA_BUILD_DIR should denote the build directory of the cpucuda_runtime built above. CUDA_SAMPLES_PATH should denote the directory of CUDA samples in a CUDA installation. Note that even when running on a machine without a GPU, one still needs the header files from a functioning CUDA installation as Rodinia uses several of the helper functions defined within. On the AMI and docker containers, we have provided such a CUDA installation. If building from source on a machine without a GPU, a CUDA installation can be copied from another system.

```
POLYGEIST_DIR=${HOME}/Polygeist/build/
POLYGEIST_LLVM_DIR=${HOME}/Polygeist/mlir-build/
CPUCUDA_BUILD_DIR = ${HOME}/src/cpucuda_runtime/build/
CUDA_PATH = /usr/local/cuda/
CUDA_SAMPLES_PATH = /usr/local/cuda/samples
```

To conclude configuration, symlink the configuration files:

```
$ cd $HOME/rodinia/common
$ ln -s ubuntu.polygeist.host.make.config host.make.config
$ ln -s ubuntu.polygeist-clang.openmp.host.make.config \
   openmp.host.make.config
```

To configure the benchmark run, one should edit the file rodinia/scripts/run_all_benches.sh. The HOST variable should be set to the hostname of the machine (ubuntu above). The NRUNS and NRUNS_SCALING variables should denote the number of runs of the ablation analysis and scaling analysis benchmarks, respectively. The variables THREAD_NUMS and THREAD_NUMS_OPENMP should contain a list of the number of threads to run the CUDA and OpenMP scaling tests, respectively.

Rodinia. The Rodinia benchmarks can be compiled and executed by running the following script. Note that the script assumes a specific machine size, but can be edited.

```
$ cd $HOME/rodinia
$ ./scripts/run_all_benches.sh
# The timing results are now at $HOME/rodinia_results/
```

The correctness of the generated code can be validated as follows. Note that this requires access to a GPU machine to run the GPU-versions of the programs.

On a GPU machine with relevant configuration set up (see rodinia/common/ kiev0.nvcc.host.make.config for an example):

```
$ cd $HOME/rodinia
$ make MY_VERIFICATION_DISABLE=0 cuda
$ ./scripts/dump_cuda_correctness_info.sh
# Now ./verification_data contains the data
```

Ensure that the verification data is available on the machine used to test GPU to CPU. If this is the same machine, no action is required. Otherwise, one can copy the verification folder. Restore the configuration for Polygeist as outlined above in the configuration section, and execute the following:

```
$ cd $HOME/rodinia
$ make MY_VERIFICATION_DISABLE=0 cuda
$ ./scripts/check_cuda_correctness.sh
```

Verification is successful if no FAIL can be seen in the output of the final script.

Matrix Multiplication (MCUDA). The following commands will compile and execute the matrix multiplication tests for both Polygeist and MCUDA. The arguments to the script are a list of matrix sizes, a list of thread numbers, and the number of runs. The timing data will be output in mm_results.py.

```
$ cd $HOME/rodinia/mcuda-test
$ make
$ ./run-scaling.sh "128 256 512 1024 2048" \
  "1 2 4 8 16 24" 10 > mm_results.py
```

MocCUDA. This section describes how to build the MocCUDA layer for Fugaku. To build MocCUDA for another system, one can edit the files in ./scripts/ to reflect the environment of their system. MocCUDA dependencies, including PyTorch and benchmarker can be built as follows.

```
$ cd $HOME/MocCUDA
$ for NR in $(seq -w 00 06); do
$       bash ./scripts/${NR}_*.sh
$ done
```

MocCUDA itself is built with the following script:

```
$ bash ./scripts/07_*.sh
```

And only required on Fugaku, the following will set up Fujitsu's custom pytorch:

```
$ bash ./scripts/08_*.sh
```

The following script will submit benchmark jobs to Fugaku and populate the MocCUDA/log directory with the results.

```
$ bash ./bench/submit_fugaku.sh
```

Alternatively, one can use a Fugaku-style MocCUDA docker container for X86 which we have created. The following command will training a single-node resnet50 in docker. Options for the BACKEND variable include moccuda, moccuda-no-polygeist, native, or dnnl. Results will be output in the MocCUDA/log directory.

```
$ cd $HOME/MocCUDA/
$ BACKEND=<backend> ./bench/02_benchmarker_train.sh
```