The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （四）

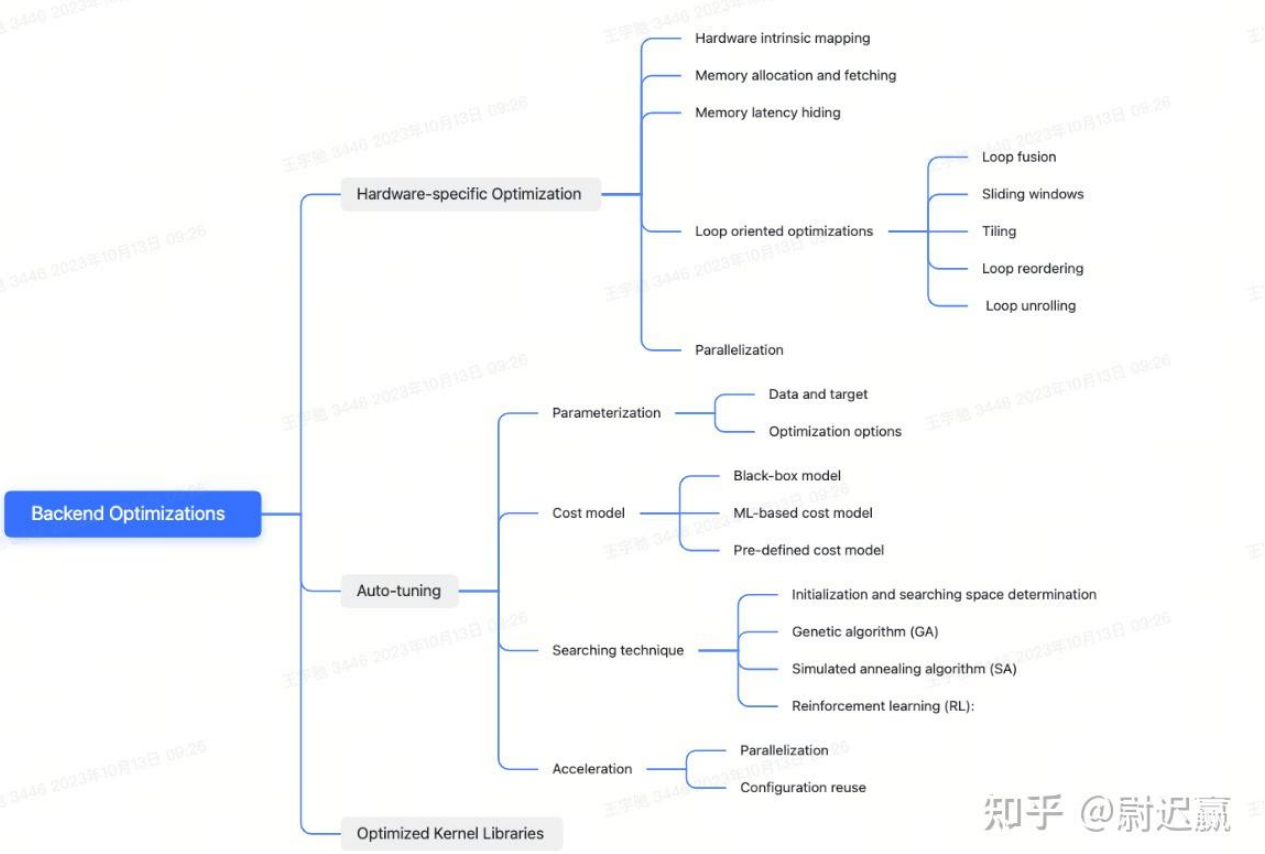
The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （一）

The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （二）

The Deep Learning Compiler- A Comprehensive Survey 深度学习编译器综述 （三）

4.4 Backend Optimizations

DL编译器的**后端**通常包括各种**硬件特定的优化、自动调优技术和优化内核库**。硬件特定的优化能够为不同的硬件目标生成高效的代码。而自动调优在编译器后端中非常重要，可以减轻手动确定最佳参数配置的工作量。此外，高度优化的内核库也广泛应用于通用处理器和其他定制化的DL加速器上。



