

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

William S. Moses<sup>1</sup>, Ivan R. Ivanov<sup>2</sup>, Jens Domke<sup>3</sup>, Toshio Endo<sup>4</sup>, Johannes Doerfert<sup>5</sup>, and Oleksandr Zinenko<sup>6</sup>

<sup>1</sup>Massachusetts Institute of Technology, USA wmoses@mit.edu

<sup>2</sup>Tokyo Institute of Technology, Japan ivanov.i.aa@m.titech.ac.jp, endo@is.titech.ac.jp

<sup>3</sup>RIKEN Center for Computational Science, Japan jens.domke@riken.jp

<sup>5</sup>Argonne National Laboratory, USA jdoerfert@anl.gov

<sup>6</sup>Google, France zinenko@google.com

**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 and 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 76% geomean speedup over hand-written 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 $\times$ .

## I. INTRODUCTION

The end of single-core performance scaling means that parallelism and domain-specificity are now the main sources of efficiency increases. Supercomputer architects compete in ingenuity to support compute- and memory-intensive applications from physics simulations to machine learning. The latest and fastest supercomputer, Fugaku, is based exclusively on A64FX CPUs that, unlike commodity CPUs, provide support for high-bandwidth memory access and energy efficiency comparable to that of recent GPUs [1].

However, efficient and productive use of such computers for practical applications is challenging as recent frameworks and high-performance libraries have been developed with NVidia GPUs in mind. For example, attempts to port PyTorch [2] to A64FX have met multiple challenges. The “native” default CPU PyTorch backend provides only naive versions for critical kernels, such as 2D convolution implemented as six nested loops. Intel’s oneDNN [3] unsurprisingly performs poorly for Arm CPUs since it is tailored for commodity CPUs without high-bandwidth memory. Fujitsu’s customized oneDNN [4] is better tuned, but not universally competitive with GPUs.

Many non-library approaches to achieve performance portability have been put forth. They range from language extensions such as OpenCL [5] or OpenACC [6] to parallel programming frameworks such as Kokkos [7], to domain-specific languages such as SPIRAL [8], Halide [9] or Tensor Comprehensions [10]. All these approaches still require legacy applications to be ported, and sometimes entirely rewritten, due to differences in the language, or the underlying programming model, of the original program and the target framework.

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 [11]–[13], 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

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

William S. Moses<sup>1</sup>, Ivan R. Ivanov<sup>2</sup>, Jens Domke<sup>3</sup>, Toshio Endo<sup>4</sup>, Johannes Doerfert<sup>5</sup>, and Oleksandr Zinenko<sup>6</sup>

<sup>1</sup>Massachusetts Institute of Technology, USA wmoses@mit.edu

<sup>2</sup>Tokyo Institute of Technology, Japan ivanov.i.aa@m.titech.ac.jp, endo@is.titech.ac.jp

<sup>3</sup>RIKEN Center for Computational Science, Japan jens.domke@riken.jp

<sup>5</sup>Argonne National Laboratory, USA jdoerfert@anl.gov

<sup>6</sup>Google, France zinenko@google.com

**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 and 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 76% geomean speedup over hand-written 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 $\times$ .

```
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);
}
```

## I. INTRODUCTION

The end of single-core performance scaling means that parallelism and domain-specificity are now the main sources of efficiency increases. Supercomputer architects compete in ingenuity to support compute- and memory-intensive applications from physics simulations to machine learning. The latest and fastest supercomputer, Fugaku, is based exclusively on A64FX CPUs that, unlike commodity CPUs, provide support for high-bandwidth memory access and energy efficiency comparable to that of recent GPUs [1]. However, efficient and productive use of such computers for practical applications is challenging as recent frameworks and high-performance libraries have been developed with NVidia GPUs in mind. For example, attempts to port PyTorch [2] to A64FX have met multiple challenges. The “native” default CPU PyTorch backend provides only naive versions for critical kernels, such as 2D convolution implemented as six nested loops. Intel’s oneDNN [3] unsurprisingly performs poorly for Arm CPUs since it is tailored for commodity CPUs without high-bandwidth memory. Fujitsu’s customized oneDNN [4] is better tuned, but not universally competitive with GPUs.

```
__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);
}
```

§2 1. A sample CUDA program `normalize`, which normalizes a vector and the CPU function `launch` which calls the kernel. Presently, the call to `sum` is called in each thread, leading to a total of  $O(N^2)$  work. The work can be partially reduced to  $O(N^2/B)$  through the use of shared memory (in the comments above), which enables the sum to be computed once per block, or completely reduced to  $O(N)$  by computing `sum` once before the kernel.

loop-invariant code motion, are hindered by the lack of analyzable representations of parallel constructs inside the compiler [14]. As an example, consider the summation in Figure 1 which could be done by a single thread per block, or by a single thread in total once all computation happens on the CPU. As representations of parallelism within a mainstream compiler have only recently begun to be explored in a mainstream compiler [15]–[19], 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. This differs from CPU parallelism, which provides a single level of parallelism, a unified memory and peer synchronization. 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 entirely from memory semantics. This both allows synchronization-based code to inter-operate with existing optimizations and enables novel parallel-specific optimizations.

Our model is implemented in the MLIR layer [20] of the LLVM compiler infrastructure [21] and it leverages MLIR’s nested-module approach for GPU codes [22]. We extended the Polygeist [23] C/C++ frontend to support CUDA and to produce MLIR which preserves high-level parallelism and program 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 uses our approach to compile and execute CUDA kernels in absence of a GPU while substituting unsupported calls.

The correctness and efficiency of our end-to-end translation is evaluated by compiling Rodinia CUDA benchmarks [24] as well as 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 by the `conv2d` kernel from Resnet-50 [25] compared to the OneDNN-based PyTorch CPU backend, and comparable performance to the hand-tuned overall training.

Overall, our paper makes the following contributions:

A common high-level and platform-agnostic representation of SIMD-style parallelism backed by a semantic definition of barrier synchronization that ensures correctness through memory semantics, which ensures 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++ frontend for MLIR which is capable of directly mapping GPU and CPU parallel constructs into our high-level parallelism primitives. An end-to-end transpilation of CUDA to CPU for a subset of the Rodinia [24] benchmark suite and the internal CUDA kernels within PyTorch [2] necessary to run a Resnet-50 on the CPU-only Fugaku supercomputer.

## II. BACKGROUND

Mainstream compilers like Clang and GCC lack a unified high-level representation of parallelism. Specifically, 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 the respective runtime. Concepts such as thread index or synchronization are then represented separately, often through opaque intrinsic

[illegible]
$$; S \quad i \in V_2 \nsubseteq Z_i \nsubseteq V_2 \subset V_2.$$

$\Gamma \in \mathbb{Z}_n$ ,  $\Gamma \in \mathbb{Z}_n$  s.t.  $\Gamma \equiv 1/2 \pmod{n}$ . representation  
 $\Gamma \in \mathbb{Z}_n$ . e.g.,  $\Gamma \in \mathbb{Z}_n$ ,  $\Gamma \in \mathbb{Z}_n$  s.t.  $\Gamma \equiv 1/2 \pmod{n}$

i 2 1/2 " (   
 ° v L y Š i 2 1/2 i 1/2 B v L i 2   
 •   
 i Polygeist C/C++Mi i 1/2 U i 1/2 GPU i CPU v   
 L i 1/2 1/2 O i 1/2 S v L i 1/2   
 i Romania [24] i 1/2 1/2 \* P i 1/2 PyTorch   
 [2]- i U Resnet-50@ " ... i CUDA... 8 i 1/2   
 L i O 1/2 i CUDA O CPU, i i 1/2 i 1/2   
 CPU, Fugaku... S i —: i 1/2

## II. BACKGROUND

; A i h 1/2 Clang E GCC : O i 1/2 E B!  
 v L' parallelism h • wS ( i 1/2 E CUDA  
 OpenMP SYCL I F ¶ - i 1/2 „ i 1/2 6 v L  
 : i 1/2 S X ( Ž \* U E 1/2 closure i p 1/2 i E 1/2 1/2  
 p 1 i E 1/2 i E 1/2 1/2 i " thread index  
 e synchronization l , i 1/2 B « U E 1/2 • € €  
 1/2 1/2 „ ... n ( intrinsic calls 1 Ž i h 1/2 i 1/2  
 : O s Ž v L' E @ % i i 1/2 1/2 1/2 H E i 1/2 L  
 1/2 \_ i 1/2 : „ œ • = i i t E i 1/2 1/2 >  
 i [14-19], [26] e 9 „ CPU v L „ „ h • F h  
 1/2 & e t • „ E 1/2 y ! < E B „ ... X B  
 ! i 1/2 1/2 1/2 i h 1/2 GPU v L' parallelism „ E 1/2  
 B ! h • i 6 E „ E 1/2

### A. GPU Compilation

Q1 1/2, CUDA, 1/2 1/2 1/2 1/2 1/2  
R S ( LLVM/Clang 1/2 GPU / \* 1/2  
1/2 1/2 1/2, 1/2 1/2: 1/2 1/2 GPU ...8 CPU  
( 1/2 1/2 1/2 1/2 ( 1/2 1/2 1/2 1/2 1/2 1/2  
-, , ; 1/2 1/2  $O(N^2)$  1/2 1/2 1/2 1/2  $O(N)$   
-, sum ( 6,  $\alpha(\dots 8 (KM)$  1/2 1/2 !  
sum ( <, 1/2 1/2 1/2 1/2 1/2 LICM  
1/2 1/2 1/2 1/2 1/2  $O(N)$  1/2 1/2 1/2 \* Hoef  
1/2 1/2 1/2 1/2 1/2 1/2  $q \ll \dots X$  1/2 1/2 1/2  $O(\frac{N^2}{B})$   
MLIR: GPU 1/2 1/2 L W! Wh representation  
/ ; ; 1/2 1/2 1/2 1/2 1/2 1/2 F \*  $\mathbb{Z}^o \vee L$  1/2 1/2 1/2  
( GPU O CPU, 1/2 1/2 1/2 LICM 1/2 1/2 1/2 1/2 1/2 1/2  
1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 X ( ; ; 1/2 1/2

### B. MLIR Infrastructure

[illegible]

```
target triple = "x86_64-unknown-linux-gnu"

define @launch(float* %d_out, float* %d_in, i32 %n) {
    call @__cudaPushCallConfiguration(...)
    call @__cudaLaunchKernel(@normalize_stub, ...)
    ret
}

target triple = "nvptx64"

define @normalize(float* %out, float* %in, i32 %n) {
    %tid = call i32 @llvm.nvvm.ptx.tid.x()
    %sum = call i32 @sum(i32* %in, i32 %n)
    %cmp = icmp slt i32 %tid, %n
    br i1 %cmp, label %body, label %exit

body:
    %gep = getelementptr float* %in, i32 %tid
    %load = load float, float* %gep
    %nrm = fdiv float %load, %sum
    %ptr = getelementptr float* %out, i32 %tid
    store float %nrm, float* %ptr
    br label %exit

exit:
    ret
}
```

Fig. 2. Simplified lowering of the launch and normalize functions from Figure 1, when compiled with LLVM/Clang. As the two functions emit different assembly codes, they are placed in separate modules with no context for how, or if, they are called.

