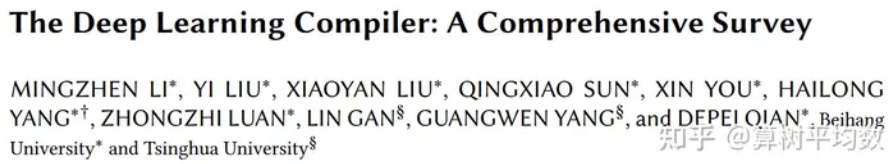
<七> 深度学习编译器综述：Backend Optimizations(2)

<七> 深度学习编译器综述：Backend Optimizations(2)

算树平均数
昏昏沉沉工程师 (寻职中)
关注他

这是这篇survey的最后一部分内容了，由于这是21年DL compiler的survey，comparison的内容可能没有太强的参考性，目前DL compiler主要还是MLIR和TVM两个部分。
笔者尚且为入门水平，有误之处请见谅，有其他问题和交流可联系我。

The Deep Learning Compiler: A Comprehensive Survey



1. Auto-tuning

auto-tuning在编译器后端至关重要，可以减轻手动获取最佳参数配置的工作量

> 深度学习编译器中的 自动调优（Auto-Tuning） 是一项重要的技术
> 它旨在自动优化深度学习模型的性能，以提高模型的训练和推理速度，同时减少资源消耗。
> Auto-Tuning 通过自动搜索和选择最佳的编译器优化参数、硬件配置以及算法变体等方式来实现这一目标

1. **背景**:

深度学习模型在不同硬件平台上运行时，性能表现会有所不同。
为了充分发挥硬件潜力，开发者需要手动调整模型的参数和编译器选项，这需要大量的时间和专业

1. **工作原理**： Auto-Tuning 的工作原理通常包括以下步骤：

a. **搜索空间定义**：首先，需要定义一个搜索空间，其中包含了各种编译器选项、硬件配置和算法变体的组合。
b. **性能评估**：针对搜索空间中的每个组合，使用一组性能指标来评估深度学习模型的性能。这些指标可以包括模型的训练速度、推理速度、内存消耗等。
c. **搜索算法**：选择合适的搜索算法，通常采用启发式搜索方法，例如遗传算法、粒子群优化或贝叶斯优化，来探索搜索空间。
d. **自动化决策**：根据性能评估的结果，自动选择最佳的编译器选项、硬件配置和算法变体组合。

由于硬件特定优化中参数调整的搜索空间巨大，因此有必要利用自动调整来确定最佳参数配置。

在本次Survey中的 DL 编译器中，TVM、TC 和 XLA 支持自动调整。

一般来说，自动调整的实现包括四个关键部分：

- **Parameterization 参数化**
- **Cost Model 损失模型**
- **Search Technology 搜索技术**
- **Acceleration 加速**

1.1 引例：autoTVM

TVM中用的auto Tuning算法有autoTVM和auto Scheduling (Ansor)，有关这两个auto tuning技术后续可能进行单独paper note。

下面简单以 `autoTVM` 作为引例，介绍涉及以上四个关键部分并了解 `autoTVM` 是如何自动找到算子最快的实现方式

现有以下矩阵相乘运算 e：

e compute expression

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024),
    lambda y, x:
    t.sum(A[k, y] * B[k, x], axis=k))
```

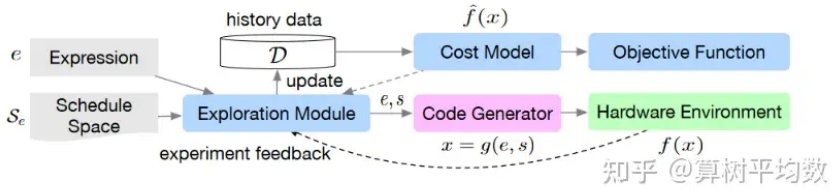
对于 S_e (优化搜索空间)，一般就是涉及到 `Loop oriented optimizations ， Hardware intrinsic mapping` 等，auto Tuning目的就是如何将算子 e 通过找到一个 $s(s \in S_e)$ 进行变换优化，得到一个在当前硬件下更高效的实现方式，例如进行变换优化后 s_2 得到如下实现：

```
yo,xo,ko,yi,xi,ki = s[C].title(y,x,k,8,8,8)
s[C].tensorize(yi, intrin.gemm8x8)
```

$$x_2 = g(e, s_2)$$