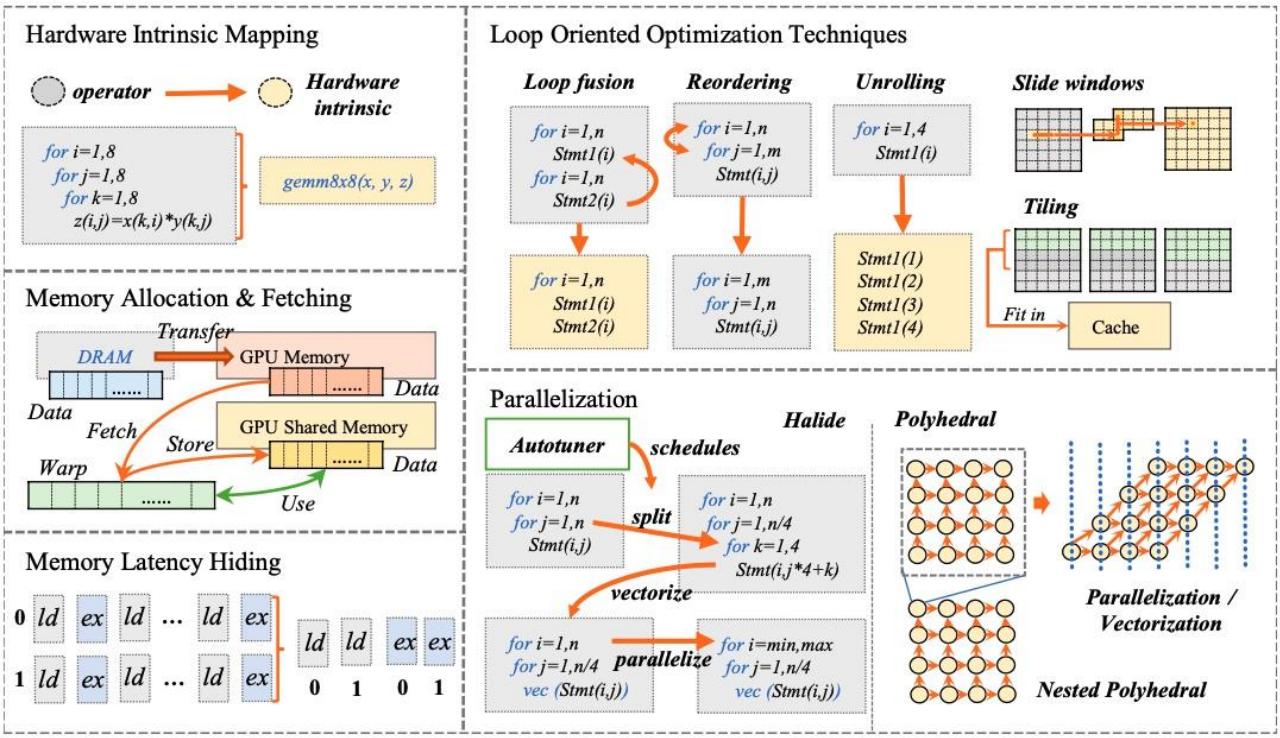


Fig. 4. Overview of hardware-specific optimizations applied in DL compilers.

4.4.1 Hardware-specific Optimization

硬件特定优化，也称为针对特定目标的优化，旨在获得针对特定硬件的高性能代码。将后端优化应用到生成优化的CPU/GPU代码的方法之一是将低级别IR转换为LLVM IR，利用LLVM基础设施来生成优化的代码。另一种方法是利用DL领域知识设计定制化的优化，更高效地利用目标硬件。由于硬件特定优化是针对特定硬件进行定制的，无法在本文中详尽列举，我们提供了现有DL编译器中采用的五种广泛使用的方法。这些硬件特定优化的概述如图4所示，详细描述如下所列。

