计算机研究与发展
Journal of Computer Research and Development
DOI: 10.7544/issn1000-1239.202330059
60(6): 1216−1231, 2023

Puzzle: 面向深度学习集成芯片的可扩展框架

王梦迪[1,2,3,4] 王 颖[1,3] 刘 成[1,3] 常开颜[1,2,3] 高成思[1,2,3,4] 韩银和[1,3] 李华伟[1,3] 张 磊[1,3,4]

[1](中国科学院计算技术研究所 北京 100190)

[2](中国科学院大学 北京 100190)

[3](处理器芯片全国重点实验室(中国科学院计算技术研究所) 北京 100190)

[4](移动计算与新型终端北京市重点实验室(中国科学院计算技术研究所) 北京 100190)

(wangmengdi17s@ict.ac.cn)

Puzzle: A Scalable Framework for Deep Learning Integrated Chips

Wang Mengdi[1,2,3,4], Wang Ying[1,3], Liu Cheng[1,3], Chang Kaiyan[1,2,3], Gao Chengsi[1,2,3,4], Han Yinhe[1,3], Li Huawei[1,3], and Zhang Lei[1,3,4]

[1](*Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190)

[2](*University of Chinese Academy of Sciences*, *Beijing* 100190)

[3](*State Key Lab of Processors* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

[4](*Beijing Key Laboratory of Mobile Computing and Pervasive Device* (*Institute of Computing Technology*, *Chinese Academy of Sciences*), *Beijing* 100190)

Abstract Chiplet integration is becoming a highly scalable solution of customizing deep learning chips for different scenarios, thus many chip designers start to reduce the chip development cost by integrating "known-good" third-party dies, which shows advantages in higher yield, design flexibility, and shorter time-to-market. In conventional chip business model, the dedicated software toolchain such as compiler is provided as part of the chip solution and plays an important role in chip performance and development. However, when it comes to chip solution that assembles multiple third-party dies, the toolchain must face the situation that is unknown to the dedicated compiler of die vendors in advance. In such a situation, how to dispatch tasks to hardware resources and manage the cooperation between the provided interfaces of independent third-party dies becomes a necessity. Moreover, designing a whole-new toolchain for each integrated chip is time-consuming and even deviating the original intention of agile chip customization. In this paper, we propose Puzzle, a scalable compilation and resource management framework for integrated deep learning chips. Puzzle contains a complete framework from profiling the input workload to run-time management of chip resources, and reduces redundant memory access and expensive inter-die communication through efficient and self-adaptive resource allocation and task distribution. Experimental results show that Puzzle achieves an average of 27.5% latency reduction under various chip configurations and workloads compared with state-of-the-art solutions.

Key words agile chip customization; chiplet; deep learning chip; neural processing unit; task dispatching

摘 要 芯粒集成逐渐成为不同场景下敏捷定制深度学习芯片的高可扩展性的解决方案,芯片设计者可以通过集成设计、验证完成的第三方芯粒来降低芯片开发周期和成本,提高芯片设计的灵活性和芯片良率.在传统的芯片设计和商业模式中,编译器等专用软件工具链是芯片解决方案的组成部分,并在芯片性

收稿日期: 2023-01-10; **修回日期**: 2023-04-10

基金项目: 国家自然科学基金项目(62090024, 62222411)

This work was supported by the National Natural Science Foundation of China (62090024, 62222411).

通信作者: 王颖(wangying2009@ict.ac.cn)

能和开发中发挥重要作用. 然而，当使用第三方芯粒进行芯片敏捷定制时，第三方芯粒所提供的专用工具链无法预知整个芯片的资源，因此无法解决敏捷定制的深度学习芯片的任务部署问题，而为敏捷定制的芯片设计全新的工具链需要大量的时间成本，失去了芯片敏捷定制的优势. 因此，提出一种面向深度学习集成芯片的可扩展框架（scalable framework for integrated deep learning chips）——Puzzle，它包含从处理任务输入到运行时管理芯片资源的完整流程，并自适应地生成高效的任务调度和资源分配方案，降低冗余访存和芯粒间通信开销. 实验结果表明，该可扩展框架为深度学习集成芯片生成的任务部署方案可自适应于不同的工作负载和硬件资源配置，与现有方法相比平均降低27.5%的工作负载运行延迟.

关键词 芯片敏捷定制；芯粒；深度学习芯片；神经网络处理器；任务调度

中图法分类号 TP389.1

近年来，深度学习技术已广泛应用于许多领域，如图像分类[1-2]、目标检测[3-4]和自然语言处理[5-7]等. 深度学习芯片作为深度学习算法的专用加速硬件，已被广泛应用在从数据中心规模的计算系统[8]到轻量级的边缘或物联网设备[9]中. 华为[10]、寒武纪[11]、英伟达[12]、谷歌[13]、特斯拉[14]等公司发布了数百款深度学习芯片，适用于不同的应用场景，但受限于各自使用场景的硬件条件限制（如功耗、面积），因此这些深度学习芯片使用不同的神经网络处理器（neural processing unit, NPU）架构，具有不同规模的算力. 例如，Cerebras CS-2[15]深度学习芯片拥有约850 000个神经网络处理器核，其峰值功率达到23 kW. 相比之下，谷歌提出的张量处理单元（tensor processing unit, TPU）v3[13]只有8个神经网络处理器核，其峰值功率约为200 W. 这2种应用于不同场景的深度学习芯片约有20倍的计算能力差异和约100倍的功耗差异，这种差异表明，深度学习芯片需要针对特定的应用场景进行定制. 然而，开发深度学习芯片非常耗时且开发成本高昂，从设计到上市通常需要数年时间[16].

芯粒集成技术正逐渐成为一种强可扩展性的芯片敏捷定制方案，芯片设计人员可以直接在硅或有机中介层上集成同构或异构的芯粒[17]，甚至可以集成不同制造工艺下的芯粒[18]. 芯片设计人员可以直接根据应用场景需求将多个设计、验证完成的处理器芯粒、存储芯粒、输入输出（input/output, I/O）芯粒等组合在一起，实现芯片敏捷定制，这种基于芯粒集成的芯片设计被称为“集成芯片”. 集成芯片设计模式可以灵活地根据需求定制芯片，且成本效益高、开发周期短，但是也存在一些亟待解决的问题. 例如，如何管理这些集成到一起的芯粒资源和如何将工作负载分配到具有不同属性的芯粒. 对于深度学习集成芯片，任务和资源分配对芯片性能尤为重要，因为它们决定了芯粒之间的计算和通信延迟，特别是基板（interposer）上的通信具有更高的延迟和能耗[19]. 第三方芯粒所提供的工具链或编译器无法预测集成芯片的资源，因此无法处理整个集成芯片中其他芯粒的属性和信息，而为每个集成芯片设计完整的工具链需要大量的时间成本和对第三方芯粒的架构知识，失去了芯片敏捷开发的优势.

针对这些问题，本文研究一种面向深度学习集成芯片的编译和资源管理框架. 该框架为敏捷定制的多样化的深度学习集成芯片提供了从输入工作负载到集成的第三方芯粒接口的自动化工作流程，生成高效的任务调度和资源分配方案，并提供运行时资源管理，处理芯粒之间的协作和任务之间的依赖关系.

本文的主要贡献包括3个方面：

1）提出了一种面向深度学习集成芯片的可扩展框架（scalable framework for integrated deep learning chips）——Puzzle，该框架为深度学习集成芯片提供了自适应的全栈解决方案，向上对接输入工作负载，向下对接芯片资源运行时管理，降低了芯粒集成方式下的深度学习芯片敏捷定制的开发成本.

2）提出了一种两级资源图的表示，用于描述集成芯片中的芯粒资源及其关系. 异构的神经网络处理器和芯粒由具有不同属性的结点表示，本文定义了一组具有不同属性的边来描述它们之间的关系. 该两级资源图表示用于后续的资源分配和任务调度，使Puzzle框架可以自适应于深度学习芯片敏捷定制的多样性.

3）提出一种自适应的任务调度和资源分配策略. 通过启发式搜索生成资源分配方案，并通过“内存借用”方式降低芯粒间开销和内存访问开销.

1 研究背景和相关工作

1.1 基于芯粒集成方式的芯片设计

基于芯粒集成的芯片设计模式将单片大芯片拆分成多个小芯粒，再使用硅或有机基板对小芯粒进行组装. 许多工业公司和研究单位提出了使用芯粒集

成的芯片设计，包括中央处理器（central processing unit, CPU）[20-22]、图形处理器（graphics processing unit, GPU）[23-24]和加速器[25-26].

