北京大学

博士研究生学位论文

题目：面向 AI 芯片的编译分析与优化

姓　　名：郑思泽

学　　号：1901111270

院　　系：计算机学院

专　　业：计算机系统结构

研究方向：云端智能计算系统

导师姓名：梁云　教授

☑ 学术学位　　☐ 专业学位

二〇二四 年 四 月

版权声明

任何收存和保管本论文各种版本的单位和个人，未经本论文作者同意，不得将本论文转借他人，亦不得随意复制、抄录、拍照或以任何方式传播。否则，引起有碍作者著作权之问题，将可能承担法律责任。

摘要

近些年，人工智能算法的飞速发展，算力与算法、数据这三个要素一齐成为了近二十年来推动人工智能技术和应用发展的重要动力。然而，随着半导体工艺发展，摩尔定律逐渐失效，通用计算设备的性能提升放缓，已无法满足人工智能算法对于训练和推理的巨大算力、存储、带宽需求。在此背景下，专用架构计算设备兴起。以 TPU, NPU 等加速器为代表的专用加速器通过将有限的芯片资源分配给专用的计算、存储、互联单元，对于人工智能算法的训练和推理有极强的加速效果。这类为人工智能算法而专门设计的加速硬件称为人工智能芯片（简写为 AI 芯片）。AI 芯片的兴起对上层配套软件栈提出了新的需求。由于专用架构与通用架构在硬件设计、指令集设计、以及优化方法不同，先前对于通用硬件的软件栈的设计也难以直接迁移到 AI 芯片上，这就需要重新思考 AI 芯片上的软件编程和优化的问题。软件栈搭建有两种方式：依赖多种手工优化库的框架，以及依赖编译技术的自动化框架。由于手工优化库的时间长、效率低，编译框架的优势逐渐凸显，因此，近年来以编译技术为基础的框架层出不穷，而这类为 AI 芯片提供编译自动化支持的技术也被称为 AI 编译。

相比于面向通用芯片的传统编译，AI 编译所处的层次更高，涵盖模型层，算子层，指令层三个层次，在三个层次各有独特的技术挑战:

- **图层次：融合中计算与访存优化困难**。在算子融合优化问题中，需要编译器具有准确的性能评估和求解能力，通过最优化数据搬移量优化寻找高效的融合代码生成方案，在已有工作中仍无法解决融合数据流表达、分析、优化的难题。
- **算子层次：循环优化空间生成和搜索困难**。针对不同硬件芯片需要自动生成不同的优化空间，并通过统一的搜索方式在优化空间中寻找最优的循环调度策略，已有工作仍无法摆脱人为干预自动组合多种优化原语，也难以高效进行搜索。
- **指令层次：硬件指令缺少计算与访存抽象**。已有工作依赖人工定制模板使用 AI 芯片特殊指令。如何将 AI 芯片的专用特殊指令进行抽象并在编译器中进行表达，从而通过编译方式自动化在输入程序中寻找特殊指令使用机会，并通过编译器自动地、正确地、高效地将指令代码进行生成仍是困难的研究问题。

为了解决上述挑战，本文提出一种从上到下的完整编译技术路线，它贯穿 AI 编译的图、算子、指令三个层次，囊括了四个编译框架：Chimera，TileFlow，FlexTensor，和 AMOS。这四个框架相互配合、相互衔接，构成了从图到指令的统一编译框架。每个框架都基于统一的张量计算表达形式（Tensor Expression）和有向无环图抽象，并进一步设计符合各个编译层次的中间表达方式（Intermediate Representation, IR），以及配套设

计对应的编译优化流程或算法。具体而言，自上而下，本文的贡献包括：

- **在图层，提出了融合优化框架和融合性能评估模型**。通过基于块抽象的图融合编译技术，设计数据搬移量分析和优化算法，自动优化融合代码的计算和访存，大幅提高融合代码性能。在 CPU，GPU，NPU 等设备上，可以达到超越手工优化效果 2 倍以上加速。
- **在算子层，提出了算子循环优化和代码生成框架**。通过基于循环抽象的算子优化技术，设计算子自动调度算法和基于强化学习的调度空间搜索算法，有效提高生成代码的性能。在 CPU，GPU，FPGA 等芯片上测试结果表明相对于手工优化的代码，可以达到 1.5 倍以上加速。
- **在指令层，提出了指令抽象和生成框架**。通过基于计算访存抽象的指令生成技术，设计软硬件映射自动生成算法和指令生成正确性验证算法，能灵活地在多种芯片上进行特殊指令生成，并提高程序性能 1.3 倍以上。

关键词：编译器；AI 芯片；代码生成；自动优化

Compiler Design and Optimization for AI Chips

Size Zheng (Computer System and Architecture)
Directed by: Prof. Yun Liang

ABSTRACT

Recently, there has been a great advancement in artificial intelligence algorithm design and acceleration. Computational power, algorithms, and data have become the three crucial elements driving the development of AI technology and applications over the past twenty years. However, as semiconductor technology advances further, Moore's Law is gradually becoming obsolete, and the performance improvements of general computing devices are slowing, unable to meet the enormous computational power, storage, and bandwidth requirements of AI algorithms for training and inference. In this context, specialized architecture computing devices have emerged. Dedicated accelerators represented by TPUs, NPUs, and others, by allocating limited chip resources to specialized computing, storage, and interconnect units, have a significant acceleration effect on the training and inference of AI algorithms. This paper refers to such hardware designed specifically for AI algorithms as AI chips. The rise of AI chips introduces new demands for the supporting software stack. Due to differences in hardware design, instruction set design, and optimization methods between specialized and general architectures, the software stack designed for general hardware cannot be directly transferred to AI chips, necessitating a re-examination of the problems of software programming and optimization on AI chips. There are two approaches to building a software stack: frameworks relying on various hand-optimized libraries and those depending on compiler technology for automation. Due to the long time and low efficiency of hand-optimized libraries, the advantages of compiler frameworks have become increasingly apparent. Compared to traditional compilers aimed at general chips, AI compilers operate at a higher level, encompassing the model layer, operator layer, and instruction layer. AI compilation technology faces unique technical challenges at these three levels:

1. **Graph level: Difficulties in computation and memory access optimization during fusion.** In the problem of operator fusion optimization, the compiler needs to have accurate performance evaluation and solving capabilities, to find efficient fusion code generation schemes by optimizing data movement, a challenge that existing work has

not yet solved in terms of expressing, analyzing, and optimizing fused data flows.

2. **Operator level: Difficulties in generating and searching optimization spaces for loops.** For different hardware chips, it is necessary to automatically generate different optimization spaces and find the optimal loop scheduling strategy through a unified search method. Existing work still cannot escape manual intervention in automatically combining various optimization primitives, nor can it efficiently conduct searches.
3. **Instruction level: Lack of computational and memory access abstraction in hardware instructions.** Existing work relies on manually customized templates to use special instructions of AI chips. How to abstract the specialized special instructions of AI chips in the compiler and automatically find opportunities to use special instructions in the input program, and generate instruction code automatically, correctly, and efficiently remains a difficult research problem.

To address these challenges, this paper proposes a comprehensive compilation technology route from top to bottom, spanning the graph, operator, and instruction levels of AI compilation and encompassing four compilation frameworks: Chimera, TileFlow, FlexTensor, and AMOS. These four frameworks work together, forming a unified compilation framework from graphs to instructions. Each framework is based on a unified tensor computation expression and directed acyclic graph abstraction, and further designs intermediate representations (IR) suitable for each compilation level, as well as corresponding compilation optimization processes or algorithms. Specifically, from top to bottom, this paper's contributions include:

- **At the graph level, a fusion optimization framework and a fusion performance evaluation model were proposed.** Utilizing block abstraction-based graph fusion compilation technology, algorithms for analyzing and optimizing data movement were designed to automatically optimize the computation and memory access of fused code, significantly enhancing the performance of the fused code. On devices such as CPUs, GPUs, and NPUs, it is possible to achieve more than double the acceleration compared to manual optimization.
- **At the operator level, a framework for operator loop optimization and code generation was proposed.** Through loop abstraction-based operator optimization technology, an automatic scheduling algorithm for operators and a scheduling space search algorithm based on reinforcement learning were designed, effectively improving the performance of the generated code. Test results on chips like CPUs, GPUs, and FPGAs indicate that an acceleration of more than 1.5 times can be achieved compared to

manually optimized code.

- **At the instruction level, an instruction abstraction and generation framework was proposed.** By employing computation and memory access abstraction-based instruction generation technology, algorithms for automatic software-hardware mapping generation and instruction generation correctness verification were designed, allowing for flexible generation of special instructions on various chips and improving program performance by more than 1.3 times.

KEY WORDS: Compiler; AI Chips; Code Generation; Automatic Optimization

目录

表格索引

插图索引

第一章 引言

自从二十一世纪初开始，对于人工智能（Artificial Intelligence，AI）的研究再次成为热点，以深度神经网络为主的 AI 算法在多个领域（如视觉，听觉等）取得了超过人类水平的识别能力[1-4]。这类 AI 算法主要针对分类和标注任务，以有监督学习为主要方式。它们的参数量往往在 1 亿以内，计算量需求在 2 亿次浮点计算以内，在大量的设备上如桌面级 CPU，消费级 GPU，甚至边缘设备都可以进行部署。进入二十一世纪二十年代之后，生成式 AI 技术逐渐兴起，以 Transformer[5]，Diffusion[6]为代表的神经网络结构成为了 AI 算法的主要构成部分。这些新兴的 AI 算法在自然语言处理（Natural Language Processing, NLP），图片生成，视频生成等方面取得了重要的突破。值得注意的是，这些 AI 算法对于计算、存储、通信的需求增长远超先前的深度神经网络。例如，由 Meta 公司开源的 LLaMA-2[7]模型最小的参数量也有 70 亿参数，在半精度下，这些参数会占据约 14GB 的存储空间，在输入语句长度为一千左右时，推理时计算量约为 14 兆次浮点计算（1.4×10^{13}），训练时计算量则约为 42 兆次浮点计算。在国内，由百度推出的大模型文心一言拥有超过 2600 亿参数，阿里巴巴开源的通义千问模型有超过 140 亿参数，他们的计算量和存储量需求则会更高。在图 1.1中展示了近十年来国内外主要的 AI 算法的参数量变化趋势，AI 算法的计算量由输入数据量和参数数量共同决定，因此一般参数量越大的算法，计算量需求也会越大。图中展示了在 2018 年以前，AI 算法的参数量增长并不明显，但是在 2018 年之后，参数量以约每一年翻 10 倍（一个数量级）的速度增长。

另一方面，随着摩尔定律逐渐失效[8]，传统架构的硬件如 CPU 和 GPGPU 已经难以满足 AI 算法的巨大的计算、存储、通信需求。因此，近年来，大量高性能的 AI 芯片被提出，为 AI 算法提供算力、存储、通信的支持。如谷歌 TPU[9-10]，寒武纪 MLU[11]，英伟达 Tensor Core[12]，华为 NPU[13]等。这些 AI 芯片通过定制化的数据流支持 AI 计算的加速，例如谷歌 TPU 使用脉动阵列（Systolic Array）架构加速矩阵乘法，华为 NPU 使用 Cube 架构加速矩阵乘法和卷积。在图 1.2中展示了近十年来国内外主要 AI 芯片的性能、存储、带宽信息。通过比较可以发现，在固定计算精度的情况下，单张 AI 芯片性能增长速度远低于 AI 算法的增长速度，平均每年的性能增长为 70%，内存容量和带宽的增长则更为缓慢。硬件性能增长缓慢与上层算法增长过快的矛盾成为了当前 AI 算法训练和部署中的主要矛盾。

在这样的背景下，我国在 2016 年提出了《新一代人工智能发展规划》，其中明确提出在“核高基”专项中加入人工智能软硬件支持。为了实现软硬件的高度协同工作，

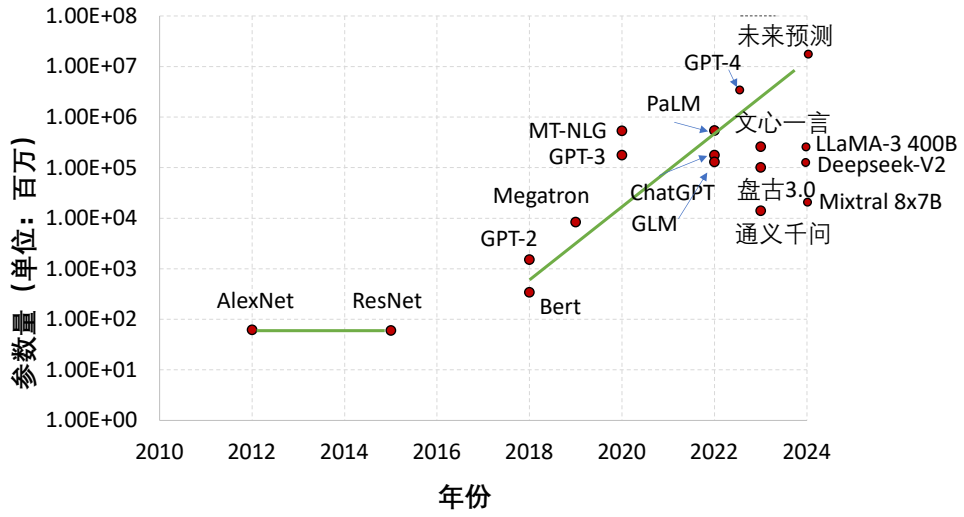


图 1.1　近十年主要 AI 算法参数量变化趋势

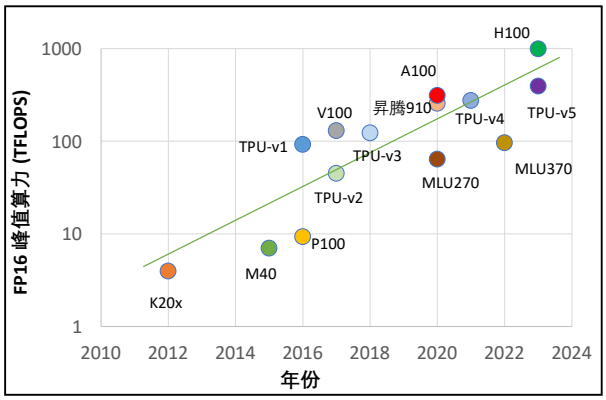
充分发挥硬件性能，满足算法的需求，就需要研制配套的高效软件栈，从高层次的算法语义逐步编译至硬件的指令，通过高效的调度和优化算法实现性能最大化和开发难度有效降低。为此，本文认为，以形式化和自动化为核心的编译技术是满足这一要求的重要技术。面向 AI 芯片的编译技术主要从三个层次开展研究：图层，算子层，指令层。图层关注 AI 算法整体的表达和优化；算子层关注 AI 算法中的算子如何进行表达和优化；指令层关注与硬件衔接的硬件指令如何生成与优化。而针对 AI 芯片实现编译优化的软件栈框架，则被称为 AI 编译器。

AI 编译技术发展多年，经历了模板调优，规则组合，分析优化三个阶段，形成了计算调度分离、多面体模型、追踪编译等多种技术路线。尽管取得了众多的成果，在实际使用时仍有许多问题尚未解决。本文总结 AI 编译技术发展至今面临的三个主要技术挑战：（1）在图层次如何协同优化计算与访存。（2）在算子层次如何针对不同硬件芯片自动生成优化空间并通过统一的搜索寻找最优的循环调度策略。（3）在指令层次如何将 AI 芯片的专用特殊指令进行抽象并通过编译器自动地、正确地、高效地将指令代码进行生成。

为了解决这些挑战，本文提出一个完整的 AI 编译器设计，其涵盖了 AI 编译的图层、算子层、指令层。在图层次，本文着重关注图融合优化，通过编译技术辅助生成高性能融合代码，并提出了框架 Chimera 和 TileFlow 完成这一优化过程；在算子层，本文关注算子循环调度，通过编译器自动化生成高性能代码，提出 FlexTensor 框架完成自动化算子编译；在指令层，本文抽象硬件指令，自动化生成 AI 芯片专用指令实现加速，提出 AMOS 框架自动完成指令生成。本文将讨论每个层次的具体抽象方法和编译技术，旨在为当前和未来的 AI 编译器设计提供借鉴和参考，帮助 AI 领域研究人员和

名称	年份	峰值算力 TFLOPS	内存 GB	内存带宽 GB/s
K20x	2012	3.95	6	250
M40	2015	7	12	288
P100	2016	9.3	16	732
TPU-v1	2016	92	8	34
V100	2017	130	16	900
TPU-v2	2017	45	16	600
TPU-v3	2018	123	32	900
昇腾910	2020	256	-	1200
MLU270	2020	64	16	102
A100	2020	312	80	1500
TPU-v4	2021	275	32	1200
MLU370	2022	96	48	614
H100	2023	989	80	3352
TPU-v5	2023	393	16	819

近十年主要AI芯片性能、存储、带宽



近十年主要AI芯片性能变化趋势

图 1.2　近十年主要 AI 芯片的（半精度）性能、存储、带宽比较

从业者更好地将高层次算法映射到底层硬件。

本章接下来的内容里，首先介绍 AI 编译器研究背景，包括 AI 算法和 AI 芯片，以及各个层次的 AI 编译的研究挑战；然后介绍本文研究动机，根据研究挑战详细阐述研究问题，展示研究内容与研究挑战的对应关系；最后介绍本文内容的组织结构。

1.1　研究背景与挑战

本章节将介绍 AI 芯片和 AI 计算各自的特点，以及 AI 编译器三个层次编译各自的主要研究挑战。

1.1.1　AI 算法简介

当前主流的 AI 算法是通过构建深度神经网络将输入的原始特征值经过复杂变换得到输出的特征值，并基于这些输出特征计算概率以进行决策。这类算法需要大量的计算步骤进行特征空间变换，每个计算步骤都可以被抽象为算子。由于输入输出的特征值往往是高维的，AI 算法使用高维度张量（类比计算机编程语言中的多维数组）数据结构表达计算中的数据。所以算子的本质是以张量为输入和输出的函数，也可以称之为张量计算。大量的算子相互衔接，以前置算子的输出为后继算子的输入，构成了数据流图结构，被称为计算图。在表1.1中列出了常用的算子的数学定义，其中包括矩阵乘法和卷积。简便起见，本文使用爱因斯坦求和约定来表示每个算子。这里使用符号$\circ$表示乘法和求和操作，右侧出现但左侧未出现的下标表示规约操作。例如，二维卷积定义 $\mathrm{O}_{b,k,i,j} = \mathrm{I}_{b,rc,i+rx,j+ry} \circ \mathrm{W}_{k,rc,rx,ry}$ 可以解释为 $\mathrm{O}_{b,k,i,j} = \sum_{rc}\sum_{rx}\sum_{ry} \mathrm{I}_{b,rc,i+rx,j+ry} \times \mathrm{W}_{k,rc,rx,ry}$。某些算子具有特殊的参数。例如，在分组卷积中，$\mathrm{P}^g, \mathrm{I}^g, \mathrm{W}^g$ 是第 g 组的张量，g 代表分

表 1.1　AI 算法算子表达式举例

算子英文名称	表达式	中文解释
GEMV	$O_i = A_{i,k} \circ B_k$	矩阵向量乘法
GEMM	$O_{i,j} = A_{i,k} \circ B_{k,j}$	矩阵乘法
Bilinear	$O_{i,j} = A_{i,k} \circ B_{j,k,l} \circ C_{i,l}$	双线性变换
1D convolution	$O_{b,k,i} = I_{b,rc,i+rx} \circ W_{k,rc,rx}$	一维卷积
Transposed 1D convolution	$O_{b,k,i} = I_{b,rc,i+rx} \circ W_{rc,k,L-rx-1}$	一维反卷积
2D convolution	$O_{b,k,i,j} = I_{b,rc,i+rx,j+ry} \circ W_{k,rc,rx,ry}$	二维卷积
Transposed 2D convolution	$O_{b,k,i,j} = I_{b,rc,i+rx,j+ry} \circ W_{rc,k,X-rx-1,Y-ry-1}$	二维反卷积
3D convolution	$O_{b,k,d,i,j} = I_{b,rc,d+rd,i+rx,j+ry} \circ W_{k,rc,rd,rx,ry}$	三维卷积
Transposed 3D convolution	$O_{b,k,d,i,j} = I_{b,rc,d+rd,i+rx,j+ry} \circ W_{rc,k,D-rd-1,X-rx-1,Y-ry-1}$	三维反卷积
Group convolution	$O^g_{b,k,i,j} = I^g_{b,rc,i+rx,j+ry} \circ W^g_{k,rc,rx,ry}$	分组卷积
Depthwise convolution	$O_{b,k,i,j} = I_{b,c,i+rx,j+ry} \circ W^c_{k,rx,ry}$	深度可分离卷积
Dilated convolution	$O_{b,k,i,j} = I_{b,rc,i+rx\times dx,j+ry\times dy} \circ W_{k,rc,rx,ry}$	空洞卷积

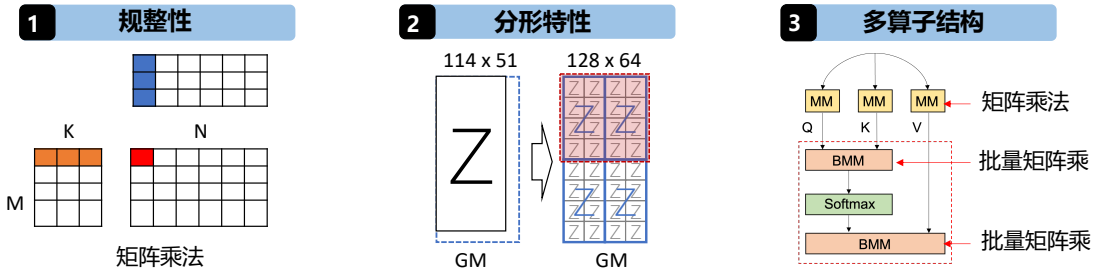


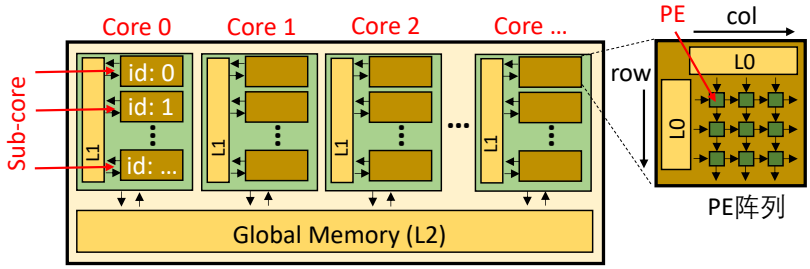
图 1.3　AI 计算共性特点

组卷积的组数；对于深度可分离卷积，输入输出有不同的通道，用 c 表示，W^c 是通道 c 的权重张量；空洞卷积使用 dx, dy 作为扩张因子，扩张因子用来扩大卷积窗口。

虽然 AI 计算种类繁多，但是仍然可以总结 AI 算法的共性特点，这些共性特点也是加速 AI 计算的关键所在。这些共性特点如下：

- **AI 计算具有规整性特点**。AI 计算往往都是在向量、矩阵、图像、高维张量等数据结构中，通过固定深度、固定长度的嵌套循环进行计算。因此无论是访存还是计算，都有很强的规整性特点。正如图 1.3中所示的矩阵乘法例子展示的那样，这些规整的计算都可以用边界清晰的张量之间的访存和计算关系表示出来。值得注意的是，尽管存在一些技术方法使得 AI 计算变得不规整，比如稀疏化技术，在实际应用中，仍然是规整的稠密计算或结构化稀疏占据主流，所以本文仅关注规整的稠密计算。
- **AI 计算具有分形特性**。所谓分形特性，是指 AI 计算往往是可以分割为同质的更小的计算，比如矩阵乘法可以分割为更小的矩阵乘法，卷积可以分割为更小的卷积。这种分形特性是 AI 计算能被加速的重要原因。因为可以通过硬件加速一小部分的 AI 计算，然后重复多次组合出完整的计算，从而使全部的计算得到加速。在图 1.3中，展示了一个矩阵乘法可以通过分形得到更多的小矩阵乘法的过程。

a) AI芯片架构举例



b) AI芯片指令举例

```
AVX-512: __m512d _mm512_add_pd (__m512d a, __m512d b)
         __m512h _mm512_loadu_ph (void const* mem_addr)
Tensor   wmma::mma_sync(c_frag, a_frag, b_frag, d_frag)
Core:    wmma::load_matrix_sync(a_frag, mem_addr, lda)
```

图 1.4　AI 芯片架构和指令举例

- **AI 计算具有多算子结构特点**。如今的 AI 算法都是需要多个计算步骤才能完成，因此优化 AI 计算需要考虑多算子结构。对于多个算子的联合优化，往往采用算子融合的方法，尽管其他技术方法如图替换、图简化等也是对多算子结构的优化技术，它们的性能收益和使用广泛度都不如融合优化，因此本文主要讨论融合优化。在图 1.3中也展示了一个多算子结构的 AI 计算例子，这个例子是 Transformer 中的自注意力层（self-attention），后文会多次用到。

1.1.2　AI 芯片简介

AI 芯片泛指对 AI 算法进行定制化加速的硬件，从低端的微控制器，到个人使用的手机电脑，再到云端使用的服务器级别 GPU，TPU，NPU 等，都可以纳入 AI 芯片的范畴。AI 芯片虽然架构种类众多，也可以总结一些共性特点，这些共性特点的存在，是受 AI 计算共性特点影响，也为本文提出统一的 AI 编译技术处理不同 AI 芯片上的代码优化和生成奠定了基础。这些共性特点如下：

- **AI 芯片往往都采用空间架构进行加速**。 AI 芯片主要使用的硬件架构是空间架构（Spatial Architecture）。空间架构加速器依靠处理单元（Processing Engine, PE）阵列实现对 AI 算子的加速。配合 PE 阵列，AI 芯片还会使用多层次的片上内存（Buffer）提高数据局部性进一步加速 AI 计算。在图 1.4中展示了 AI 芯片架构的示意图。在图中 a) 部分，这一示意图展示了一个多核架构的加速器，众多核心共享全局内存（Global Memory），每个核内部还有子核（Sub-core），子核共享局部内存（L1 Buffer），每个子核都包含一个 PE 阵列，用来加速一个小规模的

张量计算，如矩阵乘法或者向量运算。当前市场中投入使用的 AI 芯片如英伟达 Tensor Core GPU，华为 Ascend NPU，虽然具体的架构各有不同，其总体都符合这种多层次结构。

- **AI 芯片的访存操作复杂**。不同于通用芯片的使用，AI 芯片具有更多层次的存储和数据搬移路径。在 AI 芯片上完成数据搬移和计算的过程，也被称为数据流设计。数据流原本指芯片硬件设计时，如何设计 PE 阵列的排布和连接，通过控制数据在 PE 间传递和计算实现不同效果的加速，比如对于矩阵乘法加速的 PE 阵列，常见的三种数据流为输入静止（Input Stationary），权重静止（Weight Stationary），以及输出静止（Output Stationary）。但是对于 AI 芯片上层软件编程，数据流的含义则是指如何将 AI 算法的计算和数据映射到 AI 芯片的片上内存和计算核心上。例如，对于多个连续算子构成的计算图，可以使用串行映射一个一个执行算子，也可以使用流水线，通过双缓冲实现算子执行的流水线并行。本文讨论面向 AI 芯片的编译技术，主要关注第二种数据流的含义，追求自动寻找高效的数据流将 AI 计算映射到 AI 芯片实现加速。
- **AI 芯片需要专用指令集编程**。 AI 芯片需要通过软件编程来进行使用，为此 AI 芯片往往都配套有对应的指令集。其中，计算指令和访存指令最为重要。计算指令通过一条简短的指令可以控制 AI 芯片完成较为复杂的运算，如 Tensor Core GPU 的 WMMA 指令可以进行一次 $16\times16\times16$ 的矩阵乘法。通过这种方式，AI 芯片可以降低指令取址和译码的开销，提高整体的执行效率。访存指令则通过一条指令实现成片的数据搬移，如 AVX-512 指令集中的 *_m512h_mm512_loadu_ph* 指令可以一次加载 512 比特的数据，同样降低了取址和译码开销。在图 1.4的 b) 部分例举了几个 AI 芯片的计算指令和访存指令。

1.1.3 各层次编译研究挑战

本文总结了三个研究挑战，分别对应图、算子、指令层次。这些研究挑战列在了图 1.6中，下文将对各个挑战详细阐述，

图层 AI 编译挑战：融合中计算与访存优化困难。在图层面，面向 AI 芯片的编译技术关注 AI 算法的计算图在 AI 芯片上的调度，主要的优化手段为算子融合，将原本要分成多步执行的计算图合并为一个单一的计算函数（Kernel），在这一个 Kernel 内部完成所需的所有计算步骤和访存行为。融合带来性能提升的主要原因是减少了片外数据访存，中间结果不需要额外存出或读入。因此，只有当算子性能受限于访存带宽时，融合优化的收益才是明显的。在 AI 芯片兴起之前，AI 算法主要在通用硬件如 CPU 和 GPGPU 上进行加速，在这类硬件上，一个 AI 算子性能是否受限于带宽可以由算子类

型决定，如绝大多数矩阵乘法在 CPU 上都属于计算受限而不是访存受限，而一些激活函数如 ReLU[14]则总是受限于带宽的。所以 AI 算法中的算子被分为了两类：计算密集型和访存密集型。长久以来，计算密集型算子都被认为是计算受限而非访存受限。从 AI 芯片出现以后，如 Tensor Core GPU，TPU 等加速器将硬件计算峰值性能提高了一到两个数量级，但是带宽的增长速度远不如峰值性能的增长速度，这就导致许多计算密集型算子部署在 AI 芯片时，出现了性能受限于带宽的问题。在图 1.5部分 a) 中展示了批量矩阵乘法和卷积测试的结果，所选的批量矩阵乘法（Batch GEMM, BMM）和卷积（Convolution, Conv）都是典型的计算密集型算子，这里使用的输入形状是来自于真实的网络 Transformer[5]和 Yolo[15]的，设置批处理大小（Batch）为 1。在 A100 GPU 上使用手工优化的库函数 CuBlas 和 CuDNN 执行这些算子，并将这些性能结果绘制在屋顶模型图[16]（Roofline Model）中，可以看到，这些算子的执行性能都是远低于理论所能达到的最高性能，并且处于 Roofline Model 的斜线区域，代表着它们的计算密度处于访存带宽受限的位置。

这些计算密集型算子在真实网络中往往成链式结构出现，在图 1.5部分 b)，展示了 Transformer 的自注意力（self-attention）结构[5]，其中有两个批量矩阵乘法；部分 c) 中，展示卷积神经网络（Convolution Neural Network, CNN）中的卷积算子，可以发现它们也成链式出现。如果这些算子性能受限于访存带宽，这就带来了新的优化机会，将这些带宽受限的计算密集型算子进行融合，就可以进一步提高计算密度，从而提高性能。但是先前图层次编译工作主要考虑访存密集型算子融合，如图 1.5部分 d) 中所示，对于访存密集型算子与计算密集型算子之间的融合，有头部（Prologue）融合技术和尾部（Epilogue）融合技术支持[17-19]，对于访存密集型算子之间的融合，可以使用函数内联（inline）或者函数粘合（Kernel Stitching）的方式完成融合[20]。但是对于计算密集型算子的融合仍有研究空缺。

计算密集型算子融合的主要难点在于计算密集型算子中的数据访存行为较为复杂，其中存在的大量规约计算（如累加）会导致数据的冗余读取，从而导致融合后性能仍然可能受限于访存带宽。因此，如何将访存与计算密集型算子的计算协同优化，从而最小化冗余数据搬运，提高融合性能，仍然十分困难，本文将这一问题列为图层次研究的主要挑战。

算子层 AI 编译挑战：循环优化空间的生成和搜索困难。在算子层面，面向 AI 芯片的编译技术关注 AI 芯片的片上资源使用和调度。由于 AI 芯片片上计算和存储资源有限，而 AI 算法的计算量和数据量往往较大，难以全部映射到 AI 芯片上，这就需要将 AI 算法进行分割，每次映射一部分计算和数据到 AI 芯片上，这种不同的变换在编程上实现就对应着循环结构的变化，典型的循环变换包括循环分块、展开、重排等等。在表 1.2里

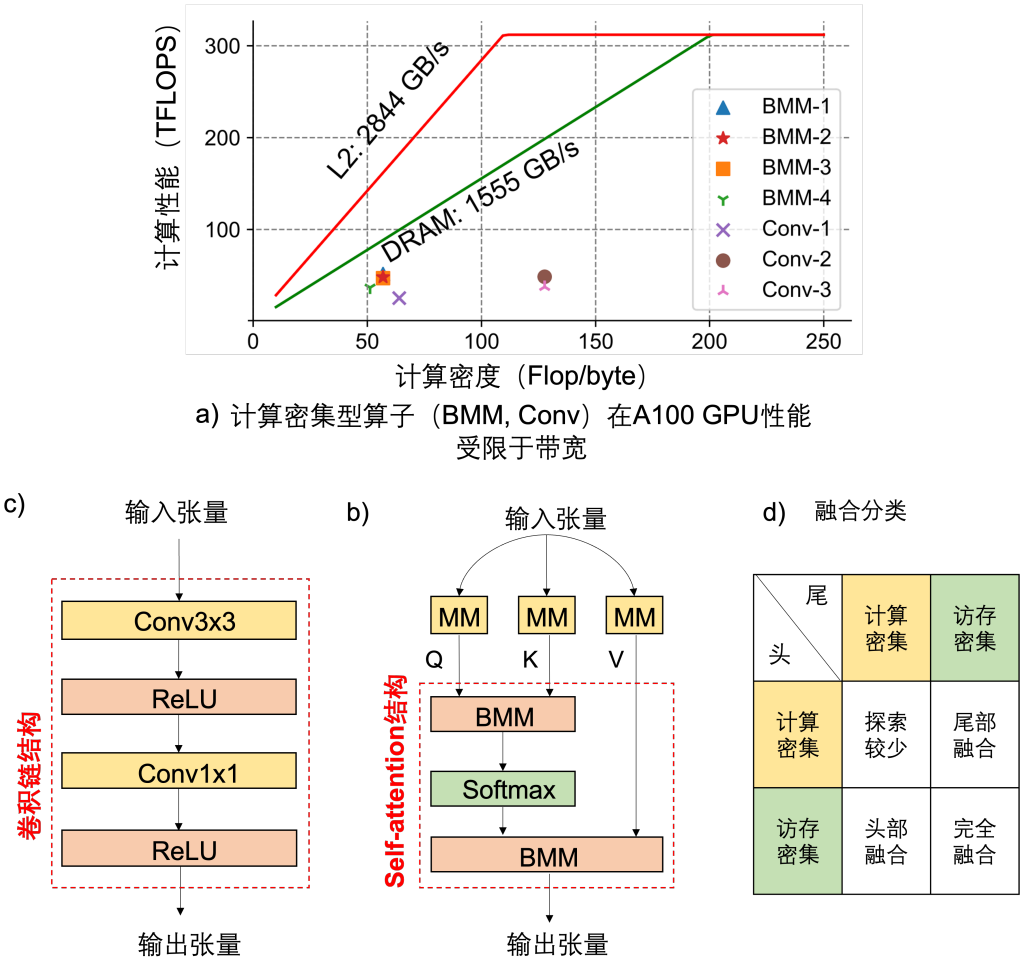


图 1.5　图层编译背景介绍：a) 计算密集型算子出现性能受限于带宽的问题。b）和 c）计算密集型算子成链式出现在 AI 算法中。d）融合优化的分类

展示了算子优化时常用的循环变换技术。同一个 AI 算子映射到 AI 芯片有多种循环结构变换的选择，这就导致人工开发需要遍历设计空间，开发效率极低。而算子层编译技术可以通过自动在设计空间中寻找高效的循环变换策略并自动生成变换后的代码解决这一问题，从而提高 AI 芯片上层软件栈的开发和迭代效率。其核心技术是通过程序分析自动枚举优化组合的设计空间，然后设计高效搜索算法在设计空间中寻找最优的优化组合，并根据这一组合最终生成代码。

但是自动化算子代码生成的主要挑战是循环优化空间过于复杂和巨大，难以通过仅枚举优化参数的方式生成完整的优化空间，需要提出能枚举组合各种优化技术（分块、重排等）生成更大的空间，才能覆盖性能更优的代码生成策略。而且在巨大的空间中如何进行有效搜索，也成为了挑战，常规的枚举和启发式算法不再有效，如何将

更先进的机器学习技术引入到搜索中，也成为值得研究的问题。

表 1.2 算子优化中常用的循环变换技术

No.	优化英文代称	优化功能中文描述
1	Tiling	循环分块，将多层循环切分后交错重排
2	Fusion	将两个紧邻的循环合并为一个循环
3	Inline	将两个完美循环体合并为一个完美循环
4	Interchange	改变循环顺序，进行重排
5	Parallelization	多线程并行循环
6	Unrolling	将循环体进行展开
7	Sectioning	将循环体的一部分展开后使用 SIMD 指令替换
8	Vectorization	使用 SIMD 指令进行计算或数据搬移
9	Tensorization	使用特殊的矩阵、张量指令完成计算或者数据搬移
10	Skewing	循环扭斜，保持迭代依赖情况下暴露并行性
11	STT	时间空间变换，控制映射到执行单元的位置和时间
12	Polyhedral	使用通用的多面体模型进行变换
13	Cache Operation	创建数据临时读写操作，对应片上内存使用
14	Software Pipeline	软件多级缓存构造流水线隐藏延迟
15	N-buffering	显式构建多级缓冲实现多级流水线
16	Register/Buffer Allocation	特殊存储空间开辟，如共享内存等
17	Parallel Reduction	使用多线程并行规约优化
18	Hardware Pipelining	构建硬件流水线，用于 FPGA 等可编程硬件
19	Memory Partition	存储分割，用于控制可编程硬件的内存使用目的
20	Quantization	量化优化，用于自动更改程序中数据精度
21	Relayout	改变算子输入输出数据的内存排布结构

指令层 AI 编译挑战：硬件指令缺少计算与访存抽象。在指令层面，面向 AI 芯片的编译技术关注 AI 芯片计算和访存指令的自动生成。由于 AI 芯片往往通过一个指令完成较为复杂的计算和数据搬移操作，编程人员需要对底层硬件行为有深刻的理解才能正确并高效地使用这些指令。这就导致面向 AI 芯片的编程开发门槛大幅提高，其软件生态建设效率远低于通用硬件。为此，指令层编译可以通过提高 AI 芯片的编程抽象层次，允许编程人员使用通用的循环控制结构编写 AI 算法，然后由编译器在循环程序中寻找使用特殊指令的机会，并自动重写程序以实现计算和访存向 AI 芯片专用硬件的映射。在表 1.3中，列举近年来主要的指令层编译器工作。他们可以分为两类：

1) 硬件感知型：编译器可以知道详细的 PE 连接和片上存储配置，因此他们可以通过特定软硬件接口将 AI 计算映射到 PE 和内存[32-35]，得到的数据流往往是固定的（例如对 FPGA 进行烧录）。这种方法主要应用于特定领域的硬件和软件协同设计。这些编译器的主要目标是在某些硬件约束下优化硬件资源利用率（提高并行性和数据重用）。他们通常通过求解一个等价的线性规划问题来解决映射问题，以避免在巨大的调度空间中进行搜索[21-23]。但是在大部分 AI 芯片上，这种硬件细节往往难以获知，而且固定的数据流往往难以适应动态的应用需求，因此硬件感知型编译器的实用范围仍然受限。

表 1.3　针对 AI 芯片的指令层代表性编译器

名称	方法	技术方法	硬件支持
Nowatzki et al.[21]	硬件感知型	线性规划	CGRA
CoSA[22]	硬件感知型	混合整数线性规划	空间加速器
SARA[23]	硬件感知型	混合整数线性规划	RDA
HASCO[24]	硬件感知型	黑盒调优	空间加速器
AutoTVM[25]	指令感知型	手写模板	CPU, GPU
Ansor [26]	指令感知型	生成规则	CPU, GPU
UNIT[27]	指令感知型	手写模板	CPU, GPU
XLA[28]	指令感知型	手写模板	CPU, GPU
ISA Mapper[29]	指令感知型	手写模板	CPU
Tiramisu[30]	指令感知型	多面体模型	CPU, GPU
AKG[31]	指令感知型	多面体模型	CPU, GPU, NPU

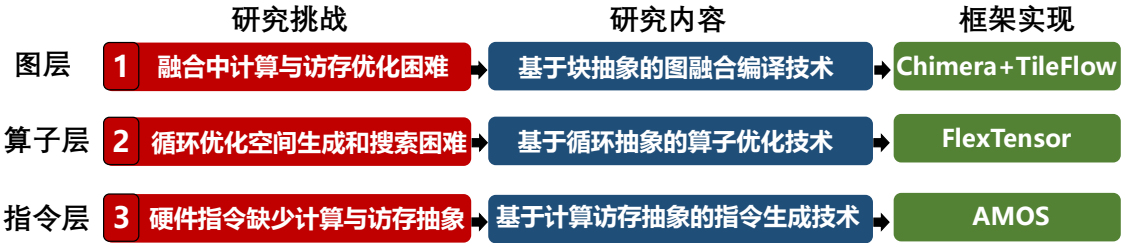


图 1.6　本文提出的 AI 编译的研究挑战、研究内容、框架实现对应关系

2) 指令语义可感知映射：当硬件细节不对上层暴露时，硬件往往都有配套的指令集用来编程，编译器可以利用指令语义作为映射指导实现软硬件映射，例如通过固定的代码生成模板[25,27]生成指令并在给定的优化调度空间内调整性能。为了支持新的 AI 算子，就需要在编译器中手动开发新的模板，这一部分需要大量的专家知识，成为了开发的瓶颈。如何避免手动增加模板，降低编译器开发和维护门槛，从而更高效对接不同 AI 芯片指令集，是亟待研究解决的问题。

自动化生成 AI 芯片特殊指令的主要挑战是寻找合适的方法形式化表达 AI 芯片指令的计算与访存语义，并设计对应的指令映射算法完成指令正确、高效的生成。当前编译器仍然缺少相关指令抽象，导致自动化生成指令困难。因此，指令层编译的挑战主要是如何抽象计算与访存抽象。

1.2　研究动机

基于上述的研究挑战，总结本文的研究动机如下。

已有工作缺少合适的图抽象完成融合中的计算访存优化。在图层，先前工作着重讨论如何进行算子融合优化。根据抽象的粒度不同，优化方法也有所不同。以 XLA[28]，DNNFusion[19]，TASO[18]，TensorRT[36]等为代表的编译器中，融合的抽象粒度是算子，多个算子融合后得到新的融合算子，融合算子进行代码生成时，则需要有对应的后端

实现，通常都会实现为算子库。如果出现了后端算子库不支持的融合算子，则会融合失败。这种方法灵活，但是开发代价高，周期长。另一方面，以 TVM[17]，Relay[37]，AKG[31]等为代表的编译器中，融合的抽象粒度是循环，编译器可以完全控制来自不同算子的循环如何合并到同一个函数（Kernel）中。尽管这种细粒度融合可以避免手动开发算子库的难题，它们的效率往往很低，生成的代码性能与手写的算子库仍有差距。其原因是过细的粒度导致融合的优化空间过大，编译器无法进行彻底地遍历，只能在有限的时间内依照启发式选取融合方案并生成代码，错失了最优的代码生成方案。本文将寻求更适合图上算子融合的抽象粒度，并设计对应的编译优化方法进行代码生成。

已有工作对算子循环优化空间生成和搜索仍不充分。在算子层，尽管先前工作对于算子内的循环表示和循环变换提供了良好的抽象，但是如何组合这些变换以得到高性能循环代码仍然是未解决的难点问题。其根本原因在于缺少自动组合这些变换的方法，只能依赖开发人员自行手动组合不同的变换方式，并写到模板中。而且开发出的模板只适用于少数算子和少数硬件后端，对于大量出现的新兴算子和新兴硬件支持困难，导致开发工作出现组合爆炸问题，大大降低了 AI 芯片软件栈开发效率。先前工作 AutoTVM[25]和 Halide[38-40]都是依赖于这种模板开发的方式进行代码生成，然后对于这些循环变换中使用的超参数进行调优。另一方面，多面体模型相关工作如 Tiramisu[30]和 AKG[31]不使用循环变换模板进行优化，而是依赖对循环迭代空间进行仿射变换实现程序的循环结构改变。这类工作适用于循环体依赖关系符合仿射约束的算子，对于更为复杂的算子循环体结构或者计算规模较大的算子，多面体模型往往难以处理或者需要较长的求解时间才能完成优化。因此，已有工作或依赖手工辅助生成优化空间，或选择将空间进行限制，无法自动化生成完整的优化空间；在生成的空间中，也难以实现有效的搜索，缺少系统性的算法设计和指导。本文将针对这一问题，设计空间生成和搜索技术，进一步提高算子层编译生成代码的性能。

已有工作缺少对指令的计算、访存抽象和指令生成算法。在指令层，已有编译技术对于 AI 芯片的特殊指令支持有限，在缺少开发人员的人为标注的情况下，编译器无法自动识别使用特殊指令的机会，也无法保证使用特殊指令的正确性。其根本原因是 AI 芯片的特殊指令行为复杂度远超过通用标量指令，而当前编译器内部没有对于这类特殊指令的形式化表达与分析的支持，从而无法完成软件计算到硬件的自动化映射。在表 1.3中展示了七个指令感知型编译器，他们主要依赖三种方法进行特殊指令生成：手写模板，规则生成，以及多面体模型（Polyhedral Model）。手写模板和规则生成需要将指令生成的方式直接编码在编译器中，其开发门槛高，且无法完整探索完整的优化空间，因此只能支持少数的 AI 算子和少数的后端设备。多面体模型本身并不支持特殊指令生成，而是能做到保持依赖的情况下进行语句调度优化，Tiramisu[30]和 AKG[31]为了

能支持特殊指令生成，手工开发了一些微内核（Micro kernel），微内核内部使用了特殊指令，这种方法灵活度非常受限。这些工作自动化能力受限的根本原因是没有对指令进行形式化抽象，也没有相应的自动化指令生成和正确性验证技术。因此，本文将寻找适用于 AI 芯片特殊指令的编译器抽象来做到更通用的自动化代码生成和验证。

1.3 本文主要贡献和内容组织

1.3.1 主要贡献及其相关性

本文上述内容阐述了当前 AI 编译技术面临的三个重要挑战，以及本文开展研究的动机，指明了当前编译研究的重点应在于对不同的层次选取合适的抽象，并设计配套的编译优化方案。通过组合三个层次的编译内容，可以得到完整的 AI 编译框架，对上对接 AI 算法，对下对接多种 AI 芯片。高度总结本文的技术内容贡献为以下三点，这些技术内容贡献与研究挑战的对应关系在图 1.6中展示：

- 在图层，本文研究内容是基于块抽象的图融合编译技术，并提出了 Chimera 自动算子融合框架和 TileFlow 融合性能评估模型。框架中实现了一种分析性方法来评估融合后的算子在不同架构配置下的数据搬移量。根据这一方法，可以进一步提出最优化方法求解对应的分块参数、循环顺序等关键信息从而指导代码生成。本文提出的性能模型经过与真实硬件的比对和校验，达到了优秀的准确率，并成功指导编译器自动寻找高效的融合方案，在真实硬件上得到超过先前工作性能的更优融合数据流。对 CPU、GPU 和 NPU 上的批量矩阵乘法链和卷积链的实验表明，Chimera 与手动优化的算子库相比可实现高达 2.87 倍、2.29 倍和 2.39 倍的加速。与最先进的编译器相比，在 CPU、GPU 和 NPU 的加速比分别高达 2.29 倍、1.64 倍和 1.14 倍。
- 在算子层，本文的研究内容是基于循环抽象的算子优化技术，提出了 FlexTensor 框架，可以自动优化 CPU、GPU 和 FPGA 上的张量计算，无需人工干预可以进行全自动优化。在 FlexTensor 前端可以进行静态分析，分析不同的张量计算特点并形成不同的优化空间；在 FlexTensor 后端，本文还提出启发式和机器学习相结合的方法来探索优化空间并自动针对不同硬件生成代码的技术。实验中，在 CPU、GPU 和 FPGA 上使用不同的 AI 算子来评估 FlexTensor，与 CuDNN 相比，FlexTensor 在 Nvidia V100 GPU 上实现平均 1.83 倍的性能速度；与用于 MKL-DNN 相比，在 Intel Xeon CPU 上对卷积的性能速度为 1.72 倍；与用手工实现的 OpenCL 代码相比，在 Xilinx VU9P FPGA 上针对卷积的性能速度为 1.5 倍。
- 在指令层，本文的研究内容是基于计算访存抽象的指令生成技术。本文设计了

AMOS 框架进行面向 AI 芯片的特殊指令生成。在 AMOS 中，提出了硬件指令抽象来正式定义 AI 芯片的计算和访存行为，以及从软件到硬件的映射。同时，本文还设计了两步映射生成流程、简洁的指令验证算法，以及用于搜索空间性能评估的性能模型。实验表明，与手工优化的库相比，AMOS 在 Tensor Core GPU 上可以实现超过 2.50 倍的速度，在 CPU 上使用 AVX-512 指令可以相比 TVM 中的手工设计模板实现 1.37 倍的加速，在 Mali GPU 上利用点乘指令可以相比 AutoTVM 达到 25.04 倍的加速。

1.3.2　本文内容及组织结构

本文采用自顶向下顺序呈现对于 AI 编译软件栈全栈分析研究的工作，从图层到算子层，再到指令层。在每个层次详细解释对应的抽象与编译算法。全文共分为六个章节。第一章为引言，主要概述 AI 编译的研究内容，并简要介绍背景知识，包括 AI 算法和 AI 芯片的特点，指令层、算子层、图层编译的研究内容。在本章的结尾，提出了本文的研究动机，并总结全文的主要贡献和组织结构。第二章为相关工作介绍。主要从图、算子、指令三个层面介绍近年来面向 AI 芯片的编译技术发展的技术脉络和主要问题。第三章到第五章为本文核心内容，分别介绍图，算子层，和指令层三个层次的特异化抽象方法、具体研究技术、和实验结果。第六章为总结与展望。

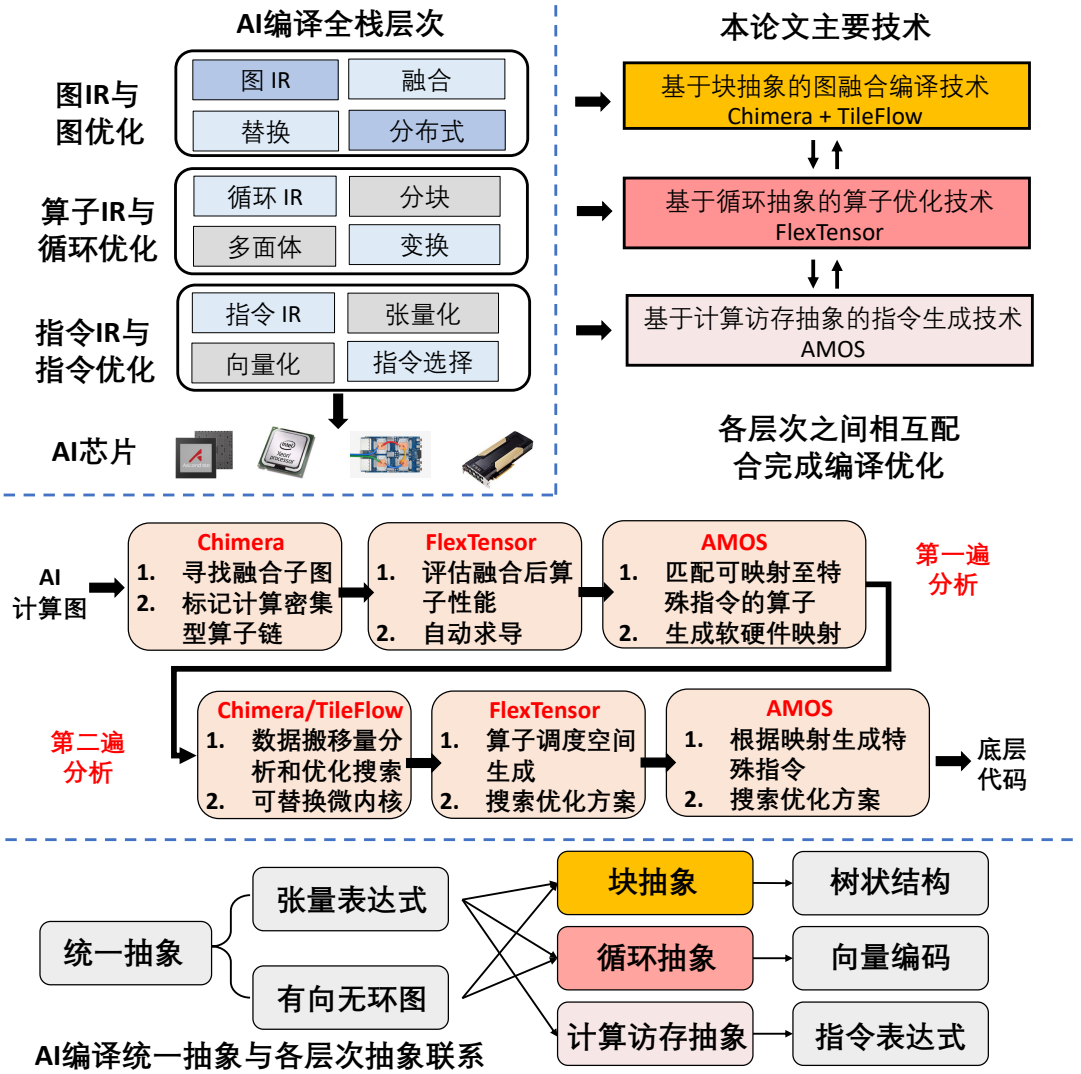


图 1.7　本文提出的 AI 编译技术全栈结构图解

第二章　研究背景与相关工作

本章节介绍近十年来主要的 AI 编译相关工作，由于 AI 编译涉及软硬件交互和映射，本文也需要简单涉及近年来 AI 芯片硬件架构的相关工作。高度概括近十年的 AI 编译技术发展，从横向切分，可以从三个层次介绍：指令层，算子层，图层；从纵向切分，可以分三个阶段：模板调优阶段，规则组合阶段，以及分析式优化阶段。贯穿其中的主要有三种编译技术路线：计算调度分离路线，多面体模型路线，以及基于追踪方法的编译路线。为了更好地展示相关工作的逻辑关系，本文在图 2.1中展示了三个层次以及三个阶段的逻辑关系，在表 2.1中总览了近十年相关工作所属层次、阶段、和技术路线。值得一提的是，尽管本文从三个技术路线介绍编译器工作，大部分编译器都会同时使用多种技术路线，如 MLIR[41]同时使用了计算调度分离，多面体模型，以及追踪编译技术。由于编译层次、发展阶段、技术路线构成了三个维度，下面本文将首先解释三种技术路线的区别；随后，从编译层次的角度去分类介绍每个层次内不同技术路线的编译器工作。最后，本文将简要介绍编译技术三个不同的发展阶段。这些介绍的内容都在表 2.1中可以找到参考。

2.1　三种 AI 编译技术路线

尽管 AI 编译的技术路线众多，为了方便总结，本文将其大致分为三条路线：计算调度分离，多面体模型，以及追踪编译。

计算调度分离路线的开端是 Halide[38]。早期 Halide 为了对不同后端硬件（特别是数字信号处理器，DSP）部署图形处理流水线，设计了优化无关的计算描述语言和配套的优化原语，从而形成了一套完整的计算调度编程系统。使用者在描述计算时，只需要使用简单的数据结构描述贴合数学定义的循环结构和访存下标，而复杂的优化（如循环分块，重排，算子融合）都可以通过简短的调度原语组合完成。计算调度分离带来最直接的优势是对于程序优化过程的形式化抽象，在计算描述上不断增加调度原语以逐步提升生成代码的性能的过程可以被自动化算法完成，为将来的自动调度、自动优化技术奠定了基础。这一技术路线的其他代表工作有 TVM[17]，Tiramisu[30]等。

多面体模型路线对于程序的抽象是由语句实例构成的集合，所谓语句实例是原程序中的语句在具体的迭代空间的实例化。迭代空间本身由程序中的循环结构决定，在多面体模型中，假设迭代空间边界是凸多面体，可以用仿射变换进行表示，这也是多面体模型名字的由来。语句实例由于对于张量的访存而存在相互之间的数据依赖，多面体模型要求数据依赖也需要满足仿射关系。在这样的限制条件下，可以给每个语句

实例赋予一个时间戳（可以是多维的，按字典序排序），从而得到所有语句实例的一种调度顺序，调度也可以被表达为仿射变换。一个正确的调度需要满足不破坏数据依赖，这一条件同样可以通过仿射变换联立不等式来描述，所有满足不等式的调度都是可行解。最后，多面体模型通过定制损失函数（cost function）的方式指导调度器在可行解空间内选择最优解，这一问题通常可以用整数线性规划算法进行求解，常用的工具库为 ISL[42]。多面体模型方法的代表工作是 Pluto[43]，尽管这一工作最开始并未使用在 AI 算法优化上，其后续改进工作 Tensor Comprehensions[44]，Tiramisu[30]和 AKG[31]都成功将多面体工作引入到 AI 编译中。

