

赞同 10

分享

<六> 深度学习编译器综述: Backend Optimizations(1)

算树平均数 昏昏沉沉工程师 (寻职中)

关注他

★ 你收藏过 编程 相关内容

<一> 深度学习编译器综述: Abstract & Introduction

<二> 深度学习编译器综述: High-Level IR (1)

<三> 深度学习编译器综述: High-Level IR (2)

<四> 深度学习编译器综述: Low-Level IR

<五> 深度学习编译器综述: Frontend Optimizations

<六> 深度学习编译器综述: Backend Optimizations(1)

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

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[§]

知乎 @算树平均数

1. Backend Optimizations

深度学习编译器的后端通常包括

- **hardware-specific optimizations**, 特定于硬件的优化

hardware-specific optimizations, 特定于硬件的优化可以为不同的硬件目标高效地生成代码

- **auto-tuning techniques**

赞同 10

添加评论

分享

喜欢

收藏

申请转载

- **optimized kernel libraries**

optimized kernel libraries也广泛应用于通用处理器和其他定制的深度学习加速器上, 例如 cuDNN, DNNL, MIOpen



2. Hardware-specific Optimizations

hardware-specific optimizations(特定于硬件的优化), 也称为target-dependent optimizations, 用于针对特定硬件的高性能代码生成。

后端优化的方法有两种方式

- 是将低级 IR 转换为 LLVM IR, 以利用 LLVM 基础设施生成优化的 CPU/GPU 代码。
- 利用深度学习领域知识设计定制优化, 更有效地利用目标硬件。

深度学习编译器后端优化时, 为什么不直接使用LLVM呢?

深度学习编译器的后端优化时不直接使用LLVM的一个原因是, 深度学习计算图 (DNN模型) 的特殊性和对性能的要求可能与通用编程语言的编译不完全匹配, 需要定制化的优化策略和技术。

以下是一些原因:

特定硬件的优化: 深度学习编译器通常需要为特定类型的硬件进行优化, 如图形处理单元 (GPU)、张量处理单元 (TPU) 或定制化的深度学习加速器。

这些硬件对于深度学习任务具有特定的计算需求, 因此需要定制化的优化策略, 而不仅仅是通用的LLVM优化。

高级抽象: 深度学习模型的计算图通常包含高级抽象, 例如卷积、池化、批归一化等操作, 这些操作在底层需要进行高效的优化, 但这种优化可能在通用编程语言中不太容易实现。

自动微调: 深度学习编译器通常需要进行自动微调 (Auto-Tuning) 来找到最佳的参数配置, 以满足模型和硬件的性能需求。

这种自动微调的过程通常需要与深度学习框架和硬件特性紧密结合, 而不仅仅是LLVM的标准优化。

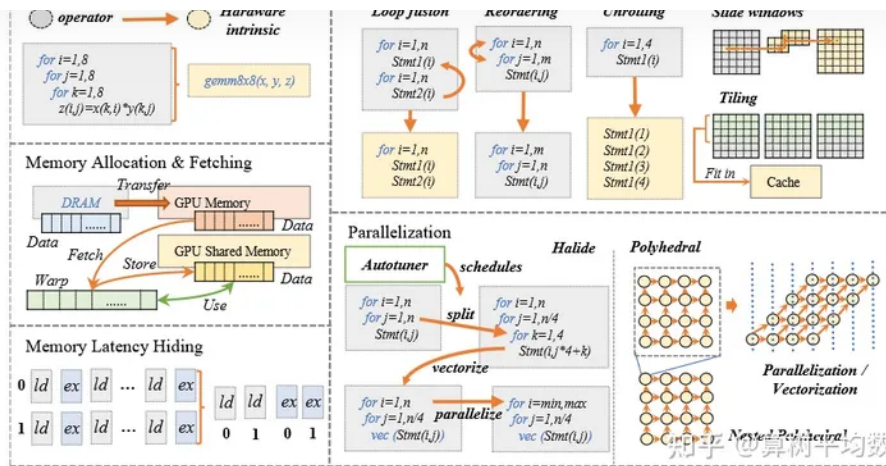
低级细节: 深度学习编译器需要处理低级细节, 如内存管理、数据布局和量化等, 以最大程度地减少计算和内存开销。这些细节需要特定于深度学习的优化技巧。