一方面，集成芯片设计模式提高了芯片的可扩展性，避免了随着芯片规模扩大出现的良率快速降低问题，且解决了异构集成的问题，降低了芯片设计成本 Pal 等人[27]使用芯粒集成技术构建了发布时最大的芯片，并保证芯片的高良率；英特尔公司的下一代 GPU Ponte Vecchi[23]芯片中集成了分别由 5 种制程生产的 47 个不同功能的芯粒，通过异构集成降低了芯片的生产成本.另一方面，将大芯片的不同功能模块拆分为芯粒，提高了这些功能模块的可重用性. 例如，AMD 公司的 Rome 和 Matisse 芯片[28]分别面向服务器和端侧市场，它们使用完全相同但是数量不同的计算和 I/O 芯粒，并用不同的方式进行芯粒间互连，在 2 种使用场景下都获得了很好的性能. 在不同的芯片设计中，重复使用设计、验证完成的芯粒模块可以降低芯片设计成本，成为了芯片敏捷定制的重要方式之一. 因此，许多公司和组织正在积极推进芯粒接口的标准化，例如英特尔、AMD、ARM 等公司提出的通用小芯片互连通道（universal chiplet interconnect express, UCIe）[29]协议，以及 Facebook 发起的开放计算项目（open compute project, OCP）提出的 ODSA（open domain-specific architecture）协议[30]等.

随着芯粒间互连从协议层到物理层逐渐统一，根据应用场景的算力需求和硬件资源约束来选择和集成第三方芯粒正在成为一种趋势. 然而，这种快速定制和组装的芯片仍然需要为其开发系统软件/工具链来处理输入的工作负载和硬件资源之间的关系. 虽然芯粒供应方通常会提供专用的工具链（如编译器），但是这种专用工具链只能用于该芯粒自身，无法预知整个集成芯片中的其他资源. 在各种不同的集成场景下，如何向各个芯粒分配任务并管理它们之间的合作仍然是亟待解决的问题.

1.2 多核多任务深度学习芯片任务部署

在现有的多核深度学习芯片中，已经有一些使用芯粒集成技术的芯片设计生产出高可扩展性的大算力芯片. Centaur[25]是一个混合使用稀疏-稠密模式的面向推荐系统的深度学习加速器芯片，它使用异构集成来优化推荐模型中 Embedding 层和多层感知机（multi-layer perceptron, MLP）层的计算模式差异. Tan.等人[31]提出了集成芯片软硬件设计协同搜索框架 NN-Baton，即在指定的硬件约束和工作负载模型下，自动地生成芯片粒度设计方案和任务负载映射策略. 然而，它们的资源管理策略都只能在其配套的硬件设计和任务负载上使用，不能处理集成芯片敏捷定制的多样化场景，也无法处理多任务和可变工作负载. Shao 等人[26]提出了 Simba 深度学习推理芯片，它由 36 个神经网络加速器芯粒组成，并使用了一种跨层的流水线方法，且在子任务分配时考虑通信延迟导致的不均匀性. 然而，该芯片的任务调度器只能处理单任务工作负载，且其中的跨层流水线方法在处理少量的批数据时会导致芯片低利用率问题，这限制了其调度器的使用场景，无法满足芯片敏捷定制的需求.

另外，近年来也有一些相关工作[32-34]提出了片上多核深度学习处理器的任务调度和资源分配方案. 特别地，还有一些研究方案[35-40]可以解决多任务或多租户场景下的任务调度和资源分配问题. 然而，这些相关工作中提出的方法只能在硬件资源满足某些特定条件时生成高效的任务调度和资源分配方案，例如具有足够大的片上缓存区、特定的神经网络加速器架构或多核使用同构设计. 另外，这些相关工作通常使用简单的通信模型，无法处理深度学习集成芯片中的复杂通信场景，因此无法为敏捷定制的深度学习芯片提供任务调度和资源分配的解决方案. 集成芯片设计方法的优势在于可以利用芯粒级重用，通过集成已经设计、验证完成的芯粒实现芯片敏捷定制，因此，任务调度和资源分配策略不应该成为硬件设计的“约束”，例如限制芯粒中加速器的架构、连接方式或存储结构设计，而是应该自动适应于硬件配置. 但现有的框架在灵活性上不足以满足集成芯片的敏捷定制需求.

在这些面向多任务场景下任务调度和资源分配的相关工作[35-40]中，将多任务工作负载映射到多核深度学习芯片的策略可以分为 2 类：时分复用[35,37,40]和空分复用[36,38-39].

在时分复用策略下，每次任务分配时只选择一个计算任务，且使用全部计算资源完成该计算，多个任务分时地占用计算资源. 时分复用策略非常适合于同构多核深度学习芯片[26,34]和分型多核深度学习芯片[41]. 以卷积层为例，可以通过卷积计算中的多个维度来进行计算划分，例如批（batch）维度、特征值张量 $\boldsymbol{F}$ 中的维度和权重张量 $\boldsymbol{W}$ 中的维度等. 时分复用策略为每个任务提供了大量的计算资源，并可以以较低延迟完成计算任务，且资源管理较为简便. 由于在时分复用策略下，同一时刻只有一个计算任务占据全部硬件资源，即资源分配方式固定. 因此，时分

复用的相关工作的优化空间集中在任务调度中.PREMA[35]是代表性的多任务深度学习处理器时分复用调度方案，它允许高优先级的任务优先使用计算资源，从待执行任务队列中根据任务优先级、任务已等待时间等属性选择当前需要调度到硬件上开始执行的任务，并设计了多种任务切换策略. AI-MT[37]将计算任务切分为更细粒度的子任务作为调度单位，使计算单元尽早开始计算. Layerweaver[40]轮流地调度计算密集型和访存密集型的神经网络层，利用其差异性降低计算单元空转等待的时间.

然而，随着多核芯片规模的增大，时分复用策略也展现了一些缺点：1)将一个层拆分到过多的计算资源会由于张量尺寸过小导致NPU的资源利用率降低；2)NPU之间的并行化通常伴随着特征值/权重的复制或部分和(partial sum)的规约，因此较多的NPU协作完成同一个计算任务会导致NPU之间产生较高的数据通信开销和同步开销，在深度学习集成芯片中，不同芯粒的NPU的通信还需要经过高开销的芯粒间互连网络(network on interposer, NoI)，因此加剧了这一问题；3)相同任务在不同架构的NPU上执行的性能差异可能超过50%[36]，因此，在时分复用策略下每个NPU执行完全相同的任务时，无法充分利用NPU异构优势.

空分复用策略同时部署多个独立的任务，并为每个任务分配一部分资源. 即同一时刻，多个任务同时在硬件上执行. Liu等人[38]发现以任务为单位分配资源会导致部分对资源需求量大的层无法获得充足的计算资源；而以层为单位分配资源会过于激进地请求资源，影响并发执行的其他任务，因此提出VELTAIR使用块粒度任务进行资源分配. Planaria[39]是一种高自由度的空分复用深度学习芯片，它利用特殊设计的全向可分解的脉动阵列结构实现计算资源可“分”可“合”，动态地将资源分配给不同的计算任务. 然而，这些相关工作[38-39]只面向同构硬件资源. Kwon等人[36]提出使用一组异构数据流加速器来适应不同神经网络工作负载，并提出面向异构计算的空分复用策略Herald，使用贪心策略为每个任务选择其计算利用率最高的数据流对应的加速器执行.

空分复用策略提高了资源利用率，但它依赖于硬件设计与工作负载的高度匹配. 当出现资源和需求不匹配问题时会降低处理器性能. 具体而言，当NPU中的片上缓存无法容纳特征值和权重时，至少会重复请求其中之一，导致高通信开销和访存开销. 另外，空分复用策略中，每个任务固定地分配部分计算资源，当同时执行的任务规模不同时，后结束的任务成为延迟瓶颈，而先结束的任务释放的资源未被充分利用，即存在大量资源空转时间问题.

综合以上分析，目前现有的相关工作主要存在2方面局限性：1)任务调度和资源分配策略受限于特定的硬件处理器设计或特定工作负载，无法满足集成芯片设计模式下重用现有芯粒灵活组合定制芯片的需求. 2)主流的基于时分复用的策略和基于空分复用的策略分别有其优势与不足，适用范围较为有限，均无法完全扩展到集成芯片的多样化调度场景. 因此，本文提出面向深度学习集成芯片的可扩展框架Puzzle，为深度学习集成芯片敏捷定制提供包括任务调度、资源分配、资源管理和同步在内的全流程管理，其中的资源分配和任务调度方案综合了时分复用和空分复用的优势，自适应于不同的集成芯片配置和可变的工作负载.