追踪编译路线指通过设计领域专用语言（Domain Specific Language,DSL）来确定性地指导编译器进行代码生成，编译器经过对 DSL 语法树的多次确定性变换和优化而生成代码，本质是一种将高层次语义确定性翻译成低层次语义的技术，前端 DSL 的每一处细节都可以确定性地翻译到后端的代码，因而被称为追踪编译。这一技术与经典编译技术最为贴近，在通用编译领域的代表工作是 LLVM[45]。在 AI 编译领域，MLIR[41]继承了 LLVM 的开发技术，引入了更多的特性（如方言，Dialect）以支持对多种芯片、多种算法的表达和变换。由于 AI 算法广泛使用 Python 编程语言进行开发，许多编译器也以 Python 语法作为前端基础语言，嵌入定制的 DSL，通过追踪编译技术生成加速器底层代码。如 TensorIR[46]，Triton[47]，TorchScript[48]等工作都是属于这一技术路线。

2.2 AI 编译三层次相关工作

2.2.1 图层相关工作

图层编译工作主要关注 AI 算法的计算图结构的优化，这类优化的范围很广，从通用优化如公共表达式化简、常量传播、死代码消除，到专用的算子融合、子图切分、子图替换。在众多的优化中，算子融合优化受到的关注最多，相关研究成果也最为丰富。算子融合是指将多个分立的算子合并在一个函数（Kernel）中实现，从而减少算子执行时中间结果的存出和读入，减轻对片外内存的访存压力，提高整体计算密度，从而提高性能。本文在图 2.1中展示图层工作的功能，在 AI 部署过程中，图层需要兼顾模型描述，优化，运行时调度。而图编译器的主要功能就是自动化计算图优化过程，通过图层中间表达方式抽象图结构和其中的计算，结合图变换、融合、切分的方式优化计算图。图层编译需要算子层的支持，对于依赖算子库的图层编译，主要借助 CuDNN[84]，CuBlas[85]，oneDNN[86]等高性能库函数支持融合后的计算图中的算子计算；对于依赖算子编译器的图层编译，则可以直接进行代码生成，完成端到端的编译和优化。

在以卷积神经网络（CNN）为主导的 AI 算法中，算子融合主要关注计算密集型算子和访存密集型算子的融合，如卷积算子和其后续的归一化算子（Normalization）、池化

表 2.1　近十年具有代表性的 AI 编译器工作

编译器	年份	技术层次	技术阶段	技术路线
Halide[38]	2013	图层，算子层，指令层	模板调优，规则组合	计算调度分离
Latte[49]	2016	图层	模板调优	追踪编译
DeepBurning[50]	2016	图层	模板调优	追踪编译
AutoCodeGen[51]	2016	图层	模板调优	追踪编译
Haddoc2[52]	2017	图层	模板调优	追踪编译
TACO[53]	2017	算子层	规则组合	计算调度分离
XLA[28]	2017	图层，算子层	模板调优	追踪编译
DLVM[54]	2018	算子层	模板调优	追踪编译
Diesel[55]	2018	算子层	模板调优	追踪编译
Relay[37]	2018	图层	模板调优	追踪编译
TVM	2018	图层，算子层，指令层	模板调优，规则组合	计算调度分离
JAX[56]	2018	图层，算子层	模板调优	追踪编译
TorchScript[48]	2019	图层，算子层	规则组合	追踪编译
TC[44]	2018	算子层	分析优化	多面体模型
nGraph[57]	2018	图层	模板调优	追踪编译
Glow[58]	2018	图层	模板调优	追踪编译
HeteroCL[59]	2019	算子层	模板调优	计算调度分离
TASO[18]	2019	图层	规则组合	追踪编译
Triton[47]	2019	算子层	分析优化	追踪编译
NeoCPU[60]	2019	图层，算子层	模板调优	计算调度分离
Tiramisu[30]	2019	图层，算子层	模板调优	计算调度分离，多面体模型
T2S[61-62]	2019	算子层	模板调优	计算调度分离
ISA Mapper[29]	2019	指令层	模板调优	追踪编译
Hummingbird[63]	2020	算子层	规则组合	追踪编译
Rammer[64]	2020	图层	规则组合	追踪编译
AKG[65]	2020	图层，算子层	分析优化	多面体模型
Ansor[26]	2020	算子层	规则组合	计算调度分离
FusionStitching[20]	2020	图层	规则组合	追踪编译
Nimble[66]	2020	图层	分析优化	追踪编译
Cortex[67]	2020	算子层	模板调优	计算调度分离
Jittor[68]	2020	图层	规则组合	追踪编译
DNNFusion[19]	2020	图层	模板调优，规则组合	追踪编译
UNIT[27]	2021	指令层	模板调优	计算调度分离
MLIR[41]	2021	指令层，算子层，图层	模板调优，规则组合，分析优化	计算调度分离，多面体模型，追踪编译
DISC[69]	2021	算子层	规则组合	追踪编译
PET[70]	2021	图层	规则组合	追踪编译
VeGen[71]	2021	指令层	分析优化	追踪编译
EXO[72]	2022	算子层	规则组合	追踪编译
Roller[73]	2022	图层	规则组合	追踪编译
AStitch[74]	2022	图层	规则组合	追踪编译
BOLT[75]	2022	图层	规则组合	追踪编译
MetaSchedule[76]	2022	算子层	规则组合	计算调度分离
DietCode[77]	2022	算子层	模板调优	计算调度分离
Alpa[78]	2022	图层	分析优化	追踪编译
DISTAL[79]	2022	图层	规则组合	计算调度分离
FreeTensor[80]	2022	算子层	规则组合	计算调度分离, 追踪编译
TensorIR[46]	2023	指令层	规则组合	计算调度分离, 追踪编译
SparseTIR[81]	2023	算子层	规则组合	计算调度分离，追踪编译
Graphene[82]	2023	指令层	规则组合	追踪编译
Heron[83]	2023	算子层	规则组合	计算调度分离

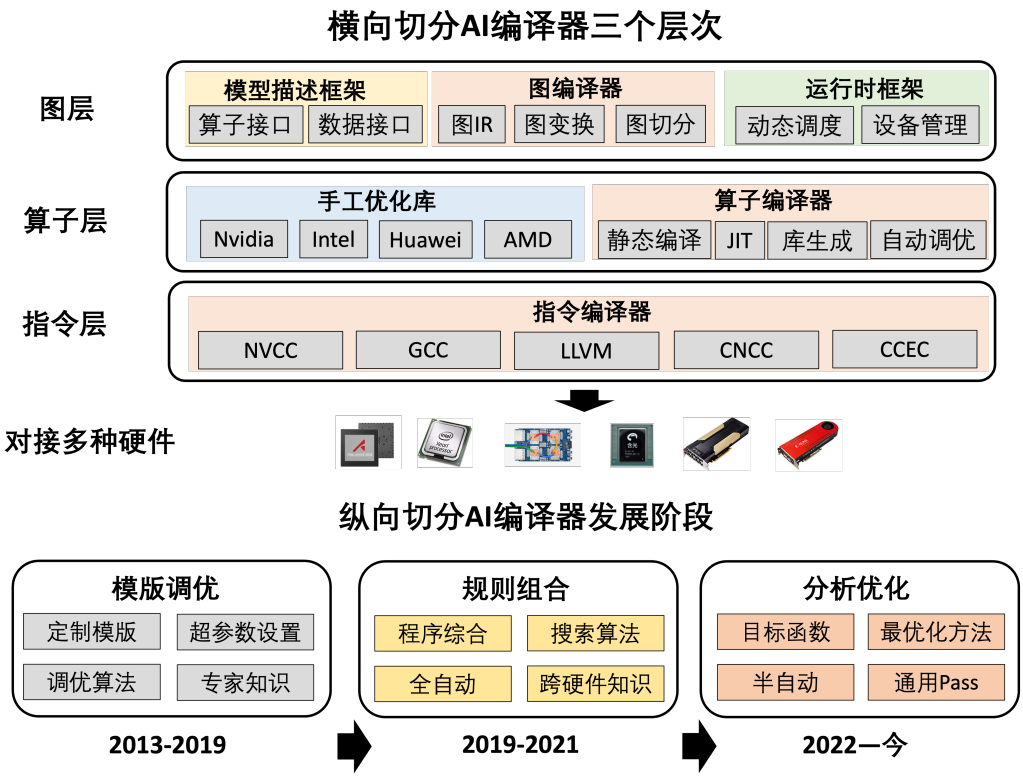


图 2.1　AI 编译技术的三个层次和三个发展阶段

算子（Pooling），以及激活函数（Activation）融合。许多编译器工作都主要支持这类融合优化。如计算调度分离路线中举例有 Halide[38]，TVM[17]，AutoTVM[25]，Ansor[26]等等；在多面体模型路线中，Tiramisu[30]，AKG[31]等也是如此；在追踪编译路线中，XLA[28]，Relay[37]，JAX[56]，TASO[18]，Rammer[64]等等也是支持这类融合。

近几年，相关研究又将重点转移到访存密集型算子之间的融合，如规约算子（Reduction）和逐点算子（Elementwise）的融合。这类工作的的代表包括 DNNFusion[19]和 AStitch[74]。DNNFusion 通过大量定制模板的方式支持各类融合后的函数代码生成，同时配合多种融合规则在计算图内系统性地搜索融合方案。AStitch 通过一种特殊的函数粘合（Kernel Stitching）的技术支持各种访存密集型算子的融合，并且取得了非常明显的性能收益。

然而随着生成式 AI 技术的发展，当前主流的 AI 算法中，以 Transformer[5,87-88]为基础的模型成为主流，尽管先前对于计算密集型和访存密集型算子相互融合的技术仍然重要，它们无法完全满足 Transformer 计算的性能需求。正如图 1.5部分 a）和 b）所示，Transformer 内部的自注意力层（self-attention）中性能瓶颈在于两个连续的批量矩阵乘法，这两个算子之间的中间数据规模随输入数据的长度成平方级增长，导致大量的开销都在数据搬移上，因此这两个计算密集型算子逐个执行的时候，呈现了访存受

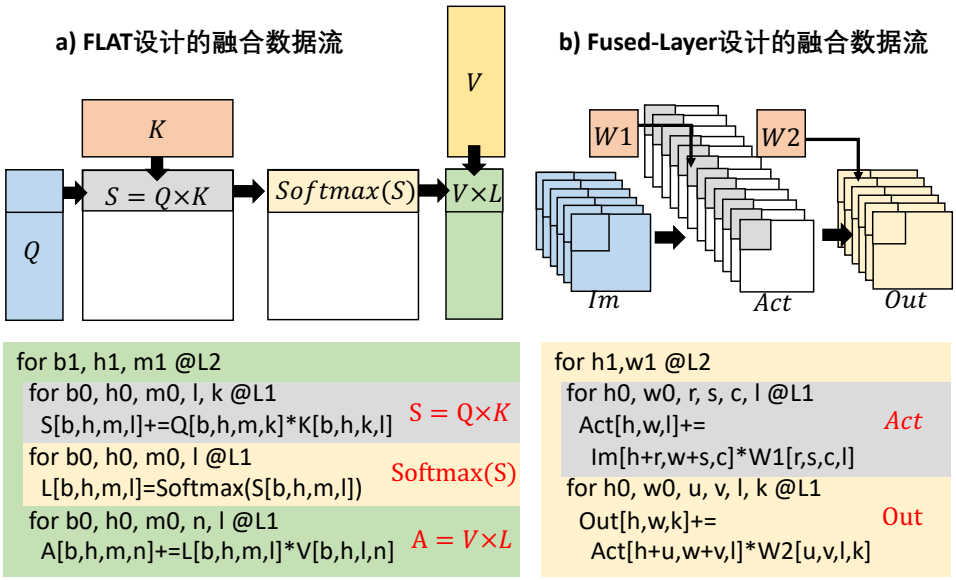


图 2.2　FLAT[89] 和 Fused-Layer[90]设计的融合数据流

限的特性，只有将它们融合起来才能提高性能。

对于这类计算密集型算子链设计融合优化十分重要，设计融合的本质是设计融合数据流。融合数据流指的是关于如何通过片上内存层次结构将数据从片外内存阶段性地传输到计算单元从而一次性完成多个算子计算的调度方案[89]。不同的数据流可能使用不同的执行顺序、资源绑定方案和循环分块策略，从而导致不同的性能（例如，延迟和能耗）。例如，FLAT[89]为自注意力层提出了一种基于行计算的数据流，它为第一个批量矩阵乘法（$S = Q \times K$）和 softmax 算子按行进行切分，并阶段性地存储中间结果。本文在图 2.2的部分 a) 展示了这种数据流。Fused-Layer[90]为卷积链提出了一种基于分块的数据流，通过在片上内存中阶段性地存储中间图像（Act）的一个矩形分块实现融合。本文在图 2.2的部分 b) 展示了这种数据流。

然而，在已有的编译工作中，融合数据流设计以及相关代码生成优化的工作仍十分稀少，在 BOLT[75]中提出了基于手写代码模板进行矩阵乘法之间以及卷积之间的融合的优化方法，但是其拓展性较弱，难以直接支持 Transformer 这类计算。因此，如何系统性地支持计算密集型算子融合，是图层编译工作将要攻克的下一个重要问题。

2.2.2　算子层相关工作

算子层编译关注对于单个 AI 算子的循环结构自动变换和调度从而优化执行性能。对于 AI 芯片，算子的不同循环结构带来的执行性能差异巨大，主要原因在于不同循环结构的并行性和局部性差异较大。并行性指同时进行的计算操作数量和需要进行这种并行计算的次数。局部性指在同一计算单元数据复用的比率，局部性越高表示数据复

用率越高。并行性和局部性往往是相互矛盾的，过高的并行性会造成局部性的降低，反之亦然。例如在 GPU 上，同时存在大量并行计算单元（流处理器），同时使用这些计算单元进行计算，可以最大化并行性，但是每个计算单元分得的数据量会减少，且共享的数据需要多次拷贝，局部性就会降低。值得注意的是，这种矛盾会在计算任务的规模达到一定阈值后减弱（例如输入的计算形状较大时），主要原因是当计算的并行度明显超过硬件能提供的并行能力之后（比如矩阵乘法分块后块的数目远超 GPU 提供的流处理器数目），并行度将趋于常量，局部性优化将成为主要问题。在本文中，主要考虑 AI 计算的部署推理场景，因此计算的并行规模尚未显著超过硬件并行能力，因此本文讨论的算子层编译优化仍然是在有限的（但是十分巨大）循环调度空间内，寻找平衡并行性和局部性，从而最大化性能的调度方案。如图 2.1中所示，算子层编译方案配合手工优化的算子库共同支撑了算子层 AI 计算，其中手工优化算子库主要用于向上对接图层工作，而算子编译器则关注单个算子的代码生成，用于用自动化方式替代和补足算子库。

在算子层做优化，当前学界和工业界已有数十年的经验积累。例如，大多数深度学习框架[91-94] 依赖于算子库以实现高性能。在 CPU 上，高性能库 MKL[95] 可以加速线性代数应用，而 oneDNN[86] 则专为深度学习应用设计。在 GPU 上，高度优化的库包括 CuBlas[85]，CuDNN[84]，和 CUTLASS[96]。CuBlas[85] 可以将线性代数核心加速到极高性能，而 CuDNN[84] 通过实现一组高效算法（如 Winograd[97] 和 FFT[98]）来加速深度学习应用。CUTLASS[96]则是通过设计一系列高性能的代码组件实现线程级别、块级别、算子级别的计算和数据搬移操作。在 FPGA 上，大量手工开发的高性能算子库[99-100] 通过设计硬件数据流来实现高性能卷积。所有这些算子库都需要手动设计高性能实现，这需要多年的算法和硬件方面的经验和专业知识。例如，在 GPU 上优化算法时，最终能否达到高性能取决于单线程指令优化和多线程并行性[101-103]；在 CPU 优化算法时，则需要复杂的指令优化技术[104-105]。

然而，随着深度学习算法的快速发展，算子库过长的开发时间和沉重的人力成本成为了软件栈迭代的瓶颈。为了解决这个问题，算子层编译工作被提出。ATLAS[106]、BTO[107]、FFTW[108] 和 Leo[109] 使用自动调优来实现高性能。Halide[38] 可以为图像处理流水线生成高性能代码。TACO[53] 可以为稀疏运算算子生成代码。TVM[17] 可以为许多不同类型的硬件生成算子代码。HeteroCL[59] 专门为 FPGA 提供了代码生成后端。

按不同技术路线来介绍，对于计算调度分离路线，Halide[38]提出了多代自动优化方法，包括基于模板的调度器[40]和基于规则组合的调度器[39]。特别的，基于规则的调度器通过使用树搜索和随机程序提供了完全自动的程序优化方法，可以扩展到整个图级别，但其主要关注点仍然是图像处理流水线的代码生成，对于 AI 算法支持仍有不

足。TVM[17]也提出了多代自动优化方法，从基于模板调优参数的 AutoTVM[17]到基于规则组合优化的 Ansor[26]，TensorIR[46]，和 SparseTIR[81]。对于多面体模型技术路线，Tensor Comprehensions[44]最早将多面体模型调度应用在 AI 算子层，但是由于调度空间设计的局限性，当时并未取得好的性能结果。后续 Tiramisu[30]和 AKG**akg**都再次尝试了这一路线，并取得了明显的性能提升，这才正式将多面体模型技术引入到 AI 编译。对于追踪编译技术路线，MLIR[41]对于单个 GPU 的矩阵乘法代码生成已经可以接近手写优化结果，TorchScript[48]对于 Python 原生代码翻译和 GPU 代码生成可以辅助长尾算子的加速，Triton[47]对于自注意力结构的代码生成也可以媲美手写。

2.2.3 指令层相关工作

指令层编译工作主要关注指令选择和指令调度优化。在传统编译器如 GCC 和 LLVM 中，这类优化使用较多。然而，已有的方法关注较多的是通用处理器的指令优化，对于 AI 芯片，特殊指令的使用更为重要。特殊指令的功能包含计算和访存，相比通用指令，特殊指令的操作数往往是多个数据（向量、矩阵，甚至图像），完成的操作也会更加复杂，比如向量运算、矩阵乘法等。因此对 AI 芯片的指令选择和调度优化成为了新的研究问题。如图 2.1中所示，指令层编译是最接近 AI 芯片硬件的，这一层在传统编译技术如 LLVM, GCC, NVCC 的基础上搭建针对 AI 芯片的定制化编译，针对专用指令集进行代码生成。

以 dMazeRunner[110]、CoSA[22]、SARA[23] 为代表的编译器通过细粒度描述处理单元（PE）的数据流来完成算子向硬件的映射。他们需要硬件所有细节（特别是 PE 之间的互联）都暴露给编译器（也就是本文之前介绍的硬件感知型方法）。然而，这类方法在实际 AI 芯片编译中使用受限。

对于计算调度分离路线，Halide[38]，TVM[17]，AKG[31]，VeGen[71]等编译器都采用只映射到指令的方法，而不关心指令如何映射到硬件（也就是本文之前介绍的指令感知型方法）。这类编译问题主要关注的问题是在给定面向通用处理器编程的循环程序（例如用 for 循环）情况下，如何自动在程序中正确并高效使用硬件提供的特殊指令。这一问题也被称为自动张量化问题（AutoTensorize）。正如表 2.1中展示的那样，Halide 是计算调度分离技术路线的最早的代表工作，它使用向量化（Vectorize）等编译优化原语支持特殊指令，由于在早期专用加速芯片的指令仅仅涉及向量运算，Halide 中并没有对于矩阵或者张量指令的特殊支持。而 TVM 通用化了这个概念，提出了张量化的概念，在其中使用 ‘tensorize‘调度原语表达对张量特殊指令的映射。

对于多面体模型路线，AKG 是重要代表工作，在此之前虽然有大量的多面体模型编译器工作[42,111-112]，它们在 AI 芯片的性能效果一直不理想，也不支持特殊指令。

AKG 通过扩展多面体模型概念，在算子数据搬运方面进行调度优化，从而达到了较好性能，在支持 CPU, GPU 的同时，也支持了华为的 NPU。AKG 支持特殊指令的方式是通过识别固定的计算模式来固定化特殊指令生成方法。例如在 Tensor Core GPU，仅支持固定计算（矩阵乘法，卷积）的最内层循环向特殊指令的映射，换做其他算子，就无法自动使用特殊指令了。

至于基于追踪编译这条路线，JAX[56]，TorchScript[48]，Triton[47]，TensorIR[46]等工作都通过从 Python 的语法树或运行时获取计算的语义和结构，通过层层编译，向 AI 编译器生成代码。对于特殊指令，需要在输入的 Python 程序中显示提供相应的函数调用，标明每个分块的循环内部的子计算的类型（如点乘、张量数据搬移等），这样就可以按照标记进行确定的指令生成。这类工作的主要局限性在于需要编程者提供特殊指令的使用指导信息，需要一定的编程开发门槛。

2.3 AI 编译发展的三个阶段

AI 编译技术自出现以来，大致经历了三个发展过程，在不同的阶段内，学术界和工业界的研究重点也有所不同。本文在图 2.1中展示了这三个阶段。第一个阶段是模板调优阶段，这一阶段的发展基础是研究人员对于少数硬件上的少数算子的长期优化和开发总结出的高层次经验，以模板的形式呈现，配合以可调优的参数，可以做到针对多种使用场景（主要是输入形状）自动搜索最优参数，然后生成代码。这一阶段的代表作有 FFTW[108]，ATLAS[106]，AutoTVM[25]等。其核心技术是设计模板和配套的搜索技术，使用启发式算法或者机器学习算法（如模拟退火[113]，决策树[39]）等辅助搜索参数。这类方法的优势是在人力探索经验较多的算子（如矩阵乘法，卷积）和硬件（如 Nvidia GPU）能取得接近最优人工方案甚至超过人手工优化的效果。但是其缺点在于不能很好地适应新兴算子和新兴硬件，仍然需要大量的专家知识设计模板。

第二个阶段是规则组合优化。这一阶段通过将人工优化算子的过程抽象为一系列的优化规则，设计自动化方法模仿人力调优的过程来自动化组合这些优化规则，实现对于大量新兴算子的自动迁移和优化。编译器可以自主分析不同算子的循环结构从而定制化地生成优化空间，不需要人工介入设计模板，从而大大提高了生产效率。同样的，这类方法也需要配合搜索算法，如遗传算法，强化学习算法等。相比模板调优，规则组合更加具有适应性和泛化性，并且取得的性能结果与模板调优不相上下。这一阶段的代表性工作有 Ansor[26]，EXO[72]，MetaSchedule[76]等。本文接下来提出的算子层技术也属于这一阶段。

第三个阶段是分析优化，这一阶段希望避免黑盒式调优带来的时间开销问题和盲目性问题，希望通过编译器提供的分析算法，快速地解出最优或者近似最优的方案。这

类技术的关键是对分析的目标建模以及设计相应的优化函数，配合自动化优化技术（如线性规划、基于梯度的优化）完成代码调优。这类方法可以配合手写代码片段进行代码综合，结合了编译技术和手工优化技术的优势，而不是像先前两个阶段明确与手写代码区隔离分开。这类方法的优势是可解释性和高效性，往往只需要较短的时间（秒级或分钟级）就可以得到生成代码，且性能仍然较好。目前大量主流编译工作都采用这一方法，代表工作有 MLIR[41]，Triton[47]，TorchScript[48]，JAX[56]，MOpt[114]等等。本文接下来提出的图层和指令层技术都属于这个阶段。

2.4　AI 芯片相关工作

本文简单介绍 AI 芯片的相关工作，主要概括有代表性的加速器设计以及针对数据流的性能预测工作。

加速器设计与搜索：近年来提出了各种加速器[9,12-13,115-122]。具有代表性的加速器包括麻省理工提出的 Eyeriss[115]系列，清华大学提出的 Thinker[123]系列，Google 提出的 TPU[9]系列，寒武纪提出的思元 NPU[11]系列，华为提出的 Ascend NPU[13]系列等等。这些加速器采用不同的数据流进行 DNN 加速。为了充分发挥这些加速器的高性能和高能效，与芯片一同被提出的还有很多数据流设计方案与配套的自动化设计系统[22-23,27,124-127]。例如，ConfuciuX[124] 使用强化学习来搜索高效的硬件资源分配和数据流。CoSA[22] 提出使用混合整数规划来优化空间架构上的计算和存储的映射。SARA[23] 可以将通用程序编译为具有高并行性的可重构数据流并映射到加速器芯片。

性能模型：性能模型[32,34-35,128-131] 对于数据流探索和架构设计都是很重要的。例如 Timeloop[32] 使用空间 / 时间描述法来描述映射，并可以分析不同定制架构的延迟、能量和数据重用。MAESTRO[34] 使用数据为主的描述法来描述数据流，并通过分析公式计算性能指标。Scale-SIM[128] 为 systolic array 架构提供了一种系统的性能建模框架。Interstellar[33] 提出使用 Halide[38] 调度原语来描述 AI 芯片的各种数据流。TENET[35] 提出了一个以关系为中心的描述法，并使用多面体方法进行延迟估算。Sparseloop[129] 通过对计算块的稀疏性和稀疏格式进行建模，为稀疏算子提供性能建模。这些模型侧重于单算子的性能推演和预测。对于多算子融合场景，性能模型更加重要，因为复杂的算子间数据移动和硬件片上内存层次结构紧密耦合，依靠人力难以计算。MAGMA[132]、NNest[133]、Dnn-chip Predictor[134]、HDA[135]、HASCO[24] 和 H2H[136] 通过分别使用单算子性能模型评估每个层的延迟，然后减去中间数据传输的开销来来预测整个计算图的延迟。然而这种简单的预测方法并不准确。本文将介绍一种更为细致的性能模型构建方法，来预测融合数据流的性能。

2.5 整体编译抽象

贯穿多层编译技术的是对于程序的形式化和自动化，因此需要借助抽象的表达方式描述程序，也就是编译器的中间表达方式（Intermediate Representation，IR）。不同层次需要不同的抽象方法，但是这些不同抽象都可以追溯到同一的编译抽象：张量表达式和有向无环图。因此本 j 节从高层次介绍每层的编译抽象与统一抽象的关系。

2.5.1 统一编译抽象

统一抽象是所有编译层次抽象的基础，包含张量表达式和有向无环图两个概念。首先介绍张量表达式。在章节 1.1.1中本文介绍了 AI 计算的基本组成单元是张量计算，同时本文也介绍了张量计算的表达方式。张量计算大多可以通过爱因斯坦求和约定表达，这也使得本文可以在编译器中使用表达式的方式描述张量计算。

定义 1 张量表达式由四部分组成：输入输出张量，访存下标，迭代循环，以及计算算符。形式化表达为 $Out(Idx_{out}) = Accum(Op(In_1(Idx_{In_1}), ..., In_N(Idx_{In_N})), loops)$。其中 Out 是输出张量，用来存放结果，$In_1, ..., In_N$ 是输入张量，用来存放输入数据，Idx_{out} 是输出张量的访存下标，$Idx_{In_1}, ..., Idx_{In_N}$ 是输入张量的访存下标。一般来说，访存下标可以是由循环变量组成的下标表达式，也可以是间接访存得来的张量中的值（如稀疏计算的下标），不过在本文中只讨论下标表达式的情形，也就是说只考虑稠密计算的场景。$loops$ 是该张量计算需要的循环，一般用循环变量代表，可以显式表达，也可以通过编译器借助访存下标推测。$Accum$ 是张量计算的聚合计算算符，一般常见的计算算符有累加（sum），最大和最小（max, min）等，不需要聚合计算的时候可以省略。聚合算符一般需要沿着一些维度进行，可以在表达式中显示表达出来，也可以略写，通过编译器分析出来。Op 是逐点运算算符，包括加减乘除等常见运算。

下面给出一些张量表达式的例子。用张量计算描述矩阵乘法，可以写成

$$C(i, j) = sum(A(i, k) * B(j, k), axis = k) \tag{2.1}$$

其中 C 是输出矩阵，A, B 是输入矩阵，逐点算符是乘法，聚合算符是累加，累加沿着维度 k 进行，访存下标是循环变量的简单组合，表达稠密运算。这里省略了 $loops$，因此编译器可以通过表达式自动推理存在三个不同的循环变量 i, j, k，因此有三个循环。也可以略写 $axis = k$，编译器推理 k 是累加维度的方式是：表达式等号右侧出现而左侧未出现的循环变量，一定对应累加维度。这一分析方法被广泛使用在许多编译工作中[53]。此外，本文约定，每个表达式自身的 $loops$ 是不同的，尽管他们的代号（循环名字）可以写成一样，但是读者应清楚他们在程序中是不同的实例（数据结构存放在不同

的内存地址）。在表达式中，由于聚合运算的存在，不同的循环的语义也不同，比如矩阵乘法里的 i, j 不参与聚合，他们的计算可以被并行化，本文称之为并行循环（spatial loop），与之对应，k 这类聚合循环，被称为规约循环（reduction loop）。这些语义也都可以在编译器中自动分析得到。为了帮助理解，本文再提供一个卷积的表达式例子

$$C(n,k,h,w) = sum(A(n,c,h+r,w+s) * B(k,c,r,s), axis = (c,r,s)) \tag{2.2}$$

和一个 ReLU[14]运算的表达式例子。

$$B(i,j) = A(i,j) >= 0\ ?\ A(i,j)\ :\ 0 \tag{2.3}$$

张量表达式描述了一个单独的计算步骤（算子），而且从定义来看，天然与完美嵌套循环对应。在 AI 计算中，一个完整的算法是通过大量算子拼接而成，为了表达这种拼接，需要下一个抽象：有向无环图。

定义 2 有向无环图 $G =< E, T >$ 包含顶点集合 E 和边集合 T，顶点集合中的元素是张量表达式，边集合中的元素是张量。对于 $expr_1 \in E$，$expr_2 \in E$，$expr_1$ 的输出张量是 t，如果 t 是 $expr_2$ 的输入张量，那么本文就有 $t \in T$，代表 $expr_1$ 和 $expr_2$ 之间存在一条边，$expr_1$ 是生产者，$expr_2$ 是消费者。

在编程时，有向无环图可以通过静态单赋值（Static Single Assignment, SSA）形式写出来，下面本文给出一些具体例子。

例 1. 卷积层： 在卷积神经网络中常用的卷积层并不是一个单独的算子，而是包含了多个算子，所以需要多个表达式组合得到。通常卷积层包含一个填充运算（padding），一个卷积运算，一个归一化运算（如 batch normalization），以及一个激活运算（如 ReLU）。这样的结构可以如下书写：

$$\begin{aligned}
Padded(n,c,p,q) &= (p - R/2 >= 0)\ \&\ (p - R/2 < H)\ \&\ (q - S/2 >= 0)\ \&\ (q - S/2 < W)\ ? \\
&\quad Img(n,c,p - R/2, q - S/2)\ :\ 0 \\
Conv(n,k,p,q) &= sum(Padded(n,c,p+r,q+s) * Weight(k,c,r,s), axis = (c,r,s)) \\
BN(n,k,p,q) &= (Conv(n,k,p,q) - Mean(k))/Std(k) \\
ReLU(n,k,p,q) &= BN(n,k,p,q) >= 0\ ?\ BN(n,k,p,q)\ :\ 0
\end{aligned} \tag{2.4}$$

这一段代码中，R, S 是卷积窗口的大小，一般 $R = S = 3$，Img 是输入的图片，数据格式是 NCHW，$Weight$ 是卷积权重，本文使用的是 batch normalization，为了简便，本文只考虑推理场景的 batch normalization，因此均值（$Mean$）和标准差（Std）都是提前计算好的。这里本文再次强调，每个表达式自身的 $loops$ 作用域仅在于该表达式本身，

因此尽管不同表达式的 $loops$ 名字有重复（如 n,k,p,q），他们在程序中是不同的实例。另外，在不同的例子中相同的符号也没有关联。

例 2. self-attention：在 Transformer 中经常使用的 self-attention 层也包含了许多算子，尽管不同的文章对于 self-attention 的结构边界在何处有些不同，在本文中，本文采用最狭义的定义，认为 self-attention 仅包括两个批量矩阵乘法和一个 softmax，但是 softmax 本身是由五个不同的算子构成的，用有向无环图抽象写出来结果如下：

$$
\begin{aligned}
S(b,h,i,l) &= sum(Q(b,h,i,k) * K(b,h,k,j), axis = k) \\
Max(b,h,i) &= max(S(b,h,i,l), axis = l) \\
Sub(b,h,i,l) &= S(b,h,i,l) - Max(b,h,i) \\
Exp(b,h,i,l) &= exp(Sub(b,h,i,l)) \\
Sum(b,h,i) &= sum(Exp(b,h,i,l), axis = l) \\
Div(b,h,i,l) &= Exp(b,h,i,l)/Sum(b,h,i) \\
A(b,h,i,j) &= sum(Div(b,h,i,l) * V(b,h,j,l), axis = l)
\end{aligned}
\tag{2.5}
$$

第一步本文计算的是 Q 和 K 矩阵的乘积，得到了注意力值，紧接着对这个中间结果进行求最大值、减法、exp、求和、除法等一系列运算，最后得到的结果再乘以 V 矩阵得到 self-attention 的结果。

张量表达式和有向无环图组成的统一抽象贯穿 AI 编译所有层次，在几乎所有的编译器实现（如 TVM[17]，Triton[47]，MLIR[41]）中都能找到。下面，本文将逐层介绍统一抽象如何使用，为了符合编译流程自然顺序，本文将从上到下进行介绍（从图层到算子层再到指令层）。

2.5.2 三层次编译抽象

这一小节简述三层次各自的编译抽象与统一抽象的联系。在图 1.7中也展示了统一抽象和各个层次抽象的关系。

- 在图层次提出的抽象是块抽象。块抽象的本质是将融合程序（融合数据流）的程序结构用树状结构进行表达，树的节点表示一个局部的嵌套循环，因此块内可以用一个张量表达式进行表达，树的公共节点的循环被所有子节点共享，对应了融合程序的公共外层循环。节点间的数据传输关系依旧可以通过有向无环图描述，因此图层次的块抽象是基于张量表示和有向无环图设计提出的。
- 在算子层次使用的抽象是循环抽象。循环抽象通过将块内的嵌套循环抽象为一列循环变量，以记录循环起点、终点、步长的方式将循环信息存储在循环变量中。因此，循环变换就变为在循环变量列表上的变换操作，在后续对程序结构

进行编码时，也可以利用向量来编码嵌套循环。由于需要处理来自多个算子的循环结构，需要使用有向无环图对多算子的图结构进行表达，循环内的计算仍然使用张量表达式进行表达。所以算子层的抽象也是基于张量表达式和有向无环图进行设计的。

- 在指令层使用的抽象是计算访存抽象，这一抽象把芯片特殊指令的计算行为和访存语义进行抽象表达，本质上是在张量表达式基础上附加额外的操作数所在内存层次信息，区分读取、计算、和写入步骤等，从而针对芯片特殊指令进行描述。因此指令层抽象主要是基于张量表达式进行设计。

2.6　整体编译流程

本章结尾，我们从高层次概括本文在三层次编译中的技术，在后文中将对这些内容进行详细展开。

如图 1.7中展示的那样，AI 编译器三个层次有不同的职责。图层次需要设计图 IR，完成图融合优化，必要时进行子图替换和分布式优化。算子层需要设计循环 IR，以及设计在循环上进行分块、调度优化、多面体变换等优化手段。指令层需要设计指令 IR，设计张量化、向量化、指令选择和优化算法。本文提出的三个主要技术分别对应图、算子、指令层次。特别地，在每个层次，本文分别专注一个研究问题：

- 图层次：针对图融合技术进行研究，提出基于块抽象的图融合编译技术，设计实现 Chimera 和 TileFlow 两个框架，Chimera 提供图编译和代码生成能力，TileFlow 提供融合数据流性能分析功能。
- 算子层次：针对循环调度变换问题进行研究，提出基于循环抽象的算子优化技术，设计实现 FlexTensor 框架，完成自动化生成调度空间并搜索高效调度策略的功能。
- 指令层次：针对指令自动生成的张量化问题进行研究，提出基于计算访存抽象的指令生成技术，设计实现 AMOS 框架，将硬件芯片的指令进行抽象，实现指令的自动匹配、生成、验证功能。

本文提出的 Chimera，TileFlow，FlexTensor，AMOS 框架相互配合，共同构建从 AI 算法的计算图到最终 AI 芯片底层代码的编译流程。在图 1.7中展示的本文主要技术之间是双向箭头，因为不同层次需要相互配合才能完成编译，下一章节会介绍这些层次如何共同构建编译流程。

第三章　基于块抽象的图层次编译

本章节将讨论图层次编译的技术。首先本文详细分析先前工作在这一层次的主要技术局限性，随后介绍本文为图层次专门设计的抽象：块（Tile），并在块抽象的基础上提出两个框架 Chimera 和 TileFlow。随后，本文介绍 Chimera 和 TileFlow 的具体编译优化技术。

3.1　图编译技术背景介绍

3.1.1　基于算子符号的图抽象

从深度学习网络成为 AI 算法的主流之后，如何快速高效搭建深度学习网络一直是系统软件希望解决的问题。最初的深度学习框架如 Caffe[94]就通过抽象出层的概念搭建网络，编程时使用者需要明确声明所使用的深度学习计算的层（卷积、全连接等），并通过张量连接这些层从而构建深度学习计算流水线。这一抽象后续被广泛使用在 TensorFlow[91]，PyTorch[92]，MXNet[93]等框架中，这对应着本文在前文中所提及的统一抽象：有向无环图。基于这样的发展脉络，图层次编译自出现就开始使用算子符号作为抽象方法。Glow[58]，nGraph[57]，NNVM[17]，TASO[18]，Relay[37]等等编译器都是使用了这样的抽象方法。所谓算子符号，就是将深度学习网络中的每个层或都使用相对应的符号进行代表，用简单的符号来简洁地概括其内部蕴含的算子信息。比如卷积层，使用 Conv 符号代表，全连接层使用 FC 符号代表。每个符号都有输入输出参数，如 Conv 符号有两个输入，一个是图像，一个是权重；有一个输出，是卷积计算后的结果。

基于算子符号的抽象方法允许编译器进行许多图层次优化，因为不用关心符号内概括掉的算子细节信息，图优化算法可以只关注符号间的相互关系，从而把算法设计得简洁高效，这些算法包括公共子图替换，拓扑结构变换，图融合，图替换等等。在这些优化中，本文着重关注图融合优化。算子符号方法进行图融合时，需要由编译器多次扫描有向无环图，匹配出可以融合的子图结构，然后将这些子图替换为新的算子符号，表明融合操作的结果。匹配过程依照的规则，是编译器开发者手工设计的，往往都是对一些常见融合策略的模式匹配。比如常见的结构有卷积与归一化（如 batch normalization）以及 ReLU 的融合。当编译器扫描到的子图与所设定的模式完全匹配时，这个子图就会被标记为融合算子。比如卷积（Conv）、batch normalization（简写为 BN）、ReLU 的融合后算子符号可以写成 ConvBNReLU。这类融合方法十分直接且容易扩展，每当有新的融合模式需要加入的时候，就可以增加对应的匹配规则以及新的融合后符号。

然而，基于算子符号的方法的缺点也很明显，编译器自身很难对于融合本身进行性能分析和指导，需要依赖专家总结的融合模式进行匹配。融合本身对于图的改写也会影响后续的融合，如果前置的融合操作破坏了一些模式，后续的融合可能就无法开展。有些工作通过增加可回溯的调优过程的方法来帮助编译器寻找更好的融合方案。比如DNNFusion[19]通过多次尝试融合的方式来选择性能最好的融合方案，TASO[18]使用等价性判断和数学规则指导探索融合策略，PET[70]使用非等价规则以及修正技术进一步拓展了 TASO 的优化空间。尽管这些探索一定程度上解决了算子符号抽象面临的问题，他们还是无法解决新算子的融合问题。如果模型图结构里出现了新的算子符号而缺少对应的融合规则，这些算子就不会被优化。为了解决这一问题，基于循环抽象的图抽象方法被提出和使用。

3.1.2 基于循环抽象的图抽象方法

为了支持新算子，图层次编译必须具备能为新算子进行代码生成的能力。由于 AI 计算的算子大多是张量计算，且可以使用前文介绍的张量表达式进行表达，许多编译器便采用循环抽象的方式来支持融合，基本思路是基于循环的抽象，增加对于非完美循环的支持。在循环抽象方式中，图中的每个算子都有其对应的循环结构，如全连接层对应三层循环的矩阵乘法，卷积层对应七层循环的卷积运算。当融合发生时，这些被融合的算子的循环结构发生改变，一些算子的循环会作为子循环嵌入到另一些算子的循环结构中。融合后生成的循环结构可以进一步进入算子层进行代码生成，这样就解决的对新算子的融合支持问题。TVM[17]，AutoTVM[25]，Ansor[26]等编译器都使用了这种方法。

不过循环抽象的图优化比基于算子符号抽象的更为复杂，其原因是编译器需要控制的粒度更细。每个算子都提供了大量的信息以便编译分析。例如，逐点计算的循环结构与规约计算的循环结构有明显的差异，规约计算存在更多的数据复用机会，但是多个规约计算的融合可能导致大量的重计算，而逐渐计算几乎没有数据复用机会，它们的融合往往是必然有收益的。另外，算子融合时循环优化空间也会更大，生产者算子的循环体理论上可以被插入到消费者算子的循环体的任意部分，这就导致融合时的选择空间组合爆炸。如果需要编译器对这样的细节都进行详尽的分析，将产生一个十分巨大的优化空间，对这样的空间进行全面的搜索是一个艰巨的挑战。现有的技术如AutoTVM[25]选择依赖人工提供的融合算子的生成模板指导融合后的代码生成，从而避免在巨大的空间中搜索，Ansor[26]则通过一些对融合设计一些代码生成规则来辅助融合优化，这些方法都是在巨大的优化空间中只关注一部分可行方案，因此也会错失一些融合优化的机会。

表 3.1　对 AI 模型的计算和访存占比的分析以及对 AI 芯片的计算与访存性能的分析

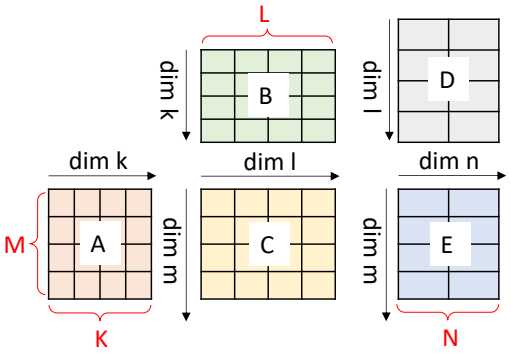
AI 模型计算和访存占比			
名字	**% 访存占比**	**% 计算占比**	**% 批量矩阵乘法占比**
Transformer	19.45%	40.51%	40.04%
Bert-Base	30.56%	42.79%	26.65%
ViT-Huge	15.63%	50.85%	33.52%
AI 芯片的计算访存性能分析			
设备	**Xeon Gold**	**A100**	**Ascend 910**
计算单元	AVX-512	Tensor Core	Cube Unit
峰值性能	12 TFlops	312 TFlops	320 TFlops
片外峰值带宽	131 GB/s	1555 GB/s	1200 GB/s
峰值性能除以峰值带宽	92 Flop/byte	200 Flop/byte	267 Flop/byte

3.1.3　计算密集型算子的访存性能受限问题

由各种张量计算组成的 AI 模型结构比较多样[2,5,15,87-88,137-139]，在其中的算子则可以分成两种类型：计算密集型算子（如矩阵乘法和卷积），这类算子占据了大部分的计算量；以及访存密集型算子（如 ReLU 和 softmax），这类算子用于连接不同的计算密集型算子。手工优化算子库[84-86,92,95-96,102,140]和代码生成编译器[17-19,26,30-31,37,44,74-75,141-142]都在着重优化这两类算子。近几年，AI 模型的结构也在发生变化，之前的 AI 模型是围绕一个计算密集型算子（如矩阵乘法和卷积）使用一些访存密集型的逐点计算或归约计算构成算法模块，而最近的模型倾向于将多个计算密集型算子组合在一起。在图 1.5中本文展示的两个典型的例子所示，在部分 b) 中展示的是一个自注意力（self-attention）层，它在基于 Transformer 的模型中广泛使用，如 Bert[87]和 ViT[139]。它的主要组成部分包括一个批量矩阵乘法链，它由两个批量矩阵乘法和一个 softmax 算子组成。因此这个算法模块中包含链式的计算密集型算子。

随着硬件架构定制化技术在 AI 芯片中普遍应用，专用计算核心峰值算力与片外存储的数据搬移速度之间的差异变得越来越明显。结果导致许多计算密集型算子受到内存带宽的限制变成了访存受限算子。在表 3.1中，本文展示了几种芯片的半精度峰值计算性能和片外内存带宽，包含了 Xeon Gold AVX-512 CPU、A100 Tensor Core GPU[143]，以及 Ascend 910 NPU[13]。这些芯片的峰值性能与内存带宽的商表示了为了达到峰值性能，张量算子所需达到的最小计算密度。在表中，可以发现，这些芯片所需的最小计算密度都较高，这表明这些芯片需要算子的计算密度足够高才能充分发挥性能。例如，要释放 Xeon Gold CPU 的计算能力，每读取一个字节至少需要进行 92 次浮点运算。

此外，计算性能与内存带宽之间的差距将继续增大。许多计算密集型算子（例如，Transformer 中的批量矩阵乘法链）的数据搬移开销将成为性能瓶颈。本文在表 3.1中展示了一些常用模型的执行时间分析（测试的输入序列长度设为 512）。列*% 访存占比*指的是所有访存密集型算子占用的执行时间比例；列*% 计算占比*指的是除了自注意



		张量的数据重用维度			
序号	顺序	A	B	D	E
1	mnkl	l	-	k	k,l
2	mnlk	-	-	k	k,l
3	mknl	n,l	-	k	k,l
4	mkln	l,n	n	k	k
5	mlnk	-	-	k	k
6	mlkn	n	n	k	k
⋮	⋮	⋮		⋮	⋮
24	lknm	-	n,m	m,k	m,k

图 3.1　以矩阵乘法链为例解释不同的执行顺序影响数据复用，从而影响性能
.

力层中的批量矩阵乘法之外的计算密集型算子所占的比例；而列% 批量矩阵乘法占比指的是批量矩阵乘法占用的比例。如表中所示，批量矩阵乘法占据了大量的执行时间（26.65% 到 40.04%），这超过了其他访存密集型算子。因此，对这些计算密集型算子进行优化，以提高局部性和减少对内存带宽的压力是必要的。

算子融合是对受限于访存性能的算子进行优化的有效方法。然而，计算密集型算子经常构成具有严格数据依赖性的链式结构，为这种计算密集型算子链生成高效的融合算子需要兼顾不同算子的输入输出的数据搬移量的最小化，这就为性能优化带来了巨大的挑战。具体来说，首先，想要找到最优的计算密集型算子链执行顺序很难。如前期工作[73,127]所指出的，链中的每个算子都可以被分解为一系列计算块。这些计算块的不同执行顺序可能会导致块之间的数据搬移量不同，因此性能也会发生巨大变化。现有工作[18-19,26,73-74]无法优化计算密集型算子链的执行顺序，因为它们缺乏一个精确的性能模型来评估不同排序情况下的数据搬移量。其次，充分利用硬件底层特点优化每个块内的计算也是具有挑战性的。当前缺乏一种针对不同 AI 芯片的可扩展且灵活的代码生成方法。先前的工作[19,74-75]使用固定的微内核（Micro Kernel）设计方法，但是它们很难泛化到其他硬件芯片上。因此需要的是更加通用的抽象，为微内核代码的替换提供便利。

3.1.4　计算密集型算子融合的主要挑战

3.1.4.1　块间的执行顺序优化困难

如先前工作[73,127]所指出的，计算密集型算子可以通过一系列计算块来表示，而块的执行顺序对于局部性优化至关重要。当融合计算密集型算子链时，主要的优化目标是选择最佳的执行顺序，以最大化数据重用。本文使用图 3.1中的矩阵乘法链示例（$C = A \times B, E = C \times D$）来说明不同执行顺序的效果。矩阵乘法链虽然看起来有六个

表 3.2　本文提出的 Chimera 框架与先前工作的对比

工作名称	块间的优化方法	块内的优化方法	支持的硬件			优化方法
			CPU	GPU	NPU	
AKG[31]	最小化数据复用距离	循环变换	Yes	Yes	Yes	多面体模型
DNNFusion[19]	基于模板的融合	定制微内核	Yes	Yes	No	黑盒调优
TASO[18]	子图替换规则	无	No	Yes	No	黑盒调优
AStitch[74]	函数粘合技术	固定的微内核	No	Yes	No	基于规则
CoSA[22]	最小化计算所需周期数	无	No	Yes	No	线性规划
Atomic[127]	最小化核间数据搬移	无	No	No	No	动态规划
MOpt[144]	优化单算子局部性	定制微内核	Yes	No	No	分析性
Roller[127]	微程序生成技术	生成微内核	No	Yes	No	性能模型
Ansor[26]	调度组合技术	循环变换	Yes	Yes	No	黑盒调优
BOLT[75]	手工优化函数	固定微内核	No	Yes	No	黑盒调优
Chimera (本文)	最小化数据搬移	可替换微内核	Yes	Yes	Yes	分析性

循环，但是其中只有四个独立的维度（m,n,k,l），两个矩阵乘法被切分成多个计算块之后，块的执行顺序就可以通过四个维度的排序来表示，如图 3.1中的表所示。顺序 *mnkl*（第一行）表示首先沿着维度 l 执行块，然后是维度 k，接着是维度 n，最后是维度 m。在这种执行顺序下，矩阵 A 沿着维度 l 被重用；矩阵 B 不被重用，因为当程序沿着维度 l 遍历块时，会访问到矩阵 B 的不同数据块，就发生了数据替换。矩阵 C 是中间结果，存储在芯片片上内存中，所以不必展示它的重用维度。矩阵 D 和 E 始终沿着 k 维度被重用，因为 k 是第一个矩阵乘法的私有维度（在后续矩阵乘法中没有用到）。此外，每个计算块的大小也会影响最终的数据搬移量。因此，优化问题应该将块切分策略和块排序选择一起建模。

先前的工作[18-19,22,26,31,73-75,114,127]只是部分解决了这个问题，如表 3.2所示。Ansor[26]、TASO[18]、DNNFusion[19]、MOpt[144]、AStitch[74]和 Roller[73]只在一次优化中优化一个计算密集型算子内的块顺序，并使用专家经验固定算子间顺序。DNNFusion [19]将计算密集型算子分类为“多对多”映射类型，这种类型的算子之间的融合往往会被忽略，并且因为其代码生成模板无法预测这种复杂融合的性能收益，DNNFusion 不会选择将两个或多个“多对多”算子融合在一起。CoSA[22]使用混合整数编程（MIP）来优化程序总执行周期，但它没有考虑块间的内存访问问题。HASCO[24]使用强化学习和贝叶斯优化来探索硬件-软件设计空间。但算子融合不在其设计空间内。AKG[31]使用多面体模型来进行融合、提高局部性。但多面体模型是一种通用方法，其优化空间太大，往往难以全面探索。因此，它依赖启发式方法来找到解决方案，这在实践中经常导致次优性能。Atomic[127]仅考虑了核间的数据重用（例如，图 3.1中的矩阵 C 的重用）。但输入/输出数据访问的数据重用对性能也有很大影响，这在 Atomic 中没有得到优化。

3.1.4.2 块内代码生成优化困难

在一个块内调度计算对于实现高性能函数（Kernel）也十分重要。块内的指令调度应考虑尽量隐藏内存访问的延迟并最大化计算流水线的利用率。如表 3.2所示，先前的工作采用了不同的方法进行块内优化。TASO[18]、CoSA[22]和 Atomic[127]不生成底层代码，它们没有块内优化。Ansor[26]使用循环变换和调优方法来生成微内核。但它们依赖于通用的指令选择逻辑（在 TVM[17]和 LLVM[45]中实现的）而没有利用硬件特殊指令，比如 AVX-512 和 Tensor Core 的指令，都没办法自动使用，需要人工介入进行使用。此外，这类编译器调优过程的时间代价也很大，因为它需要数百至上千个迭代步骤才能找到好的调度方案。DNNFusion[19]、AStitch[74]和 BOLT[75]使用手工调优的微内核来优化计算过程。然而，他们的微内核与块间优化紧密耦合，使得支持新算子或新芯片变得困难，难以复用先前的工作。

3.1.4.3 缺乏融合后的数据流性能评估工具

融合后的算子的计算、访存流程被称为融合数据流。要设计高性能的数据流，性能模型至关重要。然而，当前最先进的性能模型[32,34-35]专注于单个算子的性能推演。这些模型不能为融合数据流提供灵活且准确的性能预测功能。原因是这些模型将单个算子的计算视为一个迭代多面体，并将性能预测问题表述为由体系结构参数（如 PE 阵列大小）和输入规模参数（如迭代界限）组成的多项式。因此，本文将这些性能模型分类为基于多面体的模型。但是单个多面体表述不适合融合数据流，因为融合数据流的迭代空间不是完美嵌套的。融合会将一个算子的迭代空间插入到另一个算子的迭代空间中，形成不完美的循环嵌套。其他工作[136]通过首先使用性能模型单独对每个算子进行建模，然后从结果中剥离算子间数据搬移延迟，来修改现有模型输出的结果，从而做到能评估融合数据流的性能。本文称这些工作为基于图的，因为它们在建模中只考虑计算图的拓扑结构而不考虑硬件架构细节（如内存层次结构）。但是这种方法一般都很不准确，因为数据在复杂的内存层次中进行搬移过程不是简单的减去算子间数据搬移量就可以确定的，搬移的开销也不能通过简单的加减法进行预测。

3.2 基于块抽象的编译方案整体结构

为了应对上述三个挑战，在本章节中将提出两个框架：Chimera 和 TileFlow。这两个框架在图编译部分使用块作为抽象进行编译优化，块抽象一方面契合对算子进行分块和融合的操作，能比基于符号的方式表达更多的信息，也比基于循环的抽象方法更为简单，因此可以设计简洁有效的优化算法。两个框架分工不同，其中，Chimera 关注解决前文介绍的挑战中的前两个，解决块间数据搬移优化以及块内代码生成，TileFlow

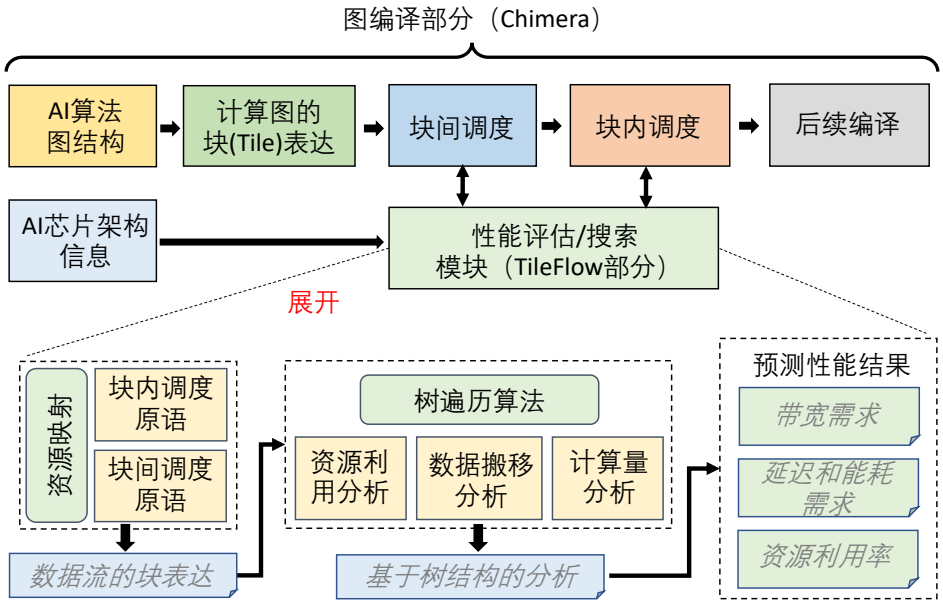


图 3.2　基于块的图层次技术整体结构：包含 Chimera 和 TileFlow 两个框架

.

