通过高级并行构造实现高性能 GPU 到 CPU 的转译和优化

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

***Abstract*—虽然并行性仍然是性能的主要来源，但架构实现和编程模型会随着每一代新硬件而发生变化，这通常会导致昂贵的应用程序重新设计。大多数性能可移植性工具都需要手动且昂贵的应用程序移植到另一个编程模型。我们提出了一种替代方法，该方法基于 Polygeist/MLIR 自动将用一种编程模型 (CUDA) 编写的程序转换为另一种编程模型 (CPU 线程)。我们的方法包括并行构造的表示，允许透明地应用传统的编译器转换而无需修改，并实现特定于并行性的优化。我们通过为多核 CPU 转译和优化 CUDA Rodinia 基准测试套件来评估我们的框架，并与手写 OpenMP 代码相比实现了 76 % 的几何平均加速。此外，我们展示了 PyTorch 中的 CUDA 内核如何在无需用户干预的情况下在仅限 CPU 的超级计算机 Fugaku 上高效运行和扩展。我们的 PyTorch 兼容层利用了转译 CUDA PyTorch 内核的性能优于 PyTorch CPU 本机后端 2.7 × 。**

I. 介绍

单核性能扩展的结束意味着并行性和领域特定性现在是提高效率的主要来源。超级计算机架构师竞相发挥创造力，以支持从物理模拟到机器学习的计算和内存密集型应用程序。最新、最快的超级计算机 Fugaku 完全基于 A64FX CPU，与商用 CPU 不同，它支持高带宽内存访问和与最近的 GPU 相当的能效 [1] 。

然而，由于最近的框架和高性能库都是针对 NVidia GPU 开发的，因此在实际应用中高效且高效地使用此类计算机是一项挑战。例如，将 PyTorch [2] 移植到 A64FX 的尝试遇到了多重挑战。“原生” 默认 CPU PyTorch 后端仅为关键内核提供 naïve 版本，例如以六个嵌套循环实现的 2D 卷积。英特尔的 oneDNN [3] 不出所料地在 Arm CPU 上表现不佳，因为它是为没有高带宽内存的商用 CPU 量身定制的。富士通定制的 oneDNN [4] 经过了更好的调整，但无法与 GPU 进行普遍竞争。

已经提出了许多实现性能可移植性的非库方法。它们包括语言扩展（例如 OpenCL [5] 或 OpenACC [6] ）、并行编程框架（例如 Kokkos [7] ）以及领域特定语言（例如 Spiral [8] 、Halide [9] 或 Tensor Comprehensions [10] ）。由于语言或底层编程模型 ，原始程序和目标框架 的差异，所有这些方法仍然需要 legacy 应用程序为 ported, and sometimes entirely rewritten, 。

我们探索了一种基于全自动编译器的替代方法，该方法采用一种编程模型（CUDA）中的代码并生成针对另一种编程模型（CPU 线程）的二进制文件。虽然过去已经探索过 GPU 到 CPU 的转换 [11]–[13] ，但很少能够生成高效的代码。事实上，由于编译器 [14] 中缺乏可分析的并行结构表示，CPU 的优化甚至通用编译器转换（如公共子表达式消除或循环不变代码移动）都受到阻碍。例如，考虑图 1 中的求和，它可以由每个块的单个线程完成，或者一旦所有计算都在 CPU 上进行，就可以由总共一个线程完成。由于主流编译器中的并行性表示最近才开始在主流编译器 [15]–[19] 中探索，现有的转换是有限的，并且往往仅适用于简单的 CPU 代码。

```
__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);
}
```

Fig. 1. 示例 CUDA 程序 `normalize` ，它规范化一个向量和调用内核的 CPU 函数 `launch` 。目前，每个线程都会调用对 `sum` 的调用，导致总共需要 $O(N^2)$ 项工作。通过使用共享内存（在上面的评论中），可以将工作部分减少到 $O(N^2/B)$ ，这使得每个块可以计算一次总和，或者通过在内核之前计算一次 `sum` 可以将工作完全减少到 $O(N)$ 。

我们为最常见的 GPU 结构提出了一个编译器模型：多级并行、级别范围同步和级别本地内存。这与 CPU 并行不同，CPU 并行提供单一级别的并行、统一的内存和对等同步。与在优化管道之前运行的源代码和 AST 级别方法以及将同步建模为“黑盒”优化屏障的现有编译器方法不同，我们完全从内存语义的角度对同步进行建模。这既允许基于同步的代码与现有优化进行互操作，又支持新颖的并行特定优化。

我们的模型在 LLVM 编译器基础结构 [21] 的 MLIR 层 [20] 中实现，它利用 MLIR 的嵌套模块方法处理 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到 LLVM 支持的任何 CPU 架构上。除了

考虑执行模型差异的转换之外，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，同时替换不受支持的调用。

通过编译 Rodinia CUDA 基准 [24] 以及 PyTorch CUDA 内核，我们评估了端到端翻译的正确性和效率。当针对商用 CPU 时，我们的 OpenMP 加速 CUDA 代码可产生与 Rodinia 套件中的参考 OpenMP 实现相当的性能，并且可扩展性得到改善。当使用我们的框架在仅限 CPU 的 Fugaku 超级计算机上运行 PyTorch 时，与基于 OneDNN 的 PyTorch CPU 后端相比，我们实现了 Resnet-50 [25] 的 `conv2d` 内核每秒处理图像的大约两倍，并且与手动调整的整体训练具有相当的性能。

总体而言，我们的论文做出了以下贡献：

- SIMT 风格并行性的通用高级和平台无关表示，由屏障同步的语义定义支持，通过内存语义确保正确性，从而确保现有优化的透明应用。
- 新颖的并行特定优化可以利用我们的高级并行语义来优化程序。
- MLIR 的 Polygeist C/C++ 前端的扩展，能够将 GPU 和 CPU 并行结构直接映射到我们的高级并行原语中。
- 针对 Rodinia [24] 基准测试套件的子集和 PyTorch [2] 中的内部 CUDA 内核进行 CUDA 到 CPU 的端到端转换，这是在仅有 CPU 的 Fugaku 超级计算机上运行 Resnet-50 所必需的。

II. 背景

主流编译器（如 Clang 和 GCC）缺乏统一的并行高级表示。具体而言，在 CUDA、OpenMP 或 SYCL 等框架中编译并行构造会强制并行区域的主体存在于由相应运行时调用的单独（闭包）函数中。然后，线程索引或同步等概念会单独表示，通常通过不透明的内部调用表示。由于编译器历来缺乏有关并行性和所涉及运行时的影响的信息，因此任何并行构造也会无意中成为优化的障碍。尽管近年来一直有人尝试改进 CPU 并行构造的表示，但加速器编程还面临着额外的挑战。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 并行的高级表示仍未得到充分探索。