2 深度学习集成芯片可扩展框架

2.1 Puzzle工作流程

本节主要介绍面向深度学习集成芯片的可扩展框架的工作流程，该流程描述框架中的每个功能模块，并展示如何使用这些功能模块实现编译和资源管理过程. 将工作负载部署到指定的多芯粒深度学习芯片需要4个步骤：1)收集工作负载信息和该集成芯片的硬件资源信息，并分析其属性；2)生成任务调度和资源分配方案，在Puzzle框架中，我们将神经网络层作为资源分配的最小粒度，然后将该层划分为多个子任务，并将每个子任务映射到指定的NPU；3)对于每个子任务，需要生成在每个NPU核心上执行的硬件指令；4)属于同一神经网络层的子任务间需要同步，而属于不同层的子任务可能包含数据依赖，因此，需要在运行时管理子任务和NPU的状态. 此外，工作负载、硬件资源和分配的子任务都需要使用统一的中间表示来进行抽象，以适应各种不同的芯粒集成的可能性.

在面向深度学习集成芯片的可扩展框架Puzzle中，共有5个关键功能模块，它们分别是工作负载解析模块、硬件资源信息收集模块、任务分发模块、代码生成模块和运行时管理模块. 图1展示了5个功能模块之间的关系和它们之间需要传递的信息. 工作负载解析模块接收用户指定的工作负载，并分析其中每个任务的属性和层间依赖，然后将计算图传递给任务分发模块. 硬件资源信息收集模块负责收集

芯片中所集成的芯粒的信息、NPU 核信息以及处理器核间和芯粒间的通信信息，然后使用两级的资源图描述硬件资源，并传递给任务分发模块. 任务分发模块分析计算图和资源图，并将计算任务映射到硬件资源上，生成子任务列表. 每个子任务将被发送到代码生成模块，并通过调用第三方芯粒所提供的工具链接口进行硬件代码生成. 子任务之间的相关关系将被传递给运行时管理模块，由芯片中的主控 CPU 芯粒完成子任务之间的同步关系和依赖关系管理.

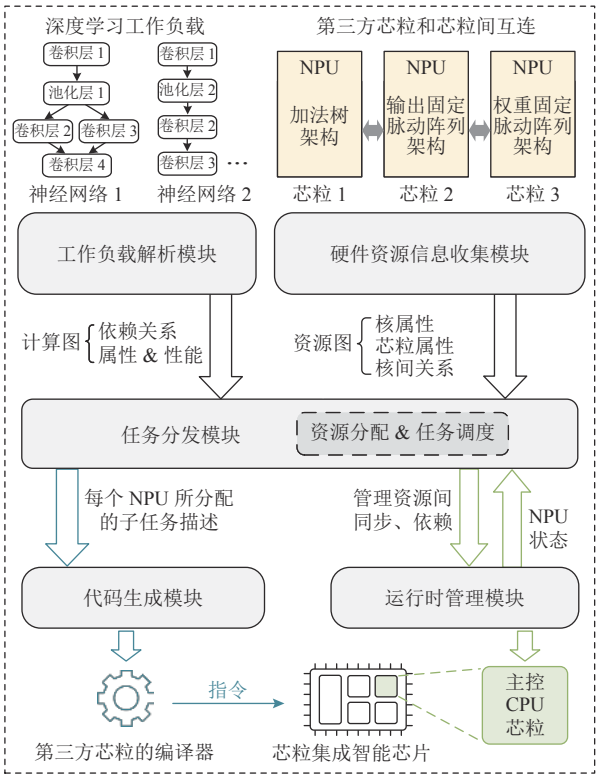


Fig. 1 Puzzle framework

图 1 Puzzle 框架

2.2 工作负载解析模块

工作负载解析模块将输入的工作负载分解为多个神经网络层，神经网络层是后续进行资源分配的最小单位. 工作负载解析模块将对每个神经网络层的层间相关性、层属性和层性能进行分析.

输入工作负载可能包含多个任务，例如在云计算场景下，多个用户可能独立地向服务器发起请求. 同一计算任务中的部分层之间存在依赖关系，而不同计算任务中的层是独立的. 层间相关性通过分析建立子任务之间的依赖关系图，确定子任务之间的调度顺序. 对于层属性分析，我们选择了对 NPU 上执行该层的计算模式有较大影响的属性组 $C=\{op, \boldsymbol{F}_{\text{in}}, \boldsymbol{F}_{\text{out}}, \boldsymbol{W}, R_{\text{density}}\}$. 其中，层操作 op 记录层的算子类型，并记录该层是否存在一些特殊计算模式(例如深度可分离卷积)；$\boldsymbol{F}_{\text{in}}$, $\boldsymbol{F}_{\text{out}}$ 和 $\boldsymbol{W}$ 分别表示输入特征值、输出特征值、权重所占的存储空间大小；R_{density} 表示计算密度与访存的比例. 在任务分发器模块中，属性组 C 将被用于确定如何为该层分配硬件资源.

此外，工作负载解析模块也需要评估层在不同 NPU 上的性能. 不同的 NPU 设计导致其在执行不同属性的神经网络层时体现出性能差异，虽然 NPU 的理想性能由其计算能力决定，但其受片上缓存区大小、数据流设计等限制，实际可达到的性能随着其执行不同的神经网络层而发生变化. 集成芯片设计模式下，芯片设计者可以选择集成异构属性的芯粒，从而在芯片执行不同计算任务时提升其性能. 因此，在任务调度和资源分配时，需要为每个计算子任务分配“最适合”的计算资源，即选择实际利用率最高的 NPU. 在重用第三方芯粒的敏捷定制设计模式下，芯片设计者并不必须了解第三方芯粒中 NPU 的具体架构细节，因此难以通过分析直接获取任务在 NPU 上的利用率. 因此，工作负载解析模块需要通过性能测试得到神经网络层部署在不同 NPU 时的延迟 l，并利用式(1)计算出利用率 UR.

$$UR=\frac{CC_{\text{practical}}}{CC_{\text{peak}}}=\frac{ops}{l\times CC_{\text{peak}}}. \tag{1}$$

其中，$CC_{\text{practical}}$为 NPU 实际达到的算力，通过神经网络层中的计算量 ops(即操作数，operations)和延迟 l 相除得到；CC_{peak}为 NPU 的峰值算力，单位为 ops/s. UR 表征了任务在该 NPU 上的计算效率，在任务分发器模块中，UR 将被用于为子任务选择计算效率高的 NPU 核.

最终，工作负载解析模块生成计算图 G_{compute}，计算图中的每个结点表示一个神经网络层和存储层的特征、性能信息；计算图中的边用于表示层间依赖.

2.3 硬件资源信息收集模块

在深度学习集成芯片设计模式下，芯片设计体现出高度灵活的特点，芯片可以由多个异构芯粒组成. 因此，硬件资源信息收集模块需要提取芯粒集成的信息，并使用统一的表示方法，便于任务分发模块进行后续的资源分配和管理.

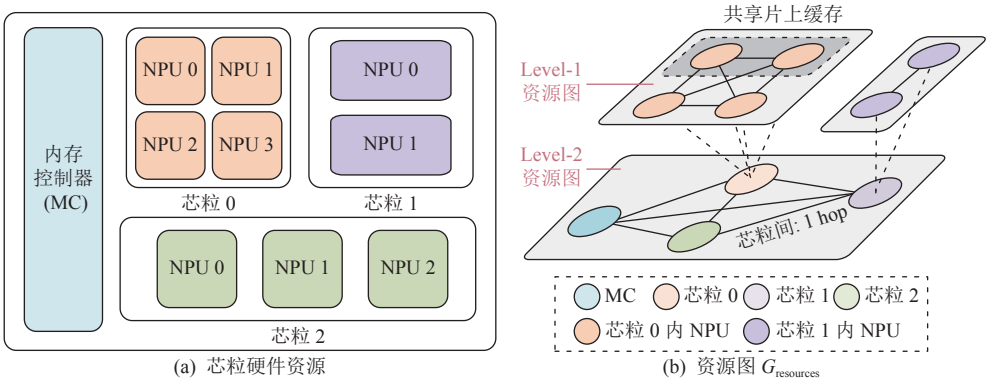
硬件资源信息收集分为处理器核属性、芯粒属性和核间关系 3 部分. 神经网络处理器核通常由运算单元和几十千字节(kilobyte, KB)至几兆字节(Mbyte, MB)的片上缓存构成. 运算单元通常直接从片上缓存中进行数据存取，因为片上缓存有较稳定的延迟，而片上缓存中的数据通常使用直接内存访问(direct memory access, DMA)模式从内存中获取，通过芯粒

间和片上的数据通信通路传输. 在收集处理器核属性时, 除了收集其支持的操作类型、算力、片上缓存大小信息, 还需收集处理器核需要的输入数据排布方式(data layout), 因为处理器核间传输数据时, 核间数据排布方式的不同会产生额外的数据转换开销.

