

清 华 大 学

综 合 论 文 训 练

题目: PLANT: 基于多面体模型的张
量编译器

系 别: 计算机科学与技术系

专 业: 计算机科学与技术

姓 名: 李晨昊

指导教师: 翟季冬 副教授

2022 年 1 月 2 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内 容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签 名： 李晨昊 导师签名： 翟季冬 日 期 2021年6月15日

中文摘要

张量计算在诸多领域占据重要地位，但随着体系结构和高层框架复杂性的增加，为异构平台生成高效的张量代码面临巨大的工程挑战。张量编译器试图通过编译优化技术自动化代码生成过程，然而现有的张量编译器技术方向通常存在自动化程度低，程序表达能力有限，生成的代码性能不理想，调度指令覆盖面有限，用户难以介入编译优化过程等缺点，在学术研究和工业应用方面都表现出明显的不足。多面体模型是编译优化领域的重要技术，近年来许多将它应用到张量编译器领域的尝试，但也未能达到足够好的效果。

本文介绍了基于多面体模型的张量编译器 PLANT。借助多面体模型，PLANT 能够表达复杂的程序语义，为用户提供精细的调度指令，在中间表示上分层应用调度，便于用户显式地管理体系结构的复杂性，从而为 CPU 和 GPU 后端生成高效的代码。PLANT 使用基于统计数据性能模型来实现自动调度，在用户指定的调度参数模板上进行高效的自动调优。PLANT 支持远程执行和 Python 接口，满足用户的开发需求。在算子性能和神经网络推理性能测试中，PLANT 均优于包括手工调优的算子库，其他最先进的张量编译器在内的常见参考平台。

关键词：张量编译器；多面体模型；机器学习；自动调度；代码优化

ABSTRACT

Tensor computation plays a predominant role in many fields, but as the complexity of hardware architecture and high-level framework increases, generating efficient tensor code for heterogeneous platforms is faced with tremendous engineering challenges. The tensor compilers attempt to automate the code generation through compiler optimization techniques. However, existing tensor compilers usually have a low degree of automation, limited ability in expressing programs, unsatisfactory performance of generated code, limited coverage of scheduling commands, and it is difficult for users to intervene. Thus they have shown obvious deficiencies in both academic research and industrial application. Polyhedral model is an important technique in the field of compiler optimization. In recent years, researchers have been attempting to apply it to tensor compilers, but yet to achieve sufficient results.

This paper introduces the tensor compiler PLANT based on the polyhedral model. With the help of the polyhedral model, PLANT can express complex program semantics, provide users with fine-grained scheduling commands, and apply hierarchical schedules on the intermediate representation, which is convenient for users to explicitly manage the complexity of the hardware architecture, thus generating efficient code for both CPU and GPU. PLANT uses a performance model based on statistical data to implement auto-scheduling, and performs efficient automatic tuning on user-defined schedule parameter templates. PLANT supports remote execution and Python interface to meet the development needs of users. In the operator performance and neural network inference performance tests, PLANT is superior to common reference platforms such as manually tuned operator libraries and other state-of-the-art tensor compilers.

Keywords: Tensor Compiler; Polyhedral Model; Machine Learning; Auto-Scheduling; Code Optimization

目 录

第 1 章 引言	1
1.1 张量计算背景	1
1.2 张量编译器现状	2
1.3 主要贡献	3
第 2 章 中间表示	5
2.1 中间表示	5
2.2 表达式 DSL	7
第 3 章 调度指令	9
3.1 分层设计	9
3.2 调度指令简介	9
3.3 多面体模型	12
3.4 对非仿射程序的支持	15
第 4 章 自动调度	19
4.1 问题描述	19
4.2 性能模型	20
4.3 搜索系统	21
第 5 章 系统实现和优化	23
5.1 ISL 接口	23
5.2 代码生成	24
5.3 运行时库	26
5.4 性能优化	26
5.5 远程执行	29
5.6 Python 接口	29
第 6 章 性能评测	31
6.1 算子性能	31
6.2 神经网络推理性能	33

6.3 自动调度.....	34
第 7 章 总结.....	37
插图索引.....	38
表格索引.....	39
参考文献.....	40
致 谢.....	43
声 明.....	45
附录 A 外文资料的书面翻译.....	47

第 1 章 引言

1.1 张量计算背景

计算机科学领域，张量通常表示数据的多维数组 (Multidimensional Array) [1] [2]。数据分布可以是稠密的或稀疏的。张量计算是多维数组上的算术运算，它通常由深层嵌套的循环中的计算密集型内核组成。张量计算在深度学习，图像处理，科学计算等领域占据重要地位 [3]，因此优化计算性能至关重要。

以深度学习应用为例，如图 1.1 所示，随着硬件（例如 CPU，GPU，FPGA [4]，ASIC [5]），高层框架（例如 TensorFlow [1]，PyTorch [6]，Caffe [7]），数据类型，算子等各方面的多样性的增加，为深度学习应用中的张量算子生成高效的代码变得越来越困难。

传统上这通过调用硬件制造商提供的人工调优的算子库（如 Intel MKL [8]，cuBLAS [9] 和 cuDNN [10]）来解决。为获得最佳性能，通常需要程序员手动为张量算子实现复杂的代码和数据布局转换，进行精细的硬件资源和内存层次控制。人工调优过程过于专业和不透明，工程量巨大，无法轻松地跨硬件移植，因此仅靠人力难以及时地为不断出现的新硬件和新算子提供高效实现。



图 1.1 多种多样的深度学习框架和硬件

此外，对神经网络的图优化 [11] 可能产生各种融合的新算子以减小运行时开销，需要支持的算子种类无法提前预知，因此无法通过预先编写好的一组算子直接实现。这种情况下高层框架可能不得不在：(1) 放弃图优化，从而避免产生算子库不支持的新算子，或者 (2) 使用未经优化的新算子实现中做出选择，二者都将

导致次优的性能。

1.2 张量编译器现状

为了解决张量计算优化中遇到的困难，目前主流的方向是通过编译优化技术进行一定程度的自动化，即借助张量编译器进行全自动或半自动的代码生成。通常情况下，即使不能做到全自动，张量编译器也能极大减少人工编写底层代码的开销。张量编译器编译优化的核心是循环变换，数据布局变换等高层次的代码变换，传统编译器通常难以进行这些变换。

多面体编译技术 [12] [13] 通过多面体模型来实现上述代码变换。多面体模型将循环迭代范围抽象成多面体，即一组仿射不等式定义的集合；将内存访问，循环变换等编译优化过程中的重要概念抽象成仿射变换，在此基础上使用线性规划等技术，手动或自动地实现代码变换。多面体模型的具体定义和技术细节在章节 3.3 中描述。

目前张量编译器主要有以下三个技术路线：

1. 人工调度的非多面体编译器

例如 Halide [14]，TVM [15]。这是目前实际上最流行的技术路线，它达到了较好的综合性能，而且有良好的生态链和社区支持。

人工调度的非多面体编译器的不足之处在于，它要求用户提供调度指令来进行代码变换。编写合适的调度指令要求用户充分理解硬件体系结构，例如，用户需要对何时进行循环分块，最优分块大小等指标做出合适的预测。然而，也存在 AutoTVM [16]，Anso [17]，FlexTensor [3] 等拓展工作，它们使用基于统计数据性能模型来实现自动调度，从而可以实现一定程度的自动化，在工程实践中有较好的使用体验。

另一个本质的不足是非多面体编译器中程序的表达能力有限。例如，TVM 采用基于区间的程序表示，难以精确表达和操作非矩形的迭代空间，非完美嵌套的循环，存在循环依赖的数据流图等复杂程序。而且，基于区间的程序表示难以进行精确的依赖关系分析，往往依靠保守的规则来确定调度是否合法，这可能导致编译器无法进行某些原本合法的优化。

2. 自动调度的多面体编译器

例如 PLuTo [18]，PPCG [19]，Polly [20]。它们基于 PLuTo [18] 算法及其衍生算法进行自动调度，能达到较高的自动化程度。例如 PPCG 直接对输入的 C 代码

进行循环转换，输出并行的 C 代码或 CUDA 代码，不需要任何用户干预，这极大地提高了代码编写效率。

但自动调度的多面体编译器也有严重的缺点，导致它几乎无法直接生成高效的代码，也就无法直接胜任工业级的代码生成任务。第一，PLuTo 自动调度算法缺少精确的代价模型来判断哪一种转换性能更好，往往无法选择最优的调度，因此生成的代码性能不理想。

原始的 PLuTo 算法试图在最大化并行度的同时最小化生产者和使用者的距离，从而使计算出的数据尽快被使用，这符合时间局部性原理。但它对数据的空间局部性考虑不足，也没有考虑冗余计算，代码的复杂控制流等对性能有重要影响的因素。从结果来看，在 PLuTo, PPCG, Polly 等编译器汇报的性能测试结果中，在多数常见算子上它们确实能显著超出传统编译器，但相比成熟的算子库还有明显差距。

第二，虽然多面体模型可以准确地描述复杂的循环和数据布局变换，但它不能自然地表达对硬件的精细控制，导致编译器的调度指令覆盖面有限，不能充分利用体系结构相关的硬件资源。此外，全自动化的调度过程使得调度算法探索的空间十分有限，而且用户难以理解和介入代码生成的过程，用领域相关的信息指导代码优化。

最后，因为调度算法基于复杂的整数线性规划，所以自动调度的多面体编译器的编译速度也较低。这在一定程度上也限制了它探索不同调度的能力，因为在有限的时间内只能生成数目较少的程序。

3. 人工调度的多面体编译器：

例如 CHiLL [21], AlphaZ [22], TIRAMISU [23]。前两种技术路线的部分缺点在这一种上也存在。与人工调度的非多面体编译器类似，它也需要用户提供调度指令，并且不像 TVM 有类似 AutoTVM, AnsoR 的拓展来实现自动化，目前它缺少自动调度相关的工作。其次，与自动调度的多面体编译器类似，它支持的调度指令覆盖面也十分有限。

1.3 主要贡献

本文介绍了 PLANT，名称来源于 PoLyhedral bAsed teNsor opTimizer 的缩写，即论文名称“基于多面体模型的张量编译器”的英文翻译。它结合了上述的技术路线一和三，主体是人工调度的多面体编译器，同时使用了非多面体编译器中的

基于统计数据的性能模型来实现自动调度，达到一定程度的自动化。

相比于已有的多面体编译器，PLANT 为用户提供丰富的调度指令，允许用户精确地控制高性能体系结构的计算和内存资源。它接受纯算法和一组调度指令作为输入，在程序的中间表示上应用调度后，为目标体系结构生成高度优化的代码。同时，借助自动调度功能，用户不必给出完整的调度，其中可以包含可调优的参数，极大减少了工程负担。

PLANT 主要做出如下贡献：

- 实现了基于多面体模型的编译器，为用户提供丰富的调度指令，在程序的中间表示上分层应用调度，从而为 CPU 和 GPU 生成高效的代码（章节 2，章节 3）。
- 使用了基于统计数据的性能模型来实现自动调度，在用户指定的搜索空间中进行高效的搜索，找出最小化运行耗时的调度（章节 4）。
- 对代码生成和运行时进行了大量优化。提供了远程执行功能，便于用户在资源受限的平台上调优和部署；提供了 Python 接口，便于用户快速调试和迭代（章节 5）。
- 在重要的深度学习算子上评估了性能，性能与 Intel MKL, cuBLAS, cuDNN 相当或更高；使用 ResNet [24] 评测了神经网络端到端的推理性能，相比 PyTorch 有至多 4.95 倍的性能提升（章节 6）。

第 2 章 中间表示

PLANT 是嵌在 Rust 中的领域专用语言 (Domain Specific Language, DSL)。用户 (即程序员) 通过 Rust 接口编写体系结构无关的算法, 以及一组指导代码生成和优化的体系结构相关的调度指令。

接受算法和调度指令作为输入后, PLANT 在内部构造程序的中间表示 (Intermediate Representation, IR), 在中间表示上应用循环和内存相关的调度, 最终生成目标平台的后端代码。PLANT 支持 CPU 和 GPU 两个后端, 为 CPU 后端生成 C++ 代码, 为 GPU 后端生成 CUDA 代码。

2.1 中间表示

以下介绍 PLANT 的中间表示中的几个重要概念。

2.1.1 计算

PLANT 中用计算 (Computation, 缩写为 Comp) 表示循环中的一条语句。包围语句的控制流仅限于步长为 1, 上下界确定的 for 循环和条件表达式, 无法表示更一般的循环, break 或 goto 等语义。单个计算只能是完美嵌套的循环, 但多个计算组合起来可以构成非完美嵌套的循环。

张量计算通常是深层嵌套的循环中的计算密集型内核, 因此这样受限的表达能力在 PLANT 的应用场景中通常已经足够。通过限制中间表示的表达能力, 可以简化分析程序和应用调度的过程。

计算主要由循环迭代空间, 待计算的表达式和计算结果存储的内存位置定义。其中循环迭代空间和存储的内存位置使用多面体模型定义, 存储的内存位置是可选的, 具体细节在章节 3.3 中描述。

2.1.2 表达式

PLANT 中用表达式 (Expression, 缩写为 Expr) 表示计算的结果。支持的表达式种类如 2.1 所示。

以下列举其中部分表达式详细介绍:

- Param(name): 对应 ISL [25] 中的参数 (Parameters), 用参数名字表示, 详细概念和用法参考章节 3.4。

指令	描述
Val	常量值
Iter	循环变量
Param	参数变量
Cast	类型转换
Unary	单目表达式
Binary	双目表达式
Select	条件表达式
Call	函数调用
Access	访问计算
Load	访问内存
Memcpy	内存拷贝
Alloc	内存申请
Free	内存释放
Ramp	取向量内存
Verbatim	原样输出

表 2.1 PLANT 支持的表达式

- `Access(c, idx)` : 访问计算 `c` 的下标列表 `idx` 位置的计算结果，列表长度和迭代空间维度数相同，且下标必须由仿射表达式组成。

- `Iter(i)` : 对循环变量的引用，其中 `i` 为整数编号，表示循环层数。为了实现简单，PLANT 中的循环变量没有名字，只由循环层数表示。这要求用户记忆调度前后循环层数的变化，可能会造成一定的不便，但不会损害程序表达能力。

- `Memcpy(to, from)` : 从缓冲区 `from` 向缓冲区 `to` 拷贝内存，自动根据 `to` 和 `from` 映射的内存层次调用合适的内存拷贝函数，例如从 CPU 向 GPU 全局内存拷贝数据时调用 `cudaMemcpy(..., cudaMemcpyHostToDevice)`。`Alloc` 和 `Free` 也具有类似的性质。

- `Ramp(stride, lanes, load)` : 以表达式 `load` 表示的地址为基地址，取元素跨度为 `stride`，元素数目为 `lanes` 的一段向量内存。

向量编程中常见的广播 (**Broadcast**)，分散 (**Scatter**)，读写连续向量内存等语义都可以借助这个表达式来实现。

- `Verbatim(s)` : 存储一段字符串，后续调度和代码生成过程不关心字符

串的内容，在最终的后端代码中直接输出它，这允许用户自由地拓展代码生成。

2.1.3 函数和缓冲区

PLANT 中用函数 (Function, 缩写为 Func) 表示一个独立的编译优化单元。函数中包含代码生成的相关参数, 若干个缓冲区, 以及若干个计算。计算在内存中的排列顺序与生成的代码中的执行顺序无关。

PLANT 中用缓冲区 (Buffer, 缩写为 Buf) 表示多维数组。缓冲区分为输入, 输出和临时三种, 前两种作为函数参数, 分别用于读和写; 临时缓冲区限制在函数内部, 它的申请和释放是特殊的计算, 用户可以手动控制, 也可以通过章节 3.2 中的内存调度指令自动控制。缓冲区可以映射到不同的硬件内存层次, 还具有初始值, 对齐字节数等可选属性。

2.2 表达式 DSL

PLANT 借助 Rust 的过程宏^① 定义了一套表达式 DSL, 实现了对条件表达式, 循环等语法结构的“重载”, 允许用户以直观的语法定义一个计算。

过程宏是 Rust 的语言特性, 它是用户编写的函数, 接受 Rust 语法树作为输入, 处理后返回新的语法树。Rust 编译器在预处理阶段调用它, 对语法树进行用户指定的转换后, 以新语法树为输入进行实际的编译过程。过程宏在静态强类型的 Rust 语言中引入了通常仅在动态语言中才能提供的灵活特性。

PLANT 定义了 `x!` (取 Expression 中的 `x` 为缩写) 和 `c!` (取 Computation 中的 `c` 为缩写) 两个过程宏, 分别辅助用户定义表达式和计算。

代码 2.1 用 `x!` 定义了卷积计算中的一个子计算, 对输入数据 `a` 进行补零 (Zero Padding) 后得到中间结果 `a_pad`, 它包含四层循环结构, 计算的表达式包含条件表达式。

```
let a_pad = f.comp("a_pad", x![n, c, h + 2 * pad, w + 2 * pad]
  x!(if i2 >= pad && i2 - pad < h && i3 >= pad && i3 - pad < w
    { a(i0, i1, i2 - pad, i3 - pad) } else { 0 }));
```

代码 2.1 用 `x!` 定义计算

该计算在语义上等效于代码 2.2。它并未指定结果存储的位置, 事实上它可能不会存储到内存, 而是内联 (`inline`) 在其他计算中。

```
for n in 0..batch {
  for c in 0..channel {
```

^① <https://doc.rust-lang.org/reference/procedural-macros.html>

```

for y in 0..h + 2 * pad {
  for x in 0..w + 2 * pad {
    if y >= pad && y - pad < h && x >= pad && x - pad < w
      { a[n][c][y - pad][x - pad] } else { 0 }
    }
  }
}

```

代码 2.2 等价的计算代码

代码 2.3 用 `c!` 定义了同一个计算，与代码 2.2 对比，定义计算的语法和计算的实际语义具有完全类似的结构，即使用户对 PLANT 框架不熟悉也能明白其含义。

```

let a_pad = c!(for n in 0..batch {
  for c in 0..channel {
    for y in 0..h + 2 * pad {
      for x in 0..w + 2 * pad {
        if y >= pad && y - pad < h && x >= pad && x - pad < w
          { a(n, c, y - pad, x - pad) } else { 0 }
        }
      }
    }
  }
});

```

代码 2.3 用 `c!` 定义计算

过程宏展开后的代码通过调用 PLANT 提供的接口函数手动构造迭代范围和表达式，十分繁琐。可见表达式 DSL 极大简化了代码编写，且符合用户对编程语言的习惯。比较两个过程宏的写法，`c!` 更符合直觉，但占用空间较多，因此论文的代码实例主要使用 `x!`。

第 3 章 调度指令

3.1 分层设计

PLANT 借鉴了 TIRAMISU [23] 中多层中间表示的思想，将完整的程序定义分离为独立的三层：

- 算法描述：使用章节 2 中的语法，使用生产者 - 消费者关系定义算法，计算通过 Access 表达式访问彼此，不涉及内存访问 (Load)。
- 计算调度：对计算顺序进行调度，应用循环变换，将计算分配到并行的处理器单元。此时无需考虑具体的内存布局。
- 内存调度：指定计算结果保存的内存位置，控制缓冲区的申请，释放，传输，映射到内存层次结构等。