A. GPU 编译

考虑图 1 中的 CUDA 程序，该程序对向量进行规范化。当使用 LLVM/Clang 编译时，GPU 程序是一个单独的编译单元，如图 2 所示。这会阻止 GPU 内核和 CPU 调用代码之间的任何优化。在图 1 的情况下，传统编译器中程序的总工作量为 $O(N^2)$，因为每个线程都会执行对 `sum` 的 $O(N)$ 调用。但是，如果对 `sum` 的调用只能在内核调用之前执行一次（例如，通过执行循环不变代码移动 (LICM) 转换），则工作量将减少到 $O(N)$。这种优化的一种效率较低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 为支持主机/设备代码移动 [22] 的 GPU 程序提供了嵌套模块表示，但尚未实现并行代码移动。在 GPU 到 CPU 的代码运动中，并行循环中的 LICM 始终是合法的，因为以前的设备内存在主机上也可用。

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

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

```
target triple = "nvptx64"

define @normalize(float* %out, float* %in, i32 %n) {
  %tid = call i32 @llvm.nvvm.ptx.tid.x()
  %sum = call i32 @sum(i32* %in, i32 %n)
  %cmp = icmp slt i32 %tid, %n
  br i1 %cmp, label %body, label %exit
body:
  %gep = getelementptr float* %in, i32 %tid
  %load = load float, float* %gep
  %nrm = fdiv float %load, %sum
  %ptr = getelementptr float* %out, i32 %tid
  store float %nrm, float* %ptr
  br label %exit
exit:
  ret
}
```

Fig. 2. 使用 LLVM/Clang 编译时，简化了图 1 中 `launch` 和 `normalize` 函数的降低。由于这两个函数发出不同的汇编代码，因此它们被放置在单独的模块中，没有关于如何调用它们或是否调用它们的上下文。

```
// 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>
      }
    }
  }
}
```

Fig. 3. Polygeist/MLIR 中图 1 中 CUDA `launch` / `normalize` 代码的共享内存变体的表示。内核调用直接在调用它的主机代码中提供。通过在块和线程中使用并行 for 循环明确了并行性，并将共享内存放置在块内，以表示可以从同一块中的任何线程访问它，但不能从不同的块访问它。

B. MLIR 基础设施

MLIR 是一种最新的编译器基础结构，专为重用和可扩展性而设计 [20] 。其内部表示 (IR) 可看作是广泛采用的 LLVM IR [21] 的概念后继者，对传统 CPU 之外的编程模型提供了更好的支持。MLIR 并不提供一组预定义的指令和类型，而是对包含可互操作的用户定义操作、属性和类型集的方言集合进行操作。操作是 IR 指令的泛化，可以任意复杂，特别是包含具有更多 IR 的区域，从而创建嵌套表示。操作定义并使用遵循单一静态赋值 (SSA) [27] 的值。例如，MLIR 方言可以模拟整个物理或虚拟指令集，例如 NVVM（Nvidia GPU 的虚拟 IR）、其他 IR，例如 LLVM IR、更高级的控制流构造，例如仿射循环、并行编程模型，例如 OpenMP 和 OpenACC、机器学习图等。MLIR 支持并鼓励在同一个编译单元中混合来自不同方言的操作。

MLIR 通过同名方言支持 GPU，该方言定义了高级 SIMT 编程模型以及主机/设备通信机制，以及一组低级平台特定方言：NVVM (CUDA)、ROCDL (ROCm) 和 SPIR-V。MLIR 对 GPU 编程方法的特殊性在于其统一的代码表示。由于 IR 的灵活性，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 blob。这种方法提供了其他编译器无法提供的主机/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. 波灵

Polygeist 是基于 Clang [23] 的 MLIR 的 C 和 C++ 前端。它能够将各种 C++ 程序转换为 MLIR 方言的混合，从而保留程序高级结构的元素。具体来说，Polygeist 将结构化控制流（循环和条件）保留为 MLIR SCF 方言。它还通过尽可能地依赖 MLIR 的多维内存引用 (memref) 类型来保留多维数组构造，从而简化分析。最后，Poylgeist 能够识别适合多面体优化 [28] 的程序部分并使用 Affine 方言表示它们。

III. 方法

我们扩展了 Polygeist 编译器 [23] ，使其直接从 CUDA 发出并行 MLIR。这利用统一的 CPU/GPU 表示，使优化器能够理解主机/设备执行，并实现跨内核边界的优化。使用现有 MLIR 的一流并行构造（`scf.parallel` 、`affine.parallel` ）使我们能够定位现有的 CPU 和 GPU 后端。最后，MLIR 的可扩展操作集使我们能够定义具有相关属性和自定义优化的自定义指令。

我们定义 GPU 内核启动的表示如下（如图 3 所示）：

- 遍历网格中所有块的 3D 并行 for 循环。
- 任何共享内存的堆栈分配，每个块的范围都是唯一的。
- 块中所有线程的 3D 并行 for 循环。
- 自定义的 Polygeist 屏障操作，为 CUDA/ROCm 同步提供等效的语义。

此过程使我们能够以保留所需语义的形式表示任何 GPU 程序。编译器完全理解它，因此适合编译器优化。此外，通过使用通用并行性、分配和同步结构表示 GPU 程序，我们不仅能够优化原始程序，还可以将其重新定位到不同的架构。

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

Fig. 4. 包含两个任意指令之间的屏障的程序。

A. 屏障语义

CUDA 或 ROCm `__syncthreads` 函数可确保块中的所有线程在函数调用之前已完成执行所有指令，之后任何线程才会在调用后执行任何指令。传统上，编译器将此类函数表示为可能触及所有内存的不透明优化屏障，并禁止涉及它们的任何转换。

在我们的系统中，我们选择通过新的 `polygeist.barrier` 操作来表示这种线程级同步。与其他方法不同，`polygeist.barrier` （因此简称为 `barrier` ）旨在仅防止会改变外部可见行为的转换。我们可以通过将 `barrier` 定义为具有特定内存属性，表示为内存位置（包括未知）和内存效应类型（读取、写入、分配、释放）的集合，这是 MLIR 中的标准。 来成功实现所需的语义，而不是禁止跨 `barrier` 的任何代码移动。考虑图 4 中的简单程序。只有当 `codeA` 和 `codeB` 访问相同的内存时，才能观察到同步的影响。此外，如果两者都只读取相同的内存位置，则同步也是不必要的。然后我们可以列举剩下的三种情况：

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

具有 `codeA` 写入行为的屏障将确保情况 1 的正确性：`codeB` 中的负载无法提升到屏障之上，因为它似乎读取了不同的值。对称地，具有 `codeB` 写入行为的屏障将确保情况 2 的正确性。因此，`codeA` 和 `codeB` 写入行为的结合足以防止负载被错误地移动到屏障上。

但是，这并不能阻止写入被移动。例如，在案例 2 中，`codeB` 可以在屏障上方复制，并且它似乎具有相同的最终内存状态，因为屏障之前的无关写入永远不会被读取。因此，我们还将屏障定义为具有 `codeA` 和 `codeB` 的读取行为。

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.