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 [14]–[19], [26] in recent years to improve representations for CPU parallelism constructs, accelerator programming comes with additional challenges. The unique programming model and complex memory hierarchy have caused high-level representations of GPU parallelism within mainstream compiler remain under-explored.

A. GPU Compilation

Consider the CUDA program in Figure 1, which normalizes a vector. When compiled using LLVM/Clang, the GPU program is a separate compilation unit, as shown in Figure 2 This prevents any optimization between the GPU kernel and the CPU calling code. In the case of Figure 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 could be performed only once prior to the kernel call, e.g., by performing the loop-invariant code motion (LICM) transformation, the work would reduce to  $O(N)$ . A

```
// 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>
            }
            // Synchronization via explicit operation.
            polygeist.barrier(%tx, %ty, %tz)
            %tid = %gx + grid.x * %tz
            if %tid < %n {
                %res = ...
                store %res, %d_out[%tid] : memref<?xf32>
            }
        }
    }
}
```

Fig. 3. Representation of the shared-memory variant of the CUDA launch/normalize code from Figure 1 in Polygeist/MLIR. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit through the use of parallel for loops across the blocks and threads, and shared memory is placed within the block to signify it can be accessed from any thread in the same block, but not from a different block.

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 [22], but parallel code motion has not been implemented. In GPU to CPU code motion, LICM out of a parallel loop is always legal as formerly device memory would also be available on the host.

B. MLIR Infrastructure

MLIR is a recent compiler infrastructure designed for reuse and extensibility [20]. Its internal representation (IR) can be seen as a conceptual successor on the widely adopted LLVM IR [21] with better support for programming models beyond the conventional CPU one. Rather than providing a

```
target triple = "x86_64-unknown-linux-gnu"

define @launch(float* %d_out, float* %d_in, i32 %n) {
    call @__cudaPushCallConfiguration(...)
    call @__cudaLaunchKernel(@normalize_stub, ...)
    ret
}

target triple = "nvptx64"

define @normalize(float* %out, float* %in, i32 %n) {
    %tid = call i32 @llvm.nvvm.ptx.tid.x()
    %sum = call i32 @sum(i32* %in, i32 %n)
    %cmp = icmp slt i32 %tid, %n
    br i1 %cmp, label %body, label %exit

body:
    %gep = getelementptr float* %in, i32 %tid
    %load = load float, float* %gep
    %nrm = fdiv float %load, %sum
    %ptr = getelementptr float* %out, i32 %tid
    store float %nrm, float* %ptr
    br label %exit

exit:
    ret
}
```

Fig. 2. LLVM/Clang representation of the launch/normalize functions from Figure 1, when compiled with LLVM/Clang. As the two functions emit different assembly codes, they are placed in separate modules with no context for how, or if, they are called.

Kernel launch is available within the calling function, enabling optimizations across the GPU/CPU boundary. Parallel for across all blocks in a grid. Parallel for across all threads in a block. Control-flow is directly preserved. Synchronization via explicit operation. Polygeist barrier. tid = gx + grid.x \* tz. if tid < n { res = ... store res, %d\_out[tid] : memref<?xf32> }

MLIR representation of the shared-memory variant of the CUDA launch/normalize code from Figure 1 in Polygeist/MLIR. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit through the use of parallel for loops across the blocks and threads, and shared memory is placed within the block to signify it can be accessed from any thread in the same block, but not from a different block.

```
// 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>
            }
            // Synchronization via explicit operation.
            polygeist.barrier(%tx, %ty, %tz)
            %tid = %gx + grid.x * %tz
            if %tid < %n {
                %res = ...
                store %res, %d_out[%tid] : memref<?xf32>
            }
        }
    }
}
```

Fig. 3. Polygeist/MLIR representation of the launch/normalize functions from Figure 1, when compiled with Polygeist/MLIR. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit through the use of parallel for loops across the blocks and threads, and shared memory is placed within the block to signify it can be accessed from any thread in the same block, but not from a different block.

C. Polygeist

Polygeist is a compiler infrastructure designed for reuse and extensibility [20]. Its internal representation (IR) can be seen as a conceptual successor on the widely adopted LLVM IR [21] with better support for programming models beyond the conventional CPU one. Rather than providing a

III. APPROACH

Polygeist representation of the launch/normalize functions from Figure 1, when compiled with Polygeist/MLIR. The kernel call is made available directly in the host code which calls it. The parallelism is made explicit through the use of parallel for loops across the blocks and threads, and shared memory is placed within the block to signify it can be accessed from any thread in the same block, but not from a different block.



predefined set of instructions and types, MLIR operates on collections of *dialects* that contain sets of 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) [27]. For example, MLIR dialects may model entire physical or virtual instruction sets such as NVVM (virtual IR for Nvidia GPUs) , other IRs such as LLVM IR, higher-level control flow constructs such as affine loops, parallel programming models such as OpenMP and OpenACC, machine learning graphs, etc. MLIR supports and encourages to mix operations from different dialects in the same compilation unit.

MLIR supports GPU thanks to the eponymous dialect, which defines the high-level SIMT programming model as well as host/device communication mechanisms, and a set of low-level platform-specific dialects: NVVM (CUDA), ROCm (ROCm) and SPIR-V. The particularity of MLIR’s approach to GPU programming consists in its *unified* code representation. Thanks to the flexibility of the IR, a module may contain other modules, e.g., 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 by freely moving code between host and device [22].

### C. Polygeist

Polygeist is a C and C++ frontend for MLIR based on Clang [23]. 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. In particular, Polygeist preserves structured control flow (loops and conditionals) as MLIR SCF dialect. It also simplifies analyses by preserving multi-dimensional array constructs whenever possible by relying on the MLIR’s multi-dimensional memory reference (memref) type. Finally, Polygeist is able to identify parts of the program suitable for polyhedral optimization [28] and represent them using the Affine dialect.

### III. APPROACH

We extended the Polygeist compiler [23] to directly emit parallel MLIR from CUDA. This leverages the unified CPU/GPU representation to allow the optimizer to understand

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

1/2 4. A program containing a barrier between two arbitrary instructions.

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. 3):

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/ROCm 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.

### A. Barrier Semantics

A CUDA or ROCm `__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 such 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

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

ii  $4\frac{1}{2}$  ( \$ \* i  $\frac{1}{2}$ i  $\frac{1}{2}$ K  $\frac{1}{2}$  \* œ • # •

$\forall (\dots \delta_i \in \mathbb{K}^i \in \mathbb{Z}^{\frac{1}{2}} \quad \circ \quad \text{MLIR}_{\text{scf}}, \quad \text{sv} L_{\text{scf}}$   
 $\text{scf}, \text{parallel} \rightarrow \text{affine.parallel} \quad \text{scf} \cdot (\cdot$   
 $i \in \mathbb{Z}^{\frac{1}{2}} \mathbb{Z}^{\frac{1}{2}} \in \mathbb{Z}^{\frac{1}{2}}, \text{CPU} \leftrightarrow \text{GPU} \quad i \in \mathbb{Z}^{\frac{1}{2}} \quad \text{MLIR}_{\text{scf}}, i \in \mathbb{Z}^{\frac{1}{2}}$   
 $U \in \mathbb{K}^i \in \mathbb{A}^{\frac{1}{2}} \quad i \in \mathbb{Z}^{\frac{1}{2}} \in \mathbb{Z}^{\frac{1}{2}} \quad i \in \mathbb{Z}^{\frac{1}{2}} \quad i \in \mathbb{Z}^{\frac{1}{2}} \quad \text{CEI} \in \mathbb{Z}^{\frac{1}{2}}$   
 $|$

GPU...8/ " , h • representation Š I ,  
 , i 302

( Q< - „ @ W İ ½ 13D v L for İ ½  
İ ½ 134 « ...Xİ ½ M İ P ½ İ ½ W/

( \* W- „ @ ė ī ħ 3Dv L forī ė ½  
 \* ī Š ½„ Polygeist Polygeist OœĖ Ÿ ĩ ½I  
 Ž CUDA/ROCm CUDA/ROCm e „ ī ħ ½

[illegible]

### A. Barrier Semantics

[illegible]

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. 4. 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 then enumerate the three 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 case 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 would ensure correctness of case 2. Thus, the union of the writing behaviors of codeA and codeB is sufficient to prevent loads from being incorrectly moved across the barrier.

However, this does not prevent writes from being moved. For example, codeB could be duplicated above the barrier in case 2, 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 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 can be done by exploring the operations within predecessors or successors, respectively. However, operating on MLIR’s structured control flow level, with explicit operations for loops and conditionals, allows the analysis to be simplified. 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. Barriers order 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. Raising memory accesses into linear forms available in the Affine dialect, when possible, enables precise analysis. Consider the code in Fig. 5. The read and write expressions around the barrier have the affine access sets

```
__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
}
```

Fig. 5. Barrier semantics can be refined to accessing memory addresses accessed by operations above/below it in all threads *except* the current one.

$A_o = fA(i) : i = txg$  where  $tx$  is the thread  $\times$  identifier. The barrier has the affine access set  $A_b = fA(i) : i \notin txg$ . Since the sets of accessed addresses do not overlap,  $A_o \setminus A_b = \emptyset$ , code motion across the barrier is allowed. Conceptually, the write to  $A[threadIdx.x]$  always happens before the read within the same thread so the barrier is unnecessary. In contrast, if the load and 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. More generally, when defining the memory properties of a given barrier by collecting the memory locations and effect types from all relevant operations, one only needs to include memory locations used by a different thread. Aliasing guarantees must be checked when more than one base address is involved.

B. 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 III-A 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 is different from the GPU execution model in which numerous explicit 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. As discussed in Section VII, several approaches have been explored in the past including loop fission and continuation passing. Our approach is an extension of the former combining two styles

```
__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
}
```

Fig. 5. Barrier semantics can be refined to accessing memory addresses accessed by operations above/below it in all threads *except* the current one.

B. Barrier Lowering

1) *Parallel Loop Splitting*: Given a parallel loop with a barrier, we can split the loop into two parts, one before the barrier and one after. The first part can be executed on the CPU, and the second part can be executed on the GPU. This ensures that the barrier is respected, as the CPU part will complete before the GPU part starts.

2) *Parallel Loop Interchange*: If the barrier is at the end of the loop, we can interchange the loop order. This allows the GPU to execute the entire loop before the CPU, ensuring the barrier is respected.

3) *Parallel Loop Fission*: We can fission the loop into smaller chunks, each of which can be executed on the CPU. This ensures that the barrier is respected, as the CPU will complete before the GPU starts.

4) *Continuation Passing*: We can pass the continuation of the loop to the GPU, ensuring that the barrier is respected.

```
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)
    ...
}
```

1) *Parallel Loop Splitting*: Given a parallel loop with a barrier, we can split the loop into two parts, one before the barrier and one after. The first part can be executed on the CPU, and the second part can be executed on the GPU. This ensures that the barrier is respected, as the CPU part will complete before the GPU part starts.

2) *Parallel Loop Interchange*: If the barrier is at the end of the loop, we can interchange the loop order. This allows the GPU to execute the entire loop before the CPU, ensuring the barrier is respected.

