融合循环划分的张量指令生成优化

梁佳利　　华保健 *　　苏少博
中国科学技术大学软件学院
{liangjl, sabre}@mail.ustc.edu.cn　　bjhua@ustc.edu.cn*

摘要—张量编译器支持将算子的张量描述和计算调度编译为目标硬件的代码。为加速张量运算，深度学习领域专用处理器被设计为包含特殊指令的专有架构，支持多核并行，多级专用内存架构和张量计算，在硬件之上还有与硬件特性紧密相关的张量指令集。在这样复杂的架构上，张量指令的使用有着许多约束与限制，并存在以下问题和挑战：首先，因计算任务划分或数据切块等循环分段引入的条件分支，增加了模式匹配难度；其次，张量指令有对齐，数据布局等硬件约束。针对上述问题和研究挑战，提出了一种融合循环划分的张量指令生成优化算法。算法通过划分循环区间，来消除因任务划分或数据切分引入的条件分支；通过补零、等价指令替换和添加额外计算，解决指令和硬件约束；并使用模式匹配的方法生成张量指令。研究并扩展开源深度学习编译器 TVM(Tensor Virtual Machine) 0.7 版本，实现了支持 DianNao 架构机器学习加速器的张量指令生成的编译器原型系统。为评测算法的有效性，在 DianNao 架构机器学习加速器硬件平台上，对逐元素二元张量操作算子、原地一元张量操作算子和卷积操作算子三类算子的算子性能和开发效率进行了测试，实验结果表明三类算子性能平均加速比为 125.00%，最大加速比为 194.00%，开发效率最高提升 7 倍。

关键词—深度学习, 张量编译器, 领域专用处理器, 张量化
中图法分类号—TP311

***Abstract*—The tensor compiler compiles the tensor algorithm and schedule of the operator into the code of the target hardware. In order to accelerate tensor operation, the special processor in the field of deep learning is designed as a special architecture containing special instructions, which supports multi-core parallel, multi-level special memory architecture and tensor calculation. On top of the hardware, there is a tensor instruction set closely related to the characteristics of the hardware. In such a complex architecture, the use of tensor instructions has many constraints and limitations, and there are the following problems and challenges: firstly, the conditional branches introduced by loop tiling such as computing task division or data chunking increase the difficulty of pattern matching; Secondly, tensor instructions have hardware constraints such as alignment and data layout. To solve the above problems and research challenges, an optimization algorithm of tensor instruction generation based on loop partitioning is proposed. By dividing the loop interval, the algorithm eliminates the conditional branches introduced by task division or data segmentation; The instruction and hardware constraints are solved by filling zeros, replacing equivalent instructions and adding additional calculations; The tensor instruction is generated by pattern matching method. This paper studies and extends the open source deep learning compiler TVM version 0.7, and implements a compiler prototype system supporting tensor instruction generation of Diannao architecture machine learning accelerator. In order to evaluate the effectiveness of the algorithm, the operator performance and development efficiency of element-wise binary tensor operator, in-place unary tensor operator and convolution operator are tested on the Diannao architecture machine learning accelerator hardware platform. The experimental results show that the average speedup of the three types of operators is 125.00%, and the maximum speedup is 194.00%, The maximum development efficiency is increased by 7 times.**

***Key words*—deep learning; tensor compiler; domain-specific processor; tensorization**

通讯作者：华保健（bjhua@ustc.edu.cn）

I. 引言

深度学习框架如 Tensor Flow [1]、PyTorch [2]、MXNet [3] 和 Caffe [4] 等，通常把深度学习网络 [5] [6] [7] [8] 描述为有向无环的计算图，图中的每个节点对应于一个张量操作，这些张量操作会被编译为目标平台的算子实现。目标平台算子库提供常见算子的优化实现，但是对于用户定制的算子、新架构上的算子实

现，或新的算子优化方案，则需要用户花费大量的时间精力去实现这些算子。张量编译器 [9] [10] [11] [12] [13] [14] 支持将算子的张量描述和计算调度编译为专用加速硬件 [15] [16] [10] [17] 上的张量指令，这个编译过程称为张量化。这些张量编译器使得算子在不同硬件，如图形处理器（Graphics processing unit，GPU）、CPU 和现场可编程门阵列（Field Programmable Gate Array，FPGA）上，不但具有更好的可移植性，还能实现算子复用。

但是，现代的深度学习加速器，如 DianNao [16] [18]、Ascend [19]、[20] 和张量处理器（Tensor Processing Unit，TPU）[21] 等，都采用了多核并行架构，并且支持复杂内存架构和多种张量指令处理器；由于这些深度加速器的特殊硬件约束、及任务划分和数据切分相关问题，现有张量编译器并不能很好地张量化优化，导致编译生成的算子不能充分发挥硬件性能。在上述现代深度学习加速器上，进行张量化优化存在以下研究挑战：首先，循环分段会引入条件分支，导致指令生成的模式匹配困难。为充分利用深度学习加速器的并行处理能力和多级缓存架构，编译器将循环计算语句进行分段或分块，输入数据被均分为若干份，分别在循环中顺序处理或分配到多个核并行处理 [22] [23] [24] [25]；循环内会生成条件分支，以保证分段后的循环在实际运行时，不超出原来的迭代空间。这些条件分支语句，增加了张量化优化时模式匹配的难度，甚至导致模式匹配失败。其次，深度学习编译器难以满足深度学习加速器的指令和硬件约束。由于深度学习加速器硬件的特殊性，部分张量指令对于输入数据有对齐或数据布局的特殊约束 [26] [27] [28]，这些约束使得编译器生成高效张量指令更加困难，也难以使用现有调度策略。

针对上述问题和研究挑战，本文提出了一种新的融合循环划分的张量指令生成优化算法。首先，算法通过使用常数零等合法常数值填补数据，使用语义相等但约束更少的指令来替换目标指令，并通过增加额外计算来解决深度学习加速器的硬件约束难以满足的问题；其次，算法收集循环分段引入的条件分支中的布尔表达式，并通过求解布尔表达式，得到循环控制变量的范围区间，进一步将循环划分为该范围区间，从而消除条件分支；最后，算法使用树模式匹配策略，生成深度学习加速器上的具体张量指令。

本文基于开源编译器 TVM 0.7 版本 [10]，给出了上述算法的原型系统实现。原型系统利用 TVM 的张量表达式、TVM IR（Intermediate representation）中间表示等相关数据结构，通过增加新的优化遍，生成 DianNao 架构上的机器学习加速器张量化指令。