则是解决第三个挑战，负责对融合数据流的性能进行预测评估，并进一步辅助 Chimera 进行更好的数据流搜索。这两个框架相互配合成为一个整体，在图 3.2中展示的是两个框架的结构和相互关系。具体而言，Chimera 负责整体的编译管线，从上层 AI 算法的图结构出发，逐层编译和优化，经过块间调度和块内调度，然后再继续后续编译（算子层和指令层）。Chimera 自身提供了对融合的分析调度方案，主要技术是通过对数据搬移量进行建模和优化从而优化整体程序的数据流，这一方案可以做到快速、高效地编译，但是只支持线性链式结构，对于更复杂的拓扑结构，建模过于复杂，需要使用 TileFlow 辅助性能预测。在 Chimera 中，块内调度主要针对三种具体的硬件（CPU，GPU，NPU）设计了相应的微内核优化分析方法，当面向新芯片，或者已有芯片发生架构改变，甚至是面向尚未实际流片、仅有最初架构设计的芯片原型的时候，这种分析也是可用的，并且借助可替换微内核概念，跨设备的迁移也是可能的。当输入的图不是线性结构时，就需要借助 TileFlow 框架的分析功能进行性能预测，这可以辅助 Chimera 在数据流设计空间中进行搜索，从而找到性能较高的融合数据流。TileFlow 能更通用的原因是它使用树状分块抽象来表达融合数据流，这一表达方式不受限于计算图拓扑，并且在树上进行块内和块间分析，可以较为准确地得到数据流的执行性能。

如图 3.2上半部分所示，Chimera 由三部分组成：计算图的块表达、块间调度、块内调度。输入的 AI 算法图中的每个算子首先被分解成一系列计算块（Tile）。然后，Chimera 会分析所有不同的块执行顺序的数据搬移量并求解出最优数据搬移量以及相配套的分块参数。之后，Chimera 通过使用可替换的微内核进行块内优化。Chimera 目

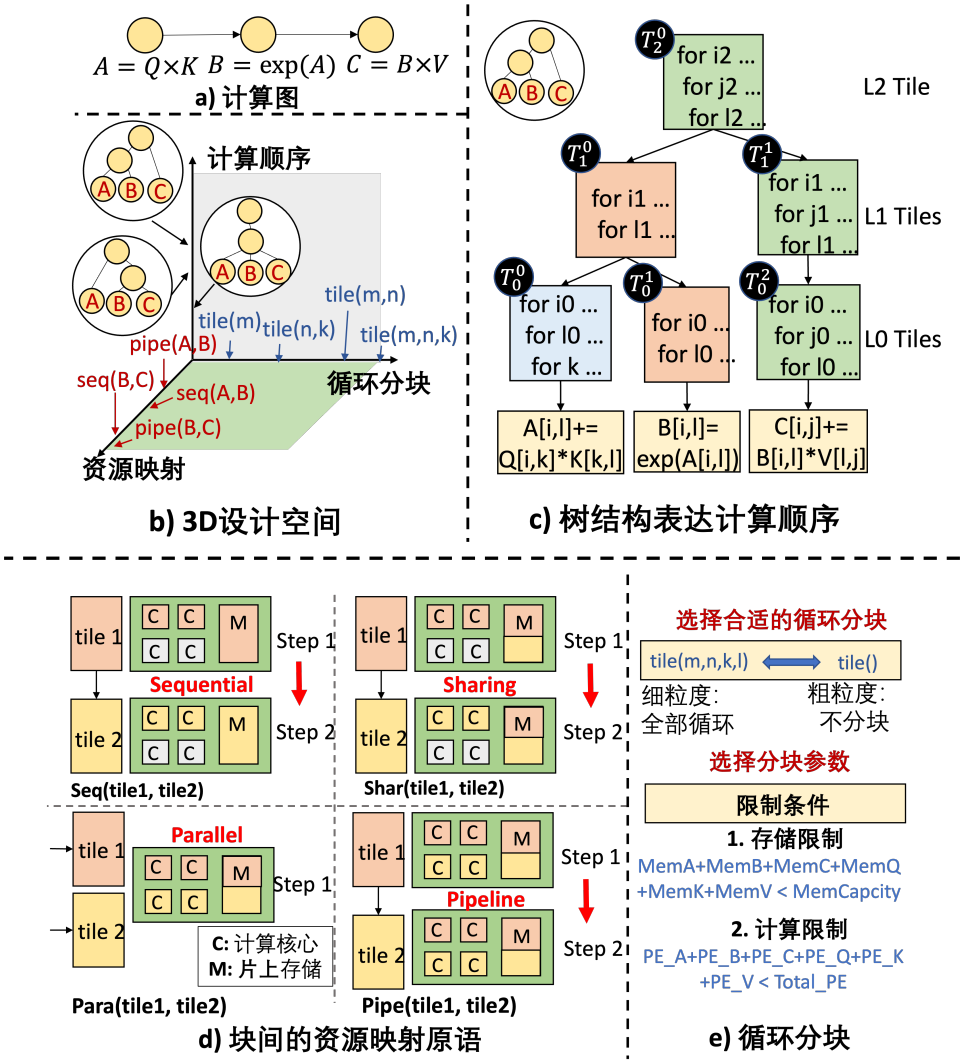


图 3.3　融合数据流设计的 3D 空间

前支持三种不同的硬件后端。对于每个后端，Chimera 将硬件特定的微内核实现注册到一个称为可替换微内核的特殊编译抽象中，以便以统一的方式处理硬件多样性。这种可替换微内核的方式也可拓展，支持更多硬件。

如图 3.2下半部分所示。TileFlow 主要由两部分组成：以块为中心的数据流表示法和基于树的分析方法。以块为中心的表示法在融合数据流的 3D 设计空间中具有完整的表达能力：计算顺序和循环分块通过以块构成的树状结构定义来表达，而资源映射通过两类映射原语表达：块内原语和块间原语。通过以块为中心的表示法表达的融合数据流可以自然地转换为分析树。然后，TileFlow 遍历分析树来计算性能相关指标：片上内存层次结构间的数据搬移量、每一级内存和计算单元的资源使用情况，以及加速

器所需的总计算操作数量。有了这些结果，TileFlow 就可以基于架构规格参数计算性能结果，包括延迟、能量开销和带宽需求等。

3.3　以块为中心的抽象

本节将详细介绍基于块的抽象内容，随后介绍融合数据流的设计空间。

3.3.1　相关记号和抽象

正如前述内容所介绍的，本文的块抽象在基于符号的抽象和基于循环的抽象之间取了折中，在保证灵活性的同时简化优化问题。在这种基于块的表示法中，一个位于内存层级 n 的块（T_n）被定义为

$$T_n = \{l_1^n, l_2^n, ...\}(T_{n-1}^1, T_{n-1}^2, ...) \tag{3.1}$$

其中 $\{l_1^n, l_2^n, ...\}$ 是一个嵌套循环，$(T_{n-1}^1, T_{n-1}^2, ...)$ 是子块列表，通过这个递归定义，本文可以定义一个树结构。这种表示法与融合数据流结构天然对应。本文可以使用这种块定义来表达不同的树结构，从而表达不同的计算顺序。另外，在不同的块内、块间，可以额外指定一些属性（如通过原语指定，后文将介绍四种原语：Seq, Shar, Para, Pipe）来表达不同块的硬件资源映射选择。而每个块中的循环的大小也是可以调整的参数，这将被用来表达分块优化。

作为例子，使用以块为中心的表示法来表达图 3.3部分 a) 中示例的融合数据流，一种可能的融合数据流定义方法如下：

块定义 (计算顺序和分块):

$$\begin{aligned} &level\ 0: T_0^0 = \{i_0, l_0, k\}(A),\ T_0^1 = \{i_0, l_0\}(B),\ T_0^2 = \{i_0, j_0, l_0\}(C) \\ &level\ 1: T_1^0 = \{i_1, l_1\}(T_0^0, T_0^1),\ T_1^1 = \{i_1, j_1, l_1\}(T_0^2) \\ &level\ 2: T_2^0 = \{i_2, j_2, l_2\}(T_1^0, T_1^1) \end{aligned} \tag{3.2}$$

块间资源映射:	**块内资源映射:**
$\mathbf{Pipe}(T_0^0, T_0^1),\ \mathbf{Shar}(T_1^0, T_1^1)$	$\mathbf{Sp}(i_2),\ \mathbf{Sp}(i_1),\ \mathbf{Sp}(i_0)$

(3.3)

这个数据流所表达的计算顺序与图 3.3部分 c) 中所展示的树结构一致。对于块间资源映射，T_0^0, T_0^1 通过 *Shar* 原语构成了流水线。对于块内资源映射，循环 i_0, i_1, i_2 被映射到空间并行单元上，其余循环都映射到不同时间步骤上，实现这种映射的原语 *Sp* 将在后文介绍。使用者也可以不明确写出所有块间的和块内的资源映射，在本文不显式指出

时，块间的映射默认原语是 *Seq*，块内的映射默认是 *Tp*，这些原语的具体含义将在后文介绍。

3.3.2 融合数据流设计的三维空间

本文将融合数据流的设计空间描述为一个由三个维度组成的 3D 空间：计算顺序、资源映射和循环分块。计算顺序是选择给定融合的图结构中所有算子的正确执行顺序。资源映射是为每个块分配计算和内存资源。循环分块涉及选择要分块的循环，并找到好的分块参数，实现在硬件资源限制条件下最大化性能。在图 3.3的部分 a) 中展示了一个带有三个算子的计算图例子。在部分 b) 中展示了这个例子的融合数据流 3D 设计空间。

首先，需要解释计算顺序这一维度的细节。这一维度中的每个数据流设计点是一个顺序树，表明了算子分块后的执行顺序。选择树结构的原因有两个方面。一方面，树结构适合于分块，因为分块后外部循环可以形成根节点，而内部循环可以形成子节点。另一方面，树结构可以自然地捕获一个算子的迭代空间（多面体）插入到另一个算子的迭代空间（也是一个多面体）中，这恰好对应于算子的融合。树中的每个节点代表一个计算块。当一个算子 Op_1 被融合到另一个算子 Op_2 中时，Op_1 的迭代空间被插入到 Op_2 的迭代空间中，这被建模为在树结构中将一个子树的根节点（代表 Op_1）作为子节点插入到另一个树（代表 Op_2）中。在图 3.3的部分 c) 中，本文放大了一个顺序树选择并详细展示了其结构。这个树结构表达的是算子 *A* 被融合到算子 *B* 且融合发生的存储层次是 *L*1，随后，这两个算子都被融合到算子 *C*（每个块的颜色指示该块属于哪个算子），且融合发生的存储层次是 *L*2。这个树表达的执行顺序是：在 *L*1 内存中，首先计算 *A* 的一个块，然后计算 *B* 的一个块；重复第一步，直到 *B* 在 *L*2 的块计算完成；此后，使用来自 *L*2 的 *B* 的数据块来在 *L*2 中计算 *C* 的一个块；最后，重复上述步骤直到 *C* 完全计算完成。

除了块的顺序外，一个块内的循环顺序也需要精心设计，因为循环顺序影响融合的粒度。具体来说，当融合两个算子（将 Op_1 融合到 Op_2）时，只有 Op_2 的归约循环被允许在融合后的顺序树的父节点块中（如图 3.3的部分 c) 所示）。否则，如果 Op_1 的归约循环出现在父节点块中（作为公共外部循环），将出现大量的冗余计算，融合数据流的性能会很差。

接下来，本文解释资源映射的细节。这一维度中的每个设计点是资源划分或共享的选择，本文通过原语来控制资源映射。对于顺序树中一个节点内的计算，本文设计的原语被称为块内原语；对于不同节点之间的计算的映射，本文也相应设计有块间原语。本文在表 3.3中展示了这两种类型的原语的含义。*Sp* 和 *Tp* 这两个原语在基于多面体的

表 3.3　TileFlow 中的资源映射原语设计

原语名称	记号	解释	例子
块内原语			
Spatial	**Sp**	把迭代实例映射到空间上不同的单元	**Sp**(loops)
Temporal	**Tp**	把迭代实例映射到同一单元的不同的时间步骤	**Tp**(loops)
块间原语			
Sequential	**Seq**	每个块的执行按顺序进行 且对硬件所有资源占用互斥	**Seq**(T_1, ..., T_M)
Sharing	**Shar**	每个块执行按顺序进行 且仅共享硬件存储资源	**Shar**(T_1, ..., T_M)
Parallel	**Para**	所有块并行执行 且共享硬件所有资源	**Para**(T_1, ..., T_M)
Pipeline	**Pipe**	所有块流水线并行执行 且共享硬件所有资源	**Pipe**(T_1, ... , T_M)

模型[32,34-35]中使用较广，相关论述比较成熟。块间原语则是本文主要创新提出的内容。在图 3.3的部分 d) 中，本文利用图例解释块间原语的作用。*Seq* 将每个计算块绑定到相同的硬件资源上，在不同的执行步骤中这些块单独占使用资源（没有任何共享）。*Shar* 也将每个块绑定到相同的计算资源上，在不同的执行步骤中不共享计算资源，但共享内存资源。*Para* 将每个块绑定到不同的计算和内存资源部分，在同一时刻实现硬件的计算和存储的空间共享。*Pipe* 也空间共享硬件资源，但是块之间是以流水线方式执行的。

这些原语的使用中，存在着延迟和资源使用之间的复杂权衡问题。根据张量计算具体的输入形状，对于不同的计算块，需要使用不同的原语组合。*Pipe* 可以减少延迟并提高吞吐量，但代价是较高的计算和内存资源的使用。但是对于输入输出张量形状较大的块，如果所需数据不能在片上存储中完整地存储，*Pipe* 可能不会带来性能上的收益。*Para* 与 *Pipe* 类似，但只适用于没有数据依赖的块。相比之下，*Seq* 不会降低延迟，但它能节省片上计算和内存资源，因为一次只能执行一个块，整体资源使用的峰值取决于资源消耗最大的块，而不是所有块的总和。*Shar* 与 *Seq* 类似，但允许更多数据在片上存储，这是以内存使用为代价来优化数据局部性。

最后，本文简要解释循环分块。循环分块是软硬件映射设计和性能调优中最常用的优化技术[38,89,126]。一个循环分块的选择由两部分组成：选择要分块的循环，以及选择分块参数。如图 3.3的部分 e) 所示，选择要分块的循环影响计算的粒度。选择更多循环进行分块会产生细粒度的数据流，而选择较少的循环进行分块则会产生粗粒度的数据流。选择分块参数这一问题则是在硬件资源限制下求解一次计算多少输出、加载多少输入数据，才能最大化性能，往往最好的性能是通过平衡数据加载延迟和计算延迟来实现。

在对张量计算进行分块的时候，本文使用向量 $\vec{S} = (s_1, s_2, ..., s_I)$（共有 I 个参数）

来表示所有的分块参数。例如，当本文分解一个矩阵乘法链时，将使用 (T_M, T_N, T_K, T_L) 代表分块参数，因为矩阵乘法链有四个维度需要分解；对于卷积链，将有多达十个维度。所以寻找最优分块策略问题，就是选择能最大化整体性能的最优分块参数 $\vec{S}$ 的问题。先前的工作[127]提出独立计算这些参数以平衡不同块的计算开销。但本文认为，分解参数不能独立选择，因为它们与块执行顺序、资源映射选择等共同影响整体数据重用。

3.4 基于块的分析和优化

本章节，本文将介绍如何在融合数据流中进行分析和优化。由于本文使用块抽象表达融合数据流，所有的分析和优化将针对块进行。对于块的分析，其核心在于分析数据在片上搬移的开销。由于本文关注的问题是在性能受限于访存时如何进行融合优化，所以对于计算延迟本身的估计不是重点，整体执行延迟的瓶颈在于访存。分析的直接作用是可以为后续优化提供性能模型，用以估计在融合数据流设计空间中不同选择对性能的影响，从而进行后续自动优化。本文的分析部分分为两类：对于算子链结构的分析法和对于任意拓扑结构的通用分析法。对于自动优化部分，本文根据分析方法的不同提供两种选择，一种是基于分析结果使用确定性优化算法计算近似最优解，这种方法适用于对于算子链结构的分析法。另一种是通过自动调优技术进行黑盒式调优，可以在其中引入多种机器学习算法辅助进行搜索，这种方法适用于任意拓扑结构的融合优化。

3.4.1 算子链数据搬移量分析

3.4.1.1 块间的分析方法

本文在块间优化中的目标是找到最优的块执行顺序和分块参数 $\vec{S}$，以最小化总数据搬移量。最小化数据搬移量等同于最大化数据局部性（或重用）。如图 3.1中所示的，来自不同算子的计算块可以被重新排序，以获得更好的数据重用。为了简化，本文假设输入的算子链中有两个计算密集型算子。对于更多的计算密集型算子，分析方法保持相似。注意，本文的融合方法和访存密集型算子融合互补，并不排斥。本文对于融合中出现的访存密集型算子的数量和类别也没有限制。对于访存密集型算子，融合优化方法可以参考[37,74]，在本文中就不多加讨论了。

相比任意拓扑，链状拓扑的分析较为容易，这部分实现在 Chimera 的块间分析和优化部分。本文假设在被融合的算子链中，第一个计算密集型算子中有 P 个循环，第二个计算密集型算子中有 Q 个循环。这些循环的不同排序表示不同的块执行顺序，如第 3.1.4.1节图 3.1中的例子所示。总的来说，整个设计空间由这些循环的所有 $(P+Q)!$

```
input  : 算子链 Ops
input  : 循环顺序 Perm = (l_{p_1}, l_{p_2}, ..., l_{p_I})
input  : 分块参数 S⃗ = (s_1, s_2, ..., s_I)
output : 数据搬移量 DV
output : 内存使用量 MU
DV = 0; MU = 0;
for op ∈ Ops do
    total_DF = 0;
    for tensor T ∈ op.allTensors() do
        DF = getFootprint(T, S⃗);
        total_DF += DF;
        if T ∈ Ops.IOTensors() then
            DM = DF;
            keep_reuse = true;
            for loop l_{p_i} ∈ reversed(Perm) do
                if l_{p_i} ∈ op.allLoops() then
                    if l_{p_i} accesses tensor T then
                        keep_reuse = false;
                    end
                    if not keep_reuse then
                        DM *= ⌈L_{p_i} / s_{p_i}⌉
                    end
                end
            end
            DV += DM;
        end
    end
    for loop l_{p_i} ∈ Perm do
        if l_{p_i} is private to op then
            Perm.erase(l_{p_i});
        end
    end
    MU = max(MU, total_DF);
end
return DV, MU;
```

Algorithm 1: 对于算子链融合的数据搬移量分析和内存使用估计算法

种不同排列组成。但是实际有效的设计空间大小可能远小于 $(P+Q)!$，因为两个计算密集型算子可能共享一些共有循环，共有循环的排序对数据重用没有影响。在图 3.1中矩阵乘法链的示例中，有 24 种不同的重排序选择，而不是 $(3+3)! = 720$。这是因为两个矩阵乘法有两个共有维度 m 和 l，只有四个独立循环（m, n, k, l）。所以设计空间大小是 4!。在下文中，本文只考虑有 I 个（$I \leq P+Q$，对应于 $\vec{S}$ 中参数的数量）独立循环 $(l_1, l_2, ..., l_I)$，实际设计空间大小是 $I!$。循环 l_i 的原始循环长度表示为 L_i。这些循环的一个排列表示为 $(l_{p_1}, l_{p_2}, ..., l_{p_I})$，其中 $(p_1, p_2, ..., p_I)$ 是 $(1, 2, 3, .., I)$ 的一个排列。块中的嵌套循环执行时，从最右侧（最内层）的循环执行到最左侧（最外层）的循环。

找到最优块执行顺序（即循环排列选择）的主要思想是针对每个排列选择，用分块参数 $\vec{S}$ 表达数据搬移量。这样就可以进一步通过找到最优的 $\vec{S}$ 来最小化数据搬移量，并得到在所有的排序可能中给出最小数据搬移量所对应的最优顺序选择。

一般来说，每个张量的数据搬移量是张量在内存中的占用空间与其访存语句使用

次数的乘积。此外，对数据搬移本文还有三个论断。首先，本文认为某些循环不会引起任何数据搬移，原因是它们的迭代变量没有在张量访问下标中使用；其次，一旦一个循环造成数据搬移发生，那么它的外层循环也必然引起数据搬移；第三，只出现在生产者算子中的循环（或者称为私有循环）不会在消费者算子中引起数据搬移。本文使用图 3.1中的矩阵乘法链示例来解释这些论断。在 $mknl$ 顺序下，循环 n,l 不会引起矩阵 A 的数据搬移，因为它们的循环变量没有被用来访问矩阵 A；在 $mnlk$ 顺序下，循环 n,l 会引起矩阵 A 的数据搬移，因为内部循环 k 已经引起了数据搬移；在任何块顺序下，循环 k 不会引起矩阵 D,E 的数据搬移，因为 k 是第一个矩阵乘法的私有归约循环，对第二个矩阵乘法没有影响。

本文用算法 1计算给定排列选择情况下的数据搬移量。对于目标算子链 Ops，对于给定的排列顺序 $(l_{p_1}, l_{p_2}, ..., l_{p_I})$，算法首先根据拓扑顺序遍历算子链（从生产者到消费者，在第 2 行）。算法只考虑整个算子链的输入/输出张量（由函数 $IOTensors$ 返回）（第 7 行），因为中间结果都存储在片上内存中。本文使用 $getFootprint$ 函数根据分块参数 $\vec{S}$ 计算每个张量的数据块占用空间（DF，第 5 行）。为了计算数据搬移量，本文需要进一步弄清楚执行过程中数据块被替换的次数。

由于只有被用来访存张量的循环会导致数据块发生替换，本文使用标志 $keep\_reuse$ 来检查当前循环 l_{p_i} 是否会导致数据替换。当替换发生的时候，算法就通过乘以循环的长度来更新当前张量 T 的数据搬移量。对于所有其他外层循环，如果内层循环已经导致了替换，$keep\_reuse$ 会一直为真，算法就会将外层的循环长度也乘进来。在算法继续运行到下一个算子前（消费者算子），需要排除生产者算子的私有循环的影响（见第 17-19 行），这样的私有循环就不会在消费者算子的数据搬移量上产生影响。最后，算法返回最大内存使用量 MU 和数据搬移量 DV，MU 这个量对后续求解优化问题起到约束作用。

3.4.1.2 块内的代码生成

本节介绍 Chimera 中的块内优化。不同的 AI 芯片硬件需要不同的优化才能实现高性能。Chimera 利用可替换的微内核来处理硬件多样性。

不同 AI 芯片的编程模型和优化方法各不相同。例如，为了实现矩阵乘法的高性能微内核，在 CPU 上，开发者需要编程汇编以使用 SIMD 单元；在 GPU 上，开发者需要使用 Tensor Core 指令（或称为 intrinsic）来将计算映射到 Tensor Core 单元；在 NPU 上，开发者需要添加程序标注（pragma）到循环中以指导底层编译器生成加速器指令。为了通过统一的方法处理硬件多样性，Chimera 使用可替换的微内核，这种可以换微内核对不同的硬件后端是可扩展的。

可替换的微内核是对计算块的一个抽象，在块所表达的嵌套循环基础上，进一步

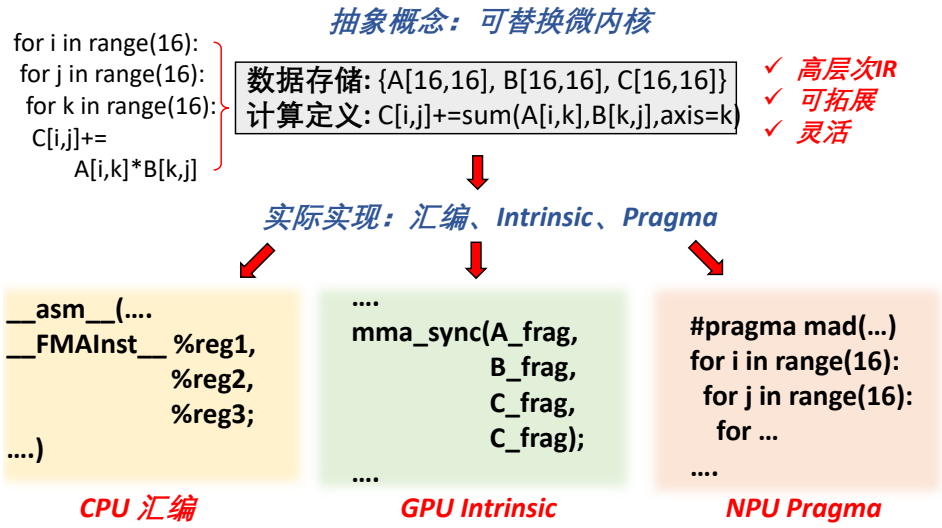


图 3.4　可替换微内核概念示意图

增加了对输入输出张量的类型和形状限制，也增加了对块内计算语义的限制。对于不同的芯片，可替换的微内核可以被底层硬件特定的实现替换，无论它是汇编、intrinsic还是 pragma。在 Chimera 中注册了不同的硬件的特定微内核，这些微内核在同一个可替换的接口下执行相同的计算功能（尽管使用不同的设备指令）。在编译和代码生成期间，Chimera 将根据目标硬件将可替换的微内核翻译为相应的注册的底层微内核实现代码。本文使用图 3.4中的一个示例来详细解释可替换的微内核。这个示例使用一个可替换的微内核来描述一个 16×16 的矩阵乘法，这个计算语义可以用高层次表达式概括，同时标明了输入输出张量的格式和形状，然后，所示例子向这个可替换的微内核注册三个不同的底层微内核实现。这三个具体实现仍然是通过底层代码编写的（例如，CPU 的微内核大约需要 140 行汇编），注册过程可以通过 Chimera 的 Python 接口，这样注册比较方便易更改。在后续编译和代码生成期间，编译器根据目标设备自动选择三种不同的实现中的一个并注射到生成的底层代码中。微内核的实现与算子紧密耦合。由于大部分计算都可以划归到矩阵乘法，本文就主要介绍矩阵乘法微内核，这些微内核可以被各种计算密集型算子复用，包括矩阵乘法、批量矩阵乘法和卷积。

CPU 微内核实现：在算法2中本文展示 CPU 微内核的伪代码。本文采用了与先前工作[144-145]类似的外积方法实现。微内核通过提供足够的并行计算来隐藏寄存器加载/存储的延迟，并通过连续发射 MI × NI 条 FMA 指令（MI × NI 是流水线深度）来保持 FMA 流水线充分运转。决定微内核的参数（MI, NI, MII, KI）的方法是在可用寄存器数量的约束下最大化计算密度（AI）。

$$
\begin{aligned}
\max_{MI,NI,MII} \quad & AI = \#ComputeInst/\#LoadStoreInst \\
\text{s.t.} \quad & RegUsed \leq \#Registers \\
\text{where} \quad & \#ComputeInst = MI \times NI \times KI \\
& \#LoadStoreInst = KI \times (MI + NI) + 2MI \times NI \\
& RegUsed = MI \times NI + NI + MII
\end{aligned}
\tag{3.4}
$$

例如，对于拥有 32 个 ZMM 寄存器的 Cascadelake 微架构 CPU，计算得到 MI, NI, MII 分别为 6, 4, 2，KI 的大小是根据问题规模动态设置的，这样计算的流水线深度是 24，可以最大化 AI。在代码生成期间，将根据算法 2所示模板和计算出的参数（MI, NI, MII, KI）生成汇编代码。

```
constant    : RegLen # the vector register length.
parameter   : MI, NI, MII, KI
input       : A[MI, KI], B[KI, NI*RegLen]
input/output: C[MI, NI*RegLen]
register    : RegA[MII], RegB[NI], RegC[MI, NI]
for m in [0, MI, 1) do
    for n in [0, NI, 1) do
        vecLoad(C[m,n*RegLen: (n+1)*RegLen], RegC[m,n])
    end
end
for k in [0, KI, 1) do
    for n in [0, NI, 1) do
        vecLoad(B[k,n*RegLen: (n+1)*RegLen], RegB[n])
    end
    for mo in [0,MI, MII) do
        for mi in [0, MII, 1) do
            vecLoad(A[mo+mi,k], RegA[mi])
        end
        for mi in [0, MII, 1) do
            for n in [0, NI, 1) do
                FMA(RegC[mo+mi,n], RegA[mi], RegB[n])
            end
        end
    end
end
for m in [0, MI, 1) do
    for n in [0, NI, 1) do
        vecStore(C[m,n*RegLen: (n+1)*RegLen], RegC[m,n])
    end
end
```

Algorithm 2: CPU 微内核设计算法

GPU 微内核实现：在 Tensor Core GPU 上，可以使用 WMMA 指令集的 *mma_sync* intrinsic 一次计算一个 $16 \times 16 \times 16$ 的矩阵乘法。然而，直接使用这个 intrinsic 并不高

效，因为每个 *mma_sync* intrinsic 需要对应一个矩阵的加载和存储操作。因此，它的计算密度很低，执行性能受限于内存操作。为了提高计算密度，可以将 GPU 的微内核循环展开，并调整 intrinsic 的顺序来形成外积计算，从而增加数据复用。具体来说，微内核一次加载两个 16×16 的输入矩阵，但是要更新 2×2 块的结果矩阵，每个结果矩阵大小都是 16×16。在这个实现中，每个加载的矩阵块被重用两次，从而提高了整体的计算密度。

NPU 微内核实现：华为 Ascend NPU 使用带有 pragma 的 Python DSL（TBE）实现微内核。被 pragma 标记的计算部分将被底层编译器（CCEC）映射到 NPU 的矩阵乘法单元，这部分编译器由相应的工具链 *CNCC*[140]提供。本文主要使用的 pragma 是 *mad*，它可以标记六个连续循环，其中三个外层的分块矩阵乘，和三个内层的核心矩阵乘法：

$$\begin{aligned} &C[m1,n1,m2,n2]+=A[m1,k1,m2,k2]*B[k1,n1,n2,k2] \\ &(m1\le M1,m2\le M2,n1\le N1,n2\le N2,k1\le K1,k2\le K2) \end{aligned} \tag{3.5}$$

为了产生预期的嵌套循环和循环顺序，还需要使用 DMA 指令将输入矩阵打包加载到片上内存中，并产生连续的数据块。这个微内核的整体计算密度是

$$AI=\frac{M1\times M2\times N1\times N2}{M1\times M2+N1\times N2} \tag{3.6}$$

最大化计算密度的结果就是得到分块参数为

$$M2=N2=Lane\_of\_cube\_units \tag{3.7}$$

$Lane\_of\_cube\_units$ 在半精度下是 16。根据 L0 内存的具体大小，可以进一步算出 $M1=N1$ 的值，保证不会超出内存限制。

3.4.2　通用数据搬移量分析

上一个小节介绍了对算子链的数据搬移量分析，这一小节本文进一步介绍通用拓扑下的数据搬移量分析。链状拓扑的分析能直接得到数据搬移量的表达公式，从而完成求解，但是对于更复杂的拓扑，列出公式比较复杂，本文选择使用性能模型加搜索的方法解决通用拓扑问题，其中性能模型实现在 TileFlow 的基于树结构的分析部分。

不同的融合数据流具有不同的计算顺序、资源映射和分块决策，可能会导致不同的数据重用，因此数据搬移量也会不同。为了计算数据搬移量，本文从一个简单的情景开始：分析树中只有一个具有完美嵌套循环结构的块。然后，进一步考虑有两个及以上块的更复杂情况。

3.4.2.1 单个块内的数据流分析

分析树中的单个块节点（没有任何子节点）代表一个完美的循环嵌套（也可以看成多面体）。在块中有两种类型的循环：并行循环和串行循环。对于块中的每个循环迭代步骤 t，对于每个张量 Z，程序都会读取或更新一片数据（张量的一个切片）。本文表示为

$$Slice_Z^t = Z[b_0^t : e_0^t,\ b_1^t : e_1^t,\ ...,\ b_{D-1}^t : e_{D-1}^t] \tag{3.8}$$

其中 b_i^t 是张量在维度 i 的开始地址，而 e_i^t 是张量在维度 i 的结束地址（但是不包括该位置）。对于每个维度 i，虽然开始地址 b_i^t 和结束地址 e_i^t 随着不同的时间步骤 t 而变化，但片段的范围（$e_i^t - b_i^t$）保持不变，并且由维度 i 处的并行循环确定。例如，本文使用图 3.5中的一维批处理卷积来解释数据片段的读取和存放。对于这个示例中的张量 A，在不同的执行时间步骤中，虽然使用了 A 的不同子矩阵进行计算（地址起点终点不同），但所有子矩阵的大小都是 4×6（片段范围固定）。

单个块的数据搬移发生在相邻的时间步骤之间（随着串行循环的进行），主要体现是下一步计算需要新的数据块而旧的数据块不再需要，这就需要通过数据搬移加载新数据。用符号表达，对于每个时间步骤 t，对于张量 Z，数据搬移量是

$$DM_Z^t = |Slice_Z^t - Slice_Z^{t-1}| \tag{3.9}$$

注意，$Slice_Z^t$ 是一个数据集合，所以 $Slice_Z^t - Slice_Z^{t-1}$ 对应的是集合差集运算，其结果是时间步骤 t 所需但在时间步骤 $t-1$ 中还没加载的数据集合。$|.|$ 是范数运算，表达集合中元素的数量，所以 DM_Z^t 是一个非负整数值。总数据搬移量 DM_Z 是

$$DM_Z = \sum_{t_i \in Bounds} U_i \tag{3.10}$$

$$\text{where} \quad U_i = \begin{cases} |l_i| \times (DM_Z^{t_i} + (|l_{i-1}| - 1) \times U_{i-1}),\ i > 1 \\ DM_Z^{t_0},\ \ i = 0 \end{cases} \tag{3.11}$$

其中 $Bounds$ 是时间步骤的边界 t_i（$0 \le i \le Dim$，Dim 是串行循环的数量）的集合。时间步骤边界表示当一个串行循环迭代达到其最大值，如果这个循环外还有循环，外层循环将前进一步，当前循环将返回到其下界值，准备重新开始迭代。在图 3.5中，本文展示了从时间步骤 $[0,0]$ 到时间步骤 $[0,1]$ 每个张量的数据搬移量。本文使用向量来表示时间步骤，这种向量表示法可以方便本文将时间步骤向量中的每个值与串行循环相关联（在这个例子中，本文有两个串行循环 i_1, j_1，所以时间向量是二元的）。张量 A

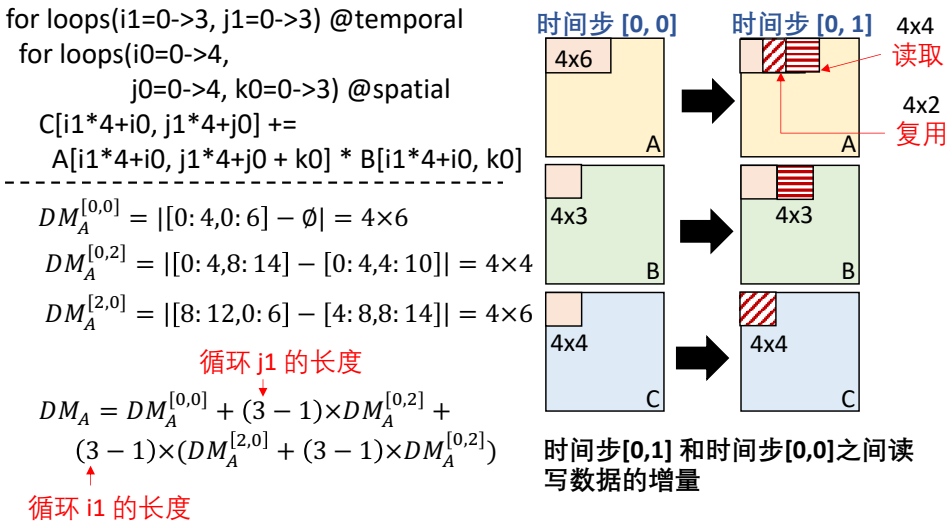


图 3.5　块内数据搬移分析例子

需要 4×4 个新元素进行下一步计算，并重用了 4×2 个来自上一步的元素，张量 $\boldsymbol{B}$ 需要 4×3 个元素，而张量 $\boldsymbol{C}$ 完全被重用，不需要数据搬移。图 3.5展示了时间步骤边界例子（t_0 =[0, 0]，t_1 =[0, 2]，和 t_2 =[2, 0]）。注意，时间步骤 [0, 0] 也是一个边界，它表示整个程序开始执行的时候的初始点。在这个例子中，张量 $\boldsymbol{A}$ 的总数据搬移量是 168 个元素。

3.4.2.2　多个块之间数据搬移分析

对于具有复杂层次结构的分析树，一个块将带有若干子块。为了计算父块的数据搬移量，可以推广单个块的分析方法，通过计算两个相邻时间步骤之间所有子块的数据块差集来计算总的数据搬移量。在给定步骤 t 中，对于每个子块，首先计算其所有输入/输出张量的数据块，然后根据父节点中所有子块的顺序排列子块，排序后，对于每两个相邻的块，计算他们的访存集合差集以得到从一个块切换到另一个块时需要多少数据搬移。通过累加所有的集合差集，最终将得到父块视角下的的数据搬移量。图 3.6（上半部分）使用一个简单的例子来展示这种方法，其中父块（块 0）只有两个串行循环。根据时间步骤列出子块（块 1 和块 2）的执行序列，对于每两个相邻的执行步骤，可以计算数据块的差集，最终的数据搬移量是所有差集的大小的和。在具体算法实现中，不需要完全展开所有时间步骤，因为对于 DNN 中的张量算子，计算和访存是规律的，这就使得算法可以只需要考虑时间步骤边界，根据循环长度推测总数据搬移量，这与单个块分析类似。

不同的块间资源映射决策可能会影响数据搬移量。对于 *Shar*、*Para* 和 *Pipe*，数据搬移量的计算方法与上述相同。但对于 *Seq*，在一个块执行后，其访问的数据块将被清除，

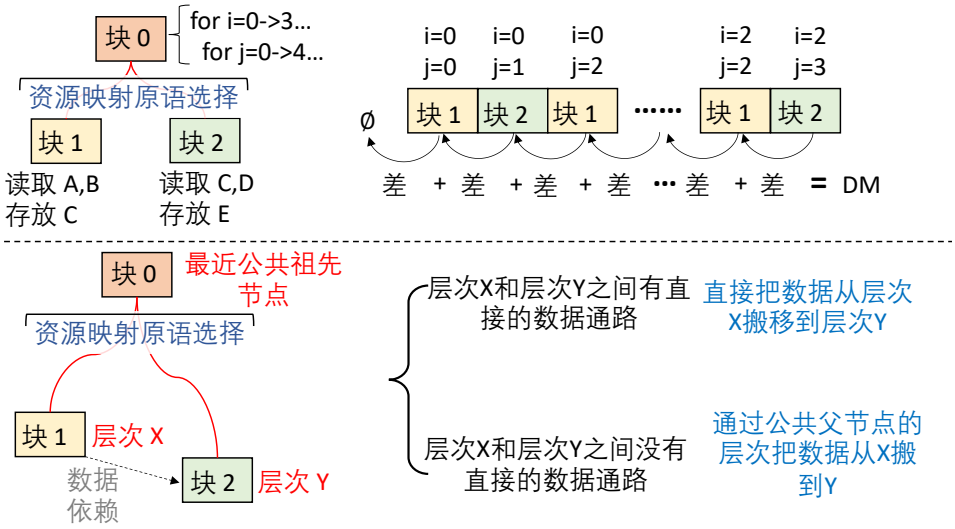


图 3.6　块间数据搬移量分析示意图

除非它们被后续的块所需要，这将在执行期间增加数据搬移总量。本文在 TileFlow 中通过清除前一块使用的张量的数据块来模拟这一点，这些张量不被后续的块使用，因此总体的数据搬移量也会偏高。

在之前的分析中，讨论的数据搬移发生在一个内存层级内。但是分析树中的两个块也可能位于不同的内存层级。对于这种情况，就需要计算不同内存层级之间的数据搬移量。在图 3.6的下半部分，展示了两个不同层级的块（X 层级的块 1 和 Y 层级的块 2）。为了应对这种情况，首先需要分别分析它们的数据搬移量，以得到块 2 需要从块 1 获取多少数据。然后根据具体芯片架构设计，有两种不同的情况。如果 X 层级的内存不能直接将数据移动到 Y 层级的内存（这在 AI 芯片中很常见[13]），那么就需要通过它们的最近公共祖先层次（块 0 所在层级）来移动数据。否则，可以直接在块 1 和块 2 之间移动数据，并记录 X 层级和 Y 层级内存之间的数据搬移量。这种差异在 TileFlow 中也被建模，相应的数据搬移量被记录到发生搬移的内存层级。

3.4.2.3　预测硬件资源使用量

除了利用分析树计算数据搬移量，还可以预测硬件资源使用情况（内存和计算资源）。对于分析树中的每个块 T_n，如果它最多只有一个子块（那么它是一个完美的循环嵌套），它使用的处理单元（PE）数量和内存占用量可以通过基于多面体技术的分析[32,34-35]来计算。如果它有两个及以上的子块，对于每对子块 T^1_{n-1}, T^2_{n-1}，父块所需要的 PE 数量取决于它的子块间资源映射原语的选择。本文列出计算资源预测的计算公式

表 3.4　*mlkn* 顺序下的矩阵乘法链的数据搬移量分析结果

	A	**B**	**C**	**D**	**E**
DM	$MK\lceil\frac{L}{T_L}\rceil$	$KL\lceil\frac{M}{T_M}\rceil$	0	$NL\lceil\frac{M}{T_M}\rceil$	$MN\lceil\frac{L}{T_L}\rceil$
DF	T_MT_K	T_KT_L	T_MT_L	T_LT_N	T_MT_N

$$NumPE(T_n)=\begin{cases}max(NumPE(T^1_{n-1}),NumPE(T^2_{n-1})),\ Seq\ 和\ Shar\\ NumPE(T^1_{n-1})+NumPE(T^2_{n-1}),\ 其他\end{cases} \tag{3.12}$$

内存资源的使用量计算公式则是

$$FootPrint(T_n)=\begin{cases}max(Footprint(T^1_{n-1}),Footprint(T^2_{n-1})),\ Seq\\ Footprint(T^1_{n-1})+Footprint(T^2_{n-1}),\ 其他\end{cases} \tag{3.13}$$

3.4.3　针对链式结构融合的分析性优化方法

本节将介绍如何基于分析结果优化融合数据流。首先介绍的是 Chimera 中实现的针对算子链的分析性优化方法，正如先前介绍，这种优化方法仅适用于算子链这类线性结构，但是它非常高效，仅需要少量的数学运算就可以求出近似最优解，这个优化方法作为 Chimera 中的默认优化方法，针对 CPU，GPU，NPU 三种平台都可以使用。后续小节将继续介绍更通用的优化方法。

分析性优化方法的优化目标是数据搬移量本身，其基本思想是认为数据搬移量和性能正相关，因此最小化数据搬移量就是最小化代码执行的延迟。首先，仅考虑一层片上内存的情形，基于前述章节对于算子链结构的数据搬移量分析，可以获得数据搬移量 DV 和内存使用量 MU，最小化数据搬移量的优化问题则可以定义如下：

$$min_{\vec{S}}\ \text{DV},\quad \text{s.t. MU} \le MemoryCapacity \tag{3.14}$$

为了求解这个约束优化问题，首先在实数域（$\mathscr{R}$）中解方程 3.14，然后通过向下取整获得近似整数解。具体而言，可以使用拉格朗日乘数法得到 DV 的极值和相应的极点 $\vec{S}^*$。然后，通过对 $\vec{S}^*$ 进行向下取整得到近似整数候选解。最后，选择使 DV 最小化的整数候选解作为最终解。

本文使用图 3.1中的矩阵乘法链示例来进一步阐述优化问题。这里使用图 3.1中的执行顺序 *mlkn*（在第 6 行）进行演示。通过使用算法 1，可以得到矩阵 A,B,C,D,E 的数据搬移量和占用空间，如表3.4所示（在此示例中，分块参数是 $\vec{S}=(T_M,T_N,T_K,T_L)$）。DM 代表数据搬移量，DF 代表每个张量的数据占用空间。C 的 DM 是 0，因为它是一

个中间结果，并且总是在片上内存中重用。因此，矩阵乘法链的总数据搬移量是

$$\begin{aligned}\mathrm{DV}_{\text{GEMM Chain}} &= DM_A + DM_B + DM_C + DM_D + DM_E \\ &= MK\lceil\frac{L}{T_L}\rceil + KL\lceil\frac{M}{T_M}\rceil + NL\lceil\frac{M}{T_M}\rceil + MN\lceil\frac{L}{T_L}\rceil\end{aligned} \tag{3.15}$$

所有张量的峰值内存占用量是 MU

$$\begin{aligned}\mathrm{MU} &= max\{\mathrm{GEMM1}_{MU}, \mathrm{GEMM2}_{MU}\} \\ \mathrm{GEMM1}_{MU} &= DF_A + DF_B + DF_C = T_M T_K + T_K T_L + T_M T_L \\ \mathrm{GEMM2}_{MU} &= DF_C + DF_D + DF_E = T_M T_L + T_L T_N + T_M T_N\end{aligned} \tag{3.16}$$

为了在不超过内存容量限制的情况下最小化总数据搬移量，优化问题如下表述：

$$\textbf{min}\ \ \mathrm{DV}_{\text{GEMM Chain}} \quad \text{s.t. } \mathrm{MU} \le MemoryCapacity \tag{3.17}$$

通过使用拉格朗日乘子法，可以得到最小点和最小数据搬移量：

$$DV^* = \frac{2ML(K+N)}{T_M^*},\ T_M^* = T_L^* = -\alpha + \sqrt{\alpha^2 + MC}, T_N^* = \alpha \tag{3.18}$$

MC 是 $MemoryCapacity$（内存容量）的缩写。α 是 T_N, T_K 的下界。本文设置下界是因为 T_N, T_K 在这个优化问题中是自由变量，其下届往往由分块最小可行值决定，如通用处理器上最小值是 1，对于使用特殊指令的情况，要为特殊指令预留一定大小，比如 Tensor Core 的下界是 16，这些细节在指令层编译时会再次提及。进一步，通过 $T_X = \min\{\lfloor T_X^* \rfloor, X\}$（$X \in \{M, N, K, L\}$）将实数值转换为整数得到近似解。通过一定的计算，可以估计近似解和最优解之间的差距。使用近似数据搬移量（DV_{app}）与最优值（DV^*）的比率来显示差异：

$$\begin{aligned}&\frac{DV_{app}}{DV^*} \le max_{X\in\{M,L\}}\{1 + \frac{T_X^*}{X} + \frac{1}{T_X}\} \le \\ &max_{X\in\{M,L\}}\{1 + \frac{\sqrt{MC}}{X} + \frac{1}{\min\{X, \sqrt{MC}\}}\},\ (MC >> \alpha)\end{aligned} \tag{3.19}$$

可以发现，这个差距是有界的。这也就说明，近似最优解和全局最优解相差不大。接下来，可以进一步考虑有多层片上内存的情形。对于多层片上内存的情形，分析树的不同层次的块可以分别对应不同层次的内存。每一个层次的块的执行顺序也影响对应层次性能，这就需要在建模的时候同时考虑多个层次的相互影响。假设芯片有 D 层片上内存。第 d 层的数据搬移量定义为 $DV_d(\vec{S}_d)$，其中 $\vec{S}_d$ 是第 d 层的分块参数。然后，

从第 $d+1$ 层到第 d 层的数据搬移成本 $Cost_d(\vec{S}_d)$ 按照如下公式计算

$$Cost_d(\vec{S}_d) = DV_d(\vec{S}_d)/bw_d \tag{3.20}$$

其中 bw_d 是内存带宽。为了最小化整体数据搬移成本，需要在所有内存层次中最小化最慢的数据搬移阶段。因此，可以如下表述优化问题

$$\begin{aligned} min_{\vec{S}_1,\vec{S}_2,...,\vec{S}_D} \ & \{max\{Cost_1(\vec{S}_1),...,Cost_D(\vec{S}_D)\}\}, \\ \text{s.t. } & \mathrm{MU}_1 \le MC_1,...,\mathrm{MU}_D \le MC_D \end{aligned} \tag{3.21}$$

MC_d 是第 d 层内存的 $MemoryCapacity$（内存容量）；MU_d 是第 d 层内存的内存使用量。通过优化这个目标函数来决定每一层内存的最优块分解参数和执行顺序，求解方法和前述单层类似，通过先在实数域求解，然后近似到整数解。

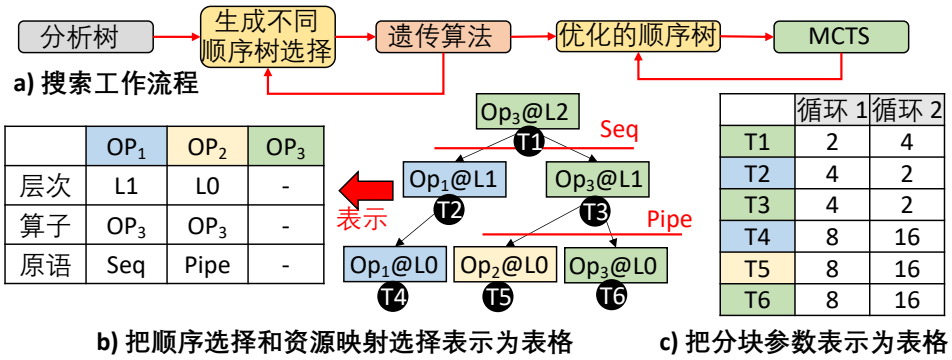
3.5　融合数据流的通用优化方法

对于一般性的拓扑和融合数据流，可以通过本节介绍的方法进行优化。首先，优化目标不是数据搬移量本身，而是优化延迟（周期数）或者功耗本身。TileFlow 可以通过数据搬移量以及其他分析结果（资源使用量、硬件参数）等估计数据流执行延迟。利用估计的结果，可以进一步使用机器学习算法在整个设计空间搜索优化的数据流，这种黑盒式优化具有通用性的优点，对于各种不同架构的 AI 芯片都可以进行支持。

3.5.1　延迟和功耗估计方法

这部分功能实现在 TileFlow 中，因此 TileFlow 可以作为一个性能模型与 Chimera 对接，如图 3.2中所展示的那样。TileFlow 计算延迟和能量时，需要硬件架构参数 $Spec$ 作为参考数据。对于每个位于第 n 层的块 T_n，它有三个执行阶段：数据加载、计算和数据存储。数据加载和存储量通过前面章节介绍的数据搬移量分析获得，数据加载/存储延迟通过将数据搬移量除以相应内存层级的带宽 BW_n（来自 $Spec$）来估算。延迟如下计算：

$$\mathrm{Lat}_{T_n} = \begin{cases} Perfect\_Tile\_Latency(T_n, Spec), & \text{如果} T_n \text{ 没有子块} \\ Max\{\dfrac{DM_n^{load}}{BW_n}, \sum\limits_i \{Lat_{T_{n-1}^i}\}, \dfrac{DM_n^{store}}{BW_n}\}, & Seq \text{ 或者 } Shar \\ Max\{\dfrac{DM_n^{load}}{BW_n}, \max\limits_i \{Lat_{T_{n-1}^i}\}, \dfrac{DM_n^{store}}{BW_n}\}, & \text{其他情况} \end{cases} \tag{3.22}$$



	OP1	OP2	OP3
层次	L1	L0	-
算子	OP3	OP3	-
原语	Seq	Pipe	-

	循环 1	循环 2
T1	2	4
T2	4	2
T3	4	2
T4	8	16
T5	8	16
T6	8	16

图 3.7　TileFlow 的搜索流程以及对不同设计维度的编码方式

表 3.5　批量矩阵乘法链的输入形状配置

代号名称	batch	M	N	K	L	来源网络
G1	8	512	64	64	512	Bert-Small (Bert-S)
G2	12	512	64	64	512	Bert-Base (Bert-B)
G3	16	512	64	64	512	Bert-Large (Bert-L)
G4	12	256	64	64	256	ViT-Base/14 (ViT/14-B)
G5	16	256	64	64	256	ViT-Large/14 (ViT/14-L)
G6	16	256	80	80	256	ViT-Huge/14 (ViT/14-H)
G7	12	208	64	64	208	ViT-Base/16 (ViT/16-B)
G8	16	208	64	64	208	ViT-Large/16 (ViT/16-L)
G9	16	208	80	80	208	ViT-Huge/16 (ViT/16-H)
G10	1	512	64	64	256	MLP-Mixer (MLP-M)
G11	1	768	64	64	384	MLP-Mixer (MLP-M)
G12	1	1024	64	64	512	MLP-Mixer (MLP-M)

其中 $Perfect\_Tile\_Latency$ 指的是使用基于多面体方法[32,34-35]来计算完美嵌套循环块的延迟。本文假设数据加载、执行和数据存储是完全流水线化的，硬件是带有双缓冲的。对于能量计算，本文使用现有的框架[32,146]的技术，通过先统计总的内存访问操作次数（这是在前面章节介绍的数据搬移分析的结果）和计算操作次数，然后查数据表得到不同操作的能量开销，最后把所有能量开销合计得到总能量开销。

3.5.2　基于机器学习的优化方法

考虑到数据流设计空间极为巨大，手动调整找到高效的融合数据流是很难的。TileFlow 中有一个基于机器学算法的搜索单元来帮助搜索数据流。搜索单元在搜索时使用遗传算法和蒙特卡洛树搜索（MCTS）[147]来寻找最优解。在图3.7的部分 a) 中展示了搜索单元的工作流程。其实也可以使用其他更复杂的探索算法[25-26,124,126,132,148]，不过这些不是本文讨论的重点，不再赘述。

搜索算法的工作流程如下：首先将不同的顺序树和资源映射原语编码到图3.7部分 b) 所示的表格中。每个算子对应一个列；行*层次*表示将中间数据存到哪一层内存；行

算子表示要融合到哪个算子；行原语表示资源映射原语。在图中展示的示例中，OP_1 在内存 *L1* 融合到 OP_3，OP_2 在内存 *L0* 融合到 OP_3，从而形成一个树结构。进一步利用遗传算法的杂交和变异操作就可以搜索好的编码选择，搜索的初始点则是从一组随机采样的编码开始组合生成新的分析树的编码。

另一方面，所有生成的分析树都传递给 MCTS 算法进行接下来的分块参数搜索，分块参数搜索需要在硬件资源限制内寻找最优化性能的分块参数。MCTS 是迭代工作的，在每一个迭代步骤，它都选择一个尚未确定如何分块的循环并在其循环长度范围内为其分配一个分块参数。然后，它使用已知参数更新约束条件（所有参数乘积小于等于循环长度，内存使用量不超过硬件限制等），并将新约束传递下去，并开始对下一个未分块的循环进行操作，至于具体选择哪个未分块循环，通过 MCTS 的内置参数决定，这种参数被称为上置信界（UCB）。在图3.7部分 c) 中，展示了一个编码分块参数的示例，这是一个包含两个循环的表格。MCTS 算法所使用的蒙特卡洛树中的每个搜索节点对应这样一个分块表，记录了每个循环在每个块中选择的分块参数。当 MCTS 达到叶节点（即分块表中的所有参数都被决定时），这就意味着算法已经得到了一个完整的融合数据流的树结构，然后就可以使用 TileFlow 模型进行性能预测评估了。评估的结果将反馈给 MCTS 算法以更新后续搜索的上置信界，从而影响后续搜索的流程。通过重复这样的搜索步骤数百次，带有优化后的分块参数的数据流树结构将被返回到遗传算法中作为遗传算法的反馈。遗传算法保留前 K 个（K 是一个参数）性能最好的分析树以产生下一代分析树。上述步骤重复数百次，就可以得到最佳融合数据流。

3.6　实验评估结果

为了验证 Chimera 和 TileFlow 的性能，本文实验分为两个部分。第一个部分，本文测试 Chimera 在针对算子链融合的时候实现的性能，使用的是 Chimera 中默认的优化方式。第二个部分，本文测试在有 TileFlow 作为性能模型的情况下，对于融合数据流的性能，使用的是 TileFlow 中的机器学习优化方法进行调优，特别地，本文测试一些非线性拓扑结构，如自动进行含有 softmax 算子的融合，也测试一些新架构（端侧和云测）的性能。

3.6.1　实验设置条件

本文将测试子图融合性能和整个网络融合的性能。使用的子图包括来自 Bert[87]、ViT[139]和 MLP-Mixer[149]的批量矩阵乘法链，以及卷积神经网络中的子图。比如从 SqueezeNet[150]和 Yolo[15,151]中选出的卷积链。对于整个网络评估性能，本文使用 Transformer[5]、Bert[87]和 ViT[139]。本文使用三种芯片：Intel Xeon Gold 6240 AVX-512 CPU

（1.125MB L1 缓存，18MB L2 缓存和 24.75MB L3 缓存）、Nvidia A100 Tensor Core GPU（高达 164KB/SM 共享内存，40.96MB L2 缓存）和华为 Ascend 910 NPU（64KB L0A/B 缓冲区，256KB L0C 缓冲区，1MB L1 缓冲区，256KB 统一缓冲区）。本文的对比对象包括手动调优的库和最先进的编译器。对于库，本文与 PyTorch[92]（在 CPU 上使用 MKL[95]和 oneDNN[86]，在 GPU 上使用 CuBlas[85]和 CuDNN[84]），TensorRT[36]和 CANN（NPU 的库）进行比较。对于编译器，本文与最近的编译器工作进行比较，包括 Relay[37]，Ansor[26]，TASO[18]，TVM+CUTLASS[75]和 AKG[31]（NPU 的编译器）。