给定一个足够富有表现力的副作用模型，屏障的内存语义可以进一步扩展。屏障对来自不同线程的对同一位置的读取/写入进行排序，自然执行顺序在一个线程内就足够了。因此，屏障不需要捕获地址是线程标识符的注入函数的操作的内存效应。在可能的情况下，将内存访问提升为仿射方言中可用的线性形式，可以实现精确的分析。考虑图 5 中的代码。屏障周围的读写表达式具有仿射访问集

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

Fig. 5. 屏障语义可以细化为访问除当前线程之外的所有线程中其上方/下方的操作访问的内存地址。

```
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)
  ...
}
```

Fig. 6. 围绕屏障分割并行循环的示例。此处，屏障上方的代码（包括 2 次加载和 3 次浮点运算）与屏障之后的代码（包括对 `@use` 的调用和其他运算）放在单独的并行 for 循环中。此转换消除了屏障，同时保留了语义。由于值 `%a`、`%b` 和 `%c` 在屏障之后使用，因此必须格外小心以确保它们可用。此处，最小割算法保留 `%x` 和 `%y`，然后重新计算 `%a`、`%b` 和 `%c`，因为这将导致保留 2 个值而不是 3 个。

$\mathcal{A}_o = \{A(i) : i = tx\}$，其中 tx 是线程 `x` 标识符。屏障具有仿射访问集 $\mathcal{A}_b = \{A(i) : i \neq tx\}$。由于访问的地址集不重叠，$\mathcal{A}_o \cap \mathcal{A}_b = \varnothing$，允许跨屏障的代码移动。从概念上讲，对 `A[threadIdx.x]` 的写入总是发生在同一线程内的读取之前，因此屏障是不必要的。相反，如果对 `A` 的加载和存储偏移了 1，那么屏障就是必要的，因为屏障之后加载的数据将由不同的线程存储。更一般地，当通过收集所有相关操作的内存位置和效果类型来定义给定屏障的内存属性时，只需要包括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。

B. 降低门槛

为了使 GPU 程序能够在 CPU 上运行，我们必须有效地模拟 GPU 程序的同步行为。虽然第 III-A 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 CPU 上实现屏障。

CPU 架构没有线程块的概念，也没有等待这种概念性线程分组的屏障指令。相反，我们使用常规 CPU 线程和工作共享来在它们之间分配线程块循环迭代。从概念上讲，这与 GPU 执行模型不同，在 GPU 执行模型中，许多显式线程各自执行一次迭代。工作共享要求每个线程按顺序执行多个迭代，因此无法在迭代中间同步，只能在循环结束时同步。

为了解决这个问题，我们为 MLIR 表示开发了一种新的障碍消除技术。如第 VII 节所述，过去已经探索了几种方法，包括循环分裂和连续传递。我们的方法是前者的扩展，结合了两种转换风格：并行循环分裂和并行循环交换。

1) 并行循环拆分： 假设一个屏障具有核函数（或者，在我们的表示中，并行 `for` 循环）作为其直接父级。可以通过将屏障周围的循环拆分为两个并行 `for` 循环来消除它，

```
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
  }
}
```

Fig. 7. 左图：共享内存添加，由一个内核调用组成，其中包含一个带有屏障的 for 循环。右图：通过将并行 for 循环与串行 for 循环进行交换，现在将带有屏障的相同代码直接置于并行循环中。

```
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)
```

Fig. 8. 围绕 `while` 循环进行并行交换。由于必须在每个线程上执行 `condition()` 函数调用才能保持正确性，因此使用辅助变量来保存第一个线程上的调用值。

这两个循环分别运行屏障之前和之后的代码。如果屏障之前的代码创建了在其之后使用的 SSA 值，则必须在第二个并行循环中存储或重新计算这些值。我们使用类似于 [29] 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 SSA 值的图表来表示问题。然后，我们将 `barrier` 之前无法重新计算的每个值定义（例如从覆盖的内存中加载）标记为源，将 `barrier` 之后使用的值标记为接收器。通过对该图执行最小分支切割，我们可以得出需要存储的最小数据量。

2) 平行环路交换： 并非所有屏障操作都具有并行 `for` 作为其直接父级，有些屏障操作可能嵌套在其他控制流操作中。我们创建了一个模型来指定哪些指令可以并行运行。除了 `barrier` 之外，我们的表示不需要对程序进行任何特定的排序或并发。因此，添加额外的屏障是合法的（尽管可能会降低并行性）。我们可以使用此属性来实现控制流的屏障降低。

考虑一个控制流构造 `C`，其中包含一个屏障，并嵌套在并行 `for` 中。在 `C` 周围立即添加屏障将导致并行循环立即在 `C` 上方和下方分裂。因此，`C` 上方和下方的操作将分离为它们自己的并行 `for`，而 `C` 将成为中间循环中的唯一操作。然后，我们可以应用以下技术之一将 `C` 与并行 `for` 互换，从而使后者成为 `barrier` 的直接父级。

考虑包含屏障的串行 `for` 循环的情况，图 7。这种模式在 GPU 代码中很常见，例如，用于实现跨线程的减少 [30]。由于 `barrier` 必须等待所有线程，因此每个线程必须执行相同数量的 `barrier`。因此，所有线程的内部循环迭代次数相同，从而允许循环交换。

虽然 `if` 语句可以被视为具有零次或一次迭代的循环，但在必要时直接将其与周围的并行 `for` 交换会更有效。

在 MLIR 中，`for` 循环必须在循环之前计算其迭代次数。`while` 循环支持动态退出条件。例如，考虑图 8 中的代码。为了保持正确性，我们必须在每个线程中执行

condition() 调用，因此直接交换是不合法的。但是，由于 GPU 同步语义，所有线程的迭代次数必须相等。因此，由于辅助变量存储来自一个线程的条件结果，用于决定是否需要另一次迭代，因此可以进行交换。

这说明了在 MLIR/Polygeist 中构建此类系统的优势之一。通过保留程序的高级结构，我们可以使用更有效的模式来消除障碍。

C. 用法

Polygeist 的 CUDA 转译旨在让 Polygeist 易于使用，因为它可以替代现有的 CUDA 编译器（如 clang）。具体来说，Polygeist 扩展了 clang 前端，因此使用与 clang 相同的标志和语法。但是，Polygeist 还引入了几个额外的标志，例如 `-cuda-lower` 用于指定 GPU 到 CPU 的转换，`-cpuify=XX` 用于指定用于生成 CPU 程序的给定方法和并行优化集（参见第 IV 节）。

IV. 并行优化

Polygeist/MLIR 提供的并行和 GPU 程序的高级表示可实现多种优化。这些优化包括适用于任何并行程序的一般优化以及 GPU 到 CPU 转换环境中的特定优化。