对于每个第三方芯粒, 需要收集其片上的 NPU 处理器核数以及 NPU 核之间的关系, 包括片上网络的通信开销、核间是否存在共享的片上缓存等. 另外, 由于芯粒间通信决定了芯粒间、芯粒与内存控制器(memory controller, MC)间的数据传输速度, 并对通信延迟产生关键影响, 因此工作负载解析模块还需要收集芯粒间链路带宽和网络拓扑. 链路带宽决定了物理层的传输延迟; 而网络拓扑决定了芯粒间数据传输数据包的跳数, 以此衡量路由转发的延迟和虚通道占用、排队延迟等.

在 Puzzle 中, 本文提出一种两级的资源图来表示集成芯片中的硬件资源. 如图 2 所示, 每一个 Level-1 资源图对应一个芯粒, Level-1 资源图中的每个结点表示一个 NPU 核, 并记录 NPU 处理器核属性; Level-2 资源图中的每一个结点表示一个芯粒, 即一个 Level-1 资源图, 其中的边用于记录多个芯粒之间的关系.

Level-1 资源图中的每一条边用于记录同一芯粒上一对 NPU 间的关系.边包含 3 方面属性: 1)NPU 间通信开销; 2)NPU 间是否共享片上缓存; 3)由芯粒设计者定义的 NPU 间的并行模式. 在后续任务调度和资源分配流程中, Puzzle 为每个任务分配资源组时, 通过属性 1)选择使组内通信开销最小化的 NPU, 且分配时使资源组中的 NPU 片上缓存总容量满足任务的存储需求. 当多个 NPU 共享一个片上缓存区时(属性 2), 在资源分配时会将这些 NPU 当成一个整体统一进行资源分配, 以便不影响利用第三方芯粒内的数据放置策略; 当任务的资源分配结束后, Puzzle 将任务并行切分(并行化)成多个子任务, 并分配到资源组中的每个 NPU.若属性 3)中记录了芯粒设计者定义的 NPU 间并行模式, 则在任务并行化时将这一组 NPU 共同作为 Puzzle 并行化的一个基本单元, Puzzle 为该基本单元分配任务后, 该单元内部的并行化遵循芯粒设计者定义的并行模式.



(a) 芯粒硬件资源 (b) 资源图 $G_{\text{resources}}$

Fig. 2 Resource graph

图 2 资源图

Level-2 资源图中的每一条边用于记录一对芯粒间的关系. 它包含 2 方面属性: 1)芯粒间通过片间网络的通信开销; 2)芯粒间使用相同/不同数据排列方式. 这 2 个属性用于共同确定总数据传输开销. 该开销等于片间网络的通信开销与数据排列方式不同时所需格式转换开销的总和(格式转换开销在系统中为一个可配置的参数). 属性 2)只在 Level-2 资源图的边属性中存在, 因为同一芯粒上的多个计算核采用相同的数据排列方式, 而不同芯粒可能来源于不同的芯片生产商, 因此需要考虑数据排列方式不同产生的额外开销.

在这种资源图表示形式下, 任意 2 个 NPU 之间的关系可以表示为两级资源图上 2 个 NPU 之间边所表示关系的总和. 与使用一级资源图表示相比, 两级资源图表示可以将存储空间由 $O(m^2n^2)$ 降低至 $O(mn^2+m^2)$, 其中 m 为芯片中的芯粒数, n 为平均每个芯粒中 NPU 的数量. 使用两级的资源图来表示深度学习集成芯片的硬件资源有 3 方面好处: 1)记录 NPU 处理器核的资源属性信息; 2)NPU 核之间的关系由边表示, 特别地, 可以处理一些由芯粒设计者定义的规则(如 Level-1 资源图中的属性 2)和 3)), 提高了对多样化集成的芯粒资源的适应性; 3)两级图降低了图的大小, 特别是对于大规模深度学习集成芯片, 降低了资源图存储和任务分发的复杂程度.

2.4 任务分发模块

任务分发模块用于生成将输入工作负载映射到硬件资源的方案,由该模块决定在某一时刻使某个神经网络层开始执行,并为该层分配硬件资源,将层分解为子任务,并将子任务映射到资源组中的每个NPU核. Puzzle提出了一种高效且自适应的任务调度和资源分配方案,详见3.2节.

任务分发模块的输出是经过分解的子任务,子任务将具体信息传递给代码生成模块,子任务之间的同步和依赖关系传递给运行时管理模块.

2.5 代码生成模块

芯片/芯粒的编译器通常作为完整解决方案的一部分. 一方面,编译器为芯片/芯粒的用户提供了一套完整易用的接口;另一方面,编译器由芯片设计者提供,其中通常包含一些细粒度的编译优化,从而最大限度地利用硬件架构设计[42]. 因此, Puzzle框架应充分利用第三方芯粒的编译器接口,换言之, Puzzle框架在将子任务分配到NPU核后,通过第三方芯粒提供的编译器接口,调用其第三方编译器进行代码生成. Puzzle为子任务定义了统一的中间表示,并将中间表示转换为每个第三方接口所需的形式.

在Puzzle中,子任务的中间表示如图3所示,该中间表示包含可唯一定义子任务的所有信息. 子任务描述记录了子任务的类型和规格,输入数据源地址在列表中包含多种数据类型,如输入特征、权重、偏置等,每种类型的数据也使用列表表示. 由于输入数据可能来自于不同的NPU或主存,因此需要提供多输入数据源选项. 输出数据目的地址定义了结果输出的地址,输出地址在与其存在依赖关系的下一个子任务分配后才能确定. 如果第三方芯粒的编译器无法接收多个数据源和目标,则Puzzle的代码生成模块会生成数据整合的通信条目,将分离的数据重排至连续的地址空间. 与其他工作中提出的低层次中间表示[43]相比,由于芯粒编译器已经提供了NPU内部的专用优化,因此Puzzle框架不再关注子任务在NPU内的执行方式. 但Puzzle提供了一组易用的接口,允许数据接收或发送到多个源/目的地,当第三方编译器无法接受多个数据源和目的地时,代码生成模块也提供了数据整合方案.

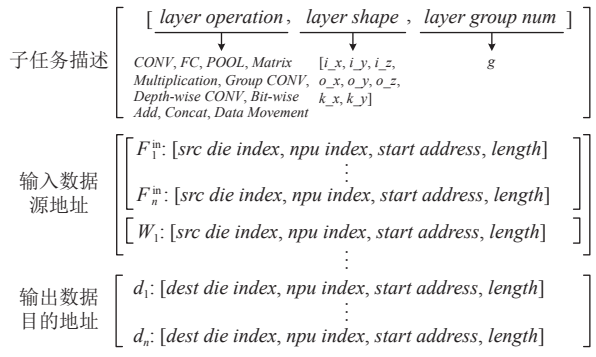


Fig. 3 Subtask description

图3 子任务描述

2.6 运行时管理模块

运行时管理模块用于控制子任务之间的同步和依赖关系. 在同一个神经网络层所切分出的子任务之间存在同步关系,在Puzzle中,具有同步关系的子任务通过同步信号被设置为BUSY或被释放到IDLE状态,因为存在同步关系的NPU间可能存在数据共享. 依赖关系发生在属于不同神经网络层的子任务之间,有2种可能性: 1)层间依赖. 发生在神经网络中的2个相邻层间. 2)片上缓存区占用. 当待执行的子任务开始请求新的输入数据时,必须确保其片上缓存中的现有数据已被使用完毕,以免发生数据覆盖,因此产生子任务间依赖关系.

在Puzzle框架中,我们对每个子任务及其数据移动操作编号,对于每个编号项,为其保存一个完整的依赖列表. 只有当依赖列表中的所有操作都完成时,新的子任务操作才会开始. 运行时管理模块通过分析依赖列表和控制每个NPU的状态来控制同步和依赖关系,并将NPU状态反馈给任务分发模块.

3 Puzzle框架任务分发方法

本节主要描述任务分发模块的详细设计,即框架中的任务分发方法. Puzzle中的任务分发模块通过避免冗余的内存访问和片间通信来生成高效的任务分发方案,并可以自适应于不同的硬件配置和工作负载.

3.1 影响深度学习集成芯片性能的因素

本节首先介绍深度学习集成芯片的2方面特点,并分析它们如何影响芯片性能.

1)芯粒集成是一种灵活的芯片设计模式,它可以集成多样化的芯粒组件,包括同构的芯粒和异构的芯粒. 因此,芯粒集成的深度学习芯片上的资源和通信是不规则的. 传统的资源分配方法通常受限于硬件资源配置,例如只能支持同构核或简单的、模式固定的异构核,以固定的方式进行多核互连. 然而,由于芯粒集成方式足够灵活,芯片设计者可以根据需求选取多种设计、验证完成的芯粒进行集成,因此无法使用传统的单一资源分配方式.