Hardware intrinsic mapping

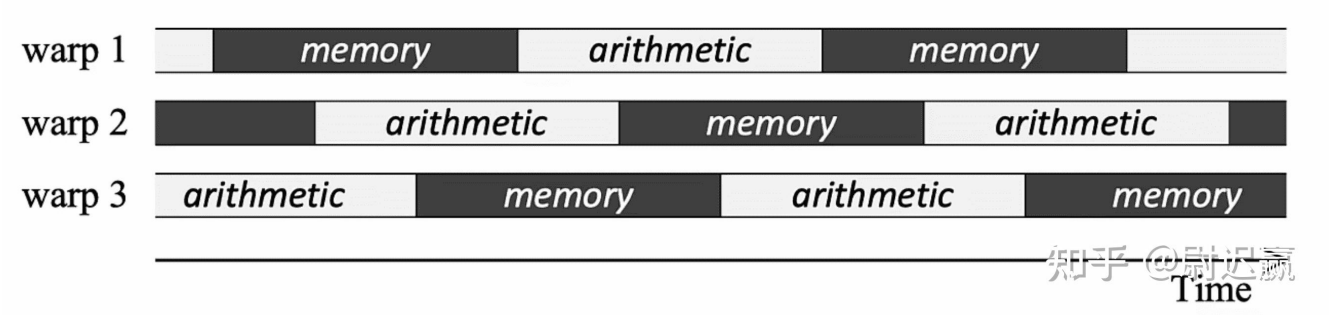
硬件内在映射可以将一定集合的低级别IR指令转换为已经在硬件上高度优化的内核。在TVM中，硬件内在映射是通过可扩展张量化（extensible tensorization）方法实现的，它可以声明硬件内在的行为以及内在映射的降低规则。这种方法使得编译器后端能够将硬件实现和高度优化的手工微内核应用于特定的操作模式（pattern of operations），从而获得显著的性能提升。另外，Glow支持硬件内在映射，如量化（quantization）。它可以估计神经网络每个阶段的可能数值范围，并支持基于剖析（profile- guided）的优化来自动进行量化。此外，Halide/TVM将特定的IR模式映射到每个架构上的SIMD指令，以避免在遇到向量模式时LLVM IR映射的低效问题。

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主要包含共享内存空间（具有有限的内存大小，但访问延迟较低）和本地内存空间（具有较大容量但访问延迟较高）。这种内存层次结构需要高效的内存分配和提取（allocation and fetching）技术以改善数据局部性。为了实现这种优化，TVM引入了内存作用域（memory scope）调度的概念。内存作用域调度原语可以将计算阶段标记为共享（shared）或线程本地（thread-local）。对于标记为共享的计算阶段，TVM会生成具有共享内存分配和协同数据提取的代码，它会在适当的代码位置插入内存屏障以确保正确性。此外，TC通过扩展PPCG [97]编译器也提供了类似的功能（称为内存提升(memory promotion)）。然而，TC仅支持有限的预定义规则。特别地，TVM通过内存作用域调度原语在加速器中实现了特殊的缓冲区功能。

Memory latency hiding

内存延迟隐藏也是后端中使用的一项重要技术，通过重新排序执行流水线来实现。由于大多数DL编译器支持在CPU和GPU上进行并行化，内存延迟隐藏可以通过硬件自然实现（例如GPU上的warp上下文切换）。但对于类似TPU的加速器，具有解耦的访问-执行（decoupled access-execute：DAE）架构，后端需要执行调度和细粒度同步以获得正确和高效的代码。为了实现更好的性能和减少编程负担，TVM引入了虚拟线程（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通过插入必要的内存屏障将这些虚拟并行化线程降低（lower），并将来自这些线程的操作交错到单个指令流中，形成每个线程的更好执行流水线，以隐藏内存访问延迟。



Loop oriented optimizations

后端也应用了面向循环的优化，以生成针对目标硬件的高效代码。由于Halide和LLVM（集成了多面体方法）已经整合了这些优化技术，一些DL编译器在其后端中利用了Halide和LLVM。**面向循环的优化**中应用的关键技术包括循**环融合、滑动窗口、切片、循环重排序和循环展开。**

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，这种优化是通过Halide调度或多面体方法来执行的，而Glow则通过其算子堆叠（operator stacking）来进行循环融合。