二者的不同之处在于，TIRAMISU 定义了多层中间表示，编译器需要在不同层级间进行多次转换，而 PLANT 在同一层中间表示上分层应用调度指令。在能够达到近似效果的前提下，PLANT 在实现上更加简洁，编译器自身的运行开销也更低。

分层设计的目的是按照一定的顺序应用调度指令，从而简化编译器的实现。以将缓冲区映射和拷贝到 GPU 共享内存为例，拷贝操作的循环迭代空间，需要拷贝的数据量等都取决于计算的调度方式，例如是否应用了循环分块。如果在应用计算调度之前决定将缓冲区映射到 GPU 共享内存，则后续编译器应用计算调度时必须维护这一内存映射，并且会引入内存相关的依赖，这会带来相当大的实现复杂性。分层设计确保某层中的调度无需担心影响到较早层中的调度，从而降低了这种复杂性。

3.2 调度指令简介

表 3.1 和表 3.2 分别列举了 PLANT 支持的循环和内存调度指令。

以下假定 c 等表示计算， i, j 等表示循环层次， b 等表示缓冲区，列举其中部分指令详细介绍：

- 调度指令支持链式调用，例如 `c.tile(i, j, x, y)` 可以简洁地实现成 `c.split(i, x).split(j+1, y).reorder(i+1, j+1)`。
- `c.skew(i, j, x)`：在循环 i 和 j 上以参数 x 进行倾斜变换。

指令	功能
split	将循环分裂为嵌套的两层
fuse	合并两个相邻的嵌套循环
reorder	调整循环的嵌套顺序
tile	循环分块
skew	循环倾斜
shift	循环索引偏移
after	控制循环体位置
separate	将循环拆分为两个并列的循环
inline	将表达式内联到使用它的计算中
tag	为循环添加标记
apply_sch	底层原语，执行任意仿射变换

表 3.1 循环调度指令

指令	功能
store	控制计算结果的保存位置
cache_identity	自动完成恒等映射的缓存
cache	用户手动控制的缓存
set_loc	将缓冲区映射到内存层次结构
alloc_at	控制缓冲区申请和释放的位置
auto_transfer	控制 GPU 缓冲区自动传输数据

表 3.2 内存调度指令

得益于 PLANT 使用多面体模型表示程序，它可以表达非矩形的迭代空间并应用任意仿射变换。

- `c.tag(i, t)` : 为循环 i 添加标记 t ，从而为它绑定硬件资源，包括循环并行化，向量化，将它映射到 GPU 块或线程等。

- `c.after(c1, i)` : 指定在包围循环 i 的一层循环中， c 在 $c1$ 之后。在部分文献中这个操作被称为循环的裂变 (fission) 或者聚变 (fusion)。

- `c.separate(i, x)` : 将循环 i 的跨度拆分成能被因子 x 整除的部分和不能整除的部分。

在此基础上应用循环分块等调度，可以将计算完整分块与部分分块的代码分开，使得内层循环具有确定的循环范围，有利于向量化，减少 CPU 分支预测错误

和 GPU 线程束分散等，提升执行效率。

应用 `separate` 的例子如代码 3.5 所示。

- `c.apply_sch(s)` : 对 `c` 的循环迭代空间应用仿射变换 `s` 表示的调度，在 PLANT 内部其他高层循环调度指令都是借助它实现的。

如果用户需要目前尚未实现的调度指令，可以通过这个底层接口以相对低的代价自己实现。

- `c.cache_identity(c1, i, loc)` : 在计算 `c` 的循环 `i` 中缓存 `c` 的表达式中对 `c1` 的访问，保存在映射到内存层次 `loc` 的缓冲区中。

`cache_identity` 根据已经应用的调度自动计算复制操作的迭代空间，需要缓存的数据量和下标映射关系，支持非连续的访问。例如，在 `c` 中访问 `c1(i0+0), c1(i0+1), c1(i0+6), c1(i0+7)`，并在第 0 层循环内部缓存它，`cache_identity` 能判断缓存大小与循环变量 `i0` 无关，并计算出缓存大小为 4 而非 8，然后建立原数组下标和缓存数组下标的映射关系，如访问缓存数组的下标 `0, 1, 2, 3` 分别对应于访问 `c1(i0+0), c1(i0+1), c1(i0+6), c1(i0+7)`。

如果缓冲区映射到 GPU 共享内存，`cache_identity` 会自动使用一个块中的 GPU 线程来优化读取，并在适当位置插入必要的同步语句以保护共享内存的读写。

- `c.cache(...)` : 用户手动指定缓存配置，需要传入一系列缓存相关的参数，包括数组大小，目标下标，源下标，使用者的新下标等。

相比于 `cache_identity`，`cache` 是更底层的接口，在 PLANT 内部前者也是通过后者实现的。后者允许用户精细地控制缓存方式，或者应用非恒等映射的缓存策略，例如在缓存的同时进行转置，从而在后续访问中获得更好的数据局部性，或者为缓存数组添加冗余元素以避免访问 GPU 共享内存时发生存储体冲突 (Bank Conflict)。

- `b.set_loc(loc)` : 将缓冲区映射到内存层次结构，`loc` 可以取 CPU 堆上内存，栈上内存，GPU 全局内存，共享内存，局部内存，常量内存。

代码 3.1 按照上述分层设计定义矩阵乘法算子。算法描述部分，定义矩阵乘法的初始化和规约，使用抽象的表达式 $c(i_0, i_1, i_2) = a(i_0, i_2) * b(i_2, i_1) + c(i_0, i_1, i_2 - 1)$ ，不对计算结果存储位置做任何假定。计算调度部分，进行循环分块，循环融合，并将最外层循环并行化。内存调度部分，设置内存映射和数据传输，指定 `c_init` 和 `c`

都存储在缓冲区 buf_c 中，至此矩阵乘法的语义才定义完整。

```

let (n, s, m) = (1024, 1024, 1024);
let f = Func::new("matmul");
// 算法描述
let a = f.buf("a", I32, In, x![n, s]);
let b = f.buf("b", I32, In, x![s, m]);
let c_init = f.comp("C_init", x![n, m], x!(0));
let c = f.comp("C", x![n, m, s], x!(0));
c.set_expr(x!(a(i0, i2) * b(i2, i1) + c(i0, i1, i2 - 1)));
// 循环调度
c_init.tile(0, 1, 32, 32);
c.tile(0, 1, 32, 32);
c.after(c_init, 4);
c.tag(0, Parallel);
// 内存调度
let buf_c = f.buf("c", I32, Out, x![n, m]);
c_init.store(buf_c);
c.store_at(buf_c, x![i0, i1]);
// 定义完成，最终生成代码
f.codegen(&[a, b, buf_c]);

```

代码 3.1 PLANT 定义矩阵乘法

3.3 多面体模型

多面体模型中两个核心概念是整数集合和整数映射。整数集合用于表示循环迭代范围，整数映射用于表示计算调度，访问下标等概念。以下简要描述这两个概念，更多细节和正式定义参考 [26]。PLANT 中的集合和映射使用 ISL (Integer Set Library) [25] 实现，以下也采用 ISL 的表示语法。

整数集合是一组 Presburger 公式 [27] 约束的的整数元组的集合。元组有可选的名字。例如：

$$\{c[i, j] : 0 \leq i \leq 1 \wedge 0 \leq j \leq 2\}$$

它等价于列出集合中的每个元组：

$$\{c[0, 0]; c[0, 1]; c[0, 2]; c[1, 0]; c[1, 1]; c[1, 2]\}$$

整数映射是两个整数集合间的关系，即整数元组序对的集合，序对用箭头隔开，分别为映射的原像和像。两个整数集合中元组的名称可以相同。例如：

$$\{c_1[i, j] \rightarrow c_2[j, i] : 0 \leq i \leq 1 \wedge 0 \leq j \leq 2\}$$

它也等价于列出集合中的每个元组序对：

$$\{c_1[0, 0] \rightarrow c_2[0, 0]; c_1[0, 1] \rightarrow c_2[1, 0]; c_1[0, 2] \rightarrow c_2[2, 0]; \\ c_1[1, 0] \rightarrow c_2[0, 1]; c_1[1, 1] \rightarrow c_2[1, 1]; c_1[1, 2] \rightarrow c_2[2, 1]\}$$

以下将整数集合和整数映射分别简称为集合和映射。以矩阵乘法为例，介绍多面体模型在编译优化过程中的应用。

1. 算法描述。

多面体模型使用集合表示循环迭代范围。代码实际运行时，每次执行到循环的语句，包围它的循环变量的取值形成一个元组，称为这条语句的一个动态实例 (dynamic instance)。所有这些元组组成一个集合，表示了迭代范围。

代码 3.1 中，用户指定循环范围 $x! [1024, 1024, 1024]$ 和计算的表达式 $a(i_0, i_2) * b(i_2, i_1) + c(i_0, i_1, i_2 - 1)$ 。据此构造集合 (3.1) 表示迭代范围。

$$\{c[i_0, i_1, i_2] : 0 \leq i_0 \leq 1023 \wedge 0 \leq i_1 \leq 1023 \wedge 0 \leq i_2 \leq 1023\} \quad (3.1)$$

PLANT 的中间表示实际使用的是式 (3.2)。约定元组下标从 0 开始计数，在式 (3.1) 偶数下标 0, 2, 4, ... 处插入整数值，奇数下标 1, 3, 5, ... 处使用表示语句动态实例的元组。式 (3.2) 中的元组 $c[s_0, i_0, s_1, i_1, s_2, i_2, s_3]$ 对应于动态实例 $c[i_0, i_1, i_2]$ 。

$$\{c[0, i_0, 0, i_1, 0, i_2, 0] : 0 \leq i_0 \leq 1023 \wedge 0 \leq i_1 \leq 1023 \wedge 0 \leq i_2 \leq 1023\} \quad (3.2)$$

偶数下标称为计算的静态维度，奇数下标称为动态维度。静态维度有唯一取值，用于表示多个计算间的执行顺序关系；动态维度在一定范围内取值，对应于程序中的循环层次。

在生成的代码中，一个或多个计算的执行顺序遵循语句的动态实例的字典序。考虑如下两个循环：

$$\{c_1[0, i, 0, j, 0] : 0 \leq i \leq 1 \wedge 0 \leq j \leq 2; c_2[0, i, 1, j, 0] : 0 \leq i \leq 1 \wedge 0 \leq j \leq 2\}$$

将所有元素按字典序排列：

$$\{c_1[0, 0, 0, 0, 0], c_1[0, 0, 0, 1, 0], c_1[0, 0, 0, 2, 0], c_2[0, 0, 1, 0, 0], c_2[0, 0, 1, 1, 0], c_2[0, 0, 1, 2, 0], c_1[0, 1, 0, 0, 0], c_1[0, 1, 0, 1, 0], c_1[0, 1, 0, 2, 0], c_2[0, 1, 1, 0, 0], c_2[0, 1, 1, 1, 0], c_2[0, 1, 1, 2, 0]\}$$

这即是代码实际运行时语句的执行顺序，即先执行 c_1 的 i 取 0, j 取 0, 1, 2 的动态实例，再执行 c_2 的 i 取 0, j 取 0, 1, 2 的动态实例，依次类推。 c_1 和 c_2 的静态维度分别是 $[0, 0, 0]$ 和 $[0, 1, 0]$ ，这在生成的代码中就表现为，在包围第 1 层循环的循环（最外层循环为第 0 层）中， c_2 在 c_1 后执行，如代码 3.2 所示。

```
for i in 0..2 {
  for j in 0..2 { c1 }
  for j in 0..2 { c2 }
}
```

代码 3.2 生成的循环

算法描述这一层不指定多个计算间的执行顺序，且定义顺序对执行顺序没有影响，执行顺序在下一层计算调度指定。

2. 计算调度。执行循环分块指令 `c.tile(0, 1, 32, 32)`，如前所述，它实际上是通过 `c.split(0, 32).split(2, 32).reorder(1, 2)` 实现的。以 `c.split(0, 32)` 为例，构造映射表示循环调度：

$$\{c[i_0, i_1, i_2, i_3, i_4] \rightarrow c[i_0, outer, 0, inner, i_2, i_3, i_4] : outer = \lfloor \frac{i_1}{32} \rfloor \wedge inner = i_1 \bmod 32\}$$

调度的实质就是，在表示循环迭代范围的集合上，应用表示循环调度的映射。应用 `c.split(0, 32)` 后新的循环迭代范围如下：

$$\{c[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0] : 0 \leq i_0 \leq 31 \wedge 0 \leq i_1 \leq 31 \wedge 0 \leq i_2 \leq 1023 \wedge 0 \leq i_3 \leq 1023\}$$

调度过程会维护计算的表达式中对循环变量的使用，从而保持正确的语义，例如 `a(i0, i2)` 部分将被转换为 `a(i0*32+i1, i3)`。

完成循环分块的三个子调度后，新的循环迭代范围如下：

$$\{c[0, i_0, 0, i_1, 0, i_2, 0, i_3, 0, i_4, 0] : 0 \leq i_0 \leq 31 \wedge 0 \leq i_1 \leq 31 \wedge 0 \leq i_2 \leq 31 \wedge 0 \leq i_3 \leq 31 \wedge 0 \leq i_4 \leq 1023\}$$

新的表达式为 `a(i0*32+i2, i4)*b(i4, i1*32+i3)+c(i0*32+i2, i1*32+i3, i4-1)`。

应用 `c.after(c_init, 4)`，实质上是通过修改 `c` 的静态维度，控制在包围第 4 层循环的循环中，`c` 在 `c_init` 之后执行。新的循环迭代范围如下：

$$\{c[0, i_0, 0, i_1, 0, i_2, 0, i_3, 1, i_4, 0] : 0 \leq i_0 \leq 31 \wedge 0 \leq i_1 \leq 31 \wedge 0 \leq i_2 \leq 31 \wedge 0 \leq i_3 \leq 31 \wedge 0 \leq i_4 \leq 1023\}$$

`c.tag(0, Parallel)` 只是简单地给循环添加标记，在代码生成时才考虑标记，以此为依据生成并行化的循环。代码生成中处理标记的具体实现细节参考章节 5.2。

3. 内存调度。`c.store_at(buf_c, x![i0, i1])` 用映射表示计算 `c` 的结果保存的位置：

$$c[i_0, i_1, i_2] \rightarrow buf_c[i_0, i_1]$$

映射的原像是调度前的迭代空间，映射的像是结果保存的缓冲区和下标列

表。c 的最后一个循环维度不出现在下标列表中，因此这一层循环执行规约操作。通过让 `c_init` 和 `c` 的结果保存在同一个缓冲区中，将这两个计算关联了起来，因此 `c_init` 相当于规约的初始化操作。

尽管这个简单程序没有体现出来，但在内存调度这一层还可以进行数据传输，缓存计算等高层的内存相关调度。

3.4 对非仿射程序的支持

如上所述，为了用集合和映射表示程序，要求程序是仿射的，即要求循环迭代范围，内存访问下标等能表示成外层循环变量的仿射表达式。但真实应用并不总能满足这个要求，动态形状 (Dynamic-Shape) 算子，稀疏 (Sparse) 算子等在实际中有广泛应用 [28]，它们的循环迭代范围，内存访问下标等依赖于运行时读取的值，因此不满足这个条件。

PLANT 通过 ISL 提供的参数 (Parameters) 机制来实现对非仿射程序的支持。ISL 允许在集合和映射中定义一组参数，使用参数来表示无法用仿射表达式表达的值，再通过人工指示参数范围来完整描述程序。

代码 3.3 中使用 PLANT 计算数组分段和，省略了部分无关代码。结果如代码 3.4 所示，计算 `x[offsets[i]]` 到 `x[offsets[i+1]]` 的和，保存到 `y[i]` 中，即 $y_i = \sum_{j=\text{offsets}_i}^{\text{offsets}_{i+1}} x_j$ 。编译器不知道 `m`, `b0`, `b1` 等变量的取值范围，会生成多余的代码用于范围检查，并可能无法进行应用实际上合法的调度。PLANT 允许用户通过 `set_constraint` 函数，人工向编译器传递变量的取值范围信息，从而解决了这个问题。

```
let m = f.comp("m", x![], x!(num_segments(0)));
let b0 = f.comp("b0", x![m,], x!(offsets(i0)));
let b1 = f.comp("b1", x![m,], x!(offsets(i0 + 1)));
let y_init = f.comp("y_init", x![m,], x!(0));
let y = f.comp("y", x![m, b1 - b0], x!(0));
y_init.store(buf_y);
y.set_expr(x!(x(i1 + b0) + y(i0, i1 - 1)));
y.store_at(buf_y, x![i0,]);
f.set_constraint(x![m > 0, b0 > 0, b1 > 0, b1 > b0]);
```

代码 3.3 PLANT 定义分段和

```
int m = num_segments[0];
for (int i0 = 0; i0 < m; i0 += 1) {
    int b0 = offsets[i0];
    int b1 = offsets[i0 + 1];
    y[i0] = 0;
    for (int i1 = 0; i1 < b1 - b0; i1 += 1)
```

```

    y[i0] = x[i1 + b0] + y[i0];
}

```

代码 3.4 生成的大致代码

可以在这个算法上应用循环分块，调用 `y.separate(0, 4).split(0, 4)`，即先将循环拆分成能被 4 整除的部分和不能整除的部分，再进行循环分块，结果如代码 3.5 所示。

如果在 `set_constraint` 传入的约束列表中添加 `m%4==0`，则 PLANT 能自动识别出第二个循环的迭代空间为空，实际上不会执行，从而只生成第一个循环。

```

int m = num_segments[0];
for (int i0 = 0; i0 < m / 4; i0 += 1) {
    for (int i1 = 0; i1 <= 3; i1 += 1) {
        int b0 = offsets[i0 * 4 + i1];
        int b1 = offsets[(i0 * 4 + i1) + 1];
        y[i0 * 4 + i1] = 0;
        for (int i2 = 0; i2 < b1 - b0; i2 += 1)
            y[i0 * 4 + i1] = x[i2 + b0] + y[i0 * 4 + i1];
    }
}
int i0 = (m - 1) / 4;
for (int i1 = 0; i1 < m - 4 * i0; i1 += 1) {
    int b0 = offsets[i0 * 4 + i1];
    int b1 = offsets[(i0 * 4 + i1) + 1];
    y[i0 * 4 + i1] = 0;
    for (int i2 = 0; i2 < b1 - b0; i2 += 1)
        y[i0 * 4 + i1] = x[i2 + b0] + y[i0 * 4 + i1];
}

```

代码 3.5 分块后生成的大致代码

TVM [15] 使用基于区间的程序表示，不能自然地表达和操作非矩形的迭代空间，这就是一个典型的例子。尽管使用 TVM 提供的原语也能表示这个程序，但不能对其进行正确的分析和调度。