3.6.2 Chimera 融合性能结果

3.6.2.1 子图性能结果

在本节中使用的子图包括批量矩阵乘法链和卷积链。本文已经在图 1.5中介绍了它们的基本结构。对于批量矩阵乘法链，本文评估了带有 softmax 作为中间内存密集型算子和没有任何中间算子两种情况的性能。对于卷积链，本文评估了使用 ReLU 作为中间算子和没有任何中间算子两种情况的性能。这些子图的输入形状配置如表 3.5和表 3.6所示。在表 3.5中，(batch, M, K)×(batch, K, L) 是第一个批量矩阵乘法大小。(batch, M, L)×(batch, L, N) 是第二个批量矩阵乘法大小。在表 3.6中，第一个卷积大小是 (batch, IC, H, W)×(OC_1, IC, k_1, k_1)，第二个卷积问题大小是 (batch, OC_1, $\lfloor H/st_1 \rfloor$, $\lfloor W/st_1 \rfloor$)×(OC_2, OC_1, k_2, k_2)。st_1 是第一个卷积的步长。st_2 是第二个卷积的步长。

AVX-512 CPU 性能结果：结果显示在图 3.8中。这里展示了与 PyTorch 相比的相对性能。Ansor 需要较长时间进行调优（大约半小时调优一个算子）。它需要对每个子图进行了 1000 次调优。Chimera 只需要几分钟就可以生成融合的算子代码，背后原因是 Chimera 的数据搬移量分析和优化算法都是可以一次性解出问题而不需要重复尝试。Relay 可以使用手动优化的代码模板而无需重复调优。对于批量矩阵乘法链融合，Chimera 可以获得与手动调优的库和编译器差不多甚至更好的性能，尽管它所需的编译用时更少。总体来说，Chimera 对 PyTorch 的平均加速比是 2.62×，对 Relay 的平均加速比是 4.78×，对 Ansor 的平均加速比是 1.40×，以及对 oneDNN 的平均加速比是 3.28×。对于将批量矩阵乘法和 softmax 融合，Chimera 实现了平均对 PyTorch 的 1.62× 加速，对 Relay 和 Ansor 的平均加速比分别为 7.89× 和 2.29×。

对于融合卷积链，本文使用了来自不同网络的卷积层[2,15,150-151]。卷积（特别是当核大小为 3×3 时）比批量矩阵乘法更复杂。3×3 卷积的滑动窗口在融合后可能导致大量的重复计算。Relay 和 Ansor 不能将这些复杂的算子融合在一起。因此，它们为卷积链生成分立的代码，这样性能会差一些。Chimera 对 Relay 和 Ansor 的加速比分别是 2.38× 和 1.94×。在图 3.8的部分 d) 中，展示了 Chimera 融合带 ReLU 的卷积链时的性

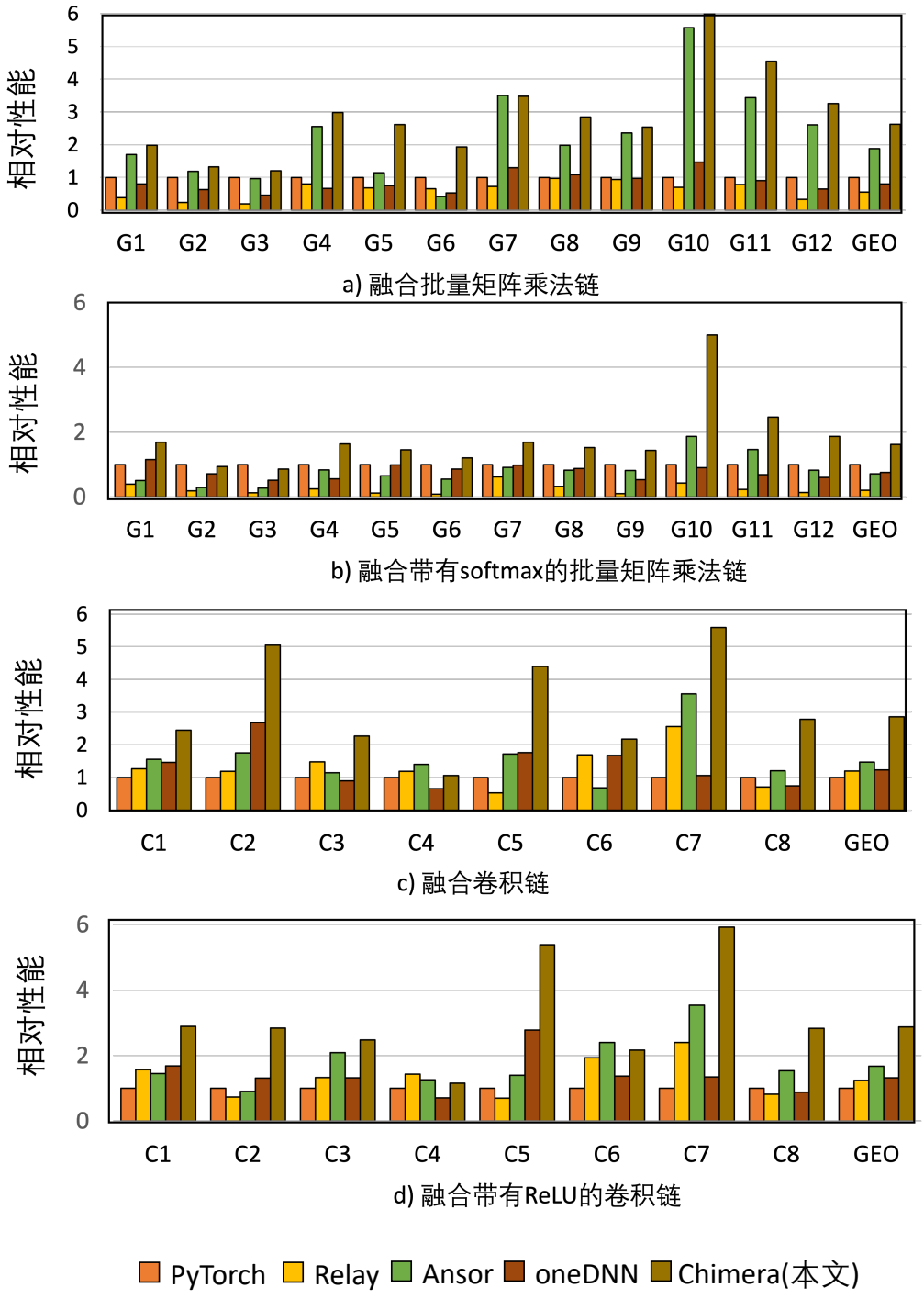


图 3.8　在 CPU 融合批量矩阵乘法链和卷积链的性能结果

能。加速比与融合两个卷积的情况相符（对 Pytorch 的 2.87×，对 Relay 的 2.30×，对 Ansor 的 1.71×），说明增加访存密集型算子并不会影响 Chimera 的融合效果。

Tensor Core GPU 性能结果：在图 3.9中本文展示了相应的性能结果。对于融合批量矩阵乘法链（图 3.9的部分 a），对 PyTorch 的平均加速比是 2.77×，对 TASO 的平均加

表 3.6　卷积链的输入形状配置

代号名称	IC	H	W	OC_1	OC_2	st_1	st_2	k_1	k_2
C1	64	112	112	192	128	2	1	3	1
C2	32	147	147	64	80	2	1	3	1
C3	64	56	56	128	64	1	1	3	1
C4	128	28	28	256	128	1	1	3	1
C5	16	227	227	64	16	4	1	3	1
C6	64	56	56	64	64	1	1	1	3
C7	64	56	56	64	64	1	1	1	1
C8	256	56	56	256	64	1	1	1	1

速比是 3.30×，对 Relay 的平均加速比是 1.69×，对 Ansor 的平均加速比是 1.33×，以及对 TensorRT 的平均加速比是 2.29×。这里取得的性能收益主要来自于将访存受限的计算密集型算子的访存操作减少，从而降低了对内存带宽的压力。经过统计，与 PyTorch 相比，Chimera 的总 DRAM 访问减少了 9.86% – 59.54%。像 TASO 和 Ansor 这样的编译器本身并不支持本文所做的融合优化，导致生成的代码中有两个单独的函数调用，性能也会差一些。

本文还与 TVM+CUTLASS[75]进行了比较，平均加速比是 1.51×。CUTLASS[96]是 Nvidia GPU 的开源 AI 算子库。最近的工作 BOLT[75]探索了使用 CUTLASS 模板融合矩阵乘法链和卷积链。相关代码已开源并可在 TVM[17]中获得。本文使用它为批量矩阵乘法链生成融合代码，并在图 3.9中展示了其性能，标记为 TVM+CUTLASS。通过对结果代码进行分析，本文发现 TVM+CUTLASS 性能无法超过 Chimera。原因有二，首先，CUTLASS 模板是由专家手动开发的，灵活性有限。具体来说，TVM 使用前端分析技术，利用模式匹配在输入程序中找到可融合的子图。这种模式匹配不够灵活，它将批量矩阵乘法链分类为非可融合子图。其次，CUTLASS 模板只使用固定的块执行顺序，当执行两个连续的矩阵乘法时优化方法和单个矩阵乘法仍然一致，这其实是次优的解决方案。相比之下，Chimera 可以通过分析模型，对融合数据流探索不同的执行顺序并选择预估性能最好的那种方案，这是 Chimera 取得加速的原因。

对于将批量矩阵乘法链与 softmax 融合的场景（图 3.9的部分 b），Chimera 对 PyTorch 的平均加速比是 2.74×，对 Relay 的平均加速比是 1.74×，对 Ansor 的平均加速比是 1.64×。本文不展示 TASO 和 TVM+CUTLASS 的性能，因为它们不支持 softmax。Relay 和 Ansor 为这个子图生成了三个内核，因为它们不能把 softmax 和其他算子融合。softmax 比逐点操作更复杂，因为它由三个基础算子组成：*exp*、*sum* 和 *div*。softmax 的拓扑结构不是线性的，所以无法被 Chimera 本身直接支持，这部分实验暂时通过先让 Chimera 分析批量矩阵乘法链的最优执行顺序，然后再手动插入 softmax 的方式完成的这部分实验。在之后加入 TileFlow 后的实验，就可以完全自动地支持这类融合而无需

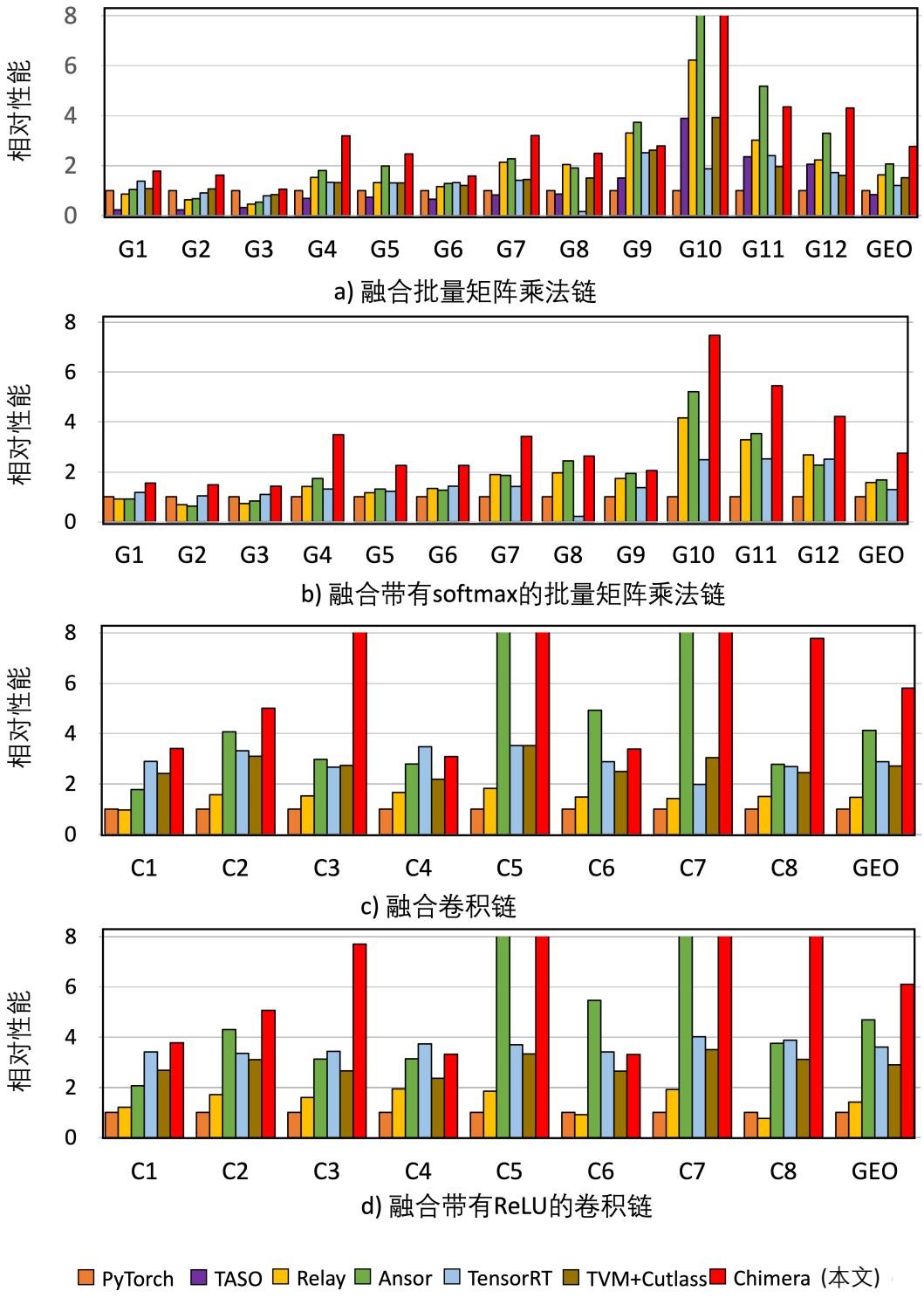


图 3.9　在 GPU 融合批量矩阵乘法链和卷积链的性能结果

手动介入。

对于融合卷积链（图 3.9的部分 c），Chimera 对 PyTorch 和 TensorRT 的平均加速比分别是 5.79× 和 2.01×。并不是所有卷积神经网络中的卷积层都适合融合。通常，只有

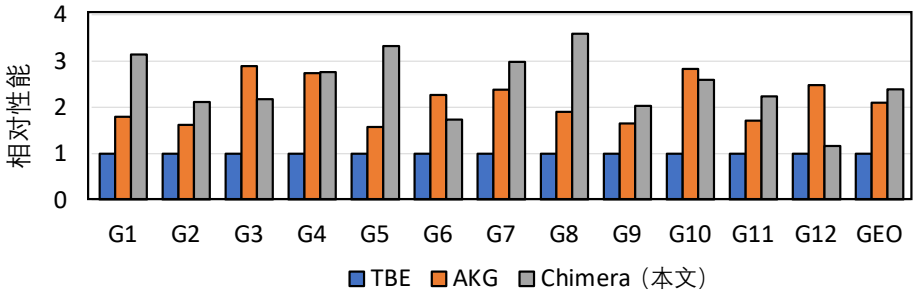


图 3.10　在 NPU 融合矩阵乘法链的性能结果

当卷积链中的第二个卷积性能受访存限制时，Chimera 才能通过融合获得加速。通常，当通道数目很小时，逐点卷积（窗口大小为 1×1 的卷积）倾向于访存密集型，它们通常用在神经网络的初始层（图像分辨率高且通道大小比较小）。但是其他卷积层（例如，3×3 卷积）通常是计算密集型的，不适合融合。本文使用表 3.6中的 C6 这一个例子来说明这一点，本文测试了融合逐点卷积与 3×3 卷积的性能。这个子图例子来自于 ResNet[2]的真实结构。如图 3.9的部分 c) 和 d) 所示，与 Ansor 相比，Chimera 在 C6 上无法获得加速，因为第二个卷积是性能受限于计算的，而不是访存，融合反而会影响第二个卷积的计算效率，拉低整体的性能。但对于其他子图，Chimera 可以一致获得比 Ansor 更好的性能。对于融合带 ReLU 的卷积链，对 Relay 的平均加速比是 4.32×；对 Ansor 平均加速比是 1.30×。

NPU 性能结果：本节最后，在 NPU 上评估矩阵乘法链融合性能。对于所有的矩阵乘法链，使用批处理大小（batch）为 1。本文的对比对象是 CANN[140]中的 TBE 库（Tensor Boost Engine）。TBE 为 Ascend NPU 提供手动优化的矩阵乘法实现。它不能在一个函数中融合两个矩阵乘法，所以性能相比 Chimera 会差一些。本文比较的另一个工作是 AKG[31]。AKG 可以在 Ascend NPU 上为矩阵乘法提供更好的性能。但是，AKG 也没有能力做矩阵乘法链的融合。如图 3.10所示，Chimera 对 TBE 实现了平均 2.39× 的加速比，对 AKG 的平均加速是 1.14×。在一些情况下，Chimera 性能略差于 AKG，因为本文使用的 NPU 架构中有一个对融合比较关键的存储层次很小，这个存储乘次叫做 Unified Buffer（UB），只有 256KB，对于本文要融合的矩阵乘法来说，这个大小很难切出足够大的块来保证计算单元充分运转。当矩阵乘法变大时，UB 成为性能瓶颈，拖慢了整体执行速度。不过总的来说，Chimera 通过融合还是能取得更好的性能的。

3.6.3　访存行为分析结果

这一小节进一步分析 Chimera 生成的代码的访存行为。这部分主要在 CPU 上进行分析，分析对象是融合后的矩阵乘法链。对于这个子图，Chimera 将两个批量矩阵乘法

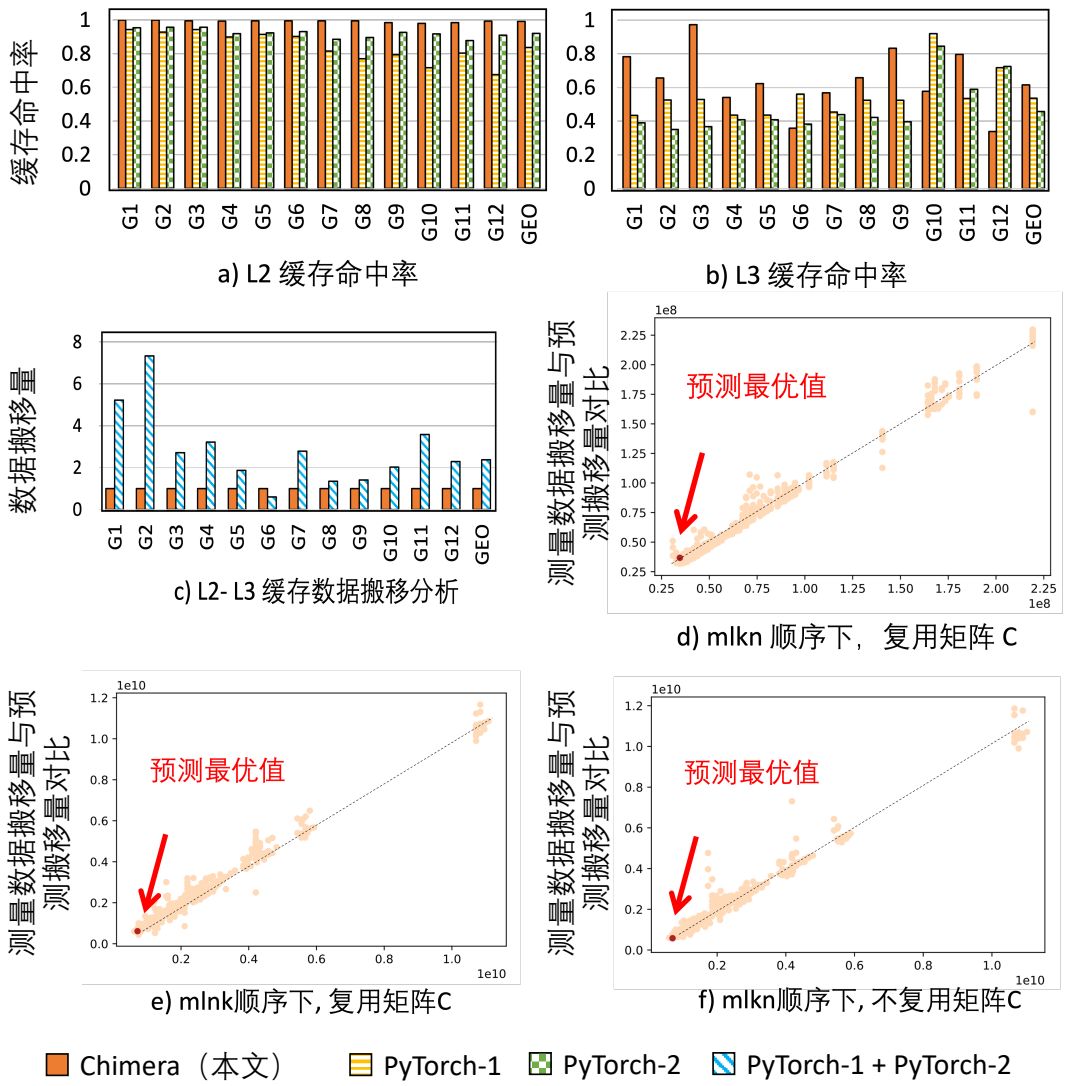


图 3.11　在 CPU 上展示 Chimera 融合矩阵乘法链带来的访存优势，以及展示 Chimera 的预测访存量与实际访存行为的高度相关关系

融合在一起并只生成一个函数。对于 PyTorch，它使用两个单独的函数去实现矩阵乘法链，因此必须分别为其分析两个批量矩阵乘法函数的性能。如图 3.11的部分 a) 和 b) 所示，Chimera 的 L2 和 L3 缓存平均命中率超过了 PyTorch。PyTorch-1 指的是 PyTorch 使用的第一个矩阵乘法函数；PyTorch-2 指的是 PyTorch 使用的第二个矩阵乘法函数。Chimera 的缓存命中率更高意味着更少的片外内存访问和更多的片上数据搬移（例如，数据搬移主要发生在 L1 和 L2 缓存中），这对于提高性能至关重要。这里还分析了不同缓存层级之间的数据搬移量，并发现与 PyTorch 相比，Chimera 也大大减少了 L2 和 L3 缓存之间的数据搬移（平均减少了 59.75%），如图 3.11的部分 c) 所示。此外，Chimera

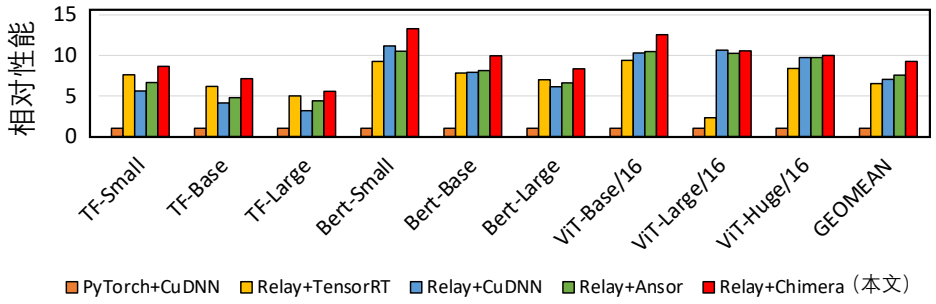


图 3.12　在 A100 GPU 上的端到端性能对比

.

的 DRAM 访问平均减少了 75.17%。同时，Chimera 在 L1 和 L2 缓存之间的数据搬移平均增加了 46%，这体现了数据局部性的提高。

为了验证 Chimera 对于链式结构的融合情况下的数据搬移优化的目标函数的准确性，本文设计了三个不同的场景以分析矩阵乘法链（$M = N = K = L = 2048$）的数据搬移量，并在图 3.11的部分 d)-f) 展示了预测和实测的数据搬移量。对于每个情况，本文测试数百个不同的数据流，每个数据流都采用了不同的分块参数设置，并在图中绘制他们的数据搬移量。图的 x 轴是 Chimera 的分析算法所计算的数据搬移量，y 轴是使用访存分析工具得到的实际访存量，通过观测 x 轴值和 y 轴值的相关度，可以判断 Chimera 的算法是否准确。如果 Chimera 的算法是准确的，图中的点将接近于 $y = x$ 这一条线，表明线性关系（最强的正相关）。实验测试的是 L1 缓存和 L2 缓存之间的数据搬移量。对于图中部分 d) 展示的情况，使用的块执行顺序是 $mlkn$。这一实验的结果显示 Chimera 中的数据搬移量算法准确性很高，回归线接近于 $y = x$，经过统计，实际值和算法预测之间的相关性也很高（$R^2 = 0.97$）。实验还在图中用红点显示了算法求解的最优数据搬移量，这一求解的值接近实际最优值（图中的左下点）。对于图中部分 e) 的情况，使用另一个顺序 $mlnk$ 来测试，结果显示 Chimera 的算法估计的搬移量也很准确（$R^2 = 0.98$）。在部分 f) 中，仍然使用顺序 $mlkn$，但强制第二个矩阵乘法不重用中间矩阵 C，也就是不融合，这将导致更多的数据搬移。这个例子用于表明融合确实相较于不融合在数据搬移量上是有优势的。在这三个例子中，部分 d) 中带有中间数据重用的 $mlkn$ 顺序实际上是 Chimera 算法解出的最优顺序。这些实验结果证明了 Chimera 的分析模型和优化算法有效且准确。

3.6.4　端到端网络测试结果

对于完整网络端到端性能评估，本文使用 Transformer（简称为 TF）、Bert 和 ViT（批处理大小为 1）。TF-Small、TF-Base、TF-Large 是 Transformers 的三种不同配置（中

间层的特征大小不同），对应了不同的批量矩阵乘法链，他们的输入序列长度都设置为512。不同配置的批量矩阵乘法链输入形状在表 3.5中展示过。

实验使用开启 CuDNN 选项的 PyTorch 作为对比对象（标记为 PyTorch+CuDNN）。实验还将 Chimera 与 TensorRT、CuDNN 和 Ansor 进行比较。Relay 能够直接调用 TensorRT 和 CuDNN（标记为 Relay+TensorRT 和 Relay+CuDNN）。Ansor 也是集成在 Relay 中的，因此可以使用 Relay+Ansor 作为对比对象。实验设置 Ansor 为每个批量矩阵乘法链搜索 1000 次后再生成代码。为了对比 Chimera 的性能，实验也将 Chimera 集成到 Relay 中，并将 Relay 的批量矩阵乘法链函数调用替换为调用 Chimera 生成的代码（标记为 Relay+Chimera）。

实验使用一块 A100 GPU 作为测试设备。性能结果展示在图 3.12中。Relay+Chimera 比 PyTorch+CuDNN 快得多，因为 Relay+Chimera 使用静态图进行融合优化，而 PyTorch 使用动态图且没有融合。与 Relay+TensorRT、Relay+CuDNN 和 Relay+Ansor 相比，Relay+Chimera 的平均加速比分别是 1.42×、1.31× 和 1.22×。Relay+TensorRT 比其他编译器生成的结果慢，因为 TensorRT 不能将 softmax 层和其他算子融合，性能较差。

3.6.5　TileFlow 融合性能结果

这一节，实验将 TileFlow 加入进来测试性能。TileFlow 作为性能模型可以为 Chimera 提供对于通用拓扑结构的图融合性能分析，结合 TileFlow 中实现的自动搜索功能，可以针对新兴架构硬件搜索更优的数据流。实验测试的图结构和前述一致。在性能实验前，首先要验证 TileFlow 的性能模型是否准确。

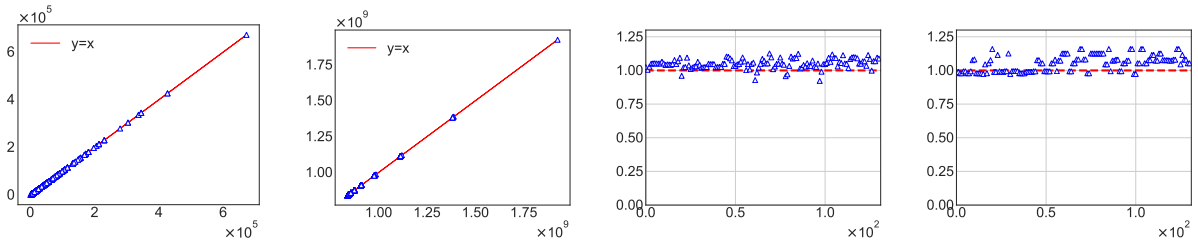
3.6.5.1　TileFlow 性能模型准确性

本文使用真实硬件设计和最先进的性能模型来验证 TileFlow 的准确性。对于性能模型的对比对象，本文使用 Timeloop[32]。对于真实硬件对比对象，本文实现了一个类似 TPU 的加速器，通过比较 TileFlow 的预测周期和加速器的寄存器传输级（Register Transfer Level，RTL）仿真周期结果之间的差异来判断 TileFlow 的准确度。

加速器实现：本文使用 Chisel 实现加速器并生成 Verilog RTL 进行实现。本文的加速器有四个核心，每个核心有两个 PE 阵列：一个用于矩阵乘法（16×16），另一个用于向量计算（16×3）。每个核心的片上存储大小为 384KB。芯片的 DRAM 带宽为 25.6GB/s，字宽是 16 位。本文使用 Cadence Genus Synthesis and Innovus 工具对 RTL 进行综合。综合报告显示，本文的加速器面积在 TSMC 22nm 制程工艺下面积是 $7.84mm^2$，频率为 400 MHz。这个加速器支持矩阵、向量的加载、计算、存储指令。使用这些指令编程可以得到多个测例，将它们编译成二进制文件就可以执行。对比性能时，可以使用 Verilator[152]（使用的版本是 4.0）进行 RTL 级别性能模拟以获得运行性能（周期数）。

表 3.7　测试 TileFlow 时使用的加速器配置

加速器名称	PE 数目	子核数目	核数	片上内存大小	DRAM 带宽
Edge	32 × 32	1	4	L1: 4MB	60 GB/s
Cloud	256 × 256	16	4	L1: 20MB L2: 40MB	384 GB/s



(a) 与 Timeloop 对齐周期数结果　(b) 与 Timeloop 对齐能耗结果　(c) 与真实加速器对齐周期结果　(d) 与真实加速器对齐能耗结果

图 3.13　TileFlow 性能模型准确性验证实验

与 Timeloop 的比较的时候，只能使用单个算子（矩阵乘法），因为 Timeloop 不支持多算子或融合。实验枚举了 1152 种不同的矩阵乘法数据流（通过改变循环分块方法）。实验比较了 TileFlow 和 Timeloop 的预测周期结果，如图 3.13(a)所示。x 轴是 Timeloop 报告的周期，y 轴是 TileFlow 报告的周期。TileFlow 和 Timeloop 的结果之间的相关性很高（$R^2 = 0.999$），结果几乎是一致的。同样，对于能耗预测，TileFlow 的结果与 Timeloop 相比仍然非常一致，如图 3.13(b)所示，二者的平均误差为 0.1%。这表明在对于单个算子的分析上，TileFlow 基本与 Timeloop 一致。

在与真实加速器的比较中，实验使用 self-attention[87]进行测试。在加速器上，可以用汇编编程实现高度优化的融合代码，并通过枚举分块参数枚举了 131 种不同的数据流。实验比较加速器的运行周期和 TileFlow 预测的结果，结果如图 3.13(c)所示。x 轴代表不同的测试例子，y 轴是相对运行周期（TileFlow Cycle/Real Cycle）。黄色圆圈是基于图的方法预测的结果[136]，蓝色三角形是 TileFlow 的结果。TileFlow 的平均误差为 5.4%，而基于图的方法的平均误差为 48.8%，这表明基于图的方法是不适合对融合数据流进行分析的。图 3.13(d)是 TileFlow 预测的能耗结果和实际能耗的比较。平均误差为 6.1%。对于部分测试例子，TileFlow 的能耗预测不准确，这主要是因为这些例子中使用了小的分块参数，TileFlow 倾向于对它们估计过高的数据搬移量，因为它假设每个循环迭代都会发生数据替换，但是在实际硬件的手写代码中，有的数据搬移会被手工优化掉，因为片上可能能放下更多的输入数据，这就导致 TileFlow 过高地估计了能耗。不过总体来说，TileFlow 的准确性还是非常高，这保证了在后续搜索优化的数据流的过程中能较为准确地定位高性能的数据流。

表 3.8　不同数据流配置及其解释

数据流名称	解释
针对 self-attention 融合的数据流	
Layerwise	不融合，一层一层执行
Uni-pipe	流水线执行 $Q \times K$ 与 softmax 层 但是不分块自注意力头维度和序列长度维度
FLAT-HGran[89]	融合 $Q \times K$ 和 softmax 层并且 对批处理维度、自注意力头维度都分块
FLAT-RGran[89]	融合 $Q \times K$ 和 softmax 并且 对批处理维度、自注意力头维度、序列长度维度都分块
Chimera	融合 self-attention 所有层，但是 softmax 层需简化且依靠手工插入
针对卷积链融合的数据流	
Layerwise	不融合，一层层卷积执行
Fused-Layer[90]	两个卷积进行融合而且分块卷积行和列维度
ISOS[153]	融合两个卷积，但是只对行或者列一个维度分块

3.6.5.2　搜索模块效率

本文还需要测试 TileFlow 中基于机器学习算法的搜索模块的效率。本文使用的是表 3.5中的网络的 self-attention 层做测试。除了 self-attention 层，本文还使用卷积链进行测试。卷积链的输入形状展示在表 3.6中。为了作对比，对 self-attention 本文配置了五种数据流，他们的名字是 *Layerwise*、*Uni-pipe*、*FLAT-HGran*、*FLAT-RGran* 和 *Chimera*，如表 3.8所示。其中 *Layerwise* 指的是不做任何融合，所以每一层都是一个个执行的。*Uni-pipe* 是融合但是不分块 self-attention 的自注意力头维度。*FLAG-HGran* 是使用了先前工作 FLAT[89]中提出的在 self-attention 的自注意力头做分块的数据流，*FLAT-RGran* 则是进一步在自注意力层的输入序列上也做分块。*Chimera* 指的是前述章节中，不包含 TileFlow 部分的融合数据流方法，与之对应，本文加入了 TileFlow 之后搜到的数据流都用 TileFlow 代表。在之前的 Chimera 中，使用的 softmax 是简化的，以便于手工插入到生成的代码中，但是在 TileFlow 中，可以使用完整的 softmax 实现，它包含五个算子（*max*、*sub*、*exp*、*sum*、*div*），构成的拓扑是非线性的，TileFlow 可以自动处理这类拓扑的融合。对于加速器配置，这部分使用表 3.7中的 Edge 规格加速器。Edge 加速器有四个核心，每个核心有 4MB L1 缓冲区。L1 带宽是 1.2TB/s。

这部分的实验机器是 Intel Xeon(R) Gold 6348 CPU @ 2.60GHz。首先，实验测试只搜索分块参数的结果。在搜索中只使用单线程，设置探索 50 轮，针对每个数据流设计方案，每轮抽样 200 个分块参数选择，每轮搜索需要大约 12 秒。在图 3.14的 a) 部分，实验绘制了随探索进行的性能变化的图，图的纵轴是归一化后的。这个结果使用的是表 3.5中的 Bert-S 网络的 self-attention 输入形状。结果显示，TileFlow 需要大约 3.2 分钟到 6.4 分钟就能收敛到比较好的结果，说明对于分块参数搜索 TileFlow 还是较快的。

此外，实验也测试在完整的数据流设计 3D 空间（计算顺序、资源映射和循环分

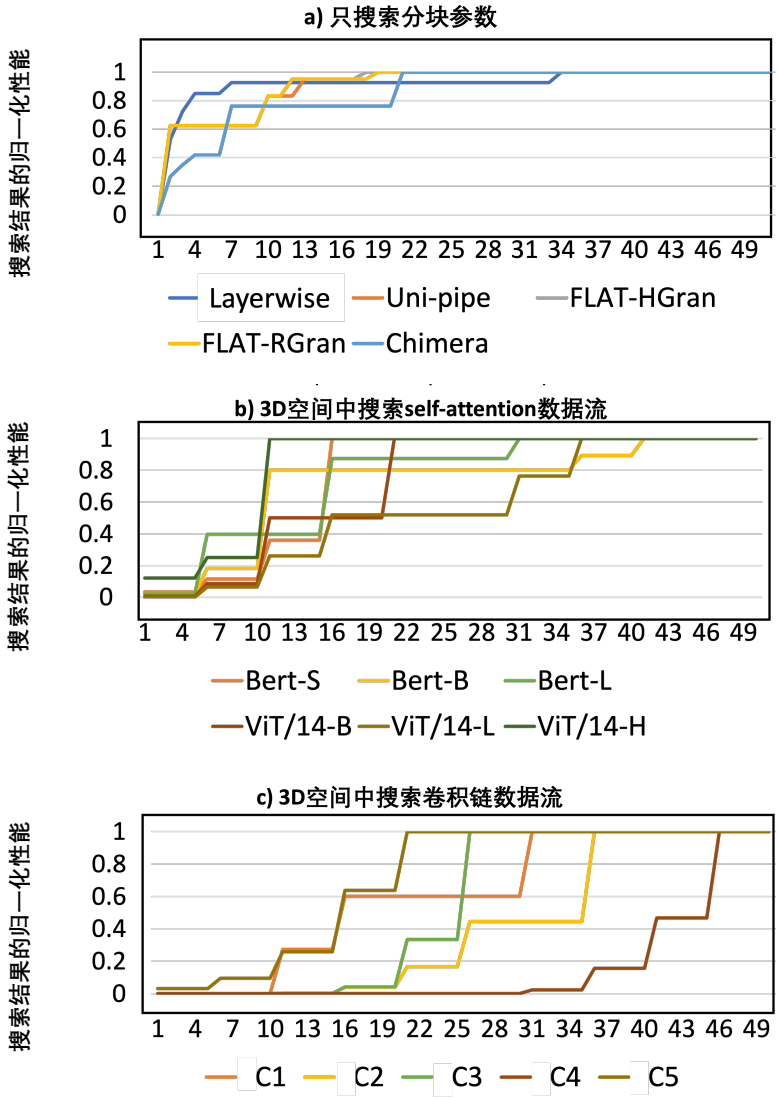


图 3.14　TileFlow 的数据流搜索算法效率测试结果

块）中进行搜索的效率。3D 空间的大小远大于单独的分块空间，比如对于实验中使用的例子，3D 空间中的可选的数据流计算顺序和资源映射设数量大概是 5103 到 20412，每种设计可能性进一步拥有自己的分块参数选择，这样整个 3D 空间的体量大概就是分块空间的 5000 倍以上。在图 3.14的 b) 和 c) 部分进一步绘制了在 3D 空间中进行搜索的结果，x 轴代表搜索轮次，代表搜索算法迭代一次。可以看出，TileFlow 需要几轮（少于 50 轮，每轮抽样 20 种融合数据流）就能收敛，在 3D 空间搜索并没有表现出比只在分块空间中搜索慢很多的问题，这主要得益于机器学习算法能快速适配搜索空间，保证每一步搜索都高效。

值得注意的是，如果不使用机器学习算法而是穷举法完全探索 3D 空间，就需要很

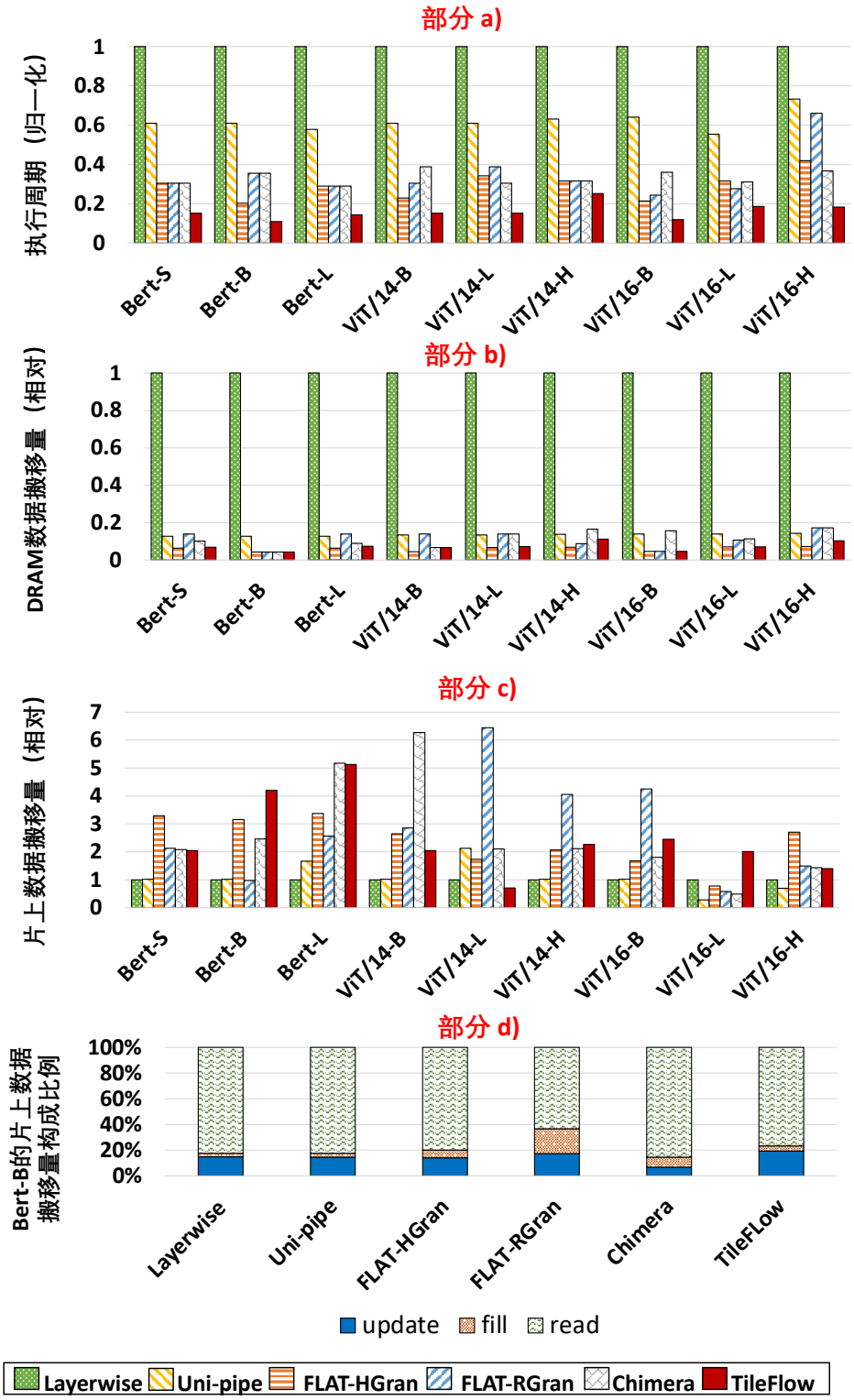


图 3.15　在 Edge 加速器上测试 self-attention 融合数据流的性能比较

长时间（1-2 天）。探索结果显示，不同输入形状偏好不同的数据流设计选择，很难选择一个对所有输入形状都能性能最好的数据流。因此，可以选择一个性能泛化性较好

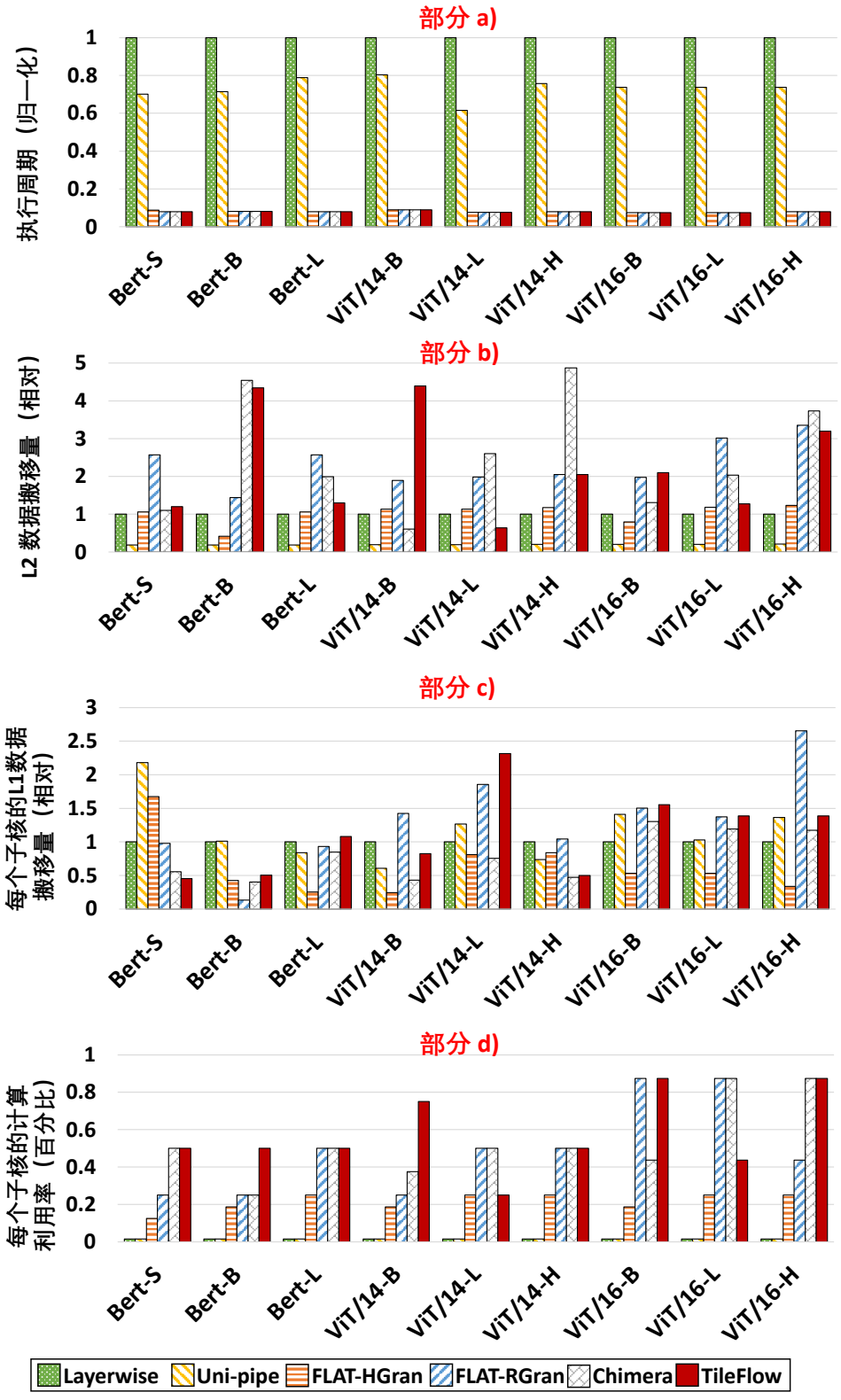


图 3.16　在 Cloud 加速器上测试 self-attention 数据流的性能比较。

的数据流作为 TileFlow 搜索数据流的代表（称为 *TileFlow* 数据流）。对于 self-attention，*TileFlow* 数据流是将所有三个计算算子：$Q \times K$、softmax 和 $L \times V$（在图 2.2中使用的

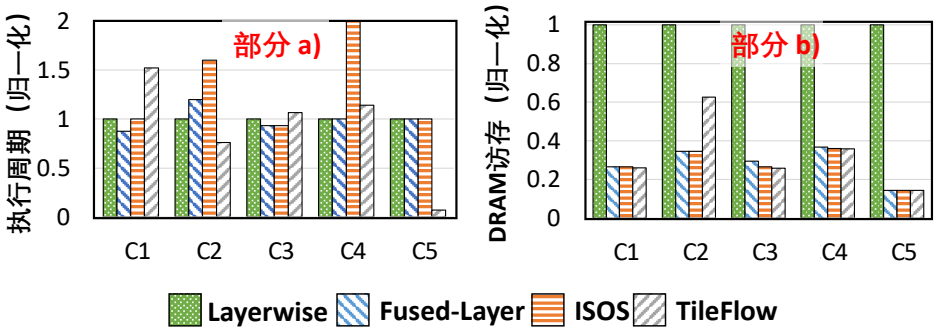


图 3.17　在 Cloud 加速器上融合卷积链的性能结果。

符号）全部流水线化，且所有循环都进行了分块。与 *Layerwise* 数据流相比，*TileFlow* 数据流达到的加速比是 6.65×。与表 3.8中最好的数据流（*FLAT-RGran*）相比，平均加速比是 1.85×。对于卷积链，*TileFlow* 数据流是将两个卷积的通道维度进行分块之后再流水线化。与 *Layerwise* 相比，*TileFlow* 数据流的加速比是 1.31×，与 *Fused-Layer* 相比的加速比是 1.28×。

3.6.5.3　TileFlow 融合数据流性能比较

在这部分，实验比较了不同的融合数据流[89-90,153-154]与 TileFlow 的数据流在延迟和访存方面的表现的差异。这里使用两种加速器配置：Edge 和 Cloud，如表 3.7所示。Edge 加速器在前面的实验中用过。Cloud 加速器规格有四个核心。每个核心进一步拥有 16 个子核心和一个 40MB 的 L2 片上缓存，L1 带宽是 9.6TB/s，L2 带宽是 1.9TB/s。为了确保不同数据流之间的公平比较，所有的数据流都在自己的设计空间中进行过搜索并保留搜索到的最优参数，比如 *FLAG-HGran* 和 *FLAG-RGran* 都搜索了最优的分块参数配置。

首先展示在 Edge 加速器上的对比结果，如图 3.15所示。在部分 a) 中，展示的是所有 self-attention 输入形状配置下所有数据流的执行周期结果。与 *Layerwise* 相比，*Uni-pipe* 数据流达到了 1.62× 的加速，*FLAT-HGran* 数据流达到了3.59× 的加速，*FLAT-RGran* 数据流达到了2.89× 的加速，*Chimera* 数据流达到了2.91× 的加速。TileFlow 在所有数据流中表现最好：与 *Layerwise* 相比达到了 6.65× 的加速；与 *Uni-Pipe* 相比达到了 4.11× 的加速；与 *FLAT-HGran* 相比达到了 1.85× 的加速；与 *FLAT-RGran* 相比达到了 2.30× 的加速；与 *Chimera* 相比达到了 2.28× 的加速。为了分析加速的来源，实验还在图 3.15中 b) 和 c) 部分展示了 DRAM 和片上缓存的数据搬移量结果。平均而言，与 *Layerwise* 数据流相比，*Uni-pipe* 和 *FLAT-HGran* 数据流可以减少90.4% 的 DRAM 访问，*FLAT-RGran* 数据流可以减少81.5% 的 DRAM 访问，*Chimera* 数据流可以减少75.1% 的 DRAM 访问，*TileFlow* 数据流可以减少87.1% 的 DRAM 访问。减少 DRAM 访问意味着片上存

储中数据重用的提高。虽然 *FLAT-HGran* 在某些输入形状下的 DRAM 数据搬移量少于 *TileFlow*，但其 PE 利用率不高（大约为 *TileFlow* 的 50%），因此其性能较差。在 c) 部分还绘制了片上缓存（Edge 的 L1）数据搬移量。总体来说，TileFlow 的 L1 数据搬移量增加了 2.01×–6.45×。对于一个特定的输入形状（Bert-B），在图 3.15的 d) 部分中还展示了详细的 L1 数据搬移量构成。*update* 指的是写回到 L1 缓存的数据搬移，*fill* 指的是从 DRAM 到 L1 缓存的初始数据加载，*read* 指的是从 L1 缓存到寄存器的数据加载。平均而言，80.9% 的 L1 数据搬移是 *read*，14.7% 是 *update*。

对于 Edge 加速器，在所有的数据流中，*Uni-pipe* 比 *Layerwise* 数据流性能更好，因为融合消除了大量的 DRAM 访问；*FLAT-HGran* 数据流比 *Uni-pipe* 数据流性能更好，因为通过分块，被分块的数据可以在空间上映射到不同的核心上，增加了并行性；*FLAT-RGran* 和 *Chimera* 数据流产生了与 *FLAT-HGran* 相似的性能，但它们的 L1 缓存占用更少，*FLAT-RGran* 仅需要 *FLAT-HGran* 28.4% 的 L1 占用量就可以完成计算，而 *Chimera* 需要的更少，仅需要 14.8%。

进一步在 Cloud 加速器上进行测试，结果展示在图 3.16中。测试结果表明，与 *Layerwise* 数据流相比，*Uni-pipe* 数据流的加速比为 1.37×，所有其他数据流的加速比则全都一致是 12.63×。*Uni-pipe* 的加速比较低是因为空间利用率低，由于缺乏分块优化，*Uni-pipe* 仅使用大约 25% 的核心进行计算，而其他数据流可以通过分块充分利用所有核心。*FLAT-HGran*、*FLAT-RGran*、*Chimera* 能够达到相同的性能，这其实表明了对于 Cloud 加速器这种硬件资源充足的场景，分块的粒度对性能的影响很小，不同的融合数据流，哪怕分块策略不同，也能取得基本一致的最优性能。

在 Cloud 加速器上对于片上内存数据搬移量的分析显示，所有融合数据流相比于 *Layerwise* 数据流都减少了相似的 DRAM 访存（平均减少了 86.6%）。因此，接下来分析重点放在片上内存数据移动上。在 L2 缓存的数据搬移量方面，除了 *Uni-pipe* 之外的所有数据流都增加了片上数据搬移量，这表明了它们在片上内存中有更高的数据复用率。*Uni-pipe* 的 L2 数据搬移量较低，因为它的数据主要存放在 L1 缓存中。对于 L1 缓存数据搬移量，以单个子核为单位的统计数据显示，尽管与 *Layerwise* 数据流相比，融合数据流的 L1 数据搬移量没有增加（大约为 *Layerwise* 的 52.1% – 108.5%），但所有子核的 L1 数据总搬移量有所增加（增加了 7.01×–33.27×）。此外，子核空间利用率的分析表明，与 *Layerwise* 相比，*FLAT-HGran* 的利用率提高了 13.46×；*FLAT-RGran* 提高了 28.34×；*Chimera* 提高了 32.02×；*TileFlow* 提高了 34.58×。这些结果揭示了在 Cloud 加速器上，TileFlow 通过有效的数据流设计和优化分块策略，可以显著提高计算资源的利用率以及减少 DRAM 访问，从而达到更高的性能。

除了 self-attention 层之外，实验还在 Cloud 加速器上展示了卷积链融合的结果，实

验使用表 3.6中的输入形状。这些输入形状来自现实使用的网络[2,15,155]。卷积链中的两个卷积使用 3×3 的窗口大小。在图 3.17中展示的执行周期和 DRAM 访存量，展示的都是归一化后的结果。与 *Layerwise* 相比，*Fused-Layer* 数据流的平均加速比为 1.01×。尽管 *Fused-Layer* 数据流对延迟改进不大，但它减少了 73.0% 的 DRAM 访问，并且减少了 30.1% 的能耗。*ISOS* 的加速效果不佳，因为 *ISOS* 最初是为稀疏卷积设计的，但本文在实验中是稠密计算。*TileFlow* 实现了相对于 *Layerwise* 和 *Fused-Layer* 能达到 1.59× 的加速。这些结果表明，对于卷积链，*TileFlow* 能够有效地提高性能，减少 DRAM 访问，并且在能耗上也有显著改进。与专门为稀疏卷积设计的数据流（如 *ISOS*）相比，*TileFlow* 在处理卷积时能够提供更好的性能提升。这进一步验证了 *TileFlow* 在多种计算密集型任务上的适用性和优越性。

3.7　本章小结

本章中，针对 AI 编译的图层次，论文指出先前工作抽象层次的局限性以及相应导致的优化的局限性，并提出了基于块的抽象技术。利用块抽象，论文进一步提出两个框架 Chimera 和 TileFlow 共同完成图层次编译任务。Chimera 提供了基本的代码生成支持和对于计算密集型算子链的自动分析和融合优化功能，TileFlow 针对更通用的图结构提供性能分析和自动优化的支持，与 Chimera 结合后，可以搜索得到更多高性能的融合数据流。在实际硬件 CPU，GPU，NPU 以及原型硬件 Edge 加速器和 Cloud 加速器上，论文提出的技术都取得了更好的性能。

第四章　基于循环抽象的算子层编译

本章中将继图层次编译之后继续向下进入算子层编译。图层编译将算子融合之后，图中剩下的就是一个个融合后的算子节点，这些节点的代码生成，在前述章节中有简要提及，例如可替换微内核内部的代码设计，不过这些内容对于算子代码生成还是远远不够的。为此，本文将在算子层这一章仔细介绍如何处理算子内的循环优化、代码生成等问题。

4.1　算子层技术背景介绍

4.1.1　基于循环的抽象

在算子层使用循环作为抽象并非本文首次创新，但是先前工作在基于循环抽象进行编译和优化时存在局限性，导致对算子层面的代码生成性能受限，本文将在这一节解释它们的局限性。

