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

William S. Moses*, Ivan R. Ivanov†, Jens Domke‡, Toshio Endo†, Johannes Doerfert¶, and Oleksandr Zinenko§

* Massachusetts Institute of Technology, USA wmoses@mit.edu
† Tokyo Institute of Technology, Japan ivanov.i.aa@m.titech.ac.jp, endo@is.titech.ac.jp
‡ RIKEN Center for Computational Science, Japan jens.domke@riken.jp
¶ Argonne National Laboratory, USA jdoerfert@anl.gov
§ Google, France zinenko@google.com

摘要—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 handwritten OpenMP code. Further, we show how CUDA kernels from PyTorch can efficiently run and scale on the CPU-only Supercomputer Fugaku without user intervention. Our PyTorch compatibility layer making use of transpiled CUDA PyTorch kernels outperforms the PyTorch CPU native backend by 2.7×.

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 naïve 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*, Ivan R. Ivanov†, Jens Domke‡, Toshio Endo†, Johannes Doerfert¶, and Oleksandr Zinenko§

* Massachusetts Institute of Technology, USA wmoses@mit.edu
† Tokyo Institute of Technology, Japan ivanov.i.aa@m.titech.ac.jp, endo@is.titech.ac.jp
‡ RIKEN Center for Computational Science, Japan jens.domke@riken.jp
¶ Argonne National Laboratory, USA jdoerfert@anl.gov
§ Google, France zinenko@google.com

摘要—*警告：该PDF由GPT-Academic开源项目调用大语言模型+Latex翻译插件一键生成，版权归原文作者所有。翻译内容可靠性无保障，请仔细鉴别并以原文为准。项目Github地址 https://github.com/binary-husky/gpt_academic/。当前大语言模型: gpt-4o-mini，当前语言模型温度设定: 0。为了防止大语言模型的意外谬误产生扩散影响，禁止移除或修改此警告。

在并行性仍然是性能的主要来源的同时，体系结构实现和编程模型随着每一代新硬件而变化，这往往导致昂贵的应用重构。大多数性能可移植性工具都需要手动和昂贵的应用移植到另一种编程模型。

我们提出了一种替代方法，它基于Polygeist/MLIR自动将用一种编程模型（CUDA）编写的程序转换为另一种（CPU线程）。我们的方法包括并行构造的表征（representation），允许常规编译器转换透明且无需修改地应用，并支持并行性特定的优化。我们通过为多核CPU转译并优化CUDA Rodinia基准测试套件来评估我们的框架，取得了相较于手写的OpenMP代码76%的几何平均加速。此外，我们展示了来自PyTorch的CUDA内核如何在无用户干预的情况下高效地在仅限CPU的超级计算机Fugaku上运行和扩展。 我们的PyTorch兼容层利用转译的 CUDA PyTorch内核的性能超过了PyTorch CPU原生后端2.7×。

I. Introduction

单核性能扩展的结束意味着并行化和领域特定性现在是效率提升的主要来源。超级计算机架构师在创新上展开竞争，以支持从物理仿真 (simulation) 到机器学习的计算和内存密集型应用。最新最快的超级计算机Fugaku完全基于A64FX CPU，这与商品CPU不同，提供高带宽内存访问的支持，并且在能效方面可与最近的GPU相媲美 [1]。

然而，如何高效且富有成效地利用这些计算机进行实际应用是一个挑战，因为最近的框架和高性能库是在考虑NVIDIA GPU的情况下开发的。例如，将PyTorch [2]移植到A64FX的尝试遇到了多重挑战。"原生"（native）默认的CPU PyTorch后端仅为关键内核提供了简单版本，如实现为六层嵌套循环的2D卷积。英特尔的oneDNN [3]在Arm CPU上表现不佳并不令人意外，因为它是为没有高带宽内存的商品CPU量身定制的。富士通定制的oneDNN [4]被更好地调优，但在与GPU的竞争中并不具备普遍的优势。

```
__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);
}
```

图 1. 一个示例 CUDA 程序 `normalize`，它对一个向量进行归一化，以及一个调用内核的 CPU 函数 `launch`。目前，`sum` 在每个线程中被调用，导致总的工作量为 $O(N^2)$。通过使用共享内存（见上面的注释），工作量可以部分减少到 $O(N^2/B)$，这使得每个块中 `sum` 的计算只需执行一次，或者通过在内核之前计算 `sum` 一次，可以完全减少到 $O(N)$。

许多非库方法被提出以实现性能可移植性。这些方法从语言扩展如OpenCL [5]或OpenACC [6]，到并行编程框架如Kokkos [7]，再到特定领域语言如SPIRAL [8]、Halide [9]或Tensor Comprehensions [10]。由于语言或原始程序与目标框架之间的编程模型（model）差异，所有这些方法仍然要求遗留（legacy）应用程序被移植,有时需要完全重

```
__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);
}
```

图 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 SIMT-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

写。

我们探索了一种基于完全自动化编译器的替代方法，该编译器接受一种编程模型（model）(CUDA）中的代码并生成目标另一个模型（model）(CPU线程）的二进制文件。尽管过去曾探索GPU到CPU的转换 [11]–[13]，但很少能生成高效的代码。事实上，针对CPU的优化，甚至是通用编译器变换，如公共子表达式消除或循环不变代码运动，因缺乏可以分析的并行结构的表征（representation）而受到阻碍 [14]。以图 1中的求和为例，可以通过每个块的单个线程完成，或者在所有计算发生在CPU上后，由一个总线程完成。主流编译器内的并行性表征（representation）在最近才开始探索 [15]–[19]，现有的变换是有限的，通常仅适用于简单的CPU代码。

我们提出了一种针对最常见GPU构造的编译器模型（model）：多级并行性、级别范围同步和级别局部存储器。这与提供单级并行性、统一内存和对等同步的CPU并行性不同。与在优化管道之前操作的源和AST级方法以及将同步建模为“黑盒”优化障碍的现有编译器方法相反，我们完全从内存语义建模同步。这既允许基于同步的代码与现有优化进行交互，也使得新颖的特定并行性优化成为可能。

我们的模型（model）在LLVM编译器基础设施的MLIR层 [20]实现，并利用MLIR的嵌套模块方法来处理GPU代码 [22]。我们扩展了Polygeist [23] C/C++前端以支持CUDA，并生成保留高级并行性和程序结构的MLIR。我们的原型编译器能够将PyTorch CUDA内核以及其他计算密集型基准编译为LLVM支持的任何CPU架构。除了考虑执行模型差异的变换之外，我们还通过OpenMP利用CPU上的并行性。最后，我们的MocCUDA PyTorch集成使用我们的方法在没有GPU的情况下编译和执行CUDA内核，同时替代不受支持的调用。

我们通过编译Rodinia CUDA基准 [24]以及PyTorch CUDA内核来评估我们端到端转换的正确性和效率。当以商品CPU为目标时，我们的OpenMP加速的CUDA代码与Rodinia套件中的参考OpenMP实现性能相当，并且可扩展性有所提高。当使用我们的框架在仅CPU的Fugaku超级计算机上运行PyTorch时，我们在处理每秒图像数量上达到了与基于OneDNN的PyTorch CPU后端相比，约为Resnet-50 [25]的`conv2d`内核的两倍，同时整体训练的性能可比。

总体而言，我们的论文做出了以下贡献：

- 一种通用的高级和与平台无关的表征（representation）,基于障碍同步的语义定义,通过内存语义确保正确性,从而确保现有优化的透明应用。
- 新的并行特定优化可以利用我们的高层并行语义来优化程序。
- 对Polygeist C/C++前端的扩展，旨在将GPU和CPU并行结构直接映射到我们的高级并行原语中。
- 对Rodinia [24]基准套件的一个子集以及在PyTorch [2]中运行Resnet-50所需的内部CUDA内核进行端到端的CUDA到CPU的转译，以便在仅支持CPU的Fugaku超级计算机上运行。

II. Background

主流编译器如 Clang 和 GCC 缺乏统一的高层次并行性（parallelism）表征。具体而言，在诸如 CUDA、OpenMP 或 SYCL 等框架中编译并行构造时，强制并行区域的主体存在于一个单独的（closure）函数中，该函数由相应的运行时调用。线程索引（thread index）或同步（synchronization）等概念通常会被单独表征，往往通过不透明的内置调用（intrinsic calls）。由于编译器历史上缺乏关于并行性和所涉及的运行时效果的信息，任何并行构造也无意中成为优化的障碍。尽管近年来已经有一些尝试 [14]–[19], [26] 来改善 CPU 并行构造的表征，但加速器编程带来了额外的挑战。独特的编程模型和复杂的内存层次结构导致主流编译器中 GPU 并行性（parallelism）的高层次表征仍然探讨不足。