3) *Parallel Loop Fission*: We can fission the loop into smaller chunks, each of which can be executed on the CPU. This ensures that the barrier is respected, as the CPU will complete before the GPU starts.

4) *Continuation Passing*: We can pass the continuation of the loop to the GPU, ensuring that the barrier is respected.



Figure 6. Example of a parallel loop splitting around a barrier. Here the code above the barrier (comprising 2 loads and 3 floating point operations) is placed in a separate parallel for loop from the code following the barrier (comprising a call to @use and other operations). This transformation eliminates the barrier, while preserving the semantics. As the values %a, %b, and %c are used after the barrier, extra care must be taken to ensure they are available. Here, the min-cut algorithm preserves %x and %y, and then recompute %a, %b, and %c as this would result in 2 values being preserved rather than 3.

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 [29] to determine the minimum amount of data that needs to be stored. Specifically, we can represent the problem by creating 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. By performing a minimum branch cut on this graph, we can derive the minimum amount of data that needs to be stored.

Fig. 7. **Left:** A shared memory addition, which consists of a kernel call which contains for loop with a barrier inside. **Right:** The same code with the barrier now directly within the parallel loop by performing an interchange of the parallel for loop with the serial for loop.

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 immediately 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 latter immediate parent of the barrier.

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.

(MLIR- for  $\bar{i} \in \frac{1}{2} \mathbb{Z}$   $\bar{i} \in \mathbb{M}_i$   
 $\rightarrow \bar{i} \in \frac{1}{2} \mathbb{Z}$  while  $\bar{i} \in \frac{1}{2} \mathbb{Z}$   $\bar{i} \in \frac{1}{2} \mathbb{Z}$   
 $\bar{i} \in \frac{1}{2} \mathbb{Z}$ ,  $Q_i \in \frac{1}{2} \mathbb{Z}$  -  $\bar{i} \in \frac{1}{2} \mathbb{Z}$ :  $\bar{i} \in \frac{1}{2} \mathbb{Z}$   $\text{cn}$ '  
 $\bar{i} \in \frac{1}{2} \mathbb{Z}$   $\bar{i} \in \frac{1}{2} \mathbb{Z}$  -  $\text{gl\_condition}()$   
 call in every thread  
 so a direct interchange  
 would not be legal.  
 However, the number of  
 iterations must be equal  
 across all threads due to  
 the GPU synchronization  
 semantics. Therefore, the  
 interchange is possible  
 thanks to a helper  
 variable which stores the  
 result of the condition  
 from one thread that is  
 used to decide whether  
 another iteration is  
 required.

C. Usage

```
CUDA 11.2.1 Transpilation
Polygeist -i  -o  -A 1/2
Polygeist \: ° CUDA 11.2.1
, clang -i  -o  -Z • ( wS
Polygeist -i  clang -M  -o 
d • ( clang -i  -x  -E  -o 
Polygeist -i  -o  -x  -o 
, -cuda-lower (Ž š GPU O
CPU -l b -i  -o  -c pui fy=XX (Ž
š (Ž CPU -y š -i  -o  -l V
-i  -l V ,
```

Pol ygei st / MLI R  $\bar{t} \frac{1}{2}$   
 „ v L ' Ē GPU • „  $\bar{t} \frac{1}{8} h \cdot$   
 representation •  $\bar{t} \frac{1}{2}$  :  
 $\bar{t} \frac{1}{2} \frac{1}{2} \frac{1}{2} \frac{1}{2} \frac{1}{2} \frac{1}{2} ( \bar{t} \frac{1}{2} \frac{1}{2} L \cdot „ , \bar{t} \frac{1}{2}$   
 $\bar{t} \frac{1}{2} GPU \text{ I b: } CPU „ \bar{t} \frac{1}{2} „ yš$

1 Ž GPU7 „ Oœi Ł ½+I b i Ł ½ CPU ¶  
 „ i g ½ d € i ½ Oœi Ł ½ > W „ q Ł ½ d  
 s • ( GPU i Ł GPU i Ł ½ Ł Oœ „ ^ d \_ /  
 ^ 8 ( „ i Ł ½ Ł e i Ł ½ Ł L ' ' i Ł ½ Ž  
 Oœ ^ d € „ i @ ½ Ł ½ Ł • Ž , I I I - A ,  
 - š I „ v ... X L : i š ½ \* Oœ B i M<sub>before</sub> ½  
 ( B K M i Ł ½ Ł ½ Oœ v L : i Ł ½ Ł ... X H œ  
 „ v i Ł ½ Ł B K i Ł ½ L : i Ł ½ Ł ... X H œ „  
 v i Ł M<sub>after</sub> , œ ( Oœ Ł ½ \$ M n j ...  
 X H œ , d † i Ł ½ Ł ½ Ł R A R K s ( M<sub>before</sub> \  
 M<sub>after</sub> ) n R A R = ? i Ł Oœ „ L : « H M „ Oœ @  
 i Ł ½ Ł ½ Ł ½ d i Ł ½ M<sub>before</sub> \ M<sub>after</sub> n R A R =  
 ? h : i Ł « i Ł Oœ @ i Ł ½ Ł ½ Ł Oœ „ \* y  
 š € U i Ł ½ O h j ... X H œ , Oœ

```

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

%hel per = alloca memref<i 1>
scf.do {
  parallel for %i=0 to N {
    run(%i)
    barrier
    %c = condition()
    if %i == 0 {
      store %c, %hel per[]
    }
  }
  %c = load %hel per[]
} while(%c)

```

Fig. 8. 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.

to decide whether another iteration is required.

This illustrates one of the advantages of building such a system within MLIR/Polygeist. By being able preserve the high level structure of the program we can use more efficient patterns to remove barriers.

#### C. Usage

CUDA transpilation with Polygeist is designed to be easy to use by allowing Polygeist to serve as a drop in replacement for an existing CUDA compiler like clang. Specifically, Polygeist extends the clang frontend, and as a result uses the same flags and syntax as clang. However, Polygeist also introduces several additional flags, such as -cuda-lower to specify the GPU-to-CPU translation and -cpuify=XX to specify a given method and set of parallel optimizations (see Section IV) used for generating the CPU program.

### IV. 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.

#### A. 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 a highly useful as any synchronization reduces parallelism. Much of the infrastructure for barrier elimination and simplification comes

directly from its memory behavior defined in Section III-A. Given a barrier B, let  $M_{before}^y$  be the union of memory effects before B until either another barrier or the start of the parallel region, and let the union of memory effects after B until the end of the parallel region  $M_{after}$ . If there are no memory effects to the same location across the barrier other than a read-after-read (RAR) (i.e.  $(M_{before}^y \setminus M_{after}) \cap \text{RAR} = \emptyset$ ), the barrier has its behavior subsumed by the prior barrier and can be eliminated. The symmetric condition  $M_{before}^y \setminus M_{after} \cap \text{RAR} = \emptyset$  indicates that the barrier is subsumed by a subsequent barrier. A specific trivial case of eliminatable barrier is one that has no memory effects at all.

For example, consider the code in Fig. 9, which comes from the backprop Rodinia benchmark [24]. 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,  $M_{before}$  (going all the way to the start) contains only a write to node and a read from input.  $M_{after}^y$  (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 for a barrier to be moved to and check if the previous memory analysis would deduce that the current barrier is unnecessary, thereby permitting barrier motion.

#### B. 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 III-A, barriers have the memory behavior of the code above and below them with the notable exception of the access from the current thread. This hole is important as it enables the memory-to-register promotion to operate on thread-local memory such as local variables. Moreover, this optimization is able to successfully replace slow memory reads with fast registers. For example, consider again the code in Figure 9. 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

```

__global__ void bpnnc_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 )
  output[by * hid + ty] = weights[tx][ty];
}