为评测本文算法对算子性能的加速效果、以及对算子开发效率的提升作用，本文在 DianNao 架构机器学习加速硬件平台上进行了实验评测。评测选取了逐元素二元张量操作算子、原地一元张量操作算子和卷积操作算子三类单算子作为评测目标；以上三类算子抽象了常见深度学习算子张量操作的基本特性，在各种深度学习网络中都属于高频出现的算子，因此，对这三类算子的评测结果能够说明本算法的有效性。实验结果表明：本文算法对算子性能平均加速比为 125.00%，最大加速比为 159.00%，开发效率最大提升 7 倍。这说明本文算法通过扩大算子的调度空间，在提升了算子性能的同时，提高了算子开发效率。

总结起来，本文主要贡献如下：

1) 提出了一种新的融合循环划分的张量指令生成优化算法；
2) 基于扩展 TVM 开源深度学习编译器，设计并实现了该算法的原型系统；
3) 对原型系统在三类重要的深度学习算子上，进行了测试与分析，验证了算法的有效性。

本文余下内容组织如下：第II小节介绍相关工作；第III小节介绍研究背景与动机；第IV节讨论融合循环划分的张量代码生成优化算法的设计，及其在 TVM 深度学习编译器上的原型系统实现；第V小节给出实验结果，并对实验结果进行讨论分析；第VI小节总结全文。

II. 相关工作

张量化和向量化是一种重要的将数据移动和计算映射到深度学习加速器硬件指令的编译技术，已经有大量文献和相关研究工作。

近年来，多面体编译优化技术在深度学习加速器上得到了广泛应用。Bhaskaracharya [29] 等研究了使用多面体编译优化技术，为 volta Tensor cores GPU 平台上自动生成矩阵乘张量指令的方法。PolyDL [30] 编译器使用多面体优化，在 CPU 上为计算的内层循环进行向量化优化；AKG（Auto Kernel Generator）[14] 编译器使用多面体优化技术，生成 Ascend 硬件上的张量指令。

已有的基于多面体优化的编译技术，都只为部分张量指令提供了自动张量化方法，难以扩展到新的张量指令。

swATOP [13] 和 TVM [10] 使用张量化原语，使用硬件张量内建指令，对指定张量操作进行替换。UNIT [31] 实现了一种指令张量化算法，在张量领域专用语言的级别，识别并替换计算模式。然而，由于这类算法过早地将计算替换为张量指令，导致很多优化的时机易被错过；并且，在张量领域语言级别进行调度时，复杂的调度策略会产生复杂的边界条件检查，导致后续指令生成失败。TVM [10]、halide [9] 和 LLVM（Low Level Virtual Machine）编译器实现了消除或外提循环内判断条件的优化算法，但是这些算法的目标是通过消除循环内条件分支，来减少循环内条件判断的次数，这类优化的负面作用是导致许多标量计算无法被张量化。Zhao [25] 等提出了一种新的算法，通过改变循环边界来保证循环分块后的迭代的正确性，但是该算法不支持循环重排等改变循环嵌套次序的优化。

Nuzman [26] [28] 等提出了一种针对不连续数据的向量化算法，给出了在单指令多数据（Single Instruction Multiple Data，SIMD）平台上实现自动向量化时，解决对齐约束和归约操作的方案。Eichenberger [27] 等针对 SIMD 架构，讨论了加载和存储操作时的对齐约束问题。

AKG [14] 和张量加速引擎（Tensor Boost Engine，TBE）编译器通过在张量领域专用语言上新增对齐原语，在将张量程序翻译为 TVM IR 时，改变循环范围以满足对齐约束。Gao [13] 等提出了补零和调整张量指令参数两个方法处理简单的指令约束。但上述算法能够解决的约束类型比较单一，难以解决补齐操作和卷积核数据布局等特殊约束。

III. 背景与研究动机

本小节讨论本文工作的研究背景，并给出本文工作的研究动机。

A. 张量表达式

张量表达式是一种深度学习领域专用语言，它支持用户自定义算子的开发与优化。深度学习编译器（如 TVM [10] 等），通过提供计算图级别 [32] 的优化和算子张量操作级别的优化，将张量表达式编译为深度学习加速器硬件上的高效代码。张量表达式遵循了 Halide [10] 提出的计算描述与调度优化分离的思想，将算子的实现分为两部分：

1) 描述算子行为的数学定义；
2) 指定计算调度策略的调度原语。

如下张量表达式，给出了一个固定规模的张量加法算子实例：

```
shape = [14, 14]
A = placeholder(shape)
B = placeholder(shape)
C = compute(shape, lambda i, j: A[i][j]+B[i][j])
s = create_schedule(B.op)
xx, yy = C.op.axis
xxo, xxi = s[C].split(xx, nparts=4)
thread_x = thread_axis("tid")
s[C].bind(xxo, thread_x)
s[C].pragma(xxi, "Intrin", "TADD")
```

`placeholder` 语法定义一个形状为 `shape` 的符号变量，`compute` 语法定义张量 `C` 如何由张量 `A` 和 `B` 计算得到。其中的 `lambda` 表达式定义张量 `C` 每一个元素 `C[i][j]`，由张量 `A` 和 `B` 对应位置的元素 `A[i][j]` 和 `B[i][j]` 相加得到；`C.op.axis` 是计算 `C` 时，两层循环的“轴”，即 `xx` 为外层循环，`yy` 为内层循环。接着，张量表达式使用调度原语 `split`，将张量加法所对应的循环划分为 4 份计算任务，并将这 4 份计算任务绑定到 4 个线程上执行。线程号变量由 `thread_axis` 定义。最后使用 `pragma` 原语，在循环 `xxi` 上添加类似“`TADD`”的特殊标记，深度学习编译器可在编译优化阶段，根据该标记进行特定优化。

上述张量表达式程序，会被深度学习编译器编译为类似如下的中间表示：

```
thread(tid, 4)
allocate(A, 56)
allocate(B, 56)
allocate(C, 56)
for (i, 0, 4)
  pragma="TADD"
  for (j, 0, 14)
    if (((tid*4) + i) < 14)
      C[(((tid*56)+(i*14))+j)] =
        A[(((tid*56)+(i*14))+j)] +
        B[(((tid*56)+(i*14))+j)]
```