尽管深度学习编译器的后端可以受益于LLVM的一些通用优化技术, 但深度学习编译器通常需要更多的领域特定优化, 以满足深度学习模型的性能需求。

因此, 通常会在LLVM之上构建定制的深度学习编译器后端, 以更好地适应深度学习任务的特殊性。

这些深度学习编译器后端会结合LLVM的一些优化技术, 但也包括许多领域特定的优化策略, 以提高深度学习模型的性能、减少计算资源的占用, 并支持不同类型的硬件加速器

由于特定于硬件的优化是针对特定硬件量身定制的, 因此无法在本文中详尽介绍, 因此我们提出了现有深度学习中广泛采用的五种方法。



2.1 Hardware intrinsic mapping

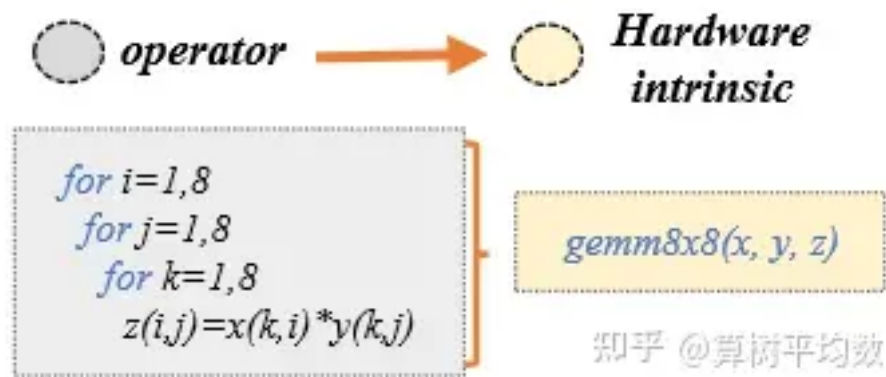
Hardware intrinsic mapping, 硬件指令映射可以将一组特定的低级IR指令转换为已经在硬件上高度优化的kernel。

在TVM中，硬件指令映射是通过 *extensible tensorization* 实现的，它可以声明硬件指令的行为以及指令映射的lower。

这种方法使编译器后端能够将硬件实现以及高度优化的手工micro-kernels应用于特定的patterns，从而显著提高性能。

此外，Halide/TVM 将特定的 IR 模式映射到每个架构上的 SIMD opcodes，以避免 LLVM IR 映射在遇到vector patterns时效率低下。

Hardware Intrinsic Mapping



以下例子采用 TVM 利用 x86 中的 AVX512_DOT_16x4_INTRIN 指令实现矩阵加速，并且对比 baseline 的执行效率。

关于TVM详细用法，在此处不作详细展开和解释。

```
import tvm
from tvm import te
import numpy as np
from tvm.tir.tensor_intrin.x86 import VN
```

```
# baseline
m, n, k = 128, 128, 128
lhs_dtype = 'uint8'
rhs_dtype = 'int8'
X = te.placeholder((m, k), name="X", dtype=lhs_dtype)
W = te.placeholder((n, k), name="W", dtype=rhs_dtype)
ak = te.reduce_axis((0, k), name="k")

matmul = te.compute(
    (m, n),
    lambda i, j: te.sum(
        X[i, ak].astype("int32") * W[j, ak].astype("int32"),
        axis=ak,
    ),
    name="compute",
)

func = te.create_prim_func([X, W, matmul])
sch_baseline = tir.Schedule(func, debug_mask="all")
print(sch_baseline.mod.script())

'''output:
# from tvm.script import ir as I
# from tvm.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(X: T.Buffer((128, 128), "uint8"), W: T.Buffer((128, 128), "int8"), compute: T
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i, j, k in T.grid(128, 128, 128):
            with T.block("compute"):
                v_i, v_j, v_k = T.axis.remap("SSR", [i, j, k])
                T.reads(X[v_i, v_k], W[v_j, v_k])
                T.writes(compute[v_i, v_j])
            with T.init():
                compute[v_i, v_j] = 0
            compute[v_i, v_j] = compute[v_i, v_j] + T.Cast("int32", X[v_i, v_k]) * T.Ca
        ...

ctx = tvm.cpu()
mod = tvm.build(sch_baseline.mod, target="llvm -mcpu=skylake-avx512")
a = tvm.nd.array(np.ones((128, 128)).astype("uint8"))
b = tvm.nd.array(np.ones((128, 128)).astype("int8"))
res = tvm.nd.array(np.zeros((128, 128)).astype("int32"))
mod(a, b, res)
evaluator = mod.time_evaluator(mod.entry_name, ctx, number=50)
print("Baseline: %f" % evaluator(a, b, res).mean)

...
Baseline: 0.000937
...'''
```