A. GPU Compilation

考虑图 1 中的 CUDA 程序，该程序对一个向量进行归一化。当使用 LLVM/Clang 编译时，GPU 程序是一个独立的编译单元，如图 II 所示。这阻止了 GPU 内核与 CPU 调用代码之间的任何优化。在图 1 的情况下，传统编译器中的程序总工作量为 $O(N^2)$，因为每个线程都执行 $O(N)$ 的 `sum` 调用。然而，如果在内核调用之前仅能执行一次 `sum` 调用，例如，通过执行循环不变代码移动（LICM）变换，则工作量将减少到 $O(N)$。这种优化的一个效果较差的变体可以通过使用共享内存将工作量减少到 $O(\frac{N^2}{B})$。MLIR 为 GPU 程序提供了嵌套模块表征（representation），支持主机/设备代码移动 [22]，但尚未实现并行代码移动。在 GPU 到 CPU 的代码移动中，LICM 从并行循环中移出始终是合法的，因为以前的设备内存在主机上也可用。

B. MLIR Infrastructure

MLIR 是一个为重用和可扩展性设计的近期编译器基础设施 [20]。其内部表征（IR）可以看作是广泛采用的 LLVM IR 的概念性继承者 [21]，并且对传统 CPU 模型

```
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
}
```

图 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.

```
// Kernel launch is available within the calling
// function, enabling optimizations across the
// GPU/CPU boundary.
func @launch(%h_out : memref<?xf32>,
             %h_in  : memref<?xf32>, %n : i64) {
  // Parallel for across all blocks in a grid.
  parallel.for (%gx, %gy, %gz) = (0, 0, 0)
           to (grid.x, grid.y, grid.z) {

    // Shared memory = stack allocation in a block.
    %shared_val = memref.alloca : memref<f32>

    // Parallel for across all threads in a block.
    parallel.for (%tx, %ty, %tz) = (0, 0, 0)
             to (blk.x, blk.y, blk.z) {
      // Control-flow is directly preserved.
      if %tx == 0 {
        %sum = func.call @sum(%d_in, %n)
        memref.store %sum, %shared_val[] : memref<f32>
      }
      // Syncronization via explicit operation.
      polygeist.barrier(%tx, %ty, %tz)
      %tid = %gx + grid.x * %tx
      if %tid < %n {
        %res = ...
        store %res, %d_out[%tid] : memref<?xf32>
      }
    }
  }
}
```

图 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.

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 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
}
```

图 2. 在与LLVM/Clang编译时，图1中的`launch`（启动）和`normalize`（归一化）函数的简化降低。由于这两个函数生成不同的汇编代码，它们被放置在没有调用上下文的独立模块中。

```
// Kernel launch is available within the calling
// function, enabling optimizations across the
// GPU/CPU boundary.
func @launch(%h_out : memref<?xf32>,
             %h_in  : memref<?xf32>, %n : i64) {
  // Parallel for across all blocks in a grid.
  parallel.for (%gx, %gy, %gz) = (0, 0, 0)
           to (grid.x, grid.y, grid.z) {

    // Shared memory = stack allocation in a block.
    %shared_val = memref.alloca : memref<f32>

    // Parallel for across all threads in a block.
    parallel.for (%tx, %ty, %tz) = (0, 0, 0)
             to (blk.x, blk.y, blk.z) {
      // Control-flow is directly preserved.
      if %tx == 0 {
        %sum = func.call @sum(%d_in, %n)
        memref.store %sum, %shared_val[] : memref<f32>
      }
      // Syncronization via explicit operation.
      polygeist.barrier(%tx, %ty, %tz)
      %tid = %gx + grid.x * %tx
      if %tid < %n {
        %res = ...
        store %res, %d_out[%tid] : memref<?xf32>
      }
    }
  }
}
```

图 3. 在Polygeist/MLIR中，图1所示的CUDA `launch`/`normalize`代码的共享内存变体的表征（representation）。内核调用在主机代码中直接可用，该主机代码调用它。通过在块和线程之间使用并行for循环，显式地体现了并行性，而共享内存则放置在块内，以表明它可以被同一块中的任何线程访问，但不能被来自不同块的线程访问。

之外的编程模型提供了更好的支持。 MLIR 并不是提供一组预定义的指令和类型，而是操作在包含可互操作的用户定义操作、属性和类型集合的 方言 上。操作是对 IR 指令的泛化，可以是任意复杂的，特别是可以包含更多 IR 的区域，从而创建嵌套表征。操作定义并使用遵循单一静态赋值（SSA） [27] 的值。例如，MLIR 方言可以对整个物理或虚拟指令集建模，例如 NVVM（针对 Nvidia GPU 的虚拟 IR）、其他 IR（例如 LLVM IR）、更高层次的控制流构造（如仿射循环）、并行编程模型（如 OpenMP 和 OpenACC）、机器学习图等。 MLIR 支持并鼓励在同一编译单元中混合来自不同方言的操作。

MLIR 支持 GPU，得益于其同名方言，该方言定义了高层次的 SIMT 编程模型以及主机/设备通信机制，并且一组低层次的特定平台方言：NVVM（CUDA）、ROCDL（ROCm）和 SPIR-V。 MLIR 对 GPU 编程的特殊方法在于其 统一 代码表征。得益于 IR 的灵活性，一个模块可以包含其他模块，例如，“主机” 翻译单元可以将“设备”翻译单元嵌入为 IR，而不是文件引用或二进制数据。这种方法提供了其他编译器无法获得的主机/设备优化机会，特别是通过在主机和设备之间自由移动代码 [22]。

C. *Polygeist*

Polygeist 是基于 Clang 的 MLIR 的 C 和 C++ 前端 [23]。它能够将广泛的 C++ 程序翻译成多种 MLIR 方言的混合，这些方言保留了程序高层结构的元素。特别是，Polygeist 将结构化控制流（循环和条件语句）作为 MLIR SCF 方言保留。它还通过依赖于 MLIR 的多维内存引用（memref）类型，在可能的情况下保留多维数组构造，从而简化分析。最后，Polygeist 能够识别适合多面体优化（polyhedral）的程序部分 [28]，并使用仿射（Affine）方言表示它们。

III. Approach

我们扩展了Polygeist编译器 [23]，以直接从CUDA生成 并 行MLIR。 这 利 用 了 统 一 的CPU/GPU 表 征（representation）， 使 优 化 器 能 够 理 解 主 机/设 备 执 行，

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), ROCDL (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, Poylgeist 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();
}
```

图 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();
}
```

图 4. 在两个任意指令之间包含一个障碍的程序。

并在内核边界之间实现优化。现有MLIR的第一级并行构造（`scf.parallel`，`affine.parallel`）的使用使我们能够针对现有的CPU和GPU后端。最后，MLIR的可扩展操作集允许我们定义自定义指令，带有相关属性和自定义优化。

我们将GPU内核启动的表征（representation）定义如下（如图 3所示）：

- 在网格中的所有块上进行 3D 并行 for 循环。
- 对任何共享内存进行堆栈分配，范围限定为每个块唯一。
- 在一个块中的所有线程上进行3D并行for循环。
- 一个自定义的Polygeist（Polygeist）屏障操作，提供等同于CUDA/ROCm（CUDA/ROCm）同步的语义。

这个过程使我们能够以一种保留所需语义的形式表示任何GPU程序（GPU program）。它被编译器完全理解，因此适合进行编译器优化。此外，通过使用一般的并行性（parallelism）、分配（allocation）和同步（synchronization）构造来表示GPU程序，我们不仅能够优化原始程序，还能将其重新定向到不同的架构（architecture）。

A. Barrier Semantics

一个 CUDA 或 ROCm `__syncthreads` 函数保证了在函数调用之前，块中的所有线程都已完成所有指令的执行，并且在调用之后，没有任何线程会执行任何指令。传统上，编译器将此类函数表示为不透明的优化屏障，这些屏障可能访问所有内存，并禁止涉及它们的任何变换。

在我们的系统中，我们选择通过一个新的 `polygeist.barrier` 操作来表示这种线程级同步。与其他方法不同，`polygeist.barrier`（因此简单称为 `barrier`）旨在仅防止改变外部可见行为的转换。我们并不禁止在 `barrier` 之间的任何代码移动，而是通过定义 `barrier` 具有特定的内存属性来成功实现所需的语义,这些内存属性表示为一组内存位置（包括未知）和内存效应类型（读取、写入、分配、释放）,这在 MLIR 中是标准的。考虑图 4 中的简单程序。只有当 `codeA` 和 `codeB` 访问相同的内存时，才可以观察到同步的影响。此外，如果两者仅读取相同的内存位置，则同步也是不必要的。我们可以列举其余的三种情况：

1) `codeA` 写入，`codeB` 加载
2) `codeA` 加载，`codeB` 写入
3) `codeA` 写，`codeB` 写

具有codeA写行为的屏障可以确保案例 1 的正确性：在codeB中的加载操作不能被提升到屏障之上，因为它将似乎读取到一个不同的　　价值）。对称地，具有　　　　写行为的屏障将确保案例　　　的正确性。因此，　　　　和　　　　的写行为的联合足以防止加载操作被错误地移动到屏障之外。

然而，这并不阻止写操作被移动。例如，　　　　可以在案例　　　　中被复制到屏障之上，并且它似乎会具有相同的最终内存（状态），因为在屏障之前的多余写操作将永远不会被读取。因此，我们还定义屏障具有　　　　和　　　　的读取行为。

这可以扩展以包括在给定屏障之前或之后可能执行的并行循环中的所有操作的内存效应。在具有显式分支的控制流图上　可以通过分别探索前驱或后继中的操作来实现。然而　在具有显式循环和条件操作的　　　　结构控制流级别上进行操作　允许分析被简化。此外　如果在同一块中存在多个屏障　则无需超越它。

考虑到一个足够表达的副作用模型，屏障的内存语义可以进一步扩展。屏障对来自不同线程的同一位置的读取　写入操作进行排序，自然执行顺序在一个线程内是足够的。因此，屏障不需要捕捉操作的内存效应，其中地址是线程标识符的单射函数。在可能的情况下，将内存访问提升到　　　　　方言中的线性形式，可以实现精确分析。考虑图　中的代码。屏障周围的读取和写入表达式具有仿射访问集$\mathcal{A}_o = \{A(i) : i = tx\}$，其中$tx$是线程　标识符。屏障具有仿射访问集$\mathcal{A}_b = \{A(i) : i \neq tx\}$。由于被访问的地址集没有重叠，$\mathcal{A}_o \cap \mathcal{A}_b = \varnothing$，因此允许在屏障之间移动代码。从概念上讲，对　　　　　　　　　　的写总是在同一线程内的读取之前发生，因此屏障是多余的。　相反　如果对　的加载和存储偏移为　，则屏障就是必要的，因为在屏障之后加载的数据将由不同的线程存储。　更一般地　通过收集所有相关操作的内存位置和效应类型来定义给定屏障的内存属性时　只需包括由不同线程使用的内存位置。当涉及多个基地址时，必须检查别名保证。

`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
}
```