A. 障碍消除 & 运动

由于 GPU 风格的屏障必须经过特殊转换才能支持 CPU 架构，因此消除或简化任何屏障都会产生显著效果。此外，即使在 GPU 上运行 GPU 代码，屏障消除也非常有用，因为任何同步都会降低并行性。屏障消除和简化的大部分基础设施直接来自于第 III-A 节中定义的内存行为。给定一个屏障 B，让 $M^{\dagger}_{before}$ 为 B 之前直到另一个屏障或并行区域开始的记忆效应的并集，让 B 之后直到并行区域结束的记忆效应的并集 M_{after}。如果除了读后读 (RAR) 之外，屏障上同一位置没有记忆效应（即 $(M^{\dagger}_{before} \cap M_{after}) \backslash \text{RAR} = \varnothing$），屏障的行为被前一个屏障所包含，可以消除。对称条件 $M_{before} \cap M^{\dagger}_{after} \backslash \text{RAR} = \varnothing$ 表示屏障被后续屏障所包含。可消除屏障的一个具体简单情况是根本没有记忆效应的屏障。

例如，考虑图 9 中的代码，它来自 backprop Rodinia 基准测试 [24]。第一个和最后一个 `__syncthreads` 指令是不必要的。这可以从我们上面基于内存的屏障消除算法中得到证明，如下所示。对于第一个屏障，M_{before}（一直到开始）仅包含对 node 的写入和从 input 的读取。$M^{\dagger}_{after}$（转到第二个 `__syncthreads`）包含对 weights 的写入和从 hidden 的读取。如果给定调用上下文，已知指针不会产生别名，则这些都不会发生冲突。因此，可以安全地消除屏障。

同样的记忆分析也可用于执行屏障运动。只需在屏障要移动到的预定位置放置一个虚拟屏障，并检查之前的记忆分析是否会推断出当前屏障是不必要的，从而允许屏障运动。

B. 跨越屏障的内存到寄存器提升

根据 barrier 的内存行为定义其语义的目标之一是使内存优化能够在包含屏障的代码中正确有效地运行。如第 III-A 节所述，屏障具有其上方和下方代码的内存行为，但当前线程的访问除外。这个漏洞很重要，因为它使

```
__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];
}
```

Fig. 9. 来自 Rodinia 反向传播测试的一个示例 CUDA 内核包含不必要的同步和不必要的共享内存的使用，而一个寄存器就足够了。

内存到寄存器的提升能够对线程本地内存（如局部变量）进行操作。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 9 中的代码。考虑标记为“不必要的存储 # 1”和“不必要的加载 # 1”的 weights[ty][tx] 的加载和存储，以及两者之间的同步。此时可以加载的唯一值是之前由同一次读取存储的值。此外，因为在其他人从 weights 读取之前同一位置被覆盖，所以一旦加载仅使用包含从 hidden 加载的值的寄存器，就可以安全地消除第一次存储。在内存到寄存器的优化过程中，Polygeist 现在可以成功推导出这种转发属性，因为第 III-A 节中描述的内存属性中的漏洞允许它推断出屏障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可以正确地在屏障代码上运行。

C. 并行循环不变代码移动

传统的循环不变代码移动优化旨在将指令 I 移出串行“for”循环，从而减少执行 I 的次数。如果 I 可能访问内存，或者有其他副作用，那么除了检查 I 的操作数本身是否循环不变之外，编译器还必须检查“for”循环中的其他代码是否与 I 执行的内存访问相冲突。

在目前的编译器中，虽然可以将循环不变代码移动应用于 GPU 内核中的串行 for 循环，但无法将循环不变代码移动应用于内核调用之外的提升指令。这在一定程度上是由于 GPU 内核与调用它们的 CPU 代码保存在不同的模块中，以及对传统编译器中的并行性缺乏理解（参见图 1）。

与直觉相反，即使我们无法将循环不变代码移动应用于等效串行循环，只要语义正确，我们也可以对并行 for 循环应用循环不变代码移动。我们将依赖这样一个事实：只要我们保持屏障所需的顺序，我们的程序语义就允许我们任意交错并行“for”循环的迭代。因此，可以想象以锁步方式运行程序。也就是说，如果并行 for 循环有 10 条指

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

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

Fig. 10. OpenMP 并行区域融合示例，应用于 MLIR。给定两个相邻的 OpenMP 并行区域，每个区域都使用给定的闭包初始化要运行的线程，将两个闭包与线程屏障融合，从而允许线程初始化一次而不是两次。

```
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
}
```

Fig. 11. OpenMP 并行区域提升的示例，应用于 C/C++。这是图 10 中 OpenMP 并行区域合并的扩展，但应用于整个 for 循环。无需为外循环的每个迭代 `i` 创建单独的闭包/线程初始化，只需创建一次闭包/线程初始化即可。

令，则每个线程都会在任何线程执行指令 2 之前执行指令 1，依此类推。因此，只要指令的操作数不变并且并行 for 循环中没有先前的指令与 `I` 冲突，就可以合法地提升该指令。换句话说，无需检查 `I` 是否与并行 for 循环中的任何后续指令冲突即可启用提升。

D. 块并行优化

OpenMP 是我们在 CPU 上并行执行的主要目标。它将并行 “for” 循环实现为两个结构。首先，将循环概括为一个函数，每个线程调用一次，代表 OpenMP 的 “并行” 结构。然后，在概括的函数中，迭代空间分布在线程之间，代表 OpenMP 的 “工作共享循环” 结构。OpenMP 还有一个 “屏障” 结构，但其语义与 GPU 屏障不同。

当连续执行多个并行循环时（例如，在从第 III-B 节降低屏障之后），可以通过融合相邻的 OpenMP “并行” 构造 [31] 而不融合工作共享循环（参见图 10），从而不取消屏障降低，从而减少线程管理的开销。并行区域融合可以扩展到各种构造，例如将 OpenMP 并行区域移到图 11 中周围的 “for” 之外。这会调用一次线程初始化，而不是 N 次。将其普遍应用于控制流构造，使通过在块上执行并行循环裂变生成的所有并行 for 循环能够融合其 OpenMP 并行（但不是工作共享循环）。

由于 GPU 程序往往以高并行性为前提编写，因此不同块提供的并行性可能已经使可用内核的数量饱和。如果不使用共享内存，块和线程并行性可以合并为单个 OpenMP 并行 `for`，这将均匀划分单个并行区域中的总迭代空间。但是，如果有共享内存，我们的工具将生成嵌套并行区域来表示共享内存分配。在这种情况下，嵌套 OpenMP 并行区域的额外开销可能超过潜在的增加的并行性。此外，并行化内循环可能会导致不良的内存效应，例如错误共享，从而进一步降低性能 [32], [33]。因此，我们还支持对任何嵌套 OpenMP 并行区域的序列化进行优化。执行这样的序列化可以利用内存局部性来提高性能。