可以看出，正常的baseline需要**0.000937**

```
m, n, k = 128, 128, 128
lhs_dtype = 'uint8'
```

```

W = te.placeholder((n, k), name="W", dtype=rhs_dtype)
ak = te.reduce_axis((0, k), name="k")

matmul = te.compute(
    (m, n),
    lambda i, j: te.sum(
        X[i, ak].astype("int32") * W[j, ak].astype("int32"),
        axis=ak,
    ),
    name="compute",
)

func = te.create_prim_func([X,W,matmul])
sch = tir.Schedule(func, debug_mask="all")
block = sch.get_block("compute")
sch.transform_layout(block, "W", lambda i, j: [i//16, j//4, i%16, j%4])
_, j, k = sch.get_loops(block)

_, ji = sch.split(j, factors=[None, 16])
ko, ki = sch.split(k, factors=[None, 4])
sch.reorder(ko, ji, ki)

sch.decompose_reduction(block, ko)

sch.tensorize(ji, AVX512_DOT_16x4_INTRIN)

print(sch.mod.script())

'''output:
# from tvn.script import ir as I
# from tvn.script import tir as T

@I.ir_module
class Module:
    @T.prim_func
    def main(X: T.Buffer((128, 128), "uint8"), W: T.Buffer((8, 32, 16, 4), "int8"), compute
        T.func_attr({"tir.noalias": T.bool(True)})
        # with T.block("root"):
        for i, j_0 in T.grid(128, 8):
            for j_1_init in range(16):
                with T.block("compute_init"):
                    v_i = T.axis.spatial(128, i)
                    v_j = T.axis.spatial(128, j_0 * 16 + j_1_init)
                    T.reads()
                    T.writes(compute[v_i, v_j])
                    compute[v_i, v_j] = 0
            for k_0 in range(32):
                with T.block("compute_update_o"):
                    v_i_o, v_j_o, v_k_o = T.axis.remap("SSR", [i, j_0, k_0])
                    T.reads(compute[v_i_o, v_j_o * 16:v_j_o * 16 + 16], X[v_i_o, v_k_o * 4:
                    T.writes(compute[v_i_o, v_j_o * 16:v_j_o * 16 + 16])
                    A = T.match_buffer(X[v_i_o, v_k_o * 4:v_k_o * 4 + 4], (4,), "uint8", of
                    B = T.match_buffer(W[v_j_o, v_k_o, 0:16, 0:4], (16, 4), "int8", offset_
                    C = T.match_buffer(compute[v_i_o, v_j_o * 16:v_j_o * 16 + 16], (16,), "
                    A_u8x4: T.uint8x4 = A[0:4]
                    A_i32: T.int32 = T.reinterpret("int32", A_u8x4)
                    A_brdcst: T.int32x16 = T.Broadcast(A_i32, 16)
                    A_u8x64: T.uint8x64 = T.reinterpret("uint8x64", A_brdcst)
                    B_i8x64: T.int8x64 = B[0,

```

```

...

ctx = tvm.cpu()
AVX512_DOT_16x4_INTRIN_mod = tvm.build(sch.mod, target="llvm -mcpu=skylake-avx512")
a = tvm.nd.array(np.ones((128,128)).astype("uint8"))
b = tvm.nd.array(np.ones((8, 32, 16, 4)).astype("int8"))
res = tvm.nd.array(np.zeros((128,128)).astype("int32"))
AVX512_DOT_16x4_INTRIN_mod(a,b,res)
evaluator = AVX512_DOT_16x4_INTRIN_mod.time_evaluator(AVX512_DOT_16x4_INTRIN_mod.entry
print("AVX512_DOT_16x4_INTRIN: %f" % evaluator(a, b, res).mean)

...

AVX512_DOT_16x4_INTRIN: 0.000026
...