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

$\mathcal{A}_o = \{A(i) : i = tx\}$ where tx is the thread `x` identifier. The barrier has the affine access set $\mathcal{A}_b = \{A(i) : i \neq tx\}$. Since the sets of accessed addresses do not overlap, $\mathcal{A}_o \cap \mathcal{A}_b = \varnothing$, 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
}
```

图 5. 障碍语义可以被细化为访问所有线程中在其之上/之下的操作所访问的内存地址，除了当前线程。

B. Barrier Lowering

为了使 程序能够在 上运行，我们必须高效地模拟 程序的同步行为。尽管第 节中的内存语义使我们能够在优化期间保持屏障的正确性，但本节讨论如何在 上实现屏障。

架构没有线程块的概念，也没有等待这种概念化线程组的屏障指令。相反，我们使用常规的 线程和工作共享在它们之间分配线程块的循环迭代。从概念上讲，这与 执行模型不同，后者是多个显式线程每个执行一次迭代。工作共享要求每个线程顺序执行多个迭代，使得在迭代中间进行同步变得不可能，只能在循环结束时同步。

为了应对这个问题，我们为我们的 （多级中间表示） 表征 （ ） 开发了一种新的屏障消除技术。如第 节所述，过去已经探索了几种方法，包括循环分裂 （ ） 和继续传递 （ ）。我们的方法是对前者的扩展，结合了两种风格的转换：并行循环分割和并行循环互换。

1) Parallel Loop Splitting: 假设一个障碍 （ ） 的核函数 （ ） 作为其直接父节点。通过将障碍周围的循环拆分成两个并行的 循环，分别执行障碍之前和之后的代码，可以消除该障碍。如果障碍之前的代码创建了在其之后使用的 价值（ ），这些必须在第二个并行循环中存储或重新计算。我们采用类似于 中的技术来确定需要存储的最小数据量。具体来说，我们可以通过创建所有 价值的图来表征（ ）这个问题。然后，我们将所有无法重新计算的价值定义（例如，从被覆盖的内存中加载）标记为源（ ），而在障碍之后使用的价值标记为汇（ ）。通过对该图执行最小分支割（ ），我们可以推导出需要存储的最小数据量。

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

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

图 6. 示例：在屏障周围拆分的并行循环。这里，屏障上方的代码（包含2次加载和3次浮点操作）与屏障下方的代码（包含对`@use`的调用和其他操作）放置在一个单独的并行for循环中。此变换消除了屏障，同时保持了语义。由于值`%a`、`%b`和`%c`在屏障后被使用，因此必须格外小心以确保它们可用。在这里，最小割算法（min-cut algorithm）保留了`%x`和`%y`，然后重新计算`%a`、`%b`和`%c`，因为这样会导致保留2个值而不是3个值。

2) Parallel Loop Interchange: 不是所有的屏障操作都有一个并行 作为其直接父节点，有些可能嵌套在其他控制流操作中。我们创建了一个模型 ，指定可以并行运行的指令。除了 （屏障）以外，我们的表征 不要求程序的任何特定顺序或并发性。因此，添加额外的屏障是合法的（尽管可能会降低并行性）。我们可以利用这个特性来实现控制流的屏障降低。

考虑一个控制流构造 ，它包含一个屏障并嵌套在一个并行 中。在 的周围立即添加屏障将导致在 的上下立即进行并行循环拆分。因此，位于 上方和下方的操作将被分离到各自的并行 中， 将成为中间循环中唯一的操作。然后，我们可以应用以下技术之一，将 与并行 进行交换，从而使后者成为 （屏障）的直接父节点。

考虑包含屏障的串行 循环的情况，见图 。这种模式在 代码中很常见，例如，实现线程之间的归约 。由于 必须等待所有线程，因此每个线程必须执行相同数量的 。因此，内部循环的迭代次数对所有线程都是相同的，允许循环交换。

虽然 语句可以被视为具有零次或一次迭代的循环，但在必要时，直接与周围的并行 进行交

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

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

图 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.

of transformations: *parallel loop splitting* and *parallel loop interchange*.

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.

2) Parallel Loop Interchange: Not all barrier operations have a parallel `for` as their immediate parent, some may be nested in other control flow operations. We created a model that specifies what instructions may run in parallel. With the sole exception of `barrier`, our representation

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

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

图 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.

does not require any specific ordering or concurrency to the program. Therefore it is legal (though potentially a reduction in parallelism) to add additional barriers. We can use this property to implement barrier lowering for control flow.

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`.

Consider the case of a serial `for` loop containing a barrier, Fig. 7. This pattern is common in GPU code, e.g., to implement a reduction across threads [30]. As `barrier` must wait for all threads, each thread must execute the same number of `barrier`s. Therefore, the number of iterations of the inner loop is the same for all threads, allowing for loop interchange.

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

In MLIR, `for` loops must have their number of iterations computed prior to the loop. A `while` loop supports a dynamic exit condition. For example, consider the code in Fig. 8. To preserve correctness, we must execute the `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

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

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

图 7. ***左:*** 一个共享内存加法，包含一个内核调用，其中有一个带屏障（barrier）的for循环。 ***右:*** 通过将并行for循环与串行for循环进行交换，现在在并行循环中直接包含屏障（barrier）的相同代码。

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

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

图 8. 在 `while` 循环周围的并行交换。由于 `condition()` 函数调用必须在每个线程上执行以保持正确性，因此使用一个辅助变量来保存第一个线程上调用的价值（value）。

换更有效。

在　　　中，　　　循环必须在循环之前计算其迭代次数。　　　循环支持动态退出条件。例如，考虑图　　中的代码。为了保持正确性，我们必须在每个线程中执行

C. Usage

　　　转译（　　　）与　　　的设计旨在通过允许　　　作为现有　　　编译器（如　　　）的替代品，便于使用。具体而言，　　　扩展了　　　前端，因此使用与　　　相同的标志和语法。然而，　　　还引入了一些额外的标志，例如　　　用于指定　　　到　　　的转换，以及　　　用于指定用于生成　　　程序的特定方法和并行优化集（参见第　　　节）。

IV. Parallel Optimization

　　　提供　　　的并行性和　　　程序的高级表征（　　　）使各种优化成为可能。这些优化包括适用于任何并行程序的一般优化，以及在　　　转换为　　　的背景下的特定优化。

A. Barrier Elimination & Motion

由于　　　样式的屏障必须特别转换以支持　　　架构，因此消除或简化任何屏障可能会产生显著的影响。此外，即使在　　　上运行　　　代码时，屏障的消除也是非常有用的，因为任何同步都会减少并行性。大部分关于屏障消除和简化的基础设施直接来源于第　　　节中定义的其内存行为。给定一个屏障　，设$M^{\dagger}_{before}$为在　之前，直到另一个屏障或并行区域开始的内存效果的并集，设在　之后，直到并行区域结束的内存效果的并集为M_{after}。　如果在屏障跨越的同一位置没有内存效果　除了读取后读取（　　　）之外（即$(M^{\dagger}_{before} \cap M_{after}) \setminus \mathrm{RAR} = \varnothing$），则这个屏障的行为被先前的屏障所涵盖，可以被消除。对称条件$M_{before} \cap M^{\dagger}_{after} \setminus \mathrm{RAR} = \varnothing$表示该屏障被后续的屏障所涵盖。可消除屏障的一个特定简单情况是完全没有内存效果的屏障。

