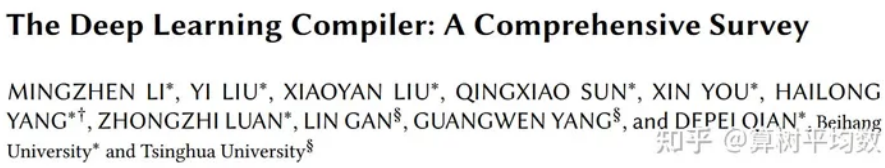
<四> 深度学习编译器综述：Low-Level IR

<四> 深度学习编译器综述：Low-Level IR

算树平均数 算树平均数 昏昏沉沉工程师 (寻职中) 关注他

The Deep Learning Compiler: A Comprehensive Survey

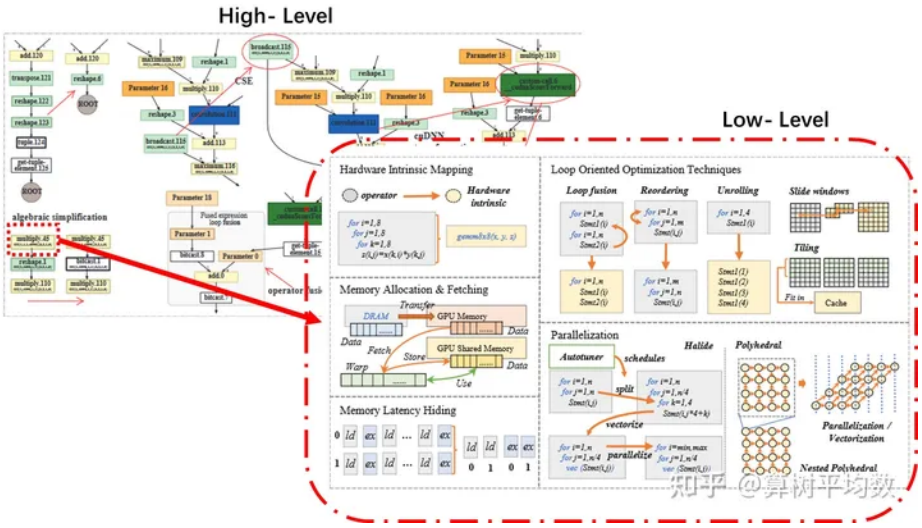

The Deep Learning Compiler: A Comprehensive Survey

MINGZHEN LI*, YI LIU*, XIAOYAN LIU*, QINGXIAO SUN*, XIN YOU*, HAILONG YANG*†, ZHONGZHI LUAN*, LIN GAN§, GUANGWEN YANG§, and DEPEI QIAN*, Beihang University* and Tsinghua University§

上次内容已经介绍完**High-Level IR**的**Graph IR**（深度学习编译器中High-Level IR一般指的是Graph IR）

本文将继续进行深入IR——**Low-Level IR**

关于High-Level IR和Low-Level IR的区别，由两者的优化方向即能看出其体现的功能。



从上图可以看出

- **High-Level IR**:通常更接近于神经网络模型的原始描述。它负责高层次的优化和模型表示，允许进行更抽象的操作。在这个层次上，针对模型的整体结构进行优化、融合、替代等操作。
- **Low-Level IR**: 更接近底层硬件的特性和操作。它负责将高级表示翻译成更接近硬件的表达形式，以便进行低层次的优化和代码生成。

Low-level IR

Implementation of Low-Level IR

Low-Level IR 比 High-Level IR更细粒度的表示形式描述 DL 模型的计算，从而通过提供调整计算和内存访问的接口来实现目标相关的优化。

在本节中，我们将低级 IR 的常见实现分为三类：基于**Halide-based IR**、**polyhedral-based IR** 和其他的 IR。

Halide-based IR

Halide 最初被提出用于并行化图像处理，并且在 DL 编译器（例如 TVM）中被证明具有可扩展性和高效性。

Halide 的基本理念是计算和调度的分离。

> Halide 的基本理念是"计算"（算法）与"调度"