2)与传统的单片芯片相比，深度学习集成芯片具有更多的存储访问层级. 按照访存开销从低到高排序，NPU 可以从 4 种存储空间获取数据：①NPU 通过线(wire)直连方式访问该 NPU 核内的片上缓存，访问速度快，且延迟固定；②NPU 通过片上互连网络(network on chip, NoC)从同一芯粒的其它 NPU 的片上缓存区获取数据，如图 4 中右侧所示；③NPU 通过基板上的芯粒间互连网络(NoI)从其他芯粒的片上缓存获取数据，如图 4 中左侧所示；④NPU 通过芯粒间互连网络 NoI 访问内存控制器，获取内存中的数据，随着 NPU 与内存控制器之间的距离增加，这类访存行为带来最高的通信开销. 内存访问和芯粒间通信是影响芯片性能的 2 个关键因素，因此在在资源分配时需尽量降低高访问开销的内存访问行为，并尽量将 NPU 间的数据交换限制在低存储层级.

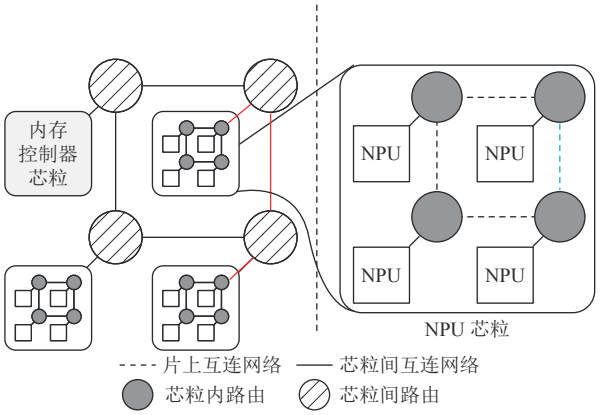


Fig. 4 Multi-level memory access architecture

图 4 多级访存结构

3.2 任务调度和资源分配方法

在第 1.2 节中分析了相关工作的局限性，主流的时分复用、空分复用调度策略受限于硬件处理器设计或工作负载，均无法广泛适用于灵活的芯粒异构集成场景. 然而，时分复用和空分复用调度方案的优势和缺陷存在互补特性，因此这 2 种调度方案可作为自适应任务分发策略的基础. 在时分复用策略下，每个任务获得充足的片上缓存资源，不需要频繁访问内存；但任务的全局化分布导致了高芯粒间通信开销. 空分复用下，芯粒间通信只发生在少量芯粒间，因此芯粒间通信开销低；然而硬件资源无法满足任务需求时会造成冗余内存访问. 另外，空分复用在不均匀的负载下会面临高资源空转时间问题，如图 5 (a)所示. 虽然现有的空分复用方法可以为长任务分配更多资源以缓解这一问题，如图 5 (a)中任务 1 占用了 NPU 0 和和 NPU 1；但在静态分配下，相对短的任务完成后释放的资源仍会被浪费，如图 5 (a)中 NPU 2 和 NPU 3. 芯粒集成的异构性加剧了这一问题，在异构资源下通过调整资源分配比例来平衡任务时间差异更加困难. 而时分复用下不存在资源空转问题，如图 5 (b)所示. 基于以上分析，时分复用和空分复用方法在内存访问、芯粒间通信、资源空转时间等方面存在互补关系，有其各自的优缺点和适用范围.

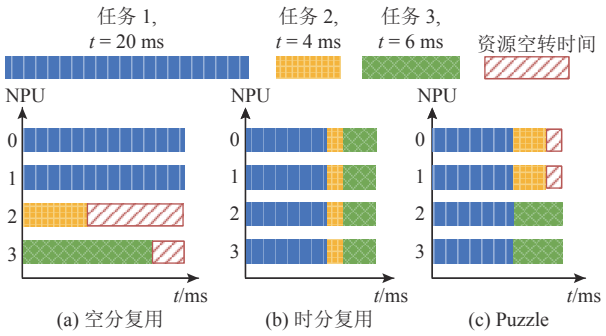


Fig. 5 Illustration of resource idle time

图 5 资源空转时间示意图

在深度学习集成芯片敏捷定制模式下，任务分发方法需要能自适应于多样化的硬件配置和工作负载，但现有的时分复用和空分复用方法都只适用于部分应用场景. 因此，需要综合二者的优势进行互补. 本文为深度学习集成芯片提出了一种基于启发式搜索的动态任务调度和资源分配方法，该方法灵活地综合了时分复用和空分复用的优势(图 5 (c)反映了 Puzzle 对 2 种现有方法的综合)，能够自适应不同的硬件资源和工作负载. 如算法 1 所示.

算法 1. Puzzle 任务调度和资源分配算法.

输入：计算图 G_{compute}，资源图 $G_{\text{resources}}$；

输出：子任务描述、子任务间同步和依赖关系.

① while !empty{NPU_{IDLE}}∧!empty{*Task Queue*} do

② NPU_{Init}←选择一个空闲 NPU；

③ t_{p}←在待执行任务队列中，使用贪心算法选择 NPU_{Init} 利用率最高的任务；

④ N_{req}=计算总片上缓存需求(t_{p})；

⑤ 创建队列 $Group_{\text{q}'}$，Enqueue($Group_{\text{q}'}$, NPU_{Init})；

⑥ N_{get}=*BufferSize*(NPU_{Init})；

⑦ *Sort*(*Dies*)；/*排序规则：其他芯粒到本芯粒的通信开销由低到高*/

⑧ while !empty{NPU_{IDLE}}∧N_{get} <N_{req} do

⑨ 创建候选队列 *Neighbors*；

⑩ for NPU_i *in* $Group_{\text{q}}$ do

⑪ 将同芯粒的 NPU 加入队列中；

⑫ end for
⑬ if empty{*Neighbors*} then
⑭ 将通信开销最低的邻居芯粒的 NPU 加入队列中(已按通信开销排序);
⑮ end if
⑯ *Sort*(*Neighbors*); /*排序规则: 候选结点距当前资源组通信开销从低到高, 通信开销相同时按计算利用率 *UR* 由高到低*/
⑰ while $N_{get} < N_{req}$ do
⑱ 从排序后的 *Neighbors* 集合中依次选择最优的 NPU_{sel};
⑲ *Enqueue*($Group_q$, NPU_{sel});
⑳ $N_{get} += BufferSize(NPU_{sel})$;
㉑ *Dequeue*(*Neighbors*, NPU_{sel});
㉒ 直到满足资源需求, 或 *Neighbors* 中候选结点被用尽, 则退出本层 while 循环;
㉓ end while
㉔ end while
㉕ if $R_{acquire} < \theta_{acquire}$ then
㉖ 循环等待空闲资源;
㉗ end if
㉘ if empty{*Task Queue*} then
㉙ 将全部剩余资源加入 $Group_q$;
㉚ end if
㉛ 并行化($Group_q$);
㉜ 生成子任务描述、子任务同步和依赖关系;
㉝ end while

Puzzle 任务分发算法的输入是资源图和计算图, 输出是子任务描述(传递到代码生成模块)及子任务间的同步和依赖关系(传递到运行时管理模块). 如算法 1 中行②所示, 当芯片上有空闲 NPU 资源且有待执行任务时, 开始执行一次任务调度和资源分配. 首先, 从硬件资源图中随机选取一个当前状态为 *IDLE* 的 NPU 结点; 然后使用贪心策略选择在 NPU_{Init} 上利用率最高的任务 t_p(行③), 利用其 NPU 的异构性; 接下来, Puzzle 使用带优先级规则的资源搜索策略, 根据 t_p 的资源需求, 以 NPU_{Init} 为起点扩展一个资源组 $Group_q$, 直到一组中的总片上缓存区资源满足层的数据尺寸. 在这样的资源分配策略下, 计算过程中的数据复制和数据交换被限制在这一组资源内, 由于 NPU 的异构性(一部分 NPU 可能片上缓存较大而计算资源少, 另一部分则相反), 每个 NPU 所需的数据可能来自于该 NPU 本身的片上缓存或附近 NPU 的片上缓存, 我们称之为“内存借用”, 即片上缓存区较大的 NPU 将空间“借”给片上缓存不足的 NPU. 因此, Puzzle 将核间通信尽量限制在较低的访存级别, 以降低通信开销, 这也同时降低了内存控制器附近的拥塞.