对于算子层编译，研究较多的两种技术路线是计算调度分离路线和多面体模型路线。在前文介绍过，多面体模型路线最初并不是为 AI 算法设计的，它更关注对于依赖关系复杂的嵌套循环的自动变换、自动寻找并行机会和数据重用机会[43]，多面体模型技术的能够对于不同语句间的数据依赖提供良好的建模，对于复杂的循环边界也能提供良好的建模。近些年，一些工作如 Tensor Comprehension[44]，Tiramisu[30]，AKG[31]等尝试将多面体模型引入到 AI 编译中。但是 AI 计算中的算子往往都是有着规整的循环边界，语句间的数据依赖也往往遵照简单、规整的多维数组访存模式，这就导致多面体模型对于 AI 计算无法体现优势，而且多面体模型尝试在一个巨大的、完整的优化空间中寻找最优解，AI 计算的循环长度如果太大，就会让优化空间过于巨大，多面体模型往往难以解出答案。而且 AI 计算一些自身的特点，如可以进行多重分块，循环顺序可以自由调换等，都没有在多面体模型中有建模分析优势，这就导致多面体模型往往显得太过通用和笨重，解出的优化方案也常常并非最优。所以，在本文中，主要沿着计算调度分离路线进行探讨。

计算调度分离路线主要思路是对循环本身进行简单抽象，每个循环都有自己的循环起点、终点、步长信息，一个 AI 算法的张量算子就是由多个循环组成的循环列表和若干计算、访存语句构成。形式化地说，一个循环的抽象就是三元组：

$$l : \langle beg, end, step \rangle \tag{4.1}$$

表 4.1　不同目标硬件上可以使用的基础原语举例

硬件	名称	解释	可控参数
通用	split	把一个循环分成两个循环	分割循环时的子循环长度
	fuse	把两个相邻嵌套循环合并为一个循环	指定需要合并的循环
	reorder	重新调整两个循环的顺序	指定交换顺序的循环
	unroll	将循环进行展开处理，内部嵌套的语句重复复制	要展开的长度
	vectorize	将循环变短，同时将内部嵌套语句替换为 SIMD 语句	向量化的长度
	inline	把一个循环体完全融入到它的消费者循环体中	无
	compute at	把生产者算子循环体插入到消费者循环体的某个层次	指定插入到消费者的哪个循环层次
CPU	parallel	用多线程并行	要并行的循环
GPU	cache	插入读写节点，软件模拟缓存行为	指定哪个算子需要缓存
	bind	把循环绑定到并行单元上	指定哪个循环绑定到哪些单元
FPGA	buffer	类似 cache，增加数据读写缓冲区	哪些张量读写需要缓冲
	pipeline	构建流水线	指定构建流水线级数
	partition	内存分块，减少读写冲突	分块数目

其中 l 是循环名称，beg 是循环开始起点，end 是循环终点（不包含）,$step$ 是循环步长。通常情况，编译器会对循环的起点做归一化，所以本文可以总认为循环的起始点是 0，步长是 1，也就是 $beg = 0$，$step = 1$。对算子的嵌套循环可以进行许多变换操作，如表 1.2所示。计算调度分离工作对循环变换的观点是，这些变换可以通过一些基础原语实现，本文在表 4.1中展示了主要使用的基础原语。不同硬件设备可以使用的原语也会有一定区别，比如 split, fuse, reorder 这类原语适用于所有硬件后端，parallel 却只能用在 CPU 上，cache 只能用在 GPU 上，partition 只能用在 FPGA 上。这些基础原语也可以基于循环抽象进行建模，如 split 的行为定义是：

$$l1 : \langle 0, \frac{end}{factor}, 1\rangle, l2 : \langle 0, factor, 1\rangle = split(l : \langle 0, end, 1\rangle, factor) \tag{4.2}$$

fuse 的行为定义是：

$$l : \langle 0, f1 \times f2, 1\rangle = fuse(l1 : \langle 0, f1, 1\rangle, l2 : \langle 0, f2, 1\rangle) \tag{4.3}$$

可以看出这些基础变换作用到循环上之后，得到的结果仍然是可以用循环抽象表示的。通过组合这些基础变换，就可以得到更复杂的循环变换，比如表达对一组循环的分块（tile），可以组合 split 与 reorder 得到：

$$l1o, l2o, l1i, l2i = tile(l1, l2) \tag{4.4}$$

等价于

$$\begin{aligned} l1o, l1i &= split(l1, factor1) \\ l2o, l2i &= split(l2, factor2) \\ reorder(l1o, l2o, l1i, l2i) & \end{aligned} \tag{4.5}$$

先前的计算调度分离工作 Halide[38]和 TVM[17]基于这些变换原语，可以完成针对

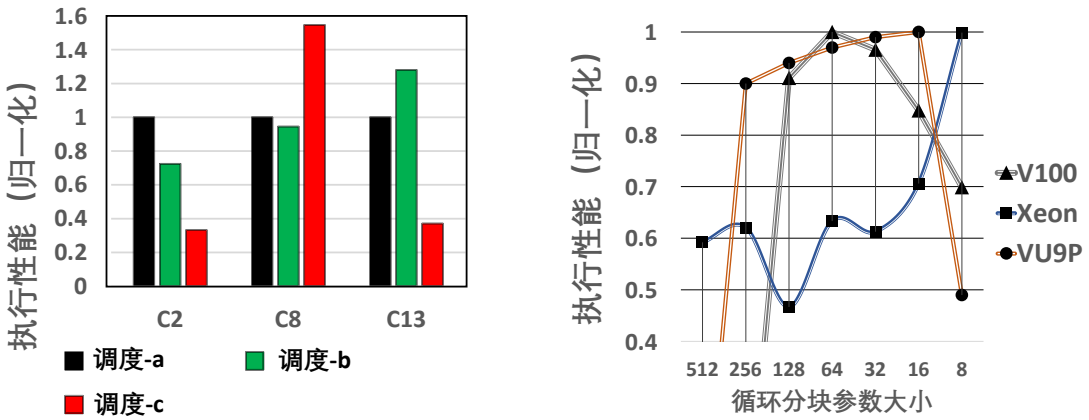


图 4.1　左侧：在同一平台上，对于卷积运算的三种不同调度策略实现的性能。右侧：在 V100、Xeon E5 和 VU9P 上，不同分块参数对于卷积的性能的影响

多种设备的、多种 AI 算子的计算调度和优化。但是他们的局限性在于，这些优化都需要人的参与。他们要求使用者利用这些调度原语开发算子优化程序，被称为调度模板，模板中根据原语的参数需求，可以暴露一系列可调优参数，Halide 和 TVM 就可以通过自身的自动调优工具优化这些参数，从而得到完整的算子级调度方案，最后再根据调度结果，把抽象的循环翻译成具体的循环（一般是 for 循环）从而完成代码生成。然而，并非所有的开发者都有能力完成模板开发，模板中使用的原语不仅和 AI 算法的循环结构相关，还和具体的硬件后端有关，需要开发者有大量的专家知识和经验。此外，原语的组合空间也十分巨大，不同的组合方式往往带来非常不同的性能结果，这就导致依靠人力搜索最优调度方案不太可能。

为了说明这个问题，本文使用两个例子来展示在不同的后端设备上实现不同的调度策略时遇到的挑战。首先，不同的调度原语组合方案会带来性能波动。在图 4.1左侧，本文使用三种不同的调度方案来优化 V100 GPU 上的卷积。本文选择了三种不同的输入形状（用 C2、C8 和 C13 代指，具体形状参数参照表 4.3）进行说明，且批处理大小为 8。调度-*a* 将批（batch）维度分块，而调度-*b* 将批（batch）维度绑定到不同的并行线程块上，调度-*c* 则简单地将所有循环融合在一起。这些调度之间的差异只是在原语的顺序上有所不同或局部原语的选择不同，但他们对性能的影响差异却很明显。本文还发现，不同的输入形状倾向于使用不同的调度策略。对于 C2，调度-*a* 是最好的；对于 C8，最好的是调度-*c*；对于 C13，调度-*b* 表现最佳。其次，硬件的多样性进一步增加了复杂性。在图 4.1右侧，本文比较了三个不同的平台：Nvidia V100 GPU、Intel Xeon E5 CPU 和 Xilinx VU9P FPGA。本文改变了三个平台上卷积循环的分块参数（从 8 到 512）并测试性能，如图所示，三个平台具有不同的性能变化趋势，并且对于不同平台最优的分块参数也不一样。

从上述例子中可以看出，在不同 AI 芯片上编写优化的调度策略对使用者来说是一个巨大的挑战。先前的方法[17,25]倾向于依赖调度模板来自动优化张量计算，但是它们仅限于自动调整调度模板的参数而已，调度模板中关于原语的使用仍然需要程序员手动编写，而且对于新的算子，仍然需要开发新的模板，这些限制都将影响 AI 芯片上层软件栈（算子库等）的开发效率和迭代速度。

4.1.2 自动化算子编译优化的主要挑战

要实现对于算子层编译的自动化，需要解决三个主要挑战。

优化空间的自动生成：为了使用自动化算法进行优化调度的生成，本文首先需要设计相应的方法自动化生成优化设计空间。优化空间包含原语的组合空间和各个原语参数的调优空间，先前工作依赖模板指定使用哪些原语、如何组合这些原语、以及参数空间如何定义，这种技术方法本质只探索了完整优化空间的子空间。本文则是追求在完整的优化空间中进行调优，这就需要自动生成原语组合的设计空间。然而，原语的组合与具体的算子和硬件都有关系，如何自动化这一过程是一个挑战。

高效通用的搜索算法：在探索完整地原语组合设计空间时，本文需要在合理的时间内搜索出性能较好的设计点，并且这种搜索方法应该对各种算子、各种硬件都通用。先前方法通过在线学习一个性能模型的方式结合启发式算法完成搜索，这种方法往往需要大量的搜索时间并且对于搜索中如何基于当前点得到更好的设计点没有任何启示作用。因此，本文认为如何设计一个高效、通用的搜索算法，对原语组合空间和参数调优空间可以进行搜索是一个挑战。

与图层次编译配合：算子层编译并非孤立的，需要与图层次编译配合。图层次的任务特点往往对于算子层编译有明显的影响作用。典型的例子是在训练任务中，一方面计算图同时包含前向图和反向图，反向图的算子可能缺失，因此需要通过算子层编译技术自动补齐；另一方面，图中进行融合需要得到算子层的支持，需要考虑算子编译优化的能力而不能进行任意融合。此外，在算子层搜索优化方案时需要大量的时间，对于实际系统使用时带来的启动延迟无法忽略，因此需要算子层编译与全图层次的系统调度配合实现系统更高利用率和更低的启动延迟。

4.2 基于循环抽象的编译方案整体结构

为了解决上述算子层编译的挑战，本文提出了 FlexTensor 框架。本文在图 4.2 中展示了 FlexTensor 的结构。FlexTensor 的框架输入是以 Python 编写的贴近数学形式（爱因斯坦求和约定）的张量计算表达式。对于融合过的算子，由于前文块抽象的存在，子块内仍然是表达式，父块是嵌套循环，同样可以作为输入经过 FlexTensor 编译生成代

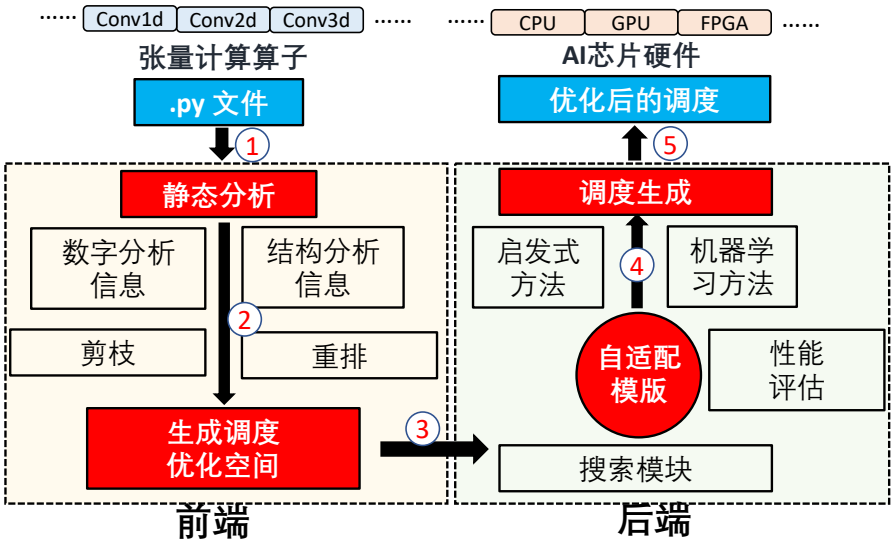


图 4.2　算子层编译框架 FlexTensor 整体结构

码。输入的算子将被作为优化任务注册到 FlexTensor 中等待调优。FlexTensor 的工作流程可以分为两部分：前端和后端。

前端分析使用静态分析技术来提取算子的数字信息以及结构信息，两类信息均已循环为基础。数字信息包括循环的数量、循环的长度，而结构信息包括被融合的算子的数量和每个被融合的算子自身的循环体使用的输入输出张量数量。FlexTensor 依赖于这些信息来生成一个调度空间。在前端分析中，FlexTensor 还会分析调度空间的不同调度选择的结构相似性，并依据此重新排列调度空间来实现对调度空间的压缩和剪枝。

后端分析采用了一种启发式算法结合机器学习算法的方法来搜索较优的调度方案。在庞大的调度空间中，FlexTensor 用启发式方法寻找值得搜索的起始点，然后用机器学习方法指导搜索路径方向。FlexTensor 通过在目标设备上测量性能或使用性能模型以估计性能，然后，FlexTensor 根据硬件特定的特征为不同的硬件设计了定制化的调度方案生成模块，这部分在图中用自适配模板生成代表。最后，FlexTensor 根据后端生成的调度方案为包括 Nvidia GPU、Intel Xeon CPU 和 Xilinx FPGA 在内的不同硬件自动生成底层代码。

为了配合图层次编译需求（特别是训练场景），FlexTensor 进一步提供了三个技术内容：算子级自动求导、融合算子性能估计、搜索与执行交叠调度。具体细节将在本章后续内容详细介绍。

4.3　前端：自动生成调度空间

本节介绍 FlexTensor 的前端。前端主要包括两部分：张量计算的静态分析和调度空间的生成。在静态分析部分，FlexTensor 收集有用的数字信息和结构信息，随后的空

间生成部分使用这些信息来生成调度空间。

4.3.1 静态分析

AI 编译的基本抽象包含张量表达式和有向无环图，对于一个算子，其内部也可能包含多个计算步骤（如图层次编译生成的融合算子节点，内部含有多个计算步骤），所以对于单个算子，也同时需要表示表达式本身和计算图结构。在 FlexTensor 中，算子被表示为一个小规模的图结构，其中节点代表张量表达式，边代表以张量。与图层次编译不同的是，图层次中每个节点都将被当成一个单独的算子生成单独的函数，但是 FlexTensor 中的所有节点将被合并在一个函数中进行代码生成，将他们用不同节点表示的原因是为了方便编译分析。图中的每个节点都有一些输入张量和输出张量，为简单起见，本文只考虑一个输出张量。为了做出区别，本文在描述 FlexTensor 中的算子的内部结构时，用“小图”来指代，对于一般的网络级别的图结构，本文用“图”代指。此外，在前文中还介绍过，每个表达式的循环体中有两种类型的循环：*并行循环*和*归约循环*。这些不同的循环在调度中需要使用不同的处理方式。

给定了一个算子后，FlexTensor 需要知道它对应的小图是如何构建的以及每个节点的嵌套循环是如何组织的。因此需要的信息分为两类：数字信息和结构信息，分别对应于小图的节点和边的特性。详细来说，数字信息包括并行循环的数量（记为 *#sl*），归约循环的数量（记为 *#rl*），并行循环的循环长度（记为 *stc*），归约循环的循环长度（记为 *rtc*）以及循环顺序（记为 *order*）。结构信息包括小图中节点的数量（记为 *#node*），每个节点的输入张量数量（记为 *#in*），每个节点的输出张量数量（记为 *#out*）以及每个节点的消费者节点数量（记为 *#cs*）。

图 4.3 使用矩阵乘法（GEMM）作为一个例子。图 4.3（a）展示了矩阵乘法的小图，节点 *op A* 和 *op B* 分别代表产生张量 A 和张量 B 的算子；张量 A 和张量 B 被矩阵乘法节点使用来产生张量 C。在矩阵乘法节点内部，嵌套循环如图 4.3（b）所示，*循环 i* 和 *循环 j* 没有数据依赖，因此它们是并行循环，而 *循环 k* 有数据依赖，因此它是一个归约循环。图 4.3（c）展示了矩阵乘法示例的数字和结构信息。

4.3.2 调度空间生成

利用静态分析得到的数字和结构信息，FlexTensor 可以枚举调度原语及其对应参数的不同组合来生成调度空间。值得注意的是，原语组合的枚举过程需要遵循特定顺序。具体来说，FlexTensor 首先尝试表 4.1中的 split、reorder 和 fuse 原语，然后是其他原语。这种顺序保证了 FlexTensor 生成的空间中的点结构都是相似的，可以进行后续空间重排操作。

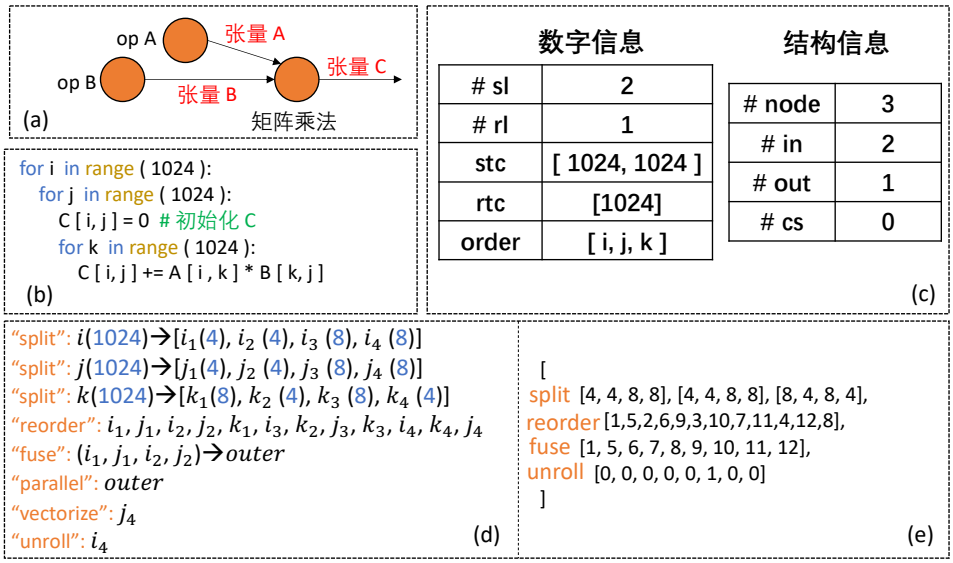


图 4.3　用矩阵乘法为例解释静态分析 (a) 矩阵乘法的小图结构。(b) 矩阵乘法伪代码。(c) 矩阵乘法例子的数字和结构信息。(d) 对于矩阵乘法的调度例子。(e) 把 (d) 中的调度编码为搜索空间中的一个点

调度空间中的每个点都使用一个向量结构进行编码，向量中的每个值代表了特定的原语或参数选择。图 4.3 (d) 中的例子是一个针对矩阵乘法的调度。该调度将矩阵乘法的三个循环分割成十二个子循环，然后重新排序它们，并通过融合生成一个更大的最外层循环。最外层循环被并行化，最内层循环被向量化。图 4.3 (e) 展示了如何将 (d) 中的调度编码为调度空间中的一个点。FlexTensor 通过循环嵌套中的嵌套深度来引用一个循环（1 代表最外层循环）。对于 split 原语的使用，FlexTensor 需要记录分块参数。对于 reorder 原语的使用，FlexTensor 记录循环的新顺序。对于 fuse 原语，FlexTensor 约定未记录的循环（消失的循环）要与它们相邻的外层循环融合。对于 unroll，每个循环对应一个值 0（不展开）或 1（展开）。parallel 和 vectorize 不进行编码，因为 FlexTensor 总是并行化最外层循环并向量化最内层循环。

为了高效地在这个庞大的调度空间中搜索，FlexTensor 首先提出剪枝掉不太可能达到较高性能的点来缩小设计空间，然后利用结构相似性重新排列空间中的点。FlexTensor 通过三种方式来进行空间剪枝：1）限制原语组合的深度；2）修剪参数空间；3）针对不同硬件预先确定某些决策。首先，一些原语组合可以递归使用，这将导致无穷的空间大小（例如，单个循环可以递归使用 split 和 fuse），为了避免这种情况，FlexTensor 设置了原语组合深度的阈值（例如，对一个循环最多使用 split 和 fuse 四次）。其次，对于参数剪枝，FlexTensor 主要考虑减少 split 参数的，因为 split 参数占据了大部分的参数空间。论文发现，在大多数情况下，可整除的 split 参数是高效的，而其他 split 选择的结果较差。因此，FlexTensor 将每个循环的 split 参数限制为可整除的因子。最后，根据

之前的研究[17,40,156]，FlexTensor 会选择为不同硬件固定某些调度决策。例如，在 CPU 上，只并行化最外层循环（融合后）并向量化最内层循环；在 GPU 上，将外层循环绑定到线程块，内层循环绑定到线程；在 FPGA 上，使用三阶段流水线设计等。

除了剪枝之外，FlexTensor 还考虑重新排列调度空间以方便后续搜索算法设计。重新排列过程在剪枝之后，这样保证了剪枝的点将不会又出现在搜索过程中。先前工作中，调度空间是使用一维列表表示的，每次搜索都在这个列表中随机采样。然而，系统化的探索需要利用搜索空间中的局部区域的特性（理想情况下需要知道沿哪个方向搜索最优），而不是随机采样。一维列表的局部性较差，因为只有两个方向可以沿着搜索，并且相邻的点无法保证有任何的相似性，无法提供局部信息。因此，本文提出将原始的一维列表转换为高维空间。在高维空间中，局部区域有更多的点。例如，考虑将循环长度为 L 的循环分块为 N 个子循环，通过分解 L 可以得到不同的分块选择，每个分块选择可以表示为 $[f_1, f_2, ..., f_N]$，其中 $f_1 \times f_2 ... \times f_N = L$。FlexTensor 为每个点额外增加一个二维方向的概念，通过增加在调度空间中的方向，FlexTensor 可以将一维列表重新定义为一个 $\frac{N\times(N-1)}{2}$ 维空间。对于任何点 $p = [f_1, f_2, ...f_N]$，在方向 (i, j) 的邻近点是 $[g_1, g_2, ...g_N]$，其中 $g_i > f_i, g_j < f_j$ 并且 $\forall k \neq i, k \neq j, g_k = f_k$。这样，相当于对原来的一维空间进行了重新排列，在这种重新排列中，邻近点将具有相似的结构，这些临近点极可能拥有相似的性能，从而提供了一种局部性信息。

4.4 后端：自动搜索调度空间

FlexTensor 的后端负责搜索调度空间并生成一个优化的调度方案。本文为 FlexTensor 设计的搜索技术结合了启发式和机器学习方法，同时，本文的调度针对不同的硬件进行了特殊的定制，从而做到对多硬件都能使用。

4.4.1 结合启发式算法和机器学习算法搜索

FlexTensor 前端生成的调度空间十分巨大（例如，体量通常大于 10^{11}），这使得穷举所有点进行搜索不现实。为了实现高效的搜索，本文将启发式方法与机器学习方法相结合。形式化地说，本文将搜索空间表示为 G，G 中的每个点都表示为一个向量（前面章节介绍）。前文介绍过，FlexTensor 前端会将原始的调度空间重新组织成了一个更高维度的空间，并且调度空间中的每个点都有多个邻近点。对于 G 中的每个点 p，其相邻点与 p 仅在 p 的编码的一个或两个位置的值不同（例如，仅修改分块参数或改变一个值从 0 变为 1 以启用/禁用一个调度原语）。

在 FlexTensor 的后端搜索过程中，FlexTensor 维护一个集合 H，H 中的每个点已经被搜索而且评估过，FlexTensor 将一个性能值 E 与每个点关联起来。为了获得性能

值，FlexTensor 可以在目标设备上运行生成的代码来收集实际性能，或者使用分析模型获得估计性能，正如前述章节中对图层次编译使用的技术那样。在探索过程中，需要做出两个关键决策：首先，选择 H 中哪个点作为搜索中每一步的起点；其次，给定起点 p，沿哪个方向 d 移动它以在 G 中获得一个新点。对于第一个决策，本文使用启发式方法；对于第二个决策，本文使用机器学习方法。

启发式方法：本文的启发式方法基于模拟退火[113]。对于 H 中的一个点 p，其性能值表示为 E_p。在搜索过程中，FlexTensor 保持追踪 H 中的最佳点（具有最高 E 值），本文将其性能值表示为 E^*。FlexTensor 选择 H 中的一个点 p 作为下一步的起点的概率为 $\exp^{-\gamma\frac{(E^*-E_p)}{E^*}}$，其中 γ 是一个超参数。E_p 越接近 E^*，p 被选为起点的可能性就越大。用这种方式，FlexTensor 首先通过随机采样的方式采样若干初始点候选，然后计算它们被选择的概率，之后生成随机值（抛硬币）决定哪些点被选作起始点。注意，FlexTensor 可以同时选择多个起点进行搜索（beam search）。

机器学习方法：在搜索时，当本文选择了一个点 p 作为起点后，在高维调度空间中就有许多可能的方向可以推进搜索。如果 FlexTensor 尝试所有的方向，探索过程将会非常漫长。相反，FlexTensor 希望在 G 中尽量只尝试 “最佳” 的方向 d。这里所谓的 “最佳” 的方向，就是能带着 FlexTensor 找到最优点的方向。

为了找到“最佳”方向，本文设计了一种基于强化学习的方法来预测“最佳”的方向。找到某个点的最佳方向的问题类似于找到某个状态的最佳动作，可以类比强化学习对问题的建模，因此本文决定使用强化学习算法[157]来解决这一问题。在强化学习理论中，有一个状态集 S，一个动作集 A，以及一个奖励函数 r。对于一个状态 $s \in S$，采取动作 $a \in A$ 会得到奖励值 $r(s,a)$。每次执行动作时，当前状态 s_t 变为新状态 $s_{t+1} = f(s_t, a_t)$，其中 t 是当前时间步，f 是变换函数。强化学习的目标是能够保持找到 “最佳” 的动作 a_t 来最大化 $\sum_{t\leq T} r(s_t, a_t)$，其中 T 是最大尝试步数。相应地，本文将点 p 视为状态，G 中的方向 d 视为动作，如果本文沿方向 d 从 p 移动，就会得到一个新点 e。本文的目的是始终找到一系列好的方向 d，最终达到最优点。因此奖励值可以设计为 $r(p,d) = E_e - E_p$，其中 E_e 是点 e 的性能值，E_p 是点 p 的性能值，从而 $\sum_{t\leq T} r(s_t, a_t) = E_{s_T}$。最大化 E_{s_T} 相当于找到最优点。在实际实现时，本文为了稳定性，将奖励值进行归一化，所以 $r(p,d) = \frac{E_e - E_p}{E_p}$。

求解强化学习问题的算法有很多，在本文中，本文使用 Q-Learning 算法来最大化目标函数。Q-Learning 是在[158]中提出的一种特殊强化学习算法。Q-Learning 的主要思想是为每个动作（在本文的问题中是方向）分配一个值，这个值称为 Q 值。Q 值越大，动作越好。Q-Learning 的关键部分是计算 Q 值。为此，本文设计了一个神经网络来预测 Q 值。更明确地说，本文构建了一个由四个全连接层组成的网络，并使用 ReLU[14]激

活函数。本文的算法可以从多个起点开始，起点的数量可以由使用者设置。对于每个起点 p，运行神经网络（使用 p 作为输入特征）以获得从 p 可达的所有方向的 Q 值。选择 Q 值最大的方向，并沿这个方向从 p 移动到邻近点 e（即单步）。搜索过程可以涉及多个步骤，这可以由用户设置。在搜索过程中，FlexTensor 记录访问过的点以避免重复搜索，并且没有回溯。FlexTensor 评估新点 e 并记录结果为一个元组（p, e, $\frac{E_e-E_p}{E_p}$）。

Q-Learning 是一个在线学习算法，所以神经网络训练是在搜索过程中进行的。训练过程类似于[159]。为了训练本文的四层网络（记为 X），需要创建另一个具有相同结构的网络（记为 Y）。开始时，两个网络用相同的参数初始化，并且因为参数未经训练，它们一开始的时候并不准确。训练过程周期性地进行（每五次搜索训练一次），使用收集到的数据来训练网络 X。对于每个数据元组（p, e, $\frac{E_e-E_p}{E_p}$），首先使用公式 $target = \alpha \times \max Y(e) + \frac{E_e-E_p}{E_p}$ 计算目标值（即标签），其中 α 是一个超参数；然后使用均方误差 $(X(e) - target)^2$ 作为损失值，并利用 AdaDelta 算法[160]对网络参数进行优化，优化的网络是 X；最后，每隔一段时间，X 的参数被复制到网络 Y 作为备份，网络 Y 的存在则是为了训练过程数值更稳定[159]。

性能评估方法：在搜索过程中评估搜索到的点的性能值有两种方法。一种是通过在目标设备上执行程序来测量真实性能，另一种是使用分析模型来估计性能。真实测量返回精确的性能值，因此能给出准确的起点和搜索方向。此外，它易于实现并且可以移植到不同的设备上。然而，如果编译和执行开销比较大，这种方法可能需要很长时间才能完成搜索。另一种方法使用分析模型，搜索速度就非常快，这在目标平台不可用或真实测量需要很长时间完成时非常有用。然而，构建分析模型非常不容易，因为它依赖于应用、架构和编译层面的许多参数，这些参数对在不同的硬件平台上差异会很大。

FlexTensor 选择在 CPU 和 GPU 上使用测量方法，因为在这些平台上的编译和测量开销相对较小（≤ 1s）。然而，在 FPGA 上，将高层次代码综合为网表可能需要数小时[161]。这个漫长的综合开销导致在 FPGA 上不能选择进行实际测量。因此，FlexTensor 使用来自先前工作[161-162]的性能模型来预测代码在 FPGA 上执行的性能。FPGA 上的性能可以通过将并行处理的轮数与一轮的执行时间相乘来估计。一轮的执行时间受到计算、数据读取和数据写入三者中最长时间的限制。这里给出简化的性能模型：

$$Execution\_time = \frac{workload}{\#PE} \times \max(R, C, W) \tag{4.6}$$

其中 R 是数据读取时间，W 是数据写入时间，C 是计算时间，$\#PE$ 是处理单元（PE）的数量。该模型符合本文为 FPGA 设计的三阶段流水线的执行，在图 4.4(c)中给出了流水线的示意图。最终的时间开销由计算量大小、流水线中最长阶段（$\max(R, C, W)$）

```
1: graph ← get_graph(operator)
2: node_lst ← post_order_traverse(graph);
3: op_config_lst ← [];
4: for every node in node_lst do
5:     op_config ← Schedule_for_node(node);
6:     op_config_lst.append(op_config)
7: end for
8: graph_config ← Schedule_for_graph(graph);
9: config ← Config(op_config_lst, graph_config);
10: return config
```

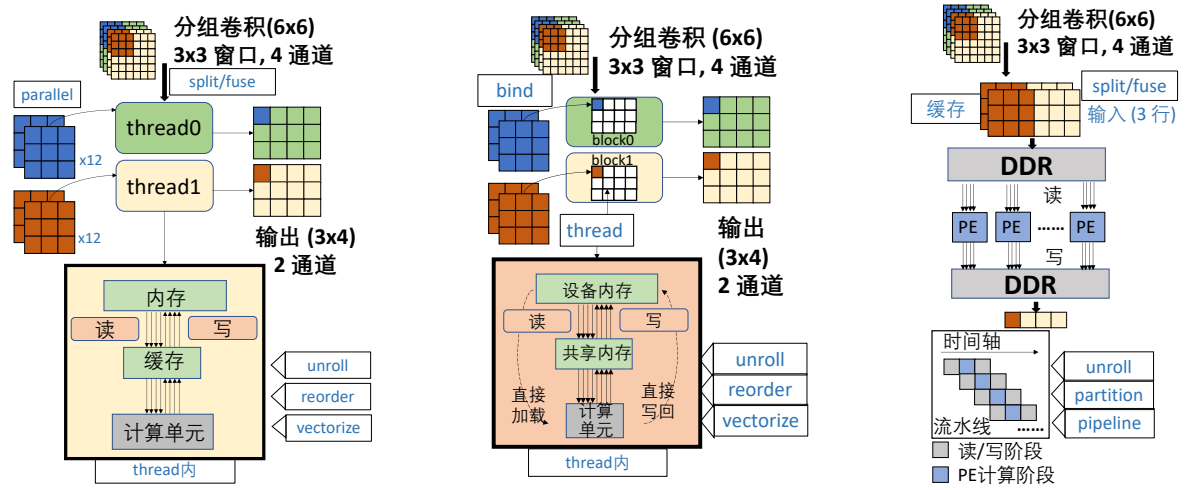
Algorithm 3: FlexTensor 调度生成流程

和 PE 的数量（$\#PE$）决定。

4.5　调度模板生成

在 FlexTensor 通过搜索找到一个好的调度方案之后，就需要使用表 4.1中列出的原语在目标设备上实现该调度。FlexTensor 可以根据给图和节点的优先级选择不同的顺序来生成调度。可选顺序有*自下而上顺序*、*自上而下顺序*和*递归顺序*。自下而上顺序首先为小图中的每个节点生成调度，然后为整个图生成调度。自上而下顺序则相反。递归顺序是重复自下而上顺序或自上而下顺序多次。FlexTensor 采用自下而上顺序，因为这种方案在大多数情况下更加有效。算法 3展示了 FlexTensor 调度生成流程。算法通过第 1 行的 *get_graph* 函数获取算子的小图结构。并且在第 2 行以后序遍历以获得每个节点。通过在第 5 行调用函数 *Schedule_for_node* 为每个节点实现调度，生成调度的方式是先生成调度原语组成的模板，再填入每个原语的参数。然后在第 8 行使用函数 *Schedule_for_graph* 为小图实现调度。接下来，论文将详细解释对每个不同的目标硬件如何生成调度模板。

针对 CPU 的调度模板：图 4.4(a)展示了一个分组卷积的 CPU 调度模板生成例子。对于 CPU，寄存器分块和向量化至关重要。通过多级分块[163]可以利用寄存器缓存临时数据。多级分块使用 split 原语和 reorder 原语递归地产生一系列小块，这些小块可以放进寄存器中。这些小块的大小需要根据搜索得到的分块参数确定，并行循环和归约循环都会参与分块，以充分利用数据局部性。为了发掘并行性，FlexTensor 动态地将几个外层循环融合成一个最外层循环并将其多线程并行化。向量化优化使用在最内层循环，以利用 CPU 的向量化指令。由于最内层循环的循环大小由分块参数决定，向量化的长度就也是可调整的，因此 FlexTensor 可以动态决定向量化长度以适应不同的指令集，如 AVX2 和 AVX512。在图 4.4(a)的例子中，FlexTensor 沿通道维度的分割产生了两个子卷积，每个线程处理一个子卷积，线程内进行循环展开、循环重排序和循环向量化的优化。



(a) 针对 CPU 的调度模板生成示意图

(b) 针对 GPU 的调度模板生成示意图

(c) 针对 FPGA 的调度模板生成示意图

图 4.4　FlexTensor 的调度模板生成示意图。在这里用的算子例子是分组卷积

针对 GPU 的调度：图 4.4(b)展示了一个 GPU 调度模板生成的例子。对于 GPU，线程块/线程分配和共享内存的配置对性能是非常重要的。线程块和线程的分配都反映在分块参数中，与多级分块相对应。分块过程可以产生许多子循环。外部循环绑定到线程块，一些内部循环绑定到线程，以便每个线程块能处理一小部分计算。搜索阶段可以探索不同的分块参数和循环顺序，通过这种方式搜索不同的线程块/线程分配策略。对于共享内存配置，每个线程块总是在计算之前先将数据加载到共享内存中。每个线程块中使用的共享内存大小由其内部循环的大小和相应的访存范围决定。为了进一步提高性能，FlexTensor 总是使用一组寄存器来存储计算的中间结果，并且只在计算完成后才写回结果。对于图中分组卷积的例子，使用两个线程块，每个线程块有十二个线程。对于内部循环，使用循环展开和重排序来进一步提升性能。

针对 FPGA 的调度：图 4.4(c)展示了 FPGA 的调度模板生成。本文利用广泛使用的三阶段粗粒度流水线架构在 FPGA 上构建算子加速器。这个三阶段流水线由三部分组成，分别是数据读取、计算和写入阶段。一方面，数据读取和写入阶段的调度是基于 DDR 带宽、片上内存缓冲区大小和传输数据的总大小来构建和配置的。另一方面，计算流水线阶段的调度是基于用于并行数据处理的 DSP 资源以及用于片上内存缓冲区的 BRAM 内存的情况来确定的。

4.6　与图层次编译结合

正如本文前述的那样，算子层编译不能孤立于图层编译。首先，对于 AI 算法，除了推理之外，训练任务也同样重要。但是很多新兴网络会使用新算子，这些新网络在

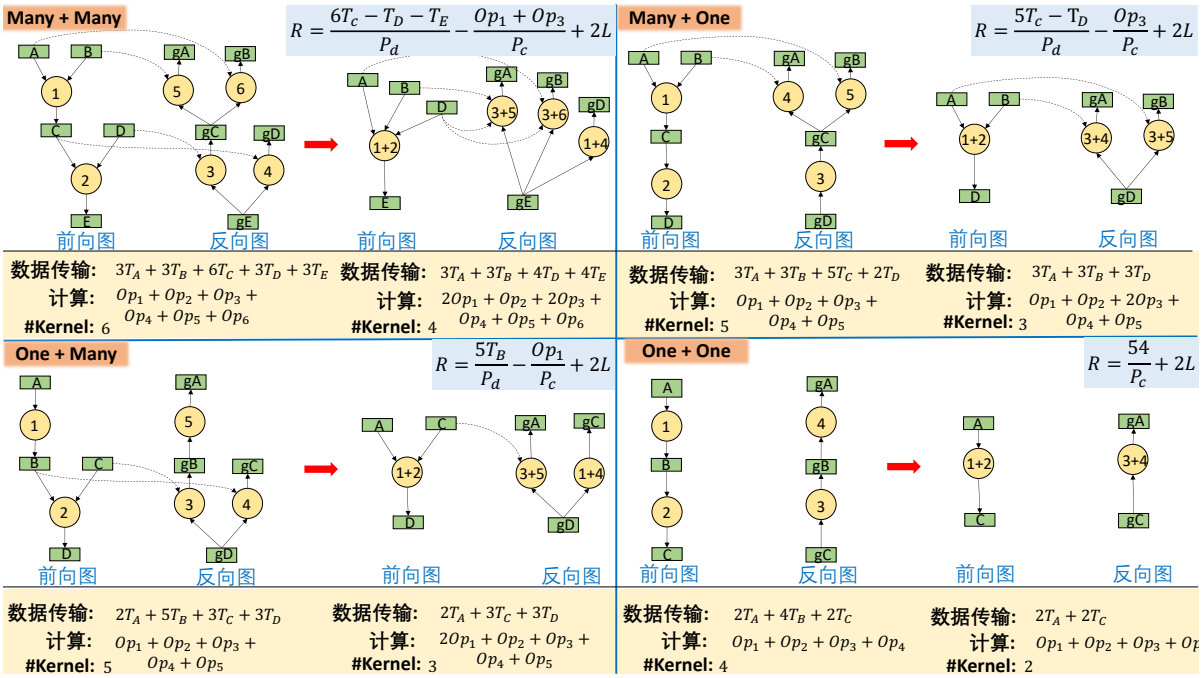


图 4.5　FlexTensor 分析融合算子的收益和代价计算方法

训练时需要反向计算的算子计算梯度，新算子的反向传播算子一般需要开发者自行推导、实现、和优化，这需要对网络求导、算子开发和优化都有很深厚的知识积累。为了降低开发门槛，编译器需要在算子层提供自动求导支持。本文这里所说的自动求导指的是从前向计算算子自动推导反向计算算子，如从卷积算子自动推理出反卷积算子，而不是许多 AI 框架如 PyTorch[92]中提供的图层次自动求导（链式法则）。其次，图层编译的重点在于算子融合，上一个章节讨论了计算密集型算子融合，但是对于涉及访存密集型算子的融合，融合后的算子如何生成代码仍然需要算子层支持，根据算子层提供的评估和反馈信息来判断图层次编译的融合是否有性能收益，从而影响融合策略和融合算子的生成。最后，算子层为每个算子搜索调度策略的时候需要大量的时间（数十分钟），如果等待整个网络所有算子的调度策略搜索完毕才能执行代码，需要等待若干小时的时间，这对于实际应用是不可忍受的。为了解决这个问题，本文进一步提出将调度搜索和全图执行交叠的方法来缩短系统启动延迟。

4.6.1　算子自动求导

为了解释算子自动求导的方法，首先需要将张量表达式抽象更详细地写出来，显式写出所有的输入张量、下标、操作函数：

$$
\begin{aligned}
&Out[x_1,\dots,x_K] = \mathbf{F}_{R=\{r_1,...,r_L\}}(\\
&In_1[f_1^1(x_1,...,x_K,r_1,...,r_L),\dots,f_{M_1}^1(x_1,...,x_K,r_1,...,r_L)],\\
&In_2[f_1^2(x_1,...,x_K,r_1,...,r_L),\dots,f_{M_2}^2(x_1,...,x_K,r_1,...,r_L)],\\
&\dots,\\
&In_N[f_1^N(x_1,...,x_K,r_1,...,r_L),\dots,f_{M_K}^N(x_1,...,x_K,r_1,...,r_L)],\\
&loops = \{x_1,...,x_K,r_1,...,r_L\})
\end{aligned}
\tag{4.7}
$$

其中 Out 是一个 N 维输出张量，In_i（$1 \le i \le N$）是一个 M_i 维输入张量，$\mathbf{F}$ 是计算操作，等价于之前公式中写的 $Accum(Op(.))$。$R = r_1,\dots,r_L$ 是表示归约循环（如果没有归约则为空），而 f_j^i（$1 \le i \le N, 1 \le j \le M_i$）是关于 $x_1, x_2, \dots, x_K$ 和 $r_1,\dots,r_L$ 的函数，用于产生输入张量的相应访问下标。在大多数情况下，f_j^i 是循环变量的仿射变换。

对于上述表达式中的每个输入 In_i，计算它梯度的表达式是：

$$
\begin{aligned}
&dIn_i[z_1^i,\dots,z_{M_i}^i] = \mathbf{H}_{R'=\{r'_1,...,r'_P\}}(\\
&dOut[g_1(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P),\dots,g_K(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P)],\\
&In_1[h_1^1(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P),\dots,h_{M_1}^1(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P)],\\
&\dots,\\
&Im_N[h_1^N(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P),\dots,h_{M_K}^N(z_1^i,\dots,z_{M_i}^i,r'_1,...,r'_P)]\\
&)
\end{aligned}
\tag{4.8}
$$

其中 dIn_i 是 In_i 的梯度张量，$dOut$ 是 Out 的梯度张量，$\mathbf{H}$ 是对应于 $\mathbf{F}$ 的梯度函数，对于初等函数，$\mathbf{H}$ 的求解规则和普通函数求导一致，对于超越函数，则需要根据函数的定义从 $\mathbf{F}$ 推导 $\mathbf{H}$（比如 exp 函数的导函数仍然是 exp 函数），$R' = r'_1,\dots,r'_P$ 是新的归约循环，g_p 和 h_t^s（$1 \le p \le K, 1 \le s \le N, 1 \le t \le M_s$）都是关于 $z_1^i,\dots,z_{M_i}^i$ 和 $r'_1,\dots,r'_P$ 的函数，用于生成梯度计算时数据访问下标。自动求导的核心问题是推导出索引函数 g_p 和 h_t^s（$1 \le p \le K, 1 \le s \le N, 1 \le t \le M_s$）并确定 $r'_1,\dots,r'_P$ 的循环迭代范围。本文认为，当正向表达式中使用的索引函数 f_j^i 是仿射变换时，这个问题可以通过一些线性代数操作解决。这个求解问题表达为解以下整数线性方程组

$$
\begin{aligned}
&f_1^i(x_1,...,x_K,r_1,...,r_L) = z_1,\\
&\dots,\\
&f_{M_i}^i(x_1,...,x_K,r_1,...,r_L) = z_{M_i}
\end{aligned}
\tag{4.9}
$$

其中 $x_1,\dots,x_K$ 和 $r_1,\dots,r_L$ 被视为未知数（整数未知数），而 $z_1,\dots,z_{M_i}$ 被视为常数。

如果能解出方程组中的未知数，那么就可以简单地将前向计算公式中的 $x_1,\ldots,x_K$ 和 $r_1,\ldots,r_L$ 用求出的解替换，并将其重新组织成符合反向计算的形式。以前的工作[164]虽然也讨论这个问题的求解，但是他们假定 f_j^i 总是 $x_1,\ldots,x_K$ 和 $r_1,\ldots,r_L$ 的线性组合，这个假设对于真实使用的 AI 算子还是太严格了，导致实用性比较差。比如，像 depth2space[165]这样的重要算子就不符合这样的假设，因为这个算子计算时访存下标存在整数除法和取模运算，它们不是单纯的线性变换。

本文提出了新的方案解决这一问题。用新变量替换 $f_1^i,\ldots,f_{M_i}^i$ 中的所有非线性的子表达式，这样剩余的表达式仍然是线性的。然后用整数矩阵的 SVD 分解（构造史密斯正规型）方法来解这个线性方程组，求解之后再单独求解整数除法和取模运算子表达式的解，并将原来被替换的子表达式带回结果中。详细来说，如果一个整数除法子表达式 x/V（V 是常数）被替换为 s_1，为了解决这个替换，首先要找到它的对称子表达式：取模运算子表达式 $s_2 = x\%V$，因为它们一起可以合并解出 $x = s_1 * V + s_2$。如果在原公式中找不到这样的除法和模运算对，则意味着结果表达式需要引入新的自由变量作为归约循环（因为原计算相当于存在广播语义，对称地，反向计算就要规约），如 $x = s_1 * V + r$，其中 r 是一个新的归约循环。最后，通过循环范围分析技术可以推断新增归约循环的迭代范围，把所有的结果按照反向传播表达式的格式汇总，这样就完成了方程求解。

4.6.2　融合算子性能估计

在图层次进行融合时，需要确认算子层生成融合算子代码时候可行且能得到性能收益，为此，需要算子层编译提供分析功能，分析计算和数据传输开销之间的平衡。这种分系统能需要放在算子层，是因为只有算子层清楚算子调度和优化的能力如何，从而判断生成代码的性能如何。特别是针对训练场景，之前的融合工作如 TVM[17]中的 Relay[37]和 TASO[18]都没有考虑到训练时候的融合和推理时候的融合是很不同的，它们只关注推理任务而忽略了训练。对于训练任务，有两种计算图：前向和后向。优化前向图和优化后向图是对偶问题。如果前向图中存在一个张量，在后向图中就会有一个操作来计算这个张量的梯度，反之亦然。此外，前向图和后向图通过许多中间结果相互连接。例如，为了计算卷积层权重的梯度，需要这个卷积层的原始输入，这可能是它的前面层（如 ReLU[14]层或另一个卷积层）的中间结果。所以，如果编译器在前向图中融合两个算子并消除一个中间张量（减少数据传输开销），就需要为后向图中需要这个中间张量作为输入的求梯度的算子重新生成这个张量（带来重计算开销）。同时，这个被消除张量的算子也应该被融合到计算梯度的算子中，因为没有必要把这个张量的梯度显式计算并存储出来。总之，前向图和后向图之间的这种耦合现象使得训练任务

的融合优化更加困难。

FlexTensor 通过分析每个融合可能性的性能收益和开销来选择哪些算子可以融合。为了简化问题，FlexTensor 考虑一次融合两个具有单一输出的算子。注意这里本文讨论融合时，与前文图层次编译技术内容中介绍计算密集型算子链融合不同，这里需要考虑的是更通用的融合问题，也就是涉及到访存密集型算子融合。根据输入数量（入度）可以将算子分为两类。对于只有一个输入的算子，称它们为 *One*，对于有多个输入的算子，称它们为 *Many*。*One* 和 *Many* 之间有四种组合：*Many*+*Many*，*Many*+*One*，*One*+*Many*，和 *One*+*One*。分别分析这四种组合下融合的性能收益和代价，就可以分析性地评估各种融合选择的优劣。融合的收益来自于对中间结果的数据搬运的减少，这个量可以用张量大小来估计；融合的代价则是重计算，由于融合可能造成反向图中需要额外的计算恢复被融合掉的中间结果，总体的计算量就可能增加。选择好的融合策略，本质上是对数据搬移减少和重计算增加的权衡。本文使用 $T_A, T_B, ...$ 来表示数据传输量，使用 $Op_1, Op_2, ...$ 来表示计算量。gA 是张量 A 的梯度，其他张量类似。#Kernels 显示生成的函数（Kernel）数量。利用这些符号，就可以定量评估不同融合组合的收益和代价。在图 4.5中，总结了四种融合组合下各自的数据搬移量减少，重计算增加量，以及融合造成的函数（Kernel）数量变化。通过比较这些融合情况的数据传输量、计算量和函数数量，可以得到以下结论：

1. *One* + *One* 融合总是有益的。可以减少 $4 \times T_B$ 的数据传输开销，而不增加任何计算量。这种融合可以应用于 ReLU[14]、Padding、Reshape 等算子。
2. *One* + *Many* 和 *Many* + *One* 融合决策很可能是有收益的，因为它们在增加一个算子的重新计算成本下，减少了 4-5 个张量的传输需求。然而，这种融合决策实际是否有利应该通过在给定具体张量形状前提下精确计算来判断。
3. *Many* + *Many* 融合（如卷积 + 卷积）增加了一个前向操作和一个后向操作的计算量，数据传输量是否减少取决于中间张量的大小（如果 $6T_C > T_D + T_E$，则数据传输量减少），这部分的判断就不由 FlexTensor 判断，而是依赖前述的图层次编译分析决定了。

在编译期间，FlexTensor 采用一个简单的收益函数来判断融合决策是否有利。预测真实的运行时行为很难，但收益函数的原则是筛选出注定性能差的融合决策。因此，收益函数使用峰值性能对每个融合决策给出乐观评估，如果在乐观假设下的收益仍然是负的，FlexTensor 就可以直接放弃当前融合计划。在图 4.5中也列出了四种融合组合下的收益函数，用 R 表示。收益函数的公式中使用 P_d 表示片外存储和片上存储之间的峰值数据传输带宽，使用 P_c 表示设备的峰值计算性能。此外还使用 L_{start} 表示算子平均启动开销。由于 AI 芯片上运行函数往往需要额外的运行时开销，融合会造成函数数量

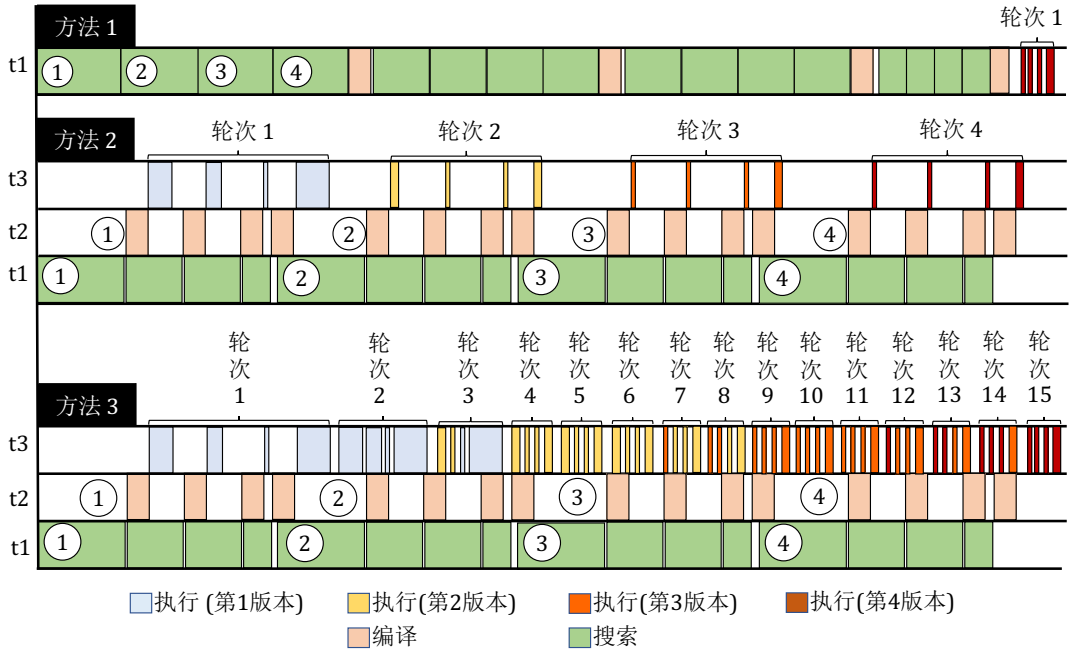


图 4.6　对比不同的系统设计方法：方法 1 是先编译再执行，方法 2 是同步的交叠编译和执行，方法 3 是异步地交叠编译和执行

的改变，统计算子启动开销就可以把函数数目变化带来的新能变化考虑进来。P_d、P_c 和 L_{start} 是设备相关的的，可以使用多种测试方法[166]得到这些数值。收益函数公式如下：

$$R = \frac{\Delta T}{P_d} + \frac{\Delta C}{P_c} + \Delta K \times L_{start} \tag{4.10}$$

其中 ΔT 是减少的数据传输量，ΔC 是减少的计算量，ΔK 是减少的算子数量，这些变化值可以是负的。比较四种可能的融合情况，可以发现，*One* + *One* 的收益总是正的，而对于其他情况，究竟是否有收益取决于具体的问题大小。FlexTensor 在使用收益函数的时候，会将被融合的图中的算子形状作为输入，在线计算出收益值。通过枚举不同的融合可能性，生成融合的搜索空间，然后利用收益函数选择预估性能最好的融合方案进行代码生成。

4.6.3　搜索与执行交叠调度

一般来说，FlexTensor 为一个算子搜索调度所用的时间是 20 分钟左右，根据调度编译一个算子生成代码的时间量级在几百毫秒，执行生成的代码需要的时间则在几毫秒到几百毫秒不等，所以如果在一个系统中（类似 PyTorch 这样的全栈框架，或者类似 TVM 这样的全栈编译器），总是先等待搜索、编译，再执行的话，就需要等待若干小时甚至几天（一个网络可能有几十到几百个算子），对于离线推理任务，这是可以接受

的，但是对于一些在线任务，或者对于训练任务（总是需要调整网络的某些部分，调整超参数以做试验），都是难以忍受的启动延迟。为了解决这个问题，本文提出了一种将搜索、编译、执行交叠起来的系统调度方法。首先，需要分析不同设计方法下的系统性能。

静态模式：在静态模式下，每个算子会先搜索、编译，然后再执行代码。本文在图 4.6的方法一对静态模式给出了示意图，在这个时间轴上，假设网络只有四个算子需要生成代码，每个算子搜索的过程被均分为四个阶段（比如，一共需要进行 1000 次搜索，每个阶段搜索 250 次），图中对第一个算子的四个阶段标记了 1,2,3,4。类似的，后三个算子也需要经历四个轮次的搜索。搜索之后，每个算子都会经历编译代码生成。直到这些都做完了，才会开始进行执行，在这个示意图中，启动时间很长，所以在有限的时间轴上只画了一个轮次的执行。

同步交叠模式：在同步交叠模式下，每个算子的每个搜索阶段之后都可以进行一次编译和执行，并且在前一个算子进行编译和执行时，后续的算子可以开始搜索，这样可以用后续算子的搜索时间掩盖之前算子的执行时间。尽管每个算子在四个阶段全搜索完成之前得到的代码版本性能不是最好的，但是这些次优的算子执行时间也是远小于搜索时间的。实践中，可以通过把搜索轮次分的更细让延迟掩盖得更好。在图 4.6中的方法 2，展示了同步交叠模式示意图，可以发现，在相同的时间窗口内，同步交叠模式可以进行更多轮次的全图执行，尽管前几次执行计算延迟比较大，但是不影响计算结果的正确性，比如对训练场景，模型此时已经完成了若干次训练迭代步骤。

异步交叠模式：尽管同步交叠模式已经让整体的瓶颈到了搜索上，看起来已经没有优化空间。但是可以注意到，其实没有必要等待整个网络的全部算子都更新到新的版本才开始全图执行，可以更细粒度地控制代码版本，全图中一些算子可以用版本较旧的代码先计算，另一些算子可以提前用上新版本的代码。依据这样的思想，可以在等待搜索和编译的间隙让全图执行从不间歇，也就是执行仅等待首次搜索和编译，而不再等待后续的搜索和编译，这种方式被称为异步交叠模式。如图 4.6中的方法 3 所示，可以在有限的时间窗口内塞入更多的执行轮次。类比到训练任务中，就可以在有限的时间内完成更多的训练迭代，执行不再受限于编译，启动延迟也被大大降低。