> 下面是一个简单的自定义图像处理函数 `gradient` ，该函数计算了一个图像的gradient，并在生成的图像上检查了计算结果的正确性。
>
> 其中计算的定义（像素计算）与调度的策略（reorder和parallel）是分开的

```
#include <iostream>
#include <Halide.h>

int main() {
 Halide::Func gradient;
 Halide::Var x, y;
 Halide::Expr e = x + y;
 gradient(x, y) = e;

 gradient.reorder(y,x);
 gradient.parallel(y);

 Halide::Buffer<int32_t> output = gradient.realize({800, 600});
 for (int j = 0; j < output.height(); j++) {
     for (int i = 0; i < output.width(); i++) {
         if (output(i, j) != i + j) {
             printf("Something went wrong!\n"
                    "Pixel %d, %d was supposed to be %d, but instead it's %d\n",
                    i, j, i + j, output(i, j));
             return -1;
         }
     }
 }
 printf("Success!\n");
 return 0;
}
```

采用Halide的深度学习编译器，这些编译器会尝试采用不同的调度策略来安排计算任务的执行，以找到性能最佳的方案。

> 换句话说，编译器会自动尝试多种不同的计算任务排序、并行化策略等，然后评估每个方案的性能
> 最终选择效果最好的一种来生成高效的底层代码，从而加速深度学习模型的执行。
> 这种自动化的过程有助于减轻开发者的工作负担，同时确保生成的代码尽可能地优化。

Halide 中内存引用和循环嵌套的边界仅限于与轴对齐的有界框。因此，Halide无法表达复杂图案（例如非矩形）的计算。

幸运的是，DL 中的计算非常规则，可以用 Halide 完美表达。

此外，Halide 可以轻松参数化这些边界并将其暴露给调整机制。

Halide的原始IR在应用于DL编译器的后端时需要修改。

例如，Halide 的输入形状是无限的，而深度学习编译器需要知道数据的确切形状，以便将运算符映射到硬件指令。

一些编译器（例如 TC）需要固定的数据大小，以确保张量数据具有更好的时间局部性。

TVM通过以下努力将Halide IR改进为独立的符号IR。

- 它消除了对 LLVM 的依赖，并重构了 Halide 的项目模块和 IR 设计的结构，追求更好的组织以及图 IR 和 Python 等前端语言的可访问性。
- 复用性也得到了提高，实现了运行时调度机制，可以方便地添加自定义算子

> TVM引入了一个运行时调度机制，使用户可以方便地添加自定义的运算符
> 举一个简单的例子：
>
> my_add_func(x) = x+ 100

```
import numpy as np
from tvm import te
import tvm
@tvm.register_func("my_add_func")
def my_add_func(x, y):
tvm.nd.array(x.numpy() + 100).copyto(y)
x = te.placeholder((100,), name="A")
y = te.extern(x.shape,[x],lambda i, j: tvm.tir.call_packed("my_add_func",i[0], j[0]),n
te_func = te.create_prim_func([x, y])
func = tvm.build(te_func, "llvm")
a = tvm.nd.array(np.ones(shape=(100,)).astype(x.dtype))
b = tvm.nd.array(np.ones(shape=(100,)).astype(y.dtype))
func(a, b)
np.testing.assert_allclose(b.numpy(), a.numpy() + 100, rtol=1e-5)
te_func.show()

'''output:'''
# from tvm.script import tir as T
@T.prim_func
def func(var_A: T.handle, var_C: T.handle):
# function attr dict
T.func_attr({"global_symbol": "main", "tir.noalias": True})
A = T.match_buffer(var_A, [100], dtype="float32", offset_factor=1)
C = T.match_buffer(var_C, [100], dtype="float32", offset_factor=1)
# body
with T.block("C"):
   T.reads(A[0 : 100])
   T.writes(C[0 : 100])
   T.tvm_call_packed("my_add_func", T.tvm_stack_make_array(A.data, T.tvm_stack_make_sh
```