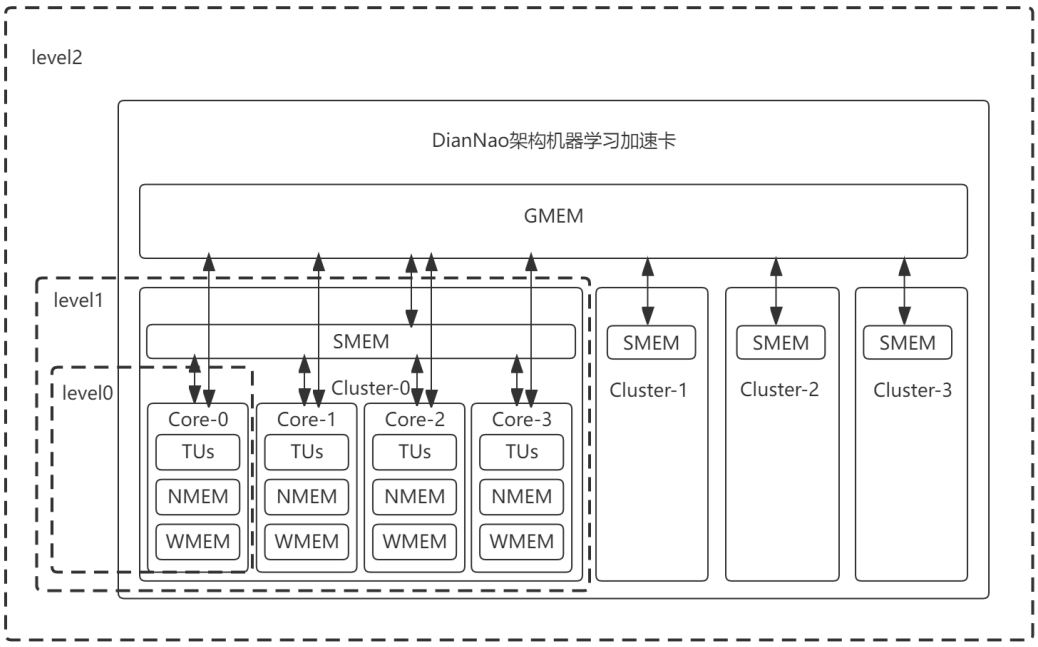


图 1: DianNao 架构图

其中 `tid` 标识不同线程，`allocate` 为数据分配片上空间；循环内生成的条件分支 `if (((tid*4) + i) < 14)` 保证了算子代码执行过程中，不会超出循环的迭代空间范围。

B. DianNao 架构机器学习加速器

本文基于 DianNao 架构的机器学习加速器，来研究和实现融合循环划分的张量指令生成优化算法，但该算法也为其它架构的深度学习专用加速器上的张量编译器实现提供了指导。图 1展示了 DianNao 架构深度学习加速器的抽象硬件的核心，忽略了和本研究无关的信息。

DianNao 架构机器学习加速器可以分为 3 个层级：level0 代表核级别的层，每个加速核由本地存储 NMEM、本地权值专用存储 WMEM、和本地张量处理单元 TUs 组成，所有的计算操作和核内的数据搬移操作都在这个级别完成；level1 代表簇级别的层，每个簇由 4 个核、和核间共享内存 SMEM 组成；level2 代表加速卡级别的层，每个加速卡有四个簇和片外存储 GMEM 组成，GMEM 是所有核共享的存储。从 level0 到 level2，每个层级上的存储容量都逐级增加，但访存速度逐级减慢。TUs 为核内的张量处理单元，支持张量级别的逐元素二元操作指令（如加、减、乘等）、张量级别的原地一元指令（如激活操作 relu 等）、以及带有规约操作的复杂张量指令（如卷积等）。支持 DianNao 架构加速硬件的高级编程语言，提供了不同类型存储间张量数据拷贝的内建指令，这些张量指令能够代替循环标量计算，从而有效提升算子执行性能。

C. 研究挑战与动机

机器学习加速器为常见的张量操作都提供了内建张量指令，显著提高了张量操作的执行效率。为了让算子取得更好的数据局部性、计算并行性和更少的冗余计算，深度学习编译器会对算子采用复杂的调度策略，在对任务划分或数据切分进行调度时，循环内可能产生复杂的条件分支，这类条件分支导致编译器在生成张量指令时，识别计算模式和提取张量指令参数非常困难。另外，机器学习加速器都具有特殊的硬件设计和指令约束，而高层张量领域专用语言设计的关注点通常在于描述算子张量操作和调度优化，没有提供完备的描述硬件特性和指令约束的机制；因此，在张量编译器为高层张量语言生成机器学习加速硬件上的算子程序时，上述问题限制了算子调度空间的大小，阻碍了一部分张量操作编译为高效内建张量指令，最终导致生成的算子无法充

分利用硬件特性，不能充分发挥算子应有性能。

针对以上挑战，本文研究融合循环划分的张量指令生成优化算法，算法在张量指令生成前，消除循环内因任务划分或数据切分引入的条件分支；通过补零、等价指令替换和添加冗余计算等技术解决硬件约束问题；并通过使用模式匹配识别并生成机器学习加速器上的张量指令。本方法能够充分利用加速器的硬件特性，在满足硬件约束的同时，提升算子性能，扩大算子的可能调度空间，优化调度方案。

IV. 系统设计与实现

本小节讨论系统的设计与实现。我们在给出系统架构后（第IV-A 小节），重点讨论指令预处理和循环划分优化（第IV-B、IV-C小节），最后给出张量指令发射算法（第IV-D小节）。

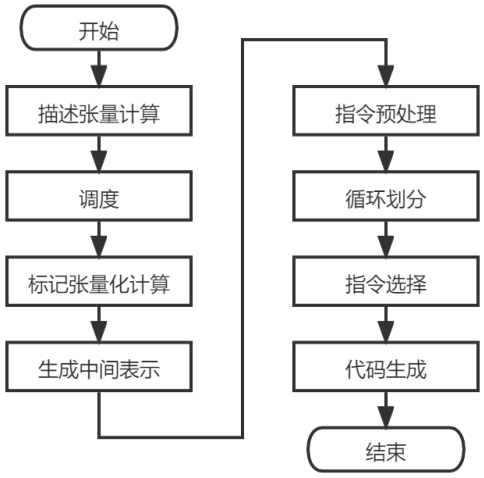


图 2: 张量化优化编译流程

A. 系统架构

图 2给出了引入张量化优化的编译系统执行流程。首先，流程从“开始”后的前三个步骤完成算子张量表达式的编写：（1）描述张量计算。描述张量计算的算子定义；（2）调度。使用一系列调度原语对张量操作进行调度；（3）标记张量化计算。对符合硬件内建张量指令语义的计算添加张量化标记。这些张量表达式是后续进行张量化优化的基础。