V. MocCUDA：集成到 PyTorch

A. 瞄准仅使用 CPU 的超级计算机

我们在 MLIR 中对 GPU 执行模型进行建模的工作目标是能够在只有 CPU 的超级计算机上执行最初为 GPU 编写的高性能代码，特别是在配备 A64FX 处理器的 Fugaku 机器上 [1]。作为一个主要示例，请考虑尚未成功移植到 A64FX 架构的 PyTorch [2]。PyTorch 使用其 “本机” 后端支持 CPU 执行，这仅为许多内核提供了简单且性能较差的实现，例如，没有针对 2D 卷积进行内存优化的 6 路嵌套循环。在 Intel CPU 上，计算密集型内核的执行委托给 oneDNN 库 [3]，该库在 Arm CPU 上的性能较差，因为它是为没有 A64FX 上可用的高带宽内存的普通 CPU 量身定制的。Fujitsu 的 oneDNN 分支 [4]（我们也在第 VI-C 节中与其进行了比较）需要手动调整才能提高性能，并且仍然无法与 GPU 进行普遍竞争。有趣的是，具有高带宽内存的 CPU 可以从类似于 GPU 的计算组织中受益。因此，我们设计了 “MocCUDA” 来模拟支持 Pytorch 的 GPU，方法是使用前面几节中描述的机制将 CUDA 内核转换为支持 OpenMP 的并行 CPU 内核。

B. MocCUDA 架构与 Polygeist 集成

我们实现了一个兼容层 MocCUDA，以便在 CPU 上透明地执行 PyTorch GPU 后端，目的是避免任何手动重新设计和与现有库的互操作性。虽然 PyTorch 确实有自己的 CPU 后端，但其设计与运行 CPU 的大规模并行计算机兼容性较差。特别是，性能问题是由同步内核执行、不匹配的内存布局和低速内存假设引起的。

考虑到该平台与 A64FX 相对相似，我们决定基于 PyTorch 的 GPU 后端设计 MocCUDA，而不是尝试具有挑战性且耗时的重新设计 PyTorch CPU 后端。因此，MocCUDA 必须处理和替换四种类型的操作：

1) 调用 CUDA 运行时（CUDART），
2) 调用 cuDNN 中的深度学习特定函数，
3) 调用辅助 CUDA 库，以及
4) 在 PyTorch 内部执行 CUDA 内核。

只有后者可以编译，因为其余函数以二进制库的形式分发。

为了尽量缩小原型的工作范围，我们分析了 PyTorch 与 CUDA 后端的所有交互，以识别 (1)-(4) 这个众所周知的深度学习问题，即 ResNet-50 [25] 神经网络。

从分析来看，PyTorch 与 CUDART 的交互主要限于识别已安装 GPU 的属性、内存管理以及 CUDA 流的管理和同步。对于原型，我们限制自己为每个 NUMA 节点模拟一个 GPU，并且仅管理显式内存分配和数据传输。为了让 PyTorch 能够访问 GPU 设备属性，我们将真实 GPU Nvidia GeForce RTX 2080 Ti 的数据转储到一个文件中，该文件稍后将在没有 GPU 的系统上由 MocCUDA 使用。我们通过 Apple 的 Grand Central Dispatch (GCD) [34] 模拟 CUDA 流，这使得模拟的 GPU 操作与 Pytorch 的管理层实现异步性。

ResNet-50 模型仅使用闭源 cuDNN 函数的子集（卷积层、批量规范化层和张量添加的前向和后向传递）。我们重新实现了所有必要的 cuDNN 函数变体，重点关注 OpenMP 并行化和 HBM 友好的 Im2Col 加 GEMM 卷积以及对 NCHW 布局（批次 N、通道 C、高度 H、宽度 W）的支持。此外，我们确定了对 cuBLAS 库的调用，并

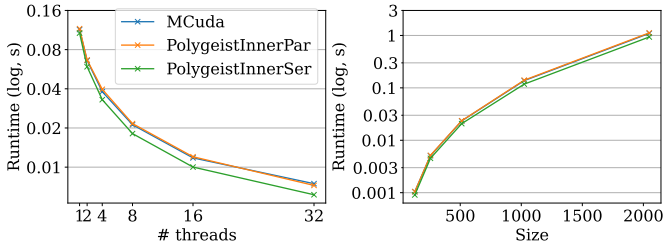

Fig. 12. PolygeistInnerPar has a similar performance to MCUDA, while PolygeistInnerSer outperforms MCUDA. PolygeistInnerSer disables inner loop parallelization similaly to MCUDA, whereas PolygeistInnerPar keeps both the blocks and threads parallel. Left: Average runtime as a function of thread count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

实现了包装函数来拦截这些调用并将它们分派到为 CPU 设计的 BLAS 库，例如 MKL、OpenBLAS 或富士通的 SSL2 [35] for A64FX。对于链接到 PyTorch 的其他库（例如 cuFFT），也可以采用类似的方法，但对于 ResNet-50 来说不是必需的。

除了 CUDART 和其他库调用之外，我们还观察了自定义内核，即在 PyTorch 中用 CUDA 实现的内核，而不是由（二进制）库提供的内核，并确定了调用它们的高级函数。ResNet-50 所需的函数包括步幅张量核（加法、乘法等）、聚合操作（如“Softmax”）等。负对数似然损失核使用 CUDA 的 `__syncthreads()` 障碍。将这些基于 CUDA 的 PyTorch 函数移植到 CPU 需要进行劳动密集型的重新设计，而 Polygeist 可以执行自动翻译。出于演示目的，我们使用 Polygeist 自动将 `ClassNLLCriterion_updateOutput`，使用 `__syncthreads()` 和 `ClassNLLCriterion_updateGradInput` 函数以及从这些函数中传递调用的函数，从 CUDA 转换为 OpenMP，并将结果集成到 MocCUDA 中。

我们将所有上面描述的包装器和重新实现组合到 MocCUDA 中，它可以与 `LD_PRELOAD` 一起使用来拦截 CUDART/cuDNN 调用，以使用 CPU 上的 CUDA 后端训练 ResNet-50。

VI. 评估

我们在两个著名的 GPU 基准测试套件上展示了我们的方法的优势和适用性：GPU Rodinia 基准测试套件的子集 [24] 和 Resnet-50 神经网络的 PyTorch 实现。选择这些基准测试的目的是 1) 在具有手工编码 CPU 版本的基准测试套件 (Rodinia) 上对我们的 GPU 到 CPU 编译进行粗略的性能比较；2) 展示我们的系统成功地端到端集成到超级计算机 Fugaku 上一个有用且真实的应用程序 (PyTorch Resnet-50) 中，该超级计算机没有任何 GPU。此外，我们将我们的方法与现有的 MCUDA [11] 工具在 CUDA 矩阵乘法上的性能进行了比较。