4.7 实验评估结果

4.7.1 实验设置条件

本文首先在不同的 AI 硬件上评估 FlexTensor 对于各种算子的实际性能。测试的算子的详细信息如表 4.2所示。这些算子在多种算法、多个领域都有广泛的应用。例如，GMV、C2D、T2D 用于图像处理[15,169]；GMM、C1D、T1D 用于自然语言处理[170]；C3D、

表 4.2　实验中使用的算子的详细介绍

算子		分析结果		库支持		FLOPs	精度	测例数目
全称	简写	#sl/rl	#node	CPU	GPU			
矩阵向量乘	GMV	1/1	1	MKL	CuBlas	16K-1M	float32	6
矩阵乘法	GMM	2/1	1	MKL	CuBlas	32K-8.6G	float32	7
双线性	BIL	2/2	1	MKL	CuBlas	1G	float32	5
1 维卷积	C1D	6/2	2	MKL-DNN	CuDNN	50M-200M	float32	7
1 维反卷积	T1D	9/2	3	PyTorch	CuDNN	50M-200M	float32	7
2 维卷积	C2D	8/3	2	MKL-DNN	CuDNN	77M-3.7G	float32	15
2 维反卷积	T2D	12/3	3	PyTorch	CuDNN	77M-3.7G	float32	15
3 维卷积	C3D	10/4	2	PyTorch	CuDNN	77M-6.6G	float32	8
3 维反卷积	T3D	15/4	3	PyTorch	CuDNN	77M-6.6G	float32	8
分组卷积	GRP	4/3	2	MKL-DNN	CuDNN	20M-900M	float32	14
深度可分离卷积	DEP	4/3	2	MKL-DNN	CuDNN	250K-3.6M	float32	7
空洞卷积	DIL	4/3	2	MKL-DNN	CuDNN	100M-1.2G	float32	11

表 4.3　来自 YOLO v1 的 15 种不同形状的卷积

层	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
C	3	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024
K	64	192	128	256	256	512	256	512	512	1024	512	1024	1024	1024	1024
H/W	448	112	56	56	56	56	28	28	28	28	14	14	14	14	7
k, st	7,2	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	1,1	3,1	3,1	3,2	3,1

T3D 用于视频处理[171-172]；DEP[173]用于移动和嵌入式设备[174]上的图像处理。

不同的算子有不同数量的并行循环和归约循环，它们的小图结构包含不同数量的节点。例如，T1D 在卷积之前需要图像扩展和填充操作，因此其小图有三个节点。表 4.2中的**分析结果**列提供了每个算子的结构和数字信息的细节。本文为每个算子提供了多个测例，这些测试用例在输入大小和计算量上有所不同。**测例数目**和 **FLOPs** 列分别显示了测试用例的数量和计算量。

本文将 FlexTensor 与 PyTorch[92]（版本 1.0）中的手动调优库函数进行性能比较。PyTorch 集成了多个库函数，如 MKL、MKL-DNN、CuBlas 和 CuDNN。对于没有支持或支持较差的算子，PyTorch[92]自己也有算子库实现（ATen 库）。在 GPU 上，通过设置 *torch.backends.cudnn.enabled = True* 可以启用 PyTorch 的 CuDNN 后端，本文为 PyTorch 使用 CuDNN v7。当 CuDNN 被禁用时，PyTorch 在 GPU 上使用 ATen 库[175]。本文使用单精度浮点数，批处理大小为 1，测试场景首先是推理。对于 GPU 实验，本文使用 Nvidia V100（16GB 设备内存）、P100（16GB 设备内存）和 Titan X（Pascal）。对于 CPU 实验，本文使用 Intel Xeon E5-2699 v4 CPU。对于 FPGA 实验，本文使用 Xilinx VU9P FPGA。

除了单个算子的测试，本文也要进行与图层次配合的全图执行性能测试。测试的网络包括卷积神经网络（CNN）和循环神经网络（RNN）。一些网络包含新的算子，这些算子在像 PyTorch 这样的框架中都需要用别的算子拼凑才能实现，因此性能不好。其中，CapsuleNet[176-178]使用了两层胶囊卷积以及它们之间的动态路由算子[177]；ShuffleNet[179]不

表 4.4　全图执行性能测试对比对象

名称	配置
PyTorch	版本：1.10，使用 CUDA Graph[167] 支持能力。
TensorFlow	版本：2.4，使用 XLA[28] 和 Eager 模式。
CuDNN	版本 8.0，使用 CUDA 版本 11.0.
AutoTVM	版本：0.8，使用 XGBoost[168] 帮助调优。

仅包含深度卷积，还包含通道转换（Shuffle）算子；MI-LSTM[180]、SCRNN[181]、sub-LSTM[182]和 LLTM[183]是 LSTM[4]的变体。本文还测试了如 ResNet[2]、MobileNet[184]和 Bert[87]这样比较常见的模型。所有性能实验都在 Nvidia Tesla V100 GPU 上进行。本文将 FlexTensor 与 PyTorch[92]、TensorFlow[91]、CuDNN[84]和 AutoTVM[25]进行比较，这些框架的设置如表 4.4所示。实验中，本文测试不同批处理大小下训练和推理的全图性能。所有网络使用单精度浮点数数据类型。在 GPU 上使用 Tensor Core 时，本文会更改为半精度浮点数。每个测试图的训练大约需要 8-10 小时（大约 60-260 轮）完成编译，本文的交叠编译和执行策略可以帮助 FlexTensor 有效降低系统启动延迟。

4.7.2　在 GPU 上测试单个算子性能

本文在 GPU 上测试了所有算子，得到的性能数据是每个算子所有测例结果的几何平均值。如图 4.7所示。本文将 FlexTensor 与 PyTorch（没有 CuDNN）和 CuDNN 进行比较，图中纵轴是归一化的性能。CuDNN 不支持 GMV、GMM 和 BIL，因此这几个算子只与 CuBlas 进行比较。FlexTensor 在大多数算子上优于 PyTorch 和 CuDNN，所有算子相对于 CuDNN 的平均加速结果是：在 V100 上为 1.83 倍，在 P100 上为 1.68 倍，在 Titan X 上为 1.71 倍。FlexTensor 之所以能够实现较好的加速主要得益于对巨大调度空间（大小范围从 3.9×10^9 到 2.4×10^{12}）的有效搜索。

本文注意到 FlexTensor 在 T2D 和 T3D 算子上表现不佳。原因是 FlexTensor 的算子实现基于空间卷积算法，而 CuDNN 使用转换为矩阵乘法的算法和快速卷积算法[185]。这种算法级别的转换需要人手工设计，FlexTensor 不能自动搜索出来。

目前，一些算子在库中支持不好，这反映了设计高性能手工算子库是非常具有挑战性的。例如，GRP、DEP 和 DIL 虽然在 PyTorch 和 CuDNN 中有库支持，但性能很差。实际上，在 CuDNN 中，GRP 和 DIL 重用了 C2D 的函数。对于 DEP 算子，CuDNN 中的实现甚至比 PyTorch 慢，因此 PyTorch 内部不使用 CuDNN 来实现这个算子。因此，本文对于 DEP 只与 PyTorch 自己的库进行比较。如图 4.7所示，FlexTensor 在 GRP 和 DIL 上优于 CuDNN，GPU 上的加速范围从 1.54 倍到 21.35 倍。对于 DEP，FlexTensor 相对于 PyTorch 的加速范围从 4.39 倍到 8.53 倍。

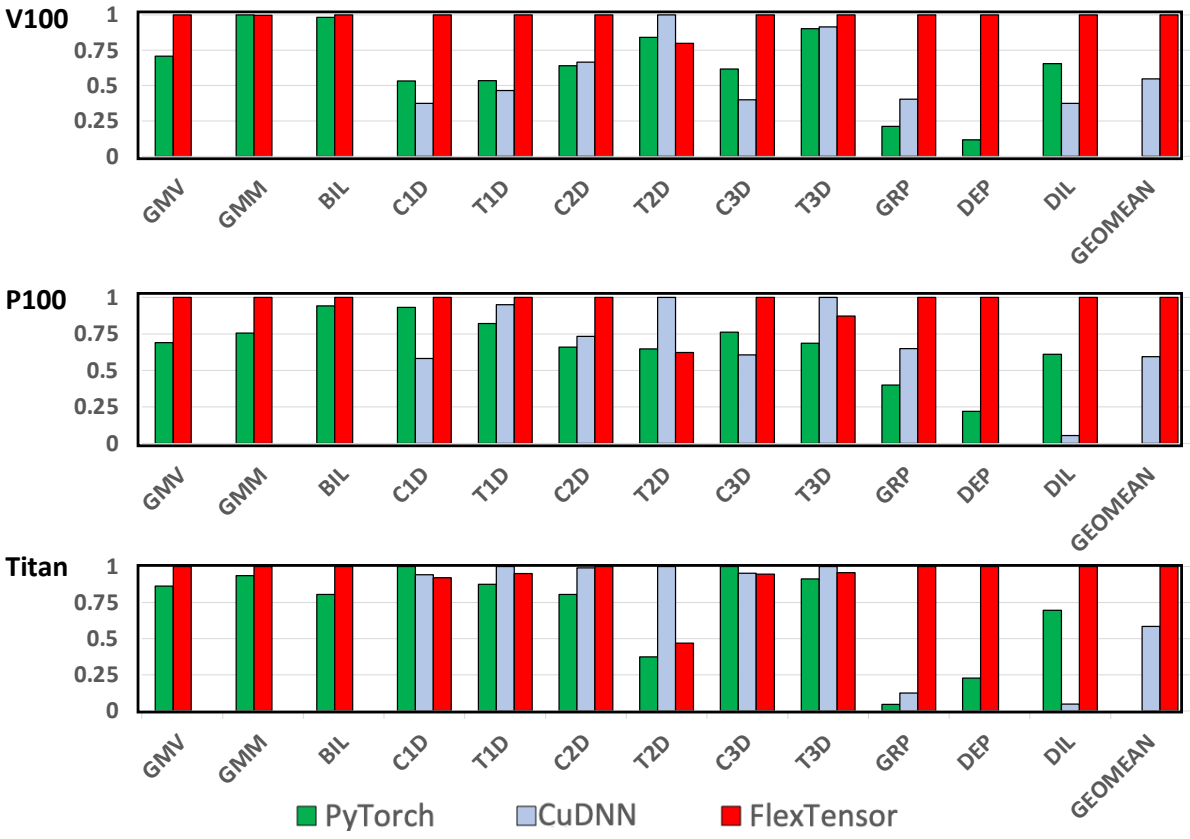
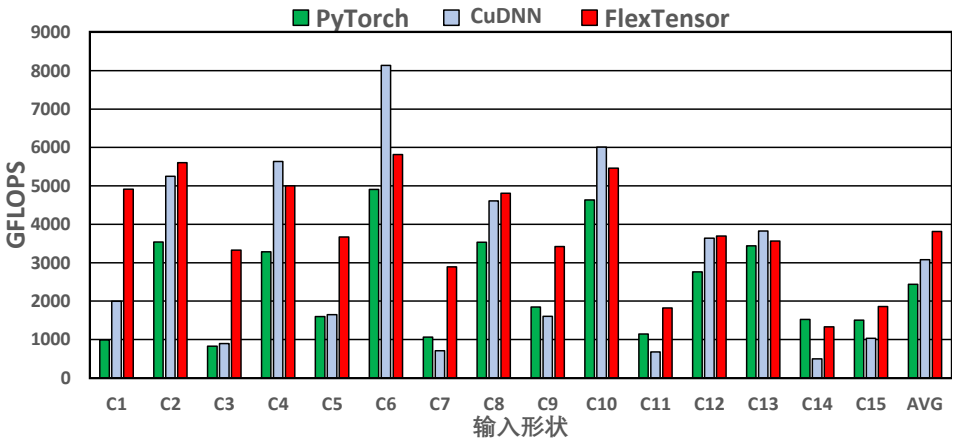


图 4.7　在不同 GPU 上比较 FlexTensor 和 PyTorch（使用 CuDNN 和不使用 CuDNN）对单算子的性能

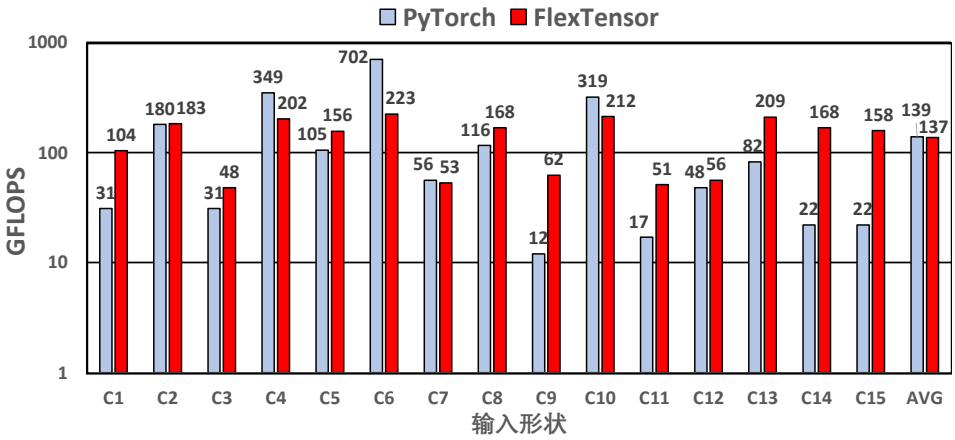
4.7.3　在其他硬件后端测试卷积算子的性能结果

接下来使用 C2D 作为一个例子，专门讨论 FlexTensor 在不同硬件上的性能结果。C2D 的测试用例的输入形状全部取自 YOLO-v1[15]中所有 15 个不同的卷积层，如表 4.3所示，其中 C 是输入通道，K 是输出通道，H 和 W 是输入高度和宽度，k 是卷积窗口大小，st 是步长。

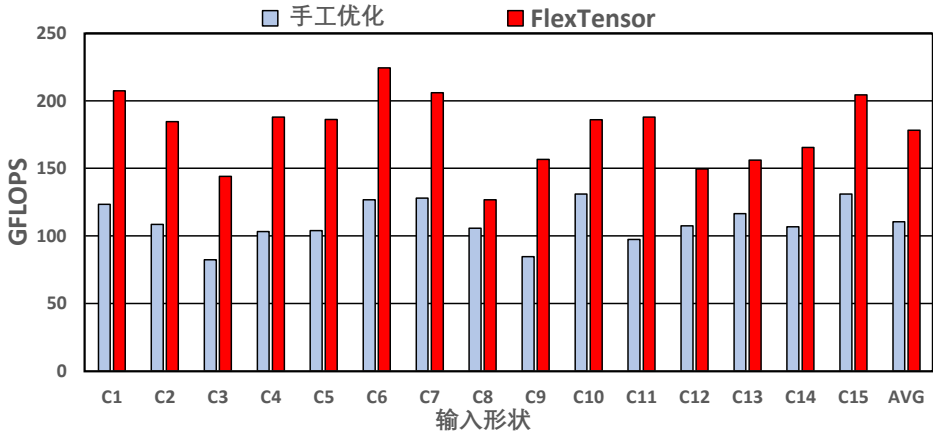
在 V100 GPU 上，FlexTensor 几乎在所有层上都超过了 PyTorch 和 CuDNN。相对于 PyTorch 和 CuDNN 的平均加速分别为 1.56 倍和 1.5 倍。绝对性能如图 4.8(a)所示。FlexTensor 可以达到平均 3519.71 GFLOPS 的吞吐量。FlexTensor 之所以能够实现高性能，是因为它通过探索大量的调度，寻找到了好的调度方案实现线程间和线程内工作负载之间的平衡。FlexTensor 可以动态适应不同的输入形状并寻找合适的调度，因此对于大多数层，它比手动优化的结果更好。本文还注意到，FlexTensor 对有些层的效果也不是特别好，如 C4 和 C6，因为 FlexTensor 只使用直接卷积算法，而 CuDNN 使用 Winograd 算法，这个算法可以大大减少计算复杂性[97]，不过在 FlexTensor 里暂时还不支持这个算法。



(a) 在 V100 GPU 上比较 FlexTensor 和 PyTorch 卷积计算性能



(b) 在 Xeon CPU 上比较 FlexTensor 和 PyTorch 卷积计算性能



(c) 在 VU9P FPGA 上比较 FlexTensor 和 PyTorch 卷积计算性能

图 4.8　展示 FlexTensor 在不同硬件上对卷积优化的结果

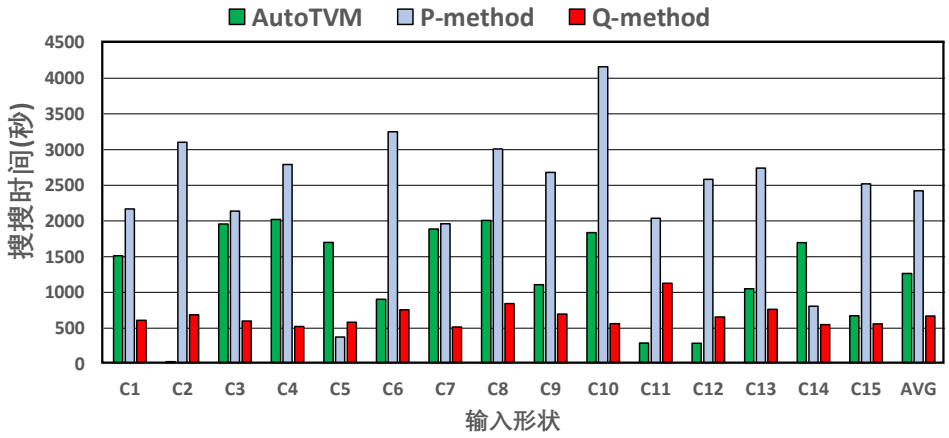


图 4.9　比较 FlexTensor 和 AutoTVM 在 V100 GPU 上达到相同性能所需搜索时间

图 4.8(b)展示了 CPU 上的绝对性能结果。在 CPU 上，FlexTensor 也在大多数层上超过使用 MKL-DNN 后端的 PyTorch 的性能，FlexTensor 相对于 MKL-DNN 的平均加速为 1.72 倍。FlexTensor 对 C2D 使用 NCHWc 数据排布[104]以利用 CPU 上的向量化特性。FlexTensor 能够实现高性能，归功于高效的分块和向量化调度决策。本文允许 FlexTensor 自动决定向量化长度以适应不同的平台，并发现在本文的实验中，FlexTensor 生成的调度都使用了长度为 8 的向量化长度，因为 Xeon E5-2699 v4 CPU 使用 AVX2 指令，最多允许 8 个浮点操作的向量化，这说明 FlexTensor 的自动适配非常有效。

在 FPGA 上，本文的对比对象是使用了之前工作[186]的调度方法进行优化的代码实现。实验得到的绝对性能结果如图 4.8(c)所示。FlexTensor 平均加速为 1.5 倍。FlexTensor 性能更好的原因有两个方面。一方面，FlexTensor 通过在特定 FPGA 资源约束下解决优化问题，使得设计空间探索更能覆盖较优解。另一方面，FlexTensor 提供了更好的调度策略，通过重叠数据通信和计算操作来减少片外存储访问开销。

本文还使用两个新的张量算子评估 FlexTensor，这些算子缺乏良好的库支持。它们是分块循环矩阵操作[187-188]（简称为 BCM）和移位卷积操作[189]（简称为 SHO）。BCM 用于嵌入式设备来剪枝参数，而 SHO 是 ShiftNet[189]中使用的无参数算子。本文将 FlexTensor 与这两个算子的手工调优实现进行比较。在手工调优实现中，使用 4 级分块与手动优化的分块参数，并将循环展开到最大深度 200。对于 BCM，在 V100 上，FlexTensor 相比于手工实现版本实现平均 2.11 倍的加速。在 Taitan X 上，对于 SHO，FlexTensor 相比于手工优化版本实现了平均 1.53 倍的加速。因此，使用这些新算子可以依赖 FlexTensor 自动生成高性能的代码实现。

接下来与 AutoTVM[25]进行比较，以展示 FlexTensor 性能和探索效率。AutoTVM 要求用户手动编写调度模板以优化算子。AutoTVM 根据模板生成调度空间，并探索该空间以寻找优化的调度。它使用 XGBoost[168]构建性能模型来指导搜索过程。为了构建

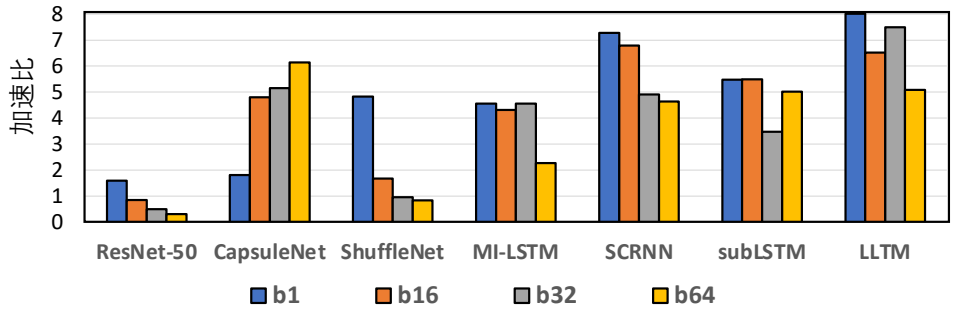


图 4.10　相比于 PyTorch 在训练上的性能提升

这样的模型，AutoTVM 需要首先尝试数千种不同调度，记录这些调度的性能作为训练数据来训练性能模型。本文在 C1D, T1D, C2D, T2D, C3D, T3D 和 GRP 这些算子上与 AutoTVM 比较。AutoTVM 目前没有为 C1D, T1D, C3D, T3D 提供模板支持，所以本文手动为这些算子设计了优化模板。FlexTensor 在所有算子上都超过了 AutoTVM，除了 T2D（0.95 倍），这些算子的平均加速为 2.21 倍。FlexTensor 之所以获得更好的性能，是因为它系统地探索了调度原语组合产生的空间，而 AutoTVM 依赖于静态模板。通过比较 AutoTVM 和 FlexTensor 对 C2D 算子的调度空间大小，可以发现 FlexTensor 搜索的空间平均比 AutoTVM 大 2027 倍。

本文将 FlexTensor 中基于 Q-Learning 的搜索方法称为 *Q-method*，将 *Q-method* 去掉而搜索所有方向的方法称为 *P-method*。二者的区别是 *P-method* 探索每个起点的所有方向，而 *Q-method* 通过询问 Q-Learning 算法获取每个点应该探索的方向，并只探索那个方向。对于 AutoTVM，它在每次搜索时随机获取若干新点进行探索。使用 V100 上的 C2D 作为测试例子进行研究，结果在图 4.9中，结果表明在达到相似性能时，*P-method*、*Q-method* 和 AutoTVM 所需的搜索时间不同。这个实验的做法是，首先运行 AutoTVM 并让它收敛到稳定的性能，然后运行 *P-method* 和 *Q-method* 达到相似的性能并记录所用的搜索时间。平均而言，*Q-method* 分别只需 *P-method* 和 AutoTVM 时间的 27.6% 和 52.9% 就能达到相似的性能。此外，实验还记录了 *P-method* 和 *Q-method* 的最终性能，在充分时间搜索后，*P-method* 的性能比 AutoTVM 好 1.41 倍，*Q-method* 的性能比 AutoTVM 好 1.54 倍。

4.7.4　在 GPU 上全图执行性能

图 4.10测试了批处理大小从 1 到 64 的情况，展示的是 FlexTensor 相对于 PyTorch 的加速比。对于 ResNet-50, 批处理大小为 1 时候, FlexTensor 是 PyTorch 性能的 1.61 倍。但对于更大的批处理大小，FlexTensor 无法超过 PyTorch，因为 PyTorch 中的反向计算时的反卷积算子使用了转换成矩阵乘法的算法进行了特别优化, FlexTensor 中没有这个

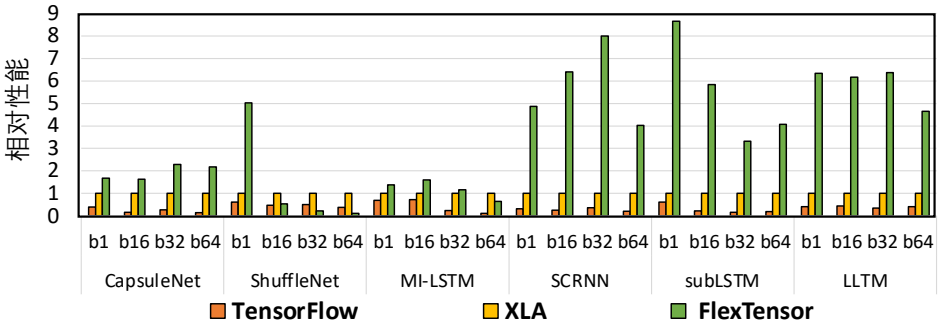


图 4.11　相比于 TensorFlow 在训练上的性能提升

优化。但对于由新算子组成的其他模型，FlexTensor 的加速是显著的。对于 CapsuleNet，PyTorch 由于频繁的函数调用和较低的设备利用率而表现出性能不佳。对于 ShuffleNet，FlexTensor 可以系统化探索算子融合方案，比如将 ReLU[14]与通道重排（Shuffle）算子、卷积与归一化算子（batch normalization）算子融合等等。ShuffleNet 的深度卷积[173]在库中由于优化不足而性能较差，但在 FlexTensor 中，可以探索多种分块策略、资源分配策略，以及循环展开方案，从而达到了更好的性能。对于 MI-LSTM、SCRNN、subLSTM 和 LLTM 这些网络，它们是 LSTM 的变体，所以它们的 PyTorch 实现必须使用多次矩阵乘法、加法、激活函数来拼凑获得最终结果。但在 FlexTensor 中，在算子层就支持了这些算子融合在一起的代码生成。总的来说，FlexTensor 在这些网络的训练任务上，实现了平均加速3.16 倍加速，相对于 CuDNN 时，这个加速比是1.92 倍。

除了和 PyTorch 对比，还可以和 TensorFlow 对比。图 4.11展示了与 TensorFlow 的比较结果。与 PyTorch 实现类似，TensorFlow 需要用多个算子拼凑来支持新算子。TensorFlow 可以选择是否开启 XLA 加速支持。XLA 是 TensorFlow 的编译器，帮助 TensorFlow 生成代码。相对于开启 XLA 的 TensorFlow，FlexTensor 的平均加速为2.43 倍。 XLA 的性能不如 FlexTensor 是因为 XLA 不能灵活调整优化参数，它更多地依赖专家知识。相比之下，FlexTensor 可以搜索完整的调度空间，在代码生成前找到更好的调度选择。

除了训练，本文还测试了全图推理性能。

使用 LLVM 后端的推理性能： 本文在图 4.12的 a) 部分展示了 FlexTensor 的全图推理性能。特别地，本文在 FlexTensor 中使用 LLVM[45]后端生成 PTX 代码。结果显示，对于批处理大小 1 和 16，FlexTensor 可以超过 PyTorch（使用 CuDNN）和 TensorFlow（开启 XLA）。相对于使用 CuDNN 的 PyTorch 的平均加速为 6.72 倍，相对于开启 XLA 的 TensorFlow 的加速为 4.96 倍。

大型模型评估：本文还使用大型模型评估性能。本文在图 4.12的 b) 部分展示了 Bert Encoder[87]的性能。本文使用 Bert 的 base 参数配置，这个配置可以在批处理大小为 1 时

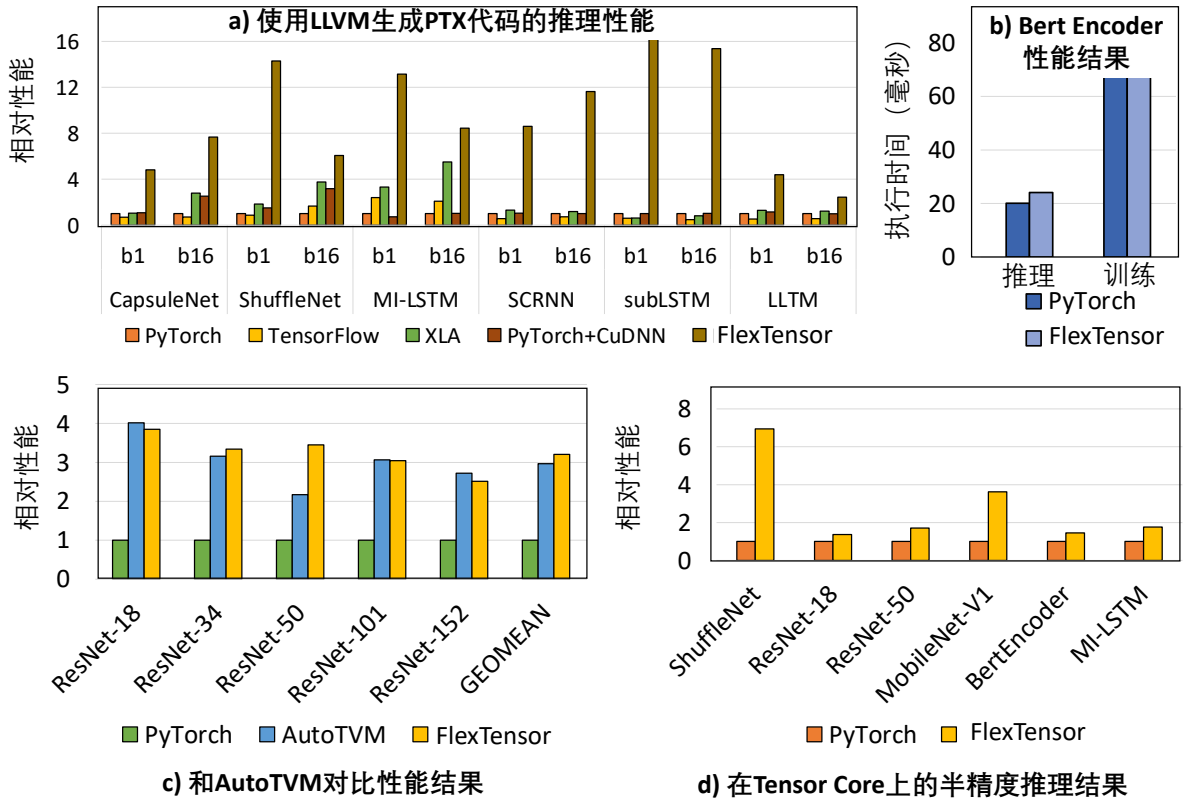


图 4.12　部分 a): 用 LLVM 生成 PTX 代码情况下的推理性能。部分 b): Bert Encoder 性能结果。部分 c): 和 AutoTVM 对比的性能结果。部分 d): 使用 Tensor Core 进行半精度推理的性能

占满 V100 的设备内存（批处理大小 16 就会超过内存容量）。根据实验结果，FlexTensor 的推理和训练执行时间与 PyTorch 相当。

与 AutoTVM 的比较： 本文与 AutoTVM 进行了推理性能比较，并在图 4.12的 c) 部分展示了结果。AutoTVM 需要 Relay[37]处理图级代码生成。Relay 的 IR 与 AutoTVM 使用的 IR 不同，且 Relay 的自动求导模块与 AutoTVM 不兼容。因此，实验无法获取 AutoTVM 的训练性能。对于推理性能，FlexTensor 与 AutoTVM 相当（平均加速为 1.08 倍）。

使用半精度 Tensor Core 的代码生成： 在 V100 GPU 上，FlexTensor 还可以通过生成 CUDA WMMA 指令来利用 Tensor Core 加速。在图 4.12的 d) 部分展示了使用 Tensor Core 的 FlexTensor 性能。与 PyTorch（也使用 Tensor Core）相比的平均加速为 2.31 倍。

4.7.5　搜索、编译、执行交叠的优势分析

为了展示 FlexTensor 的搜索、编译、执行交叠方法的好处，本文使用 MI-LSTM[180]作为例子进行讨论。MI-LSTM 全图包含了数百个算子（种类有 35 个）。在图 4.13中展示它的训练性能。批处理大小为 64，运行训练任务时间是 10 小时。图 4.13的上半部分显示了每次迭代的延迟（以毫秒为单位）。蓝色虚线是对比对象（PyTorch），红色实线是

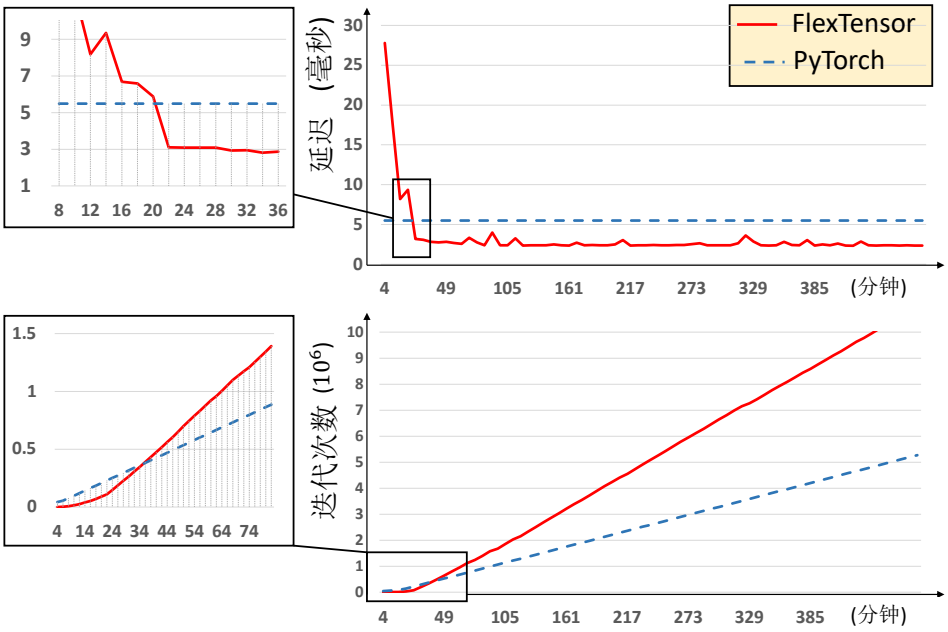


图 4.13　MI-LSTM 训练性能剖析

FlexTensor。由于编译的启动延迟，FlexTensor 等待了 3.96 分钟才开始真正的训练，初始全图迭代一次的延迟为 27.791 毫秒。尽管接近四分钟的延迟还是比较明显，但是相比于每个算子花费 20 分钟，生成 35 个算子 700 分钟的代价，还是小多了。训练进行 19.11 分钟后，由于不断的搜索和编译，生成的全图代码执行的延迟降就会低到 5.35 毫秒，这比 PyTorch 的 5.49 毫秒好。图 4.13中的性能波动是测量误差引起的，并非搜索会导致效果回弹。大约 1.5 小时后，全图的迭代延迟变得稳定，且优于 PyTorch。本文设计的编译与执行交叠技术允许用户在几分钟内就开始实际训练，而无需等待整个计算图的编译完成（需要几小时）。另一方面，本文在图 4.13的下半部分展示了随着时间推移完成的迭代次数（以 10^6 次迭代为单位）。PyTorch 以稳定的速度增加，FlexTensor 尽管一开始落后，但是在 41 分钟后就超过了它。最终 FlexTensor 在 10 小时内完成的迭代次数大约是 PyTorch 的 2.22 倍。

4.8　本章小结

本章中，针对 AI 编译的算子层次进行讨论并提出新的技术方案。基于先前工作的循环抽象，本文创新性提出对优化原语组合以及其优化参数的一并自动空间生成，并设计了配套的搜索算法辅助寻找较优的调度方案。此外，本文还介绍了算子层编译如何配合图层次编译完成算子级别的自动求导，指导图融合，通过交叠算子搜索、编译和实际执行来降低全图执行启动延迟的技术。总体来说，本文无论在单个算子还是全图训练、推理，都能取得明显有效的加速效果。

第五章　基于计算访存抽象的指令层编译

本章，将进入 AI 编译的第三个层次指令层。指令层编译占据了传统编译器技术的绝大部分篇幅，在 AI 编译中也有着不可替代的重要作用。本文主要侧重 AI 编译相对于传统编译不同的部分展开技术讨论和研究，对于已经比较成熟的指令层编译技术和算法则不进行深入讨论。AI 编译指令层直接对接 AI 芯片硬件指令集，承担着将上层软件抽象（循环）映射到底层硬件抽象（指令）的任务。本文将就如何让编译器自动地、正确地、高效地完成这样的映射进行深入讨论。

5.1　指令层技术背景介绍

5.1.1　硬件感知与 ISA 感知编译

正如之前在引言中介绍指令层编译时所说的那样，根据通用性和可编程性的程度，硬件加速器的设计可以有所不同，因此需要不同的软件到硬件的映射策略。本文将现有的面向 AI 芯片的映射流程分为两类：**硬件感知**和 **ISA 感知**。对于特定领域，硬件设计和软件映射可以通过定制的原语或者编程接口来结合在一起[32-35]。对于**硬件感知**映射，编译器需要知道硬件架构的细节，如处理单元（PE）的数量及其互联结构。这种情况下映射可以被表述为关于硬件参数的约束优化问题[21-23]。这种方法在特定应用领域实现了非常高的能效，代价是牺牲了灵活性。对于 **ISA 感知**映射，硬件加速器具有可编程的 ISA，这样本文可以将算法抽象与硬件架构细节分离。这些指令通常作为特殊的指令（称为 **intrinsic**）暴露出来，使用这些 intrinsic 实现张量计算被称为**张量化**[17]。本文关注的是 ISA 感知映射问题。

虽然 intrinsic 提供了可编程性，但 ISA 感知映射的设计仍然十分困难，主要有两个原因。首先，使用 intrinsic 组成映射的方式有很多种。例如，本文发现有 35 种不同的方式将二维卷积的 7 个循环映射到 Tensor Core 的 3 个维度上。此外，映射的质量显然对性能至关重要，因为不同的映射通过影响数据局部性和并行性而有很大的性能差异。但现有的编译器[17,25,27,29,31]严重依赖于使用 intrinsic 进行手写编程来开发库或模板，这可能错过最佳的映射选择，因此它们只能尝试有限的可能性。其次，不同的加速器提供了具有复杂计算和内存语义的 intrinsic。例如，Tensor Core 使用不同的 intrinsic 来描述矩阵加载/存储、矩阵乘法和初始化语义，而对于 Mali GPU，单个 *arm_dot* intrinsic 可以在没有其他显式加载/存储 intrinsic 的情况下工作。因此，为了在不同加速器上支持同一算法，程序员实际上需要为每个目标平台单独实现和优化算法，一般来说，现

表 5.1 XLA 能成功识别并映射到 Tensor Core 的算子数量和本文的方法能映射到 Tensor Core 的算子数量对比

网络名称	不同种类算子数量	XLA 映射的数量	XLA 失败案例	本文映射的数量
ShuffleNet	70	6	深度卷积	50
ResNet-50	71	15	步长大于 1 的卷积	54
MobileNet	30	7	分组卷积	29
Bert	204	42	self-attention	84
MI-LSTM	11	0	线性映射	9

有的编译工作还是依赖手工设计模板来实现软硬件映射，这导致了现有工作的效果非常有限。

为了说明先前工作的局限性，本文展示一个真实的测例，对于在 Nvidia V100 GPU 上对面向 Tensor Core 的软硬件映射进行分析，测试使用的网络包括 ShuffleNet[179]、ResNet-50[2]、MobileNet[184]、Bert[87]和 MI-LSTM[180]。所有的网络都是推理场景下测试的，批处理大小为 1。这里对比的对象是 XLA 编译器[28]生成的代码。XLA 使用复杂的专家设计的模板，这些模板通过手工调整来优化算子性能。特别是，其中一些模板匹配了 Tensor Core 的计算模式从而支持 Tensor Core intrinsic 的生成，或者使用一些手工优化的库如 CuDNN[84]和 CUTLASS[96]。

表 5.1列出了本文提出的方法和 XLA 对比的结果。虽然 XLA 中的模板设计得很好，但这些网络中只有少数算子成功映射到 Tensor Core 上。此外，本文发现一些 XLA 没成功映射的算子是有机会映射到 Tensor Core 上的。例如，ShuffleNet 中的深度卷积[190]和分组卷积[191]是二维卷积的变体，这些算子中的乘加操作也是可以映射到 Tensor Core 的；MI-LSTM[180]中的线性映射计算也可以映射到 Tensor Core 上，但是 XLA 并没有映射成功，可能是因为这个计算是矩阵向量乘，在 XLA 中无法被替换成矩阵乘法。但是本文能将这些算子映射到 Tensor Core 上，从而达到更好的性能。由于这些算子的计算模式与 XLA 的手写模板不匹配，导致无法映射，最终导致 XLA 对于 Tensor Core 利用率也很低（对于上述 5 个模型，平均利用率不超过 10%）。相比之下，本文的方法可以成功地将所有能映射的算子都映射到 Tensor Core，那些本质上与矩阵乘之间无法进行转换的计算，就根本无法映射到 Tensor Core（例如，ReLU 和池化运算），本文自然也就不考虑他们。表 5.1展示了本文在 Tensor Core 上成功映射的算子数量，明显是高于 XLA 的。这个实验充分展示了已有工作的局限性，也表明本文方法的有效性。显然，一个理想的映射解决方案应该是使用统一的方法来进行自动搜索和生成映射的，而不是依靠人力进行固定的模板设计。

5.1.2 指令层编译优化的主要挑战

本文认为面向 AI 芯片的指令层层编译主要挑战有三点：

如何自动判断 intrinsic 使用的机会：由于不同硬件对于指令的设计各有不同，即使对于同样的功能（如矩阵乘法），intrinsic 的语义和使用限制条件也不尽相同，这些指令相关的说明往往通过文档的方式呈现给开发者，开发者首先阅读理解文档，然后再进行编程，通过开发者的经验自行判断所实现的算子是否可以使用 intrinsic，以及在哪些地方使用这些 intrinsic。如果希望使用编译技术自动化映射到 AI 芯片，就必须解决让编译器理解指令语义并自动在软件程序中寻找可以使用 intrinsic 的位置，这对于编译器设计是一个挑战。

如何自动验证 intrinsic 使用的正确性：相比常用的标量计算指令集，AI 芯片使用的 intrinsic 往往具备更复杂的语义，在自动生成时需要保证指令使用的正确性，特别是 intrinsic 的一次计算往往涉及到多个输入输出数据和多次循环迭代，如果将数据位置或者迭代循环映射错误，会导致计算结果的错误，这种错误往往是难以发现的，因为错误的映射也是可以编译生成代码的，之后在运行时实际使用时才能发现错误。因此，本文需要相应的技术在编译时就判断出这类映射错误，以保证生成的映射的正确性。

如何与图层和算子层编译结合：指令层映射往往需要进行严格的模式匹配和验证，这需要进入编译器的原始代码具有一定的特征，但是由于 AI 编译的软件栈比较深，经过了图层次编译和算子层次编译的代码，可能就失去了一些重要的特征，导致指令层无法进行映射。比如，在 Tensor Core GPU 上实现一个矩阵乘法，图层次可以进行激进的融合，把矩阵乘法前后的其他算子全部融合进来，导致矩阵乘法的输入输出数据物理排布和 Tensor Core 要求的不一致，算子层又在循环分块时候忽略了 Tensor Core 对输入矩阵形状的要求，分块后的最内层循环形状和顺序都不符合 Tensor Core 的语义要求，那么到了指令层，想要恢复符合 Tensor Core 的语义并完成代码生成就难上加难，几乎不可能完成。所以，指令层编译需要图层和指令层编译的配合。

5.2　基于计算访存抽象的编译方案整体结构

在本章中，将提出在 intrinsic 之上提供一层编译抽象来解决 ISA 感知映射问题的方法。本技术有以下两个观点：1) 尽管不同的加速器使用不同的 intrinsic，但这些 intrinsic 可以被重写为等价的循环程序格式，这不仅打破了原来不透明的 intrinsic 语义壁垒，让 intrinsic 成为一个可分析的数据结构，还实现了高层次张量程序向低层次 intrinsic 之间映射；2) 尽管不同的 AI 芯片在架构设计上有所不同，但它们在映射设计上有相似性和共通之处。例如，它们的 intrinsic 的硬件约束可以统一归结为计算规模大小约束和内存容量约束。基于这些观点，本文能够使用统一的编译抽象来描述不同芯片的不同 intrinsic，并为不同硬件设计自动映射生成、验证和搜索的解决方案。

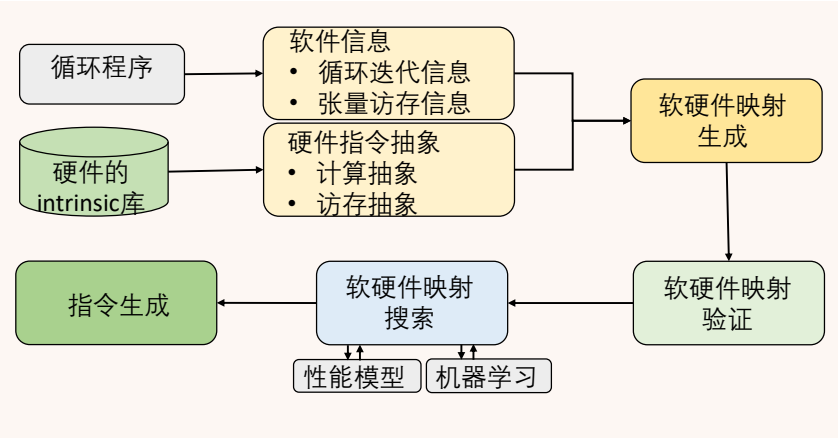


图 5.1　AMOS 整体框架结构

具体来说，本文所提出的抽象包括两部分：计算抽象和访存抽象，它们以循环程序的格式描述了一个 intrinsic 内的计算和数据访问操作语义。基于所提出的抽象，本文进一步设计了一个两步映射生成流程和一个验证算法，用来进行自动生成正确的软硬件映射。本文的生成流程首先将软件计算映射到没有硬件约束的虚拟硬件上，然后根据实际的物理硬件约束逐步修改映射以得到实际映射。并非所有生成的映射都是正确的，所以本文设计了一个基于矩阵的验证算法来检查映射的正确性。

本文将所提出的指令层编译技术实现为一个叫做 AMOS 的框架。在图 5.1中，本文展示 AMOS 的整体结构，AMOS 的输入是用循环语义[17,31]编写的张量计算函数，其中涵盖了每个张量的循环结构和张量访存索引，这样就可以对接上层算子层。图5.2的 a) 部分展示了二维卷积的一个循环程序例子。AMOS 使用本文提出的硬件计算访存抽象来表示计算和访存语义并把不同硬件的 intrinsic 抽象起来维护成一个库。借助这种抽象，AMOS 可以生成不同的软硬件映射（或者称为软件与指令的映射）。一个软硬件映射由计算映射和访存映射组成。计算映射指定软件中定义的循环迭代和硬件指令语义中的循环迭代的对应关系，这反映了如何使用计算 intrinsic 完成计算。访存映射指定软件程序的张量中的数据元素和指令操作数中的元素的对应关系，这反映了如何使用访存 intrinsic 实现从全局/共享内存加载数据到寄存器的过程。生成的映射经过验证之后就会进入性能调优部分，这部分需要 AMOS 在多版本的映射中选择性能最好的，AMOS 可以借助性能模型，也可以借助机器学习算法，正如前文在图和算子层面做过的那样。最后，性能最好的映射方案会被用来生成实际的指令程序。

5.3　计算访存抽象与软硬件映射

在本节中，介绍计算访存抽象和软硬件映射。抽象的主要思想是将低层次 intrinsic 转化为等价的高层次循环程序的表示方法。这里将抽象分为两部分：1) 计算抽象；2)

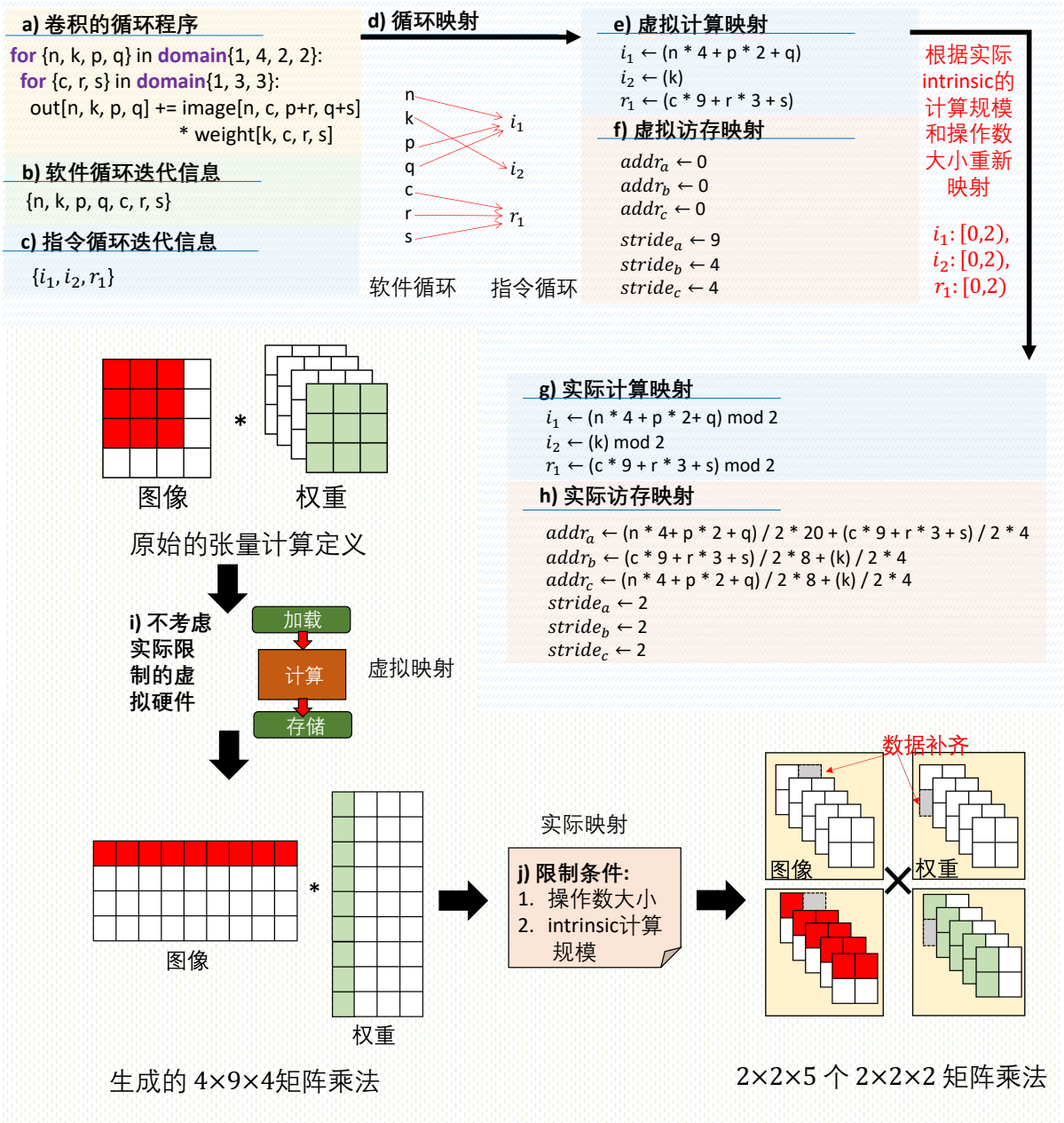


图 5.2　AMOS 的映射生成历程。这个例子展示了如何把一个小的二维卷积映射到一个简化的 Tensor Core (只有 $2 \times 2 \times 2$ 的计算规模)。部分 a): 软件循环程序。部分 b): 软件的循环迭代信息。部分 c): Tensor Core 的指令循环迭代信息。部分 d): 软件循环与指令循环之间的映射信息。部分 e) 和 f): 不考虑实际限制的虚拟计算和访存映射。部分 g) 和 h): 考虑了实际限制的计算和访存映射。部分 i): 虚拟硬件示意图。部分 j): 映射中的限制条件。

访存抽象。

定义 3 计算抽象是一个循环体语句，其中定义了一个硬件计算 *intrinsic* 的操作数、操作数之间的算术操作以及操作数的数据访存下标。下标的范围也在抽象中表示，主要

依靠循环的范围推断。假设有 M 个输入操作数，第 m 个操作数 Src_m 是一个 D_m 维数组。输出 Dst 是一个 N 维数组。输出数组的访存下标表示为 $\vec{i} = [i_1, i_2, .., i_N]$，第 m 个输入数组的访存下标表示为 $\vec{j}^m = [j_1^m, j_2^m, ..., j_{D_m}^m]$。那么这个 *intrinsic* 的计算抽象就写为

$$\begin{aligned} Dst[\vec{i}] &= \mathscr{F}(Src_1[\vec{j}^1], Src_2[\vec{j}^2], ..., Src_M[\vec{j}^M]), \\ s.t.\ A\vec{i} + \sum_m B^m \vec{j}^m + C &< 0 \end{aligned} \tag{5.1}$$

解释： 函数 $\mathscr{F}$ 表示算术操作，如加法、乘法或乘加操作。矩阵 A、B^m（$1 \le m \le M$）和 C 用于以仿射方式表达下标的访存范围。常数矩阵 C 用来表达索引的边界。例如，如果想表示一个 Tensor Core *mma_sync* intrinsic，并且它的计算规模是 $32 \times 8 \times 16$ 的矩阵乘法，可以写成这样的抽象：

$$\begin{aligned} &\mathrm{Dst}[i_1, i_2] = \text{multiply-add}(\mathrm{Src}_1[i_1, r_1], \mathrm{Src}_2[r_1, i_2]), \\ &\text{s.t.} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} r_1 \end{bmatrix} + \begin{bmatrix} -32 \\ -8 \\ -16 \end{bmatrix} < 0 \end{aligned} \tag{5.2}$$

定义 4 访存抽象是一个语句列表。列表中的每个语句指定访存范围、操作数和内存访问下标。与本文在计算抽象中使用的符号类似，本文使用 Dst 表示输出结果，Src_m 表示输入数据数组。前缀 $global$、$shared$ 和 reg 分别代表全局内存、共享内存和寄存器。$\vec{i}, \vec{j^m}, \vec{k}$ 和 $\vec{l^m}$ 是不同范围内数据数组的下标。

$$\begin{aligned} &reg.Src_1[\vec{j^1}] = shared.Src_1[\vec{l^1}] \\ &... \\ &reg.Src_M[\vec{j^M}] = shared.Src_M[\vec{l^M}] \\ &global.Dst[\vec{k}] = reg[\vec{i}] \end{aligned} \tag{5.3}$$

解释： 上面展示的访存抽象表明输入数据来自共享内存，中间数据加载到寄存器，输出数据存储在全局内存中。类似的本文也可以写从全局内存到寄存器的加载和从全局内存到共享内存的加载，不过在这里本文都略掉了。同一个操作数在不同内存范围内的访存下标可以不同。这里再举一个具体的例子，如果想表示从共享内存加载数据到 Tensor Core 寄存器的 intrinsic *load_matrix_sync* 和将数据从 Tensor Core 寄存器存储到全局内存的 intrinsic *store_matrix_sync*：

$$\begin{aligned} &reg.\mathrm{Src}_1[i_1, r_1] = shared.\mathrm{Src}_1[addr_a + i_1 * stride_a + r_1] \\ &reg.\mathrm{Src}_2[r_1, i_2] = shared.\mathrm{Src}_2[addr_b + r_1 * stride_b + i_2] \\ &global.\mathrm{Dst}[addr_c + i_1 * stride_c + i_2] = reg.\mathrm{Dst}[i_1, i_2] \end{aligned} \tag{5.4}$$

其中 $addr_a$、$addr_b$ 和 $addr_c$ 是内存基地址，$stride_a$、$stride_b$ 和 $stride_c$ 是内存访问步幅。当将软件映射到硬件时，这些参数都是由编译器设置的。

计算和访存抽象清晰地描述了硬件 intrinsic 的行为，并将原来不透明的 intrinsic 分解为一系列等价的透明的循环操作，以便进行分析和变换。现有加速器中的各种 intrinsic[9,12-13]实际上是不同的向量操作与矩阵操作，统称为类 SIMD 操作，而类 SIMD 操作可以自然地转换为等价的循环程序，这也是 AMOS 用这种方式抽象 intrinsic 的原因。

接下来，介绍软件到硬件的映射。首先，需要引入**软件循环迭代**的概念。张量计算可以表示为嵌套循环。软件循环迭代是软件程序的循环体中的所有循环[42,111]的列表。例如，在图 5.2的 b) 部分，展示了一个卷积的软件循环迭代。

类似地，基于先前引入的计算抽象，可以为 intrinsics 定义**指令循环迭代**。指令循环迭代对应计算抽象的下标循环迭代范围。例如，如公式 5.2所示，Tensor Core intrinsic 中有三个指令循环迭代（i_1, i_2, r_1）。给定软件循环迭代、指令循环迭代、计算抽象和访存抽象的定义，本文将软硬件映射定义如下