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

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

图 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 $M^{\dagger}_{before}$ 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. $(M^{\dagger}_{before} \cap M_{after}) \setminus \text{RAR} = \varnothing$), the barrier has its behavior subsumed by the prior barrier and can be eliminated. The symmetric condition $M_{before} \cap M^{\dagger}_{after} \setminus \text{RAR} = \varnothing$ 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`. $M^{\dagger}_{after}$ (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 bpnn_layerforward(...) {
  __shared__ float node[HEIGHT];
  __shared__ float weights[HEIGHT][WIDTH];
  if ( tx == 0 )
   node[ty] = input[index_in] ;
  // Unnecessary Barrier #1
  __syncthreads();
  // Unnecessary Store #1
  weights[ty][tx] = hidden[index];
  __syncthreads();

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

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

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

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

图 9. 一个来自Rodinia反向传播测试的CUDA内核示例，其中包含不必要的同步和不必要的共享内存使用，而寄存器（register）就足够了。

例 如， 考 虑 来 自 基准测试 的图 中的代码。第一个和最后一个 指令是不必要的。可以通过上面的基于内存的屏障消除算法证明这一点。对于第一个屏障，M_{before}（一直到开始）仅包含对 的写入和对 的读取。$M^{\dagger}_{after}$（直到第二个 ）包含对 的写入和对 的读取。如果在调用语境中已知指针不会发生别名，则这些之间没有冲突。因此，消除这个屏障是安全的。

同样的内存分析也可以用于执行屏障移动。只需在拟议移动的位置放置一个虚构的屏障，并检查先前的内存分析是否会推导出当前的屏障是多余的，从而允许屏障移动。

B. Memory-to-register promotion across barriers

从其内存行为定义 语义的目标之一是使内存优化能够在包含障碍的代码中正确有效地进行。如在第 节中所述，障碍有上下文代码的内存行为，显著的例外是来自当前线程的访问。这一缺口很重要，因为它使得内存到寄存器的提升能够在线程本地内存（例如局部变量）上进行。此外，这种优化能够成功地用快速寄存器替换慢速内存读取。例如，再次考虑图 中的代码。考虑标记为“无用存储 ”和“无用加载 ”的对 的加载和存储，以及两者之间的同步。在那个时刻，唯一可以加载的值是之前由同一读取存储的值。此外，由于在任何其他人可以从 读取之前，该位置已被覆盖，因此只要加载使用包含从 加载的值的寄存器，第一次存储就可以安全地消除。在内存到寄存器优化期间 现在能够成功推导出这一转发属性 因为第 节中描述的内存属性的缺口使其能够推断出障碍操作不会覆盖当前线程的存储。因此，传统的加载和存储转发能够正确地在障碍代码上操作。

C. Parallel loop-invariant code motion

传统的循环不变代码移动优化旨在将指令 移动到串行 循环之外，从而减少 的执行次数。如果 可能访问内存或具有其他副作用，除了检查 的操作数本身是循环不变的之外，编译器还必须检查 循环内没有其他代码与 执行的内存访问冲突。

在目前的编译器上，虽然可以将循环不变代码移动应用于 核心内的串行 循环，但无法将其应用于推动指令（ ）到核心调用外。这部分归因于 核心与调用它们的 代码保存在不同模块中的事实，以及传统编译器对并行性的理解不足（见图 ）。

反直觉的是，凭借正确的语义，即使我们无法将循环不变代码移动应用于等效的串行循环，我们也可以将其应用于并行 循环。我们将依赖于程序的语义，它允许我们在保持屏障所需顺序的前提下，任意交错并行 循环的迭代。因此，可以合法地想象在锁步运行程序。也就是说，如果一个并行 循环有 条指令，则每个线程会在任何线程执行指令 之前先执行指令 ，依此类推。因此，只要其操作数是不变的，并且并行 循环中的没有 先前 指令与 冲突，就可以合法地推动一条指令。换句话说，不需要检查 是否与并行 循环中的任何后续指令存在冲突即可允许推动。

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

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

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

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

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

图 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

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

图 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 `I`.

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

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

图 10. 在 MLIR 上应用的 OpenMP 并行区域融合示例。给定两个相邻的 OpenMP 并行区域，每个区域都用给定的闭包（closure）初始化线程，沿着线程屏障（thread barrier）融合这两个闭包，从而允许线程只初始化一次，而不是两次。

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

图 11. OpenMP并行区域提升的示例，如在C/C++中应用。这是图 10中OpenMP并行区域合并的扩展，不同之处在于它应用于整个for循环。与其为外层循环的每次迭代i创建一个单独的闭包（closure）/线程初始化，不如只创建一次闭包/线程初始化。

D. Block Parallelism Optimizations

开放 是我们在 上进行并行执行的主要目标。它通过两个构造实现并行“ ”循环。首先，循环被概述为一个函数，该函数每个线程调用一次，代表了开放 的“并行”( ) 构造。然后，在概述的函数内，迭代空间在线程之间分配，代表了开放 的“工作共享循环”( ) 构造。开放 还有一个“屏障”( ) 构造，但其语义与 屏障的语义不同。

当多个并行循环连续执行时，例如，在第 节中的屏障降低之后，通过融合相邻的开放 “并行”构造 而不融合工作共享循环（见图 ），可以减少线程管理的开销，从而不撤销屏障降低。并行区域融合可以扩展到各种构造，例如将开放 并行区域移至图 中包围的“ ”之外。这将使线程初始化只调用一次，而不是N次。将其一般应用于控制流构造使得通过对一个块执行并行循环裂变生成的所有并行 循环都可以进行开放 并行（但不是工作共享循环）的融合。

由于 程序往往是在考虑高并行性的情况下编写的，不同块提供的并行性可能已经单独饱和了可用核心的数量。如果没有使用共享内存，块和线程并行性可以折叠为一个单一的开放 并行 ，这将在单个并行区域中均匀划分总的迭代空间。然而，如果存在共享内存，我们的工具将生成嵌套并行区域以表示共享内存的分配。在这种情况下，嵌套的开放 并行区域带来的额外开销可能会超过潜在的附加并行性。此外 内循环的并行化可能会导致不利的内存效应 如假共享（ ） 进一步影响性能 。因此，我们还支持一种优化，用于序列化任何嵌套的开放 并行区域。进行这样的序列化可能会利用内存局部性来提高性能。

V. MocCUDA: Integration into PyTorch

A. Targeting CPU-only Supercomputers

我们工作的一个目标是对 中的 执行模型进行建模，以便将为 原本编写的高性能代码在仅有 的超级计算机上执行，尤其是在配备 处理器的 机器上 。作为一个主要例子，我们考虑 ，它尚未成功移植到 架构。 使用其“ ”( ) 后端支持 执行，但对于许多内核只提供了一个 ( ) 且表现不佳的实现，例如，对于二维卷积没有内存优化的 层嵌套循环。在英特尔 上，计算密集型内核的执行被委托给 库 ，而在 上表现不佳，因为它是为普通 定制的，而 上没有可用的高带宽内存。富士通的 分支 （我们在第 节中也进行了比较）需要手动调优以提高性能，而且仍然无法在普遍情况下与 竞争。有趣的是，具备高带宽内存的 可以受益于与 类似的计算组织。因此，我们设计了“ ”来模拟 中的 后端，通过将 内核转化为支持 的并行 内核，使用前面章节中描述的机制。

B. Architecture of MocCUDA and Integration of Polygeist

我们实现了一个兼容层 ，以便在 上透明地执行 的 后端，目的是避免手动重构和与现有库的互操作性。虽然 确实有其自身的 后端，但其设计与在 上运行的超大规模并行计算机的兼容