经过调度后的张量表达式具有与张量化标记等价语义，张量表达式被翻译为标量形式的中间表示—TVM IR；接着，系统使用指令预处理和循环划分两个优化步骤，来解决硬件约束、消除标记内的条件分支，并将中间表示变换为易于进行指令选择的形式。

指令选择模块识别表达式的计算模式，提取内建张量指令的相关参数，并将标记的中间表示改写为张量指令。最后，代码生成模块将张量化优化后的代码翻译为合法的目标平台的二进制代码。

接下来，我们对其中重要模块和优化算法分别进行讨论。

B. 指令预处理

指令预处理针对有对齐约束或数据布局约束等特殊计算约束的张量指令进行预处理。提前对这些约束进行预处理，方便后续规范化统一消除被标记张量化指令的计算内条件分支；指令预处理还通过添加额外计算，提前消除一部分复杂张量操作内的条件分支，从而减少因循环划分而导致的算子增大。

算法 1 指令预处理算法

输入： 原始的程序 P

输出： 优化后的程序

```
1:  procedure PreProcess(P)
2:      for each statement S ∈ P do
3:          pragma = getPragma(S)
4:          consType = getConstraintType(pragma)
5:          switch consType do
6:              case ALIGN:
7:                  ExpandAllocation(S)
8:              case LAYOUT:
9:                  S ← InstructionReplace(S)
10:             case COM:
11:                 S ← AddExtraCompute(S)
12:             case OTHERS:
13:                 continue
14:     return P
```

指令预处理根据张量指令标记获取约束信息，并分成以下三类情况进行相应处理。

1) 对齐约束: 解决方法为内存扩充。DianNao 架构机器学习加速器中的部分张量指令对张量元素有对齐约束，对于此类对齐约束，采用扩充分配内存的方法。算法根据张量标记得到对应张量指令的对齐要求后，将

把张量数据的已分配内存，扩大到满足对齐约束的大小。

2) 数据布局约束: 解决方法为等价指令替换。DianNao 架构机器学习加速器的部分张量指令，对于输入数据有特殊的数据布局约束。例如，卷积的权重数据必须按照一定的间隔摆放；补齐操作的输入数据必须连续摆放等。但是，高层的张量表达式侧重描述输入数据和输出数据之间的关系，对于中间结果的数据布局形式，并没有提供完备的原语描述。对于此类数据布局约束，编译器在生成 TVM IR 后，使用等价语义的一条或多条指令替换原始指令。如对于不连续的数据做补齐时，可以用一条零填充指令和带步长的数据拷贝指令替换。

3) 复杂张量计算内的条件分支: 解决方法为增加额外计算。卷积计算和矩阵乘计算都属于复杂张量计算，通过循环划分消除计算内的分支，会降低编译器的执行效率，同时也会增大生成代码的规模。但是，这类张量计算在执行时，每个输出数据只依赖于固定范围的相邻输入数据，因此可以通过增加额外计算的方法来消除分支。该方法扩充输入数据，扩大了循环的迭代空间，因此能够保证移除条件分支后，计算的语义仍然正确。

算法 1给出了指令预处理算法 PreProcess() 的伪代码。函数 $getPragma$ 获取语句的指令标记；函数 $getConstraintType$ 获取对应指令的约束类型。算法接受程序的中间表示 P 作为输入，逐条遍历程序 P 中的每条语句 S，并根据语句 S 对应的约束类型 $consType$ 进行相应的指令预处理：按照上述讨论，ALIGN 分支处理对齐约束，LAYOUT 分支处理数据布局约束，COM 分支处理复杂指令，OTHERS 分支代表不需要进行预处理的其它语句类型。每个分支分别调用了相应函数对语句 S 进行处理，其中，函数 ExpandAllocation() 扩充不符合对齐约束的数据空间，解决对齐约束；函数 InstructionReplace() 使用等价指令替换解决数据布局约束；而函数 AddExtraCompute() 通过增加冗余计算，来消除循环内的分支语句。

由于预处理函数 PreProcess() 只对程序 P 中的每条语句 S 处理一次，而每条语句都是 TVM IR 中的一个节点，因此其运行时间复杂度为 $O(N)$，其中 N 是 TVM IR 中间表示的节点数。

C. 循环划分

编译器对张量程序进行调度，会对数据切块，使得标记张量指令的语句内产生条件分支，导致指令选择时模式匹配失败。为保证每一个循环在实际执行时不超出原来循环的迭代空间，我们可以把变换后可能会遇到边界情况的段或块循环的范围，划分为恰好在原来循环迭代空间内和在原来循环迭代空间外，从而消除循环内的条件分支。

循环划分算法基于上述核心设计思想，通过将循环控制变量的范围划分为若干个不相交的区间，来判断这些区间中循环条件恒为真或假，从而消除标记内的条件分支。

循环划分算法会递归处理程序并记录循环控制变量的约束条件，解得循环内所有判断条件恒为真/假时对应的循环控制变量的解区间集合。剩余的区间则是无法确定判断条件真/假的解区间集合。按照这些子区间在原来循环区间内的出现顺序，依次处理：

1) 复制循环体，重新创建一个循环范围为子区间的循环语句；
2) 消除新的循环语句内恒为真/假的条件分支；
3) 递归地处理该循环的循环体。

最后将所有新的循环语句依次组合成新的语句并返回。

算法 2给出了循环划分算法 PartitionBranchLoop() 的伪代码。

该算法接收原始程序 P 和变量约束映射 C 作为输入，返回优化后的程序。算法首先判断程序 P 是否为循环，这里我们把带有线程标识的语句认为是一种特殊的循环，循环控制变量为线程号，循环范围为从 0 到线程数量减一。对于一个循环控制变量为 var、循环控制变量范围为 $[min, max]$、且循环体为 $body$ 的给定循环，算法在约束 C 中加入循环控制变量 var 的循环范围 $[min, max]$，并调用求解函数 FindPartition() 分别求解循环体 $body$ 中，与循环控制变量 var 相关的判断条件，在 C 的约束下，全部为真的 var 区间集合 $TSet$ 和全部为假的 var 区间集合 $FSet$。注意到函数 FindPartition() 的实现依赖于整数线性规划求解器（Integer Linear Programming，ILP），求解器对给定的一组整数线性约束，求得目标函数的最优解。虽然一般情况下的整数线性规划问题是 NP 难问题，但是本研究的中遇到约束求解问题只是整数线性规划问题的一个子集，即变量的取值范围限定为常数值，因此能够使用