定义 5 软硬件映射由两部分组成：计算映射和访存映射。计算映射是将每个软件循环迭代对应到一个指令循环迭代。访存映射是为访存抽象中的每个操作数对应到一个软件程序的数组地址（基地址和步幅）。

$$
\begin{aligned}
&\Theta = < \mathit{Compute\text{-}Map} : \mathbb{C},\ \mathit{Memory\text{-}Map} : \mathbb{M} > \\
&\mathbb{C} = \{\mathit{Software}[\vec{s}] \ \rightarrow \ \mathit{Intrinsic}[\vec{i}, \vec{j^1}, ..., \vec{j^M}]\} \\
&\mathbb{M} = \{\mathit{Software}[\vec{s}] \rightarrow \mathit{operand}[\vec{addr}]\} \\
&\qquad \text{对于访存抽象中的每一个操作数都如此}
\end{aligned}
\tag{5.5}
$$

解释：为了通过 intrinsic 将软件映射到硬件，必须知道软件中定义的每个计算访存操作是如何通过相应的计算/访存 intrinsic 实现的。基于计算/访存抽象，原始 intrinsic 被分解为一系列循环操作，以便编译器可以建立软件循环迭代和指令循环迭代之间的映射关系。除了计算的映射外，内存访问的映射也是类似的。在访存抽象中，所有的访存下标可以通过计算抽象唯一确定。如果可以将每个软件循环迭代与每个操作数的一组访存下标相关联，那么就可以进一步推理出指令操作数与软件程序中的每个数据元素的对应关系。接下来的内容将解释如何自动生成软硬件映射，验证其正确性并寻找性能最好的映射方案。

5.4　映射生成技术

为了生成软硬件映射，本文提出了一个分两步生成的方法。首先，本文将软件循环迭代映射到一个虚拟硬件上，该虚拟硬件只有一个数据加载引擎、一个计算引擎和

input : 软件访存矩阵: X
input : 映射矩阵: Y
input : 指令访存矩阵: Z
output : 是否正确

1: // $\star$ 是布尔矩阵乘法;
2: $X' := Z \star Y$; // 获取软件访存指令操作数的关系;
3: $Z' := X \star Y^T$; // 获取指令访存软件数组的关系;
4: **return** $(X' = X)$ 且 $(Z' = Z)$

Algorithm 4: 映射验证算法

一个数据存储引擎，如图 5.2的 i) 部分所示。在此步骤中，可以先不对内存容量和计算规模进行限制。在第二步，会再加回硬件的约束，并修改之前的映射，输出一个实际的映射，根据这个实际映射就可以完整软件到硬件的部署。本文使用图 5.2中的例子进行更详细的解释。这个例子使用卷积和一个简化的 Tensor Core（硬件加速的计算是 $2\times2\times2$ 的矩阵乘法）。

在映射生成的第一步中，AMOS 假设虚拟硬件可以将任意大小的数据加载到片上寄存器中，并且假设硬件有足够的资源容纳所有数据，并一次性完成所有计算。因此，这一步的主要问题是如何将软件定义的计算操作放置到对应的硬件提供的计算单元上，对于指令感知型映射，问题就是如何把计算操作映射到指令提供的计算操作上。在图 5.2的 d) 部分，展示了一个匹配示例，将软件循环迭代 n, p, q 与指令循环迭代 i_1 匹配，软件循环迭代 k 与指令循环迭代 i_2 匹配，以及软件循环迭代 c, r, s 与指令循环迭代 r_1 匹配。根据这种匹配关系，原始的卷积计算（批处理大小是 1, 输入通道是 1, 输出通道是 4, 高度是 4, 宽度是 4, 卷积窗口大小是 3）被变换为等价的矩阵乘法（高度是 4, 归约维长度是 9, 宽度是 4）。注意，这种能将计算进行等价转换的编译能力是在前述图层次和算子层次都没有达到的。虚拟硬件会将两个矩阵（形状为 4×9 和 9×4）加载到寄存器中，同时完成 $4\times9\times4$ 的矩阵乘法，并将输出矩阵存储到全局内存中。这一步虚拟映射的访存映射中的基地址设置为零，步幅根据矩阵形状设置，如图 5.2的 e) 和 f) 部分所示，因为假设所有数据都驻留在寄存器中，所以这个访存映射十分简单。然而，虚拟映射和实际可行的映射还是有差距的，因为真实的硬件在内存容量和计算能力方面都会受到限制。因此，本文需要在下一步中考虑这些约束，修改虚拟映射。

对于映射生成的第二步，需要考虑两种约束：指令计算规模大小和指令操作数所需的内存容量。硬件芯片上的计算单元一次只能计算固定大小的结果，这被称为问题规模大小约束。AMOS 通过加入取模运算将虚拟映射中的计算再次进行分割以保证每个小份都符合硬件单元的限制。这种限制条件其实在抽象计算的时候就写出来了，所以编译器可以从计算抽象获取信息完成这一步修改。在图 5.2的 g) 部分，匹配的软件循环迭代按照模 2 的方式进行分块，以匹配 Tensor Core 的处理规模大小：$2\times2\times2$。

对于访存时指令操作数内存容量的约束，AI 芯片中的每个寄存器堆只能容纳有限大小的数据，而之前的虚拟映射假设寄存器无限多。因此，AMOS 将虚拟映射中的整个输入/输出数据也进行分割，分割后得到多个小块，并通过多次加载/存储这些数据块逐步完成计算，这意味着访存映射中的基地址和步长应相应改变，根据当前计算所使用的小块标号而变化。定位这些小块的具体下标，可以用计算分块后外层循环的迭代下标组合得到。至于步长，他们应该与寄存器容量大小匹配。在图 5.2的例子中，Tensor Core 的每个操作数（图像、权重和输出）是一个 2×2 矩阵。在 h) 部分，新的下标 $(n*4+p*2+q)/2$ 就是对应了分块循环 i_1 后的外层循环，下标 $(c*9+r*3+s)/2$ 就是对应分块循环 r_1 得到的外层循环。根据循环分块时候的对应关系，可以重建访存地址 $(n*4+p*2+q)/2*20+(c*9+r*3+s)/2*4$，其中 4 是子矩阵内的元素数量，$20=5\times4$，这里的 5 对应 $(c*9+r*3+s)/2$ 的范围（如图 5.2所示）。在分块时候，数据不一定是严格对齐的，所以还需要在尾部补充适当的 0 来保证每个小块都符合 Tensor Core 指令的需求。在这个例子中，一个 4×9 矩阵被分割成 2×5 个小的 2×2 子矩阵，所以子矩阵中存在尾部填充数据的需求。

5.5　映射验证技术

在实际生成映射时，一些生成的映射方案可能不是正确的，AMOS 使用一个验证算法来确保只有正确的映射才会继续生成代码。为了说明，这里举一个错误映射的例子，如果在图 5.2的 d) 部分将软件循环迭代 n,k 映射到相同的指令循环迭代 i_1，这样做既满足了硬件相关的约束，也不会带来任何的编译错误，但是它是一个语义错误的映射。原因是软件循环迭代 n 和 k 在原始的卷积中有不同的访存语义：n 是输出和图像的公共下标但从不出现在权重访存下标中，而 k 是输出和权重的公共下标但从不出现在图像访存下标中。因此，它们在语义上就不应该被映射到同一个指令循环迭代上，强行映射只会导致计算结果的错误。

本文的验证算法在算法 4中展示出来。首先解释算法的输入。输入的访存矩阵是一个布尔矩阵，描述了循环和数据数组之间的数据访问关系。矩阵的每一行代表一个数组，每一列代表一个循环。如果列 col 指代的循环用于访问行 row 指代的数组的数据，则矩阵中 (row,col) 位置的对应值设置为 1，否则为 0。输入的映射矩阵就是对本章节提出的软硬件映射的矩阵化描述，它也是一个布尔矩阵，描述了软件循环迭代和指令循环迭代之间的匹配关系。例如，在图 5.3中，展示了卷积的访存矩阵、Tensor Core 的指令访存矩阵和它们之间的映射矩阵。这个映射矩阵中反映的映射关系与图 5.2的 d) 部分所示的相同。

算法 4通过计算和对比软件访存指令操作数关系和指令访存软件数组关系来验证

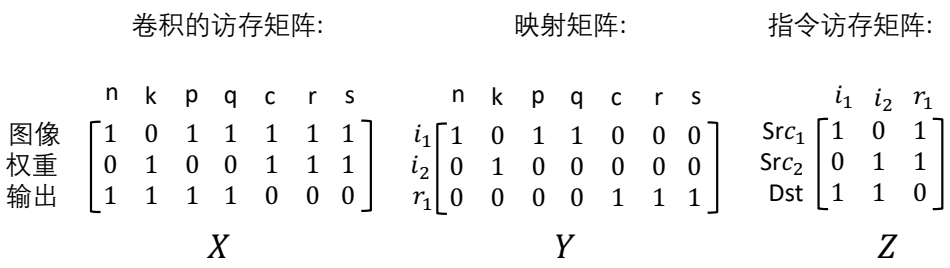


图 5.3　访存矩阵和映射矩阵举例

映射的正确性。软件访存指令操作数关系定义了哪些软件循环迭代访存硬件内存（寄存器）的哪些位置；指令访存软件数组关系定义了指令循环迭代访存哪些软件数组元素。为此，算法首先使用指令访存矩阵 Z 和映射矩阵 Y 进行 $Z \star Y$ 计算得到软件访存指令操作数关系，这由另一个布尔矩阵 X' 表示。这里使用 $\star$ 表示布尔值矩阵之间的矩阵乘法操作。如果 X' 与 X 相同，则意味着对于每一对输入/输出张量（在软件定义中）和输入/输出操作数（在指令定义中），所有下标的访存行为保持不变，这个映射是正确的，否则，访存语义没有被保留，映射是错误的。类似地，也可以计算 $X \star Y^T$ 得到指令访存软件数组关系，其中 X 是软件访存矩阵。结果 Z' 也是一个布尔矩阵，如果映射矩阵 Y 正确，那么 Z' 应该与指令访存矩阵 Z 相同。算法 4对于验证软硬件映射正确性是高效且准确的，因为它只需要简单的布尔矩阵乘法。AMOS 利用这个算法保证生成的映射保留了软件中的原始语义。对于图 5.2中的示例，AMOS 能在发现的 $3^7 = 21087$ 个可能的映射中选出仅有的 35 个正确的映射。

5.6　与图层、算子层编译结合

正如前文所说，为了保证输入到指令层编译的程序能被分析和匹配，需要在图层和算子层就留意保留一些重要信息给指令层。

在图层，需要保证图上的融合不会破坏底层指令需要的数据排布格式，甚至在必要时，需要在图中插入节点以保证数据排布格式的正确。这里用一个例子说明这一点。在图 5.4中展示如何端到端地对一个 AI 模型应用指令层编译。首先，对于输入的图结构，要找到哪些算子需要使用 AI 芯片的 intrinsic 进行加速，这部分可以通过抽象语法树匹配完成，在当前例子中，展示的是面向 Tensor Core 的映射例子，可以看到图中匹配到了一个卷积算子，说明卷积算子能向 Tensor Core 映射，能发现这一匹配，是因为卷积表达式的抽象语法树存在与 Tensor Core 计算抽象的抽象语法树匹配的子结构。然后，进一步对这个卷积算子进行前文介绍的映射生成和验证流程，并保留得到的映射信息。随后，根据映射信息，编译器可以得知卷积的输入输出数据需要被重新组织成矩

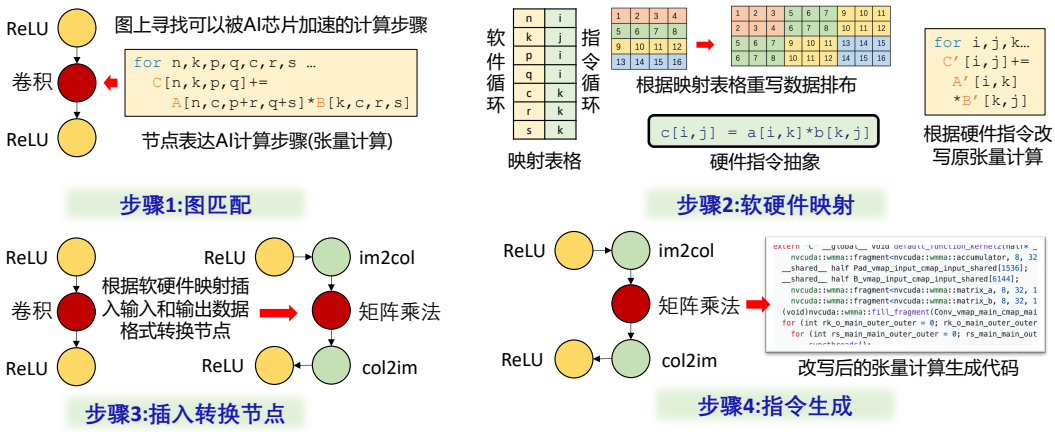


图 5.4　图层次编译配合指令层编译示意图

阵乘法的格式，同时要对数据进行分块，分块不仅仅是循环的变换，数据在物理内存中的排布也要相应改变以适配数据加载和存储时指令对数据连续性的要求。为此，在这个例子的步骤 3，需要在计算图中插入两个数据格式转换节点，对于这个特殊的例子，为了方便介绍，插入的节点是 im2col 和 col2im，但是对于通用的张量计算，具体插入什么节点需要编译器自行生成（不需要手工指定，也不需要指出数据格式变换算子的名称）。这样，在之后的编译步骤中，全图的计算是仍然正确的，并没有因为卷积被变换成了矩阵乘而导致前后算子的输入输出数据格式不一致。

通过这个例子，可以发现，尽管本文在行文过程中严格按照图层、算子层、指令层的顺序进行介绍，但是其实真正组成编译软件栈的时候，不同层次的工作是交叉完成的，比如这里的例子里是按照图层-指令层-图层-指令层的顺序完成了编译工作。交叉进行的过程中，不同层次的编译工作会保留自己之前的分析结果，以便于之后再次启动的时候可以直接进行后续工作。

除了和图层编译需要配合，指令层编译更要和算子层编译进行配合。因为目前介绍的指令层编译仅仅考虑如何在生成的代码中正确地插入 AI 芯片特殊指令，但是实际上一个算子的代码实现远比单纯调用一个指令要复杂，算子代码的外层还有非常深的循环嵌套、片上缓存分配和调度问题，这就需要算子层的相关编译技术进行辅助。

那么，将算子层的技术也加入之后，三个层次相互配合的总体的工作流程如下：

第一遍分析：一个 AI 计算图在经过图层编译时，首先通过和算子层的配合完成自动求导、算子融合性能评估，然后再通过指令层匹配可以使用特殊指令的的算子，分析不同的指令映射方式。

第二遍分析：图层次根据上一遍分析的结果，将图中融合的算子分块，然后块间进行数据搬移量分析和优化，块内信息根据指令层匹配结果，用可替换微内核进行语义保护，内部的循环结构不进行改变。这样图层次的产出是一个个融合的算子。融合的

表 5.2　结合指令层进行设计空间搜索时使用的符号

符号	含义
$\mathscr{L}_l$	层次 l 完成计算所需的延迟
$\mathscr{R}_l$	层次 l 完成数据加载所需的延迟
$\mathscr{W}_l$	层次 l 完成数据存储所需的延迟
$\vec{S}_l$	层次 l 上没有并行起来的循环的列表 l
DataIn_l	第 l 层内存需要读取的数据总量
DataOut_l	第 l 层内存需要存放的数据总量
in_bw_l	第 l 层内存的读入数据带宽
out_bw_l	第 l 层内存的数据写出带宽

算子交给算子层后，仅对块间的循环进行调度原语空间生成和搜索，对于块内的信息，如果被可替换微内核保护，则不进行调度调优。这些被可替换微内核保护的块，将算子层对外层循环和内存调度结束后，被交给指令层，如果可替换微内核有对应的手工优化实现，则直接使用这些手工代码进行代码生成，否则，指令层就要根据之前的分析结果（软硬件映射），把块内可替换微内核替换成对应的特殊指令调用并生成代码。

通过两遍分析，交叠了两次图、算子、指令层次的编译，最终完成了从图到底层代码的生成。值得注意的是，每一次执行两遍分析，都会得到一个版本的代码，为了寻找最优的代码生成方案，可以选择生成多版本代码之后选性能最好的那个，这时候，整体的优化空间是十分完整的图-算子-指令三维度空间。虽然理论上这样的空间搜索是最优的，但是仅仅在一个层次上，就会得到很大的设计空间（比如算子层），无法进行穷举搜索，所以实际使用中，并没有对三维空间进行全面搜索，而是分解这个空间为两份，从算子层向前向后分割。图层次与算子层次的空间，可以根据本文在图层次和算子层次介绍的分析方法和搜索技术确定融合方案。算子层和指令层的空间，则通过搜索循环变换优化和指令选择的不同方案来生成。这样分别完成图-算子和算子-指令联合优化，虽然空间减小了，实际生成的代码效果也依旧很好。

图层，算子层都采用性能模型和调优技术结合的方式进行搜索，因此图-算子联合空间已经介绍完毕了。只需要再补充介绍算子-指令联合空间中的指令层选择空间的设计和搜索，就完成了从上到下全部编译优化空间的介绍了。考虑到 AI 芯片是按层次设计的，指令层的性能模型也是按层次构建的（第 0 层对应于 intrinsic 本身，也就是寄存器）。使用的符号解释列在表 5.2中。性能模型可以写为

$$\text{Perf} = \mathscr{L}_{L-1}, \quad L \text{ is the number of levels of hardware} \tag{5.6}$$

$$\mathscr{L}_l = \begin{cases} (\prod \vec{S}_l) \times \max(\mathscr{L}_{l-1}, \mathscr{R}_{l-1}, \mathscr{W}_{l-1}), & l > 0 \\ (\prod \vec{S}_l) \times \text{latency\_of\_intrinsic}, & l = 0 \end{cases} \tag{5.7}$$

$$\mathscr{R}_l = \frac{\text{DataIn}_l}{\text{in\_bw}_l} \qquad \mathscr{W}_l = \frac{\text{DataOut}_l}{\text{in\_bw}_l} \tag{5.8}$$

将这个性能模型与 FlexTensor 的搜索空间集成到一起，就可以探索指令映射和循环优化调度的联合搜索空间。具体来说，在搜索开始时，首先通过映射生成和验证过程枚举所有可能的映射，并随机为每个映射选择生成循环调度空间，这些映射及其循环调度通过算子层和指令层的性能模型进行性能评估。根据评估结果，就可以选择一组好的映射加循环调度去生成代码。当映射方案很多的时候，可以每次只评估一小部分映射，多次迭代完成搜索过程。

5.7　实验评估结果

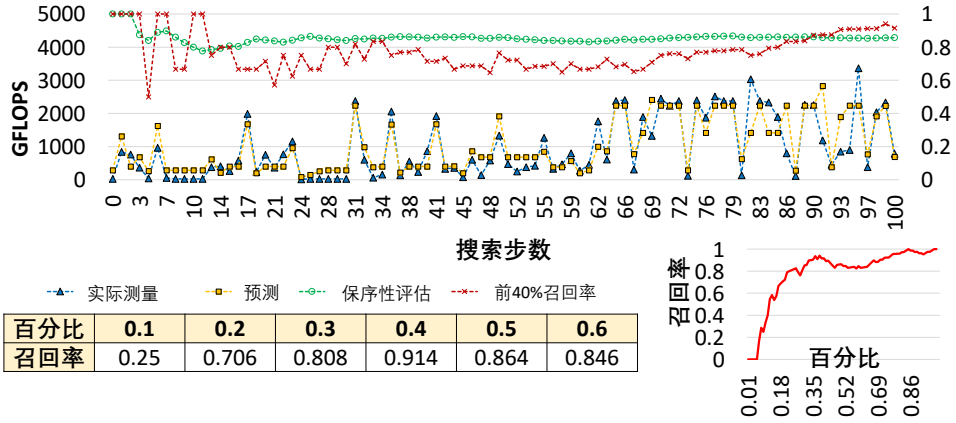
5.7.1　实验设置条件

本文使用三款 AI 芯片评估 AMOS，这些芯片包含包括 Tensor Core GPU（V100[192]和 A100[143]），使用的指令代表是 *mma_sync*，然后是支持 AVX-512 指令集的 Intel CPU（Xeon(R) Silver 4110），代表性指令是 *_mm512_dpbusds_epi32*，以及 Mali Bifrost GPU（G76[193]），代表性指令是 8 比特的向量计算 *arm_dot*。

接下来的实验中，首先验证 AMOS 性能模型的准确性。然后在 Tensor Core GPU 上评估 AMOS 对单个算子和整个 AI 网络的性能。对于单个算子，实验测试了矩阵向量乘（GMV）、矩阵乘法（GEM）、1 维卷积（C1D）、2 维卷积（C2D）、3 维卷积（C3D）、2 维反卷积（T2D）、分组卷积[191]（GRP）、空洞卷积卷积（DIL）、深度卷积[190]（DEP）、胶囊卷积[178]（CAP）、批处理卷积[194]（BCV）、分组全连接层[195]（GFC）、矩阵均值和方差（MEN 和 VAR）以及矩阵扫描算子[196]（SCN）。测试中所有算子共有 113 种不同输入配置（平均每个算子 7-8 种），对比的时候用相对性能，汇报几何平均加速比。所有输入配置都是从真实的网络中得到的[2,87,138,178,194-195,197]。除了 GPU，本文还在 AVX-512 CPU 和 Mali GPU 上评估 AMOS 对卷积的性能。此外还与 PyTorch[92]、CuDNN[84]、Ansor[26]、AutoTVM[25]、UNIT[27]和 AKG[31]比较性能。对于全图性能测试，本文使用来自图像处理和自然语言处理领域的 AI 算法。它们包括 ShuffleNet[179]、ResNet-18 和 ResNet-50[2]、MobileNet-V1[184]、Bert[87]（base 配置）和 MI-LSTM[180]。

5.7.2　性能建模准确性评估

本文使用 Tensor Core GPU 验证 AMOS 性能模型的准确性。实验根据 V100 的硬件参数[166,192]设置 V100 GPU 的性能模型参数，包括 SM（流多处理器）的数量、一个 SM 内的子核心数量、内存大小和带宽。本文使用来自 ResNet-18 的卷积层进行测试，并在图 5.5中比较不同搜索步数下的实际性能与模型预测性能的差异。实验还在图中展



百分比	0.1	0.2	0.3	0.4	0.5	0.6
召回率	0.25	0.706	0.808	0.914	0.864	0.846

图 5.5　在 Tensor Core GPU 上验证 AMOS 的性能模型准确性，使用的测试例子是卷积

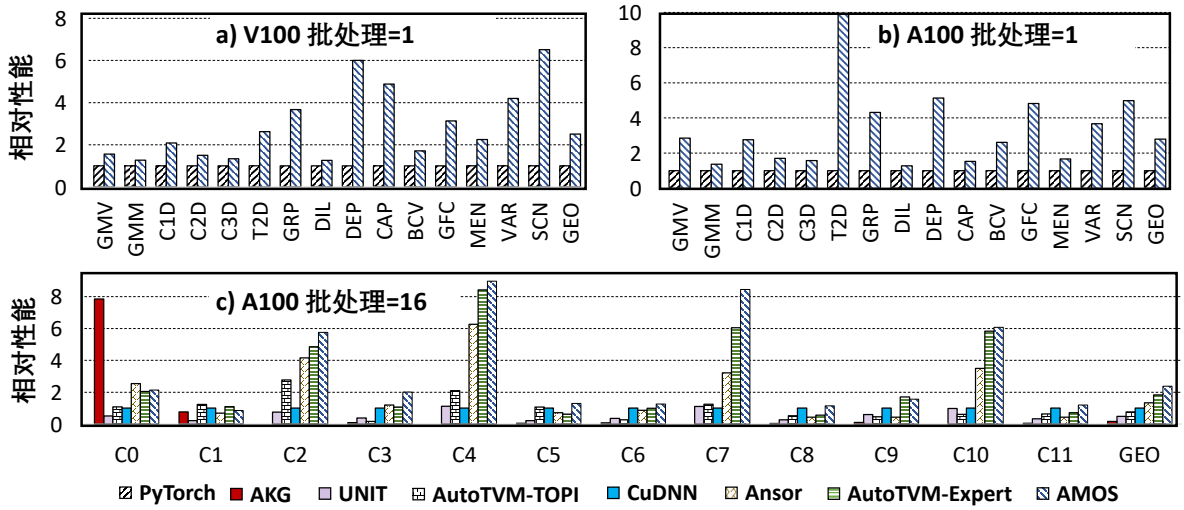


图 5.6　部分 a) 和 b): 和 PyTorch 对比的单算子性能。部分 c): 和 CuDNN 对比 C2D 算子性能，测试硬件是 A100

示了排序准确率（绿线）和性能前 40% 高的映射的召回率（红线）。对于每个搜索步骤，预测的性能在趋势上（而不是绝对值上）接近真实性能。排序准确率显示 AMOS 可以很好地预测映射的相对性能（总体准确性为 85.69%），这表示 AMOS 可以预测哪些映射性能更好，哪些性能不好。前 40% 召回率显示 AMOS 可以以高概率（总体召回率为 91.4%）识别性能前 40% 性能好的映射。这个召回率表明 AMOS 的性能模型保留下来的映射以极高的概率就是好的映射。实验还在图 5.5中展示了不同百分比的召回结果。结果表明，AMOS 在搜索过程中可以以超过 80% 的概率召回性能好的映射，这里选择的性能好的映射是指性能能排进前 30% 的映射。基于这个性能模型，AMOS 可以成功过滤掉劣质映射，只选择优秀的映射进行尝试。

表 5.3　对 Tensor Core 找到的不同软硬件映射数目

GMV	GMM	C1D	C2D	C3D	T2D	GRP	DIL
1	1	6	35	180	7	35	35
DEP	**CAP**	**BCV**	**GFC**	**MEN**	**VAR**	**SCN**	
11	105	11	1	1	1	1	

表 5.4　对 ResNet-18 中每层 C2D AMOS 找到的映射方案举例。n, k, p, q, c, r, s 是批处理大小，输出通道数，输出图像高度，输出图像宽度，输入通道数，卷积窗口高度，卷积窗口宽度

层	n	c	k	p	q	r	s	映射方案举例 (主要是计算映射)
C0	16	3	64	112	112	7	7	$[i_1, i_2, r_1] \leftarrow [(n*112+q)\ \%\ 16, k\ \%\ 16, (c*49+r*7+s)\ \%\ 16]$
C1	16	64	64	56	56	3	3	$[i_1, i_2, r_1] \leftarrow [(n*56+q)\ \%\ 16, k\ \%\ 16, (c*3+r)\ \%\ 16]$
C2	16	64	64	56	56	1	1	$[i_1, i_2, r_1] \leftarrow [(p*56+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C3	16	64	128	28	28	3	3	$[i_1, i_2, r_1] \leftarrow [(n*784+p*28+q)\ \%\ 16, k\ \%\ 16, (c*3+s)\ \%\ 16]$
C4	16	64	128	28	28	1	1	$[i_1, i_2, r_1] \leftarrow [(p*28+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C5	16	128	128	28	28	3	3	$[i_1, i_2, r_1] \leftarrow [(p*28+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C6	16	128	256	14	14	3	3	$[i_1, i_2, r_1] \leftarrow [n\ \%\ 16, k\ \%\ 16, (c*3+s)\ \%\ 16]$
C7	16	128	256	14	14	1	1	$[i_1, i_2, r_1] \leftarrow [(n*196+p*14+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C8	16	256	256	14	14	3	3	$[i_1, i_2, r_1] \leftarrow [(p*14+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C9	16	256	512	7	7	3	3	$[i_1, i_2, r_1] \leftarrow [(n*49+p*7+q)\ \%\ 16, k\ \%\ 16, (c*9+r*3+s)\ \%\ 16]$
C10	16	256	512	7	7	1	1	$[i_1, i_2, r_1] \leftarrow [(n*49+p*7+q)\ \%\ 16, k\ \%\ 16, c\ \%\ 16]$
C11	16	512	512	7	7	3	3	$[i_1, i_2, r_1] \leftarrow [n\ \%\ 16, k\ \%\ 16, (c*9+r*3+s)\ \%\ 16]$

5.7.3　映射生成能力评估

实验还测试了 AMOS 对不同算子寻找软硬件映射的能力，在表 5.3中列出了找到的映射数目。这些不同的映射的区别在于，它们把不同的软件循环映射到了指令循环的不同维度。例如，AMOS 可以为 C3D 生成 180 种不同的映射，它们都需要复杂的数据排布转换和循环映射，一些映射将通道维度和图像高度/宽度/深度维度映射到 Tensor Core，另一些则将三维卷积转换为矩阵乘法，其他映射则首先将三维卷积转换为二维卷积，然后将这些二维卷积再转换为矩阵乘法。

5.7.4　结合算子层编译性能评估

本文将 AMOS 与 PyTorch 对单算子性能进行比较。PyTorch 使用手动优化的库，如 CuDNN[84]、CuBlas[85]和 CUTLASS[96]，以支持不同种类的算子。本文在图 5.6的 a）部分和 b）部分展示了 V100 和 A100 GPU 上批处理大小为 1 情况下的所有算子的性能结果。AMOS 在所有算子上都超越了 PyTorch，分别在 V100 和 A100 上实现了 2.50 倍和 2.80 倍的平均加速比。AMOS 通过映射和调度的联合搜索，做到了在不同 GPU 平台上生成高性能代码。AMOS 之所以能产生加速，是因为它进行了全面的软硬件映射搜索，而 PyTorch 仅使用手动优化库中实现的固定映射方案，这导致它的性能容易变成次优。

本文还与最先进的编译器进行性能比较。AMOS 使用 NCHW 数据格式的 C2D 进行性能测试。测试使用来自 ResNet-18[2]的所有卷积层（共 12 种不同配置），并使用

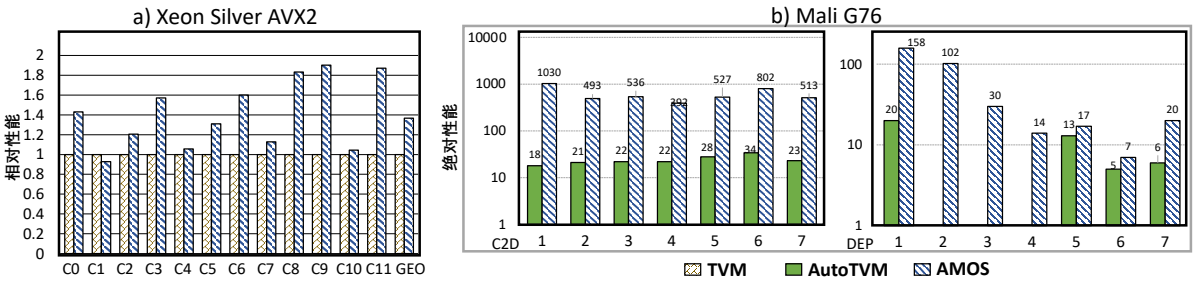


图 5.7　部分 a): 在 Intel Xeon(R) Silver 4110 CPU 上的性能，对比对象是 TVM。部分 b): 在 Mali G76 GPU 上的性能结果，对比对象是 AutoTVM，展示的纵轴是对数值

C0-C11 来表示它们（见表 5.4）。测试结果在图 5.6的 c）部分展示。AMOS 超过了其他编译器，达到了最好的性能（相对于 CuDNN 平均加速比是 2.38 倍，相对于 Ansor 平均加速比是 1.79 倍，相对于 AutoTVM-Expert 平均加速比是 1.30 倍，相对于 UNIT 平均加速比是 4.96 倍）。AKG[31]只将少数层映射到 Tensor Core，因为其多面体模型无法做特殊指令匹配，只能依靠手工设计的模板进行匹配，固定的模板无法识别所有将卷积映射到 Tensor Core 的机会。Ansor[26]没有为 Tensor Core 设计生成代码的规则，所以它生成的代码完全没有用 Tensor Core，它生成的代码全都使用 CUDA Core。由于不同编译器使用不同的优化技术，它们在 CUDA Core 上的性能也不同。UNIT 的手工优化模板总是将 C2D 的高度和宽度维度映射到 Tensor Core 上，并忽略批处理维度，这导致并行度低，因此比 AMOS 慢得多。AutoTVM[25]的手写模板仅为 NHWC/HWNC 数据排布设计映射到 Tensor Core 的方案，而对 NCHW 排布没有支持，因此也没有使用到 Tensor Core。为了和 AutoTVM 对比，实验专门做了 NHWC 数据排布的测试，与 AutoTVM 比较，AMOS 的平均加速比是 2.83 倍。NCHW 数据格式在像 PyTorch 这样的框架中广泛使用，AutoTVM 不支持这种 layout 势必会带来性能损失。AMOS 不受任何特定数据排布格式的限制，就会通用很多。此外，本文还为 NCHW 数据排布手动添加一个 AutoTVM 新模板（采用半精度，称为 AutoTVM-Expert），这样就能比较 AutoTVM 和 AMOS 在 NCHW 格式下的性能。尽管已经尽力优化了 AutoTVM 的模板，它的总体性能仍然不如 AMOS，这是因为模板采用了固定映射策略，并不能自动适配不同的输入形状，而 AMOS 当场生成新的调度和映射方案，能自动适配不同的输入形状。

AMOS 最终为 12 个卷积选择了 8 种不同类型的映射（如表 5.4所示）。例如，C7 的映射将循环n、p、q 映射到i_1，并将循环c 映射到r_1。其他编译器没有实现这种映射，因为它们的模板是固定的，而这些映射往往很少有人尝试，开发者更倾向于遵守既有的经验，例如，UNIT 的模板只将迭代p、q 映射到i_1。

CPU 测试结果：在 Intel CPU 上，AMOS 使用 AVX-512 VNNI intrinsic（一种矩阵-向量

乘法）为 C2D 生成代码。对比对象是 TVM，TVM 使用手写模板生成 VNNI intrinsic。图 5.7的 a) 部分展示了比较结果。C0-C11 指的是表 5.4中的 C2D 算子。AMOS 在除了 C2 的所有测试中都超过了 TVM, 平均来说，相对于 TVM 的加速比是 1.37 倍。

Mali GPU 测试结果：在 Mali GPU 上（Bifrost 架构[193]），AMOS 使用了点乘计算指令，与 AutoTVM 进行性能比较，实验使用的算子是 C2D 和 DEP。C2D 和 DEP 的输入配置来自 MobileNet-V2[174]（共 7 个深度卷积层）。AutoTVM 有为 Bifrost 架构专门设计的手写模板，但是这个模板不支持使用点乘指令。实验结果显示，AutoTVM 对某些 DEP 算子（第 2、3、4 个）性能很差，有时甚至因内部错误无法生成代码。图 5.7的 b) 部分展示的是绝对性能结果，由于 AMOS 远超 AutoTVM，（加速比高达 25.04 倍），图里使用了对数坐标。

新加速器：为了展示 AMOS 的通用性，实验进一步使用三种新的加速器原型进行测试，尽管这些加速器不是真实的，但是他们能体现 AMOS 对于不同指令的适配能力。这些加速器分别支持不同的 intrinsic。实验选择了常数向量乘（AXPY）、矩阵向量乘（GEMV）和逐点卷积（CONV）这三种计算设计 intrinsic，因为它们正好对应于 BLAS[198]操作的三个级别（GEMM 是第三级别的，已在 Tensor Core 实验中演示了，所以这部分使用更复杂的 CONV 作为第三级的代表）。实验使用 C3D 作为测试例子。AMOS 可以为 AXPY 加速器、GEMV 加速器和 CONV 加速器分别找到 15 种、7 种和 31 种不同类型的映射。例如，对 CONV 加速器的一个映射是将输出通道、高度、宽度和输入通道映射到卷积单元，其他维度则作为外层循环被调度优化。这个实验显示，AMOS 确实适用于具有不同 intrinsic 的新一代加速器，证明了 AMOS 在指令层编译技术的通用性。

5.7.5　结合图层编译性能评估

AMOS 能够使更多的算子被映射到 Tensor Core，并且与手动调优的库和模板相比，AMOS 能为这些算子找到更好的映射。本文使用批处理大小 1 和 16 两种场景来评估完整网络端到端的性能。在图 5.8的 a) 到 d) 部分中展示对比结果。由于内存限制，V100 上 Bert 的批处理大小到不了 16，所以略掉了。AMOS 在所有测试中都超过了 PyTorch，除了在 A100 上批处理大小为 16 的 Bert（加速范围从 0.91× 到 10.42×）。Bert 主要由矩阵乘法组成。而矩阵乘法库[85,96]的优化已经持续了几十年，在输入形状较大的时候，很难被超越。即便如此，AMOS 仍然达到了库性能的 90% 以上。对 ShuffleNet 的显著加速来自于对分组卷积（GRP）和深度卷积（DEP）的加速，这些算子在手动调优库中还没有相应的 Tensor Core 实现，或者实现并不高效。

本文还与 UNIT 和 TVM 进行了比较，测试的网络使用了不同批处理大小，图中的 bs 表示批处理大小，比如 bs16 表示批处理大小是 16。实验使用 ResNet-18、ResNet-50

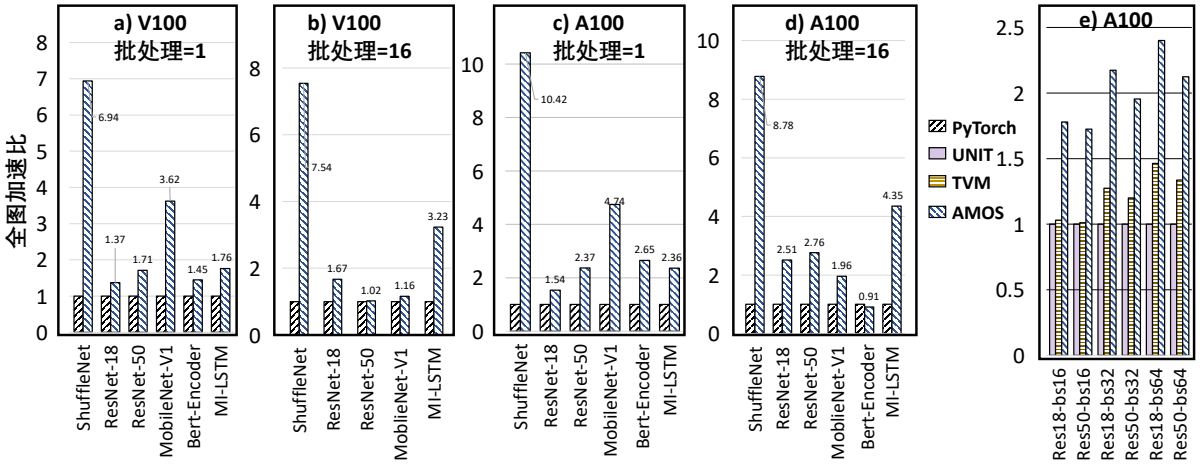


图 5.8　从部分 a) 到部分 d): 在 V100 和 A100 GPU 上和 PyTorch 对比的全图性能。部分 e): 在 A100 GPU 上和 TVM 以及 UNIT 对比全图性能

和 MobileNet-V1 这三个网络。图 5.8的 e) 部分显示了在 A100 上的结果。在大多情况下，AMOS 都是性能更好的那一个。对于 ResNet，AMOS 的全图加速来自于卷积的加速，特别是步长大于 1 的卷积。步长大于 1 的卷积的访存不是连续的，TVM 还处理不了这样的情况，所以完全使用不了 Tensor Core。UNIT 的模板在映射时没有考虑处理批处理维度，所以它的性能不如 TVM 和 AMOS。

5.8　本章小结

本章介绍了指令层编译技术，为了面向多种 AI 芯片的特殊指令进行代码生成，本文提供了指令层级的抽象，并相应地定义软硬件映射。通过两步映射生成方法以及映射正确性验证算法，本文能够生成多种映射方案。结合图层次编译和算子层编译，可以实现全图向特殊指令的映射，结合算子层的自动调度优化技术，可以搜索出最优的映射方案，在实际芯片上测试多种算子和网络，都能证明本文的技术可以达到超越先前工作的性能。

第六章　总结与展望

6.1　本文内容总结

随着 AI 算法的飞速发展，专用 AI 芯片已经成为了 AI 算法训练和推理必不可少的设备。AI 芯片种类众多，且大部分初创公司的新兴芯片都缺少良好的软件栈支持，如果依赖人工开发算子库以及上层框架，迭代周期过长，难以满足实际使用需求。为了解决这一问题，AI 编译技术应运而生。AI 编译涵盖图层，算子层，指令层三个层次，每个层次都有独特的编译抽象方法和优化技术。本文针对这三个层面，全栈式地从上到下介绍了在每个层次的技术细节以及测试结果。总结来说，本文的内容包含以下三个部分：

1. **本文提出了以块为抽象的图层次融合优化框架 Chimera 和 TileFlow：**Chimera 通过将算子分解为块，将算子融合问题抽象为块调度问题。本文提出了一种计算数据搬移量的方法，用来评估不同块间调度方式的性能，结合最优化方法，本文可以确定性地求解出最优或近似最优的分块策略、调度顺序。本文进一步提出 TileFlow 性能模型，与 Chimera 框架结合，帮助 Chimera 分析通用融合场景下的性能。经过验证，TileFlow 框架可以做到高精度性能预测。TileFlow 还提供启发式与机器学习相结合的搜索算法，帮助寻找最优的融合数据流设计方案。实验证明，无论在真实芯片还是硬件原型上，Chimera+TileFlow 这一框架都可以达到超越先前工作的优异融合性能。
2. **本文提出了以循环为抽象的算子层次自动调度优化框架 FlexTensor：**FlexTensor 可以自动分析算子的循环结构，生成包括调度原语组合和原语参数调优在内的完整调度设计空间。FlexTensor 还提供了基于模拟退火和强化学习的搜索算法，在巨大的设计空间中可以有效定位较优的调度策略，针对算子代码生成，可以显著提升性能。在与图层次配合下，可以进行自动算子求导，图融合性能评估，以及编译、执行交叠的系统调度。
3. **本文提出了以计算访存为抽象的指令层次自动软硬件映射框架 AMOS：**AMOS 通过对硬件指令进行抽象，自动分析软件映射到硬件的多种可能性。通过两步生成映射的方案，AMOS 可以枚举出许多超越人类设计的映射方案。此外，AMOS 的映射验证算法保证了生成的映射的正确性。AMOS 可以与图层次、算子层次编译技术配合，在全图定位可以映射到特殊指令的算子，并对每个算子生成多版本映射方案，结合调度调优技术寻找最优的软硬件映射生成代码。实验证明，AMOS 可以适配多种 AI 芯片的指令，并且生成代码无论对手写库还是先前编

译器都有明显的性能提升。

三个层次相互配合，构成完整地端到端编译流水线。端到端编译流程指的是从高层次 AI 算法的计算图描述生成 AI 芯片的底层代码的过程。本文提出的四个框架可以通过两遍编译分析完成端到端编译。

在第一遍分析中，Chimera 首先遍历全图，寻找可以被融合的子图，标记其中的计算密集型算子链，将非计算密集型算子链传递到算子层 FlexTensor 进行融合后算子性能估计，如果是训练场景，FlexTensor 还要完成自动求导，经过 FlexTensor 评估的子图进一步传递到指令层 AMOS，AMOS 寻找匹配可以生成特殊指令的算子，并在其中匹配不同使用指令的映射方案。

在第二遍分析中，Chimera 使用上一次分析的融合性能估计选择融合非计算密集型算子链，对于计算密集型算子链，通过 Chimera 和 TileFlow 自身的数据搬移量分析和优化搜索功能完成程序优化和变换。根据第一次分析得到的使用特殊指令的映射方案，在算子中插入可替换微内核标记可以使用特殊指令的子程序。融合后的算子传递到 FlexTensor，FlexTensor 对每一个算子生成调度空间并进行调度搜索，根据搜索到的高性能调度策略生成代码，对于被可替换微内核标记的部分，传递到指令层 AMOS，由 AMOS 根据映射方案生成特殊指令，从而完成全部代码生成。

6.2 未来工作展望

AI 编译技术依托于成熟的传统编译技术，将形式化和自动化技术应用于 AI 算法优化和部署。对于其未来工作的展望，主要考虑未来 AI 算法发展和未来 AI 芯片发展。本文认为，未来 AI 算法发展有三点趋势：混合专家模型架构将成为主流、多模型配合场景愈发重要、AI 算法小型化与巨型化将同步推进。同时，未来 AI 芯片发展也有三点趋势：多芯片互联将愈发普遍、确定性数据流架构将再次兴起、异构化算力将更加普及。针对这六点预测，本文提出未来工作如下：

1. **利用编译技术优化混合专家模型：**由于存储墙问题，稠密模型参数量增大的代价越来越大，未来混合专家模型（Mixture of Expert，MoE）将成为模型架构主流。随着生成式 AI 进一步发展，未来 AI 算法的基本组成算子变动将会减少，主要的算子以矩阵乘法及其变种构成。AI 算法的复杂性将体现在如何组合这些矩阵乘法及其变种算子。在当前生成式 AI 算法中，矩阵乘法占据计算量比重最大。矩阵乘法主要分布在自相关（self-attention）层和前馈网络层（Feed Forward Network，FFN），二者之中，FFN 中的矩阵乘法占比更多。对于 self-attention 中的冗余计算的研究相对较多，涌现了大量的 attention 优化，KV Cache 优化工作。对于 FFN，则是主要探索结构化稀疏，体现为将完整地 FFN 拆解为动态路由的

多个小 FFN，每个 FFN 就是一个专家，多个专家一起构成 MoE 架构。MoE 架构具有代表性的工作有 Mixtral[199]的对称专家和 Deepseek-MoE[200]的非对称专家，并且当前 MoE 模型架构尚未收敛，设计空间仍然较大，可以预见未来模型架构变动仍会较多，每一种架构都会带来性能优化问题，主要体现在多个专家如何共享硬件资源，专家间的通信开销和计算开销如何相互隐藏等，这个问题依赖手工优化的库函数很难快速解决，因此这是一个使用编译技术进行自动优化的合适问题。解决该问题，核心是定义 MoE 设计空间，将不同的设计维度进行分离，然后为每个设计维度定制编译优化技术完成对全设计空间的覆盖。

2. **应用驱动的模型编译器技术：**尽管当前大模型发展迅速，很多模型取得了令人赞叹的优秀成绩，各个厂商仍然不能依赖某一单独模型完成应用和产品。在实际应用中，需要多个模型相互配合，这种配合可以是多模态的，如 GPT4o 的图像、语音、文字配合能力，也可以是多任务的，如智能助手功能需要将语音提出的指令翻译为对多个应用的调用甚至是级联调用。在这种场景下，需要同时优化多个模型，还需要动态决定模型的种类、调用顺序、调用输入和输出。因此，一方面，未来需要多模态情况下的优化技术，其中关于动态形状、多模型静态资源切分、多模型代码生成的工作，就都是适合编译技术的问题；另一方面，对于动态解析指令并生成对多种模型的调用，其问题可以类比于代码生成，只不过这里的指令都是对大模型的某种调用，可以抽象出指令码、操作数等概念，从而将编译技术引入进来。无论这两方面的哪一种，都将是非常值得研究的编译优化问题。
3. **大规模并行切分与模型压缩量化：**这一点其实是两个问题，之所以放在一点是因为他们都是与模型大小相关的问题。一方面，为了提升模型能力，各个公司、厂商、科研机构仍然在不断增大模型参数量和训练数据，以期能按照 Scaling Law[201]获得更好的模型效果；另一方面，大模型难以在个人设备上部署，即使部署在云端，其开销也是巨大的（包含设备费用和电费），因此如何将训练好的模型小型化也是一个重要的问题。在这种背景下，编译技术可以起到作用的机会有两个，对于大模型不断增大，如何在集群中进行高效并行切分以及进行通信-计算延迟隐藏将更为重要，对于并行的六种策略（数据并行、参数并行、流水线并行、专家并行、优化器状态并行、输入序列并行），研究者早已有相关探索，如何系统化整合这些并行策略，不需人力参与就能灵活地针对不同集群配置生成最优的并行策略组合，是一个可以借助编译技术解决的自动化问题。另一个机会是，针对多种多样的模型小型化技术，如量化、稀疏化、蒸馏等，如何提出一个较为统一的框架完成小型化的模型的更改、优化、部署，也是一个蕴

藏着巨大学术潜力和商业潜力的问题。

4. **针对多芯片互联系统的系统编译技术：**从当前芯片发展情况观测，单张芯片的性能很难提升，未来芯片工艺达到成熟的 3nm 后，再继续推进将更加困难，投入和产出比也越来越令人失望，与其只依赖单张芯片性能提升，将多张芯片互联成为一个紧耦合的系统显得更具有吸引力。最近的 Nvidia NVL72 超节点，未来华为的 Ascend 超节点，都在说明这一趋势。虽然芯片互联构成更大的系统看似很直白，其背后的技术需求却并不简单。从编译的角度来说，如何对上层用户暴露一个一致的统一的加速器抽象，是一个重要的问题。想要做到数十个到数百个芯片被抽象为一个统一的系统，其整体行为就如同一个加速器，就需要通过编译技术将单线程程序高效编译为多进程、多芯片程序。此外，由于多芯片互联的规模增大，编程模型也不能只局限在 SPMD，而是需要考虑更为复杂的 MPMD，其本质是对图编译技术在多芯片互联场景下的进一步研究和探索。
5. **数据流架构编译技术：**数据流架构经过数十年的研究，曾对单芯片的特殊功能单元设计起到了极大的推进作用，如 TPU 的矩阵乘法单元，就是一个脉动阵列。但是更大规模的数据流架构技术，一直难以推进，本质原因是硬件的简易化带来的软件开发困难的问题难以解决。但是在未来，数据流架构将再次兴起，主要原因是当前 AI 算法对于算力的更大要求和编译技术的进一步成熟和推广。数据流架构可以通过分散计算、通信、访存来应对内存墙和功耗墙问题，但是这种非冯诺依曼架构编程的复杂性很高，编译器需要根据每个单元的确定性行为来编排指令，通过软件技术保证计算的正确性和高效性。已经落地的数据流架构加速器代表有 Groq[202]和 Dojo[203]，它们都配备了强力的编译和运行时系统配合完成运行。在未来，伴随着对极致性能的进一步追求，这类数据流加速器将愈发普遍，其配套的编译系统也将带来更多的研究问题。
6. **异构芯片、系统、集群的优化问题：**未来的算力必然是异构的，而且在所有的尺度上都是异构的。这样判断的原因有三个。首先，芯片制造愈发定制化，对不同的 AI 计算天然需要不同的加速器单元，当前比较典型的就是矩阵乘法单元和向量处理单元的联合使用。其次，芯片产能越发不足，单独厂商难以供给芯片算力缺口，因此需要多家芯片厂商同时提供算力芯片，而购买芯片的公司就需要将不同厂商的异构算力同时使用，组成异构系统和集群。最后，政治因素和经济因素影响，政治上，被制裁国家将不再信任某个单一国家供应的芯片，被制裁国家也会通过政策法令要求其所管辖的公司必须采购一定数量的本土算力芯片，这在客观上造成了企业或政府的算力中心算力异构化，经济上，芯片采购方不再满足于被芯片提供方攫取大量利润，而转向自研芯片，也会造成算力

的异构化。在异构化算力场景中，编译技术尤为重要，通过编译技术可以进行多种算力资源的组合，将原本需要人力解决的组合优化问题变为由编译器自动完成，体现在不同层面，可以是代码生成的异构化，调度优化的异构化，集群并行和调度的异构化等。

参考文献

[1] LECUN Y, BOTTOU L, BENGIO Y, Gradient-based learning applied to document recognition. Proc. IEEE, 1998, 86(11): 2278-2324. https://doi.org/10.1109/5.726791. DOI: 10.1109/5.726791.

[2] HE K, ZHANG X, REN S, Deep Residual Learning for Image Recognition//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. 2016: 770-778. https://doi.org/10.1109/CVPR.2016.90. DOI: 10.1109/CVPR.2016.90.

[3] XIE S, GIRSHICK R B, DOLLÁR P, Aggregated Residual Transformations for Deep Neural Networks//2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017. IEEE Computer Society, 2017: 5987-5995. https://doi.org/10.1109/CVPR.2017.634. DOI: 10.1109/CVPR.2017.634.

[4] SAK H, SENIOR A W, BEAUFAYS F. Long short-term memory recurrent neural network architectures for large scale acoustic modeling//LI H, MENG H M, MA B, INTERSPEECH 2014, 15th Annual Conference of the International Speech Communication Association, Singapore, September 14-18, 2014. ISCA, 2014: 338-342. http://www.isca-speech.org/archive/interspeech_2014/i14_0338.html.

[5] VASWANI A, SHAZEER N, PARMAR N, Attention is all you need. Advances in neural information processing systems, 2017, 30: 5998-6008.

[6] ROMBACH R, BLATTMANN A, LORENZ D, High-Resolution Image Synthesis with Latent Diffusion Models//IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022. IEEE, 2022: 10674-10685. https://doi.org/10.1109/CVPR52688.2022.01042. DOI: 10.1109/CVPR52688.2022.01042.

[7] TOUVRON H, MARTIN L, STONE K, Llama 2: Open Foundation and Fine-Tuned Chat Models. CoRR, 2023, abs/2307.09288. arXiv: 2307.09288. https://doi.org/10.48550/arXiv.2307.09288. DOI: 10.48550/ARXIV.2307.09288.

[8] TRACK E K, FORBES N, STRAWN G O. The End of Moore's Law. Comput. Sci. Eng., 2017, 19(2): 4-6. https://doi.org/10.1109/MCSE.2017.25. DOI: 10.1109/MCSE.2017.25.

[9] JOUPPI N P, YOUNG C, PATIL N, In-Datacenter Performance Analysis of a Tensor Processing Unit//Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017. ACM, 2017: 1-12. https://doi.org/10.1145/3079856.3080246. DOI: 10.1145/3079856.3080246.

[10] JOUPPI N P, YOON D H, ASHCRAFT M, Ten Lessons From Three Generations Shaped Google's TPUv4i : Industrial Product//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 1-14. https://doi.org/10.1109/ISCA52012.2021.00010. DOI: 10.1109/ISCA52012.2021.00010.

[11] LI J JIANG Z. Performance Analysis of Cambricon MLU100//GAO W, ZHAN J, FOX G C, Lecture Notes in Computer Science: Benchmarking, Measuring, and Optimizing - Second BenchCouncil International Symposium, Bench 2019, Denver, CO, USA, November 14-16, 2019, Revised

Selected Papers: vol. 12093. Springer, 2019: 57-66. https://doi.org/10.1007/978-3-030-49556-5_5. DOI: 10.1007/978-3-030-49556-5_5.

[12] MARKIDIS S, CHIEN S W D, LAURE E, NVIDIA Tensor Core Programmability, Performance & Precision. CoRR, 2018, abs/1803.04014. arXiv: 1803.04014. http://arxiv.org/abs/1803.04014.

[13] LIAO H, TU J, XIA J, Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing : Industry Track Paper//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021. IEEE, 2021: 789-801. https://doi.org/10.1109/HPCA51647.2021.00071. DOI: 10.1109/HPCA51647.2021.00071.

[14] AGARAP A F. Deep Learning using Rectified Linear Units (ReLU). CoRR, 2018, abs/1803.08375. arXiv: 1803.08375. http://arxiv.org/abs/1803.08375.

[15] REDMON J, DIVVALA S K, GIRSHICK R B, You Only Look Once: Unified, Real-Time Object Detection//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. 2016: 779-788. https://doi.org/10.1109/CVPR.2016.91. DOI: 10.1109/CVPR.2016.91.

[16] WILLIAMS S, WATERMAN A, PATTERSON D A. Roofline: an insightful visual performance model for multicore architectures. Commun. ACM, 2009, 52(4): 65-76. https://doi.org/10.1145/1498765.1498785. DOI: 10.1145/1498765.1498785.

[17] CHEN T, MOREAU T, JIANG Z, TVM: An Automated End-to-End Optimizing Compiler for Deep Learning//13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018. 2018: 578-594. https://www.usenix.org/conference/osdi18/presentation/chen.

[18] JIA Z, PADON O, THOMAS J J, TASO: optimizing deep learning computation with automatic generation of graph substitutions//BRECHT T WILLIAMSON C. Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. ACM, 2019: 47-62. https://doi.org/10.1145/3341301.3359630. DOI: 10.1145/3341301.3359630.

[19] NIU W, GUAN J, WANG Y, DNNFusion: accelerating deep neural networks execution with advanced operator fusion//FREUND S N YAHAV E. PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021. ACM, 2021: 883-898. https://doi.org/10.1145/3453483.3454083. DOI: 10.1145/3453483.3454083.

[20] ZHENG Z, ZHAO P, LONG G, Fusionstitching: boosting memory intensive computations for deep learning workloads. arXiv preprint arXiv:2009.10924, 2020.

[21] NOWATZKI T, SARTIN-TARM M, CARLI L D, A Scheduling Framework for Spatial Architectures Across Multiple Constraint-Solving Theories. ACM Trans. Program. Lang. Syst., 2014, 37(1): 2:1-2:30. https://doi.org/10.1145/2658993. DOI: 10.1145/2658993.

[22] HUANG Q, KALAIAH A, KANG M, CoSA: Scheduling by Constrained Optimization for Spatial Accelerators//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA

2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 554-566. https://doi.org/10.1109/ISCA52012.2021.00050. DOI: 10.1109/ISCA52012.2021.00050.