在为该资源组搜索资源结点时, 选择资源的优先级规则为: 1)优先选择同一芯粒上的 NPU 结点, 因为与芯粒内的片上互连通信相比, 芯粒间的链路会产生更高的通信延迟和拥塞概率. 2)优先选择与本芯粒通信开销低的邻居芯粒中的 NPU 结点. 3)优先选择对该资源组要执行的任务计算利用率高的 NPU 结点, 计算利用率信息 *UR* 来源于工作负载解析模块. 基于这 3 条规则, 在算法 1 行⑨中, 我们创建候选队列 *Neighbors*, 按照优先级规则, 依次选取与当前 $Group_q$ 中的结点同芯粒的 NPU、通信延迟最低的邻居芯粒的 NPU 加入候选队列. 在队列中, 使用候选结点到当前资源组的通信延迟和 NPU 执行该任务的性能, 对计算图中的每个 NPU 结点排序, 依次将最优的 NPU 候选结点加入资源组. 若 *Neighbors* 中的候选结点用尽后仍没有使资源组满足任务对片上缓存的需求, 则重复这一过程, 继续向外扩张资源组, 直至资源组的片上缓存资源满足任务需求.

在算法 1 行㉕~㉗中, 如果在资源分配结束后, 获取到的资源比例低于系统中设定的资源阈值, 则认为该资源不足情况会造成重复的内存访问和冗余的数据传输, 并带来较高额外开销, 那么该任务需要挂起等待更多空闲资源. 在算法 1 行㉘~㉚中, 描述了一种边界条件: 如果队列中没有待执行的任务, 但是存在剩余的空闲硬件资源, 则将剩余资源添加到资源组 $Group_q$ 中. 随后, 任务分发器将以数据并行方式将任务 t_p 并行化分配给 $Group_q$ 中的每个 NPU, 从批(batch)和特征值的 3 个维度中, 选择开销最小的维度进行任务分割, 子任务的大小比例由 NPU 的算力决定. 若资源图中表示芯粒设计者定义了部分 NPU 间的并行规则, 则 Puzzle 的并行化过程会将这部分 NPU 视为一个整体, 进行任务并行化分配, 而其内部的并行使用芯粒设计者定义的并行规则. 最终, 任务分发模块将生成的子任务发送给代码生成模块, 将子任务间的同步和依赖关系发送给运行时管理模块.

Puzzle 实现了一种动态、自适应的任务调度和资源分配策略. 与时分复用策略相比, Puzzle 的资源分配策略将通信限制在一部分芯粒的范围内, 且优先选择较低的通信层级, 降低了不必要的芯粒间通信; 与空分复用策略相比, Puzzle 的资源分配策略可

以灵活地满足任务对资源的需求，以避免冗余的内存访问开销，且空分复用在不均匀工作负载下面临高资源空转时间问题. Puzzle 的资源分配方式介于空分复用和时分复用之间，与空分复用相比均摊了负载的不均匀性，降低了资源空转时间，如图 5 所示. Puzzle 的任务分发策略综合了这 2 种原有调度方式的优势，优化了这 2 种原有调度方式的瓶颈问题，在芯粒集成的多样化硬件配置和工作负载下，均可自适应地生成高效的任务分发方案.

3.3 任务调度和资源分配算法复杂度分析

Puzzle 的任务调度和资源分配策略中使用了贪心策略和优先级规则来降低算法的运行开销，本节中将量化分析算法 1 的复杂度.

算法 1 中行①表示系统循环等待芯片上有空闲 NPU 资源且有待执行任务. 为了衡量任务分发算法的运行开销，从行②起，分析一次任务调度和资源分配的时间复杂度、空间复杂度. 假定芯片中共有 m 个芯粒，每个芯粒上包含 $n_1, n_2, \cdots, n_m$ 个 NPU 核. 从统计学意义上，假定每个芯粒上平均包含 n 个核. 另假定待执行任务队列中共有 k 个任务.

1)时间复杂度分析. 在任务选择阶段(算法 1 行②~⑥)，贪心算法需要遍历待执行的任务列表，时间复杂度为 $O(k)$，其余操作均为常数时间. 在资源分配阶段(算法 1 行⑦~㉔)需要进行多轮资源扩展，因此行⑦对芯粒通信开销进行排序预处理，只需要对 Level-2 图中表示初始芯粒与其他芯粒通信开销的边进行排序，因此时间复杂度为 $O(m\log m)$. 从行⑧起为循环的多轮选择过程，每一轮选择同芯粒或通信延迟最低的邻居芯粒上的 NPU 加入候选队列 *Neighbors*. 在硬件中，可能同时存在多个通信延迟最低且相等的邻居芯粒被同时加入候选，然而由于硬件路由端口数量有限，加入的芯粒只可能为常数个. 因此，*Neighbors* 中包含 $O(n)$ 个元素，对 *Neighbors* 排序的时间复杂度为 $O(n\log n)$. 经过 $<m$ 轮后，资源组扩展到 $<m$ 个芯粒，达到任务的资源要求，多轮扩展的复杂度为 $O(mn\log n)$. 因此，资源分配阶段的总时间复杂度为 $O(m\log m+mn\log n)$. 在大部分实际情况中，任务选择阶段复杂度 $O(k)$ 小于资源分发阶段 $O(m\log m+mn\log n)$. 因此，算法复杂度为 $O(m\log m+mn\log n)$，即与芯粒数量 m、芯粒中平均的 NPU 数量 n 均呈线性对数关系. 使用 N 表示总 NPU 核数，即 $N=m\times n$，因此有 $O(m\log m+mn\log n)<O(N\log N)$，即算法时间复杂度与芯粒中的 NPU 总数也呈线性对数关系，并随系统规模增长速度变缓.

2)空间复杂度分析. 算法 1 所需的运行空间为资源组 $Group_q$ 和候选队列 *Neighbors*，分别占用 $O(mn)$ 和 $O(n)$ 的额外空间. 因此，算法空间复杂度为 $O(mn)=O(N)$，并随芯粒数量、芯粒中 NPU 数量线性增长.

4 实验评估

4.1 实验设置

1)芯粒硬件资源配置. 本文在表 1 中列出了实验中使用的硬件配置，包括表征不同场景的 3 种集成芯片硬件资源组合(PA 表示 Post-Assembly，即集成芯片通过组装已有芯粒的方式形成的资源组合). PA-Small 中集成了多个算力较低、片上缓存较小的 NPU 芯粒，适用于边缘设备等低功耗场景；PA-Big 集成了高算力和较大片上缓存的 NPU 芯粒，代表了对功耗不敏感但需要高性能和实时服务(如云服务器)的场景；PA-Hybrid 同时集成了较大的 NPU 核和较小的 NPU 核，这种集成配置可能出现于工作负载可变的多租户应用场景，可以用于灵活处理用户上传的计算任务. 每个芯粒都使用现有的神经网络处理器相关工作中提出的智能处理器配置.

除了表 1 中列出的 NPU 芯粒配置外，芯片中还设置一个主控 CPU 芯粒和一个 I/O 芯粒. 本文使用双核 Arm Cortex-A53 CPU 作为主控制器，负责任务分配、同步和管理 NPU 核状态. 采用 16 GB DDR4 作为主存储器，它提供 25.6 GB/s 的带宽，并通过 I/O 芯粒上的内存控制器与其他芯粒通信. PA-Small 采用 128 b 链路带宽的芯粒间互连网络，而 PA-Big 和 PA-Hybrid 采用 256 b 链路带宽，数据包大小设为 4 微片(Flit). 对于 PA-Small, PA-Big 和 PA-Hybrid, 芯粒间互连网络的带宽分别为 12.8 GB/s, 25.6 GB/s, 25.6 GB/s.

2)系统模拟. 本文实现了一个系统级模拟器，用于评估延迟、内存访问和通信等芯片行为. 该模拟器中的 NPU 模拟模块可以准确计算深度学习集成芯片中 NPU 的性能(与 Scale-Sim[45] 相互验证)，并调用 Booksim 2.0[46] 获取通信延迟.

3)基线设置. 本文使用 PREMA[35] 和 Herald[36] 这 2 个代表性的多任务神经网络调度的相关工作作为基线. PREMA 使用时分复用的算法，并允许高优先级的工作任务抢占硬件资源. Herald 使用空分复用算法，基于任务在 NPU 上的利用率优先，将多个任务分配到多个异构的 NPU 处理器中.