```
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
}
```

图 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, MocCUDA, 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

性较差。尤其是，性能问题是由同步内核执行、不匹配的内存布局和假设低速内存引起的。

我们并没有尝试对 的 后端进行艰巨而耗时的重新设计，而是决定基于 的 后端设计来构建 ，因为该平台与 有相对的相似性。因此， 必须处理并替换四种类型的操作：

1)
2) 在 中调用深度学习特定函数，
3) 对辅助 库的调用，并且
4) 在 中 执 行 内 核 （ ）。

只有后者可以被编译，因为其余的功能作为二进制库分发。

为了最小化原型的工作范围，我们分析了 与 后端的所有交互，以确定（ ） （ ）对于一个众所周知的深度学习问题，即 神经网络。

从分析中可以看出， 与 的交互主要限于识别已安装 的属性、内存管理以及 流的管理和同步。对于原型，我们将自己限制为每个 节点模拟一个 ，并只管理显式内存分配和数据传输。为了让 访问 设备属性，我们将真实 （ ）的数据转储到一个文件中，后者稍后由没有 的系统上的 使用。我们通过苹果的

模拟 流，使得模拟 操作与 的管理层之间的异步性得以实现。

模型仅使用一部分闭源 函数（卷积层的前向和后向传递、批量归一化层以及张量加法）。我们重新实现了所有必要的 函数变体，重点是 并行化和 友好的 加上 卷积，以及对 布局（批次 、通道 、高 、宽 ）的支持。此外，我们识别了对 库的调用，并实现了包装函数来拦截这些调用，并将它们分发到为 设计的 库，如 、 或富士通的 对于 。对于其他链接到 的库（如 ），采用类似的方法是可能的，但对 来说并不是必要的。

除了 和其他库调用外，我们观察到自定义内核，即在 中用 实现的内核，而不是由（二进制）库提供的内核，并识别出调用它们的高阶函数。 所需的函数包括跨步张量内核（加法、乘法等）、聚合操作如“ ”等。负对数似然损失内核使用 的 屏障。将这些基于 的函数移植到 需要大量的改造工作，而 则执行自动翻译。为了演示目的，我们使用 自动将使用 的 数及其递归调用的函数从 翻译到 ，并将结果集成到 中。

我们将上述所有的包装器和重新实现合并到 中， 该工具可与 一起使用，以拦截 调用，用于在 上训练带有 后端的 。

VI. Evaluation

我们展示了我们方法在两个著名的 基准测试套件上的优势和适用性： 基准测试套件的一个子集 和 神经网络的 实现。这些基准测试的选择是为了 提供我们 在具有手工编码 版本的基准测试套件（ ）上的与 编译的粗略性能比较，以及 展示我们的系统成功地集成到一个有用且实际的应用（ ）中，在不具有任何 的超级计算机 上。此外，我们还将我们方法的性能与现有的 工具在 矩阵乘法上的性能进行了比较。

对于 ，我们将翻译后的 与 代码与 版本的基准进行比较（在存在的情况下），以及在 上的运行。对于 ，我们与“原生”（ ）和 后端进行比较。

[1] 是在提交

[1]相关版本的MocCUDA和Polygeist可以在 https://anonymous-data.s3.amazonaws.com/MocCUDA-master.zip 和 https://anonymous-data.s3.amazonaws.com/Polygeist-main.zip 获得。

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 demonstration purposes, we use Polygeist to automatically translate the `ClassNLLCriterion_updateOutput`, which uses `__syncthreads()`, and `ClassNLLCriterion_updateGradInput` functions, and the functions transitively called from these, from CUDA to OpenMP, and integrate the result into MocCUDA.

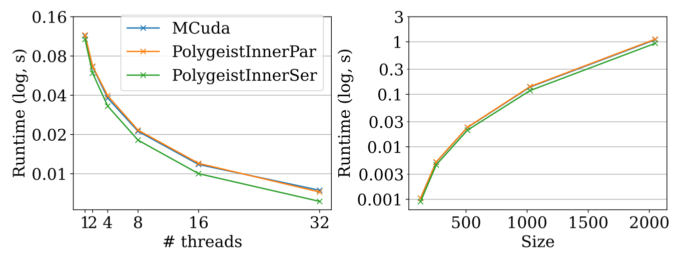

图 12. PolygeistInnerPar has a similar performance to MCUDA, while PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similaly to MCUDA, whereas PolygeistInnerPar keeps both the blocks and threads parallel. Left: Average runtime as a function of thread count (averaging over matrix sizes). Right: Average runtime as a function of matrix size (averaging over thread counts).

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[1] was compiled against LLVM 15 at commit

[1]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.

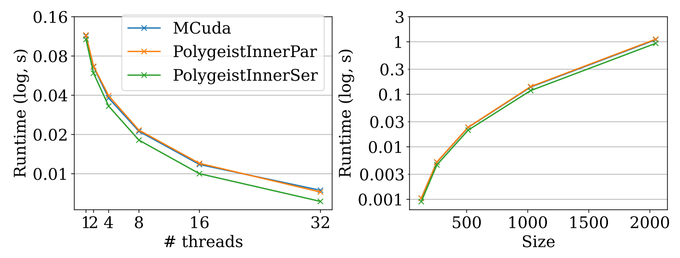

图 12. PolygeistInnerPar 的性能与 MCUDA 相似,而 PolygeistInnerSer 超过了 MCUDA。PolygeistInnerSer 类似于 MCUDA 禁用内循环并行化,而 PolygeistInnerPar 保持了块和线程的并行性。左:平均运行时间随线程数的变化（对矩阵大小进行平均）。右:平均运行时间随矩阵大小的变化（对线程数进行平均）。

下针对 编译的。对于 ，我们使用 的 为 ， ，以及 的 库进行 的编译。对于基准 测量，我们使用 预安装的 （ ）。

我们在运行 的 实例（双插槽 ，频率为 ，每个插槽有 个核心和 内存）上评估 和矩阵乘法测试。测量是在第一个插槽上进行的，同时禁用了超线程和涡轮提升。每个数字是至少 次重复的中位数。

A. Comparison to MCUDA

首先，我们将我们的方法与 中的先前工作进行比较。 是一个 级别的工具，它生成新的 作为输出，并使用类似的循环分裂技术来处理同步。作为一种源到源的工具，它仅处理输入语言的一部分，这使得它无法处理 程序。相反，我们在图 中比较了在不同线程数（ ）和矩阵大小（ ）下的矩阵乘法内核的运行时间。 在不包含内循环序列化（ ）的所有优化下 平均产生的代码与 相差。具体而言 在 个 线程下有的减速 在 个线程下有的加速。这种行为是由 在处理嵌套并行构造时的开销引起的。实际上 仅对最外层循环进行并行化。当 对内循环进行序列化（ ）时 它在整体上实现了相较于 的加速 其中在 个线程下加速 在 个线程下加速。

[2]尽管我们将在没有GPU的系统上运行PyTorch,但我们必须在支持CUDA的系统上编译PyTorch,以确保发出正确的代码。我们还防止了三个Pytorch函数的内联。

B. Use case 1: Rodinia Benchmarks

我们基准测试了一 由 个基准组成，这些基准目前被 支持，并且具有非平凡的运行时间。 我们通过比较用 编译并在 上执行的程序输出与通过我们的流程编译并在 上执行的程序输出来验证我们流程的正确性。我们在内核和 或包含内核的代码的计算部分中插入了时间测量，在某些情况下，每个基准多次测量。可能的情况下，我们计时相同基准的 版本的等效部分。

我们在图 右中比较了编译为 的 基准与 基准版本。尽管基准之间存在一些变化，但总体而言，我们的方法与手动编码的基准版本相当，甚至在启用内部串行化优化时净获得了 的几何平均性能提升。在没有内部串行化的情况下，我们仍然看到 的几何平均加速。一些基准如 和 采用了用于模板计算的优化技术，通过在线程之间复制计算以减少同步开销，更好地利用 中可用的并行性。这使得 代码比 版本复杂得多，导致其性能较差。由于程序在共享内存中执行额外的工作进行数据缓存 和 的 版本速度较慢。 的 本获得了相对加速，因为纯 版的代码通过单独的 " " 循环获得期望的依赖结构，而在 代码中则用 来完成。 能够成功优化围绕屏障的代码，从而实现加速。 的加速部分归因于并行优化

[3]`hybridsort`、`kmeans`、`leukocyte`、`mummergpu`、`huffman`和`heartwall`在Polygeist中使用了不被支持的C++或CUDA特性（虚函数和纹理内存）。`lavaMD`和`dwt2d`基准使用了格式不正确的C++，由于从未初始化内存中读取而导致未定义行为（GPU驱动程序将共享内存零初始化，但没有这个要求）。`nn`和`gaussian`测试在≤ 0.005秒内完成。

[4]由于`nn`的运行时间很短，已经被排除，这两个版本在数据加载方面有所不同（在运行时动态加载，在OpenMP代码中，以及在CUDA代码中预加载所有数据），因此也应该排除。

`89525cbf`. For the PyTorch Resnet-50, we compile Pytorch v1.4.0 using Nvidia's CUDA 11.6 SDK for Arm[2], LLVM 13, and Fujitsu's SSL2 v1.2.34 library. For the baseline PyTorch measurements , 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.[3] We verified correctness of our flow by comparing the program outputs produced by compiling with `nvcc` and

[2]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.

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

We compare the Rodinia CUDA benchmarks compiled for the CPU with the Rodinia OpenMP verions 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 `srad_v2` 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

[4]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.

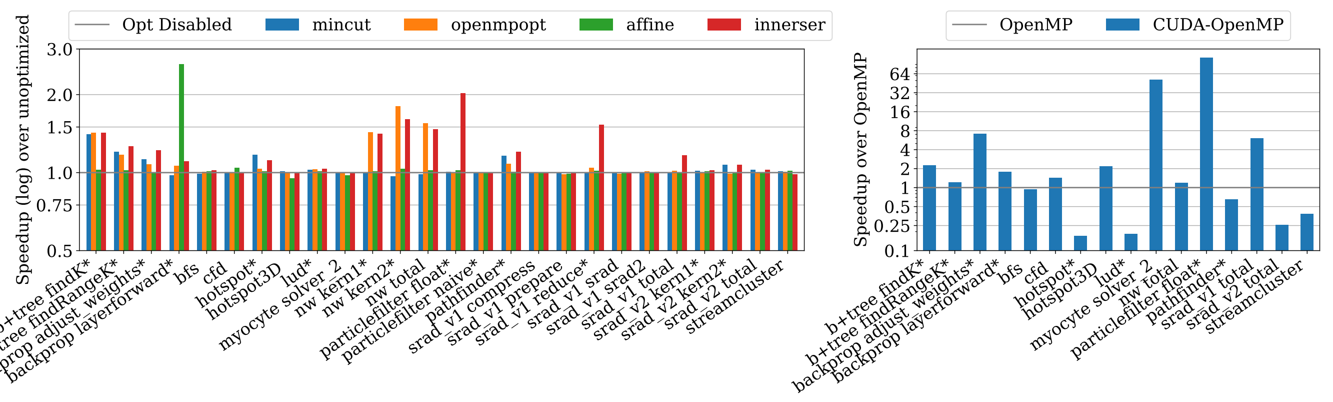

图 13. 左侧：应用各种并行和/或CPU优化的内核的相对加速（越高越好）。右侧：Rodinia CUDA代码编译为OpenMP与原生Rodinia OpenMP代码的加速对比（在可用的情况下）。包含屏障的基准测试用星号标记。