```

Fig. 9. Backprop Rodinia layer 8: CUDA to CPU conversion. •, !, &, &#220, &#221, &#222, &#223, &#224, &#225, &#226, &#227, &#228, &#229, &#230, &#231, &#232, &#233, &#234, &#235, &#236, &#237, &#238, &#239, &#240, &#241, &#242, &#243, &#244, &#245, &#246, &#247, &#248, &#249, &#250, &#251, &#252, &#253, &#254, &#255, &#256, &#257, &#258, &#259, &#260, &#261, &#262, &#263, &#264, &#265, &#266, &#267, &#268, &#269, &#270, &#271, &#272, &#273, &#274, &#275, &#276, &#277, &#278, &#279, &#280, &#281, &#282, &#283, &#284, &#285, &#286, &#287, &#288, &#289, &#290, &#291, &#292, &#293, &#294, &#295, &#296, &#297, &#298, &#299, &#300, &#301, &#302, &#303, &#304, &#305, &#306, &#307, &#308, &#309, &#310, &#311, &#312, &#313, &#314, &#315, &#316, &#317, &#318, &#319, &#320, &#321, &#322, &#323, &#324, &#325, &#326, &#327, &#328, &#329, &#330, &#331, &#332, &#333, &#334, &#335, &#336, &#337, &#338, &#339, &#340, &#341, &#342, &#343, &#344, &#345, &#346, &#347, &#348, &#349, &#350, &#351, &#352, &#353, &#354, &#355, &#356, &#357, &#358, &#359, &#360, &#361, &#362, &#363, &#364, &#365, &#366, &#367, &#368, &#369, &#370, &#371, &#372, &#373, &#374, &#375, &#376, &#377, &#378, &#379, &#380, &#381, &#382, &#383, &#384, &#385, &#386, &#387, &#388, &#389, &#390, &#391, &#392, &#393, &#394, &#395, &#396, &#397, &#398, &#399, &#400, &#401, &#402, &#403, &#404, &#405, &#406, &#407, &#408, &#409, &#410, &#411, &#412, &#413, &#414, &#415, &#416, &#417, &#418, &#419, &#420, &#421, &#422, &#423, &#424, &#425, &#426, &#427, &#428, &#429, &#430, &#431, &#432, &#433, &#434, &#435, &#436, &#437, &#438, &#439, &#440, &#441, &#442, &#443, &#444, &#445, &#446, &#447, &#448, &#449, &#450, &#451, &#452, &#453, &#454, &#455, &#456, &#457, &#458, &#459, &#460, &#461, &#462, &#463, &#464, &#465, &#466, &#467, &#468, &#469, &#470, &#471, &#472, &#473, &#474, &#475, &#476, &#477, &#478, &#479, &#480, &#481, &#482, &#483, &#484, &#485, &#486, &#487, &#488, &#489, &#490, &#491, &#492, &#493, &#494, &#495, &#496, &#497, &#498, &#499, &#500, &#501, &#502, &#503, &#504, &#505, &#506, &#507, &#508, &#509, &#510, &#511, &#512, &#513, &#514, &#515, &#516, &#517, &#518, &#519, &#520, &#521, &#522, &#523, &#524, &#525, &#526, &#527, &#528, &#529, &#530, &#531, &#532, &#533, &#534, &#535, &#536, &#537, &#538, &#539, &#540, &#541, &#542, &#543, &#544, &#545, &#546, &#547, &#548, &#549, &#550, &#551, &#552, &#553, &#554, &#555, &#556, &#557, &#558, &#559, &#560, &#561, &#562, &#563, &#564, &#565, &#566, &#567, &#568, &#569, &#570, &#571, &#572, &#573, &#574, &#575, &#576, &#577, &#578, &#579, &#580, &#581, &#582, &#583, &#584, &#585, &#586, &#587, &#588, &#589, &#590, &#591, &#592, &#593, &#594, &#595, &#596, &#597, &#598, &#599, &#600, &#601, &#602, &#603, &#604, &#605, &#606, &#607, &#608, &#609, &#610, &#611, &#612, &#613, &#614, &#615, &#616, &#617, &#618, &#619, &#620, &#621, &#622, &#623, &#624, &#625, &#626, &#627, &#628, &#629, &#630, &#631, &#632, &#633, &#634, &#635, &#636, &#637, &#638, &#639, &#640, &#641, &#642, &#643, &#644, &#645, &#646, &#647, &#648, &#649, &#650, &#651, &#652, &#653, &#654, &#655, &#656, &#657, &#658, &#659, &#660, &#661, &#662, &#663, &#664, &#665, &#666, &#667, &#668, &#669, &#670, &#671, &#672, &#673, &#674, &#675, &#676, &#677, &#678, &#679, &#680, &#681, &#682, &#683, &#684, &#685, &#686, &#687, &#688, &#689, &#690, &#691, &#692, &#693, &#694, &#695, &#696, &#697, &#698, &#699, &#700, &#701, &#702, &#703, &#704, &#705, &#706, &#707, &#708, &#709, &#710, &#711, &#712, &#713, &#714, &#715, &#716, &#717, &#718, &#719, &#720, &#721, &#722, &#723, &#724, &#725, &#726, &#727, &#728, &#729, &#730, &#731, &#732, &#733, &#734, &#735, &#736, &#737, &#738, &#739, &#740, &#741, &#742, &#743, &#744, &#745, &#746, &#747, &#748, &#749, &#750, &#751, &#752, &#753, &#754, &#755, &#756, &#757, &#758, &#759, &#760, &#761, &#762, &#763, &#764, &#765, &#766, &#767, &#768, &#769, &#770, &#771, &#772, &#773, &#774, &#775, &#776, &#777, &#778, &#779, &#780, &#781, &#782, &#783, &#784, &#785, &#786, &#787, &#788, &#789, &#790, &#791, &#792, &#793, &#794, &#795, &#796, &#797, &#798, &#799, &#800, &#801, &#802, &#803, &#804, &#805, &#806, &#807, &#808, &#809, &#810, &#811, &#812, &#813, &#814, &#815, &#816, &#817, &#818, &#819, &#820, &#821, &#822, &#823, &#824, &#825, &#826, &#827, &#828, &#829, &#830, &#831, &#832, &#833, &#834, &#835, &#836, &#837, &#838, &#839, &#840, &#841, &#842, &#843, &#844, &#845, &#846, &#847, &#848, &#849, &#850, &#851, &#852, &#853, &#854, &#855, &#856, &#857, &#858, &#859, &#860, &#861, &#862, &#863, &#864, &#865, &#866, &#867, &#868, &#869, &#870, &#871, &#872, &#873, &#874, &#875, &#876, &#877, &#878, &#879, &#880, &#881, &#882, &#883, &#884, &#885, &#886, &#887, &#888, &#889, &#890, &#891, &#892, &#893, &#894, &#895, &#896, &#897, &#898, &#899, &#900, &#901, &#902, &#903, &#904, &#905, &#906, &#907, &#908, &#909, &#910, &#911, &#912, &#913, &#914, &#915, &#916, &#917, &#918, &#919, &#920, &#921, &#922, &#923, &#924, &#925, &#926, &#927, &#928, &#929, &#930, &#931, &#932, &#933, &#934, &#935, &#936, &#937, &#938, &#939, &#940, &#941, &#942, &#943, &#944, &#945, &#946, &#947, &#948, &#949, &#950, &#951, &#952, &#953, &#954, &#955, &#956, &#957, &#958, &#959, &#960, &#961, &#962, &#963, &#964, &#965, &#966, &#967, &#968, &#969, &#970, &#971, &#972, &#973, &#974, &#975, &#976, &#977, &#978, &#979, &#980, &#981, &#982, &#983, &#984, &#985, &#986, &#987, &#988, &#989, &#990, &#991, &#992, &#993, &#994, &#995, &#996, &#997, &#998, &#999, &#1000, &#1001, &#1002, &#1003, &#1004, &#1005, &#1006, &#1007, &#1008, &#1009, &#1010, &#1011, &#1012, &#1013, &#1014, &#1015, &#1016, &#1017, &#1018, &#1019, &#1020, &#1021, &#1022, &#1023, &#1024, &#1025, &#1026, &#1027, &#1028, &#1029, &#1030, &#1031, &#1032, &#1033, &#1034, &#1035, &#1036, &#1037, &#1038, &#1039, &#1040, &#1041, &#1042, &#1043, &#1044, &#1045, &#1046, &#1047, &#1048, &#1049, &#1050, &#1051, &#1052, &#1053, &#1054, &#1055, &#1056, &#1057, &#1058, &#1059, &#1060, &#1061, &#1062, &#1063, &#1064, &#1065, &#1066, &#1067, &#1068, &#1069, &#1070, &#1071, &#1072, &#1073, &#1074, &#1075, &#1076, &#1077, &#1078, &#1079, &#1080, &#1081, &#1082, &#1083, &#1084, &#1085, &#1086, &#1087, &#1088, &#1089, &#1090, &#1091, &#1092, &#1093, &#1094, &#1095, &#1096, &#1097, &#1098, &#1099, &#1100, &#1101, &#1102, &#1103, &#1104, &#1105, &#1106, &#1107, &#1108, &#1109, &#1110, &#1111, &#1112, &#1113, &#1114, &#1115, &#1116, &#1117, &#1118, &#1119, &#1120, &#1121, &#1122, &#1123, &#1124, &#1125, &#1126, &#1127, &#1128, &#1129, &#1130, &#1131, &#1132, &#1133, &#1134, &#1135, &#1136, &#1137, &#1138, &#1139, &#1140, &#1141, &#1142, &#1143, &#1144, &#1145, &#1146, &#1147, &#1148, &#1149, &#1150, &#1151, &#1152, &#1153, &#1154, &#1155, &#1156, &#1157, &#1158, &#1159, &#1160, &#1161, &#1162, &#1163, &#1164, &#1165, &#1166, &#1167, &#1168, &#1169, &#1170, &#1171, &#1172, &#1173, &#1174, &#1175, &#1176, &#1177, &#1178, &#1179, &#1180, &#1181, &#1182, &#1183, &#1184, &#1185, &#1186, &#1187, &#1188, &#1189, &#1190, &#1191, &#1192, &#1193, &#1194, &#1195, &#1196, &#1197, &#1198, &#1199, &#1200, &#1201, &#1202, &#1203, &#1204, &#1205, &#1206, &#1207, &#1208, &#1209, &#1210, &#1211, &#1212, &#1213, &#1214, &#1215, &#1216, &#1217, &#1218, &#1219, &#1220, &#1221, &#1222, &#1223, &#1224, &#1225, &#1226, &#1227, &#1228, &#1229, &#1230, &#1231, &#1232, &#1233, &#1234, &#1235, &#1236, &#1237, &#1238, &#1239, &#1240, &#1241, &#1242, &#1243, &#1244, &#1245, &#1246, &#1247, &#1248, &#1249, &#1250, &#1251, &#1252, &#1253, &#1254, &#1255, &#1256, &#1257, &#1258, &#1259, &#1260, &#1261, &#1262, &#1263, &#1264, &#1265, &#1266, &#1267, &#1268, &#1269, &#1270, &#1271, &#1272, &#1273, &#1274, &#1275, &#1276, &#1277, &#1278, &#1279, &#1280, &#1281, &#1282, &#1283, &#1284, &#1285, &#1286, &#1287, &#1288, &#1289, &#1290, &#1291, &#1292, &#1293, &#1294, &#1295, &#1296, &#1297, &#1298, &#1299, &#1300, &#1301, &#1302, &#1303, &#1304, &#1305, &#1306, &#1307, &#1308, &#1309, &#1310, &#1311, &#1312, &#1313, &#1314, &#1315, &#1316, &#1317, &#1318, &#1319, &#1320, &#1321, &#1322, &#1323, &#1324, &#1325, &#1326, &#1327, &#1328, &#1329, &#1330, &#1331, &#1332, &#1333, &#1334, &#1335, &#1336, &#1337, &#1338, &#1339, &#1340, &#1341, &#1342, &#1343, &#1344, &#1345, &#1346, &#1347, &#1348, &#1349, &#1350, &#1351, &#1352, &#1353, &#1354, &#1355, &#1356, &#1357, &#1358, &#1359, &#1360, &#1361, &#1362, &#1363, &#1364, &#1365, &#1366, &#1367, &#1368, &#1369, &#1370, &#1371, &#1372, &#1373, &#1374, &#1375, &#1376, &#1377, &#1378, &#1379, &#1380, &#1381, &#1382, &#1383, &#1384, &#1385, &#1386, &#1387, &#1388, &#1389, &#1390, &#1391, &#1392, &#1393, &#1394, &#1395, &#1396, &#1397, &#1398, &#1399, &#1400, &#1401, &#1402, &#1403, &#1404, &#1405, &#1406, &#1407, &#1408, &#1409, &#1410, &#1411, &#1412, &#1413, &#1414, &#1415, &#1416, &#1417, &#1418, &#1419, &#1420, &#1421, &#1422, &#1423, &#1424, &#1425, &#1426, &#1427, &#1428, &#1429, &#1430, &#1431, &#1432, &#1433, &#1434, &#1435, &#1436, &#1437, &#1438, &#1439, &#1440, &#1441, &#1442, &#1443, &#1444, &#1445, &#1446, &#1447, &#1448, &#1449, &#1450, &#1451, &#1452, &#1453, &#1454, &#1455, &

```
__global__ void bpnnp_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
wei ghts[ty][tx] = wei ghts[ty][tx] * node[ty];
__syncthreads();
```

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

```
hi dden[i ndex] = wei ghts[ty][tx];  
// Unnecessary Barrier #2  
__syncthreads();
```

```

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

```

9. An example CUDA kernel from the Rodinia backprop test that contains unnecessary synchronization and unnecessary use of shared memory where a register would have sufficed.

loaded at that point is the same one which was stored earlier by the same read. Moreover, because that same location is overwritten before anyone else could read from `weights`, the first store can be safely eliminated once the load simply uses the register containing the value loaded from `hidden`. During the memory-to-register optimization, Polygeist can now successfully derive this forwarding property, since the hole in the memory properties described in Section III-A 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 operate on the barrier code.

### C. 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

<pre> omp. parallel {     omp. wsl oop %i = 1 to 10 {         codeA(%i)     } }  omp. parallel {     omp. wsl oop %i = 1 to 10 {         codeA(%i)     } } </pre>	<pre> omp. parallel {     omp. wsl oop %i = 1 to 10 {         codeA(%i)     }      omp. barrier      omp. wsl oop %i = 1 to 10 {         codeA(%i)     } } </pre>
---	---

§2 10. Example of OpenMP parallel region fusion, as applied on MLIR. Given two adjacent OpenMP parallel regions, each of which initializes threads to be run with the given closure, fuse the two closures along with a thread barrier, thereby allowing the threads to be initialized once instead of twice.

must check that no other code within the "for" loop conflicts with the memory access performed by `l`.