使用 TVM 应用类似的循环分块，生成的代码中的第二个循环的语义是错误的。类似的问题^①已经被发现和讨论，并且尚未得到修复。如代码 3.6 所示，循环中会在检查 `4*i0+i1` 的范围合法性之前读取 `offsets[i0*4+i1]` 和 `offsets[i0*4+i1+1]`，从而导致越界的内存访问。

```

int i0 = (m - 1) / 4;
for (int i1 = 0; i1 < 4; i1 += 1) {
    if (4 * i0 + i1 < m)
        y[i0 * 4 + i1] = 0;
    for (int i2 = 0, b0 = offsets[i0 * 4 + i1],
         b1 = offsets[i0 * 4 + i1 + 1]; i2 < b1 - b0; i2 += 1)
        if (4 * i0 + i1 < m)

```

^① <https://github.com/apache/tvm/issues/6596>

```

    y[i0 * 4 + i1] = x[i2 + b0] + y[i0 * 4 + i1];
}

```

代码 3.6 TVM 生成的大致代码

这在根本上源于 TVM 的代码变换实现中没有考虑内层循环的迭代范围可能依赖于外层循环变量和访存的结果，错误地认为迭代范围总是可以在任意位置读取。而 PLANT 中没有这种假定，可以在正确的位置检查范围合法性和读取迭代范围。

一个更复杂的例子是稀疏矩阵乘法 (Sparse Matrix Multiplication, SPMM)。代码 3.7 使用 PLANT 定义了稀疏矩阵 a 和稠密矩阵 b 的乘法，结果保存在稠密矩阵 y 中，省略了初始化和部分无关代码。稀疏矩阵 a 采用 CSR [29] 格式存储，其中 ptr 数组记录某行包含的元素下标范围，idx 数组记录元素列号，val 数组记录元素取值。即计算 $y_{i,j} = \sum_{k=ptr_i}^{ptr_{i+1}} val_k b_{idx_k,j}$ 。

```

let b0 = f.comp("b0", x![m, ], x!(ptr(i0)));
let b1 = f.comp("b1", x![m, ], x!(ptr(i0 + 1)));
let y = f.comp("y", x![m, b1 - b0, m], x!(0));
y.set_expr(x!(val(i1 + b0) * b(idx(i1 + b0), i2) +
  y(i0, i1, i2 - 1)));
f.set_constraint(x![m > 0, b0 > 0, b1 > 0, b1 > b0]);
y.split(1, 32);
y.cache_identity(val, 1, Local);
y.cache_identity(idx, 1, Local);

```

代码 3.7 PLANT 定义稀疏矩阵乘法

循环和内存调度部分，对上述算法应用循环分裂，并在分裂后的第 1 层循环内缓存对 val 数组和 idx 数组的读取，最终结果如代码 3.8 所示。PLANT 能计算出第 1 层循环内每次至多访问 val 数组和 idx 数组的 32 个元素，因此用于缓存的数组大小可以定为 32。

这个示例有意演示了不拆分完整和部分分块的情形，b1 - b0 不一定是 32 的倍数，所以缓存部分和访问部分都自动插入了适当的范围检查，如果调用 separate，也会如代码 3.5 一样，在第一个循环中省去范围检查。

```

for (int i0 = 0; i0 < m; i0 += 1) {
  int b0 = ptr[i0];
  int b1 = ptr[i0 + 1];
  for (int i1 = 0; i1 <= (-b0 + b1 - 1) / 32; i1 += 1) {
    int cache_val[32];
    for (int i2 = 0; i2 <= 31; i2 += 1) {
      if (b1 >= (b0 + i2 % 32) + 1)
        cache_val[i2] = val[(b0 + i1 * 32) + i2];
    }
    int cache_idx[32];
    for (int i2 = 0; i2 <= 31; i2 += 1) {

```

```

    if (b1 >= (b0 + i2 % 32) + 1)
        cache_idx[i2] = idx[(b0 + i1 * 32) + i2];
}
for (int i2 = 0; i2 <= min(31, -b0 + b1 - 32 * i1 - 1);
     i2 += 1) {
    for (int i3 = 0; i3 < m; i3 += 1) {
        y[i0 * m + (i1 * 32 + i2)] = y[i0 * m + (i1 * 32 + i2)] +
            cache_val[i2] * b[cache_idx[i2] * m + i3];
    }
}
}
}

```

代码 3.8 生成的大致代码

TVM 依然不能为这个程序生成正确的代码，因为它会过高估计访问的数组元素数，从而保守地缓存整个数组，这样完全失去了缓存的意义，效率很差。

第 4 章 自动调度

如前所述，为日渐复杂的硬件和算子生成高效代码变得越来越困难。尽管与直接编写 C++/CUDA 代码相比，使用手工调度的张量编译器已经极大地减少了用户的工作量，但编写高效的调度仍然需要用户对硬件体系结构的细致理解。

对于典型的算子，存在数量极其庞大调度组合，它们在逻辑上不改变程序语义（不考虑浮点运算顺序的影响），但由于流水线调度，访存局部性，向量化，多线程等硬件因素而在性能上有显著差异。用户必须从这些调度中进行选择，因此生成高效的代码仍然构成巨大的工程量。

以通用矩阵乘法为例，它是机器学习等领域中的重要算子。高度优化的通用矩阵乘法 CPU 实现需要包括循环分块，循环并行化，循环向量化，循环展开等调度，以及可能的数据布局转换，所有这些调度都包含数目庞大的参数可能取值。高度优化的 GPU 实现甚至更复杂，需要考虑不同内存层级间的数据传输，内存访问合并，线程束分化等 GPU 特有的问题。

PLANT 的自动调度器通过机器学习算法自动优化调度的参数取值，从而减轻这种工程负担。它借鉴了 AutoTVM [16] 的设计架构，使用基于统计数据的性能模型和模拟退火算法，在用户指定的搜索空间中进行高效的搜索，找出最小化运行耗时的调度。

4.1 问题描述

在将计算 c 生成后端代码前，PLANT 会对 c 应用一系列的调度，用 S_c 表示所有可能调度的空间。对于 $s \in S_c$ ，用 $f(c, s)$ 表示计算 c 经过调度 s 和固定的后端代码生成后，在硬件上的实际运行耗时。自动调度的目标是找到最小化运行耗时的调度，即 $\arg \min_{s \in S_c} f(c, s)$ 。

一般地， $|S_c|$ ，即算子的调度空间大小可能是无限的。PLANT 的自动调度器不追求完全的自动化，而是要求用户给出搜索空间的有限子集 $S \subset S_c$ 。具体操作上，用户定义一组可调优的参数，并提供使用这些参数的调度模版，这样就把调度空间限定在了有限但仍然相当大的范围内，自动调度器通过搜索来寻找最优的参数取值。

PLANT 目前实现了如下调优参数：

- Knob: 用户提供一组可选取值
- Split: 自动计算循环分裂的可能参数, 可选按循环跨度的因子分裂, 或按小于等于循环跨度的 2 的幂次分裂
- Tag: 调优循环标记
- Reorder: 调优多层循环嵌套顺序

代码 4.1 展示了用户使用自动调度器定义搜索空间和应用搜索空间中的调度的流程。

```
// define the search space
space.define_split("sp", SplitPolicy::new(oc)
    .set_pow2(true).set_n_output(4));
...
// apply a schedule in the space
let sp = cfg.get("sp");
b.split(0, sp[0]).split(0, sp[1]).split(0, sp[2]);
...
```

代码 4.1 自动调度器使用示例

4.2 性能模型

获取一组 $f(c, s)$ 的准确取值的唯一可靠方式是在实际硬件上运行实验, 收集性能数据。对于张量编译器而言, 编译和运行一个程序通常不超过几秒, 实验成本相对较低。但搜索过程需要探索非常多的调度 s , 为每个 s 都运行实验仍然是不可接受的。因此需要建立性能模型, 在不实际运行的情况下, 用更低的代价对 $f(c, s)$ 进行预测。

自动调度的多面体编译器试图通过 PluTo [18] 算法等选择最优的调度, 这本质上就是一个对 f 进行预测的性能模型。实践表明这样的性能模型难以达到足够高的准确度, 与之相反, PLANT 的自动调度器不对 f 的表达式做出任何先验假设。它完全依赖基于统计数据性能模型, 通过运行实验收集具体的 $f(c, s)$ 的取值, 从而对其他输入下 f 的取值进行预测。这里同样需要考虑实验成本相对较低的性质, 用多次探索实验换取更精确的性能模型是值得的。

PLANT 使用 XGBoost [30] 算法构建性能模型。为了从搜索过程收集的历史数据 D 中训练模型, 需要设定损失函数。PLANT 支持两种损失函数:

- 回归损失函数 Reg: $\sum_i (f(c, s_i) - \hat{f}(c, s_i))^2$ 。模型会尽可能准确地拟合运行耗时。
- 排序损失函数 Rank [31]: $\sum_{i,j} \log(1 + e^{-\text{sgn}(f(c, s_i) - f(c, s_j))(\hat{f}(c, s_i) - \hat{f}(c, s_j))})$ 。因为最

终目标是找到最小化耗时的调度，搜索过程不必关心耗时的绝对取值，只关心相对顺序。

为了将一组 (c, s) 转化为模型的输入，需要从中抽取特征向量。PLANT 支持两种特征抽取方法：

- 参数抽取 Knob：直接使用 s 中的参数值作为向量。
- 循环特征抽取 Iter：对调度后的程序中的每层循环抽取特征。

具体实现上，Knob 对其中 Tag 参数采用独热 (One-Hot) 编码 [32]；Iter 抽取的特征包括循环中的浮点运算次数，循环变量参与的访存的跨度 (stride)，次数 (count)，重用次数 (reuse) 等信息 [33]，还会考虑并行的循环中的访存特征，将这些数值展开成向量。代码生成器可以配置成保留循环跨度为 1，即退化 (degenerate) 的循环，从而保证循环层数固定，因此向量长度是固定的，这样才能合法地作为 XGBoost 模型的输入。

Knob 的运行代价更小，适合调优运行耗时较低的算子；Iter 能给出更精确的预测结果，适合调优运行耗时较高的算子。章节 6.3 对这些方法的效果和性能进行了详细比较。

4.3 搜索系统

PLANT 的自动调度器建立了如图 4.1 所示的搜索系统。搜索模块依据性能模型给出的预测值 $\hat{f}(c, s)$ ，决定需要在真实硬件上运行实验的调度，将调度和实验反馈的运行时间保存在数据库中，数据库又用于训练更新性能模型。

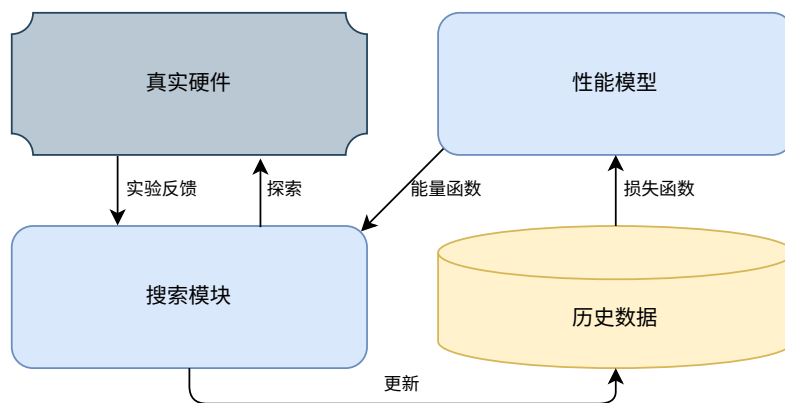


图 4.1 搜索系统架构

搜索模块使用算法 4.1，由于搜索空间规模巨大，不能在每轮迭代中遍历空间来获取 b 个候选调度，而是维护一组当前调度，对其进行随机修改，用性能模

型的预测值作为能量函数，使用模拟退火算法 (Simulated Annealing [34]) 选择保留下来的候选调度。此外，搜索采用 ϵ -贪心策略 [35]，将一定比例的候选调度替换成随机调度，以保证搜索结果的多样性。

算法 4.1 在搜索空间中寻找最佳调度

输入: 计算 c ，搜索空间 S ，尝试次数 $trails$

输出: 最佳调度 s^*

$D \leftarrow \emptyset$

$S' \leftarrow$ 在 S 中随机选择 b 个调度

$t \leftarrow 0$

while $t < trails$ **do**

$S' \leftarrow$ 在上一轮的 S' 上应用模拟退火算法，选择 b 个调度

$S' \leftarrow$ 将 S' 中 ϵb 个元素随机替换成 S 中的调度

for all $s \in S'$ **do**

$f' \leftarrow$ 在硬件上测量实际运行耗时 $f(c, s)$

$D \leftarrow D \cup \{(s, f')\}$

end for

 用 D 更新性能模型，从而更新模拟退火的能量函数 \hat{f}

$t \leftarrow t + b$

end while

$(s^*, _) \leftarrow \arg \min_{(s, f') \in D} f'$

为了提高搜索算法的运行效率，实验探索部分会并行地应用 S' 集合中的多个调度并编译，然后将编译好的程序依次串行执行。

搜索用到的超参数包括批次大小 b ，贪心参数 ϵ ，XGBoost 模型也包含许多可调节的参数，默认值一般就可以达到较好的效果。

第 5 章 系统实现和优化

5.1 ISL 接口

ISL 由 C 语言编写，提供了官方的 C++ 和 Python 接口，但没有提供 Rust 接口。PLANT 的整体系统由 Rust 实现，为了在其中使用 ISL，配套实现了 ISL 的 Rust 接口 `isl-rs` 作为基础库。接口生成的技术流程大致是通过 Clang 解析 ISL 头文件，访问 C 语言语法树，从而自动生成大量有规律的接口代码。

Rust 已经有成熟的 C 语言接口生成工具^①，但生成的是不安全的原始接口代码，不便于使用，一般还需要手工包装一层。与之相反，`isl-rs` 直接生成易用，安全的封装后代码，且具有以下特点：

1. 零额外开销：将 ISL 中的结构体指针无包装地用 Rust 中的类型表示，且接口函数全部内联，从而避免了任何不必要的开销。例如指针类型 `isl_map *` 被映射到 Rust 类型 `Map`，其内部表示如代码 5.1 所示，保证内存布局与 C 语言中的指针相同，且只能保存非空指针。

```
#[repr(transparent)]  
pub struct Map(pub NonNull<c_void>);
```

代码 5.1 Map 定义

2. 安全可靠：借助 Rust 的类型系统限制输入取值，并强制用户对错误显式处理，保证安全性和可靠性。以代码 5.2 为例，ISL 要求输入指针参数非空，同时可能返回空指针。使用 C 语言接口的用户可能忘记检查返回值的有效性，直接用于后续操作，引发错误。

```
isl_map *isl_map_apply_domain(isl_map *map1, isl_map *map2);
```

代码 5.2 ISL 函数示例

这个函数在 Rust 接口中对应代码 5.3，其中 `self` 和 `map2` 都是非空指针，且返回类型为 `Option<Map>`，确认有效后才能作为其他函数的输入。借助 Rust 的 `?` 运算符^②，有效性检查的代码编写负担很小。

此外，通过为这些 Rust 类型实现析构函数，避免了用户手动释放资源的麻烦和可能的重复释放错误。

^① <https://github.com/rust-lang/rust-bindgen>

^② <https://doc.rust-lang.org/edition-guide/rust-2018/error-handling-and-panics/the-question-mark-operator-for-easier-error-handling.html>

```
impl Map {
  fn apply_domain(self, map2: Map) -> Option<Map> { ... }
}
```

代码 5.3 Map 函数示例

3. 节省资源: ISL 对指针类型的参数有两种标记^①: `__isl_keep` 表示函数只使用指针, 用户可以继续使用它, 并仍然需要负责释放它; `__isl_take` 表示函数负责释放指针, 用户不能再使用或释放它。

这与 Rust 的所有权系统^② 中借用和移动的概念非常匹配, 被移动的变量将不再调用析构函数, 所以可以移动标记有 `__isl_take` 的参数, 借用标记有 `__isl_keep` 的参数。

与之相反, 因为缺少语言特性, C++ (也有移动的概念, 但被移动的对象仍然会调用析构函数) 和 Python 接口中不能释放用户传入的参数, 必须对标记有 `__isl_keep` 的参数进行复制操作以保证用户仍能使用, 产生额外开销。

5.2 代码生成

PLANT 为 CPU 后端生成 C++ 源码, 为 GPU 后端生成 CUDA 源码。主流的张量编译器通常为 CPU 后端生成 LLVM IR (如 [15], [14], [23]), 但综合考虑项目工作量, 调试便捷性和代码可读性, PLANT 没有选择 LLVM IR。而且实验表明, 配合章节 5.4 描述的优化手段, 精细控制下生成的 C++ 源码由 Clang 编译生成 LLVM IR 和直接生成 LLVM IR 在最终生成的可执行代码性能上没有本质差距。

在依次完成循环调度和内存调度后, 代码生成器得到最终形式的中间表示作为输入。进入代码生成部分, 将所有计算的循环迭代空间组合起来, 使用 ISL 提供的 CLooG [36] 代码生成算法, 从迭代空间生成一棵 ISL 定义的抽象语法树 (Abstract Syntax Tree, AST), AST 中计算的排列顺序体现了计算的执行顺序, 遵循各个计算迭代空间的字典序。

得到 AST 后, 代码生成器遍历它以生成目标后端的代码, 遍历会进行多次以收集必要的信息。依据计算中循环的标记进行代码生成, 标记处理方式如下:

- GPU 块或线程 (GPU*): 要求映射到 GPU 块或线程的一组循环在嵌套层次上是相邻的, 在最外层循环处生成并调用 GPU 内核函数 (Compute Kernel), 用

^① <http://isl.gforge.inria.fr/user.html#Memory-Management>

^② <https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html>

块和线程编号（如 `threadIdx.x` 等）表示这几层循环迭代变量。

映射到 GPU 块或线程的循环的迭代空间可以是非矩形的。代码生成器利用多面体模型分析出循环跨度的上界作为内核函数的启动参数，然后在内核函数中进行必要的范围检查。

例如为循环 $\{c[i_0, i_1] : 0 \leq i_0 < 100 \wedge 0 \leq i_1 < i_0\}$ 生成代码，其中第 0 层循环绑定到 `GPUBlockX`，第 1 层循环绑定到 `GPUThreadX`，结果如代码 5.4 所示，`exec_kern` 函数只是进行简单的转发。

```
auto _kern = [=]__device__(int *restrict a) {
    int i0 = blockIdx.x + 1;
    int i1 = threadIdx.x + 0;
    if (i1 < i0) { ... }
};
exec_kern<<<dim3(99, 1, 1), dim3(99, 1, 1)>>>(_kern, a);
```

代码 5.4 三角形迭代空间

多面体模型能分析出第 0 层循环仅在 $1 \leq i_0 \leq 99$ 时执行（因为 i_0 取 0 时 i_1 没有合法的取值），因此 `GPUBlockX` 启动参数为 99。第 1 层循环的跨度是 i_0 ，最大值为 99，因此 `GPUThreadX` 启动参数为 99。范围检查仅在必要时进行，启动参数指定的每个 `blockIdx.x + 1` 恰好对应于 i_0 的每个合法取值，因此不需要检查 i_0 的范围。