4)工作负载配置. 基于常见的神经网络模型，构建了 Light, Mix 和 Heavy 这 3 种不同强度的工作负载，

Table 1 Integrated Deep Learning Chip Resource Configurations

表 1 深度学习集成芯片资源配置

芯片类型	芯粒间互连	NPU 芯粒类型	芯粒总数量	NPU 核数（芯粒数量）	片上网络类型	片上互连网络带宽 GB/s	NPU PE 阵列尺寸	NPU 片上缓存容量	数据流（芯粒数量）
PA-Small	3x3 Mesh	Eyeriss[44]	4	8（2） 3（2）	3×3 Mesh 2×2 Mesh	28.8	12×14	108KB	OS（2） WS（2）
		NN-Baton[31]	2	3	2×2 Mesh	28.8	8×8	64KB	WS
		TANGRAM[32]	2	8（1） 3（1）	3×3 Mesh 2×2 Mesh	28.8	16×16	128KB	OS
PA-Big	3x3 Mesh	TPU[13]	4	3（2） 1（2）	2×2 Mesh	57.6	256×256	8MB	OS
		PREMA[35]	2	3	2×2 Mesh	57.6	128×128	8MB（特征值） 4MB（权重）	WS
		AI-MT[37]	2	3	2×2 Mesh	57.6	128×128	18MB（特征值） 1MB（权重）	OS
PA-Hybrid	3x3 Mesh	TPU[13]	2	3	2×2 Mesh	57.6	256×256	8MB	OS
		Eyeriss[44]	2	8	3×3 Mesh	28.8	12×14	108KB	OS
		AI-MT[37]	1	3	2×2 Mesh	57.6	128×128	18MB（特征值） 1MB（权重）	OS
		TANGRAM[32]	1	3	2×2 Mesh	28.8	16×16	128KB	OS
		PREMA[35]	1	1		57.6	128×128	8MB（特征值） 4MB（权重）	WS
		NN-Baton[31]	1	8	3×3 Mesh	28.8	8×8	64KB	WS

注：Mesh 表示网格；WS 表示权重固定；OS 表示输出固定.

如表 2 所示. 其中，Mix 是较小的神经网络模型和较大神经网络模型的混合，用于模拟多租户场景下的各种可能的请求组合.

Table 2 Workload Configurations

表 2 工作负载配置

工作负载类型	神经网络模型	Batch 数量
Light	AlexNet[47]	8
	GoogleNet[48]	8
	MobileNet[49]	4
	ResNet-18[2]	8
Heavy	VGG-16[1]	8
	YOLO-v2[50]	16
	YOLO v3-tiny[51]	16
	ResNet-50[2]	16
Mix	AlexNet[47]	8
	ResNet-18[2]	8
	YOLO v3-tiny[51]	16
	ResNet-50[2]	1

4.2 实验结果

3.2 节中总结了基线策略所使用的时分复用、空分复用的优缺点呈现出的互补特性，以及在芯粒间通信、访存、资源空转时间方面分别存在优势和性能瓶颈. 而本文所提出的任务调度和资源分配方法是对时分复用和空分复用这 2 种基线方法的折中，综合了 2 种方法的优势，缓解 2 种方法的瓶颈问题导致的性能损失，最终降低总负载延迟.

本节首先依次展示 Puzzle 与基线方法在芯粒间通信、访存、资源空转时间的对比实验结果，并展示使用本文方法带来的总延迟变化，以及测试了 Puzzle 的可扩展性.

4.2.1 内存访问

内存访问是影响深度学习集成芯片性能的重要因素，因为多核芯片中的内存访问会导致共享资源的抢占，从而导致 I/O 芯粒路由器的排队延迟、I/O 芯粒附近链路的拥塞，以及 DRAM 的访问延迟. 在 Herald 使用的空分复用策略下，多个待执行任务同时执行，每个任务占据的资源有限，且由于第三方芯粒资源的异构性，Herald 使用的基于计算利用率的空分复用方法可能会导致计算任务的存储需求和分配的硬件资源之间的不匹配，即需要频繁与内存进行数据交换，导致冗余内存访问. 而在 PREMA 所使用的时分复用策略下，同一时刻只有一个任务占据全部硬件资源，是系统中片上缓存资源最充足的调度方案. 因此，PREMA 策略下产生的内存访问反映了系统内存访问的下界，即“必须”的内存访问.

Puzzle 生成的资源分配方案为 Herald 和 PREMA 策略的折中，通过启发式搜索，为任务分配满足需求的硬件资源组，且当空闲资源不足时，将挂起该任务以避免高冗余开销. 与 Herald 相比，Puzzle 降低了资

源不满足任务需求导致的冗余内存访问. 与 PREMA 策略下每个任务均使用全部资源相比, Puzzle 策略仍可能造成少量的冗余内存访问, 但已接近 PREMA 策略下产生的内存访问(即系统“必须”的内存访问). 如图 6 所示, PA-Small 中的 NPU 片上缓存较小(KB 级), 因此不适当的资源分配很容易导致冗余内存访问, 尤其是在 Heavy 工作负载下. 从 Light, Mix 到 Heavy 工作负载, 与 Herald 相比, Puzzle 减少了 21.9%、65.7% 和 86.0% 的内存访问. 对于 PA-Big, 其 NPU 均包含 MB 级别的片上缓存, Puzzle 与 Herald 均只进行必要的访存, 几乎没有冗余的内存访问, 因此访存数据量相似. 但 PA-Hybrid 在执行 Heavy 工作负载时, 由于其芯片配置中包含片上缓存较小的 NPU, 在这种配置下 Puzzle 相比于 Herald 可以减少 24.9% 的访存数据. 在这 3 种硬件配置下, Puzzle 产生的内存访问均只略高于最低内存访问(PREMA).

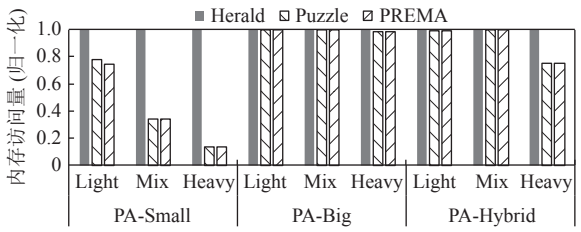


Fig. 6 Comparison of memory access amount

图 6 内存访问量对比

4.2.2 芯粒间通信

芯粒间通信是影响深度学习集成芯片性能的另一重要因素, 因为芯粒间通信需要在基板层完成, 而基板层的通信开销比片上互连网络高. 另外, 基板上的芯粒间通信链路和路由可能同时被多个芯粒抢占, 进一步降低其数据通信速度. 在 PREMA 所使用的时分复用策略下, 同一时刻只有一个任务占据全部硬件资源, 即所有 NPU 协作完成同一计算任务, 因此产生大量冗余的芯粒间通信. Herald 策略下计算任务在空间上复用硬件, 每个任务占用少量芯粒, 因此显示了最低的芯粒间通信开销. 而 Puzzle 为任务分配的资源组兼顾满足任务需求和减少不必要的芯粒间通信, 因此其芯粒间通信低于 PREMA 但高于 Herald.

如图 7 所示, 与 PREMA 相比, Puzzle 只为任务分配满足或接近于满足任务需求的资源, 且优先聚合通信距离近的资源, 因此降低了最高达 66.6% 的芯粒间通信. 在硬件配置 PA-Small 下, 当工作负载为 Light 时, Puzzle 分配给每个任务的资源远远小于 PREMA 分配的资源, 因此 Puzzle 显著降低了芯粒间通信. 随着工作负载的增加, 每个任务均需要更多的片上存储, 因此 Puzzle 也会为每个任务分配更多的资源, 即资源分配策略趋近于时分复用, 因此运行 Heavy 负载时的芯粒间通信量接近于 PREMA. 在硬件配置 PA-Big 下的实验结果反映了硬件资源相对充足的场景, 即硬件资源大于工作负载需求. 当执行 Heavy 工作负载时, Puzzle 可以降低冗余的芯粒间通信达 54.9%; 当执行 Light 和 Mix 工作负载时, 任务较小, 不需要大量资源, 然而, 为了避免大量资源空闲而导致高延迟, Puzzle 会将空闲的 NPU 附加到当前正在分配的资源组, 因此芯粒间通信的下降不如 Heavy 工作负载下显著, 但与 PREMA 相比, 仍然可以带来可观的芯粒间通信降低.

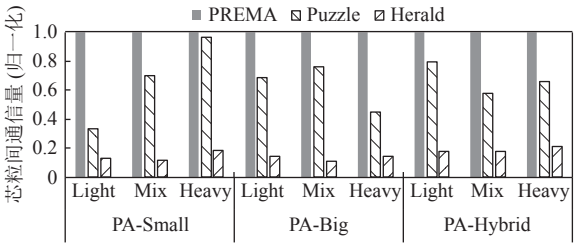


Fig. 7 Comparison of inter-die communication amount

图 7 芯粒间通信量对比

4.2.3 计算资源空转时间

如 3.2 节中所述, 在不均匀的多任务工作负载下, Herald 所使用的空分复用策略中短任务释放的资源有时无法被及时利用. 芯粒集成的异构性加剧了这一问题, 在异构资源下平衡任务时间差异更加困难.