```
for yo in range(128):
  for xo in range(128):
    intrin.fill_zero(C[yo*8:yo*8+8][xo*8:xo*8+8])
    for ko in range(128):
      intrin.fused_gemm8x8_add(
        C[yo*8:yo*8+8][xo*8:xo*8+8],
        A[ko*8:ko*8+8][yo*8:yo*8+8],
        B[ko*8:ko*8+8][xo*8:xo*8+8])
```


1.1.1 autoTVM算法原理



- 通过搜索算法（模拟退火）以$\hat{f}(x)$作为energy function 并行地获取 $candidates(s_1, s_2, s_3, ...)$
- 通过一个考虑 `quality` 和 `diversity` 的评价函数从 `candidates` 中选取 `schedule` 集合 S
- `code Generator` $x = g(e, s)$: 表示算子 e 通过 s 变换后的生成代码 x

`Hardware Environment` $f(x)$: 表示代码 x 在 Hardware上的性能

```
# 计算集合S并收集运行性能
for s in S:
  c = f(g(e,s))
  D =+{(e,s,c)}
```

- 根据收集性能数据的 D 优化 `cost model` $\hat{f}(x)$
- 深度学习编译器迭代以上步骤直至到设置的迭代数，最后输出best schedule s^*

1.2 autoTVM 示例

我们利用一个矩阵乘法作为示例，并对比baseline 和 auto tuning的性能对比：

```
import logging
import sys
import numpy as np
import tvm
from tvm import te
import tvm.testing
from tvm import autotvm
```

```
    A = te.placeholder((N, L), name="A", dtype=dtype)
    B = te.placeholder((L, M), name="B", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)

    # 调度
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    yo, yi = s[C].split(y, 8)
    xo, xi = s[C].split(x, 8)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]
s, arg_bufs = matmul_basic(512,512,512,'float32')
func = tvm.build(s, arg_bufs)
ctx = tvm.cpu()
a = tvm.nd.array(np.ones((512,512)).astype("float32"))
b = tvm.nd.array(np.ones((512,512)).astype("float32"))
res = tvm.nd.array(np.zeros((512,512)).astype("float32"))
evaluator = func.time_evaluator(func.entry_name, ctx, number=10)
print("Baseline: %f" % evaluator(a, b, res).mean)
'''
Baseline: 0.013417
'''
@autotvm.template("tutorial/matmul")
def matmul(N, L, M, dtype):
    A = te.placeholder((N, L), name="A", dtype=dtype)
    B = te.placeholder((L, M), name="B", dtype=dtype)

    k = te.reduce_axis((0, L), name="k")
    C = te.compute((N, M), lambda i, j: te.sum(A[i, k] * B[k, j], axis=k), name="C")
    s = te.create_schedule(C.op)

    # 调度
    y, x = s[C].op.axis
    k = s[C].op.reduce_axis[0]

    ##### 开始定义空间 #####
    cfg = autotvm.get_config()
    cfg.define_split("tile_y", y, num_outputs=2)
    cfg.define_split("tile_x", x, num_outputs=2)
    ##### 结束定义空间 #####

    # 根据 config 进行调度
    yo, yi = cfg["tile_y"].apply(s, C, y)
    xo, xi = cfg["tile_x"].apply(s, C, x)

    s[C].reorder(yo, xo, k, yi, xi)

    return s, [A, B, C]

N, L, M = 512, 512, 512
task = autotvm.task.create("tutorial/matmul", args=(N, L, M, "float32"), target="llvm"
logging.getLogger("autotvm").setLevel(logging.DEBUG)
logging.getLogger("autotvm").addHandler(logging.StreamHandler(sys.stdout))
measure_option =autotvm.measure_option(builder='local',runner=autotvm.LocalRunner(numb
# 用 RandomTuner 开始调优, 日志记录到 `matmul512.log` 文件中
```

```
tuner.tune(
    n_trial=50,
    measure_option=measure_option,
    callbacks=[autotvm.callback.log_to_file("matmul512.log")],
)
# 从日志文件中应用历史最佳
with autotvm.apply_history_best("matmul512.log"):
    with tvm.target.Target("llvm"):
        s_autotvm, arg_bufs_autotvm = matmul(512, 512, 512, "float32")
        func_autotvm = tvm.build(s_autotvm, arg_bufs_autotvm)
ctx = tvm.cpu()
a = tvm.nd.array(np.ones((512,512)).astype("float32"))
b = tvm.nd.array(np.ones((512,512)).astype("float32"))
res = tvm.nd.array(np.zeros((512,512)).astype("float32"))
evaluator = func_autotvm.time_evaluator(func_autotvm.entry_name, ctx, number=10)
print("autoTVM : %f" % evaluator(a, b, res).mean)