[23] ZHANG Y, ZHANG N, ZHAO T, SARA: Scaling a Reconfigurable Dataflow Accelerator//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 1041-1054. https://doi.org/10.1109/ISCA52012.2021.00085. DOI: 10.1109/ISCA52012.2021.00085.

[24] XIAO Q, ZHENG S, WU B, HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 1055-1068. https://doi.org/10.1109/ISCA52012.2021.00086. DOI: 10.1109/ISCA52012.2021.00086.

[25] CHEN T, ZHENG L, YAN E Q, Learning to Optimize Tensor Programs//Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. 2018: 3393-3404. http://papers.nips.cc/paper/7599-learning-to-optimize-tensor-programs.

[26] ZHENG L, JIA C, SUN M, Ansor: Generating High-Performance Tensor Programs for Deep Learning//14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. USENIX Association, 2020: 863-879. https://www.usenix.org/conference/osdi20/presentation/zheng.

[27] WENG J, JAIN A, WANG J, UNIT: Unifying Tensorized Instruction Compilation//LEE J W, SOFFA M L, ZAKS A. IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2021, Seoul, South Korea, February 27 - March 3, 2021. IEEE, 2021: 77-89. https://doi.org/10.1109/CGO51591.2021.9370330. DOI: 10.1109/CGO51591.2021.9370330.

[28] Specific compiler for linear algebra to optimize tensorflow COMPUTATIONS X D. https://www.tensorflow.org/xla/jit.

[29] SOTOUDEH M, VENKAT A, ANDERSON M J, ISA mapper: a compute and hardware agnostic deep learning compiler//PALUMBO F, BECCHI M, SCHULZ M, Proceedings of the 16th ACM International Conference on Computing Frontiers, CF 2019, Alghero, Italy, April 30 - May 2, 2019. ACM, 2019: 164-173. https://doi.org/10.1145/3310273.3321559. DOI: 10.1145/3310273.3321559.

[30] BAGHDADI R, RAY J, ROMDHANE M B, Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. CoRR, 2018, abs/1804.10694. arXiv: 1804.10694. http://arxiv.org/abs/1804.10694.

[31] ZHAO J, LI B, NIE W, AKG: automatic kernel generation for neural processing units using polyhedral transformations//FREUND S N YAHAV E. PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 20211. ACM, 2021: 1233-1248. https://doi.org/10.1145/3453483.3454106. DOI: 10.1145/3453483.3454106.

[32] PARASHAR A, RAINA P, SHAO Y S, Timeloop: A Systematic Approach to DNN Accelerator Evaluation//IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019. IEEE, 2019: 304-315. https://doi.org/10.1109/ISPASS.2019.00042. DOI: 10.1109/ISPASS.2019.00042.

[33] YANG X, GAO M, LIU Q, Interstellar: Using Halide's Scheduling Language to Analyze DNN Accelerators//LARUS J R, CEZE L, STRAUSS K. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020. ACM, 2020: 369-383. https://doi.org/10.1145/3373376.3378514. DOI: 10.1145/3373376.3378514.

[34] KWON H, CHATARASI P, SARKAR V, MAESTRO: A Data-Centric Approach to Understand Reuse, Performance, and Hardware Cost of DNN Mappings. IEEE Micro, 2020, 40(3): 20-29. https://doi.org/10.1109/MM.2020.2985963. DOI: 10.1109/MM.2020.2985963.

[35] LU L, GUAN N, WANG Y, TENET: A Framework for Modeling Tensor Dataflow Based on Relation-centric Notation//48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Valencia, Spain, June 14-18, 2021. IEEE, 2021: 720-733. https://doi.org/10.1109/ISCA52012.2021.00062. DOI: 10.1109/ISCA52012.2021.00062.

[36] Nvidia TensorRT. https://docs.nvidia.com/deeplearning/tensorrt/developer-guide/index.html.

[37] ROESCH J, LYUBOMIRSKY S, KIRISAME M, Relay: A High-Level IR for Deep Learning. CoRR, 2019, abs/1904.08368. arXiv: 1904.08368. http://arxiv.org/abs/1904.08368.

[38] RAGAN-KELLEY J, BARNES C, ADAMS A, Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines//ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013. 2013: 519-530. https://doi.org/10.1145/2491956.2462176. DOI: 10.1145/2491956.2462176.

[39] ADAMS A, MA K, ANDERSON L, Learning to optimize halide with tree search and random programs. ACM Trans. Graph., 2019, 38(4): 121:1-121:12. https://doi.org/10.1145/3306346.3322967. DOI: 10.1145/3306346.3322967.

[40] MULLAPUDI R T, ADAMS A, SHARLET D, Automatically scheduling halide image processing pipelines. ACM Trans. Graph., 2016, 35(4): 83:1-83:11. https://doi.org/10.1145/2897824.2925952. DOI: 10.1145/2897824.2925952.

[41] LATTNER C, PIENAAR J A, AMINI M, MLIR: A Compiler Infrastructure for the End of Moore's Law. CoRR, 2020, abs/2002.11054. arXiv: 2002.11054. https://arxiv.org/abs/2002.11054.

[42] VERDOOLAEGE S. *isl*: An Integer Set Library for the Polyhedral Model//FUKUDA K, van der HOEVEN J, JOSWIG M, Lecture Notes in Computer Science: Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software, Kobe, Japan, September 13-17, 2010. Proceedings: vol. 6327. Springer, 2010: 299-302. https://doi.org/10.1007/978-3-642-15582-6_49. DOI: 10.1007/978-3-642-15582-6_49.

[43] BONDHUGULA U, HARTONO A, RAMANUJAM J, Pluto: A practical and fully automatic polyhedral program optimization system//Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008). 2008.

[44] VASILACHE N, ZINENKO O, THEODORIDIS T, Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. CoRR, 2018, abs/1802.04730. arXiv: 1802.04730. http://arxiv.org/abs/1802.04730.

[45] LATTNER C ADVE V S. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation//2nd IEEE / ACM International Symposium on Code Generation and Optimization

(CGO 2004), 20-24 March 2004, San Jose, CA, USA. IEEE Computer Society, 2004: 75-88. https://doi.org/10.1109/CGO.2004.1281665. DOI: 10.1109/CGO.2004.1281665.

[46] FENG S, HOU B, JIN H, TensorIR: An Abstraction for Automatic Tensorized Program Optimization//AAMODT T M, JERGER N D E, SWIFT M M. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, 2023: 804-817. https://doi.org/10.1145/3575693.3576933. DOI: 10.1145/3575693.3576933.

[47] TILLET P, KUNG H, COX D D. Triton: an intermediate language and compiler for tiled neural network computations//MATTSON T, MUZAHID A, SOLAR-LEZAMA A. Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2019, Phoenix, AZ, USA, June 22, 2019. ACM, 2019: 10-19. https://doi.org/10.1145/3315508.3329973. DOI: 10.1145/3315508.3329973.

[48] PyTorch TorchScript. https://pytorch.org/docs/stable/jit.html.

[49] TRUONG L, BARIK R, TOTONI E, Latte: a language, compiler, and runtime for elegant and efficient deep neural networks//KRINTZ C BERGER E. Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. ACM, 2016: 209-223. https://doi.org/10.1145/2908080.2908105. DOI: 10.1145/2908080.2908105.

[50] WANG Y, XU J, HAN Y, DeepBurning: automatic generation of FPGA-based learning accelerators for the neural network family//Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016. ACM, 2016: 110:1-110:6. https://doi.org/10.1145/2897937.2898003. DOI: 10.1145/2897937.2898003.

[51] LIU Z, DOU Y, JIANG J, Automatic code generation of convolutional neural networks in FPGA implementation//SONG Y, WANG S, NELSON B, 2016 International Conference on Field-Programmable Technology, FPT 2016, Xi'an, China, December 7-9, 2016. IEEE, 2016: 61-68. https://doi.org/10.1109/FPT.2016.7929190. DOI: 10.1109/FPT.2016.7929190.

[52] ABDELOUAHAB K, PELCAT M, SÉROT J, Tactics to Directly Map CNN Graphs on Embedded FPGAs. IEEE Embed. Syst. Lett., 2017, 9(4): 113-116. https://doi.org/10.1109/LES.2017.2743247. DOI: 10.1109/LES.2017.2743247.

[53] KJOLSTAD F, KAMIL S, CHOU S, The tensor algebra compiler. PACMPL, 2017, 1(OOPSLA): 77:1-77:29. https://doi.org/10.1145/3133901. DOI: 10.1145/3133901.

[54] WEI R, SCHWARTZ L, ADVE V S. DLVM: A modern compiler infrastructure for deep learning systems//6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings. OpenReview.net, 2018. https://openreview.net/forum?id=HJxPq4ywz.

[55] ELANGO V, RUBIN N, RAVISHANKAR M, Diesel: DSL for linear algebra and neural net computations on GPUs//GOTTSCHLICH J CHEUNG A. Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. ACM, 2018: 42-51. https://doi.org/10.1145/3211346.3211354. DOI: 10.1145/3211346.3211354.

[56] FROSTIG R, JOHNSON M J, LEARY C. Compiling machine learning programs via high-level tracing. Systems for Machine Learning, 2018.

[57] CYPHERS S, BANSAL A K, BHIWANDIWALLA A, Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning. CoRR, 2018, abs/1801.08058. arXiv: 1801.08058. http://arxiv.org/abs/1801.08058.

[58] ROTEM N, FIX J, ABDULRASOOL S, Glow: Graph Lowering Compiler Techniques for Neural Networks. CoRR, 2018, abs/1805.00907. arXiv: 1805.00907. http://arxiv.org/abs/1805.00907.

[59] LAI Y, CHI Y, HU Y, HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing//BAZARGAN K NEUENDORFFER S. Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019. ACM, 2019: 242-251. https://doi.org/10.1145/3289602.3293910. DOI: 10.1145/3289602.3293910.

[60] LIU Y, WANG Y, YU R, Optimizing CNN Model Inference on CPUs//MALKHI D TSAFRIR D. 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. USENIX Association, 2019: 1025-1040. https://www.usenix.org/conference/atc19/presentation/liu-yizhi.

[61] SRIVASTAVA N K, RONG H, BARUA P, T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations//27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019. IEEE, 2019: 181-189. https://doi.org/10.1109/FCCM.2019.00033. DOI: 10.1109/FCCM.2019.00033.

[62] LAI Y, RONG H, ZHENG S, SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs//IEEE/ACM International Conference On Computer Aided Design, ICCAD 2020, San Diego, CA, USA, November 2-5, 2020. IEEE, 2020: 73:1-73:9. https://doi.org/10.1145/3400302.3415644. DOI: 10.1145/3400302.3415644.

[63] NAKANDALA S, SAUR K, YU G, A Tensor Compiler for Unified Machine Learning Prediction Serving. CoRR, 2020, abs/2010.04804. arXiv: 2010.04804. https://arxiv.org/abs/2010.04804.

[64] MA L, XIE Z, YANG Z, Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks//14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20). 2020: 881-897.

[65] ZHAO J DI P. Optimizing the Memory Hierarchy by Compositing Automatic Transformations on Computations and Data//53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020. IEEE, 2020: 427-441. https://doi.org/10.1109/MICRO50266.2020.00044. DOI: 10.1109/MICRO50266.2020.00044.

[66] SHEN H, ROESCH J, CHEN Z, Nimble: Efficiently compiling dynamic neural networks for model inference. arXiv preprint arXiv:2006.03031, 2020.

[67] FEGADE P, CHEN T, GIBBONS P, Cortex: A Compiler for Recursive Deep Learning Models. arXiv preprint arXiv:2011.01383, 2020.

[68] HU S M, LIANG D, YANG G Y, Jittor: a novel deep learning framework with meta-operators and

unified graph execution. Information Sciences, 2020, 63(222103): 1-21.

[69] ZHU K, ZHAO W Y, ZHENG Z, DISC: A Dynamic Shape Compiler for Machine Learning Workloads//YONEKI E PATRAS P. EuroMLSys@EuroSys 2021, Proceedings of the 1st Workshop on Machine Learning and Systemsg Virtual Event, Edinburgh, Scotland, UK, 26 April, 2021. ACM, 2021: 89-95. https://doi.org/10.1145/3437984.3458838. DOI: 10.1145/3437984.3458838.

[70] WANG H, ZHAI J, GAO M, PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections//15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21). 2021: 37-54.

[71] CHEN Y, MENDIS C, CARBIN M, VeGen: a vectorizer generator for SIMD and beyond// SHERWOOD T, BERGER E, KOZYRAKIS C. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021. ACM, 2021: 902-914. https://doi.org/10.1145/3445814.3446692. DOI: 10.1145/3445814.3446692.

[72] IKARASHI Y, BERNSTEIN G L, REINKING A, Exocompilation for productive programming of hardware accelerators//JHALA R DILLIG I. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. ACM, 2022: 703-718. https://doi.org/10.1145/3519939.3523446. DOI: 10.1145/3519939.3523446.

[73] ZHU H, WU R, DIAO Y, {ROLLER}: Fast and Efficient Tensor Compilation for Deep Learning //16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). 2022: 233-248.

[74] ZHENG Z, YANG X, ZHAO P, AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures//FALSAFI B, FERDMAN M, LU S, ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022. ACM, 2022: 359-373. https://doi.org/10.1145/3503222.3507723. DOI: 10.1145/3503222.3507723.

[75] XING J, WANG L, ZHANG S, Bolt: Bridging the Gap between Auto-tuners and Hardware-native Performance//MARCULESCU D, CHI Y, WU C. Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org, 2022. https://proceedings.mlsys.org/paper/2022/hash/38b3eff8baf56627478ec76a704e9b52-Abstract.html.

[76] SHAO J, ZHOU X, FENG S, Tensor Program Optimization with Probabilistic Programs//NeurIPS. 2022. http://papers.nips.cc/paper_files/paper/2022/hash/e894eafae43e68b4c8dfdacf742bcbf3-Abstract-Conference.html.

[77] ZHENG B, JIANG Z, YU C H, DietCode: Automatic Optimization for Dynamic Tensor Programs //MARCULESCU D, CHI Y, WU C. Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org, 2022. https://proceedings.mlsys.org/paper/2022/hash/fa7cdfad1a5aaf8370ebeda47a1ff1c3-Abstract.html.

[78] ZHENG L, LI Z, ZHANG H, Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning//AGUILERA M K WEATHERSPOON H. 16th USENIX Symposium on Oper-

ating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022. USENIX Association, 2022: 559-578. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

[79] YADAV R, AIKEN A, KJOLSTAD F. DISTAL: the distributed tensor algebra compiler//JHALA R DILLIG I. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. ACM, 2022: 286-300. https://doi.org/10.1145/3519939.3523437. DOI: 10.1145/3519939.3523437.

[80] TANG S, ZHAI J, WANG H, FreeTensor: a free-form DSL with holistic optimizations for irregular tensor programs//JHALA R DILLIG I. PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022. ACM, 2022: 872-887. https://doi.org/10.1145/3519939.3523448. DOI: 10.1145/3519939.3523448.

[81] YE Z, LAI R, SHAO J, SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning//AAMODT T M, JERGER N D E, SWIFT M M. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, 2023: 660-678. https://doi.org/10.1145/3582016.3582047. DOI: 10.1145/3582016.3582047.

[82] HAGEDORN B, FAN B, CHEN H, Graphene: An IR for Optimized Tensor Computations on GPUs //AAMODT T M, JERGER N D E, SWIFT M M. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, 2023: 302-313. https://doi.org/10.1145/3582016.3582018. DOI: 10.1145/3582016.3582018.

[83] BI J, GUO Q, LI X, Heron: Automatically Constrained High-Performance Library Generation for Deep Learning Accelerators//AAMODT T M, JERGER N D E, SWIFT M M. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, 2023: 314-328. https://doi.org/10.1145/3582016.3582061. DOI: 10.1145/3582016.3582061.

[84] Nvidia CuDNN. https://developer.nvidia.com/cudnn.

[85] Nvidia CuBlas. https://developer.nvidia.com/cublas.

[86] Intel oneAPI Deep Neural Network Library. https://github.com/oneapi-src/oneDNN.

[87] DEVLIN J, CHANG M, LEE K, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. CoRR, 2018, abs/1810.04805. arXiv: 1810.04805. http://arxiv.org/abs/1810.04805.

[88] BROWN T B, MANN B, RYDER N, Language models are few-shot learners. arXiv preprint arXiv:2005.14165, 2020.

[89] KAO S, SUBRAMANIAN S, AGRAWAL G, FLAT: An Optimized Dataflow for Mitigating Attention Bottlenecks//AAMODT T M, JERGER N D E, SWIFT M M. Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023. ACM, 2023: 295-310. https://doi.org/10.1145/3575693.3575747. DOI: 10.1145/3575693.3575747.

[90] ALWANI M, CHEN H, FERDMAN M, Fused-layer CNN accelerators//49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016. IEEE Computer Society, 2016: 22:1-22:12. https://doi.org/10.1109/MICRO.2016.7783725. DOI: 10.1109/MICRO.2016.7783725.

[91] ABADI M, BARHAM P, CHEN J, TensorFlow: A System for Large-Scale Machine Learning//12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016. 2016: 265-283. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[92] PASZKE A, GROSS S, MASSA F, PyTorch: An Imperative Style, High-Performance Deep Learning Library//Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. 2019: 8024-8035. http://papers.nips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.

[93] CHEN T, LI M, LI Y, MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. CoRR, 2015, abs/1512.01274. arXiv: 1512.01274. http://arxiv.org/abs/1512.01274.

[94] JIA Y, SHELHAMER E, DONAHUE J, Caffe: Convolutional Architecture for Fast Feature Embedding. arXiv preprint arXiv:1408.5093, 2014.

[95] Intel oneAPI Math Kernel Library. https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

[96] Nvidia CUTLASS. https://github.com/NVIDIA/cutlass.

[97] LAVIN A. Fast Algorithms for Convolutional Neural Networks. CoRR, 2015, abs/1509.09308. arXiv: 1509.09308. http://arxiv.org/abs/1509.09308.

[98] MATHIEU M, HENAFF M, LECUN Y. Fast Training of Convolutional Networks through FFTs. CoRR, 2013, abs/1312.5851. arXiv: 1312.5851. http://arxiv.org/abs/1312.5851.

[99] XIAO Q, LIANG Y, LU L, Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs//Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017. ACM, 2017: 62:1-62:6. https://doi.org/10.1145/3061639.3062244. DOI: 10.1145/3061639.3062244.

[100] LU L LIANG Y. SpWA: an efficient sparse winograd convolutional neural networks accelerator on FPGAs//Proceedings of the 55th Annual Design Automation Conference, DAC 2018, San Francisco, CA, USA, June 24-29, 2018. 2018: 135:1-135:6. https://doi.org/10.1145/3195970.3196120. DOI: 10.1145/3195970.3196120.

[101] XIE X, LIANG Y, LI X, CuLDA: Solving Large-scale LDA Problems on GPUs//Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2019, Phoenix, AZ, USA, June 22-29, 2019. 2019: 195-205. https://doi.org/10.1145/3307681.3325407. DOI: 10.1145/3307681.3325407.

[102] LI X, LIANG Y, YAN S, A coordinated tiling and batching framework for efficient GEMM on GPUs //HOLLINGSWORTH J K KEIDAR I. Proceedings of the 24th ACM SIGPLAN Symposium on

Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. ACM, 2019: 229-241. https://doi.org/10.1145/3293883.3295734. DOI: 10.1145/3293883.3295734.

[103] XIE X, LIANG Y, LI X, Enabling coordinated register allocation and thread-level parallelism optimization for GPUs//Proceedings of the 48th International Symposium on Microarchitecture, MICRO 2015, Waikiki, HI, USA, December 5-9, 2015. 2015: 395-406. https://doi.org/10.1145/2830772.2830813. DOI: 10.1145/2830772.2830813.

[104] GEORGANAS E, AVANCHA S, BANERJEE K, Anatomy of high-performance deep learning convolutions on SIMD architectures//Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018. 2018: 66:1-66:12. http://dl.acm.org/citation.cfm?id=3291744.

[105] VENKAT A, RUSIRA T, BARIK R, SWIRL: High-performance many-core CPU code generation for deep neural networks. The International Journal of High Performance Computing Applications, 2019, 33(6): 1275-1289. eprint: https://doi.org/10.1177/1094342019866247. https://doi.org/10.1177/1094342019866247. DOI: 10.1177/1094342019866247.

[106] Automatically Tuned Linear Algebra Software (ATLAS)//PADUA D A. Encyclopedia of Parallel Computing. Springer, 2011: 101. https://doi.org/10.1007/978-0-387-09766-4_2061. DOI: 10.1007/978-0-387-09766-4_2061.

[107] BELTER G, JESSUP E R, KARLIN I, Automating the generation of composed linear algebra kernels//Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009, November 14-20, 2009, Portland, Oregon, USA. 2009. https://doi.org/10.1145/1654059.1654119. DOI: 10.1145/1654059.1654119.

[108] FRIGO M JOHNSON S G. FFTW: an adaptive software architecture for the FFT//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98, Seattle, Washington, USA, May 12-15, 1998. 1998: 1381-1384. https://doi.org/10.1109/ICASSP.1998.681704. DOI: 10.1109/ICASSP.1998.681704.

[109] FAROOQUI N, ROSSBACH C J, YU Y, Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications//2014 Conference on Timely Results in Operating Systems, TRIOS '14, Broomfield, CO, USA, October 5, 2014. 2014. https://www.usenix.org/conference/trios14/technical-sessions/presentation/farooqui.

[110] DAVE S, KIM Y, AVANCHA S, dMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators. ACM Trans. Embed. Comput. Syst., 2019, 18(5s): 70:1-70:27. https://doi.org/10.1145/3358198. DOI: 10.1145/3358198.

[111] BONDHUGULA U, HARTONO A, RAMANUJAM J, A practical automatic polyhedral parallelizer and locality optimizer//GUPTA R AMARASINGHE S P. Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008. ACM, 2008: 101-113. https://doi.org/10.1145/1375581.1375595. DOI: 10.1145/1375581.1375595.

[112] VERDOOLAEGE S, JUEGA J C, COHEN A, Polyhedral parallel code generation for CUDA. ACM Trans. Archit. Code Optim., 2013, 9(4): 54:1-54:23. https://doi.org/10.1145/2400682.2400713.

DOI: 10.1145/2400682.2400713.

[113] KIRKPATRICK S, JR. C D G, VECCHI M P. Optimization by Simulated Annealing//Science, 220(4598):671-680. 1983.

[114] LI R, XU Y, SUKUMARAN-RAJAM A, Analytical characterization and design space exploration for optimization of CNNs//SHERWOOD T, BERGER E, KOZYRAKIS C. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021. ACM, 2021: 928-942. https://doi.org/10.1145/3445814.3446759. DOI: 10.1145/3445814.3446759.

[115] CHEN Y, KRISHNA T, EMER J S, Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. IEEE J. Solid State Circuits, 2017, 52(1): 127-138. https://doi.org/10.1109/JSSC.2016.2616357. DOI: 10.1109/JSSC.2016.2616357.

[116] DU Z, FASTHUBER R, CHEN T, ShiDianNao: shifting vision processing closer to the sensor //MARR D T ALBONESI D H. Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015. ACM, 2015: 92-104. https://doi.org/10.1145/2749469.2750389. DOI: 10.1145/2749469.2750389.

[117] LU L, JIN Y, BI H, Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture//MICRO '21: 54th Annual IEEE/ACM International Symposium on Microarchitecture, Virtual Event, Greece, October 18-22, 2021. ACM, 2021: 977-991. https://doi.org/10.1145/3466752.3480125. DOI: 10.1145/3466752.3480125.

[118] GENC H, HAJ-ALI A, IYER V, Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. CoRR, 2019, abs/1911.09925. arXiv: 1911.09925. http://arxiv.org/abs/1911.09925.

[119] GOVINDARAJU V, HO C, NOWATZKI T, DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing. IEEE Micro, 2012, 32(5): 38-51. https://doi.org/10.1109/MM.2012.51. DOI: 10.1109/MM.2012.51.

[120] PRABHAKAR R, ZHANG Y, KOEPLINGER D, Plasticine: A Reconfigurable Accelerator for Parallel Patterns. IEEE Micro, 2018, 38(3): 20-31. https://doi.org/10.1109/MM.2018.032271058. DOI: 10.1109/MM.2018.032271058.

[121] LI J, LOURI A, KARANTH A, GCNAX: A Flexible and Energy-efficient Accelerator for Graph Convolutional Neural Networks//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021. IEEE, 2021: 775-788. https://doi.org/10.1109/HPCA51647.2021.00070. DOI: 10.1109/HPCA51647.2021.00070.

[122] KININGHAM K, LEVIS P A, RÉ C. GRIP: A Graph Neural Network Accelerator Architecture. IEEE Trans. Computers, 2023, 72(4): 914-925. https://doi.org/10.1109/TC.2022.3197083. DOI: 10.1109/TC.2022.3197083.

[123] YIN S, OUYANG P, TANG S, A High Energy Efficient Reconfigurable Hybrid Neural Network Processor for Deep Learning Applications. IEEE J. Solid State Circuits, 2018, 53(4): 968-982. https://doi.org/10.1109/JSSC.2017.2778281. DOI: 10.1109/JSSC.2017.2778281.

[124] KAO S, JEONG G, KRISHNA T. ConfuciuX: Autonomous Hardware Resource Assignment for

DNN Accelerators using Reinforcement Learning//53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020. IEEE, 2020: 622-636. https://doi.org/10.1109/MICRO50266.2020.00058. DOI: 10.1109/MICRO50266.2020.00058.

[125] ZHENG S, CHEN R, WEI A, AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction//SALAPURA V, ZAHRAN M, CHONG F, ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022. ACM, 2022: 874-887. https://doi.org/10.1145/3470496.3527440. DOI: 10.1145/3470496.3527440.

[126] HEGDE K, TSAI P, HUANG S, Mind mappings: enabling efficient algorithm-accelerator mapping space search//SHERWOOD T, BERGER E, KOZYRAKIS C. ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Virtual Event, USA, April 19-23, 2021. ACM, 2021: 943-958. https://doi.org/10.1145/3445814.3446762. DOI: 10.1145/3445814.3446762.

[127] ZHENG S, ZHANG X, LIU L, Atomic Dataflow based Graph-Level Workload Orchestration for Scalable DNN Accelerators//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022. IEEE, 2022: 475-489. https://doi.org/10.1109/HPCA53966.2022.00042. DOI: 10.1109/HPCA53966.2022.00042.

[128] SAMAJDAR A, JOSEPH J M, ZHU Y, A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim//IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020. IEEE, 2020: 58-68. https://doi.org/10.1109/ISPASS48437.2020.00016. DOI: 10.1109/ISPASS48437.2020.00016.

[129] WU Y N, TSAI P, PARASHAR A, Sparseloop: An Analytical Approach To Sparse Tensor Accelerator Modeling//55th IEEE/ACM International Symposium on Microarchitecture, MICRO 2022, Chicago, IL, USA, October 1-5, 2022. IEEE, 2022: 1377-1395. https://doi.org/10.1109/MICRO56248.2022.00096. DOI: 10.1109/MICRO56248.2022.00096.

[130] WU Y N, SZE V, EMER J S. An Architecture-Level Energy and Area Estimator for Processing-In-Memory Accelerator Designs//IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2020, Boston, MA, USA, August 23-25, 2020. IEEE, 2020: 116-118. https://doi.org/10.1109/ISPASS48437.2020.00024. DOI: 10.1109/ISPASS48437.2020.00024.

[131] YANG T, CHEN Y, EMER J S, A method to estimate the energy consumption of deep neural networks//MATTHEWS M B. 51st Asilomar Conference on Signals, Systems, and Computers, ACSSC 2017, Pacific Grove, CA, USA, October 29 - November 1, 2017. IEEE, 2017: 1916-1920. https://doi.org/10.1109/ACSSC.2017.8335698. DOI: 10.1109/ACSSC.2017.8335698.

[132] KAO S KRISHNA T. MAGMA: An Optimization Framework for Mapping Multiple DNNs on Multiple Accelerator Cores//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022. IEEE, 2022: 814-830. https://doi.org/10.1109/HPCA53966.2022.00065. DOI: 10.1109/HPCA53966.2022.00065.

[133] KE L, HE X, ZHANG X. NNest: Early-Stage Design Space Exploration Tool for Neural Network Inference Accelerators//Proceedings of the International Symposium on Low Power Electronics

and Design, ISLPED 2018, Seattle, WA, USA, July 23-25, 2018. ACM, 2018: 4:1-4:6. https://doi.org/10.1145/3218603.3218647. DOI: 10.1145/3218603.3218647.

[134] ZHAO Y, LI C, WANG Y, DNN-Chip Predictor: An Analytical Performance Predictor for DNN Accelerators with Various Dataflows and Hardware Architectures//2020 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2020, Barcelona, Spain, May 4-8, 2020. IEEE, 2020: 1593-1597. https://doi.org/10.1109/ICASSP40776.2020.9053977. DOI: 10.1109/ICASSP40776.2020.9053977.

[135] KWON H, LAI L, PELLAUER M, Heterogeneous Dataflow Accelerators for Multi-DNN Workloads//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2021, Seoul, South Korea, February 27 - March 3, 2021. IEEE, 2021: 71-83. https://doi.org/10.1109/HPCA51647.2021.00016. DOI: 10.1109/HPCA51647.2021.00016.

[136] ZHANG X, HAO C, ZHOU P, H2H: heterogeneous model to heterogeneous system mapping with computation and communication awareness//OSHANA R. DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022. ACM, 2022: 601-606. https://doi.org/10.1145/3489517.3530509. DOI: 10.1145/3489517.3530509.

[137] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. ImageNet Classification with Deep Convolutional Neural Networks//BARTLETT P L, PEREIRA F C N, BURGES C J C, Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States. 2012: 1106-1114. http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.

[138] REN S, HE K, GIRSHICK R B, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks//CORTES C, LAWRENCE N D, LEE D D, Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. 2015: 91-99. http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.

[139] DOSOVITSKIY A, BEYER L, KOLESNIKOV A, An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale//9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021. OpenReview.net, 2021. https://openreview.net/forum?id=YicbFdNTTy.

[140] Huawei Compute Architecture for Neural Networks (CANN). https://e.huawei.com/hk/products/cloud-computing-dc/atlas/cann.

[141] WAHIB M MARUYAMA N. Scalable Kernel Fusion for Memory-Bound GPU Applications//DAMKROGER T DONGARRA J J. International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2014, New Orleans, LA, USA, November 16-21, 2014. IEEE Computer Society, 2014: 191-202. https://doi.org/10.1109/SC.2014.21. DOI: 10.1109/SC.2014.21.

[142] SIVATHANU M, CHUGH T, SINGAPURAM S S, Astra: Exploiting Predictability to Optimize Deep Learning//BAHAR I, HERLIHY M, WITCHEL E, Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019. ACM, 2019: 909-923. https://doi.org/10

.1145/3297858.3304072. DOI: 10.1145/3297858.3304072.

[143] Nvidia Ampere Whitepaper. https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf.

[144] LI R, XU Y, SUKUMARAN-RAJAM A, Analytical Characterization and Design Space Exploration for Optimization of CNNs. CoRR, 2021, abs/2101.09808. arXiv: 2101.09808. https://arxiv.org/abs/2101.09808.

[145] LOW T M, IGUAL F D, SMITH T M, Analytical Modeling Is Enough for High-Performance BLIS. ACM Trans. Math. Softw., 2016, 43(2). https://doi.org/10.1145/2925987. DOI: 10.1145/2925987.

[146] WU Y N, EMER J S, SZE V. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs//PAN D Z. Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019. ACM, 2019: 1-8. https://doi.org/10.1109/ICCAD45719.2019.8942149. DOI: 10.1109/ICCAD45719.2019.8942149.

[147] SILVER D, HUANG A, MADDISON C J, Mastering the game of Go with deep neural networks and tree search. Nat., 2016, 529(7587): 484-489. https://doi.org/10.1038/nature16961. DOI: 10.1038/nature16961.

[148] ZHENG S, LIANG Y, WANG S, FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System//LARUS J R, CEZE L, STRAUSS K. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]. ACM, 2020: 859-873. https://doi.org/10.1145/3373376.3378508. DOI: 10.1145/3373376.3378508.

[149] TOLSTIKHIN I O, HOULSBY N, KOLESNIKOV A, MLP-Mixer: An all-MLP Architecture for Vision. CoRR, 2021, abs/2105.01601. arXiv: 2105.01601. https://arxiv.org/abs/2105.01601.

[150] IANDOLA F N, HAN S, MOSKEWICZ M W, SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size. arXiv preprint arXiv:1602.07360, 2016.

[151] REDMON J FARHADI A. Yolov3: An incremental improvement. arXiv preprint arXiv:1804.02767, 2018.

[152] SNYDER WVerilator. https://www.veripool.org/verilator/.

[153] YANG Y, EMER J S, SÁNCHEZ D. ISOSceles: Accelerating Sparse CNNs through Inter-Layer Pipelining//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023. IEEE, 2023: 598-610. https://doi.org/10.1109/HPCA56546.2023.10071080. DOI: 10.1109/HPCA56546.2023.10071080.

[154] ZHENG S, CHEN S, SONG P, Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion//IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023. IEEE, 2023.

[155] SZEGEDY C, VANHOUCKE V, IOFFE S, Rethinking the Inception Architecture for Computer Vision//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016: 2818-2826. https://doi.org/10.1109/CVPR.2016.308. DOI: 10.1109/CVPR.2016.308.

[156] ZHANG J LI J. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network//Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2017, Monterey, CA, USA, February 22-24, 2017. 2017: 25-34. http://dl.acm.org/citation.cfm?id=3021698.

[157] ANDREW A M. *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto, Adaptive Computation and Machine Learning series, MIT Press (Bradford Book), Cambridge, Mass., 1998, xviii + 322 pp, ISBN 0-262-19398-1, (hardback, £31.95). Robotica, 1999, 17(2): 229-235. http://journals.cambridge.org/action/displayAbstract?aid=34601.

[158] WATKINS C J C H DAYAN P. Technical Note Q-Learning. Machine Learning, 1992, 8: 279-292. https://doi.org/10.1007/BF00992698. DOI: 10.1007/BF00992698.

[159] MNIH V, KAVUKCUOGLU K, SILVER D, Human-level control through deep reinforcement learning. Nature, 2015, 518(7540): 529-533. https://doi.org/10.1038/nature14236. DOI: 10.1038/nature14236.

[160] ZEILER M D. ADADELTA: An Adaptive Learning Rate Method. CoRR, 2012, abs/1212.5701. arXiv: 1212.5701. http://arxiv.org/abs/1212.5701.

[161] SHAO Y S, REAGEN B, WEI G, Aladdin: A pre-RTL, power-performance accelerator simulator enabling large design space exploration of customized architectures//ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014. 2014: 97-108. https://doi.org/10.1109/ISCA.2014.6853196. DOI: 10.1109/ISCA.2014.6853196.

[162] LIANG Y, WANG S, ZHANG W. FlexCL: A Model of Performance and Power for OpenCL Workloads on FPGAs. IEEE Trans. Computers, 2018, 67(12): 1750-1764. https://doi.org/10.1109/TC.2018.2840686. DOI: 10.1109/TC.2018.2840686.

[163] KIM D, RENGANARAYANAN L, ROSTRON D, Multi-level tiling: M for the price of one// Proceedings of the ACM/IEEE Conference on High Performance Networking and Computing, SC 2007, November 10-16, 2007, Reno, Nevada, USA. 2007: 51. https://doi.org/10.1145/1362622.1362691. DOI: 10.1145/1362622.1362691.

[164] URBAN S van der SMAGT P. Automatic Differentiation for Tensor Algebras. CoRR, 2017, abs/1711.01348. arXiv: 1711.01348. http://arxiv.org/abs/1711.01348.

[165] SHI W, CABALLERO J, HUSZAR F, Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016: 1874-1883. https://doi.org/10.1109/CVPR.2016.207. DOI: 10.1109/CVPR.2016.207.

[166] JIA Z, MAGGIONI M, STAIGER B, Dissecting the NVIDIA Volta GPU Architecture via Microbenchmarking. CoRR, 2018, abs/1804.06826. arXiv: 1804.06826. http://arxiv.org/abs/1804.06826.

[167] NvidiaCUDA Grah. https://developer.nvidia.com/blog/cuda-graphs/. Accessed 2021-11-5. 2021.

[168] CHEN T GUESTRIN C. XGBoost: A Scalable Tree Boosting System//KRISHNAPURAM B, SHAH M, SMOLA A J, Proceedings of the 22nd ACM SIGKDD International Conference on

Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. ACM, 2016: 785-794. https://doi.org/10.1145/2939672.2939785. DOI: 10.1145/2939672.2939785.

[169] LECUN Y, BOSER B E, DENKER J S, Backpropagation Applied to Handwritten Zip Code Recognition. Neural Computation, 1989, 1(4): 541-551. https://doi.org/10.1162/neco.1989.1.4.541. DOI: 10.1162/neco.1989.1.4.541.

[170] KIM Y. Convolutional Neural Networks for Sentence Classification. CoRR, 2014, abs/1408.5882. arXiv: 1408.5882. http://arxiv.org/abs/1408.5882.

[171] TRAN D, BOURDEV L D, FERGUS R, Learning Spatiotemporal Features with 3D Convolutional Networks//2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015. 2015: 4489-4497. https://doi.org/10.1109/ICCV.2015.510. DOI: 10.1109/ICCV.2015.510.

[172] HARA K, KATAOKA H, SATOH Y. Learning Spatio-Temporal Features with 3D Residual Networks for Action Recognition//2017 IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2017, Venice, Italy, October 22-29, 2017. 2017: 3154-3160. https://doi.org/10.1109/ICCVW.2017.373. DOI: 10.1109/ICCVW.2017.373.

[173] CHOLLET F. Xception: Deep Learning with Depthwise Separable Convolutions. CoRR, 2016, abs/1610.02357. arXiv: 1610.02357. http://arxiv.org/abs/1610.02357.

[174] SANDLER M, HOWARD A G, ZHU M, MobileNetV2: Inverted Residuals and Linear Bottlenecks //2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018. 2018: 4510-4520. http://openaccess.thecvf.com/content_cvpr_2018/html/Sandler_MobileNetV2_Inverted_Residuals_CVPR_2018_paper.html. DOI: 10.1109/CVPR.2018.00474.

[175] HTTPS://GITHUB.COM/TORCH/CUNN T. https://github.com/torch/cunn.

[176] HINTON G E, KRIZHEVSKY A, WANG S D. Transforming Auto-Encoders//HONKELA T, DUCH W, GIROLAMI M A, Lecture Notes in Computer Science: Artificial Neural Networks and Machine Learning - ICANN 2011 - 21st International Conference on Artificial Neural Networks, Espoo, Finland, June 14-17, 2011, Proceedings, Part I: vol. 6791. Springer, 2011: 44-51. https://doi.org/10.1007/978-3-642-21735-7_6. DOI: 10.1007/978-3-642-21735-7_6.

[177] SABOUR S, FROSST N, HINTON G E. Dynamic Routing Between Capsules//GUYON I, von LUXBURG U, BENGIO S, Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA. 2017: 3856-3866. http://papers.nips.cc/paper/6975-dynamic-routing-between-capsules.

[178] HINTON G E, SABOUR S, FROSST N. Matrix capsules with EM routing//6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings. OpenReview.net, 2018. https://openreview.net/forum?id=HJWLfGWRb.

[179] ZHANG X, ZHOU X, LIN M, ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. CoRR, 2017, abs/1707.01083. arXiv: 1707.01083. http://arxiv.org/abs/1707.01083.

[180] WU Y, ZHANG S, ZHANG Y, On Multiplicative Integration with Recurrent Neural Networks//LEE D D, SUGIYAMA M, von LUXBURG U, Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain. 2016: 2856-2864. http://papers.nips.cc/paper/6215-on-multiplicative-integration-with-recurrent-neural-networks.

[181] MIKOLOV T, JOULIN A, CHOPRA S, Learning Longer Memory in Recurrent Neural Networks //BENGIO Y LECUN Y. 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings. 2015. http://arxiv.org/abs/1412.7753.

[182] COSTA R P, ASSAEL I A M, SHILLINGFORD B, Cortical microcircuits as gated-recurrent neural networks//GUYON I, von LUXBURG U, BENGIO S, Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA. 2017: 272-283. http://papers.nips.cc/paper/6631-cortical-microcircuits-as-gated-recurrent-neural-networks.

[183] Facebook"PyTorch LLTM Implementation". https://pytorch.org/tutorials/advanced/cpp_extension.html. Accessed 2020-08-16. 2020.

[184] HOWARD A G, ZHU M, CHEN B, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR, 2017, abs/1704.04861. arXiv: 1704.04861. http://arxiv.org/abs/1704.04861.

[185] VASILACHE N, JOHNSON J, MATHIEU M, Fast Convolutional Nets With fbfft: A GPU Performance Evaluation//3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. 2015. http://arxiv.org/abs/1412.7580.

[186] ZHANG C, LI P, SUN G, Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks//FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. Monterey, California, USA: ACM, 2015: 161-170. http://doi.acm.org/10.1145/2684746.2689060. DOI: 10.1145/2684746.2689060.

[187] CHENG Y, YU F X, FERIS R S, An Exploration of Parameter Redundancy in Deep Networks with Circulant Projections//2015 IEEE International Conference on Computer Vision, ICCV 2015, Santiago, Chile, December 7-13, 2015. 2015: 2857-2865. https://doi.org/10.1109/ICCV.2015.327. DOI: 10.1109/ICCV.2015.327.

[188] WANG S, LI Z, DING C, C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs//Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2018, Monterey, CA, USA, February 25-27, 2018. 2018: 11-20. https://doi.org/10.1145/3174243.3174253. DOI: 10.1145/3174243.3174253.

[189] WU B, WAN A, YUE X, Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions //2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018. 2018: 9127-9135. http://openaccess.thecvf.com/content_cvpr_2018/html/Wu_Shift_A_Zero_CVPR_2018_paper.html. DOI: 10.1109/CVPR.2018.00951.

[190] CHOLLET F. Xception: Deep Learning with Depthwise Separable Convolutions. CoRR, 2016,

abs/1610.02357. arXiv: 1610.02357. http://arxiv.org/abs/1610.02357.

[191] HOWARD A G, ZHU M, CHEN B, MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR, 2017, abs/1704.04861. arXiv: 1704.04861. http://arxiv.org/abs/1704.04861.

[192] Nivida Volta Whitepaper. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[193] ARM Mali Bifrost. https://www.hotchips.org/wp-content/uploads/hc_archives/hc28/HC28.22-Monday-Epub/HC28.22.10-GPU-HPC-Epub/HC28.22.110-Bifrost-JemDavies-ARM-v04-9.pdf.

[194] YANG B, BENDER G, LE Q V, Condconv: Conditionally parameterized convolutions for efficient inference//Advances in Neural Information Processing Systems. 2019: 1307-1318.

[195] MA N, ZHANG X, HUANG J, Weightnet: Revisiting the design space of weight networks. arXiv preprint arXiv:2007.11823, 2020.

[196] DAKKAK A, LI C, XIONG J, Accelerating reduction and scan using tensor core units// Proceedings of the ACM International Conference on Supercomputing. 2019: 46-57.

[197] CHEN L, PAPANDREOU G, KOKKINOS I, DeepLab: Semantic Image Segmentation with Deep Convolutional Nets, Atrous Convolution, and Fully Connected CRFs. IEEE Trans. Pattern Anal. Mach. Intell., 2018, 40(4): 834-848. https://doi.org/10.1109/TPAMI.2017.2699184. DOI: 10.1109/TPAMI.2017.2699184.

[198] DONGARRA J J, DU CROZ J, HAMMARLING S, A set of level 3 basic linear algebra subprograms. ACM Transactions on Mathematical Software (TOMS), 1990, 16(1): 1-17.

[199] JIANG A Q, SABLAYROLLES A, ROUX A, Mixtral of Experts. CoRR, 2024, abs/2401.04088. arXiv: 2401.04088. https://doi.org/10.48550/arXiv.2401.04088. DOI: 10.48550/ARXIV.2401.04088.

[200] DAI D, DENG C, ZHAO C, DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models. CoRR, 2024, abs/2401.06066. arXiv: 2401.06066. https://doi.org/10.48550/arXiv.2401.06066. DOI: 10.48550/ARXIV.2401.06066.

[201] KAPLAN J, MCCANDLISH S, HENIGHAN T, Scaling Laws for Neural Language Models. CoRR, 2020, abs/2001.08361. arXiv: 2001.08361. https://arxiv.org/abs/2001.08361.

[202] ABTS D, KIMMELL G, LING A C, A software-defined tensor streaming multiprocessor for large-scale machine learning//SALAPURA V, ZAHRAN M, CHONG F, ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022. ACM, 2022: 567-580. https://doi.org/10.1145/3470496.3527405. DOI: 10.1145/3470496.3527405.

[203] TALPES E, SARMA D D, WILLIAMS D, The Microarchitecture of DOJO, Tesla's Exa-Scale Computer. IEEE Micro, 2023, 43(3): 31-39. https://doi.org/10.1109/MM.2023.3258906. DOI: 10.1109/MM.2023.3258906.

附录 A　攻读博士期间发表学术论文

攻读博士期间，本文作者发表期刊和会议论文共 18 篇，参与一次 MICRO Tutorial。以第一作者（包含共同一作）身份发表 7 篇论文，其中 6 篇 CCF-A 类论文。

会议论文

1. **Size Zheng**, Renze Chen, Meng Li, Zihao Ye, Luis Ceze, Yun Liang. "**vMCU: Coordinated Memory Management and Kernel Optimization for DNN Inference on MCUs**." Seventh Conference on Machine Learning and Systems. (**MLSys**). 2024.
2. Cong Li, Zhe Zhou, **Size Zheng**, Jiaxi Zhang, Yun Liang, Guangyu Sun. "**SpecPIM: Accelerating Speculative Inference on PIM-Enabled System via Architecture-Dataflow Co-Exploration**". International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**). 2024.
3. Renze Chen, Zijian Ding, **Size Zheng**, Chengrui Zhang, Jingwen Leng, Xuanzhe Liu, Yun Liang. "**MAGIS: Memory Optimization via Coordinated Graph Transformation and Scheduling for DNN**." International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**). 2024.
4. Hanyu Zhang, Liqiang Lu, Siwei Tan, **Size Zheng**, Jia Yu and Jianwei Yin. "**SpREM: Exploiting Hamming Sparsity for Fast Quantum Readout Error Mitigation**." Design Automation Conference (**DAC**). 2024.
5. Renze Chen, Zijian Ding, **Size Zheng**, Meng Li, Yun Liang. "**MoteNN: Memory Optimization via Fine-grained Scheduling for Deep Neural Networks on Tiny Devices**." Design Automation Conference (**DAC**). 2024.
6. Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, **Size Zheng**, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, Baris Kasikci. "**ATOM: LOW-BIT QUANTIZATION FOR EFFICIENT AND ACCURATE LLM SERVING**." Seventh Conference on Machine Learning and Systems. (**MLSys**). 2024.
7. **Size Zheng**, Siyuan Chen, Siyuan Gao, Liancheng Jia, Guangyu Sun, Runsheng Wang, Yun Liang. "**TileFlow: A Framework for Modeling Fusion Dataflow via Tree-based Analysis**." To appear in Annual IEEE/ACM International Symposium on Microarchitecture (**MICRO**). 2023.
8. Xiuping Cui, **Size Zheng**, Tianyu Jia, Le Ye and Yun Liang "**ARES: A Mapping Framework of DNNs towards Diverse PIMs with General Abstractions**." To appear in IEEE/ACM International Conference on Computer-Aided Design (**ICCAD**). 2023.
9. **Size Zheng**, Siyuan Chen, Yun Liang. "**Memory and Computation Coordinated Mapping of DNNs onto Complex Heterogeneous SoC**." Design Automation Conference (**DAC**). 2023.
10. Zizhang Luo, Liqiang Lu, **Size Zheng**, Jieming Yin, Jason Cong, Jianwei Yin, Yun Liang. "**Rubick: A Synthesis Framework for Spatial Architectures via Dataflow Decomposition**." Design Au-

tomation Conference (**DAC**). 2023.

11. **Size Zheng**, Siyuan Chen, Peidi Song, Renze Chen, Xiuhong Li, Shengen Yan, Dahua Lin, Jingwen Leng, Yun Liang. "**Chimera: An Analytical Optimizing Framework for Effective Compute-intensive Operators Fusion**." IEEE International Symposium on High-Performance Computer Architecture (**HPCA**). 2023.
12. **Size Zheng**, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, Yun Liang. "**AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction**." IEEE/ACM International Symposium on Computer Architecture (**ISCA**). 2022.
13. Zizhang Luo, Liqiang Lu, **Size Zheng**, Liancheng Jia, Yun Liang. "**AHS: An Agile Framework for Hardware Specialization and Software Mapping**." Annual IEEE/ACM International Symposium on Microarchitecture. **(MICRO Tutorial)**, 2021.
14. Qingcheng Xiao, **Size Zheng**, Bingzhe Wu, Pengcheng Xu, Xuehai Qian, Yun Liang. "**HASCO: Towards Agile HArdware and Software CO-design for Tensor Computation**." ACM/IEEE Annual International Symposium on Computer Architecture (**ISCA**), 2021.
15. **Size Zheng**, Yun Liang, Shuo Wang, Renze Chen, Kaiwen Sheng. "**FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System**." International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**), 2020.
16. Yi-Hsiang Lai, Hongbo Rong, **Size Zheng**, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher J. Hughes, Pradeep Dubey. "**SuSy: A Programming Model for Productive Construction of High-Performance Systolic Arrays on FPGAs**." IEEE/ACM International Conference On Computer Aided Design (**ICCAD**), 2020.

期刊和杂志

1. Liqiang Lu, Zizhang Luo, **Size Zheng**, Jieming Yin, Jason Cong, Yun Liang, Jianwei Yin. "**Rubick: A Unified Infrastructure for Analyzing, Exploring, and Implementing Spatial Architectures via Dataflow Decomposition**." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. (**TCAD**). 2023.
2. **Size Zheng**, Renze Chen, Yicheng Jin, Anjiang Wei, Bingyang Wu, Xiuhong Li, Shengen Yan, Yun Liang. "**NeoFlow: A Flexible Framework for Enabling Efficient Compilation for High Performance DNN Training**." IEEE Transactions on Parallel and Distributed Systems (**TPDS**), 2022.
3. Liqiang Lu, **Size Zheng**, Qingcheng Xiao, Deming Chen, Yun Liang. "**Accelerating convolutional neural networks on FPGAs**" (in Chinese). **SCIENTIA SINICA Informationis**, 2019.

附录 B　攻读博士学位期间专利申请情况

博士期间申请两项专利，一项已授权，另一项已通过初审。

- 专利名称：张量计算代码优化方法、装置、设备及介质。国家专利。申请人：浙江省北大信息技术高等研究院; 杭州未名信科科技有限公司。专利号：2019110258915。授权公开日：2023 年 6 月 20 日。发明人：梁云，**郑思泽**。
- 专利名称：一种面向微控制器的循环内存管理方法。国家专利。申请人：北京大学。专利/申请号：202410023984.9。发明人：梁云，**郑思泽**。

附录 C　个人履历、所获奖项以及参与的科研项目

C.1　个人履历

- 1997 年 12 月 2 日出生于黑龙江省鹤岗市宝泉岭
- 2012 年 7 月毕业于黑龙江省鹤岗市宝泉岭局直中学
- 2015 年 7 月毕业于黑龙江省鹤岗市宝泉岭高级中学，通过高考进入北京大学信息科学技术学院
- 2018 年任北京大学信息科学技术学院团委学生科学技术协会主席
- 2019 年 7 月毕业于北京大学信息科学技术学院智能科学系，获得理学学士学位
- 2019 年 9 月保研进入北京大学信息科学技术学院计算机系（现计算机学院）攻读博士学位至今
- 2023 年 9 月到 2024 年 1 月在美国华盛顿大学计算机系访问交流。

C.2　所获奖项

- 2016：北京大学信息科学技术学院”天创“奖学金
- 2017：北京大学 POSCO 奖学金，北京大学三好学生
- 2018：北京大学”金龙鱼“奖学金，北京大学三好学生
- 2019：北京大学优秀本科毕业生
- 2020：商汤科技优秀实习生
- 2021：北京大学三好学生
- 2023：国家奖学金，奋进奖学金，北京大学优秀科研奖，北京大学华为奖学金，北京大学 DACS 产业贡献奖
- 2024：北京市普通高等学校优秀毕业生，北京大学优秀毕业生

C.3　参与科研项目

- 面向智能芯片的高性能编译技术研究: 国家自然科学基金项目. (NO.U21B2017) 20220101-20240701. 针对 AI 芯片设计并实现编译器框架，涉及计算图、算子、模型量化、多请求等技术内容
- 面向嵌入式智能的软硬件协同优化研究: 省 (自治区、直辖市) 项目. (NO.JQ19014) 20191201-20231201. 设计实现嵌入式芯片的代码生成框架; 设计软硬件协同优化框架中的软件优化算法部分
- 面向智能芯片的编译器设计: 企、事业单位委托项目. (北大-百度基金) 20210101- 20221201. 针对多款 AI 芯片设计代码优化和自动生成技术，并取得超过手工优化结果的水平

附录 D　参与开源项目总结

D.1　开源项目

- FlexTensor (https://github.com/pku-liang/FlexTensor) Github 160+ 星，实现自动算子调优和代码生成。主要贡献者。
- AMOS (https://github.com/pku-liang/AMOS) Github 90 星, 实现面向 AI 芯片自动指令生成。主要贡献者。
- TileFlow (https://github.com/pku-liang/TileFlow) Github 40+ 星，实现针对融合数据流的性能模型和自动优化模块。主要贡献者。
- 面向 Nvidia GPU 的高性能矩阵乘法实现 (https://github.com/KnowingNothing/MatmulTutorial) Github 190+ 星。主要贡献者。
- AI 编译器论文列表总结 (https://github.com/KnowingNothing/compiler-and-arch) Github 310+ 星。主要贡献者。
- 参与 TVM (https://github.com/apache/tvm, Github 1 万 1 千星) 项目开发, 贡献 PTX Tensor Core 代码生成功能。
- 参与 MLC-LLM (https://github.com/mlc-ai/mlc-llm, Github 1 万 6 千星) 项目开发，贡献预测解码（Speculative Decoding）功能。

致谢

博士五年时光匆匆过去，不知不觉间已经在研究路上走了很远，虽然未来是否会继续从事研究工作尚未可知，但是仍然感谢北京大学提供的宝贵机会，让我拥有五年的博士研究生生活。科研道路上一直以来是导师梁云老师在指导我，仍然记得博士一年级时梁老师带着我在暑假连续修改 ASPLOS 论文的时光，那时的梁老师和我两个星期不间断的早七点到实验室，晚九点离开，保持高强度的讨论和修改，经常在中午去找梁老师时候发现梁老师在办公室的沙发上休息，尽管那是一段辛苦的时光，但是换来了北大首篇一作 ASPLOS 的成果，现在回味，仍是十分怀念。只有经历了多年科研生活，才能明白当时梁老师的耐心和付出是多么宝贵。

感谢孙广宇老师在 9 年前的招生中向我推荐信息科学技术学院，将我引向计算机科学的道路，在博士期间仍然对我提供指导和帮助。同时也感谢罗国杰老师，林亦波老师，李萌老师的帮助和指导。

感谢实验室的师兄弟的帮助，感谢李秀红师兄对我科研的指导，秀红师兄工作后仍然帮助我顺利完成实习项目和科研项目；感谢王硕师兄为我选择未来方向和职业提供的建议；感谢卢丽强师兄帮我和华为建立联系，为我找到方向匹配的部门，并在我求职过程中不断提供建议和支持；感谢我的师弟陈仁泽、陈思元，魏安江与我共同完成了许多科研项目，取得了许多科研成果；感谢师兄师弟们的陪伴和支持。

感谢父母对我二十多年的培养和照顾，在我经历大大小小的挫折和困难时仍然鼓励我，为我提供一个温暖的家。感谢我的老婆郭潇陪我度过博士五年时光，和我分享科研成功的喜悦和挫败的苦闷。

北京大学学位论文原创性声明和使用授权说明

原创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不含任何其他个人或集体已经发表或撰写过的作品或成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本声明的法律结果由本人承担。

论文作者签名：郑思泽　日期：2024年 4月16日

学位论文使用授权说明

（必须装订在提交学校图书馆的印刷本）

本人完全了解北京大学关于收集、保存、使用学位论文的规定，即：

- 按照学校要求提交学位论文的印刷本和电子版本；
- 学校有权保存学位论文的印刷本和电子版，并提供目录检索与阅览服务，在校园网上提供服务；
- 学校可以采用影印、缩印、数字化或其它复制手段保存论文；
- 因某种特殊原因需要延迟发布学位论文电子版，授权学校☑一年/☐两年/☐三年以后，在校园网上全文发布。

（保密论文在解密后遵守此规定）

论文作者签名：郑思泽　导师签名：

日期：2024年 4月16日