内核函数使用 `lambda` 表达式生成^①，自动捕获标量，手动传入指针，原因参考章节 5.4.5。

- 并行化 (Parallel): 类似于映射到 GPU 块或线程，也使用 `lambda` 表达式生成并行的内核函数，以此为参数调用 PLANT 运行时实现的并行库中的 `parallel_launch` 函数。范围检查机制也是类似的。

- 向量化 (Vectorize): 只支持向量化最内层循环，且要求循环跨度是常数。分析语句中对循环变量的引用，从而计算生成的 Ramp 表达式的元素跨度和数目，然后将整个循环用一条向量操作表示。

生成的代码中的向量宽度可能不在处理器支持范围内，需要后端编译器翻译为合适的向量指令。

- 循环展开 (Unroll 和 UnrollExplicit): `Unroll` 为生成的循环添加 `#pragma unroll` 编译指令。`UnrollExplicit` 要求循环跨度是常数，彻底展开循环，为循环变量的每个取值复制一遍代码。

^① <https://developer.nvidia.com/blog/new-compiler-features-cuda-8>

5.3 运行时库

5.3.1 内存管理

PLANT 的运行时库提供了 Array 和 Slice 两种多维数组类型，二者区别在于前者拥有并负责释放内存，后者只是一片内存的视图 (View)，其余属性和功能一致，因此以下只介绍 Array。

与章节 2.1.3 中介绍的缓冲区 Buf 相比，Array 是实际分配内存的多维数组，在代码运行阶段使用；而 Buf 是中间表示中概念上的多维数组，在代码生成阶段使用。

Array 具有维度 (dims，部分框架中也称为形状，shape)，数据类型，存储位置三个属性。维度是整数列表，每个元素表示这一维的大小；数据类型与 Buf 的数据类型是同一个概念；存储位置支持 CPU 和 GPU 两种，支持双向传输数据。

Array 默认按照行主序 (Row Major) 存储。支持全零，无初值，随机值三种初始化方式。支持相等和近似相等检测。

5.3.2 代码运行

为了运行生成的算子 (例如在自动调度器中需要频繁地测试算子性能)，先将算子代码编译成动态库，然后加载动态库并调用其中的函数。

由于算子的函数参数个数是不确定的，为了能正确调用函数，代码生成器额外定义一个包装 (Wrapper) 函数，它接受指针的数组，从中读取出实际调用的指针参数并调用函数。

5.4 性能优化

为了获得更好的整体性能，PLANT 对代码生成部分和运行时库部分都进行了大量的优化，以下介绍重要的几点。

5.4.1 向量化

PLANT 使用 Clang (采用 LLVM 后端) 为 CPU 后端生成二进制代码。实验表明 (通过性能测试和阅读输出的汇编代码)，LLVM 能进行一定程度的自动向量化，Clang 前端也可以接受类似 `#pragma vectorize` 的编译指令，指示 LLVM 进行向量化。但两种方法性能都差于手动编写向量代码，因此代码生成器在处理 `Vectorize` 标签时就进行了向量化。

在向量宽度选择上，AVX512^① 提供 512 位向量运算功能，一次能对更多数据进行操作，但可能导致 CPU 降频^②，不一定能提高整体性能，因此目前 Clang 默认不启用。实验表明，CPU 上的矩阵乘法和卷积两个算子中启用 AVX512 后能够达到更好的性能，所以通过编译选项 `-mprefer-vector-width=512` 手动指示 Clang 启用。

5.4.2 循环展开

实验表明，LLVM 的自动循环展开已经相当成熟，几乎不需要编译指令或手动展开。PLANT 使用 NVCC 为 GPU 后端生成二进制代码，与 LLVM 相反，NVCC 几乎不进行自动循环展开，而且类似 `#pragma unroll` 的编译指令效果也较差，手动展开效果明显更好。

这体现了张量编译器和传统编译器任务的划分，张量编译器擅长进行高层次的优化，如循环迭代空间的变换，而底层的代码生成和优化部分则较少关心，可以充分利用传统编译器已有的成果。当底层的传统编译器因为信息不足或实现缺陷无法进行某种优化时，高层的张量编译器也可以予以辅助。

5.4.3 并行库优化

为了避免额外开销以及对生成的代码实现精细控制，PLANT 的运行时库没有使用 OpenMP 等高层的并行库，而是基于更底层的 `pthread` 库实现了并行库。并行库的实现需要考虑超线程，绑定核心，线程池等因素。

超线程 [37] 允许一个物理 CPU 内核提供两个或更多逻辑线程。轻负载任务可以借此充分利用 CPU 运算单元，隐藏内存延迟。但在重负载任务，尤其是张量计算这种浮点密集场合中，运算单元可以被单个线程充分利用，此时多个逻辑线程竞争执行单元，反而降低效率 [38]。因此，运行时在确定并行执行的任务数时，会依据是否存在超线程将逻辑线程数减少。

核心绑定是将操作系统线程绑定到 CPU 线程上，从而减少操作系统调度的开销，运行时通过 `pthread` 的线程亲和性设置函数实现这一点。线程池是预先启动多个线程，收到执行请求时，将任务分配到已有的线程上执行，从而减少线程创建和销毁的开销。

^① <https://en.wikipedia.org/wiki/AVX-512>

^② <https://stackoverflow.com/questions/56852812/simd-instructions-lowering-cpu-frequency>

5.4.4 刷新缓存

为了缓解单次运行时间的随机性的影响，测试算子性能时需要多次重复运行算子并取耗时平均值或中位数，这种情况下算子的输入数据极有可能被保留在高速缓存中。

这不完全符合运行神经网络的端到端推理的实际情况。例如，单精度浮点数的 ResNet152 模型的参数大小超过 200MiB^①，这远超出常见 CPU 的各级缓存大小。在网络中运行算子时，其输入可能不在高速缓存中，从而导致自动调度得到的单个算子的最优调度在网络中不是最优的。为解决这个问题，性能测量模块支持在测试性能中每次运行前清除输入数据的缓存，从而使自动调度更接近实际情况，得到使整体推理性能最优的调度。

具体实现上，运行时在 x86_64 平台上使用 `clflush` 指令^②，在 ARM 平台上使用 `dc` 指令^③，其他平台尚不支持。

5.4.5 保留指针 `restrict` 属性

`restrict` 是 C/C++ 语言中的关键字（或编译器拓展的关键字），修饰指针类型，它指示编译器指针指向的值只可能通过这个指针修改，不会受到对其他指针的写入的影响。`restrict` 属性允许编译器更激进地分析程序，是编译器进行自动向量化等高级优化的基础。

实验表明，在性能评测使用的工具链版本中，`lambda` 表达式捕获的指针会丢失 `restrict` 属性，从而导致编译器只能进行保守的优化，生成次优的代码。为解决这个问题，代码生成器采用类似代码 5.5 的模式，额外增加一层调用，将指针的 `restrict` 属性表示在参数列表中。额外的调用会被编译器内联优化，不会对性能产生负面影响。

```
float *a;
float *restrict b;
// 原始版本，丢失 restrict 属性，性能低下
auto _kern = [=]() { /* use a, b */ };
// 优化版本，保留 restrict 属性，性能明显提升
auto _kern = [=]() {
    [=](float *a, float *restrict b)
        { /* use a, b */ }(a, b);
};
```

代码 5.5 `restrict` 优化示例

① <https://github.com/albanie/convnet-burden>

② <https://www.felixcloutier.com/x86/clflush>

③ <https://developer.arm.com/documentation/dui0801/g/A64-General-Instructions/DC>

5.5 远程执行

目前一种常见的需求是在移动或嵌入式设备上运行深度学习模型的推理，这要求为目标设备生成高效的张量算子。然而目标设备一般性能低下，内存和磁盘空间有限，无法运行完整的张量编译器，需要在性能和资源充足的主机设备上完成编译过程，然后在目标设备上运行算子。

为此 PLANT 实现了远程执行功能，它定义了一套轻量级的网络协议，将编译生成的二进制代码传输到目标设备上，后者执行后返回结果。LLVM 默认就能生成多种平台的代码，但仍需要目标平台的链接器，头文件，库文件等才能完成完整的编译过程。用户可以选择在主机端安装交叉编译器，也可以在目标平台上调用链接器。

与 TVM 的远程过程调用（Remote Procedure Call, RPC）模式相比，PLANT 的远程执行功能仅要求目标设备执行网络请求，加载和执行动态库，这均只依赖基本的 C 标准库，无需安装包括 Python, Numpy, TVM Runtime 在内的一系列复杂的依赖。值得注意的是，microTVM^① 能支持更底层的目标设备，但它仍处于开发阶段，尚未提供方便稳定的应用接口。

5.6 Python 接口

PLANT 提供了简易的 Python 接口，允许用户通过 Python 调用编译好的算子库，从而可以方便地将算子库接入已有系统，进行快速调试和迭代等。

使用实例如代码 5.6 所示。程序先用 NumPy 生成随机输入，计算矩阵乘法，然后加载 PLANT 编译生成的动态库并调用，最后分别用二者提供的测试函数检测结果的正确性，中间进行了必要的数据传输。

```
import plant
from plant import Array, Func
import numpy as np

M = N = K = 2048

a = np.random.uniform(size=(M, N)).astype(np.float32)
b = np.random.uniform(size=(N, K)).astype(np.float32)
c = np.dot(a, b)

a_gpu = Array.from_np(a).to_gpu()
b_gpu = Array.from_np(b).to_gpu()
c_gpu = Array.alloc(c.shape, ty=plant.F32, loc=plant.GPU)
f = Func('./matmul_gpu.so')
f(a_gpu, b_gpu, c_gpu)
```

^① <https://tvm.apache.org/docs/microtvm/index.html>

```
c_cpu = c_gpu.to_cpu()
# 使用NumPy测试, 允许相对误差 1e-5
np.testing.assert_allclose(c, c_cpu.as_np(), rtol=1e-5)
# 使用PLANT测试, 允许绝对误差 1e-2
c_cpu.assert_close(Array.from_np(c), threshold=1e-2)
```

代码 5.6 Python 接口使用示例

从示例代码可见, Python 接口与 NumPy 有良好的兼容性, 且可以方便地在 CPU 和 GPU 间传输数据。值得注意的是, 这里只是展示 Python 接口与 NumPy 的互操作, 不意味着前者依赖于后者, 事实上前者可以独立于后者运行, 直接从 Python 列表或 Python 的 C 语言接口^① 中获取数据。

^① <https://docs.python.org/3/library/ctypes.html>

第 6 章 性能评测

6.1 算子性能

算子性能评测实验分别在 CPU 和 GPU 上评测了单精度浮点数的矩阵乘法 (General Matrix Multiplication, GEMM) 和卷积 (Convolution, CONV) ^①, 二者都是深度学习等应用中的重要算子, 具体运行环境软硬件配置如表 6.1 所示。

对于 GPU 后端, 测量和比较的是 GPU 净执行时间, 不包括 GPU 上的数据分配开销和 CPU 与 GPU 间的数据通信开销。

类型	参数
操作系统	Debian GNU/Linux 10 (buster)
内核	4.19.0-13-amd64
CPU	双插槽 Intel Xeon Gold 5218 @ 2.3GHz
GPU	NVIDIA Tesla P100
RAM	377 GiB DDR4
LLVM 版本	11.0.0
MKL 版本	2021.1
NVIDIA 驱动	450.80.02
CUDA 版本	10.2.89
cuBLAS 版本	10.2.2
cuDNN 版本	7.6.5
Python 版本	3.7.3
PyTorch 版本	1.7.1, torchvision 0.9.0

表 6.1 运行环境

图 6.1 展示了 PLANT 生成的算子与成熟的算子库的执行时间对比。CPU 上采用 Intel MKL [8] 作为参考实现, GPU 上对于矩阵乘法和卷积分别采用 cuBLAS [9] 和 cuDNN [10] 作为参考实现。在四种情形中 PLANT 的性能均略优于算子库。

CPU 和 GPU 上的官方算子库均以其手工精细优化而达到的性能闻名, 因此这个结果是相当难得的。为了达到这样的性能, PLANT 的算子实现使用了包括数

^① 实验选择的算子形状和数据格式分别为: GEMM: $M \times K \times N = 2048 \times 2048 \times 2048$, CONV: $NCHW * OIHW = (256 \times 256 \times 14 \times 14) * (512 \times 256 \times 3 \times 3)$

组打包 [39]，多层循环分块，读写缓存（使用 GPU 共享内存作为读缓存，局部内存作为写缓存），循环展开，向量化在内的众多调度来进行优化，并通过自动调度功能寻找性能最佳的调度参数。

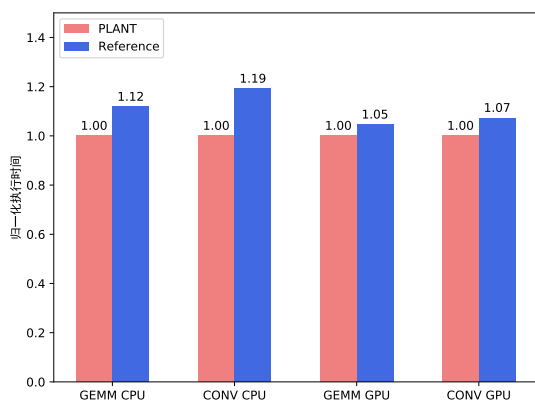


图 6.1 算子性能，与算子库比较

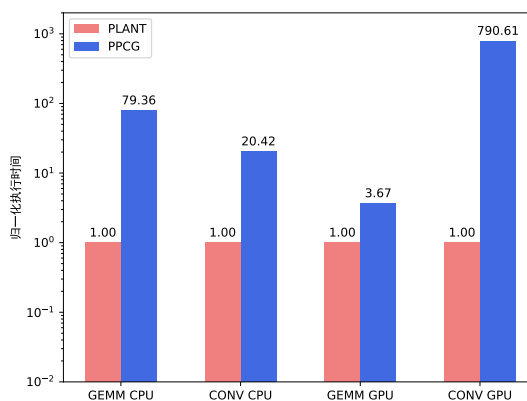


图 6.2 算子性能，与 PPCG 比较

为了进一步体现算子的高性能，可将实际性能与计算设备的理论峰值性能进行比较。CPU 的理论峰值性能计算如式 (6.1) 所示。

$$\begin{aligned} \text{物理核心数} * \text{单个核心 FMA 单元数} * 2 * \frac{\text{向量宽度}}{\text{浮点宽度}} * \text{CPU 频率} = \\ 32 * 1 * 2 * \frac{512}{64} * 2.3 * 10^9 = 235.5\text{GFLOPS} \end{aligned} \quad (6.1)$$

查阅官网^①得知 GPU 的理论峰值单精度浮点性能是 9.3TFLOPS。据此计算，四种情况中，实际浮点运算性能分别达到了理论峰值性能的 86.1%，77.1%，85.4%，77.8%。

需要说明的是，PLANT 和 Intel MKL 中卷积均使用 DIRECT 卷积算法。对于特定形状的卷积，这并非目前最高效的算法，但用来衡量张量编译器的性能是合适的。目前 cuDNN 卷积支持 IMPLICIT GEMM, IMPLICIT PRECOMP GEMM, GEMM, FFT, FFT TILING, WINOGRAD, WINOGRAD NONFUSED 算法，但不支持 DIRECT 算法^②。其中只有 IMPLICIT GEMM 算法不需要在计算前申请额外的内存，因此选择用它和 PLANT 的 DIRECT 算法做比较。但因为本质上使用的底层算法并不一样，这个比较结果仅供参考。

PLANT 与 PPCG [19] 生成的算子的执行时间对比如图 6.2 所示。PPCG 是全

① <https://www.nvidia.com/en-us/data-center/tesla-p100>

② https://docs.nvidia.com/deeplearning/cudnn/api/index.html#cudnnConvolutionFwdAlgo_t

自动的多面体编译器，直接对输入的 C 代码进行循环变换，为 CPU 后端生成并行化的 C 代码，为 GPU 后端生成 CUDA 代码。PPCG 生成的 CUDA 代码默认包括数据分配和复制，输入范围检查等额外操作，为了测量 GPU 净执行时间，实验预先对它的代码生成逻辑进行了一定的修改。

尽管 PPCG 不需要人工编写调度指令，达到了较高的自动化程度，但它的性能非常糟糕，在四种情形下比 PLANT 慢至多数百倍。PPCG 没有应用数组打包；对于 CPU 上的矩阵乘法和卷积，PPCG 没有应用任何循环变换，只是简单地将最外层循环并行化；对于 GPU 上的卷积，PPCG 没有使用共享内存作为输入数据的读缓存和局部内存作为输出数据的写缓存。此外，PPCG 几乎没有对生成的代码进行任何底层优化，例如循环展开和向量化，这导致它不能充分利用体系结构相关的硬件资源。

6.2 神经网络推理性能

神经网络推理性能评测实验在 CPU 上评测了不同网络深度的 ResNet 的端到端推理性能，选择 PyTorch [6] 作为参考实现，批次大小为 1，数据布局为 NCHW。执行时间对比如图 6.3 所示，依据网络深度不同，PLANT 相比 PyTorch 有至多 4.95 倍的性能提升。

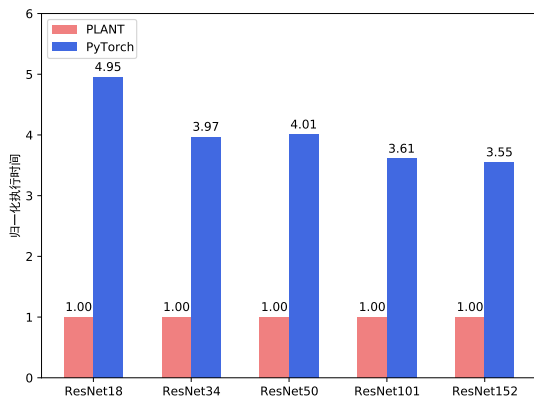


图 6.3 神经网络性能

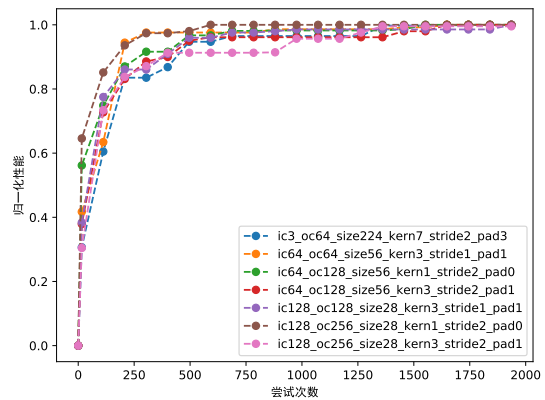


图 6.4 自动调度不同形状的卷积

ResNet 的主要计算量来自各种形状的卷积，手工为每个卷积逐个调优的工作量是不可接受的。PLANT 的神经网络推理实现通过自动调度器为每个卷积生成了合适的调度，其中部分卷积自动调度的性能曲线如图 6.4 所示，具体细节在章节 6.3 描述。

借助 PLANT 的远程执行功能，实验还在瑞芯微 RK3399 CPU 上评测了 ResNet 的端到端推理性能，具体运行环境软硬件配置如表 6.2 所示。启用全部核心和仅启用大核心的执行时间对比分别如图 6.5 和图 6.6 所示。