'''
autoTVM : 0.007993
'''
```

相比于baseline，通过autoTVM进行auto tuning后，可以达到1.5x的加速

1.2 Parameterization

1.2.1 Data and Target:

- `data parameter` 描述了数据的规格，例如输入形状。
- `target parameter` 描述了优化调度和代码生成期间要考虑的硬件特定特征和约束。

例如，对于GPU目标，需要指定共享内存和寄存器大小等硬件参数。

1.2.2 Optimization options:

优化选项包括优化调度和相应的参数，例如 `loop oriented optimization` 和 `tile size` 。下·

在TVM中， `pre-defined` (预定义)和 `user-defined` (用户定义)的调度和参数都被考虑在内。

而TC和XLA更喜欢与性能有很强的相关性的参数化优化，并且可以在以后以较低的成本进行更改。

例如， `minibatch dimension` 是 `CUDA` 中通常映射到 `grid dimensions` 的参数之一，可以在 `auto tuning` 过程中进行优化。

1.3 Cost model

1.3.1 Black-box model:

该模型只考虑最终的执行时间而不考虑编译task的特性。

构建黑盒模型很容易，但如果没有task特性的指导，很容易导致更高的开销并且最终得到非最优的解决方案。

1.3.2 ML-based cost model:

基于机器学习的cost model是一种使用机器学习方法预测性能的统计方法。

TVM和XLA分别采用这种模型，例如梯度树提升模型（GBDT）和前馈神经网络（FNN）。

1.3.3 Pre-defined cost model:

基于预定义成本模型的方法期望建立一个基于编译任务特征的完美模型，并且能够评估任务的整体性能。

与基于 ML 的模型相比，预定义模型在应用时产生的计算开销较少，但需要大量的工程工作来在每个新的 DL 模型和硬件上重新构建模型。

1.4 Searching technique

1.4.1 Initialization and searching space determination:

初始选项可以随机设置，也可以基于已知配置，例如用户给出的配置或历史最优配置。

在搜索空间方面，应在自整定前指定。

TVM 允许开发人员使用其特定领域的知识指定搜索空间，并根据计算描述为每个硬件目标提供自动搜索空间提取。

相比之下，TC 依赖于编译缓存和预定义的规则。

1.4.2 Genetic algorithm (GA):

遗传算法将每个调整参数视为基因，将每个配置视为候选。

根据适应度值，通过交叉、变异和选择迭代生成新的候选者，这是受自然选择过程启发的元启发式算法。

最后，得出最佳候选者。交叉、变异和选择的速率用于控制探索和利用之间的权衡。 TC在其自整定技术中采用了GA

1.4.3 Simulated annealing algorithm (SA):

模拟退火算法也是一种受退火启发的元启发式算法。

它允许我们以递减的概率接受更差的解，从而可以在固定的迭代次数中找到近似的全局最优，并避免精确的局部最优。

TVM在其自动调整技术中采用了SA。

1.4.4 Reinforcement learning (RL):

强化学习通过学习在探索和利用之间进行权衡来最大化给定环境的奖励。 Chameleon（基于 TVM 构建）在其自动调整技术中采用了 RLRL。

1.5 Acceleration

1.5.1 Parallelization:

加速自动调优的方向之一是并行化。考虑到遗传算法需要评估每一代中的所有候选者，TC 提出了多线程、多 GPU 策略。

生成的代码在 GPU 上并行评估，每个候选者都拥有其父选择步骤所使用的适应度。

整个评估完成后，生成新的候选，并将新的编译task入队，等待CPU上的编译。

同样，TVM支持交叉编译和RPC，允许用户在本地机器上编译并在多个目标上运行具有不同自动调优配置的程序。

1.5.2 Configuration reuse

加速自动调整的另一个方向是重复使用以前的自动调整配置。

TC 通过编译缓存存储与给定配置相对应的已知最快生成代码版本。

在编译过程中，每次内核优化前都会对缓存进行查询，如果缓存缺失，就会触发自动调整。

同样，TVM 会生成一个日志文件，存储所有调度算子的最优配置，并在编译过程中查询日志文件以获得最佳配置。

值得一提的是，TVM 会对 Halide IR（如 conv2d）中的每个运算符执行自动调整，因此每个运算符的最佳配置都是单独确定的。

2. Discussion

后端负责**bare-metal**优化和基于底层IR的代码生成。

虽然后端设计可能因底层IR的不同而不同，但它们的优化可归类为以下：

- **Hareware-specific optimizations**
- **Auto-tuning techniques**
- **Optimizated kernel libraries**

这些优化可以单独执行，也可以组合执行，通过利用软硬件的特性来实现更好的数据局部性和并行化。

最终将DL模型的高层IR转化为不同硬件上的高效代码实现。

编辑于 2024-01-04 17:44 · IP 属地中国香港

编译器　深度学习（Deep Learning）