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 in traditional compilers (see Figure 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 loop-invariant code motion 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 to imagine running the program in lock-step. That is to say, if a parallel for loop had 10 instructions, each thread would execute instruction 1 before any thread would execute 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  $I$ . In other words, one does not need to check if  $I$  conflicts with any subsequent instruction in the parallel for loop to enable hoisting.

#### D. Block Parallelism Optimizations

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

<pre> omp.parallel {   omp.wsl oop %i = 1 to 10 {     codeA(%i)   } }  omp.parallel {   omp.wsl oop %i = 1 to 10 {     codeA(%i)   } } </pre>	<pre> omp.parallel {   omp.wsl oop %i = 1 to 10 {     codeA(%i)   }   omp.barrier   omp.wsl oop %i = 1 to 10 {     codeA(%i)   } } </pre>
---	---

[illegible]

```
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
}
```

$\bar{i} \in \mathbb{I}_{1/2}$  OpenMPv L:  $\bar{i} \in \mathbb{I}_n$ ,  $\langle \cdot, \cdot \rangle$  (C/C++- " (  $\bar{i} \in \mathbb{I}_{2 \times 10}$  )  
 - OpenMPv L:  $\bar{i} \in \mathbb{I}_n$  i U K (  $\bar{Z} f^n$  ) (  $\bar{Z} t^*$  for  $\bar{i} \in \mathbb{I}_n$  )  
 $v: B \bar{i} \in \mathbb{I}_{2 \times 1/2} \bar{i} \in \mathbb{I}_{2 \times 1/2} \bar{i} \in \mathbb{I}_2 U \bar{i} \in \mathbb{I}_{2 \times 1/2}$  closure /  $\bar{i} \in \mathbb{I}_n$   
 $\bar{i} \in \mathbb{I}_{2 \times 1/2} \bar{i} \in \mathbb{I}_{2 \times 1/2} \bar{i} \in \mathbb{I}_{2 \times 1/2}$

#### D. Block Parallelism Optimizations

[illegible][illegible]

## V. MocCUDA: INTEGRATION INTO PYTORCH

### A. Targeting CPU-only Supercomputers

[illegible]

### B. Architecture of MocCUDA and Integration of Polygeist

$\frac{1}{2}$  i ž  $\frac{1}{2}$  t \* | i B  $\frac{1}{2}$  MocCUDA i ž  $\frac{1}{2}$   
 CPU OgL PyTorch „ GPU  
 i ž  $\frac{1}{2}$  ž  $\frac{1}{2}$  MK i ž  $\frac{1}{2}$  E ° „ „ ' ž  $\frac{1}{2}$  ' } 6  
 PyTorch nž v i ž  $\frac{1}{2}$  CPU i ž  $\frac{1}{2}$  v  
 i ž  $\frac{1}{2}$  ( CPU i ž  $\frac{1}{2}$  ... i ž  $\frac{1}{2}$  v L i — „ | i ž  $\frac{1}{2}$



```

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
}

```

½ 11. Example of OpenMP parallel region hoisting, as applied on C/C++. This is an extension of the OpenMP parallel region merging in Figure 10, except as applied to an entire for loop. Rather than creating a separate closure / thread initialization for each iteration *i* of the outer loop, create the closure / thread initialization once.

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 III-B, the overhead of thread management can be reduced by fusing adjacent OpenMP “parallel” constructs [31] *without* fusing the worksharing loops (see Fig. 10), thus not undoing the barrier lowering. Parallel region fusion can be extended a variety of constructs such as moving the OpenMP parallel region outside the surrounding “for” in Fig. 11. This calls thread initialization once rather than  $N$  times. Applying this generally to control flow constructs enables all of the parallel 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 [32], [33]. As such, we also support an optimization for serializing any nested OpenMP parallel regions. Performing such serialization may leverage memory locality to improve performance.

## V. MocCUDA: INTEGRATION INTO PYTORCH

### A. Targeting CPU-only Supercomputers

An aim of our work to model the GPU execution model in MLIR is the ability to execute high-performance codes originally written for GPUs on a supercomputer that has only CPUs, in particular, on the Fugaku machine with A64FX processors [1]. As a prime example, consider PyTorch [2] that has not been successfully ported to the A64FX architecture. PyTorch supports CPU execution using its “native” backend, which offers only a naive and thus poorly performing implementation for many kernels, e.g., a 6-way nested loop with no memory optimization for a 2D convolution. On Intel CPUs, the execution of computationally-intensive kernels is delegated to the oneDNN library [3] which performs poorly on Arm CPUs as its tailored for common CPUs without high-bandwidth memory available on A64FX. Fujitsu’s fork of oneDNN [4] (which we also compare against in Section VI-C) required manual tuning to improve performance, and is still not universally competitive with GPUs. Interestingly, CPUs with high-bandwidth memory can benefit from computation organization similar to that of GPUs. We therefore design “MocCUDA” to mock the GPU backed for Pytorch by transpiling CUDA kernels into OpenMP-enabled parallel CPU kernels using the mechanism described in the previous sections.

### B. Architecture of MocCUDA and Integration of Polygeist

We implemented a compatibility layer, MocoCUDA, to enable transparent execution of PyTorch GPU backend exclusively on CPU with the goal of avoiding any manual re-engineering and interoperability with existing libraries. While PyTorch does have its own CPU backend, its design is poorly compatible with massively parallel computers running CPUs. In particular, performance issues are caused by synchronous kernel execution, mismatched memory layouts, and a presumption of low-speed memory.

Rather than attempting a challenging and time-consuming redesign the PyTorch CPU backend, we decided to base MocCUDA on the design of PyTorch’s GPU backend given the relative similarity of the platform to A64FX. Therefore, MocCUDA must handle and replace four types of operations:

- 1) calls to the CUDA runtime (CUDART),
- 2) calls to deep-learning specific functions in cuDNN,
- 3) calls to auxiliary CUDA libraries, and

[illegible]

## VI. EVALUATION

[illegible]

4) executions of CUDA kernels from within PyTorch.

Only the latter can be compiled since the rest of the functions are distributed as binary libraries.

To minimize the scope of work for the prototype, we profiled all interactions of PyTorch with the CUDA backend to identify (1)–(4) for a well-known deep learning problem, namely the ResNet-50 [25] neural network.

From profiling, PyTorch’s interaction with the CUDART is mostly limited to identifying properties of installed GPUs, memory management, and management and synchronization of CUDA streams. For the prototype, we limit ourselves to emulating one GPU per NUMA node, and only managing explicit memory allocation and data transfer. To give PyTorch access to GPU device properties, we dump the data of a real GPU, Nvidia GeForce RTX 2080 Ti, into a file which is used later by MocCUDA on the system without GPU. We emulate CUDA streams through Apple’s Grand Central Dispatch (GCD) [34] which enables asynchronicity of the emulated GPU operation from Pytorch’s management layers.

The ResNet-50 model only exercises a subset of closed-source cuDNN functions (forward and backward passes for the convolutional layers, batch normalization layers, and tensor additions). We re-implemented all necessary cuDNN function variations with focus on OpenMP-parallelization and HBM-friendly Im2Col plus GEMM convolutions as well as support for the NCHW layout (batch N, channels C, height H, width W). Furthermore, we identified calls to the cuBLAS library, and implemented wrapper functions to intercept these calls and dispatch them instead to the BLAS library designed for CPUs, such as MKL, OpenBLAS, or Fujitsu’s SSL2 [35] for A64FX. A similar approach is possible for other libraries linked into PyTorch, such as cuFFT, but is not necessary for ResNet-50.

Besides CUDART and other library calls, we observed *custom kernels*, i.e. kernels implemented in CUDA within PyTorch rather than provided by (binary) libraries, and identified the high-level functions which invoke them. Functions required by ResNet-50 include strided tensor kernels (add, multiply, etc), aggregation operations like “Softmax”, among others. The negative log-likelihood loss kernel uses CUDA’s `__syncthreads()` barriers. Porting these CUDA-based PyTorch functions to CPU involves labor-intensive re-engineering whereas Polygeist performs automatic translation. For demonstra-

Fig. 12. PolygeistInnerPar has a similar performance to MCUDA, while PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similarly 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).

tion purposes, we use Polygeist to automatically translate the `Cl assNLLCri teri on_updateOutput`, which uses `__syncthreads()`, and `Cl assNLLCri teri on_updateGradI nput` functions, and the functions transitively called from these, from CUDA to OpenMP, and integrate the result into MocCUDA.

We combine all the above described wrappers and re-implementations into MocCUDA, which can be used with LD\_PRELOAD to intercept CUDART/cuDNN calls to train ResNet-50 with the CUDA backend on CPUs.

## VI. 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 [24] 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 [11] 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<sup>1</sup> was compiled against LLVM 15 at commit

<sup>1</sup>Relevant versions of MocCUDA and Polygeist are available at <https://anonymous-data.s3.amazonaws.com/MocCUDA-master.zip> and <https://anonymous-data.s3.amazonaws.com/Polygeist-main.zip>.

### B. Use case 1: Rodinia Benchmarks

$\tau \in \mathbb{Z}/2$  PolygeistInnerPar,  $\mu \in \mathbb{Z}/2$  PolygeistInnerSer ...  
 $\tau \in \mathbb{Z}/2$  PolygeistInnerSer {  $\leq \mathbb{Z}/2$  MCUDA  $\cdot$  ( ... )  $\mathbb{Z}/2$  L,  
 PolygeistInnerPar  $\in \mathbb{Z}/2$  WE  $\in \mathbb{Z}/2$  v L'  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  
 $\mu \in \mathbb{Z}/2$   $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  $\in \mathbb{Z}/2$  G  
 $\tau \in \mathbb{Z}/2$  p  $\in \mathbb{Z}/2$  G

```
^ i 1.2.15 i 1.2.15 PyTorch
Resnet-50 i 1.2.15 Nvdi a, CUDA
11.6 SDK: Arm LLVM 13 i
i 1.2.15 Fujitsu, SSL2 v1.2.34 i
LPyTorch v1.4.0 i 1.2.15
i 1.2.15 PyTorchK i 1.2.15 Fujitsu, %o
, PyTorch v1.5.0
```

i 1/2 Ubuntu 20.04, AWS  
 c6i, metal ž i 1/2 Intel Xeon  
 Platinum 8375C CPU i 1/2 2.9  
 GHz i 1/2 2\* 8i 1/2 56GB...X i 1/2  
 ORodini a i 1/2 X i 1/2 1/2 ( i 1/2  
 i 1/2 i 1/2 i 1/2 i 1/2 i 1/2 i 1/2 i 1/2  
 5! i 1/2 - Mp