- TVM 将变量定义从字符串匹配简化为指针匹配，保证每个变量都有一个定义位置（静态单赋值，SSA）。

Polyhedral-based IR

多面体模型是深度学习编译器采用的一项重要技术。它使用线性规划、仿射变换和其他数学方法来优化具有边界和分支的静态控制流的基于循环的代码。

与 Halide 不同的是，内存引用和循环嵌套的边界可以是多面体模型中任意形状的多面体。

> 多面体模型（polyhedral model）是在编译器优化中应用的一种高级技术，旨在优化循环结构的并行性和局部性。
> 它将程序的循环依赖关系转化为多维几何空间中的多面体，从而提供了一种抽象的方式来分析、优化和重组循环代码。
>
> 简单举一个多面体模型的应用例子。
> 对于以下循环计算操作

```
for (int i = 1; i < N; ++i)
for (int j = 1; j < N; ++j)
A[i][j] = A[i-1][j] + A[i][j-1];
```

其中数据的依赖关系图如下所示：

图源https://zhuanlan.zhihu.com/p/619003579

其中循环迭代顺序图如下所示：

图源https://zhuanlar

如果经过倾斜变换，可得到

```
for (int d = 2; d <= 2 * 5; ++d) {
for (int j = max(1, d - 5); j <= min(5, d - 1); ++j) {
   int i = d - j;
   A[i][j] = A[i-1][j] + A[i][j-1];
}
}
```

其中数据的依赖关系图如下所示：

图源https://zhuanlan.zhihu.com/p/619003579

其中循环迭代顺序图如下所示：

图源https://zhuanlan.zhihu.com/p/619003579

因为在j维度上，没有数据依赖，我们将可以在j维度上实现并行，优化循环结构的并行性。

这种灵活性使得多面体模型广泛应用于通用编译器中。然而，这种灵活性也阻碍了与调整机制的集成。

然而，由于能够处理深度嵌套循环，许多深度学习编译器，例如 TC 和 PlaidML（作为 nGraph 的后端），都采用多面体模型作为其low-level IR。

基于多面体的 IR 可以轻松应用各种多面体变换（例如fusion、tiling、sinking和mapping），包括设备相关和设备无关的优化。基于多面体的编译器借用了许多工具链，例如 isl 、Omega 、PIP 、Polylib 和 PPL 。

TC在低强度IR方面有其独特的设计，它结合了Halide和多面体模型。它使用基于Halide的IR来表示计算，并采用基于多面体的IR来表示循环结构。 TC通过抽象实例呈现详细的表达，并引入具体的节点类型。

简而言之，TC使用域节点来指定索引变量的范围，并使用上下文节点来描述与硬件相关的新迭代变量。它使用带节点来确定迭代的顺序。过滤器节点表示与语句实例组合的迭代器。 Set和sequence是关键字，用于指定过滤器的执行类型（并行和串行执行）。此外，TC使用扩展节点来描述代码生成的其他必要指令，例如内存移动。

PlaidML 使用基于多面体的 IR（称为 Stripe）来表示张量运算。它通过将并行多面体块的嵌套扩展到多个级别来创建可并行代码的层次结构。此外，它允许将嵌套多面体分配给嵌套内存单元，提供了一种将计算与内存层次结构相匹配的方法。在Stripe中，硬件配置独立于内核代码。 Stripe 中的标签（在其他编译器中称为 pass）不会更改内核结构，但提供有关优化遍的硬件目标的附加信息。 Stripe 将 DL 运算符拆分为适合本地硬件资源的图块。