类型	参数
操作系统	Ubuntu 18.04.5 LTS
内核	4.4.194-59308-g3c831b7a7534
CPU	2 * A72 @ 1.8GHz + 4 * A53 @ 1.4GHz
GPU	Mali-T860
RAM	1.91 GiB DDR3
主机 LLVM 版本	11.1.0
OpenCL 版本	1.2
Python 版本	3.6.9
PyTorch 版本	1.1.0, torchvision 0.3.0

表 6.2 RK3399 运行环境

RK3399 是 ARM64 平台的嵌入式 CPU，具有 ARM big.LITTLE 架构^①，两颗 Cortex-A72 大核心和四颗 Cortex-A53 小核心。嵌入式平台往往有电压和功耗限制需求，因此一种常见的执行模式是在大核心上运行计算密集应用，从而保证较高的数据吞吐速率；在小核心上运行轻量级的系统服务，从而保证较低的系统延迟和功耗。当然，为了极致的数据吞吐速率，也存在使用全部核心运行计算密集应用的场景。

为此，实验评测了启用全部核心和仅启用大核心两种情形。PLANT 的运行支持这样的启动配置；通过环境变量和运行时的适当配置，也可以让 PyTorch 的后端达到一样的效果。可见在两种情况下，PLANT 相比于 PyTorch 均有 2 倍左右的推理性能提升。

6.3 自动调度

图 6.4 展示了自动调度 ResNet 中部分形状的卷积的性能曲线。图例的名称表示卷积的形状，例如 ic128_oc128_size28_kern3_stride1_pad1 表示卷积输入通道数为 128，输出通道数为 128，图片长宽均为 28，卷积核长宽均

^① https://en.wikipedia.org/wiki/ARM_big.LITTLE

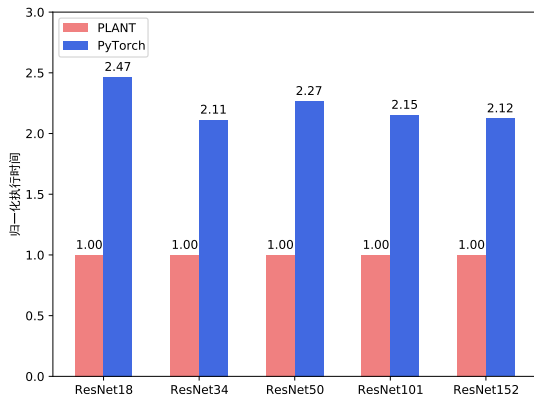


图 6.5 RK3399 神经网络性能，全部核心

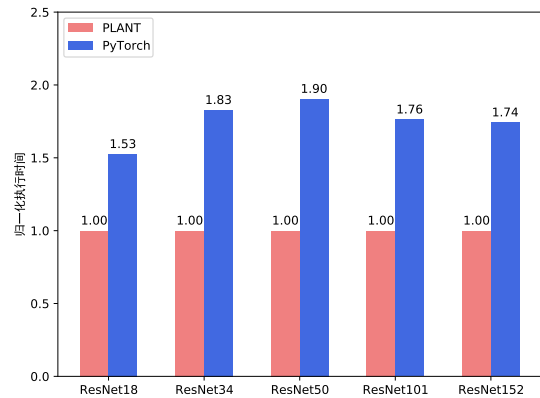
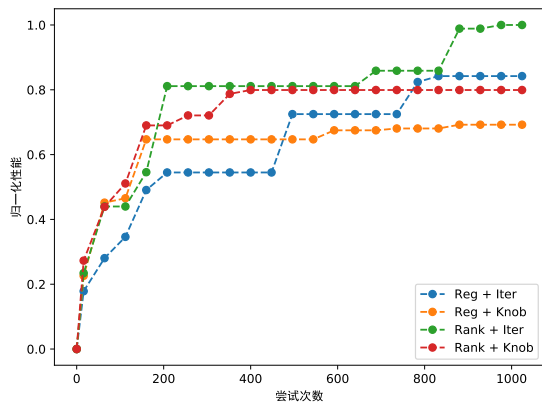


图 6.6 RK3399 神经网络性能，大核心

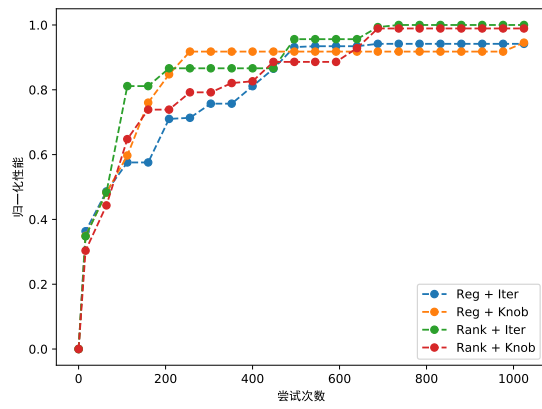
为 3，卷积跨度为 1，补零尺寸为 1。各种形状的卷积调度的搜索空间最大达到了 $3 * 10^7$ 级别，但搜索基本都在 10^3 次尝试内达到收敛或达到最终性能的 90% 左右，可见搜索算法配合基于统计数据性能模型发挥了相当好的效果，避免了大量无意义的探索。

前述章节描述了两种损失函数（回归损失函数 **Reg** 和排序损失函数 **Rank**）和两种特征抽取方法（参数抽取 **Knob** 和循环特征抽取 **Iter**）。在选择不同损失函数和特征抽取方法的四种组合下，分别为四个形状的卷积进行自动调度，性能曲线如图 6.7 所示。可见在多数情况下，只从搜索到的最优调度的性能来评价，经过相同的尝试次数，使用 **Rank** 效果好于使用 **Reg**，使用 **Iter** 效果好于使用 **Knob**。

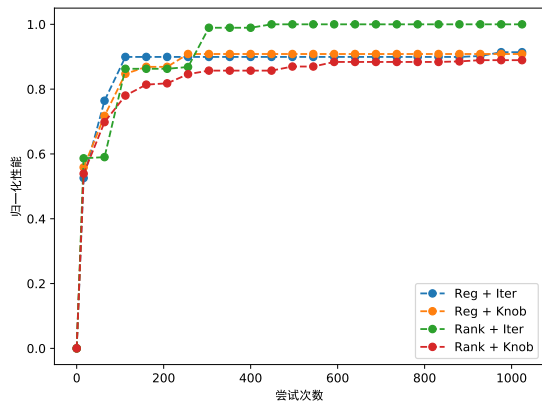
需要说明的是，四种组合下自动调度搜索，模型构建和更新本身的时间开销也略有差距，差距大小取决于不同的系统配置和参数取值。在实验使用的测试环境中，使用 **Rank** 慢于 **Reg**，使用 **Iter** 慢于使用 **Knob**，差距均在 10% 到 50% 间。因此，如果限定系统整体运行时间，搜索效果较差的方法组合在某些场合下也可能达到更好的效果（例如搜索空间大，但单次运行成本相对较小的算子）。



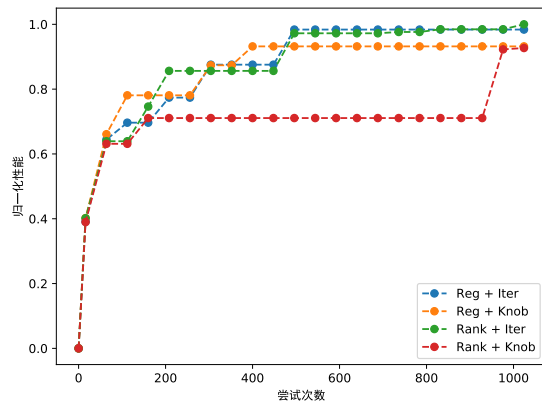
(a) ic3_oc64_size224_kern7_stride2_pad3



(b) ic64_oc64_size56_kern3_stride1_pad1



(c) ic64_oc128_size56_kern1_stride2_pad0



(d) ic64_oc128_size56_kern3_stride2_pad1

图 6.7 比较自动调度的不同方法

第 7 章 总结

本文介绍了基于多面体模型的张量编译器 PLANT。它为用户提供丰富的调度指令，能为 CPU 和 GPU 后端生成高效的代码，并实现了诸多系统优化。它提供了自动调度功能，能减少用户编写调度指令的工程量。它提供了远程执行功能和 Python 接口，满足用户的开发需求。在算子性能和神经网络推理性能测试中，PLANT 的性能均优于常见的参考平台。

目前 PLANT 已经在 GitHub 上开源^①。后续工作有以下几个可能的方向：

- 提供更多调优过的张量算子代码示例，如 NLP 模型，空洞卷积，分组卷积，图像处理和科学计算领域的算子等。
- 实现更多调度指令，例如封装 GPU 张量化操作，`compute_at`^② 等。
- 支持为更多后端生成代码，如分布式系统，FPGA 等。
- 为常见的深度学习前端框架设计前端，自动解析模型文件。
- 提供使用教程，完善代码注释和文档，方便其他人参与二次开发。

^① <https://github.com/mashplant/plant>

^② https://tvm.apache.org/docs/tutorials/language/schedule_primitives.html#compute-at

插图索引

图 1.1	多种多样的深度学习框架和硬件	1
图 4.1	搜索系统架构	21
图 6.1	算子性能, 与算子库比较	32
图 6.2	算子性能, 与 PPCG 比较	32
图 6.3	神经网络性能	33
图 6.4	自动调度不同形状的卷积	33
图 6.5	RK3399 神经网络性能, 全部核心	35
图 6.6	RK3399 神经网络性能, 大核心	35
图 6.7	比较自动调度的不同方法	36

表格索引

表 2.1	PLANT 支持的表达式	6
表 3.1	循环调度指令	10
表 3.2	内存调度指令	10
表 6.1	运行环境	31
表 6.2	RK3399 运行环境	34

参考文献

- [1] ABADI M, BARHAM P, CHEN J, et al. Tensorflow: A system for large-scale machine learning [C]//12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16). 2016: 265-283.
- [2] FLEISCH D A. A student's guide to vectors and tensors[M]. Cambridge University Press, 2011.
- [3] ZHENG S, LIANG Y, WANG S, et al. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system[C/OL]//ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2020: 859–873. <https://doi.org/10.1145/3373376.3378508>.
- [4] VENIERIS S I, KOURIS A, BOUGANIS C S. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions[J]. arXiv preprint arXiv:1803.05900, 2018.
- [5] REUTHER A, MICHALEAS P, JONES M, et al. Survey of machine learning accelerators[C]//2020 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2020: 1-12.
- [6] PASZKE A, GROSS S, MASSA F, et al. Pytorch: An imperative style, high-performance deep learning library[J]. arXiv preprint arXiv:1912.01703, 2019.
- [7] JIA Y, SHELHAMER E, DONAHUE J, et al. Caffe: Convolutional architecture for fast feature embedding[C]//Proceedings of the 22nd ACM international conference on Multimedia. 2014: 675-678.
- [8] INTEL(R). Oneapi math kernel library[Z].
- [9] NVIDIA(R). Cublas library[Z].
- [10] CHETLUR S, WOOLLEY C, VANDERMERSCH P, et al. cudnn: Efficient primitives for deep learning[J]. arXiv preprint arXiv:1410.0759, 2014.
- [11] LI M, LIU Y, LIU X, et al. The deep learning compiler: A comprehensive survey[Z]. 2020.
- [12] KARP R M, MILLER R E, WINOGRAD S. The organization of computations for uniform recurrence equations[J]. Journal of the ACM (JACM), 1967, 14(3): 563-590.
- [13] LAMPORT L. The parallel execution of do loops[J]. Communications of the ACM, 1974, 17(2): 83-93.
- [14] RAGAN-KELLEY J, BARNES C, ADAMS A, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines[J/OL]. SIGPLAN Not., 2013, 48(6): 519–530. <https://doi.org/10.1145/2499370.2462176>.

- [15] CHEN T, MOREAU T, JIANG Z, et al. Tvm: An automated end-to-end optimizing compiler for deep learning[C]//OSDI'18: Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2018: 579–594.
- [16] CHEN T, ZHENG L, YAN E, et al. Learning to optimize tensor programs[C]//NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems. Red Hook, NY, USA: Curran Associates Inc., 2018: 3393–3404.
- [17] ZHENG L, JIA C, SUN M, et al. Anso: Generating high-performance tensor programs for deep learning[C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). 2020: 863-879.
- [18] BONDHUGULA U, HARTONO A, RAMANUJAM J, et al. A practical automatic polyhedral parallelizer and locality optimizer[C/OL]//PLDI '08: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2008: 101–113. <https://doi.org/10.1145/1375581.1375595>.
- [19] VERDOOLAEGE S, CARLOS JUEGA J, COHEN A, et al. Polyhedral parallel code generation for cuda[J/OL]. ACM Trans. Archit. Code Optim., 2013, 9(4). <https://doi.org/10.1145/2400682.2400713>.
- [20] LENGAUER C. Polly—performing polyhedral optimizations on a low-level intermediate representation[J/OL]. Parallel Processing Letters, 2012, 22. DOI: 10.1142/S0129626412500107.
- [21] HALL M, CHAME J, CHEN C, et al. Loop transformation recipes for code generation and auto-tuning[C/OL]//volume 5898. 2009: 50-64. DOI: 10.1007/978-3-642-13374-9_4.
- [22] YUKI T, GUPTA G, DAEGON K, et al. Alphaz: A system for design space exploration in the polyhedral model[C/OL]//2013: 17-31. DOI: 10.1007/978-3-642-37658-0_2.
- [23] BAGHDADI R, RAY J, ROMDHANE M B, et al. Tiramisu: A polyhedral compiler for expressing fast and portable code[C]//CGO 2019: Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization. Washington, DC, USA: IEEE Press, 2019: 193–205.
- [24] HE K, ZHANG X, REN S, et al. Deep residual learning for image recognition[C]//Proceedings of the IEEE conference on computer vision and pattern recognition. 2016: 770-778.
- [25] VERDOOLAEGE S. isl: An integer set library for the polyhedral model[C]//FUKUDA K, HOEVEN J V D, JOSWIG M, et al. Mathematical Software – ICMS 2010. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010: 299-302.
- [26] VERDOOLAEGE S. Presburger formulas and polyhedral compilation[J]. 2016.
- [27] HAASE C. A survival guide to presburger arithmetic[J/OL]. ACM SIGLOG News, 2018, 5 (3): 67–82. <https://doi.org/10.1145/3242953.3242964>.

- [28] ZHANG Z, XU Y, YANG J, et al. A survey of sparse representation: algorithms and applications[J]. IEEE access, 2015, 3: 490-530.
- [29] SAAD Y. Sparskit: A basic tool kit for sparse matrix computations[J]. 1990.
- [30] CHEN T, GUESTRIN C. Xgboost: A scalable tree boosting system[C/OL]//KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: Association for Computing Machinery, 2016: 785–794. <https://doi.org/10.1145/2939672.2939785>.
- [31] BURGESS C, SHAKED T, RENSHAW E, et al. Learning to rank using gradient descent[C]// Proceedings of the 22nd international conference on Machine learning. 2005: 89-96.
- [32] HARRIS D, HARRIS S L. Digital design and computer architecture[M]. Morgan Kaufmann, 2010.
- [33] SNAVELY A, CARRINGTON L, WOLTER N, et al. A framework for performance modeling and prediction[C]//SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing. IEEE, 2002: 21-21.
- [34] KIRKPATRICK S, GELATT C D, VECCHI M P. Optimization by simulated annealing[J]. science, 1983, 220(4598): 671-680.
- [35] KULESHOV V, PRECUP D. Algorithms for multi-armed bandit problems[Z]. 2014.
- [36] BASTOUL C. Code generation in the polyhedral model is easier than you think[C]//PACT '04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques. USA: IEEE Computer Society, 2004: 7–16.
- [37] MARR D T, BINNS F, HILL D L, et al. Hyper-threading technology architecture and microarchitecture.[J]. Intel Technology Journal, 2002, 6(1).
- [38] SAINI S, JIN H, HOOD R, et al. The impact of hyper-threading on processor resource utilization in production applications[C/OL]//2011 18th International Conference on High Performance Computing. 2011: 1-10. DOI: 10.1109/HiPC.2011.6152743.
- [39] GOTO K, GEIJN R A V D. Anatomy of high-performance matrix multiplication[J/OL]. ACM Trans. Math. Softw., 2008, 34(3). <https://doi.org/10.1145/1356052.1356053>.