（见图　　　左　），部分原因是　　　代码采用线性数组实现，这是　　　的要求，而不是　　　代码中使用的双指针。　　　和　　　均因在并行区域边界内的代码优化以及内部串行化而获得加速。

我们通过在图　　中比较转换后的　　　与原生　　　内核来测试我们方法的缩放特性。转换后的　　　代码通常比原生　　　版本扩展得更好。由于大多数　　　程序在设计时考虑了成千上万的线程，这表明我们的框架能够在将　　特定构造重写为　　兼容的等效项时保留这种并行性。在　个线程且没有内部串行化的情况下，所有测试的转换后　　　代码具有的几何平均加速。由于并非所有测试都存在　　　　版本的基准，如果我们仅考虑存在　　　　版本的　　　代码，我们发现其几何平均加速为，而　　　　的加速仅为。对内部循环进行串行化略微降低了可扩展性，但仍然在所有启用内部串行化的测试中获得的几何平均加速，并在具有　　　　版本的代码中获得的加速。

我们进行了一次消融分析，以研究单个优化对性能的影响。图　　　左　中的　　　　　　系列显示了我们的方法在第　　　　节中概述的优化下的性能测量，以减少在屏障之间保留的数据量。这仅与包含屏障的基准测试相关（在图中以星号标记）。在适用的基准测试中，　　　　提供了　　　的几何平均加速比（　　　　　　　　　　）。图　　　左　中的　　　　　　　系列展示了　　　　区域合并及类似优化的影响，并带来了　　　的几何平均加速比。图　　　左　中的　　　　　　系列显示了将控制流提升

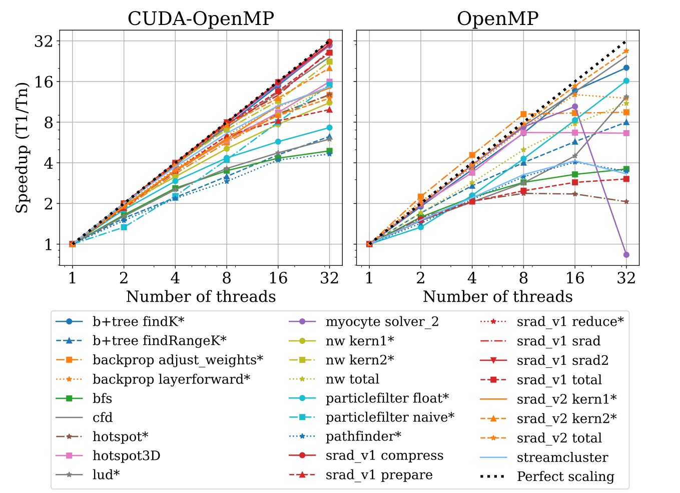

图 14. CUDA Rodinia 内核的可缩放性行为，当在使用 32 个线程的 CPU 上运行并利用 OpenMP 时，以及 OpenMP Rodinia 内核（如有）之间的比较。并非所有的 Rodinia CUDA 内核都有 OpenMP 版本。

到其仿射（　　　　　）变体并启用简单循环优化（例如循环展开）的结果。尽管这在整体上产生了　　　的几何平均加速比，但对于反向传播层前向测试（

　　　），它产生了的加速，因为它将一个包含同步的循环完全展开并简化为　　语句。

C. *Use case 2: Pytorch/Resnet50 Test*

为了评估　　　　　　　　　　　　　　　，我们在　　　　　　　　　　　　超算的一个　　　　单元上执行了完整的节点并行训练，与原生的　　　　　　　后端和优化的　　　　　后端进行比较。

我们以数据并行的方式在　　　的　　　　　上进行了多次　　　　　的正向和反向传播。我

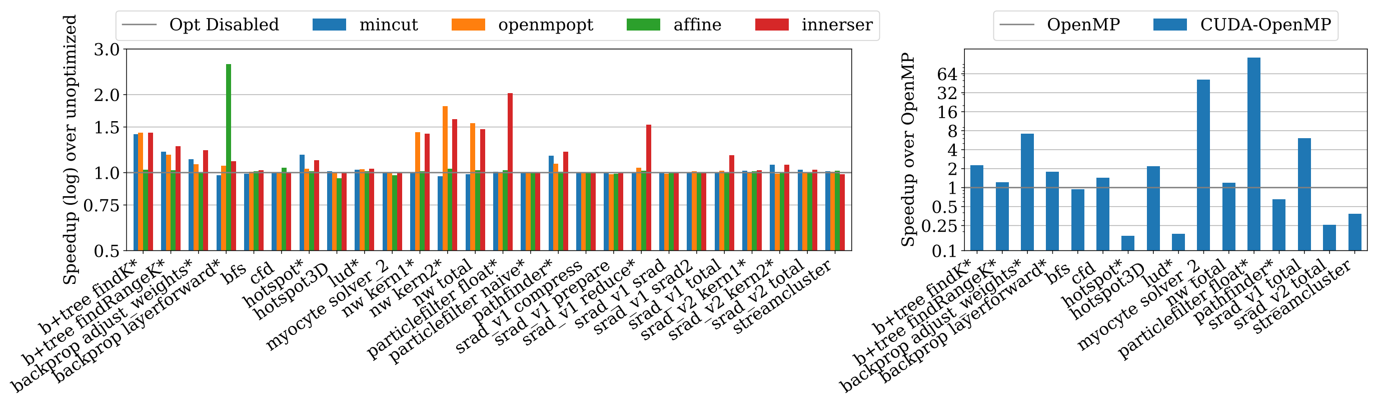

图 13. Left: Relative speedup (higher is better) of kernels with various parallel and/or CPU optimizations applied. Right: Speedup of Rodinia CUDA code when compiled to OpenMP compared against native Rodinia OpenMP code (when available). Benchmarks containing barriers are marked with an asterisk.

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 "openmpopt" 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 result 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.

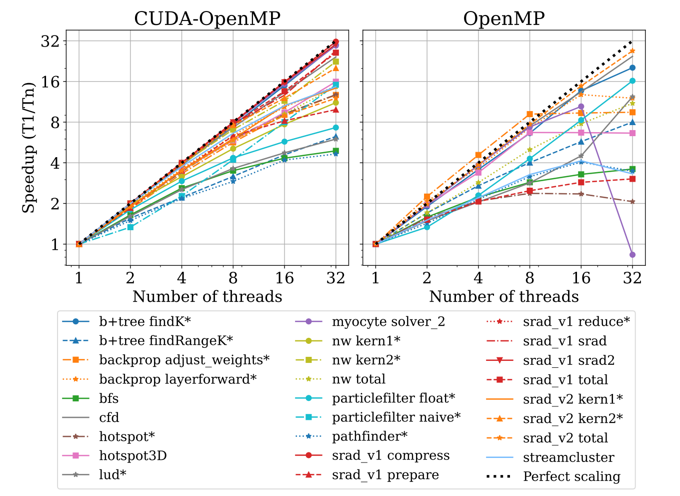

图 14. Scaling behavior behavior of CUDA Rodinia kernels, when run on the CPU with OpenMP, and OpenMP Rodinia kernels (where available), using 32 threads. Not all Rodinia CUDA kernels have OpenMP versions.

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

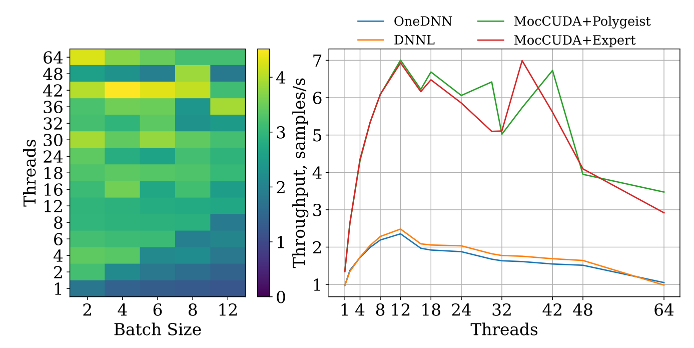

图 15. 在Fugaku节点上对ResNet50进行训练。左:"MocCUDA+Polygeist"相对于富士通-调优版oneDNN的相对吞吐量增加的热图，值越高越好。右：不同批次大小的几何平均吞吐量；"MocCUDA+Expert"使用专家编写的OpenMP内核；"MocCUDA+Polygeist"使用生成的内核。

们使用 的合成基准测试脚本（为 神经网络模型配置） 。我们构建了带有 、 和富士通 库的 ，以支持基于 的多节点、分布式深度学习。我们为每个 核心内存组（ ）分配一个 排名，模拟每个节点最多 个 ，并将测试范围从一个节点（ 个排名）扩展到 个节点（ 个排名）在一个 单元中（最小为 环形拓扑），同时将 线程数固定为 ，以容纳每个核心一个线程。我们的方法使用 ，而其他后端依赖于 。性能测量使用 进行，该工具通过 组织设置神经网络，创建图像，使用 执行层，并返回图像每秒的吞吐量指标。我们在批量大小 线程数 的情况下运行，跨轮次进行平均。

我们观察到 在批量大小和线程数量方面系统性地优于 调优的 最高可实现 的吞吐量提升（几何均值 最小值 ）如图 所示。使用我们专家编写的内核的 与使用由 生成的内核的 相当，如 节所述。

这种改进可以通过第 节中描述的 设计和 性能特征相结合来解释。由于 的 未考虑 上可用的 ，因此它使用对缓存友好的直接卷积，而不是基于 的卷积，在存在高带宽内存的 上效率较低。尽管 调优的 的定制分支 提高了 的性能（尽管几何均值仅提高），但仍留有改进性能的空间。

这表明，我们的方法能够从 版本自动推导出深度学习内核（以及潜在的其他应用程序）的高效版本，从而解决缺失或低效内核在具有高带宽内存的 上的局限，而不需要反向工程或重新工程应用程序。

VII. Related Work

A. GPU to CPU Synchronization

直接提供的第一个用于在 上模拟 的工具之一是为了调试目的而创建的，它在 上模拟每个线程，并配备一个独立的 线程。尽管功能上可行，但可用线程数的巨大差距使得仿真效率低下。

（ ）对 代码执行 转换，以生成新的 代码，该代码调用一个与线程无关的并行 例程。 开创了使用"深裂变"（ ）处理同步的使用方法，该方法在同步点拆分并行循环和其他构造，目的是消除它们。这种裂变技术也应用于其他工具： （ ），一个将 汇编语言解析为 并即时编译内核函数的二进制翻译工具； （ ），一个用于 的 编译器传递； （ ），另一个采用裂变的 翻译 转换传递，并显著处理 级原语；甚至还有本研究。 虽然裂变方法背后的直觉与此处使用的方法类似 但我们是在一个高级编译器内部应用裂变 而不是在源代码或低级 中。如在第 节所示，对结构化程序执行裂变使代码变换更加高效。在源级别执行裂变会错失在裂变之前运行优化（如屏障消除）的机会，而在低级别执行裂变又需尝试重建高级结构，在 中工作使我们能够同时应用优化并保留高级结构。此外，源级工具往往相当脆弱，因为它们必须重新实现解析和目标语言（如 ）的语义，因此仅在输入语言的有限子集上操作，导致需要工程重构以替代不受支持的构造（如指针算术）。

另一种方法使用继续传递（ ）来处理同步，通过创建所有同步点的状态机（例如"微线程"（ ））（ ）。 和

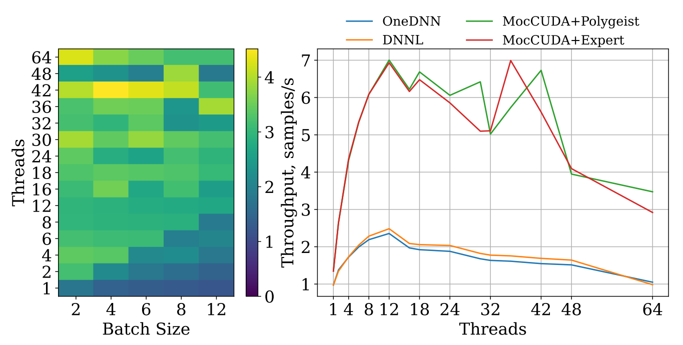

图 15. ResNet50 training on Fugaku node. Left: heatmap of relative throughput increase of "MocCUDA+Polygeist" over Fujitsu-*tuned* oneDNN, higher is better. Right: geomean throughput across batch sizes; "MocCUDA+Expert" uses an expert-written OpenMP kernel; "MocCUDA+Polygeist" uses the generated kernel.

one TofuD unit (smallest 2×3×2 torus) while keeping the number of OpenMP threads fixed at 12 to accommodate one thread per core. We use Pytorch v1.4.0 for our approach, while the other backends depend on Pytorch v1.5.0. Performance measurements were taken using Benchmarker [37], which orchestrates sets up the neural network via torchvision, creates images, executes the layer using PyTorch, and returns an images/s throughput metric. We run with batch sizes 1–12 on 1–64 threads, averaging across epochs.

We observed that MocCUDA systematically outperforms Fujitsu's tuned oneDNN across batch sizes and thread counts, yielding up to 4.5× throughput increase (geomean 2.7×, min 1.2×) as shown in Fig. 15. MocCUDA with our expert-written kernels is comparable to MocCUDA with the kernels generated by Polygeist, described in Section V-B.

The improvement can be explained by a combination of the PyTorch CPU design and performance characteristics of oneDNN described in Section V-B. As Intel's oneDNN [3] does not account for HBM available on A64FX, it uses cache-friendly direct convolutions instead of GEMM-based convolutions, less efficient in presence of HBM for Arm CPUs. While the custom fork of oneDNN tuned by Fujitsu [4], improves upon Intel oneDNN's performance (though by a geomean of 6%), it still leaves room for performance improvements.