算法 2 循环划分算法

输入： 原始的程序 P，变量约束 C

输出： 优化后的程序

1: **procedure** PARTITIONBRANCHLOOP(P,C)
2: **if** P is $loop(var, min, max, body)$ **then**
3: $C[var] = [min, max]$
4: $TSet$ = FindPartition($body, var, C$, **True**)
5: $FSet$ = FindPartition($body, var, C$, **False**)
6: $BSet$ = diff($C[var], (TSet \cup FSet)$)
7: $C[var] = \phi$
8: $Intervals$ = Sort($TSet \cup FSet \cup BSet$)
9: $S = [\ ]$
10: **for** each interval $IV : [a, b] \in Intervals$ **do**
11: s = MakeFor($var, [a, b], body$)
12: $C[var] = [a, b]$
13: s = EliminateCondition(s, C)
14: $s.body$ =PartitionBranchLoop($s.body, C$)
15: $C[var] = \phi$
16: $S.add(s)$
17: P = ComposeStmts(S)
18: **else**
19: $P.body$ = PartitionBranchLoop($P.body, C$)
20: **return** P

TVM 本身的提供的整数约束求解函数在线性时间内求解。

集合 $BSet$ 为循环控制变量 var 的迭代区间，等于集合 $TSet$ 和 $FSet$ 的所有区间并集的差集；完成后将约束 C 中关于循环控制变量 var 对应的约束置空。算法调用 Sort() 函数将 TSet、FSet 和 BSet 内所有区间按照区间起始值递增排序，并赋值给 $Intervals$。接着，算法依次处理 $Intervals$ 内的每个区间 $[a, b]$，使用 MakeFor() 函数创建循环体为 $body$，循环变量为 var，循环区间为 $[a, b]$ 的新的循环。算法在约束 C 中设置循环控制变量 var 的区间为 $[a, b]$，并使用 EliminateCondition() 函数消去能够被证明恒为真或假的条件分支，并对 $s.body$ 递归调用算法 PartitionBranchLoop() 进行处理，处理完 $s.body$ 后把 C 中 var 的约束置空，并把变换后的语句 s 追加到语句数组 S 末尾。算法完成对 $Intervals$ 内所有区间的处理后，调用函数 ComposeStmts() 依次组合 S 数组内的所有语句，最终得到程序 P。如果 P 不为循环（算法第 18 行），则算法递归调用 PartitionBranchLoop() 处理程序体 $P.body$，并最终返回优化后的程序 P。

函数 FindPartition() 具有线性时间复杂度 $O(N)$，其中 N 是 TVM IR 中间表示的节点数；排序函数 $Sort$ 是对整数区间进行排序，时间复杂度为 $O(M * \log M)$，其中 M 为区间长度；消除条件分支的函数 EliminateCondition() 需要遍历中间表示，因此为线性时间复杂度 $O(N)$。由于 PartitionBranchLoop() 采用递归实现，最终循环划分算法的时间复杂度为 $O(N^2 + N * M * \log M)$。

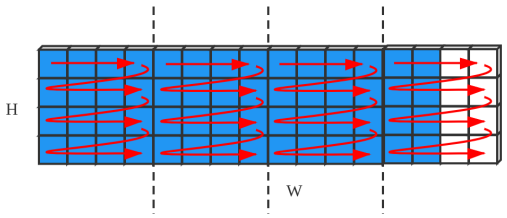
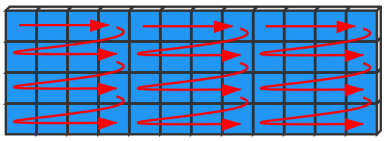
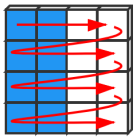


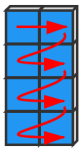
图 3: 数据拷贝任务在 W 方向上的划分



(a) tid<3 时的数据拷贝



(b) tid>=3 时的数据拷贝



(c) 最后一个核的合法迭代空间

图 4: 循环划分优化数据拷贝示意图

接下来讨论该算法运行的一个示例。以 DianNao 架构机器学习加速器上进行数据拷贝的任务分配为例，如图 3所示，对于 (H, W) 为 $(4, 14)$ 的数据，我们在 W 方向上对输出数据切分四份的计算任务被分配到 4 个核上，红色箭头为循环迭代访问数据的顺序。生成的示例代码如下（cid 和 tid 分别表示簇标识号和核标识号）：

```
thread(tid, 4)
allocate(AL, 16)
pragma = "MEMCPY"
for (i, 0, 4)
  for (j, 0, 4)
    if (tid*4+j<14)
      AL[(i*4)+j]=A[i*14+(tid*4)+j]
```

由于 14 不能被 4 整除，因此循环内的条件分支语句，保证了分块后的内部循环在实际执行时，不超过原来循环的迭代空间，即迭代访问数据时，不去访问图 3中白色部分的数据。循环划分算法在处理 tid 控制变量对应的特殊循环时，可以把循环划分为 tid<3 和 tid>=3 两部分，分别对应于图 4a和图 4b。图 4a部分数据在循环内迭代处理时，不会超过原来的循环迭代空间，因此判断条件恒为真，条件分支可以被删除。图 4b部分数据内的条件分支不恒为真，因此无法删除。上述程序经过循环划分算法优化后得到以下程序：

```
thread(tid, 4)
allocate(AL, 16)
if (tid<3)
  pragma = "MEMCPY"
  for (i, 0, 4)
    for (j, 0, 4)
      AL[(i*4)+j]=A[i*14+(tid*4)+j]
else
  pragma = “MEMCPY”
  for (i, 0, 4)
    for (j, 0, 4)
      if (j<2)
        AL[(i*4)+j]=A[i*14+(tid*4)+j]
```

循环划分算法在处理二个 MEMCPY 标记内的 j 控制变量对应的循环时，可以把循环划分为 $j < 2$ 和 $j >= 2$ 两部分，分别对应于图 4b中蓝色部分和白色部分迭代空间。图 4b中蓝色部分在循环内迭代处理时不会超过原来的循环迭代空间，因此判断条件恒为真，条件分支可以被删除。图 4b中白色部分是非法迭代空间，因此判断条件恒为假，条件分支可以被删除，变换后的迭代空间如图 4c所示。上述程序经过循环划分算法优化、并删除空循环语句后，得到以下程序：

```
thread(tid, 4)
allocate(AL, 16)
if (tid<3) {
pragma = "MEMCPY"
for (i, 0, 4)
  for (j, 0, 4) {
    AL[(i*4)+j]=A[i*14+(tid*4)+j]
else
  pragma = "MEMCPY"
  for (i, 0, 4)
    for (j, 0, 2)
      AL[(i*4)+j]=A[i*14+(tid*4)+j]
```