致 谢

感谢我的导师翟季冬教授和陈渝教授。在大学低年级，陈渝教授培养了我对计算机系统和编译方向的科研兴趣，指导我学习了 Rust 语言并培养了诸多实用技能。在大学高年级，翟季冬老师给予了我科研方向和论文写作方面的指导，鼓励我无后顾之忧地探索未知领域，并为我提供了宽松的学术科研环境以及充足的硬件资源。没有两位老师的悉心辅导就没有我现在的成果。

感谢唐适之学长，翁家翌学长，黄可钊学长，陈晟祺学长，张晨同学，陈嘉杰同学。他们在科研问题，课程学业和课余生活等方面给了我极大的帮助。

感谢清华大学计算机系提供给我这样一个平台，能接触到如此优秀的良师益友，让我能不断突破自我上限，在四年里积累了受用一生的能力和品质，过出了我想要的大学生活。

声 明

本人郑重声明：所提交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签 名： 李晨昊 日 期： 2021年6月15日

附录 A 外文资料的书面翻译

TIRAMISU: 用于表达快速和可移植代码的多面体编译器

摘要: 本文介绍了 **TIRAMISU**, 一个旨在为包括多核 CPU, GPU 和分布式系统在内的多个平台生成高性能代码的多面体编译框架。**TIRAMISU** 引入了带有新颖的调度指令的调度语言, 以明确管理为这些系统生成代码时的复杂性。该框架设计用于图像处理, stencil, 线性代数和深度学习领域。**TIRAMISU** 具有两个主要特点: 它使用基于多面体模型的灵活表示形式, 而且具有语义丰富, 可以对优化进行细粒度控制的调度语言。**TIRAMISU** 使用四层中间表示 (简称 **IR**), 在算法, 循环转换, 数据布局和通信之间进行完全隔离。这种隔离简化了使用相同算法为多个硬件体系结构生成代码的过程。我们通过编写一组图像处理, 深度学习和线性代数基准测试来评估 **TIRAMISU**, 并将它们与最新的编译器和手工优化的库进行比较, 从而证明 **TIRAMISU** 在不同的硬件体系结构 (包括多核 CPU, GPU 和分布式计算机) 上与现有的编译器和库性能相当或更好。

关键词: 代码优化, 代码生成, 多面体模型, 深度学习, 张量, GPU, 分布式系统

A.1 引言

随着体系结构的复杂性和多样性的增加, 为它们生成高效代码变得越来越困难。要获得最佳性能, 需要复杂的代码和数据布局转换, 复杂的内存层次结构管理以及有效的数据通信和同步。

例如, 考虑通用矩阵乘法 (简称 **gemm**), 它计算 $C = \alpha AB + \beta C$, 是众多算法 (包括仿真和卷积神经网络) 的基础单元。高度优化的实现需要融合乘法和加法循环, 以及应用两级循环分块, 向量化, 循环展开, 数组打包, 寄存器分块和数据预取。此外, 由于部分分块无法从相同的优化中完全受益, 优化的实现将部分分块与完整分块的情形分别处理。高性能的 GPU 实现需要更多的优化, 包括合并内存访问, 管理全局内存, 共享内存和寄存器内存之间的数据移动, 并插入同步原语。自动生成这种复杂的代码超出了最新的编译器的能力。**gemm** 这样的 **kernel** 的重要性促使供应商为这些 **kernel** 发布极其复杂的手工优化库。但对于大多数用户来说, 让自己的代码达到这种性能水平很有挑战, 因为在对手工实现复

杂的代码转换时，探索可能的实现方式所需的空间非常棘手。

此前使用多面体模型的工作已经成功实现了复杂的迭代空间转换，数据局部性优化，和内存管理优化。尽管多面体编译器可以表示这些程序和数据转换，但它们仍无法成功选择带来最佳性能转换。当前，对于像 `gemm` 这样的算法，这些编译器的性能与手工优化的 `kernel` 相差很大。图 A.1 中的蓝条展示了 `gemm` 的最新多面体编译器的性能与 Intel MKL 和 Nvidia cuBLAS 库相比的结果。全自动多面体编译器（例如 Polly 和 Pluto）提高了代码编写效率，但未获得所需的性能，因为它们的搜索技术仅考虑必要优化方法的一个子集，并且依赖于精度较低的代价模型，从而导致编译器做出次优的决定。其他多面体框架，例如 AlphaZ 和 CHiLL，则避免了完全自动化，而是提供了一个调度语言，使用户能够高效地探索可能的转换空间。尽管这些框架实现了更好的性能，但它们的调度语言没有对分布式系统的优化。例如它们不允许用户对计算进行划分，跨节点发送数据或插入所需的同步原语。

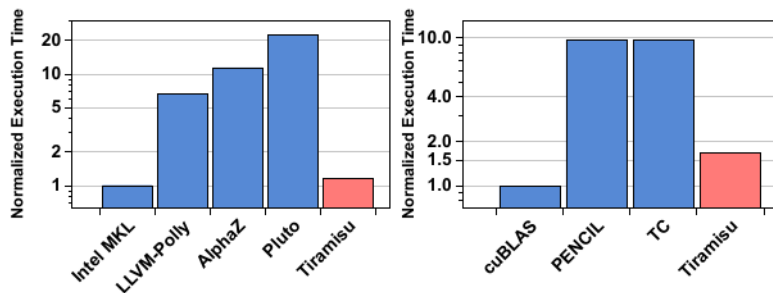


图 A.1 `sgemm` 归一化执行时间，左侧 CPU，右侧 GPU

在本文中，我们介绍了 TIRAMISU，一种具有调度语言的多面体编译器，该调度语言具有针对多种高性能体系结构的新颖的调度指令。TIRAMISU 非常适合用于实现数据并行算法（通过嵌套循环操作数组）。它采用程序的高级表示形式（纯算法和一组调度指令），应用必要的代码转换，并为目标体系结构生成高度优化的代码。除了调度用于循环和数据布局转换的调度指令之外，TIRAMISU 的调度语言还引入了用于显式通信和同步，以及将缓冲区映射到不同的内存层次结构的指令。为了简化调度语言的实现，TIRAMISU 将体系结构无关的算法与代码转换，数据布局和通信分离，从而将 IR 明确划分为四层，以隐藏目标平台的复杂性和多样性。TIRAMISU 面向多核 CPU，CUDA GPU，分布式系统和 FPGA。本文介绍了前三个后端，Del Sozzo 等人的描述了 FPGA 后端。

CHiLL，AlphaZ 和 Halide 等多个编译器证明调度语言对生成高效代码十分

有效。特别是与 Halide 相比，TIRAMISU 不仅引入了新颖的调度指令，而且根本区别还在于，TIRAMISU 依赖于表达多面体模型的表示，而不是 Halide 使用的基于区间的表示。这使 TIRAMISU 可以自然地表达非矩形的迭代空间，以支持具有循环的数据流的程序，并应用任意的仿射变换（包括迭代空间倾斜），而这些仿射变换在 Halide 中都不能自然地表达。

本文做出以下几点贡献：

1. 我们介绍了一种具有调度语言的多面体编译器，该语言具有新颖的调度指令，可用于控制数据通信，同步以及把缓冲区映射到不同的内存层次结构。这些扩展允许了 TIRAMISU 为多种高性能架构生成代码，包括多核 CPU，GPU 和分布式系统。
2. 我们将 IR 明确分为四个层，以简化调度语言的实现。四层 IR 将算法与代码转换和数据布局转换分开，从而实现了可移植性并简化了特定于架构的降低转换的组成。
3. 我们在一组深度学习和线性代数 kernel 上评估了 TIRAMISU，结果显示 TIRAMISU 可以生成比 Intel MKL 性能高出 2.3 倍的高效代码。我们还根据一组图像处理基准对 TIRAMISU 进行了评估，结果表明 TIRAMISU 在不同的硬件体系结构（包括多核 CPU，GPU 和分布式系统）上于最新的编译器性能相当或更好。

A.2 相关工作

(A) 具有自动调度功能的多面体编译器

多面体编译器，例如 PENCIL, Pluto, Polly, Tensor Comprehensions 和 PolyMage 是全自动的。其中一些是针对特定领域设计的（例如 Tensor Comprehensions 和 PolyMage），而 Pluto, PENCIL 和 Polly 则更为通用。尽管全自动编译器可提高代码编写效率，但它们不总能得到最佳性能。次优的性能是由于以下几个原因：首先，这些编译器未实现某些关键的优化，例如数组打包，寄存器分块，数据预取和异步通信（TIRAMISU 均支持）；其次，他们没有精确的代码模型来决定哪些优化是有利的。例如，Pluto 自动调度算法（在 Pluto, PENCIL 和 Polly 中使用）试图在最大程度地提高并行度的同时最小化生产者和使用用户语句之间的距离，但它没有考虑数据布局，冗余计算和控制生成代码的复杂性。TIRAMISU 依赖于一组调度指令，而不是全自动调度，使用户可以完全控制调度。

Amarasinghe 等人和 Bondhugula 等人提出的多面体框架解决了分布式系统自动代码生成的问题。TIRAMISU 并非完全自动化，而是依靠用户提供调度指令来控制生成代码中的选择（同步/异步通信，通信粒度，缓冲区大小，发送和接收时间，通信与重新计算的成本比较，等等）。

(B) 具有调度语言的多面体编译器

AlphaZ, CHiLL 和 URUK 是允许用户使用调度指令表达高级转换的多面体框架。由于这些框架基于多面体模型，它们可以表达任意仿射变换。但是，它们的调度语言并不支持分布式体系结构。相比之下，TIRAMISU 具有对计算进行划分（针对分布式系统），同步和跨节点的数据分布的调度指令。

(C) 具有调度语言的非多面体编译器

Halide 是具有调度语言的图像处理 DSL，该语言使用区间而非多面体模型来表示迭代空间。这限制了 Halide 的表达力。例如，与 TIRAMISU 不同，Halide 无法自然地表示非矩形迭代空间，这也是分布式 Halide 在生成分布式代码时过高估计通信（发送和接收）数据量的原因。这也会使某些 Halide pass 过高估计非矩形迭代空间，从而可能导致代码效率降低（例如，它阻止 Halide 对非矩形迭代空间执行精确的边界推断）。使用区间还阻止了 Halide 执行许多复杂的仿射变换，例如迭代空间倾斜。

Halide 没有依赖关系分析，依靠保守的规则来确定调度是否合法。例如，如果第二个循环读取第一个循环产生的值，则 Halide 不允许两个循环的合并（使用 `compute_with` 指令）。尽管此规则避免了非法的循环融合，但也阻止了很多合法的情形，从而导致性能欠佳。Halide 还假定该程序具有非循环的数据流图，以便简化检查调度的合法性。这样组织了用户使用循环的数据流来表达许多程序。在某些情况下可以绕过上述限制，但是这种解决方法并不通用。TIRAMISU 通过依赖分析来检查代码转换的正确性，从而避免了过度保守的约束，从而实现了更多可能的调度。

Vocke 等人将 Halide 扩展到 DSP，并添加调度指令（如 `store_in`）以指定应在哪个存储器层次结构中存储数据。TVM 是另一个与 Halide 有许多相似之处的系统。它在内部使用修改的 Halide IR。由于 TVM 还是非多面体编译器，因使用多面体模型而导致的 Halide 和 TIRAMISU 之间的差异也适用于 TVM。POET 是一个使用基于 XML 来描述代码和转换行为来参数化循环转换的系统。它使用的转换比 TIRAMISU 中使用的多面体转换要少。GraphIt 是另一种具有调度语言的编译器，但它主要用于图形应用程序领域。

(D) 其他编译器：Delite 是用于构建 DSL 编译器的通用框架。它提供了几种 DSL 可用于表达并行性的并行计算模式。NOVA 和 Lift 是 DSL 编译器的 IR。它们是函数式语言，依赖于一组更高阶的函数，例如 map, reduce 和 scan 以表示并行性。TIRAMISU 是这些框架的补充，因为 TIRAMISU 允许复杂的仿射变换，这些变换在多面体模型中更容易表达。

A.3 TIRAMISU 的嵌入式 DSL

TIRAMISU 是嵌在 C++ 中的特定领域语言 (DSL)。它提供了 C++ API，允许用户编写高层的，体系结构无关的算法，以及一组指导代码生成的调度指令。输入的 TIRAMISU 代码既可以由程序员直接编写，也可以由其他 DSL 编译器生成。然后，TIRAMISU 构造一个高层的中间表示 (IR)，应用用户指定的循环和数据布局转换，并生成利用目标硬件功能的优化后端代码。(多核和分布式系统使用 LLVM IR, GPU 使用 LLVM IR 和 CUDA)。

(A) TIRAMISU 的适用范围

TIRAMISU 设计用于表达数据并行算法，尤其是那些使用嵌套循环和语句序列在密集数组上运行的算法。这些算法经常出现在图像处理，深度学习，密集线性代数，张量运算和 stencil 计算领域。

(B) 描述算法

TIRAMISU 程序的第一部分描述了算法，而没有指定循环优化 (何时何地地进行计算)，数据布局 (数据应如何存储在内存中) 或通信。在这一级上，没有数据位置的概念，而是通过明确的生产者 - 消费者关系来传递值。算法是具有输入，输出并由一系列计算组成的纯函数。TIRAMISU 中用计算 (computation) 来表示一个语句。围绕计算的流控制仅限于循环和条件，而无法表示 while 循环，提前 break 和 goto。要声明计算，用户要提供计算的迭代域和要计算的表达式。

```
1 // Declare the iterators i, j and c.
2 Var i(0, N-2), j(0, M-2), c(0, 3);
3
4 Computation bx(i, j, c), by(i, j, c);
5
6 // Algorithm.
7 bx(i, j, c) = (in(i, j, c)+in(i, j+1, c)+in(i, j+2, c))/3;
8 by(i, j, c) = (bx(i, j, c)+bx(i+1, j, c)+bx(i+2, j, c))/3;
```

图 A.2 模糊 (blur) 算法，无调度指令

图 A.2 显示了用 TIRAMISU 编写的模糊 (blur) 算法。该算法声明了两个计算 bx 和 by。第一个计算 bx 计算输入的水平模糊，而第二个计算 by 通过第一阶

段的输出的平均值来计算最终模糊。第 2 行中的迭代器 i , j 和 c 定义 bx 和 by 的迭代域 (为简便起见, 我们忽略边界条件)。该算法在语义上等效于以下代码:

```
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      bx[i][j][c] = (in[i][j][c]+in[i][j+1][c]+in[i][j+2][c])/3
for (i in 0..N-2)
  for (j in 0..M-2)
    for (c in 0..3)
      by[i][j][c] = (bx[i][j][c]+bx[i+1][j][c]+bx[i+2][j][c])/3
```

(C) 调度指令

TIRAMISU 提供了一组用于通用优化的高级调度指令。表 2 展示了一些例子。调度指令有四种类型:

- 用于循环转换的指令: 包括常见的仿射转换, 例如循环分割, 分割, 移位等。例如, 可以通过调用 `C.tile(i, j, 32, 32, i0, j0, i1, j1)`, 其中 i 和 j 是原始循环迭代器, $i0$, $j0$, $i1$ 和 $j1$ 是分块后的循环迭代器。
- 用于将循环映射到硬件的指令: 包括循环并行化, 矢量化以及将一层循环映射到 GPU 块或 GPU 线程。例如, 调用 `C.vectorize(j, 4)` 会将 j 循环除以 4, 并将内部循环映射到矢量通道。
- 用于操作数据的指令: 包括 (1) 申请数组; (2) 设置数组属性, 包括 GPU 中数组是存储在 `host`, `device`, `shared` 或 `local` 内存中; (3) 在不同存储器层次结构之间或节点之间复制数据; (4) 设置数组访问。多数情况下, 用户只需要使用高级指令操作数据。如果高级指令表达能力不够, 用户可以使用更具表达能力的低级指令。
- 用于添加同步操作的指令: 用户可以声明同步障碍, 也可以使用发送和接收函数进行点对点同步。

TIRAMISU 引入了新颖的调度指令指令, 它们包括申请数组, 在不同存储器层次结构之间复制数据, 在节点之间发送和接收数据以及同步。调用 `cache_shared_at`, `cache_local_at`, `allocate_at`, `copy_at`, `barrier_at` 会返回可以像其他任何计算一样进行调度的操作 (TIRAMISU 中操作是一种特殊的计算, 不返回任何值)。GPU 上 `cache_shared_at` 和 `cache_local_at` 可用于为缓冲区创建缓存。它们自动计算需要缓存的数据量, 执行数据复制并插入任何必要的同步。

通过使用 `allocate_at`, `copy_at` 和 `barrier_at`, TIRAMISU 可以自动计算用于数据复制, 分配和同步操作的迭代域。这很重要, 因为它使用户不必手动猜测或计算迭代域, 特别是在探索不同的可能调度时。例如, 考虑在 GPU 上执行的嵌套

循环中将缓冲区从全局内存复制到共享内存。要复制的区域的大小以及复制操作本身的迭代域（在这种情况下是一个简单的赋值）取决于循环是否分块，分块大小以及是否应用任何其他循环转换。TIRAMISU 根据调度自动计算迭代域和要复制的数据区域，从而简化这个步骤。

为了说明更多 TIRAMISU 调度指令，我们再次使用图 A.2 中的模糊示例，并将它映射到在多核体系结构上执行。必要的调度指令如图 A.3 -(a)(左) 所示。tile 指令对计算进行分块。compute_at 指令在循环 j0 处计算 by 所需的 bx 分块。在这种情况下这种转换引入了冗余计算，这被称为重叠分块。parallelize 指令并行化 i0 循环。

我们使用相同的示例，但将 bx 和 by 的两个最外层循环映射到 GPU。必要的调度指令如图 A.3 -(b)(左) 所示。tile_gpu 指令将 by 的计算分块，然后将新循环映射到 GPU 块和线程。compute_at 指令在循环 j0 处计算 by 所需的 bx 分块（这会引入冗余计算）。cache_shared_at 指示 TIRAMISU 将 bx 计算的结果存储在共享内存中。从全局内存复制到共享内存将在 by 的循环 j0 处完成。随后的 store_in 指令指定 bx 和 by 的访问函数。在这个例子中，store_in 把计算结果存储在 SOA（数组的结构体）数据布局中，从而允许合并的访问。最后的指令创建 host 到 device 和 device 到 host 的数据复制操作并调度它们。

假设我们要在分布式系统上运行这个例子，该系统上有 Nodes 个多核 CPU 节点。图 A.3 -(c)(左) 显示了在这种情况下要使用的调度指令。假设数组 in[][][] 最初分布在各个节点上，节点 n 具有由 $in[n*((N-2)/Nodes)..(n+1)*((N-2)/Nodes),*,*]$ 中的数据。换句话说，这对应于行 $n*(N-2)/Nodes$ 到行 $(n+1)*((N-2)/Nodes)$ 。该块存储在本地数组 lin[][][] 中。

send 和 recv 定义边界区域的通信。假设每个节点都有 in 中的一块。对节点 n 中存储的块的模糊计算需要从节点 n+1 中存储的块的前两行数据开始，这两行称为边界区域。Send 将从节点 is 发送从 lin(0, 0, 0) 开始的两行数据（ $M \times 2 \times 3$ 个连续数据）到节点 is-1，这些数据对应于节点 is 的 in 中的部分的前两行。相应地，节点 ir 的 recv 将从节点 ir+1 接收 2 行（ $M \times 2 \times 3$ 连续数据元素），这对应于 ir 从节点 ir+1 接收前两行。节点 ir 将这些元素从 lin(N, 0, 0) 开始放置在其本地块的末尾。此外，ASYNC 定义异步发送，SYNC 定义同步接收。最后，我们标记要分布的循环（bx, by, s 和 r 的外层循环），即将每次迭代标记为在不同的节点上运行。