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

VII. Related Work

A. GPU to CPU Synchronization

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

MCUDA [11] (2008) performs an AST transformation of C GPU code to generate new C CPU code that calls a thread-independent parallel `for` routine. MCUDA pioneered the use of "deep fission" to handle synchronization, which splits parallel loops and other constructs at synchronization points in order to eliminate them. This fission technique is also applied in other tools: Ocelot [12] (2010), a binary-translation tool that parses PTX assembly into LLVM and just-in-time compiles kernel functions; POCL [38] (2015), a Clang/LLVM compiler pass for OpenCL; COX [13] (2021), another LLVM transformation pass for translation of CUDA that uses fission, and notably handles warp-level primitives; and even this work. While the intuition behind the fission approach is similar to that used here, we apply fission inside of a high-level compiler, rather than either source or a low-level IR. As demonstrated in Section III-A, performing fission on structured programs enables more efficient code transformations. While performing fission at a source-level misses the opportunity to run optimizations (such as barrier elimination) before fission and applying fission at a low-level requires attempting to reconstruct the high-level structure, operating within MLIR allows us to both apply optimization and preserve high-level structure. Moreover, source-level tools tend to be quite fragile as they must re-implement parsing and semantics or the target language (e.g. C++), and as a result only operate on a limited subset of the input language, requiring re-engineering effort to replace unsupported constructs (like pointer arithmetic).

Another approach uses continuation-passing to handle synchronization by creating state machine of all synchronization points (e.g. "microthreading") [39] (2010). Karrenberg and Hack [40] (2012) propose a continuation-passing approach in LLVM that includes an algorithm for detecting and reducing divergence in the control-flow-graph, with follow up work to minimizing live values to reduce memory traffic [41].

VGPU [42] (2021) is Similar to NVidia's original virtual GPU, except now uses C++ `std::thread` and performs

（ ）提出一种在 中采用继续传递的方法，包括检测和减少控制流图中分歧的算法，后续工作旨在最小化活跃值以减少内存流量 。

（ ） 类 似 于 的原始虚拟 ， 但现在使 用 的 并 通 过 执行同步。共享内存在 中作为单个全局变量实现，并按块数进行扩展。

在低级 上工作的先前研究付出了巨大努力，以重建高层构造，例如循环和内核配置，这些在高效裂变或继续传递中是必需的。例如，

进行各种规范化和循环变换，以重写控制流图并尝试将其识别为可以处理的几种形式之一。在源 级别工作的先前研究（例如 ），除了仍需识别 级概念外，无法受益于简化代码的优化，从而导致更易于控制的流。

相比之下，通过在 的混合抽象上操作，我们能够同时保留源级结构并执行程序变换，例如循环展开或 运动，可以，如，删除嵌套同步。

B. Parallel Portablity/IR, & OpenMP Optimizations

几个工具在主机语言中定义了新的抽象，这些抽象自然适合于 或 的执行。示例包括 中的 、 、 或 （仅限于 计算）， 中的 ，以及 中的 。这些方法为使用它们编写的任何新代码提供了性能可移植性。然而，任何现有代码都必须在所述框架中重写（并且可能与其他框架或其他语言编写的代码不兼容）。

几项先前的工作讨论了并行性的中间表征（ ），例如， 用于在 中表征 ； 用于在 中表征 ， 用于模式树，以及 方言；以及 用于以控制流图的形式直观地表征并发。这些工作主要集中在特定风格并行性的表征（ ）（例如 中的 任务），而不包括 风格的屏障，而不是并行特定的转换（ ）（如屏障消除）或优化，除了一致性 竞争检查或自动并行化 。

已知 并行区域扩展的使用是有益的 。 可选择以较弱的形式支持该转换，即在相同控制级别内合并并行区域 。

C. Barriers

几项先前的研究探讨了屏障或同步指令的语义，包括与 的相关性。已有研究验证了屏障的正确性 。 探索并实验评估了不同 供应商的前进进度 公平性模型。 实现了一种在工作组之间应用的 屏障操作，而不仅仅是在一个工作组内。 将 内存屏障添加到程序中，以确保弱一致性和顺序一致性语义。他们发现，在没有同步和延迟设置分析的情况下，引入一致性语义的平均减速为 ，而使用这些分析插入更少的同步操作时，弱一致性和顺序一致性分别可以实现 和 的减速。

VIII. Conclusion

通过扩展 ，我们开发了一个端到端系统，能够表征（ ）、 优化和转译 和 并行程序。能够同时表征和在不同并行框架之间转换是至关重要的，因为高性能计算（ ）越来越依赖（异构）（ ）并行性来处理其工作负载。我们框架的重要组成部分是高阶屏障操作的开发，这对于表征 程序至关重要，其语义可以通过其内存行为完全定义。与以前的并行屏障表征不同，我们的语义使得在优化过程中能够直接集成屏障。为了验证我们方法的有效性，我们展示了在商品 上对 基准套件的一个子集进行 到 的优化和转译，并将一个高效的 从 源转译为可在仅有 的超级计算机 上运行的代码。 尽管由于 和 之间实现的差异 性能在每个案例中有所不同 但 基准套件在转译的 代码与手写的 版本之间实现了 的几何平均加速。同样 内核在原生 后端之上也观察到了的加速。

目前 在 上运行的转译 代码保持相同的调度 除了最内层循环的序列化 这是为了提高性能。未

synchronization using `std::atomic_thread_fence`. Shared memory, implemented as a single global in LLVM, is expanded by the number of blocks.

Prior work that operates at the low-level LLVM IR extends significant effort to reconstruct high-level constructs, such as loops and kernel configurations, required for either efficient fission or continuation passing. For example, POCL [38] runs various canonicalizations and loop transformations to rewrite the control flow graph and attempt to recognize it as one of several forms that can be handled. Prior work that operates at source/AST level (e.g. MCUDA), beyond still needing to recognize GPU-level concepts, cannot benefit from optimizations that simplify the code resulting in easier control flow.

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

B. Parallel Portablity/IR, & OpenMP Optimizations