### A. Comparison to MCUDA

- H    i    ½ i    ½ ½ MCUDA [ 11 ] -  
 „ HMİ ½ ½ ½ MCUDA/ \*ASTS  
 + „ i ½ f ° „ CPU C/C++\ : “  
 i ½ • ( { < „ i ½ ½ / e e \ :  
 i ½ 10 „ i ½ f i ½ “ e i ½ i ½ i  
 • - f i ½ Rodini a • i ½ ½ i  
 ( i ½ 2- ½ + (    ½ p 1-- 24 E  
 ½ ‘ - - „ ½ X ½ 8 „ i ½  
 i ½ ½ Polygei st (    + ... i ½

```

Pol ygei st l nner Par      , @
, s G$      , i ½ MCUDA i ½ ½ WS
, Pol ygei st l nner Par ( 1 *      , i ½ ½
: / 1 OpenMP(      L W v L      , i ½

```

\* Pytorch: 0.12...T

$w_n \in E$ ,  $MCUDA_i \in \mathbb{R}^{1/2 \times 1/2}$   $B_i \in \mathbb{R}^{1/2 \times 1/2}$   
 $v_L \in SPolygeist_i \in \mathbb{R}^{1/2 \times 1/2}$   
 $Pol ygeist l n n e r S e r \quad i \in \mathbb{R}^{1/2} (t S$   
 $\hat{z}^o \dagger i \in \mathbb{R}^{1/2} MCUDA_i, \quad v - (1 * \hat{z}$   
 $, (32 * \hat{z}$

$i \in \mathbb{Z}_{\geq 1/2}$  subset  $114 * i \in \mathbb{Z}_{\geq 1/2}$   
 $i \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2}$  Polygeist / v w  
 $\wedge s i \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2}$   $i \in \mathbb{Z}_{\geq 1/2}$  (nvcc  $i \in \mathbb{Z}_{\geq 1/2}$ )  
v (GPU gL, "  $i \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2}$   $i \in \mathbb{Z}_{\geq 1/2}$ )  
(CPU gL, "  $i \in \mathbb{Z}_{\geq 1/2} \in \mathbb{Z}_{\geq 1/2}$   $i \in \mathbb{Z}_{\geq 1/2}$ , cn'

[illegible][illegible]

(GPU- i j L' - CUDA i  
OpenMP, B - i j f  
1 Z • ( q « ... X - g L • , i j n  
X, l u d E s r a d \_ v 2 , C U D A - O p e n M P  
, | f b p a r t i c l e f i l t e r , C U D A  
, - + i j OpenMP, i  
j OpenMP parallel  
for i - , V i ( C U D A i  
- (\_\_syncthreads) E  
P o l y g e i s t i Y i , i i  
ž ° backprop, i R i j L

3hybrid sort kmeans leukocyte mummergpu  
huffmanCheartwal l (Polygeist- • ( † « / „ C++ CUDAy  
' Z i p 1/2 i 1/2 X l avaMD C dwt2d i 1/2 1/2 † < c n  
„ C++ 1 Z i 1/2 1/2 1/2 ... X - i 1/2 1/2 1/2 1/2 L : GPUq  
• q « ... X i 1/2 1/2 F i i 1/2 B n n C g a u s s i a n K i  
( 0.005 i 1/2  
41 Z n n , i 1/2 1/2 1/2 1/2 i 1/2 1/2 d i 1/2 H , ( p n ) i 1/2  
@ ( i 1/2 1/2 1/2 } ( OpenMP i 1/2 i 1/2 C U D A i 1/2  
„ } @ p n i 1/2 i 1/2

89525cbf. For the PyTorch Resnet-50, we compile Pytorch v1.4.0 using Nvidia’s CUDA 11.6 SDK for Arm<sup>2</sup>, 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.

A. Comparison to MCUDA

First, we compare our approach to the previous work in MCUDA [11] . MCUDA is an AST-level tool which produces new CPU C/C++ as an output and uses a similar loop fission technique to handle synchronization. As a source-to-source tool, it handles only a fraction of the input language, making it unable to handle 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. 12. Polygeist with all optimization excluding serialization of the inner loop (PolygeistInnerPar) produces code within 1.3% of MCUDA on average. Specifically 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 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.

B. Use case 1: Rodinia Benchmarks

We benchmarked a subset of 14 benchmarks that are currently supported by Polygeist, and had a nontrivial runtime.<sup>3</sup> We verified correctness of our flow by comparing the program outputs produced by compiling with nvcc and

<sup>2</sup>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.

<sup>3</sup>The hybrid sort, 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 (GPU driver zero-initializes shared memory, but is not required to do so). The nn and gaussian tests ran in 0.005 seconds.

executed on a GPU, and compiled by our flow and executed on a CPU. 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.<sup>4</sup>

We compare the Rodinia CUDA benchmarks compiled for the CPU with the Rodinia OpenMP versions of the benchmark in Fig. 13(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 76% geomean performance improvement, when the inner serialization optimization is enabled. Without inner serialization, we still see a geomean speedup of 43.7%. Some benchmarks such as hotspot and pathfinder employ optimizations techniques for stencil computations which duplicate computation across threads in order to reduce synchronization overhead and make better use of the parallelism available in a GPU. This makes the CUDA code significantly more complex than the OpenMP version which causes them to perform worse. The CUDA-OpenMP versions of lud and sradv2 tests are slower as the program performs additional work to cache data within shared memory. The CUDA-OpenMP version of particlefilter receives a relative speedup as the pure OpenMP version of the code achieves the desired dependency structure through separate OpenMP “parallel for” loops, whereas in the CUDA code, this was done with a \_\_syncthreads. Polygeist was able to successfully optimize code surrounding the barrier, resulting in the speedup. The speedup for backprop is partially due to parallel optimizations (see Fig. 13(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 myocyte and sradv1, both achieve speedups due to code optimization across the parallel region boundary, as well as inner serialization.

We test the scaling properties of our approach by comparing transpiled CUDA with native OpenMP kernels in Fig. 14. Transpiled CUDA codes generally scale much better than the native OpenMP versions. As most CUDA programs are written

<sup>4</sup>For nn, already excluded due to its trivial runtime, the two versions differ in data loading (dynamically as it is run, in the OpenMP code, and preloading all data, in the CUDA code) and ought to be excluded for this reason as well.

Fig. 13. Left: Speedup of Rodinia benchmarks over OpenMP. Right: Relative performance of Rodinia benchmarks over OpenMP. The x-axis represents the number of threads used in the OpenMP version of the benchmark. The y-axis represents the relative performance of the CUDA version of the benchmark over the OpenMP version. The legend indicates the benchmarks: lud, sradv2, particlefilter, hotspot, pathfinder, myocyte, sradv1, affine, and backprop.

Figure 13 shows the speedup of Rodinia benchmarks over OpenMP. The left plot shows the speedup of the benchmarks over OpenMP, and the right plot shows the relative performance of the benchmarks over OpenMP. The x-axis represents the number of threads used in the OpenMP version of the benchmark. The y-axis represents the relative performance of the CUDA version of the benchmark over the OpenMP version. The legend indicates the benchmarks: lud, sradv2, particlefilter, hotspot, pathfinder, myocyte, sradv1, affine, and backprop.

Figure 14 shows the scaling properties of our approach by comparing transpiled CUDA with native OpenMP kernels in Fig. 14. Transpiled CUDA codes generally scale much better than the native OpenMP versions. As most CUDA programs are written in a way that is more amenable to parallelization, the transpiled CUDA codes generally scale much better than the native OpenMP versions. The figure shows the relative performance of the benchmarks over OpenMP for different numbers of threads. The x-axis represents the number of threads used in the OpenMP version of the benchmark. The y-axis represents the relative performance of the CUDA version of the benchmark over the OpenMP version. The legend indicates the benchmarks: lud, sradv2, particlefilter, hotspot, pathfinder, myocyte, sradv1, affine, and backprop.

C. Use case 2: Pytorch/Resnet50 Test

Figure 15 shows the performance of PyTorch Resnet-50 on AWS c6i.metal instance. The figure shows the relative performance of the benchmarks over OpenMP for different numbers of threads. The x-axis represents the number of threads used in the OpenMP version of the benchmark. The y-axis represents the relative performance of the CUDA version of the benchmark over the OpenMP version. The legend indicates the benchmarks: lud, sradv2, particlefilter, hotspot, pathfinder, myocyte, sradv1, affine, and backprop.



Fig. 13. Left: Relative speedup (higher is better) of kernels with various parallel and/or CPU optimizations applied. Right: Speedup of Rodinia CUDA code when compiled to OpenMP compared against native Rodinia OpenMP code (when available). Benchmarks containing barriers are marked with an asterisk.

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 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.

We perform an ablation analysis to study how individual optimizations impact performance. The “mincut” series in Fig. 13(left) shows performance measurements for our approach with the optimization outlined in Section III-B1 to reduce the amount of data preserved across barriers. This is only relevant for benchmarks containing barriers (marked by an asterisk in the Figure). On applicable benchmarks, mincut provides a 4.1% geomean speedup. The “openmpt” series in Fig. 13(left) demonstrates the impact of OpenMP region merging and similar optimizations and results in a 8.9% geomean speedup. The “affine” series in Fig. 13(left) shows the results of raising control flow to their affine variants and enabling simple loop optimizations (such as loop unrolling). While this produces a geomean speedup of 4.6% across the board, it results in a 2.6 speedup for the backprop layerforward test as it results in a loop containing synchronization being fully unrolled and reduced to if statements.

14. 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.

### C. 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.

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 (configured for the Resnet-50 neural network model) [36]. We build Horovod v0.19.5 with CUDA SDK, 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

[illegible]