只要语义正确，TIRAMISU 中的所有其他调度指令都可以由 send, recv 和分布的循环组成。

TIRAMISU Scheduling Commands	Pseudocode Representing Code Generated by TIRAMISU
<pre> 1 // Scheduling commands for targeting 2 // a multicore architecture. 3 4 // Tiling and parallelization. 5 Var i0, j0, i1, j1; 6 by.tile(i, j, 32, 32, i0, j0, i1, j1); 7 by.parallelize(i0); 8 bx.compute_at(by, j0); </pre>	<pre> 1 2 Parallel for(i0 in 0..floor((N-2)/32)) 3 for(j0 in 0..floor((M-2)/32)) 4 bx[32,34,3]; 5 // Tiling with redundancy 6 for(i1 in 0..min((N-2)%32,32)+2) 7 for(j1 in 0..min((M-2)%32,32)+2) 8 int i = i0*32+i1 9 int j = j0*32+j1 10 for (c in 0..3) 11 bx[i1][j1][c]= 12 (in[i][j][c] + in[i][j+1][c] 13 + in[i][j+2][c])/3 14 15 for(i1 in 0..min(N-2,32)) 16 for(j1 in 0..min(M-2,32)) 17 int i = i0*32+i1 18 int j = j0*32+j1 19 for (c in 0..3) 20 by[i][j][c]= 21 (bx[i][j][c] + bx[i+1][j][c] 22 + bx[i+2][j][c])/3 23 24 25 26 </pre>
<pre> 1 // Scheduling commands for targeting GPU. 2 // Tile i and j and map the resulting dimensions 3 // to GPU 4 Var i0, j0, i1, j1; 5 by.tile_gpu(i, j, 32, 32, i0, j0, i1, j1); 6 bx.compute_at(by, j0); 7 bx.cache_shared_at(by, j0); 8 9 // Use struct-of-array data layout 10 // for bx and by. 11 bx.store_in({c,i,j}); 12 by.store_in({c,i,j}); 13 14 // Create data copy operations 15 operation cp1 = in.host_to_device(); 16 operation cp2 = by.device_to_host(); 17 18 // Specify the order of execution of copies 19 cp1.before(bx, root); 20 cp2.after(by, root); </pre>	<pre> 1 host_to_device_copy(in_host, in); 2 3 GPUBlock for(i0 in 0..floor((N-2)/32)) 4 GPUBlock for(j0 in 0..floor((M-2)/32)) 5 shared bx[3,32,34]; 6 // Tiling with redundancy 7 GPUBlock for(i1 in 0..min((N-2)%32,32)+2) 8 GPUBlock for(j1 in 0..min((M-2)%32,32)+2) 9 int i = i0*32+i1 10 int j = j0*32+j1 11 for (c in 0..3) 12 bx[c][i1][j1]= 13 (in[i][j][c] + in[i][j+1][c] 14 + in[i][j+2][c])/3 15 16 GPUBlock for(i1 in 0..min(N-2,32)) 17 GPUBlock for(j1 in 0..min(M-2,32)) 18 int i = i0*32+i1 19 int j = j0*32+j1 20 for (c in 0..3) 21 by[c][i][j]= 22 (bx[c][i][j] + bx[c][i+1][j] 23 + bx[c][i+2][j])/3 24 25 26 device_to_host_copy(by, by_host); 27 28 // We assume that in[][][] is initially 29 // distributed across nodes. Each node 30 // has a chunk of the original 31 // in[][][] that we call lin[][]]. 32 33 // Start by exchanging border rows of 34 // lin[][]] 35 distributed for (is in 1..Nodes) 36 send(lin(0,0,0), M*2*3, is-1, {ASYNC}) 37 distributed for (ir in 0..Nodes-1) 38 recv(lin(N,0,0), M*2*3, ir+1, {SYNC}) 39 40 distributed for (i0 in 0..Nodes) 41 parallel for (i1 in 0..(N-2)/Nodes) 42 for (j in 0..M-2) 43 for (c in 0..3) 44 bx[i][j][c] = 45 (lin[i][j][c] + lin[i][j+1][c] 46 + lin[i][j+2][c])/3 47 48 distributed for (i0 in 0..Nodes) 49 parallel for (i1 in 0..(N-2)/Nodes) 50 for (j in 0..M-2) 51 for (c in 0..3) 52 by[i][j][c] = 53 (bx[i][j][c] + bx[i+1][j][c] 54 + bx[i+2][j][c])/3 55 56 // We assume that no gather operation on 57 // by[][][] is needed </pre>
<pre> 1 // Scheduling commands for targeting 2 // a distributed system 3 4 // Declare additional iterators 5 Var is(1, Nodes), ir(0,Nodes-1), i0, i1; 6 7 // Split loop i into loops i0 and i1 and 8 // parallelize i1 9 bx.split(i,N/Ranks,i0,i1); bx.parallelize(i1); 10 by.split(i,N/Ranks,i0,i1); by.parallelize(i1); 11 12 // Communicate the border rows where necessary 13 send s = 14 send({is}, lin(0,0,0), M*2*3, is-1, {ASYNC}); 15 recv r = 16 receive({ir}, lin(N,0,0), M*2*3, ir+1, {SYNC},s); 17 18 // Order execution 19 s.before(r,root); 20 r.before(bx,root) 21 22 // Distribute the outermost loops 23 bx.distribute(i0); by.distribute(i0); 24 s.distribute(is); r.distribute(ir); </pre>	<pre> 1 2 // We assume that in[][][] is initially 3 // distributed across nodes. Each node 4 // has a chunk of the original 5 // in[][][] that we call lin[][]]. 6 7 // Start by exchanging border rows of 8 // lin[][]] 9 distributed for (is in 1..Nodes) 10 send(lin(0,0,0), M*2*3, is-1, {ASYNC}) 11 distributed for (ir in 0..Nodes-1) 12 recv(lin(N,0,0), M*2*3, ir+1, {SYNC}) 13 14 distributed for (i0 in 0..Nodes) 15 parallel for (i1 in 0..(N-2)/Nodes) 16 for (j in 0..M-2) 17 for (c in 0..3) 18 bx[i][j][c] = 19 (lin[i][j][c] + lin[i][j+1][c] 20 + lin[i][j+2][c])/3 21 22 distributed for (i0 in 0..Nodes) 23 parallel for (i1 in 0..(N-2)/Nodes) 24 for (j in 0..M-2) 25 for (c in 0..3) 26 by[i][j][c] = 27 (bx[i][j][c] + bx[i+1][j][c] 28 + bx[i+2][j][c])/3 29 30 31 // We assume that no gather operation on 32 // by[][][] is needed </pre>

图 A.3 三个说明 TIRAMISU 调度指令（左）和生成的代码（右）的例子。(a) 展示映射到多核 CPU 的调度指令；(b) 展示映射到 GPU 的调度命令；(c) 展示映射到分布式 CPU 的调度命令。

A.4 TIRAMISU IR

TIRAMISU 的多层 IR 的主要目标是通过按特定顺序应用调度指令来简化调度指令的实现。本节说明了原因，并介绍了 TIRAMISU IR 的各层。

(A) 多层 IR 的意义

在本节中，我们提供一些示例，说明为什么当前的 IR 对 TIRAMISU 来说不够，以及为什么我们需要多层 IR。

当前多数 IR 使用内存在程序语句之间进行通信。这将在程序中创建基于内存的依赖关系，并迫使编译器在决定优化和硬件映射之前选择数据布局。针对不同的硬件架构优化程序通常需要修改数据布局并消除基于内存的依赖性，因为它们会限制优化。因此，必须撤消在调度之前指定的任何数据布局，以允许更大的调度自由，并且代码必须使用最适合目标硬件的数据布局。应用这些数据布局转换和消除基于内存的依赖关系很有挑战性。

另一个说明代码生成复杂性的示例是将缓冲区映射到 GPU 上的共享内存和本地内存。需要复制到共享内存的数据量以及何时执行同步都取决于代码的优化方式（例如代码是否具有两级分块）。这同样适用于在生成分布式代码时确定要发送或接收的数据量。因此，在调度之前，不改应在决定调度之前把缓冲区映射到内存层次结构，管理通信和同步。

TIRAMISU 通过使用多层 IR 解决了代码生成中的这些复杂问题，该 IR 将体系结构独立的算法与循环转换，数据布局和通信完全分开。第一层表示法描述了使用生产者 - 消费者关系而没有存储位置的纯算法。第二层指定计算顺序，以及处理器计算每个值的顺序，这一层适合于执行很多优化而无需处理具体的内存布局。第三层指定在使用中间数据之前将它们存储在何处。第四层添加必要的通信和同步操作。

IR 层次的分离定义了应用优化的特定顺序，并确保某层的编译器 pass 无需担心修改或撤消在较早层中做出的决定。例如，指定计算顺序及位置的阶段可以安全地假定不需要进行数据布局转换。这个简单的假设使 TIRAMISU 不必依赖大量研究数据布局转换以允许调度的研究工作。

(B) 背景

在本节中，我们概述了多面体模型中使用的两个主要概念：整数集合和映射。这两个概念将在后面的部分中使用，以定义不同的 IR 层。

整数集合表示迭代域，而映射用于表示内存访问，转换迭代域和内存访问（应用嵌套循环和内存访问转换）。这些概念的更多细节和正式定义在中提供。整数集合是使用仿射约束描述的一组整数元组，一个例子是：

$$(1, 1); (2, 1); (3, 1); (1, 2); (2, 2); (3, 2)$$

除了像这个集合一样列出所有元组，还可以在循环迭代器和符号常量上使用仿射约束来描述该组，如下所示：

$$S(i, j) : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2$$

其中 i 和 j 是集合中元组的维度。

一个映射是两个整数集合间的关系，例如：

$$S_1(i, j) \rightarrow S_2(i + 2, j + 2) : 1 \leq i \leq 3 \wedge 1 \leq j \leq 2$$

是一个集合 S_1 中的元组和集合 S_2 中的元组间的映射（例如，元组 $S_1(i, j)$ 映射到元组 $S_2(i + 2, j + 2)$ ）

TIRAMISU 中的所有集合和映射都是使用 Integer Set Library (ISL) 实现的。我们还将使用 ISL 中集合和映射的表示方法。

(C) 多层 IR

图 A.4 说明了使用 TIRAMISU 的典型工作流程。用户编写纯算法并提供一组调度指令。然后将 IR 的第一层转换为较低层，最后 TIRAMISU 生成 LLVM 或其他合适的低级 IR。TIRAMISU 使用整数集合表示每层 IR，并使用映射表示迭代域和数据布局上的转换。本节的其余部分描述了 TIRAMISU IR 的四层。

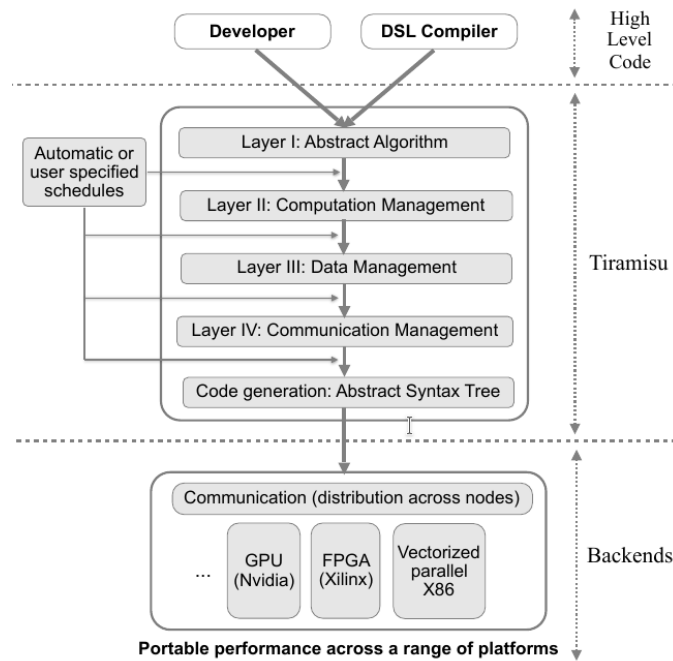


图 A.4 TIRAMISU 架构

(a) 第一层（抽象算法）：

TIRAMISU 的第一层指定了算法，而没有指定何时何地进行计算，如何将数据存储在内存中或进行通信。值通过明确的生产者 - 消费者关系传递。

例如，图 A.2 中用于计算的代码的第一层表示如下：

$$\{by(i, j, c) : 0 \leq i < N - 2 \wedge 0 \leq j < M - 2 \wedge 0 \leq c < 3\} :$$

$$(bx(i, j, c) + bx(i + 1, j, c) + bx(i + 2, j, c))/3$$

第一部分， $\{by(i, j, c) : 0 \leq i < N - 2 \wedge 0 \leq j < M - 2 \wedge 0 \leq c < 3\}$ ，指明了 by 计算的迭代域，而剩余部分是要计算的表达式。迭代域是满足 $0 \leq i < N - 2 \wedge 0 \leq j < M - 2 \wedge 0 \leq c < 3$ 的元组的集合 $by(i, j, c)$ 。第一层中的计算没有顺序，声明顺序不影响第二层中指定的执行顺序。

(b) 第二层（计算管理）：

TIRAMISU 的第二层指定了计算的执行顺序以及其执行的处理器。该层未指定中间结果如何在内存中存储；这些转换不需要执行复杂的数据布局转换，因此简化了优化过程。第一层到第二层的转换通过调度指令自动完成。

图 A.3 -(b)(右) 显示了代码的 GPU 优化版本，由左侧的调度和数据布局指令生成。下面显示了 by 计算的相应第二层表示形式：

$$\{by(1, i0(gpuB), j0(gpuB), i1(gpuT), j1(gpuT), c) :$$

$$i0 = floor(i/32) \wedge j0 = floor(j/32) \wedge i1 = i\%32 \wedge j1 = j\%32 \wedge 0 \leq i < N - 2$$

$$\wedge 0 \leq j < M - 2 \wedge 0 \leq c < 3\} :$$

$$(bx(i0 * 32 + i1, j0 * 32 + j1, c) + bx(i0 * 32 + i1 + 1, j0 * 32 + j1, c) +$$

$$bx(i0 * 32 + i1 + 2, j0 * 32 + j1, c))/3$$

第二层中的计算根据其字典顺序进行排序。冒号之前的集合是计算的有序集合。维度 $i0$ 和 $j0$ 的标签 $gpuB$ 表示每次迭代 ($i0, j0$) 都映射到 GPU 块 ($i0, j0$)。在第二层中，这些元组的顺序决定了执行顺序。

该层中的计算被排序并分配给特定的处理器。顺序由时间维度和空间维度决定。时间维指定相对于其他计算的执行顺序，而空间维指定每个计算在哪个处理器上执行。使用由处理器类型组成的标签将空间维度与时间维度区分开。当前，TIRAMISU 支持以下空间标记：

- **cpu**：这一维在共享内存系统中的 CPU 上运行
- **node**：这一维映射到分布式系统中的节点
- **gpuT**：这一维映射到 gpu 线程
- **gpuB**：这一维映射到 gpu 块

用处理器类型标记维度表示这一维将分布在该类型的处理器上。例如，使用 **cpu** 标记维度将在单独的 CPU 上执行该循环维度的每次迭代。

转换维度的其他标签包括：

- `vec(s)`: 将这一维向量化
- `unroll`: 将这一维循环展开

映射到同一处理器的计算依据投影到时间维度上比较字典顺序来进行排序。

(c) 第三层（数据管理）：

第三层通过指定中间结果的存储位置来明确数据布局。在这一层中还将构造必要的缓冲区分配/释放。`TIRAMISU` 通过应用调度指令进行数据映射来从第二层自动生成该层。

数据管理层指定用于存储计算值的存储位置。它由第二层表示以及分配释放语句和一组访问关系组成，这些访问关系将第二层的计算映射到该计算读取或写入的数组元素。标量被视为单元素数组。对于每个缓冲区，都会创建一个指定缓冲区的类型及其大小的分配语句。同样地，也会创建一个释放语句。

`TIRAMISU` 中可能的数据映射包括将计算映射到结构体的数组，数组的结构体，以及将多维数组压缩为维数较少的数组或标量。还可以指定更复杂的访问，例如将计算 $c(i, j)$ 存储到数组元素 $c(i\%2, j\%2)$ 或 $c(j, i)$ 中。

在图 A.3 -(b)(左)，通过 `by.store_in(c, i, j)` 来设置数据访问，这表示计算 `by(i, j, c)` 的结果保存在数组元素 `by[c, i, j]` 中。这条指令在第三层中生成如下的映射：

$$\{by(1, i0(gpuB), j0(gpuB), i1(gpuT), j1(gpuT), c) \rightarrow by[c, i0 * 32 + i1, j0 * 32 + j1] : \\ i0 = floor(i/32) \wedge j0 = floor(j/32) \wedge i1 = i\%32 \wedge j1 = j\%32 \wedge 0 \leq i < N - 2 \\ \wedge 0 \leq j < M - 2 \wedge 0 \leq c < 3\}$$

`TIRAMISU` 中的数据映射是一个把计算映射到缓冲区元素的仿射关系。`TIRAMISU` 允许任何仿射关系可以表达的数据映射。

(d) 第四层（通信管理）：

第四层将同步和通信操作添加到表示中，将它们映射到时空域，并明确何时发生用于缓冲区分配/释放的语句。该层是通过应用用户指令从第三层自动生成的。第三层中添加的任何分配/释放操作也都映射到该层中的时空域。

A.5 编译器实现

由于本文的主要贡献不是在介绍用于代码生成的新技术，因此我们仅概述 `TIRAMISU` 如何生成 IR 层和目标代码。在本节中，我们会向读者介绍适当的文献以提供更多详细信息。

在本节的其余部分中，我们描述调度指令如何转换这些 IR 层次。我们还将描述如何从第四层生成目标代码。

(A) 将第一层转换为第二层：

使用两种调度命令将第一层转换为第二层：(1) 用于循环嵌套转换的命令（例如 `tile`, `split`, `shift`, `interchange`）；(2) 用于将循环映射到硬件的命令（包括 `parallelize`, `vectorize`, `gpu`）。

第一种调度命令应用映射来转换迭代域。例如，在图 A.2 中的 `by` 计算上应用分块命令时，它会转换为如下映射：

$$\{by(i, j, c) \rightarrow by(i0, j0, i1, j1, c) : i0 = \text{floor}(i/32) \wedge i1 = i\%32 \wedge j0 = \text{floor}(j/32) \wedge j1 = j\%32 \wedge 0 \leq i < N \wedge 0 \leq j < N\}$$

然后将这个映射应用于第一层 IR，从而生成第二层 IR。合成变换是通过组合多个映射来完成的，因为两个仿射映射的组合仍是仿射映射。

第二种类型的命令在尺寸上添加了空间标记，以指示循环要并行化，向量化，映射到 GPU 块等。

(B) 将第二层转换为第三层：

这是通过使用访问关系扩展第二层来完成的。默认情况下，TIRAMISU 使用恒等访问关系（即将计算 $C(i, j)$ 存储到缓冲区 $C[i, j]$ 中的访问关系）。如果使用 `store_in` 命令，则从该命令推导访问关系。在将第二层转换为第三层时，还会添加缓冲区分配。调度命令 `b.allocate_at(C, i)` 创建一条新语句，在计算 C 的嵌套循环的循环 i 中分配缓冲区 b 。

(C) 将第三层转换为第四层：

将用于数据通信（发送和接收），同步以及用于在全局，共享和本地内存之间复制数据的调度命令转换为语句。例如，`send` 和 `receive` 命令被转换为函数调用，这些函数调用将在代码生成过程中转换为 MPI 调用。

(A) 代码生成

从第四层中的一组计算生成代码等于生成恰好访问该组中的每个计算各一次，同时遵循计算之间的字典顺序的嵌套循环。TIRAMISU 依赖于 ISL 库提供的 Cloog 代码生成算法的实现。TIRAMISU 代码生成器接受第四层 IR 并生成抽象语法树（AST）。然后遍历 AST 以生成用于特定硬件体系结构的低级代码（取决于目标后端）。

多核 CPU 代码生成器从 AST 生成 LLVM IR。为了生成 LLVM IR，我们使用 Halide 作为库：首先生成 Halide IR，然后使用 Halide 将 Halide IR lower 到 LLVM

IR。我们不使用 Halide 执行任何高级代码优化。在生成 Halide IR 之前, TIRAMISU 执行所有代码。然后, Halide 编译器将 Halide IR lower 到 LLVM IR。

GPU 代码生成器生成 LLVM IR 作为 host 代码, 生成 CUDA 作为 kernel 代码。数据复制命令和有关缓冲区存储位置(共享, 常量或全局内存)的信息均在第四层中提供。TIRAMISU 将它们转换为生成的代码中等效的 CUDA 数据复制调用和缓冲区分配。标有 GPU 线程或 GPU 块标签的循环维度将转换为低级代码中合适的 GPU 线程和块 ID。TIRAMISU 代码生成器可以生成合并的数组访问, 并可以使用共享和常量内存。通过将完整分块(大小为分块大小倍数的循环嵌套)与部分分块(循环的其余部分)分开, 也可以避免线程发散。