对于 Rodinia，我们将翻译后的 CUDA 代码与基准测试的 OpenMP 版本（如果存在）以及在 GPU 上的运行进行比较。对于 PyTorch Resnet-50，我们与“本机”和 oneDNN 后端进行比较。

Polygeist [1] 是在提交 `89525cbf` 时针对 LLVM 15 编译的。对于 PyTorch Resnet-50，我们使用 Nvidia 的 CUDA 11.6 SDK for Arm [2]、LLVM 13 和 Fujitsu 的 SSL2 v1.2.34 库编译 Pytorch v1.4.0。对于基线 PyTorch 测量，我们使用 Fujitsu 预装的 PyTorch (v1.5.0)。

我们在运行 Ubuntu 20.04 的 AWS c6i.metal 实例（双插槽 Intel Xeon Platinum 8375C CPU，频率 2.9 GHz，每个 32 个内核，256 GB RAM）上评估 Rodinia 和矩阵乘法测试。测量是在第一个插槽上进行的，禁用了超线程和睿频加速。每个数字都是至少 5 次重复的中位数。

A. 与 *MCUDA* 的比较

首先，我们将我们的方法与 MCUDA [11] 中的先前工作进行比较。MCUDA 是一种 AST 级工具，它生成新的 CPU C/C++ 作为输出，并使用类似的循环裂变技术来处理同步。作为源到源工具，它仅处理输入语言的一小部分，因此无法处理 Rodinia 程序。相反，我们在图 12 中比较了矩阵乘法内核在一系列线程（1-24）和矩阵大小（128×128 - 2048×2048）中的运行时间。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. 用例 *1*：*Rodinia* 基准

我们对 Polygeist 目前支持的 14 个基准测试子集进行了基准测试，并且运行时间不长。[3] 我们通过比较使用 `nvcc` 编译并在 GPU 上执行的程序输出，以及通过我们的流程编译并在 CPU 上执行的程序输出来验证流程的正确性。我们在内核和/或包含内核的代码的计算部分插入了计时测量，在某些情况下每个基准测试有多个。在可能的情况下，我们对相同基准测试的 OpenMP 版本的等效部分进行计时。[4]

在图 13（右）中，我们将为 CPU 编译的 Rodinia CUDA 基准与 Rodinia OpenMP 版本的基准进行了比较。虽然基准之间存在一些差异，但总体而言，我们的方法与基准的手工编码版本相当，甚至在启用内部序列化优化时，几何

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

[2] 即使我们将在无 GPU 的系统上运行 PyTorch，我们也必须在支持 CUDA 的系统上编译 PyTorch，以确保发出正确的代码。我们还阻止了三个 Pytorch 函数的内联。

[3] `hybridsort`、`kmeans`、`leukocyte`、`mummergpu`、`huffman` 和 `heartwall` 使用 Polygeist 中不受支持的 C++ 或 CUDA 功能（虚拟函数和纹理内存）。`lavaMD` 和 `dwt2d` 基准测试使用格式错误的 C++，由于从未初始化的内存读取而导致行为未定义（GPU 驱动程序将共享内存初始化为零，但这不是必需的）。`nn` 和 `gaussian` 测试在 ≤ 0.005 秒内运行。

[4] 对于 `nn`，由于其运行时间简单而被排除在外，两个版本在数据加载方面有所不同（在 OpenMP 代码中，运行时动态加载，在 CUDA 代码中预加载所有数据），因此也应该出于这个原因将其排除在外。

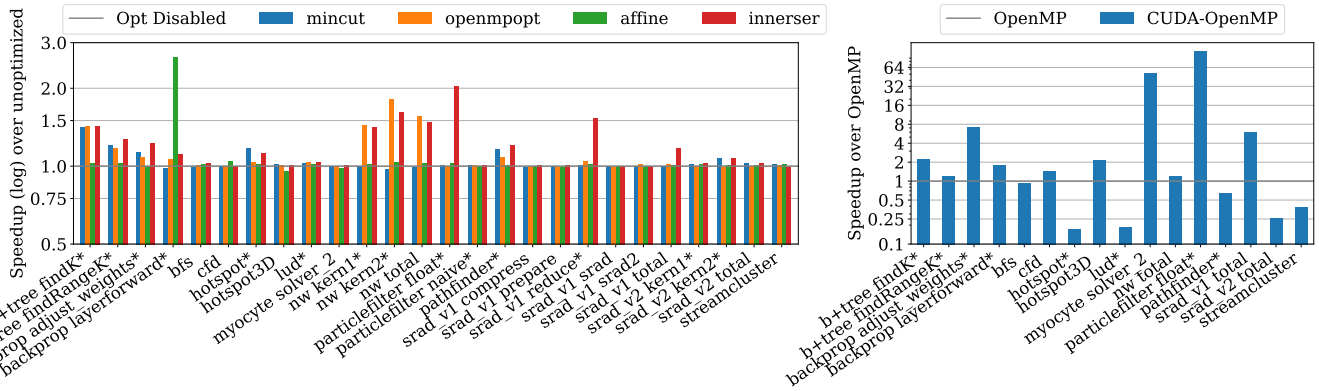

Fig. 13. 左图：应用各种并行和/或 CPU 优化后内核的相对加速比（越高越好）。右图：编译为 OpenMP 后 Rodinia CUDA 代码的加速比与原生 Rodinia OpenMP 代码（可用时）的加速比。包含障碍的基准测试以星号标记。

平均性能提高了 76 %。如果没有内部序列化，我们仍然可以看到几何平均加速 43.7 %。一些基准（例如 `hotspot` 和 `pathfinder`）采用了模板计算优化技术，这些技术会在线程之间重复计算，以减少同步开销并更好地利用 GPU 中可用的并行性。这使得 CUDA 代码比 OpenMP 版本复杂得多，从而导致它们的性能更差。CUDA-OpenMP 版本的 `lud` 和 `srad_v2` 测试速度较慢，因为程序需要执行额外的工作来在共享内存中缓存数据。CUDA-OpenMP 版本的 `particlefilter` 相对加速，因为纯 OpenMP 版本的代码通过单独的 OpenMP “并行 for” 循环实现了所需的依赖结构，而在 CUDA 代码中，这是通过 `__syncthreads` 实现的。Polygeist 能够成功优化屏障周围的代码，从而实现加速。`backprop` 的加速部分归功于并行优化（见图 13（左）），部分归功于 CUDA 代码是用线性数组实现的（这是 CUDA 的要求），而不是 OpenMP 代码中使用的双指针。`myocyte` 和 `sradv1` 都实现了加速，这都是由于跨并行区域边界的代码优化以及内部序列化。