```

经过AVX512_DOT_16x4_INTRIN的硬件指令映射，使得计算加速36x+

2.2 Memory allocation and fetching

内存分配是代码生成中的另一个挑战，特别是对于 GPU 和定制加速器而言。

例如，GPU主要包含 shared memory （内存大小有限，latency较低）和 local memory （容量大，latency较高）。

这种内存层次结构需要有效的 Memory allocation 和 fetching (获取)技术来提高数据局部性。

为了实现这种优化，TVM引入了内存范围的调度概念。

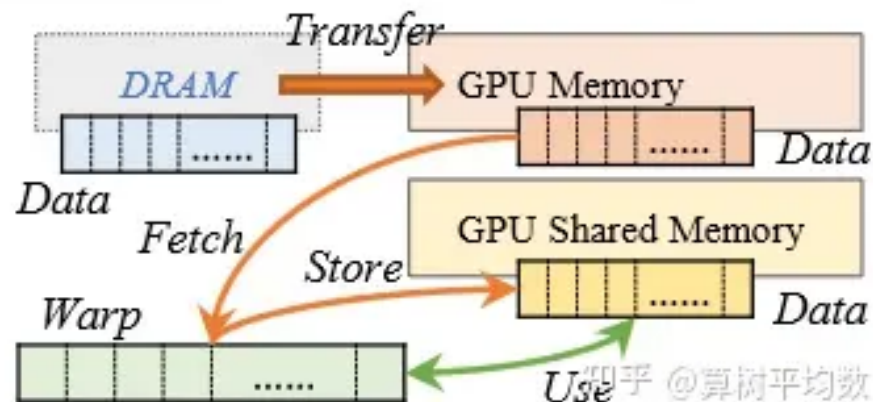
内存范围调度原语可以将计算 stage 记为 shared 或者 thread-local 。

对于标记为 shared 的计算 stage，TVM 生成具有shared memory allocation和协作数据 fetching的代码，这会在正确的代码位置插入 memory barrier 以保证正确性。

此外，TC还通过扩展PPCG编译器提供类似的功能（称为内存提升）。

然而，TC仅支持有限的预定义规则。特别的，TVM 通过内存范围调度原语在加速器中启用特殊缓冲。

Memory Allocation & Fetching



本节Memory allocation and fetching 就是如何提高数据的读取速度，在cuda中，采用合适的编程模型提高数据和时间局部性。主要涉及Memory Coalescing 和 Data Reuse。其中share memory的使用又会涉及到bank conflict问题，此处涉及的内容篇幅太多不做展开。



2.3 Memory latency hiding

通过对执行流水线重新排序，Memory latency hiding (内存延迟隐藏)也是后端使用的一项重要技术。

Memory Latency Hiding



由于大多数 DL 编译器都支持 CPU 和 GPU 的并行化，内存延迟隐藏自然可以通过硬件实现（例如 GPU 上的 warp 上下文切换）。

但对于采用解耦访问-执行（DAE）架构的 TPU 类加速器，后端需要执行调度和细粒度同步，以获得正确、高效的代码。

为了获得更好的性能并减轻编程负担，TVM 引入了 virtual threading 调度原语，使用户能够在虚拟多线程架构上指定数据并行性。

然后，TVM 通过插入必要的 memory barrier 来降低这些虚拟并行线程的并行性，并将这些线程的操作交错到一个指令流中，从而形成一个更好的各线程执行流水线，以隐藏内存访问延迟。

Virtual threading

虚拟线程是一种在 VTA 硬件设计中提高任务级流水线并行性的机制。

换句话说，它通过隐藏内存访问延迟来提高计算资源利用率。

在下面的实现过程中，虚拟线程将工作分配给两个线程，并沿输出通道轴分割。

下图中展示计算二维卷积时如何分配工作。



参考: zhuanlan.zhihu.com/p/46...