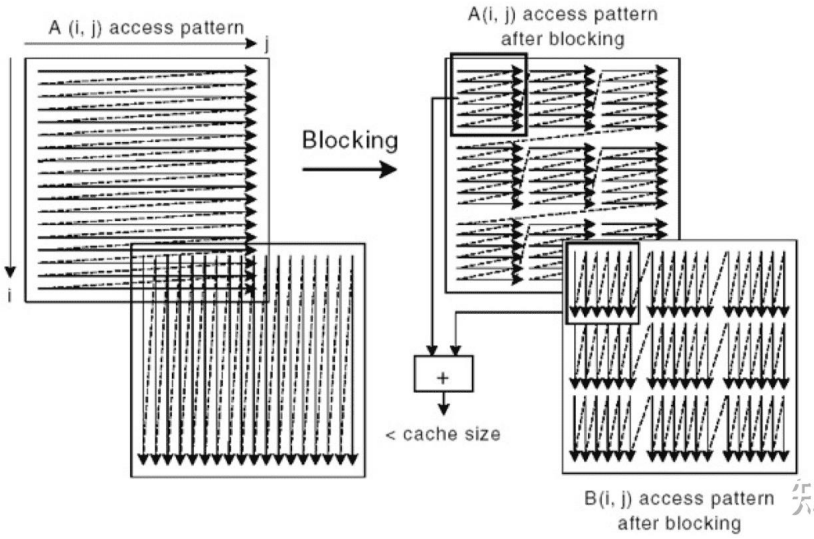
```
for i = 0, n
    A(i) = a(i) + b(i);
    c(i) = 2 * a(i);
endfor
for i = 1, n - 1
    D(i) = c(i) + a(i);
endfor

# After fusion

A(0) = a(0) + b(0);
c(0) = 2 * a(0);
A(n - 1) = a(n - 1) + b(n - 1);
c(n - 1) = 2 * a(n - 1);
for i = 1, n - 1
    A(i) = a(i) + b(i)
    c(i) = 2 * a(i)
    D(i) = c(i) + a(i)
endfor
```


2) Sliding windows

滑动窗口是Halide采用的一种循环优化技术。其核心概念是在需要时计算值并即时存储以进行数据重用，直到不再需要为止。由于滑动窗口交错了两个循环的计算并使它们变为串行，因此它是并行性和数据重用之间的一种权衡。

3) Tiling

切片将循环分割为多个小块，因此循环被分为外循环（iterating through tiles）和在内循环（iterating inside a tile）。这种转换通过**将小块适应硬件缓存（fitting a tile into hardware caches）**，实现了小块内更好的数据局部性。由于小块的大小是硬件特定的，许多DL编译器通过自动调优来确定小块的切片模式和大小。



```
for j = 0, n
    for i = 1, m
        A(i) += B(j)
    endfor
endfor

# After Tiling

for j_o = 0, m, T:
    for i = 0, n:
        for j_i = j_o, j_o + T:
            A[i] += B[j_i]
        endfor
    endfor
endfor
```


4) Loop reordering

循环重新排序（也称为循环排列）改变了嵌套循环中的迭代顺序，可以优化内存访问，从而增加空间局部性。它需要考虑数据布局和硬件特性。然而，当迭代顺序中存在依赖关系时，执行循环重新排序是不安全的。

```
for i = 1, n
    for j = 1, m
        A(i,j) = B(i, j) * C(i, j)
    endfor
endfor

# After Reorder

for j= 1, m
    for i = 1, n
        A(i, j) = B(i, j) * C(i, j)
    endfor
endfor
```


5) Loop unrolling

循环展开可以将特定的循环展开成固定数量的循环体副本，使编译器能够应用激进的指令级并行性。通常，循环展开与循环分割（loop split）结合使用，首先将循环分割为两个嵌套循环，然后完全展开内部循环。

```
for j = 1,2 * n
    for i = 1,m
        A(j) = A(j) + B(i)
    endfor
endfor

# After Unrolling

for j = 1,2 * n by 2
    for i = 1,m
        A(j) = A(j) + B(i)
        A(j+1) = A(j+1) + B(i)
    endfor
endfor
```