在图 14 中，我们通过比较转译后的 CUDA 与原生 OpenMP 内核来测试我们方法的扩展属性。转译后的 CUDA 代码通常比原生 OpenMP 版本扩展性好得多。由于大多数 CUDA 程序都是以数千个线程为前提编写的，这表明我们的框架能够在将 GPU 特定的构造重写为与 CPU 兼容的等效构造时保持并行性。在没有内部序列化的 32 个线程上，转译后的 CUDA 代码在所有测试中的几何平均加速比为 16.1×。由于并非所有测试都有基准的 OpenMP 版本，如果我们只考虑存在 OpenMP 版本的 CUDA 代码，我们会发现几何平均加速比为 14.0×，而 OpenMP 的加速比仅为 7.1×。序列化内部循环会稍微降低可扩展性，但仍可提高 OpenMP 上的可扩展性，在启用内部序列化的所有测试中发现几何平均加速 14.9×，在启用 OpenMP 版本的代码上发现几何平均加速 12.5×。

我们执行消融分析来研究各个优化如何影响性能。图 13（左）中的 “mincut” 系列显示了我们的方法的性能测量，其中采用了第 III-B1 节中概述的优化，以减少跨障碍保留的数据量。这仅适用于包含障碍（图中用星号标记）的基准测试。在适用的基准测试中，mincut 提供了 4.1 % 几何平均加速。图 13（左）中的 “openmpopt” 系列展示了 OpenMP 区域合并和类似优化的影响，并产生了 8.9 % 几何平均加速。图 13（左）中的 “affine” 系列显示了将控制流提升到其仿射变体并启用简单循环优化（如循环展开）的结果。虽然这会产生全面 4.6 % 的几何平均加速，但它会导致反向传播层前向测试的 2.6× 加速，因为它会导致包含同步的循环被完全展开并减少到 `if` 语句。

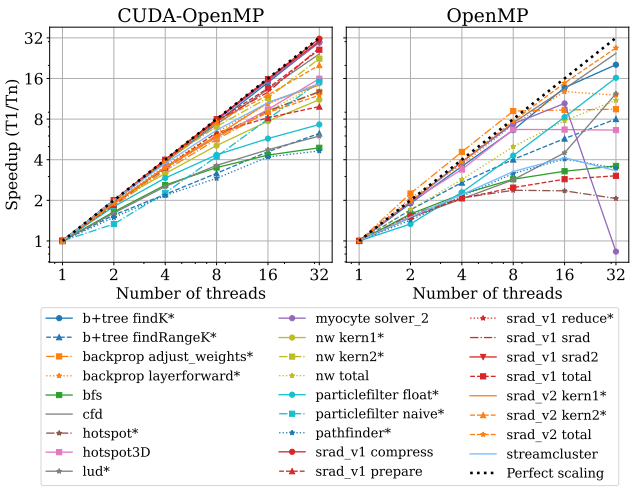

Fig. 14. CUDA Rodinia 内核在 CPU 上使用 OpenMP 运行时的扩展行为，以及使用 32 个线程的 OpenMP Rodinia 内核（如有）。并非所有 Rodinia CUDA 内核都有 OpenMP 版本。

C. 用例 2：*Pytorch/Resnet50* 测试

为了评估 PyTorch Resnet-50，我们在 Fugaku FX1000 超级计算机的一个 TofuD 单元上执行了完整的节点并行训练运行，并与原生 PyTorch CPU 后端和优化的 oneDNN 后端（如果可用）进行了比较。

我们以数据并行的方式在 224 × 224 ImageNet 上对 Resnet-50 进行了多次前向和反向传播。我们使用 Horovod 的合成基准测试脚本（针对 Resnet-50 神经网络模型配置）[36]。我们使用 CUDA SDK、LLVM 和富士通的 MPI 库构建 Horovod v0.19.5，以在 Pytorch 上实现多节点分布式深度学习。我们为每个 A64FX 核心内存组 (CMG) 分配一个 MPI 等级，每个节点模拟最多 4 个 GPU，并将测试从一个节点（2 个等级）扩展到一个 TofuD 单元（最小 2 × 3 × 2 环面）中的 12 个节点（48 个等级），同时保持 OpenMP 线程数固定为 12，以容纳每个核心一个线程。我们使用 Pytorch v1.4.0 来实现我们的方法，而其他后端则依赖于 Pytorch v1.5.0。使用 Benchmarker [37] 进行性能

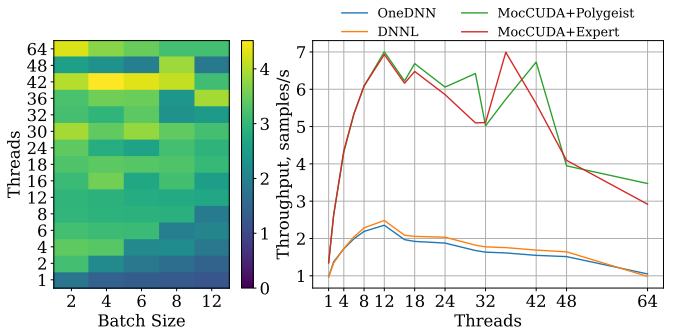

Fig. 15. 在 Fugaku 节点上训练 ResNet50。左图："MocCUDA+Polygeist" 相对于富士通调整的 oneDNN 的相对吞吐量增加热图，值越高越好。右图：不同批次大小的几何平均吞吐量；"MocCUDA+Expert" 使用专家编写的 OpenMP 内核；"MocCUDA+Polygeist" 使用生成的内核。

测量，它通过 torchvision 协调设置神经网络，创建图像，使用 PyTorch 执行层，并返回每秒图像吞吐量指标。我们使用批处理大小 1-64 个线程上 1-12 运行，并在各个时期取平均值。

我们观察到，MocCUDA 在批处理大小和线程数方面系统地优于富士通调整后的 oneDNN，吞吐量提高了 4.5 ×（几何平均值 2.7 ×，最小值 1.2 ×），如图 15 所示。采用我们专家编写的内核的 MocCUDA 与采用 Polygeist 生成的内核的 MocCUDA 相当，如第 V-B 节所述。

这种改进可以通过 PyTorch CPU 设计和第 V-B 节中描述的 oneDNN 性能特征的结合来解释。由于英特尔的 oneDNN [3] 不考虑 A64FX 上可用的 HBM，它使用缓存友好的直接卷积而不是基于 GEMM 的卷积，在 Arm CPU 有 HBM 的情况下效率较低。虽然由富士通调整的 oneDNN 自定义分支 [4] 改进了英特尔 oneDNN 的性能（尽管是 6% 的几何平均值），但仍有性能改进的空间。

这表明我们的方法能够从 CUDA 版本中自动获取深度学习内核（以及潜在的其他应用程序）的高效版本，从而解决高带宽内存的 CPU 缺少或内核效率低下的限制，而无需对应用程序进行逆向或重新设计。

VII. 相关工作

A. GPU 与 CPU 同步

NVIDIA 直接提供了用于调试目的的首批在 CPU 上模拟 GPU 的工具之一，并使用不同的 CPU 线程模拟 GPU 上的每个线程。虽然功能齐全，但可用线程数量的巨大差距导致模拟效率低下。