左侧有4个处于可调度状态的warp，当warp0执行一条高延迟指令（如内存读取指令）时。可以先让出计算单元并调度warp1、warp2或warp3的指令执行直到warp0读取状态就绪后再继续执行它。

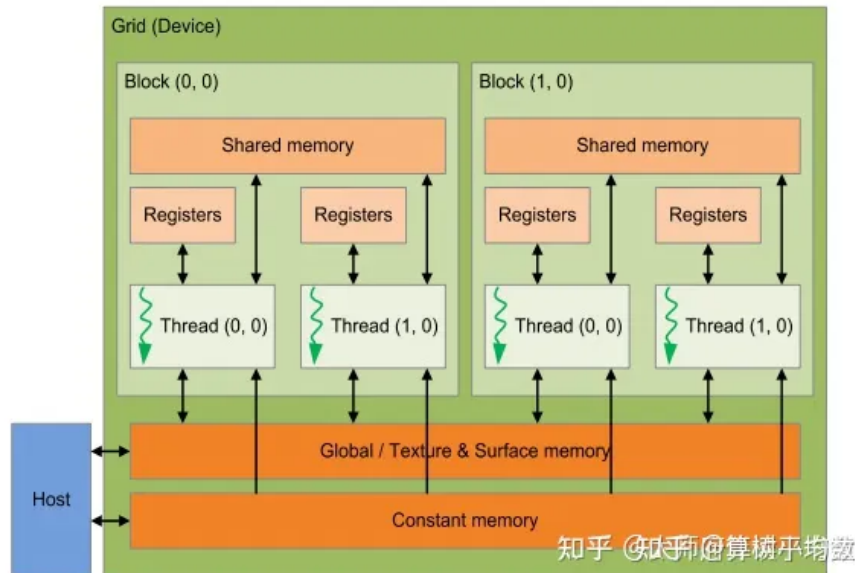
右侧只有2个处于可调度状态的warp，那么当warp0执行到高延迟指令（如内存读取）时，调度器会继续执行warp1，但如果warp1恰巧在warp0未就绪之前也遇到了一条高延迟指令，那么整个运算单元就只能等二者其中一个就绪后再继续执行了。

最终的时序上看，计算单元就出现了一段时间的空等待。然而，可调度warp的数目怎么来的呢？是程序员可控的吗？

首先，得看看硬件条件限制。

一个warp中的各个线程在同一时刻执行的是一条指令，所以一个warp至少需要一个程序计数器，记录当前运行的指令位置。这个值要存在寄存器里，而寄存器的数量显然是受GPU硬件限制的。

此外，每个线程计算指令过程中的中间结果也要用到寄存器，根据程序需要不同，也和编译器的实现好坏有关。同时在一个block的线程间共享内存的大小也是受硬件限制的。



img

GPU单核(SM)典型的限制如下：

1. 最多支持64个warp
2. 最多支持32个block
3. 寄存器总大小：256KB，需要支持SM上运行的所有线程。
4. 共享内存总大小：64KB，需要分配给SM上所有运行的各个block，因为内存是以block为单位进行共享的。

所以实际上，根据每个block对于共享内存大小的需求和单个warp对于寄存器的数量需求，同一时刻单个gpu核上能够并发调度执行的warp数量是不一样的，而前两者的大小和具体程序任务的需要与实现细节有关。