Parallelization

由于现代处理器通常支持多线程和SIMD并行性，编译器后端需要利用并行性以最大化硬件利用率，实现高性能。Halide使用了一种称为"parallel"的调度原语，用于指定循环的并行化维度，实现线程级并行化，并通过将循环维度映射到标记为"parallel" 并以 block 和 thread进行注释来支持GPU并行化。还有，它将大小为n的循环替换为n-wide的向量语句，通过硬件内在映射将其映射到硬件特定的SIMD指令。Stripe开发了多面型模型的一种变体，称为嵌套多面型模型（nested polyhedral model），它引入了并行多面体（parallel polyhedral）块作为迭代的基本执行单元。在此扩展之后，嵌套多面型模型可以检测在切片和步进的多个层级之间的层次并行化。此外，一些深度学习编译器依赖于手工编写的库，如Glow，或硬件供应商提供的优化数学库（在4.4.3节中讨论）。与此同时，Glow将向量化任务交给LLVM，因为当提供张量维度信息和循环迭代次数时，LLVM的自动向量化器效果很好。然而，要完全利用编译器后端的并行性可以通过应用更多与领域特定的深度学习模型相关的知识，从而在付出更多的工程努力的代价下实现更高的性能。

4.4.2 Auto-tuning

由于硬件特定优化中参数调优的巨大搜索空间，有必要利用自动调优来确定最佳的参数配置。在本文调研的DL编译器中，TVM、TC和XLA都支持自动调优。通常，**自动调优**的实现包括四个关键组成部分，如**参数化、成本模型、搜索技术和加速**。

> *对于给定的程序和目标架构，找到最优的编译优化方法。使用哪些优化方法？选择什么参数集？用什么顺序应用优化方法以达到最佳性能？*

Parameterization

> *参数化：对调度优化问题进行建模，参数化优化空间一般由可参数化变换（loop）的可能参数取值组合构成，因此需要调度原语进行参数化表示。Halide 将算法和调度解耦，TVM 则提供调度模板。*

1）Data and target

数据参数描述了数据的规范，例如输入形状。目标参数描述了硬件特定的特征和约束，在优化调度和代码生成过程中要考虑这些参数。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。

2）Optimization options

优化选项包括优化调度和相应的参数，例如循环导向的优化和切片大小。在TVM中，既考虑了预定义的调度和参数，也考虑了用户定义的调度和参数。然而，TC和XLA更喜欢对优化进行参数化，这与性能具有很大的相关性，并且可以在后期以较低的成本进行更改。例如，小批量维度通常是其中之一的参数，通常被映射到CUDA中的网格维度，并可以在自动调优过程中进行优化。

Cost model

成本模型：用来评价某一参数化下的调度性能。可以从运行时间、内存占用、编译后指令数来评价。

应用于自动调优中的不同成本模型的比较如下。

1）Black-box model

这种模型只考虑最终的执行时间，而不考虑编译任务的特性。构建一个黑盒模型很容易，但在没有任务特性的指导下，很容易产生更高的开销和非最优的解决方案。TC采用了这种模型。

2）ML-based cost model

基于机器学习的成本模型是一种统计方法，用于利用机器学习方法预测性能。它使得模型可以在探索新配置时进行更新，从而提高预测准确性。例如，TVM和XLA分别采用了这种模型，如梯度树提升模型（GBDT）和前馈神经网络（FNN）[47]。

3）Pre-defined cost model

基于预定义成本模型的方法期望构建一个完全建立在编译任务特征之上的完美模型，并能够评估任务的整体性能。与基于机器学习的模型相比，预定义模型在应用时产生的计算开销较小，但需要大量的工程努力，在每个新的DL模型和硬件上重新构建模型。

Searching technique

搜索算法：确定初始化和搜索空间后，在搜索空间找找到达到性能最优的参数配置。常用的搜索算法有遗传算法、模拟退火算法、强化学习等。

1）Initialization and searching space determination - 初始化和搜索空间的确定

初始选项可以随机设置，也可以基于已知的配置，例如用户给出的配置或历史优化配置。就搜索空间而言，应在自动调优之前明确指定。TVM允许开发者利用领域特定的知识来指定搜索空间，并基于计算描述为每个硬件目标提供自动的搜索空间提取。相比之下，TC依靠编译缓存和预定义规则。