MCUDA [11] (2008) 对 C GPU 代码进行 AST 转换，以生成新的 C CPU 代码，该代码调用线程独立的并行 `for` 例程。MCUDA 率先使用 "深度裂变" 来处理同步，它在同步点处拆分并行循环和其他构造以消除它们。这种裂变技术也应用于其他工具：Ocelot [12] (2010)，一种二进制翻译工具，可将 PTX 汇编解析为 LLVM 并即时编译内核函数；POCL [38] (2015)，一种用于 OpenCL 的 Clang/LLVM 编译器；COX [13] (2021)，另一种用于 CUDA 翻译的 LLVM 转换过程，它使用裂变，并且特别处理 warp 级原语；甚至这项工作。虽然裂变方法背后的直觉与此处使用的方法类似，但我们在高级编译器中应用裂变，而不是在源或低级 IR 中。如第 III-A 节所示，对结构化程序执行裂变可以实现更高效的代码转换。虽然在源级执行裂变会错失在裂变之前运行优化（例如屏障消除）的机会，而在低级应用裂变需要尝试重建高级结构，但在 MLIR 中操作可以让我们既应用优化又保留高级结构。此外，源级工具往往非常脆弱，因为它们必须重新实现目标语言（例如 C++）的解析和语义，因此只能对输入语言的有限子集进行操作，需要重新设计工作来替换不受支持的构造（如指针算法）。

另一种方法是使用连续传递来处理同步，方法是创建所有同步点的状态机（例如 "微线程"） [39] (2010)。Karrenberg 和 Hack [40] (2012) 提出了一种 LLVM 中的连续传递方法，其中包括一种用于检测和减少控制流图中的发散的算法，随后进行的工作是最小化实时值以减少内存流量 [41]。

VGPU [42] (2021) 与 NVidia 的原始虚拟 GPU 类似，但现在使用 C++ `std::thread` 并使用 `std::atomic_thread_fence` 执行同步。在 LLVM 中作为单个全局变量实现的共享内存通过块数进行扩展。

在低级 LLVM IR 上运行的先前工作需要付出巨大努力来重建高效裂变或连续传递所需的高级构造，例如循环和内核配置。例如，POCL [38] 运行各种规范化和循环转换来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。在源代码/AST 级别（例如 MCUDA）上运行的先前工作，除了仍然需要识别 GPU 级别的概念之外，无法从简化代码的优化中受益，从而简化控制流。

相比之下，通过对 MLIR 的混合抽象进行操作，我们能够同时保留源级结构并执行程序转换，例如循环展开或 LICM 运动，例如可以删除嵌套同步。

B. 并行可移植性/IR、& OpenMP 优化

有几种工具在宿主语言中定义了新的抽象，这些抽象自然适合 CPU 或 GPU 执行。示例包括 C++ 中的 ISPC [43]、RAJA [44]、Kokkos [45] 或 MapCG [46]（仅限于 map-reduce 计算）、Python 中的 Loo.py [47] 和 Julia 中的 KernelAbstractions.jl [48]。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。但是，任何现有代码都必须在上述框架中重写（并且可能会或可能不会与用其他框架或其他语言编写的代码组合）。

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].

众所周知，使用 OpenMP 并行区域扩展是有益的 [31]。Clang/LLVM 可选择以较弱的形式支持转换，即在同一控制级别合并 OpenMP 并行区域 [55]。

C. 障碍

先前的几项工作探索了屏障或同步指令的语义，包括与 GPU 相关的语义。已经完成了验证屏障正确性的工作

[56] 。[57] 探索并通过实验评估了各种 GPU 供应商的前向进展/公平模型。[58] 实现了适用于工作组的 GPU 屏障操作，而不仅仅是适用于工作组内部。[59] 向程序中添加 Java 内存屏障以确保弱一致性和顺序一致性语义。他们发现，如果没有同步和延迟集分析，引入一致性语义平均会减慢 26.5× 速度，而当使用这些分析插入较少的同步时，弱一致性和顺序一致性的速度分别会减慢 10 % 和 26 %。

VIII. 结论

通过扩展 Polygeist/MLIR，我们开发了一个端到端系统，能够表示、优化和转译 CPU 和 GPU 并行程序。能够同时表示和转换不同的并行框架至关重要，因为 HPC 越来越依赖（异构）并行性来处理其工作负载。我们框架的一个关键组成部分是开发高级屏障操作，这是表示 GPU 程序的关键，其语义可以通过其内存行为完全定义。与之前的并行屏障表示不同，我们的语义支持将屏障直接集成到优化中。为了验证我们方法的有效性，我们在商用 CPU 上演示了 GPU 到 CPU 的优化和 Rodinia 基准测试套件子集的转译，并从 PyTorch CUDA 源转译了一个高效的 Resnet-50，以在仅使用 CPU 的超级计算机 Fugaku 上运行。虽然由于 CPU 和 GPU 之间的实现差异，存在个案差异，但 Rodinia 基准测试套件实现了转译的 GPU 代码比手写 OpenMP 版本快 76 % 的几何平均加速。同样，可以观察到 CUDA PyTorch 内核比原生 PyTorch CPU 后端快 $\approx 2\times$

目前，编译后的 GPU 代码在 CPU 上运行时保持相同的调度，除了最内层循环序列化可以提高性能。未来富有成效的工作途径可能是对代码进行高级重新调度，以更好地利用 CPU 风格的内存层次结构。

IX.

致谢 感谢麻省理工学院的 Valentin Churavy 和谷歌的 Albert Cohen 对 MLIR 中的转换进行的深入讨论。感谢 Douglas Kogut、Jiahao Li 和 Bojan Serafimov 对编译器中的并行优化进行的深入讨论。William S. Moses 的部分资金来自美国能源部计算科学研究生奖学金 DE-SC0019323，部分资金来自洛斯阿拉莫斯国家实验室拨款 531711，部分资金来自美国空军研究实验室和美国空军人工智能加速器，合作协议编号为 FA8750-19-2-1000。Johannes Doerfert 的部分资金来自美国能源部、科学办公室、高级科学计算研究计划下应用数学活动，合同编号为 DE-AC02-06CH11357；部分由百亿亿次级计算项目 (17-SC-20-SC) 发起，该项目由美国能源部的两个组织 (科学办公室和国家核安全局) 共同发起，负责规划和准备一个功能强大的百亿亿次级生态系统，包括软件、应用程序、硬件、高级系统工程和早期测试平台，以支持国家百亿亿次级计算的迫切需求。这项工作得到了日本学术振兴会 KAKENHI 资助号 19H04119 和日本新能源和工业技术开发组织 (NEDO) 的部分支持。本文中的观点和结论均为作者的观点和结论，不应被解释为代表美国空军或美国政府的官方政策，无论是明示的还是暗示的。美国政府有权为政府目的复制和分发重印本，而不受本文任何版权声明的限制。

References

[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 intermedi-

ate 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 dataflow 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.