反过来，由于不同的硬件平台核心内部的实现也不一样，所以同样的一个程序，能够达到Occupancy的程度也不一样。

不过作为程序员，我们还是能够基本估算出，对应指标的，根据下面几个参数，一个block中有多少个线程（程序员决定）

上面三个数定了之后，根据对应硬件平台的特性，基本上能够推断出单核可调度状态warp的数量。

当然了，可能还需要辅助以一些基准测试，包括一些经验规则，让我们决定前两者的时候更加的自动和高效。

最终达到一个较高的Occupancy程度，从而达成Latency hiding的目的，提高gpu运算单元使用率。

然而，warp是越多越好吗？其实也不一定。

如果是存储访问密集型的任务，太多的warp反而造成了存储读取子系统的压力，让后者成为整体的性能瓶颈。

不过通常而言，提高Occupancy是一个初始优化的方向。

最终可能还是要根据对应硬件平台进行基准性能测试，才能决定如何组织我们的block任务。

2.4 Loop oriented optimizations

面向循环的优化也应用于后端，以便为目标硬件生成高效代码。

由于 Halide 和 LLVM（集成了多面体方法）已经集成了此类优化技术，一些 DL 编译器在其后端利用了 Halide 和 LLVM。

应用于面向循环优化的关键技术包括：

- **loop fusion**, 循环融合
- **sliding windows**, 滑动窗口
- **tiling**, 分块
- **loop reorder**, 循环重排
- **loop unrolling**, 循环展开

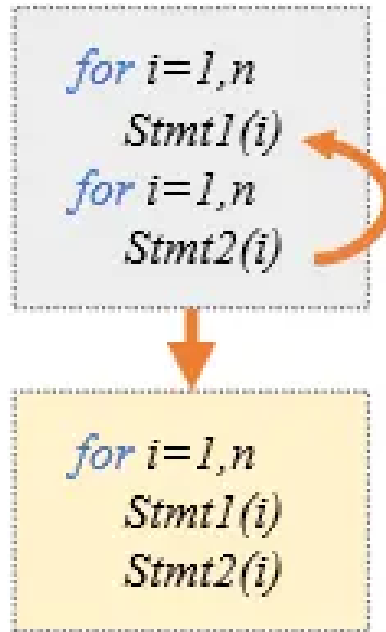
2.4.1 Loop fusion:

循环融合是一种循环优化技术，可以将具有相同边界的循环融合在一起，以实现更好的数据重用。

对于 PlaidML、TVM、TC 和 XLA 等编译器来说，这种优化是通过 Halide 计划或多面体方法来实现的

而 Glow 则通过算子堆叠来实现循环融合。

Loop fusion



2.4.2 Sliding windows

滑动窗口是 Halide 采用的一种循环优化技术。

它的核心理念是在需要时计算数值，并在不再需要时将其存储起来以重复使用数据。

由于滑动窗口将两个循环的计算交错进行，并使其串行化，因此需要在并行性和数据重用之间做出权衡。

Slide windows

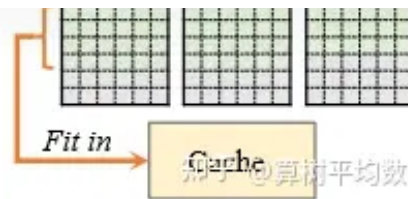


2.4.3 Tiling

分块法将循环拆分为多个块，从而将循环分为通过分块迭代的外循环和在分块内迭代的内循环。

这种转换通过将一块放入硬件缓存，使分块内部的数据具有更好的定位性。

由于分块的大小取决于硬件，因此许多 DL 编译器通过自动调整来确定分块模式和大小。

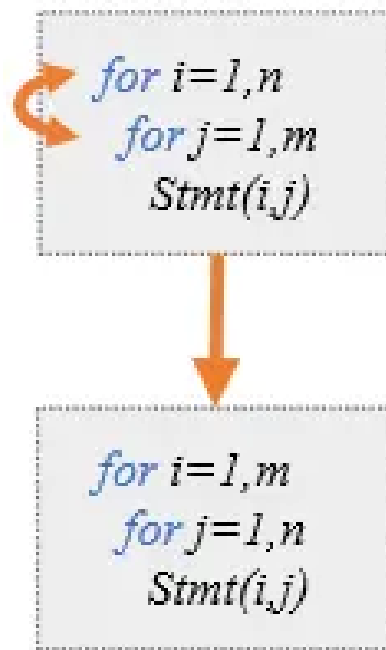


2.4.4 Loop reordering

循环重新排序（又称循环置换）改变嵌套循环中的迭代顺序，可以优化内存访问，从而提高空间局部性。

它与数据布局和硬件特性有关。不过，当迭代顺序存在依赖关系时，执行循环重排序并不安全。

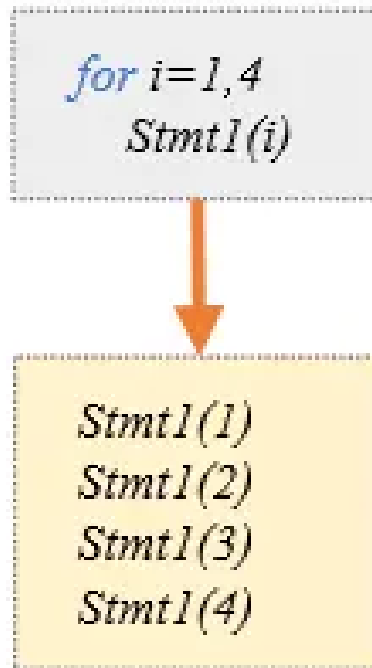
Reordering



2.4.5 Loop unrolling

循环展开可以将特定循环解卷为固定数量的循环体副本，从而允许编译器应用积极的指令级并行性。

通常，loop split 与 loop unroll 结合使用，首先将循环拆分为两个嵌套循环，然后完全解卷内部循环。



2.5 Parallelization

由于现代处理器普遍支持多线程和SIMD并行，编译器后端需要利用并行性来最大限度地提高硬件利用率，从而实现高性能。

Halide使用名为parallel的调度原语来指定线程级并行化的循环并行化维度，并通过映射为并行的循环维度与块和线程注释来支持GPU并行化。

此外，它还用一个 n-width vector语句取代了大小为 n 的循环，并可通过硬件内在映射将其映射到特定于硬件的 SIMD 操作码。

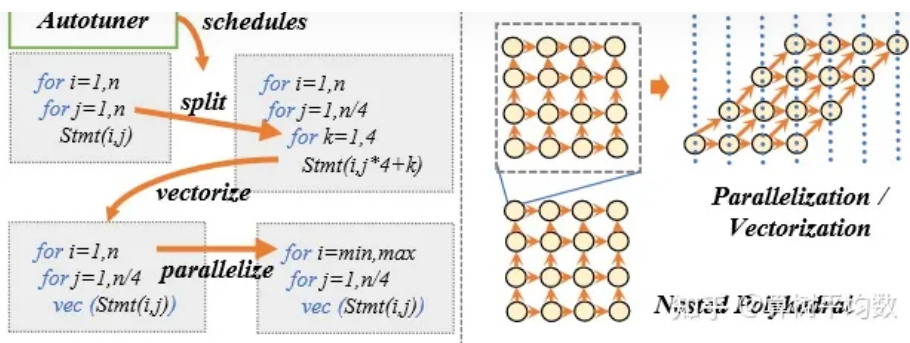
Stripe 开发了多面体模型的一种变体，称为嵌套多面体模型，它引入了并行多面体块作为迭代的基本执行元素。

经过这种扩展，嵌套多面体模型可以检测平铺和分层之间的层次并行化。

此外，一些 DL 编译器还依赖于 Glow 等手工库或硬件供应商提供的优化数学库。

同时，Glow 会将矢量化工作lower to LLVM，因为当提供张量维度和循环次数信息时，LLVM 自动矢量化器会工作得很好。

然而，完全由编译器后端利用并行性可以应用更多 DL 模型的特定领域知识，从而在牺牲更多工程努力的情况下获得更高的性能。



编辑于 2024-01-04 17:44 · IP 属地中国香港

深度学习（Deep Learning） 编译器 深度学习编译器

发布一条带图评论吧

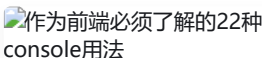


还没有评论，发表第一个评论吧

文章被以下专栏收录

深度学习编译器 深度学习编译器


推荐阅读



作为前端必须了解的22种console用法

作为前端必须了解的22种console用法

素颜



第二届TVM与深度学习编译器会议总结

第二届TVM与深度学习编译器会议总结

陈天奇 发表于tvm



深度学习编译器之TVM

【从零开始学深度学习编译器二，TVM中的scheduler

BBuf 发表于Gia