Other unique IR

有些 DL 编译器可以在不使用 Halide 和多面体模型的情况下实现定制的low-level IR。

在定制的low-level IR 上，他们应用特定于硬件的优化并lowers to LLVM IR。

Glow 中的低级 IR 是基于指令的表达式，对地址引用的张量进行操作。

> Glow（Graph Lowering）是一个开源的深度学习编译器项目，由Facebook AI Research（FAIR）开发

Glow 低级 IR 中有两种基于指令的函数：声明和编程。

- 第一个声明了在程序的整个生命周期中存在的恒定内存区域的数量（例如输入、权重、偏差）。
- 第二个是本地分配区域的列表，包括函数（例如 conv 和 pool）和临时变量。

指令可以在全局内存区域或本地分配的区域上运行。

此外，每个操作数都用限定符之一进行注释：

- @in 表示操作数从缓冲区读取；
- @out表示操作数写入缓冲区；
- @inout 表示操作数读取和写入缓冲区。

这些指令和操作数限定符帮助 Glow 确定何时

MLIR 深受 LLVM 的影响，它是比 LLVM 更纯粹的编译器基础设施。 MLIR 重用了 LLVM 中的许多思想和接口，位于模型表示和代码生成之间。

MLIR 具有灵活的类型系统并允许多个抽象级别，并且它引入了dialect来表示这些多个抽象级别。每种dialect都包含一组定义的不可变操作。

MLIR 当前的dialect包括 TensorFlow IR、XLA HLO IR、实验性多面体 IR、LLVM IR 和 TensorFlow Lite。还支持dialect之间的灵活转换。

此外，MLIR 可以创建新的dialect来连接到新的低级编译器，这为硬件开发人员和编译器研究人员铺平了道路。

XLA 的 HLO IR 可以被视为高级 IR 和低级 IR，因为 HLO 的粒度足够细，可以表示特定于硬件的信息。此外，HLO 支持特定于硬件的优化，并可用于发出 LLVM IR。

Code Generation based on Low-Level IR

大多数DL编译器采用的低级IR最终可以lowers to LLVM IR，并受益于LLVM成熟的优化器和代码生成器。

此外，LLVM 可以从头开始显式地为专用加速器设计自定义指令集。

然而，传统编译器在直接传递给 LLVM IR 时可能会生成质量较差的代码。

为了避免这种情况，DL 编译器应用两种方法来实现硬件相关的优化：

1. 在 LLVM 的上部 IR 中执行特定于目标的循环转换（例如，基于 Halide 的 IR 和基于多面体的 IR），
2. 提供有关优化过程的硬件目标的附加信息。

大多数深度学习编译器都应用这两种方法，但侧重点不同。

一般来说，更喜欢前端用户的 DL 编译器（例如 TC、TVM、XLA 和 nGraph）可能会关注 1。

而更倾向于后端开发人员的 DL 编译器（例如 Glow、PlaidML 和 MLIR）可能会关注 1，重点关注2。

DL编译器中的编译方案主要分为两类：即时（JIT）和提前（AOT）。

- 对于 JIT 编译器来说，它可以即时生成可执行代码，并且可以通过更好的运行时知识来优化代码。
- AOT 编译器首先生成所有可执行二进制文件，然后执行它们。因此，它们在静态分析方面比 JIT 编译具有更大的范围。
- 此外，AOT 方法可以应用于嵌入式平台的交叉编译器（例如 C-GOOD ），并支持在远程计算机（TVM RPC）和定制加速器上执行。

Discussion

在深度学习编译器中，低级IR是深度学习模型的细粒度表示，它反映了深度学习模型在不同硬件上的详细植入。

低级 IR 包括基于Halide的 IR、基于多面体的 IR 和其他独特的 IR。

尽管它们在设计上有所不同，但它们利用成熟的编译器工具链和基础设施来提供特定于硬件的优化和代码生成的定制接口。

低级 IR 的设计还会影响新型深度学习加速器（例如 TVM HalideIR 和 Inferentia，以及 XLA HLO 和 TPU）的设计。