## VII. RELATED WORK

A. GPU to CPU Synchronization

i ½ (Horovod „ i ½ K ½ i ½ , = i Y i ½ ½ F i ½ p „ i ½ ½ — i ½  
 : Resnet - 50 ^ i Q ½ ½ Mn [ 36] H † N  
 i ½ ½ CUDA SDK LLVM E i ½ MCUDA [ 11] 2008 i ½ GPU i ½  
 MPI “ „ Horovod vO . 19 . 5 i ½ gLASTI b i ½ ° „ C CPU i ½ ½ ½  
 i Ž PyTorch „ , ½ i ½ i ½ : ( \* i ½ vLfor < MCUDA  
 ½ A64FX 8 i ½ GMG M \* MPI ’ t • ( i ½ deep fi ssi on  
 ! ½ , ½ 4 \* GPU v Ki ½ ½ \* e „ • ( i ½ ½ e ½ vLi ½  
 , ½ 2 \* ’ i UO12 \* , ½ 48 \* ’ ( vi ½ i ½ ^ d f i ½ ½ \_ ” ( Ž  
 \* TofuDUC- : 232 ^ b ½ Q i ½ i ½ Ocel ot [ 12] 2010 \*  
 OpenMP ½ pi ½ 12 i ½ ½ 8 i ½ PTXG i ½ : LLVM vs i ½ ½ i p ½  
 i ½ ½ (Pytorch v1 . 4 . O „ Ei ½ POCL [ 38] 2015  
 vi ½ Ž Pytorch v1 . 5 . O ’ \* ( Ž OpenCL „ Clang / LLVM i ½  
 i ½ (Benchmarker [ 37] i ½ i ½ COX [ 13] 2021 i ½ ½  
 i ½ i ½ archvi si oni ½ i ½ i ½ i ½ i ½ i ½ VMI b v > W  
 i ½ • (PyTorchgLB vi ½ ½ warpS Y i ½ ½ v } 6 ½ ½  
 i ½ i ½ i ½ 1 - - 12 , i ½ d • ( „ i ½ < , F i ½ \* ½  
 p1 - - 64 „ i ½ ½ i ½ i ½ G i h ½ ž ½ ½ / ( • i ½ NSIR - , (  
 i ½ i ½ MocCUDA( y ½ E , lll - A , @ : i ½ • gL ½ ½ ½  
 i ½ p ½ ½ i ½ Ž Fujitsu b i ½ ( • S + gL ½ ½ 1 ( ½ K ½ ½  
 „ oneDNN, ½ ½ 4 . 5 „ ½ , Oæ ^ d „ : ( NS + gL ½ ½ i ½  
 i ½ < 2 . 7 , < 1 . 2 , , i ½ 5 @ : ½ ½ i ½ (MLIR - i ½ i ½ ž  
 • ( i ½ ™ „ ... 8 „ MocCUDA • ( v ½ ž i ½ d • Si ywE i ½ 1 i ½  
 1 Polygeist „ ... 8 „ MocCUDA i ½ i ½ ž ° ½ Ei ½ ½ , C++ „ i ½  
 S , V - B , @ i ½ i ½ ½ ei ½ PPI ½ i ½ i ½  
 i ½ ½ i ½ ½ i ½ V - B , - ½ ½ ½ / „ „ , ^ — /  
 „ PyTorch CPU i ½ oneDNN' i ½ i ½ i ½ ½ • ( i ½ i ½  
 y • i ½ ½ ½ 1 Ž Intel „ oneDNN continuati on - passing e  
 [ 3 ] \* QA64FX i ½ HBM i ½ • e i ½ @ ½ e ½ ¶ : < , ®  
 ( i ½ i ½ i ½ / i Ž EMM „ w i ½ mi crothreading [ 39]  
 i ½ X ( ½ i ½ „ Arm CPU H† fN 2010 KarrenbergEHack







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 thoughtful discussions about transformations within MLIR. Thanks to Douglas Kogut, Jiahao Li, and Bojan Serafimov for thoughtful discussions about parallel optimizations within the compiler.

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 Applied Mathematics activity within the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research Program, under contract number DE-AC02-06CH11357; 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. 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. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

[1] M. Sato, Y. Ishikawa, H. Tomita, Y. Kodama, T. Odajima, M. Tsuji, H. Yashiro, M. Aoki, N. Shida, I. Miyoshi, K. Hirai, A. Furuya, A. Asato, K. Morita, and T. Shimizu, “Co-design for A64FX manycore processor and “Fugaku”,” in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–15.

[2] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimselshin, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[3] Intel, “Oneapi-src/onednn: Oneapi deep neural network library (onednn).” [Online]. Available: <https://github.com/oneapi-src/oneDNN>

[4] Fujitsu. [Online]. Available: [https://github.com/fujitsu/dnnl\\_aarch64](https://github.com/fujitsu/dnnl_aarch64)

[5] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a performance-portable solution for multi-platform gpu programming,” *Parallel Computing*, vol. 38, no. 8, pp. 391–407, 2012. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0167819111001335>

[6] J. A. Herdman, W. P. Gaudin, O. Perks, D. A. Beckingsale, A. C. Mallinson, and S. A. Jarvis, “Achieving portability and performance through OpenACC,” in *2014 First Workshop on Accelerator Programming using Directives*, 2014, pp. 19–26.

[7] H. Carter Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014, domain-Specific Languages and High-Level Frameworks for High-Performance Computing. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0743731514001257>

[8] F. Franchetti, T. M. Low, D. T. Popovici, R. M. Veras, D. G. Spampinato, J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, “Spiral: Extreme performance portability,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[9] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “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*, ser. PLDI ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: <https://doi.org/10.1145/2491956.2462176>

[10] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, oct 2019. [Online]. Available: <https://doi.org/10.1145/3355606>

[11] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu, “MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs,” in *Languages and Compilers for Parallel Computing*, J. N. Amaral, Ed. Springer Berlin Heidelberg, 2008, vol. 5335, pp. 16–30, series Title: Lecture Notes in Computer Science. [Online]. Available: [http://link.springer.com/10.1007/978-3-540-89740-8\\_2](http://link.springer.com/10.1007/978-3-540-89740-8_2)

[12] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, “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, 2010, pp. 353–364.

[13] R. Han, J. Lee, J. Sim, and H. Kim, “COX: CUDA on X86 by exposing warp-level functions to CPUs,” *arXiv preprint arXiv:2112.10034*, 2021.

[14] W. S. Moses, “How should compilers represent fork-join parallelism?” Master’s thesis, Massachusetts Institute of Technology, 2017.

[15] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding recursive fork-join parallelism into llvm’s intermediate representation,”

J. R. Johnson, M. Püschel, J. C. Hoe, and J. M. F. Moura, “Spiral: Extreme performance portability,” *Proceedings of the IEEE*, vol. 106, no. 11, pp. 1935–1968, 2018.

[9] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “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*, ser. PLDI ’13. New York, NY, USA: Association for Computing Machinery, 2013, p. 519{530. [Online]. Available: <https://doi.org/10.1145/2491956.2462176>

[10] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically,” *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, oct 2019. [Online]. Available: <https://doi.org/10.1145/3355606>

[11] J. A. Stratton, S. S. Stone, and W.-m. W. Hwu, “MCUDA: An efficient implementation of CUDA kernels for multi-core CPUs,” in *Languages and Compilers for Parallel Computing*, J. N. Amaral, Ed. Springer Berlin Heidelberg, 2008, vol. 5335, pp. 16--30, series Title: Lecture Notes in Computer Science. [Online]. Available: [http://link.springer.com/10.1007/978-3-540-89740-8\\_2](http://link.springer.com/10.1007/978-3-540-89740-8_2)

[12] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, “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, 2010, pp. 353--364.

[13] R. Han, J. Lee, J. Sim, and H. Kim, “COX: CUDA on X86 by exposing warp-level functions to CPUs,” *arXiv preprint arXiv:2112.10034*, 2021.

[14] W. S. Moses, “How should compilers represent fork-join parallelism?” Master’s thesis, Massachusetts Institute of Technology, 2017.

[15] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding recursive fork-join parallelism into llvm’s intermediate representation,” *ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 4, pp. 1--33, 2019.

[16] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, “HPVM: Heterogeneous parallel virtual machine,” in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018, pp. 68--80.

[17] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick, “Openmpir: Implementing openmp tasks with tapir,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, 2017, pp. 1--12.

[18] J. Doerfert and H. Finkel, “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*, ser. Lecture Notes

- ACM Transactions on Parallel Computing (TOPC)*, vol. 6, no. 4, pp. 1–33, 2019.
- [16] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, “HPVM: Heterogeneous parallel virtual machine,” in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018, pp. 68–80.
- [17] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick, “Openmpir: Implementing openmp tasks with tapir,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, 2017, pp. 1–12.
- [18] J. Doerfert and H. Finkel, “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*, ser. Lecture Notes in Computer Science, M. W. Hall and H. Sundar, Eds., vol. 11882. Springer, 2018, pp. 112–119. [Online]. Available: [https://doi.org/10.1007/978-3-030-34627-0\\_9](https://doi.org/10.1007/978-3-030-34627-0_9)
- [19] J. Doerfert, J. M. M. Diaz, and H. Finkel, “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*, ser. Lecture Notes in Computer Science, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds., vol. 11718. Springer, 2019, pp. 153–167. [Online]. Available: [https://doi.org/10.1007/978-3-030-28596-8\\_11](https://doi.org/10.1007/978-3-030-28596-8_11)
- [20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.
- [21] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.
- [22] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, “Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation,” *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, sep 2021. [Online]. Available: <https://doi.org/10.1145/3469030>
- [23] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, “Polygeist: Raising C to polyhedral MLIR,” in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 45–59.
- [24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.
- [25] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [26] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. N. Garcia, “LLVM compiler implementation for explicit parallelization and SIMD vectorization,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017*. ACM, 2017, pp. 4: 1–4: 11. [Online]. Available: <https://doi.org/10.1145/3148173.3148191>
- [27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “An efficient method of computing static single assignment form,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, in Computer Science, M. W. Hall and H. Sundar, Eds., vol. 11882. Springer, 2018, pp. 112–119. [Online]. Available: <https://doi.org/10.1145/3178372.3179507>
- [19] J. Doerfert, J. M. M. Diaz, and H. Finkel, “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*, ser. Lecture Notes in Computer Science, X. Fan, B. R. de Supinski, O. Sinnen, and N. Giacaman, Eds., vol. 11718. Springer, 2019, pp. 153–167. [Online]. Available: <https://doi.org/10.1145/3178372.3179507>
- [20] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, “MLIR: Scaling compiler infrastructure for domain specific computation,” in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.
- [21] C. Lattner and V. Adve, “LLVM: a compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.
- [22] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, “Domain-specific multi-level ir rewriting for gpu: The open earth compiler for gpu-accelerated climate simulation,” *ACM Trans. Archit. Code Optim.*, vol. 18, no. 4, sep 2021. [Online]. Available: <https://doi.org/10.1145/3458744.3473356>
- in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York, NY, USA: Association for Computing Machinery, 1989, p. 25–35. [Online]. Available: <https://doi.org/10.1145/75277.75280>
- [28] P. Feautrier and C. Lengauer, “Polyhedron model,” *Encyclopedia of parallel computing*, pp. 1581–1592, 2011.
- [29] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, “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*, 2021, pp. 1–16.
- [30] M. Harris *et al.*, “Optimizing parallel reduction in cuda,” *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.
- [31] J. Doerfert and H. Finkel, “Compiler optimizations for openmp,” in *International Workshop on OpenMP*. Springer, 2018, pp. 113–127.
- [32] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, “Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling,” in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3–13. [Online]. Available: <https://doi.org/10.1145/3178372.3179507>
- [33] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin, “Joint scheduling and layout optimization to enable multi-level vectorization,” *IMPACT, Paris, France*, 2012.
- [34] K. Sakamoto and T. Furumoto, “Grand central dispatch,” in *Pro Multi-threading and Memory Management for iOS and OS X*. Springer, 2012, pp. 139–145.
- [35] “Fujitsu SSL II User’s Guide (Scientific subroutine library),” Fujitsu, Japan.
- [36] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in tensorflow,” *arXiv preprint arXiv:1802.05799*, 2018.
- [37] A. Drozd, “Benchmarker,” Online GitHub repository: <https://github.com/undertherain/benchmarker/>, commit e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d, 2021.
- [38] P. Jäskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, “pocl: A performance-portable OpenCL implementation,” *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.
- [39] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, “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*, 2010, pp. 111–119.
- [40] R. Karrenberg and S. Hack, “Improving performance of OpenCL on CPUs,” in *International Conference on Compiler Construction*. Springer, 2012, pp. 1–20.
- [41] S. Moll, J. Doerfert, and S. Hack, “Input space splitting for opencl,” in *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*, A. Zaks and M. V. Hermenegildo, Eds. ACM, 2016, pp. 251–260. [Online]. Available: <https://doi.org/10.1145/2892208.2892217>
- [42] A. Patel, S. Tian, J. Doerfert, and B. M. Chapman, “A virtual GPU as developer-friendly openmp offload target,” in *ICPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021*, F. Silla and O. Marques, Eds. ACM, 2021, pp. 24:1–24:7. [Online]. Available: <https://doi.org/10.1145/3458744.3473356>
- [23] W. S. Moses, L. Chelini, R. Zhao, and O. Zinenko, “Polygeist: Raising C to polyhedral MLIR,” in *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2021, pp. 45–59.
- [24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee, 2009, pp. 44–54.
- [25] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [26] X. Tian, H. Saito, E. Su, J. Lin, S. Guggilla, D. Caballero, M. Masten, A. Savonichev, M. Rice, E. Demikhovsky, A. Zaks, G. Rapaport, A. Gaba, V. Porpodas, and E. N. Garcia, “LLVM compiler implementation for explicit parallelization and SIMD vectorization,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC, LLVM-HPC@SC 2017, Denver, CO, USA, November 13, 2017*. ACM, 2017, pp. 4: 1–4: 11. [Online]. Available: <https://doi.org/10.1145/3148173.3148191>
- [27] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, “An efficient method of computing static single assignment form,” in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’89. New York,