MEMCPY 标记的循环内部没有条件分支，在指令发射阶段生成如下张量指令：

```
thread(tid, 4)
allocate(AL, 16)
if (tid<3)
  memcpy(AL, A+tid*4, 14, 4, 4)
else
  memcpy(AL, A+tid*4, 14, 2, 4)
```

张量指令 memcpy(dst, src, stride, len, count) 的各个参数分别给出了拷贝目的地址、拷贝源地址、数据间隔步长、数据大小，拷贝数据次数。

D. 指令选择

指令选择使用扩展的树模式匹配算法，和指令选择的经典树模式匹配的算法相比，本文使用的算法为每一个张量指令都提供对应的树模式和参数提取规则，指令发射检查指令标记内语句，与对应指令的树模式匹配，并从中提取张量指令的所需的参数信息，最后用张量指令替换标记内的语句。指令发射分为三个具体步骤：（1）树模式识别。树模式是一个树状表示，匹配的目标是 TVM IR 的树状中间表示，匹配过程会递归的比较树模式和要匹配的目标语句这两颗树的所有节点；（2）参数提取。参数提取是在树模式识别过程中，记录各个循环和表达式计算的信息，根据参数提取规则，计算出张量指令的所有参数；（3）指令重写。若树模式匹配和参数提取过程都成功完成，则生成张量指令替换标记内的计算，否则，算法不对标记内语句做修改。

假设树模式的节点数为 K，而要被匹配的程序的树节点数为 N，由于算法在最坏情况下，需要对程序中的每个节点都进行树模式的匹配，因此算法的最坏运行时间复杂度为 $O(K * N)$；而由于树模式的节点数 K 一般是一个小的常数，因此算法实际上具有线性时间复杂度 $O(N)$。

V. 实验与结果分析

本小节详细介绍所进行的实验，并对实验结果进行分析。第V-A小节给出实验要回答的研究问题；第V-B小节给出实验平台的各项配置；第 V-C小节给出实验的数据集；第V-D讨论实验结果，并对不同的实验结果进行分析；第V-E小节对实验进行总结。

A. 研究问题

实验主要回答以下两个研究问题：

（1）**算子性能**。算子库提供算子的实现结合了专家的经验，使用智能编程语言编写而成，且支持不同的输入规模，这类算子在常用的输入规模上通常做了较好的优化，但是对于其他不常见的输入规模，则并不是最优的实现。本次实验探究与算子库提供的算子手工优化实现相比，本文提出的张量化优化算法，对算子性能提升的效果如何？对算子性能的评测，将通过算子运行时间来衡量。

（2）**开发效率**。本文提出的指令选择的张量化优化算法，对于算子开发效率的影响如何？对开发效率的评测，将通过算子实现代码行数（即工作量）来衡量。

B. 实验平台

本次实验都在基于表（I）所示配置的服务器上进行。协处理器为 DianNao 架构机器学习加速器，其核心架构已经在第III小节进行了描述。本实验中的协处理器 SMEM 大小配置为 2MB，NMEM 大小配置为 512KB，WMEM 大小配置为 1MB。

表 I: 实验环境配置

	参数配置
操作系统	Ubuntu 16.04.4 LTS
CPU	Intel® Xeon® Gold 6154 CPU @ 3GHz，144 逻辑核
内存	1TGB
协处理器	DianNao 架构机器学习单元

C. 数据集

本次研究选择了卷积算子、二元逐元素张量运算算子，和一元原地张量运算算子这三类算子作为实验评测对象，将 DianNao 架构机器学习加速器上的算子库优化实现作为基准测试集；为统计并分析本文算法对算子性能和算子开发效率的影响，在实验中，我们为每一类算子都提供了 16 个不同的输入规模的测例，总共 48 个不同规模的测例。这三类算子抽象了大部分深度学习算子中张量操作的基本特性，在各种深度学习网络中都属于高频出现的算子类型。实验选择的测试数据规模，也包含了会造成数据切分不均的各种可能输入规模。因此，在本次选择的数据集上进行张量化优化的实验，能够充分说明本文算法的有效性。

D. 实验结果与分析

1) 算子性能结果: 为了评测算子的性能，本实验对三类算子的不同输入规模测例进行了测试，记录每个测例的运行时间，记 T_0 为基准算子运行时间，T_1 为本次实验的原型系统生成算子的运行时间。对每个测例，计算其相对性能加速比 S 为

$$S = \frac{T_0}{T_1}$$

并定义每个算子在所有测例上的相对性能加速比 A，为其在单个测例上加速比 S_i 的几何平均值

$$A = \left(\prod_{i=1}^{n} S_i\right)^{\frac{1}{n}}。$$

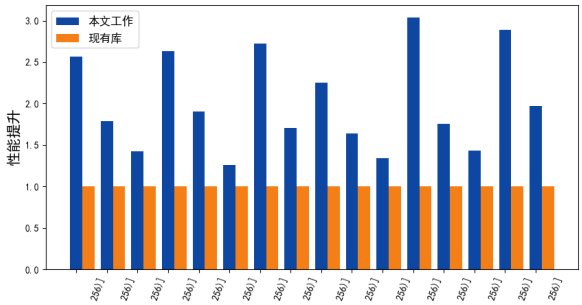


图 5: 卷积性能加速比

图 5、图 6和图 7 分别展示了卷积、逐元素二元张量运算和原地一元张量运算三类操作的性能实验结果。横坐标代表输入算子的规模，纵坐标表示相对性能加速比。图中蓝色柱子表示本文的原型系统生成的算子性能，黄色柱子表示算子库的基准性能。卷积算子的最高加速比为 203.63%，输入规模为 ((55, 55, 128),(128, 3, 3, 256))，最低加速比为 125.00%，输入规模为 ((56, 56, 128),(32, 3, 3, 256))；逐元素二元张量运算操作的最高加速比为 149.93%，输入规模为 (256,56,56,128)，最低加速比为 104.09%，输入规模为 (255,55,55,1)；原地一元张量运算操作的最高加速比为 144.44%，输入规