分布式存储系统的代码生成器利用 MPI。在代码生成期间, 用于数据复制的函数调用都转换为等效的 MPI 函数调用。生成的代码经过后处理, 每个分布式循环都被转化为基于进程的 MPI 编号的条件语句。例如:

```
for (q in 1..N-1) {...} // distribute on q
```

转化为

```
q = get_rank(); if (q ≥ 1 and q < N-1) {...}
```

B. 支持非仿射的迭代空间

TIRAMISU 用类似于 Benabderrahmane 等人的方式表示非仿射的数组访问, 循环边界和条件语句。例如, 将条件语句转换附加到计算中的条件。计算的访问是条件语句的两个分支中计算的访问的并集, 这是一个过度估计。在代码生成期间, 预处理步骤将条件重新插入到生成的代码中。Benabderrahmane 等人证明了这些技术的效率, 并在 PENCIL 编译器中得到确认。我们的一般经验以及本文中的实验表明, 这些近似不会损害性能。

A.6 性能评估

我们根据两组基准评估 TIRAMISU。首先是一组深度学习和线性代数基准。第二个是一组图像处理基准。

我们在 16 个节点的群集上进行评估。每个节点都是双插槽计算机, 具有两个 24 核的 Intel Xeon E5- 2680v3 CPU, 128 GB RAM, Ubuntu 14.04 和 Infiniband 互连。我们使用 MPI 的 MVAPICH2 2.0 实现进行分布式测试。在这些节点之一上执行多核实验 (CPU)。GPU 实验是在具有 12 GB RAM 的 NVIDIA Tesla K40 上执行的。每个实验重复 30 次, 并报告中位时间。

(A) 深度学习和线性代数基准

我们通过实现一组深度学习和线性代数基准测试来评估 TIRAMISU，包括 Conv（神经网络卷积层的直接实现），VGG（VGG 神经网络的一个块）和 sgemm（用于实现卷积的矩阵乘法），HPCG（多网格预处理共轭梯度 (conjugate gradient, CG) 的基准）和 Baryon（用于构造 Baryon 块的密集张量收缩代码）。对于所有这些基准，我们将 TIRAMISU 实现与 Intel MKL 进行了比较，但 HPCG 和 Baryon 除外，对于这两个我们将 TIRAMISU 与参考实现进行了比较。图 A.5 显示了 TIRAMISU 生成的 CPU 代码与参考实现之间的性能比较。对于 sgemm 和 HPCG，我们使用大小为 1060×1060 的矩阵和大小为 1060 的向量。对于 Conv 和 VGG，我们使用 512×512 作为数据输入大小，使用 16 作为输入/输出特征的数量，batch 大小为 32。对于 Baryon，我们使用与参考代码中相同的张量大小。

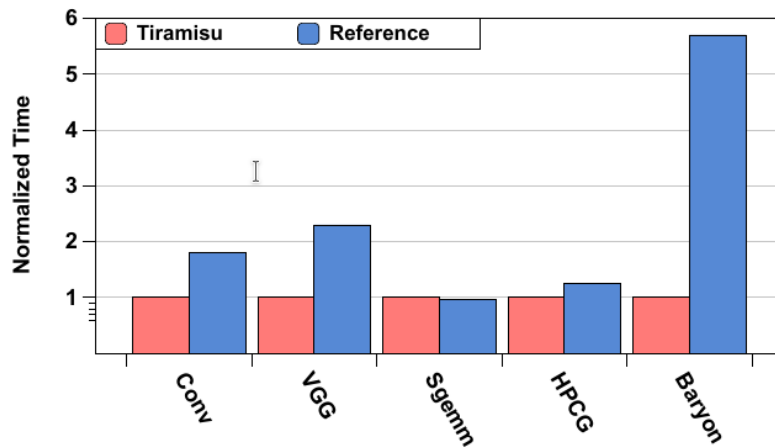


图 A.5 深度学习，线性代数和张量代数基准测试的归一化执行时间

sgemm 上 TIRAMISU 的性能与 Intel MKL 相当。sgemm 有趣的点在于该 kernel 的 Intel MKL 实现以其手动优化的性能而闻名。我们使用了大量优化来赶上英特尔 MKL。这些优化包括三维 sgemm 循环的两级分块，矢量化，循环展开，数组打包，寄存器分块以及完整和部分分块的分离（这对于实现矢量化，循环展开和减少控制开销至关重要。我们还使用自动调优为运行实验的机器找到了最佳的分块尺寸和展开系数。

对于 Conv kernel，TIRAMISU 生成的代码使用固定大小的卷积滤波器，因此 TIRAMISU 优于 Intel MKL 实现。我们为常见的卷积滤波器大小 (3×3 、 5×5 、 7×7 、 9×9 和 11×11) 生成专用版本。这使 TIRAMISU 编译器可以应用英特尔 MKL 无法执行的优化。例如，这允许 TIRAMISU 展开最内层的（卷积滤波器）循环，因

为在编译时它们的大小已知。在 VGG 中，TIRAMISU 将 VGG 块的两个卷积环融合在一起，从而改善了数据局部性。另外，我们为卷积滤波器生成具有固定大小的代码（就像在 Conv 中所做的那样）。与 Intel MKL 相比，这可提供 2.3 倍的加速。TIRAMISU 在 Baryon 参考代码上的加速是通过矢量化实现的，但是这种矢量化并不平凡，因为它需要应用数组扩展，然后使用散布/聚集操作，这两个操作均未在参考的 Baryon 代码中实现。

(B) 图像处理基准

我们在评估中使用了以下图像处理基准：edgeDetector，进行环形模糊，然后进行 Roberts 边缘检测；cvtColor，将 RGB 图像转换为灰度；conv2D，一个简单的 2D 卷积；warpAffine，它会在图像上进行仿射变形；gaussian，执行高斯模糊；nb，由 4 个阶段组成的合成流水线，可根据同一输入图像计算负片和增亮图像；ticket #2373，这是针对 Halide 的一个错误的代码片段。该代码仅有一个为数组赋值的循环，但是迭代空间不是矩形的（它测试 $x \geq r$ ，其中 x 和 r 是循环迭代器）。此代码中的推断的范围是过度估计，导致生成的代码因为执行期断言而无法运行。这些基准中的四个具有非仿射数组访问和用于 clamping（以处理边界情况）的非仿射条件：edgeDetector，conv2D，warpAffine 和 gaussian。我们使用 2112×3520 RGB 输入图像进行实验。

我们将 TIRAMISU 与其他两个编译器进行了比较：Halide，一种具有调度语言的工业级图像处理 DSL，以及 PENCIL，一种最新的全自动多面体编译器。

图 A.6 将 TIRAMISU 生成的代码的归一化的执行时间在三种架构上（单节点多核 CPU，GPU 和分布式系统（16 节点））与其他最新框架进行了比较。对于单节点多核 CPU 和 GPU，我们将 TIRAMISU 与 Halide 和 PENCIL 进行了比较。对于分布式系统，我们将其与分布式 Halide 进行比较。

(a) 单节点多核 CPU：

在四个基准测试中，TIRAMISU 生成的代码的性能与 Halide 的性能相当。两种实现方式使用相同的调度，调度是由 Halide 专家手写的。具有非仿射数组访问和条件的 edgeDetector，conv2D，warpAffine 和 gaussian 的结果表明，TIRAMISU 有效地处理了此类访问模式。

其他两个基准（edgeDetector 和 ticket #2373）无法在 Halide 中实现。以下代码片段展示了 edgeDetector：

Architectures	Frameworks	Benchmarks						
		edge Detector	cvtColor	Conv2D	warp Affine	gaussian	nb	ticket #2373
Single-node multicore	Tiramisu	1	1	1	1	1	1	1
	Halide	-	1	1	1	1	3.77	-
	PENCIL	2.43	2.39	11.82	10.2	5.82	1	1
GPU	Tiramisu	1.05	1	1	1	1	1	1
	Halide	-	1	1.3	1	1.3	1.7	-
	PENCIL	1	1	1.33	1	1.2	1.02	1
Distributed (16 Nodes)	Tiramisu	1	1	1	1	1	1	1
	Dist-Halide	-	1.31	3.25	2.54	1.57	1.45	-

图 A.6 将 TIRAMISU 生成的代码与其他框架的归一化执行时间进行比较（越低越好）。比较在以下三种架构上进行：多核 CPU，GPU，分布式 CPU（16 个节点）。“-”表示不支持

```

/* Ring Blur Filter */
R(i, j) = (Img(i-1, j-1) + Img(i-1, j) + Img(i-1, j+1) + Img(i, j-1) +
  Img(i, j+1) + Img(i+1, j-1) + Img(i+1, j) + Img(i+1, j+1)) / 8
/* Roberts Edge Detection Filter */
Img(i, j) = abs(R(i, j) - R(i+1, j-1)) + abs(R(i+1, j) - R(i, j-1))

```

edgeDetector 创建一个循环长度 ≥ 1 的循环依赖图（R 在第一条语句中写入并在第二条语句中读取，而 Img 在第二条语句中写入并在第一条语句中读取），但是 Halide 只能表达具有非循环依赖图的程序，尽管有一些例外；Halide 语言和编译器施加了此限制，以避免需要证明某些优化的合法性（因为在 Halide 基于区间的表示中很难证明某些优化的合法性）。TIRAMISU 没有此限制，因为它使用依赖性分析检查了转换合法性。

在具有三角形迭代域的 ticket #2373 中，Halide 的边界推断过度估计了计算的边界，这导致生成的代码无法执行。Halide 中的这种过度逼近是由于使用区间来表示迭代域，这阻止了 Halide 对非矩形迭代空间执行精确的边界推断。TIRAMISU 可以自然地处理这种情况，因为它依赖于多面体模型，其中集合除了可以包含循环边界外还可以包含任何仿射约束。这些示例表明，与高级，成熟的 DSL 编译器相比，TIRAMISU 提供的模型自然支持更复杂的代码模式。

对于 nb，由 TIRAMISU 生成的代码比 Halide 生成的代码提高了 3.77 倍。这主要是由于循环融合。在此代码中，TIRAMISU 通过将循环融合为一个循环来增强数据局部性。这在 Halide 中是不可能的，如果它们更新相同的缓冲区，则它们无法融合循环。Halide 做出了这种保守的假设，因为否则无法证明融合是合法的。TIRAMISU 并非如此，而是使用依赖分析来证明正确性。

PENCIL 编译器在 gaussian 上的速度降低是由于 PENCIL 做出的次优决策。

gaussian kernel 由两个连续的嵌套循环组成（每个嵌套循环包含三个循环层次）。PENCIL 决定互换两个最内层的循环级别，以实现两个连续循环嵌套的融合。该决定最大程度地减少了生产者 and 使用者语句（两个嵌套循环）之间的距离，但它导致不连续的内存访问，因此减少了空间局部性。在这种情况下，正确的决定是一个权衡。PENCIL 中使用的 Pluto 自动调度算法无法感知这种权衡。对于其他 kernel，TIRAMISU 和 Halide 都在最内层的循环上应用矢量化和循环展开，而 PENCIL 则不这样做，因为 PENCIL 的多核代码生成器未实现这两个优化。对于 warpAffine，TIRAMISU 和 Halide 的速度都比 PENCIL 高，这是因为此基准测试中的唯一嵌套循环具有 25 条语句，并且对最内层的循环进行矢量化可将所有这些语句转换为它们的矢量等效形式，同时循环展开会增加寄存器重用和测试机的 24 个核心上的指令级并行度。

(b) GPU

对于 GPU 后端，报告的时间是总执行时间（复制数据和执行 kernel）。TIRAMISU 为 conv2D 和 gaussian 生成的代码比 Halide 快，因为 TIRAMISU 生成的代码使用常量内存来存储权重数组，而当前版本的 Halide 对其 PTX 后端不使用常量内存。在这些基准测试中，TIRAMISU 和 Halide 的调度之间的唯一区别是 TIRAMISU 中使用了 tag_gpu_constant。对于所有过滤器，TIRAMISU 和 Halide 的数据复制时间相同。对于 nb，由于 TIRAMISU 能够应用循环融合，而 Halide 无法应用，因此 TIRAMISU 生成的代码比 Halide 生成的代码快 1.7 倍。

与 PENCIL 相比，conv2D 和 gaussian 加速的原因是 PENCIL 在 CUDA kernel 中生成不必要的复杂控制流，从而导致线程发散。

(c) 分布式系统

我们假设数据已经按行分布在节点上。在这些基准测试中，nb，cvtColor 和 ticket #2373 不需要任何通信。由于分布数据中边界区域的重叠，其他四个需要通信。

图 6 比较了分布式 TIRAMISU 和分布式 Halide 的执行时间。在每种情况下，TIRAMISU 都比分布式 Halide 要快。对于 conv2D，它可以达到 3.25 倍的加速。对于涉及通信的 kernel，与 TIRAMISU 相比，分布式 Halide 生成的代码有两个问题：分布式 Halide 高估了它需要发送的数据量，并且在发送之前不必要地将连续的数据打包到一个单独的缓冲区中。分布式 Halide 高估了它需要发送的数据量，因为基准测试具有无法静态分析的数组访问权限（对数组访问进行了 clamp 以处理边界情况），因此分布式 Halide 无法计算要发送的确切数据量。为避免此

问题，TIRAMISU 使用 send 和 receive 调度命令使用显式通信。这两个命令的使用是 TIRAMISU 和分布式 Halide 之间的唯一区别。这些命令允许用户精确指定要发送的数据量，并且还允许编译器避免不必要的打包。

图 A.7 显示了在 2、4、8 和 16 个节点上运行分布式 TIRAMISU 的 kernel 的加速情况。该图显示随着节点数量的增加，TIRAMISU 生成的分布式代码可以很好地拓展（强可拓展）。

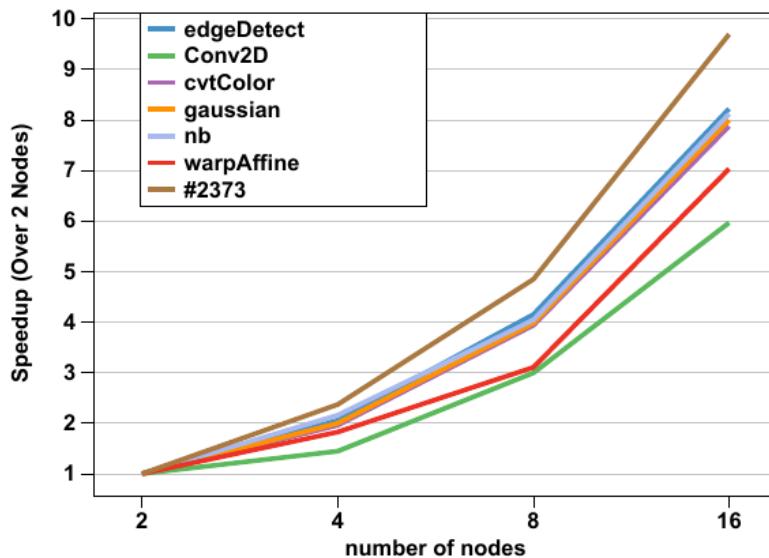


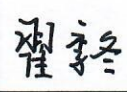
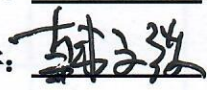

图 A.7 TIRAMISU 为 2, 4, 8, 16 个节点生成的代码的加速比。基准执行时间是 2 个节点的

注: $\text{clamp}(i, 0, N)$ 当 $i < 0$ 时取 0, 当 $i > N$ 时取 N , 否则取 i

A.7 总结

本文介绍了 TIRAMISU, 一个多面体编译器框架, 具有调度语言, 该调度语言带有针对多核 CPU, GPU 和分布式系统的命令。编译器使用四层 IR, 用于分隔算法, 何时何地计算, 数据布局和通信。我们以各种后端为目标对 TIRAMISU 进行了评估, 并演示了它可生成与最新框架和手工优化的代码性能相当或更好的代码。

综合论文训练记录表

学生姓名	李晨昊	学号	2017011466	班级	计 72
论文题目	PLANT: 基于多面体模型的张量编译器				
主要内容以及进度安排	<p>实现人工调度的基于多面体模型的张量编译器，向用户提供调度指令。同时借鉴非多面体编译器中的自动化方法，实现基于模板的自动调度器。</p> <p>进度安排：</p> <p>一月：构建系统，实现大部分关键调度指令和 CPU 代码生成</p> <p>二月：实现大部分调度指令和 GPU 代码生成，测试运行简单 kernel</p> <p>三月：实现自动调度器，测试运行一些有代表性的网络模型</p> <p>四、五月：性能测试及调优，论文撰写</p> <div style="text-align: right; margin-top: 20px;"> <p>指导教师签字： </p> <p>考核组组长签字： </p> <p>2021 年 1 月 5 日</p> </div>				
中期考核意见	<p>工作进展顺利，希望做好后续安排，高质量完成综合论文训练。</p> <div style="text-align: right; margin-top: 20px;"> <p>考核组组长签字： </p> <p>2021 年 4 月 8 日</p> </div>				

<p style="writing-mode: vertical-rl; text-orientation: upright;">指导教师评语</p>	<p>张量计算在包括人工智能、量子计算等很多领域具有重要意义。本文采用多面体模型实现了一个基础张量编译器。本文利用多面体模型表达复杂的程序语义，为用户提供精细的调度指令，在中间表示上分层调度，便于用户显式地管理硬件的复杂性，为后端生成高效的代码。在算子性能和神经网络推理性能测试中，实现的系统优于包括手工调优的算子库，典型的张量编译器在内的常见参考平台。多面体模型本身理论具有一定的复杂性，本文的工作能够把多面体模型用于复杂芯片的代码生成，工作本身具有一定的复杂度。本文工作扎实，逻辑清楚，达到本科毕业设计的要求，是一篇优秀的论文。</p> <p style="text-align: right;">指导教师签字: <u>翟宇</u></p> <p style="text-align: right;">2021年6月10日</p>
<p style="writing-mode: vertical-rl; text-orientation: upright;">评阅教师评语</p>	<p>本文研究了张量计算代码自动优化的问题，基于多面体模型设计实现了一个张量编译器，采用分级设计的思想将张量计算代码分为算法描述、计算调度、内存调度三层，并在此基础上通过搜索算法实现自动调优。实验表明，本文工作能够取得比已有的算子库和深度学习框架更优的计算性能。本工作选题具有重要意义，设计合理，论文撰写符合规范，很好地完成了综合论文训练的要求。</p> <p style="text-align: right;">评阅教师签字: <u>韩文波</u></p> <p style="text-align: right;">2021年6月10日</p>
<p style="writing-mode: vertical-rl; text-orientation: upright;">答辩小组评语</p>	<p>本文研究工作选题具有重要意义，设计合理，实验方法正确，效果明显，论文撰写符合规范，答辩陈述清晰，是一项优秀的本科综合论文训练工作。</p> <p style="text-align: right;">答辩小组组长签字: <u>韩文波</u></p> <p style="text-align: right;">2021年6月10日</p>

总成绩: A+

教学负责人签字: 翟宇

2021年6月10日