Several tools define new abstractions in the host language that are naturally amenable to CPU or GPU execution. Examples include ISPC [43], RAJA [44], Kokkos [45], or MapCG [46] (limited to map-reduce computation) in C++, Loo.py [47] in Python, and KernelAbstractions.jl [48] in Julia. These approaches provide performance portability for any new code written with them. However, any existing code must be rewritten in said framework (and may or may not compose with code written in other frameworks or other languages).

Several pieces of prior art discuss intermediate representations for parallelism, such as Tapir [15] for representing Cilk [49] in LLVM; OpenMPIR [50] for representing OpenMP in LLVM, PPIR [51] for pattern trees, and the MLIR OpenMP Dialect; as well as SDf3 [52] for visually representing concurrency as a control-flow graph. These works primarily focus on the *representation* for their particular style of parallelism (e.g. OpenMP tasks in OpenMPIR), which does not include GPU-style barriers, rather than on parallel-specific *transformations* (such as barrier elimination) or optimizations, with the exception of consistency/race checks or automatic parallelization [53], [54].

The use of OpenMP parallel region expansion is known to be beneficial [31]. Clang/LLVM optionally supports the transformation in a weaker form, namely merging of OpenMP parallel regions in the same control level [55].

C. Barriers

Several pieces of prior work have explored the semantics of barrier or synchronization instructions, including as it relates to GPUs. Work has been done to verify the correctness of barriers [56]. [57] explores and experimentally evaluates the forward progress / fairness models of various GPU vendors. [58] implements a GPU barrier operation that applies across work-groups, as opposed to just within a work group. [59] add Java memory barriers to programs to ensure weak and sequential consistency semantics. They find that without synchronization and delay set analysis, introducing consistency semantics has an average $26.5\times$ slowdown, whereas when using these analyses to insert fewer syncrhonzations can achieve a 10% and 26% slowdown for weak and sequential consistency, respectively.

VIII. Conclusion

By extending Polygeist/MLIR, we have developed an end-to-end system capable of representing, optimizing, and transpiling CPU and GPU parallel programs. Being able to simultaneously represent and convert between distinct parallel frameworks is crucial as HPC increasingly relies on (heterogeneous) parallelism for its workloads. A key component of our framework is the development of a high-level barrier operation, key to representing GPU programs, whose semantics can be fully defined by its memory behavior. Unlike prior representations of parallel barriers, our semantics enable direct integration of barriers within optimization. To validate the efficacy of our approach we demonstrate GPU to CPU optimization and transpilation of a subset of the Rodinia benchmark suite on a commodity CPU and transcompile an efficient Resnet-50 from the PyTorch CUDA source to be run on the CPU-only Supercomputer Fugaku. While there is case-by-case variance due to differences in implementation between CPU and GPU, the Rodinia benchmark suite achieves a 76% geomean speedup of the transpiled GPU code over handwritten OpenMP versions. Similarly, a $\approx 2\times$ speedup of CUDA PyTorch kernels above the native PyTorch CPU backend can be observed

Currently, the transpiled GPU code keeps the same schedule when run on the CPU, except for the innermost loop 来的一个有前途的研究方向可能是在代码的高级重调度上进行 以更好地利用 风格的内存层次结构。

Acknowledgment

感谢麻省理工学院的 和 谷 歌 的 就 中的转换进行的深思熟虑的讨论。感谢 、 和 就编译器中的并行优化进行的深入讨论。

在一定程度上得到了 计算科学研究生奖学金 的支持，在一定程度上得到了洛斯阿拉莫斯国家实验室的 号拨款的支持，并在一定程度上得到了美国空军研究实验室和美国空军人工智能加速器的支持，并在合作协议编号 下完成。 在一定程度上得到了美国能源部科学办公室应用数学活动的支持，获得合同编号 ；在一定程度上，得到了超算项目（ ）的支持，这是美国能源部两个组织（科学办公室和国家核安全管理局）负责规划和准备一个强大的超算生态系统的协作努力，涵盖软件、应用程序、硬件、先进系统工程和早期测试平台，以支持国家的超算需求。这项工作在一定程度上得到了日本学术振兴会 资助编号 的支持，以及日本新能源和工业技术开发机构（ ）的支持。

本文件中包含的观点和结论是作者的观点，不应被解释为代表美国空军或美国政府的正式政策，无论是明示还是暗示。美国政府被授权在不影响本文件中任何版权说明的情况下，为政府目的复制和分发印刷品。

参考文献

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.

参考文献

[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. Gimelshein, 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

[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

[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 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

[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

[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, 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 Multithreading 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. Jääskeläinen, 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

[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: 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^ 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. Rakamarić, "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.

e1f22da320b0c7384cbd2f4df50255c7c2fa6b9d

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

–

[illegible]

،

[illegible]