表 II: 测试数据

算子类型	具体算子	算子规模
逐元素二元张量运算	Add Sub Mul	(256,56,56,15),(256,56,56,8),(256,56,56),(128,56,56),(64,56,56),(32,56,56),(256,28,28),(128,28,28), (255,55,55,14),(255,55,55,7),(255,55,55),(127,55,55), (63,55,55),(31,55,55),(255,27,27),(127,27,27)
原地一元张量运算	Relu	(256,56,56,15), (256,56,56,8),(256,56,56), (128,56,56),(64,56,56), (32,56,56), (256,28,28), (128,28,28), (255,55,55,14), (255,55,55,7),(255,55,55), (127,55,55),(63,55,55), (31,55,55), (255,27,27),(127,27,27)
卷积	Conv	((56, 56, 256), (128, 3, 3, 256)),((56, 56, 256),(64, 3, 3, 256)),((56, 56, 256) ,(32, 3, 3, 256)),((56, 56, 128), (128, 3, 3, 256)), ((56, 56, 128), (64, 3, 3, 256)),((56, 56, 128), (32, 3, 3, 256)),((56, 56, 64), (128, 3, 3, 256)), ((56, 56, 64), (64, 3, 3, 256)), ((55, 55, 256), (128, 3, 3, 256)),((55, 55, 256),(64, 3, 3, 256)),((55, 55, 256), (32, 3, 3, 256)),((55, 55, 128), (128, 3, 3, 256)), ((55, 55,128), (64, 3, 3, 256)),((55, 55,128), (32, 3, 3, 256)), ((55, 55, 64), (128, 3, 3, 256)),((55, 55, 64), (64, 3, 3, 256)

模为 (32,56,56)，最低加速比为 94.77%，输入规模为 (256,56,56,30)。

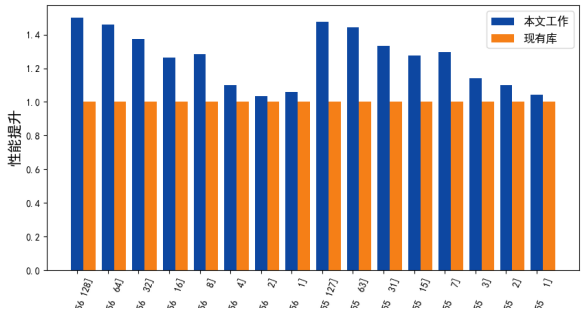


图 6: 逐元素二元张量运算加速比

图 8中展示了本文的原型系统生成算子相比于标准算子库的相对性能加速比。横坐标代表输入算子类别，纵坐标表示相对性能加速比。图中蓝色柱子表示本文的原型系统生成的算子性能，黄色柱子表示算子库的基准性能。三类算子的性能都优于算子库提供算子，其中卷积算子平均性能提升最为明显，加速比为 194.33%；其次是二元张量操作算子，平均加速比为 125.11%；一元张量操作算子 relu 的性能提升较小，平均加速比为 117.89%。

算子性能结果分析：本次实验对三类算子性能提升和提升效果差异进行了详细的分析，相比于算子库提供算子，本原型系统生成的三类算子性能提升的一部分共同原因是：算子库提供算子的实现结合了专家的经验，使用智能编程语言编写而成，为了支持不同的输入规模，该实现往往并非最优。而指令预处理和循环划分优化隐藏了指令和硬件约束，增大了 TVM 调度原语的调度空间。这使得在使用张量表达式编写算子时忽略和计

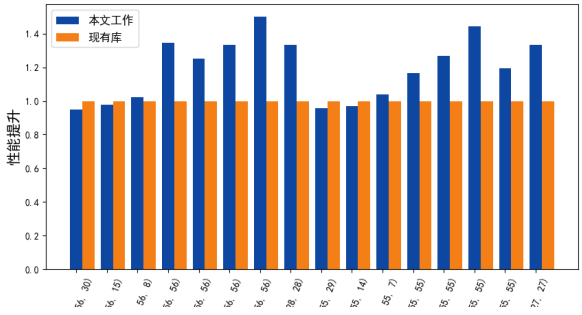


图 7: 原地一元张量运算加速比

算无关的信息，借助不断调整数据切分和计算规模来找到接近或更优于算子库实现的调度策略。

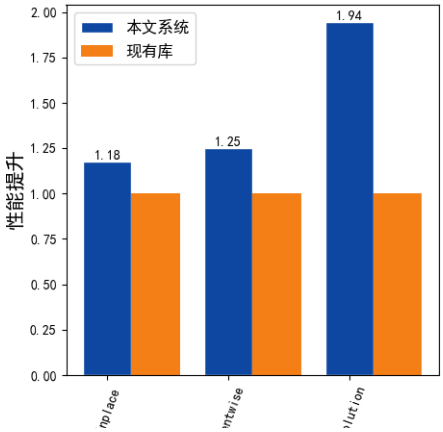


图 8: 算子相对性能加速比

二元张量操作算子的性能提升主要上述原因，(256,56,56,128) 输入规模最大，调度空间最大，由此带来的性能提升效果最明显。随着输入规模的减小，性

能提升效果逐渐下降。

相比于一元张量操作算子和二元张量操作算子，卷积算子的性能提升更大，这部分性能提升的来源主要是以下两方面。一方面原因是片外访存的减少。算子库提供算子没有充分利用 SHMEM 存储，导致中间数据在主机端和设备端的来回拷贝；并且 pad 操作是在主机端上完成。当输入数据不能全部拷入片上存储 CMEM 时，需要对数据进行切分，数据切分方案决定了片外 GMEM 访存次数。pad 操作和数据切分方案共同决定了片外数据拷贝大小。本文实现原型系统生成的算子使用容量更大的片上存储 SHMEM 作为缓存，做中间数据片上驻留以 SHMEM 大小为根据对数据进行第一次切分，以 CMEM 大小为根据对数据进行第二次切分，尽管没有改变需要进行拷贝的数据大小，但是减少了片外访存的次数。另一方面原因是 pad 操作的优化。指令预处理的对 pad 操作进行了等价指令替换，使用数据拷贝指令和置零指令，在数据拷贝的过程中完成了 pad 操作，既消除了额外的 pad 数据的拷贝，也避免了在主机端进行 pad 操作。除了这两部分主要原因外，循环划分算法也减少了因数据划分而产生的条件分支。((55, 55, 128),(128, 3, 3, 256)) 规模的算子运算量大，并且数据分块时难以避免产生条件分支，因此优化效果最为明显。其他规模较小的算子则优化效果有所下降。