- [43] M. Pharr and W. R. Mark, “ispc: A SPMD compiler for high-performance CPU programming,” in *2012 Innovative Parallel Computing (InPar)*. IEEE, 2012, pp. 1–13.
- [44] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, “Performance portable c++ programming with raja,” in *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, 2019, pp. 455–456.
- [45] H. C. Edwards, C. R. Trott, and D. Sunderland, “Kokkos: Enabling manycore performance portability through polymorphic memory access patterns,” *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.
- [46] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, “Mapcg: writing parallel program portable between cpu and gpu,” in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010, pp. 217–226.
- [47] A. Klockner, “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)*. New York, NY, USA: Association for Computing Machinery, 2014, p. 82–87. [Online]. Available: <https://doi.org/10.1145/2627373.2627387>
- [48] V. Churavy, D. Aluthge, L. C. Wilcox, S. Byrne, M. Waruszewski, A. Ramadhan, Meredith, S. Schaub, J. Schloss, J. Samaroo, J. Bolewski, C. Kawczynski, J. E. Kozdon, J. Liu, O. Schulz, Oscar, P. Haraldsson, T. Arakaki, and T. Besard, “Juliagpu/kernelabstractions.jl: v0.8.0,” Mar. 2022. [Online]. Available: <https://doi.org/10.5281/zenodo.6324344>
- [49] M. Frigo, C. E. Leiserson, and K. H. Randall, “The implementation of the cilk-5 multithreaded language,” in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, 1998, pp. 212–223.
- [50] G. Stelle, W. S. Moses, S. L. Olivier, and P. McCormick, “OpenMPIR: Implementing openmp tasks with tapir,” in *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*. New York, NY, USA: ACM, 2017, pp. 3:1–3:12. [Online]. Available: <http://doi.acm.org/10.1145/3148173.3148186>
- [51] A. Schmitz, J. Miller, L. Trümper, and M. S. Müller, “Ppir: Parallel pattern intermediate representation,” in *2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2021, pp. 30–40.
- [52] S. Stuijk, M. Geilen, and T. Basten, “Sdf<sup>2</sup> 3: Sdf for free,” in *Sixth International Conference on Application of Concurrency to System Design (ACSD’06)*. IEEE, 2006, pp. 276–278.
- [53] S. Moon and M. W. Hall, “Evaluation of predicated array data-flow analysis for automatic parallelization,” *ACM SIGPLAN Notices*, vol. 34, no. 8, pp. 84–95, 1999.
- [54] C. E. Oancea and L. Rauchwerger, “Logical inference techniques for loop parallelization,” in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 509–520.
- [55] LLVM Contributors, “OpenMP-aware optimizations,” Online: <https://openmp.llvm.org/optimizations/OpenMPOpt.html>.
- [56] A. Aiken and D. Gay, “Barrier inference,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’98. New York, NY, USA: Association for Computing Machinery, 1998, p. 342–354. [Online]. Available: <https://doi.org/10.1145/268946.268974>
- [57] T. Sorensen, L. F. Salvador, H. Raval, H. Evrard, J. Wickerson, M. Martonosi, and A. F. Donaldson, “Specifying and testing gpu workgroup progress models,” *Proceedings of the ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–30, 2021.
- [58] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakić, “Portable inter-workgroup barrier synchronisation for GPUs,” in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016, pp. 39–58.
- [59] Z. Sura, X. Fang, C.-L. Wong, S. P. Midkiff, J. Lee, and D. Padua, “Compiler techniques for high performance sequentially consistent java programs,” in *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2005, pp. 2–13.
- [28] P. Feautrier and C. Lengauer, “Polyhedron model,” *Encyclopedia of parallel computing*, pp. 1581–1592, 2011.
- [29] W. S. Moses, V. Churavy, L. Paehler, J. Hückelheim, S. H. K. Narayanan, M. Schanen, and J. Doerfert, “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*, 2021, pp. 1–16.
- [30] M. Harris et al., “Optimizing parallel reduction in cuda,” *Nvidia developer technology*, vol. 2, no. 4, p. 70, 2007.
- [31] J. Doerfert and H. Finkel, “Compiler optimizations for openmp,” in *International Workshop on OpenMP*. Springer, 2018, pp. 113–127.
- [32] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, “Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling,” in *Proceedings of the 27th International Conference on Compiler Construction*, ser. CC 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 3{13. [Online]. Available:
- [33] N. Vasilache, B. Meister, M. Baskaran, and R. Lethin, “Joint scheduling and layout optimization to enable multi-level vectorization,” *IMPACT*, Paris, France, 2012.
- [34] K. Sakamoto and T. Furumoto, NY, USA: Association for Computing Machinery, 1989, p. 25{35. [Online]. Available:
- [35] “Fujitsu SSL-11 User’s Guide (Scientific subroutine library),” Fujitsu, Japan.
- [36] A. Sergeev and M. Del Balso, “Horovod: fast and easy distributed deep learning in tensorflow,” *arXiv preprint arXiv:1802.05799*, 2018.
- [37] A. Drozd, “Benchmarker,” *Online GitHub repository*: <https://github.com/alexdrozd/benchmarker>, commit e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d, 2021.
- [38] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raikila, J. Takala, and H. Berg, “pocl: A performance-portable OpenCL implementation,” *International Journal of Parallel Programming*, vol. 43, no. 5, pp. 752–785, 2015.
- [39] J. A. Stratton, V. Grover, J. Marathe, B. Aarts, M. Murphy, Z. Hu, and W.-m. W. Hwu, “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*, 2010, pp. 111–119.
- [40] R. Karrenberg and S. Hack, “Improving performance of OpenCL on CPUs,” in *International Conference on Compiler Construction*. Springer, 2012, pp. 1–20.
- [41] S. Moll, J. Doerfert, and S. Hack, “Input space splitting for opencl,” in *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, Barcelona, Spain, March 12–18, 2016, A. Zaks



- and M. V. Hermenegildo, Eds. ACM, 2016, pp. 251--260. [Online]. Available:
- [42] A. Patel, S. Tian, J. Doerfert, and B. M. Chapman, "A virtual GPU as developer-friendly openmp offload target," in ICCPP Workshops 2021: 50th International Conference on Parallel Processing, Virtual Event / Lemont (near Chicago), IL, USA, August 9-12, 2021, F. Silla and O. Marques, Eds. ACM, 2021, pp. 24:1--24:7. [Online]. Available:
- [43] M. Pharr and W. R. Mark, "ispc: A SPMD compiler for high-performance CPU programming," in 2012 Innovative Parallel Computing (InPar). IEEE, 2012, pp. 1--13.
- [44] D. Beckingsale, R. Hornung, T. Scogland, and A. Vargas, "Performance portable c++ programming with raja," in Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, 2019, pp. 455--456.
- [45] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," Journal of parallel and distributed computing, vol. 74, no. 12, pp. 3202--3216, 2014.
- [46] C. Hong, D. Chen, W. Chen, W. Zheng, and H. Lin, "Mapcg: writing parallel program portable between cpu and gpu," in Proceedings of the 19th international conference on Parallel architectures and compilation techniques, 2010, pp. 217--226.
- [47] A. Klöckner, "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). New York, NY, USA: Association for Computing Machinery, 2014, p. 82{87. [Online]. Available:
- [48] V. Churavy, D. Aluthge, L. C. Wilcox, S. Byrne, M. Waruszewski, A. Ramadhan, Meredith, S. Schaub, J. Schloss, J. Samaroo, J. Bolewski, C. Kawczynski, J. E. Kozdon, J. Liu, O. Schulz, Oscar, P. Haraldsson, T. Arakaki, and T. Besard, "JuliaGPU/kernel abstractions.jl: v0.8.0," Mar. 2022. [Online]. Available:
- [49] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, 1998, pp. 212--223.
- [50] G. Stelling, W. S. Moses, S. L. Olivier, and P. McCormick, "OpenMPI R: Implementing openmp tasks with tapir," in Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC. New York, NY, USA: ACM, 2017, pp. 3:1--3:12. [Online]. Available:
- [51] A. Schmitz, J. Müller, L. Trümpel, and M. S. Müller, "Ppir: Parallel pattern intermediate representation," in 2021 IEEE/ACM International Workshop on Hierarchical Parallelism for Exascale Computing (HiPar). IEEE, 2021, pp. 30--40.
- [52] S. Stuijk, M. Geilen, and T. Basten, "Sdf^3: Sdf for free," in Sixth International Conference on Application of Concurrency to System Design (ACSD'06).

- IEEE, 2006, pp. 276- - 278.
- [ 53] S. Moon and M. W. Hall, ``Evaluation of predicated array data-flow analysis for automatic parallelization,`` ACM SIGPLAN Notices, vol. 34, no. 8, pp. 84- - 95, 1999.
- [ 54] C. E. Oancea and L. Rauchwerger, ``Logical inference techniques for loop parallelization,`` in Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation, 2012, pp. 509- - 520.
- [ 55] LLVM Contributors, ``OpenMP-aware optimizations,`` Online: .
- [ 56] A. Aiken and D. Gay, ``Barrier inference,`` in Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '98. New York, NY, USA: Association for Computing Machinery, 1998, p. 342{354. [Online]. Available:
- [ 57] T. Sorensen, L. F. Salvador, H. Raval, H. Evrard, J. Wickerson, M. Martonosi, and A. F. Donaldson, ``Specifying and testing gpu workgroup progress models,`` Proceedings of the ACM on Programming Languages, vol. 5, no. OOPSLA, pp. 1- - 30, 2021.
- [ 58] T. Sorensen, A. F. Donaldson, M. Batty, G. Gopalakrishnan, and Z. Rakamaric, ``Portable inter-workgroup barrier synchronisation for GPUs,`` in Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2016, pp. 39- - 58.
- [ 59] Z. Sura, X. Fang, C. - L.

Wong, S. P. Midkiff, J. Lee, and D. Padua, ``Compiler techniques for high performance sequentially consistent java programs,`` in Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, 2005, pp. 2- - 13.