2）Genetic algorithm （GA) - 遗传算法

GA方法将每个调优参数视为对照组，每个配置视为候选解。根据适应度值，新的候选解通过交叉、突变和选择的迭代生成，这种方法受自然选择过程的启发，是一种元启发式方法。最终得到最优的候选解。交叉、突变和选择的比率用于控制探索（exploration）和利用（exploitation）之间的权衡。TC在其自动调优技术中采用了遗传算法。

3）Simulated annealing algorithm (SA) - 模拟退火算法

SA（模拟退火）也是一种受退火过程启发的元启发式算法。它允许在逐渐减少的概率下接受较差的解决方案，可以在固定数量的迭代中找到近似的全局最优解，并避免陷入精确的局部最优解。TVM在其自动调优技术中采用了SA算法。

4）Reinforcement learning (RL) - 强化学习

强化学习（RL）通过探索和利用之间的权衡，在给定环境中学习如何最大化奖励。Chameleon（基于TVM构建）在其自动调优技术中采用了强化学习算法。

Acceleration

自动调优加速

1）Parallelization

加速自动调优的一个方向是并行化。TC提出了一种多线程、多GPU的策略，考虑到遗传算法需要在每一代中评估所有候选解。首先，它将候选配置入队并在多个CPU线程上进行编译。生成的代码在多个GPU上并行评估，每个候选解都拥有自己的适应度，用于父节点选择步骤。在完成整个评估之后，新的候选解被生成，并且新的编译作业被入队，等待在CPU上进行编译。类似地，TVM支持交叉编译和RPC，允许用户在本地机器上进行编译，并在多个目标上以不同的自动调优配置运行程序。

2）Configuration reuse

加速自动调优的另一个方向是重复使用先前的自动调优配置。TC通过编译缓存将已知的生成的最快代码版本与给定的配置对应起来，以供后续使用。在每次编译过程中，在进行每个内核优化之前查询缓存，并且只有在缓存未命中时才会触发自动调优。类似地，TVM生成一个日志文件，其中存储了所有调度运算符的最佳配置，并在编译过程中查询日志文件以获取最佳配置。值得一提的是，TVM对于Halide IR中的每个运算符（例如conv2d）都进行自动调优，因此最优配置是针对每个运算符单独确定的。

4.4.3 Optimized Kernel Libraries

使用手工编写的加速库。

有几个经过高度优化的内核库被广泛用于加速各种硬件上的深度学习训练和推理。来自英特尔的DNNL（之前称为MKL-DNN），来自NVIDIA的cuDNN和来自AMD的MIOpen都是广泛使用的库。根据硬件特性（例如AVX-512指令集、tensor cores），针对计算密集型的基本操作（例如卷积、GEMM和RNN）和受内存带宽限制的基本操作（例如batch normalization, pooling, and shuffle）进行了高度优化。还支持可定制的数据布局，以便轻松集成到深度学习应用程序中，避免频繁的数据布局转换。此外，还支持低精度的训练和推断，包括FP32、FP16、INT8和非IEEE浮点格式bfloat16 [45]。其他定制化的深度学习加速器也维护着它们特定的内核库[43,

57]。

现有的深度学习编译器，如TVM、nGraph和TC，可以在代码生成过程中生成对这些库的函数调用。然而，如果深度学习编译器需要利用现有的优化内核库，它们首先应该将数据布局和融合风格转化为预定义在内核库中的类型。这种转换可能会破坏最优控制流。此外，深度学习编译器将内核库视为黑匣子。因此，在调用内核库时，它们无法跨运算符应用优化（例如，运算符融合）。总的来说，使用优化的内核库会在计算可以满足特定高度优化原语的情况下实现显著的性能提升，否则，它可能受到进一步优化的限制，并且会导致性能不够好。

4.4.4 Discussion

后端负责基于低级IR进行裸机优化和代码生成。虽然由于不同的低级IR设计，后端的设计可能会有所不同，但它们的优化可以分为硬件特定的优化、自动调优技术和优化的内核库。这些优化可以单独或结合进行，通过利用硬件/软件特性来实现更好的数据局部性和并行化。最终，DL模型的高级IR将转化为在不同硬件上高效的代码实现