原地一元张量操作算子相比于其他两类算子，性能提升较小，这是因为 relu 操作比较简单，计算量本身就很小，算子运行的时间开销基本由数据拷贝时间决定，因此性能提升有限。另外对于较大的输入规模，如 (256,56,56,30)，性能反而出现了下降，对于较小的输入规模，性能提升则更为明显。这个反常现象的原因是，算子库提供算子对于不同大小范围的输入，采用了不同的调度策略，因此在算子代码中会有较多的条件分支语句，而本原型系统生成的算子用 TVM 的张量表达式编写而成，支持运行时编译。虽然也是采用条件判断来使用不同的调度策略，但是在编译张量表达式到目标代码的过程中，输入规模已经确定，即输入规模为常数，因此判断条件也是常数，能够在编译过程中被优化消除。对于某一固定的规模，运行时编译的开销只在第一次测试时产生，多次测试的平均性能并不会受太大的影响。所以对于较小输入规模，分支指令带来的开销相比于整个算子运算较大，因此优化效果明显。对于较大的输入规模，分支指令带来的开销相比于整个算子运算较小，并且算子库简单算子的调度策略较为单一，因此优化效果不明显，甚至出现性能下降。

表 III: 算子代码规模

	卷积运算	二元张量运算	一元张量运算
算子库提供算子行数	687	149	49
张量程序行数	96	20	17
效率提升比	7x	7x	3x
算子结构	复杂	中等	简单

2) 开发效率实验结果: 表 III给出了本文研究的指令选择张量化优化对算子开发效率的实验结果。为了评测本算法对算子开发效率的提升，本实验记录了三类算子实现的源代码行数，用算子源代码行数来评估算子开发效率。实验结果表明，与智能编程语言实现算子相比，三类算子的张量程序源代码行数都明显小于前者；开发效率最多提升 7 倍，至少提升 3 倍。

开发效率实验结果分析：本次测试对简单，中等，复杂三类结构的算子都进行了测试，从测试结果来看复杂算子开发效率提升最大，简单算子开发效率提升最小。之所以产生这样的结果，是因为复杂结构算子有较为复杂的定义和优化调度策略，而张量领域专用语言提供的调度原语简化了这些优化调度过程，并且在编译过程中自动地解决了一些指令约束，大大降低了代码行数，开发效率提高明显；而简单结构的算子本身实现简单，留给张量编译器提升开发效率的空间较少，因此开发效率提高较少。从整体来看，本实验实现的原型系统能够明显提升算子开发效率。

E. 结论

本文在 TVM 0.7 版本上的编译器上，实现了张量指令生成优化算法的原型系统，并针对深度学习网络中高频出现的三类重要算子，进行了 48 种不同输入测例的测试，并收集实验结果；并围绕两个主要研究问题：算子性能和开发效率来展开实验结果分析。实验结果表明：本文提出的张量化指令生成优化方法，在编译过程中解决了因为硬件约束和条件分支而产生的张量指令生成困难的挑战，显著提升了算子性能，降低了算子实现的程序规模，提高了开发效率。

VI. 结语

张量编译器提供了在多种硬件后端快速开发生成深度学习算子的能力，但是在机器学习加速器这类多核

并行架构、有复杂内存架构和多种特殊张量计算单元的新硬件架构上生成张量指令时，存在因数据切块而产生条件分支和硬件约束难以满足等限制，这些限制常常导致指令张量化的失败。本文提出的融合循环划分的张量指令生成优化，通过指令前处理算法和循环划分算法解决了这一问题，并且在开源编译器 TVM 0.7 版本上，通过新增优化遍的形式实现了算法的原型系统，并进行了系统的实验评测，实验结果表明本文提出的算法能够通过扩大算子的合法调度空间和找到更优调度方案，在提升算子的性能，同时提升了算子的开发效率。

参考文献

[1] Mart in Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, and Others. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, and Others. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32:8026–8037, 2019.

[3] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[4] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678, 2014.

[5] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[7] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 1725–1732, 2014.

[8] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. In *Proceedings of the IEEE international conference on computer vision*, pages 4489–4497, 2015.

[9] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 519–530, 2013.

[10] T V M End-to-end Optimization, Deep Learning, Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: End-to-End Optimization Stack for Deep Learning. *University of Washington Technical Report UW-CSE-2017-12-01*, 2018.

[11] Changxi Liu, Hailong Yang, Rujun Sun, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. swtvm: Exploring the automated compilation for deep learning on sunway architecture. *arXiv preprint arXiv:1904.07404*, 2019.

[12] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[13] Wei Gao, Jiarui Fang, Wenlai Zhao, Jinzhe Yang, Long Wang, Lin Gan, Haohuan Fu, and Guangwen Yang. SwATOP: Automatically optimizing deep learning operators on SW26010 many-core processor. *ACM International Conference Proceeding Series*, 2019.

[14] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, and Others. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 1233–1248, 2021.

[15] Jack Choquette and Wish Gandhi. Nvidia A100 GPU: Performance & innovation for GPU computing. In *2020 IEEE Hot Chips 32 Symposium (HCS)*, pages 1–43. IEEE Computer Society, 2020.

[16] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News*, 42(1):269–284, 2014.

[17] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, and Others. T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 181–189. IEEE, 2019.

[18] Yunji Chen, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. DianNao family: Energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[19] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing: Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021.

[20] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019.

[21] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, and Others. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[22] Kathryn S McKinley, Steve Carr, and Chau-Wen Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):424–453, 1996.

[23] Emna Hammami and Yosr Slama. An overview on loop tiling techniques for code generation. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*, pages 280–287. IEEE, 2017.

[24] Daniel Cociorva, John W Wilkins, C Lam, Gerald Baumgartner, J Ramanujam, and P Sadayappan. Loop optimization for a class of memory-constrained computations. In *Proceedings of the 15th international conference on Supercomputing*, pages 103–113, 2001.

[25] Jisheng Zhao, Matthew Horsnell, Mikel Luján, Ian Rogers, Chris Kirkham, and Ian Watson. Adaptive loop tiling for a multi-cluster cmp. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 220–232. Springer, 2008.

[26] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices*, 41(6):132–143, 2006.

[27] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. Vectorization for SIMD architectures with alignment constraints. *Acm sigplan notices*, 39(6):82–93, 2004.

[28] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *International Symposium on Code Generation and Optimization (CGO'06)*, pages 11—-pp. IEEE, 2006.

[29] Somashekaracharya G Bhaskaracharya, Julien Demouth, and Vinod Grover. Automatic kernel generation for volta tensor cores. *arXiv preprint arXiv:2006.12645*, 2020.

[30] Sanket Tavarageri, Alexander Heinecke, Sasikanth Avancha, Bharat Kaul, Gagandeep Goyal, and Ramakrishna Upadrasta. Polydl: Polyhedral optimizations for creation of high-performance dl primitives. *ACM Transactions on Architecture and Code Optimization (TACO)*, 18(1):1–27, 2021.

[31] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. UNIT: Unifying Tensorized Instruction Compilation. *CGO 2021 - Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization*, pages 77–89, 2021.

[32] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 58–68, 2018.