图 8 展示了 Herald 和 Puzzle 调度策略下导致的资源空转时间对比. 从图 8 中可见, Herald 在一些配置下资源空转比例较低, 例如在 PA-Small 硬件配置下运行 Mix 和 Heavy 工作负载. 在另一些配置下资源空转比例较高, 例如在 PA-Big 硬件配置下运行 Mix 工作负载, 在 PA-Hybrid 下运行 Light 工作负载. 因此, Herald 仅适用于有限的部分配置, 不均匀的工作负载或硬件配置都会对 Herald 策略产生较大影响, 降低计算资源利用率. 而 Puzzle 策略根据系统中剩余任务和硬件资源状态动态地分配资源和调度任务, 更大程度上均摊了负载的不均匀性, 将空分复用策略下的平均资源空转时间比例由 31.0% 降低至 3.4%.

4.2.4 延 迟

根据 4.2.1 节、4.2.2 节、4.2.3 节的 3 个实验评估结果, PREMA 的策略存在访存量低、资源利用率高的优势, 但也存在高芯粒间通信开销. Herald 策略中芯粒间通信开销低, 但存在大量冗余访存和资源空

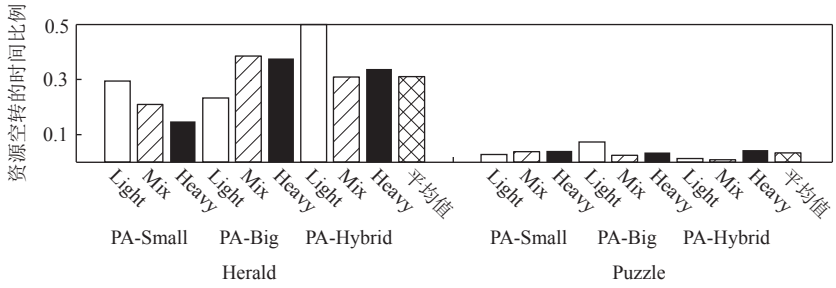


Fig. 8 Proportion of resource idle time

图 8 资源空转时间比例

转. Puzzle 实现了一种自适应的调度方案，根据系统中任务队列、任务大小和硬件资源状态生成任务调度和资源分配方案. Puzzle 综合了 PREMA 和 Herald 方案中的优势，在各种芯片配置和工作负载下均降低任务总延迟，如图 9(a)~9(c)所示.

在图 9(a)中，PA-Small 表示集成芯片中资源较少的场景，对于工作负载 Light，计算任务对硬件资源的需求也较少，PREMA 的策略会导致大量的冗余芯粒间通信(如 4.2.2 节所示)，因此 PREMA 的延迟最高. 对于工作负载 Mix 和 Heavy，Herald 策略会导致大量冗余内存访问(如 4.2.1 节所示)，造成高延迟. Puzzle 通过合理的资源分配策略实现了折中，既将片间通信控制在合理的范围内，又尽量不产生冗余的访存，与 PREMA 和 Herald 相比，Puzzle 的任务总延迟分别降低了 41.1% 和 67.0%.

在图 9(b)中，PA-Big 表示集成芯片中资源较充足的场景. 在这种情况下，只需要少量的 NPU 资源即可满足单个任务的需求. 因此，与 PREMA 相比，Puzzle 不会为单个任务分配过多的 NPU 资源，因此产生较少的片间通信，并降低总延迟 14.7%~36.4%. 此外，由于工作负载不均衡会导致较高的资源空转时间，Herald 在 Mix 工作负载下的延迟较高，而 Puzzle 通过合理的资源分配灵活地为任务生成资源组合，与 Herald 相比平均降低总延迟 21.8%. 在图 9(c)中，PA-Hybrid 表示集成芯片资源异构性较大的场景下，Puzzle 仍然保持自适应能力，自适应于硬件配置和工作负载，与基线策略相比，降低总延迟 8.0%~42.9%. 实验结果表明，Puzzle 可以自适应于多种工作负载和硬件资源配置，产生高效的任务调度和资源分配方案，降低工作负载总延迟.

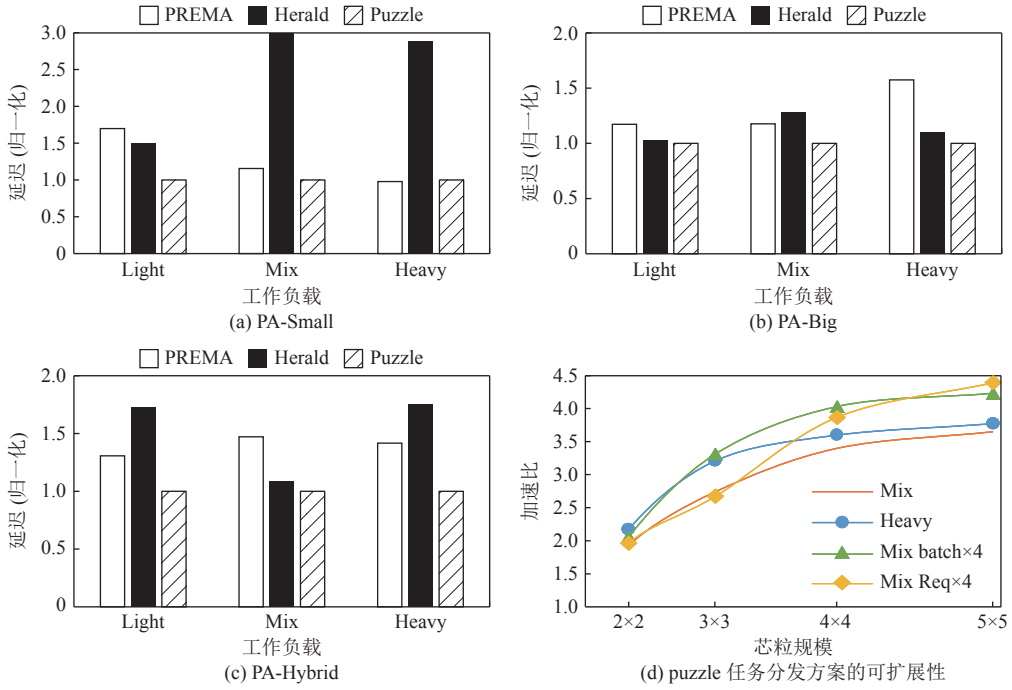


Fig. 9 Comparison of task latency and scalability

图 9 任务延迟对比和可扩展性

4.2.5 可扩展性

芯片的可扩展性通常从2个方面进行评估：纵向扩展(scale-up)和横向扩展(scale-out). 由于不同的硬件配置已经证明了Puzzle在scale-up方面的扩展性，本节评估Puzzle在scale-out方面的扩展性. 通过修改NPU芯粒的数量，将PA-Small缩放到2×2, 4×4, 5×5, 并评估同一个工作负载在这些缩放下的加速比. 如图9(d)所示，运行Mix工作负载时的可扩展性包含了4个不同的神经网络请求，可以看出，在Mix工作负载下，在25个芯粒的场景下仍能保持持续加速趋势. 另外，也评估了更大的工作负载强度的场景下，Puzzle的可扩展性. 实验结果表明，当工作负载强度更大时，Puzzle的可扩展性也仍然随之提高，尤其在部署请求数量更多的工作负载场景下.

5 结 论

基于芯粒集成技术进行深度学习芯片的敏捷定制可以降低芯片的开发周期和成本，因此成为新的发展趋势. 然而为定制化的深度学习集成芯片设计工具链会延长芯片的设计周期，并要求芯片开发人员熟悉第三方芯粒的架构设计细节. 本文提出了一种面向深度学习集成芯片的自适应编译和资源管理框架Puzzle，它使用两级资源图表示和管理深度学习集成芯片中的硬件资源，并根据不同的芯片硬件资源配置和工作负载自适应地生成高效的任务调度和资源分配方案. Puzzle提供了从输入工作负载到第三方芯粒接口的完整工作流程，为芯片敏捷定制的工具链设计提供了自动化的解决方案. 与主流的其他方案相比，Puzzle在不同的集成芯片配置和工作负载下均可以降低任务延迟，平均达到27.5%.

另外，Puzzle框架中仍有待完善之处：1)由于Puzzle框架中动态进行任务分发，因此，为了保证低运行时的开销，Puzzle的任务调度和资源分配算法中使用了部分贪心策略，所以并未达到全局最优，未来工作将尝试更多的优化策略，进行运行时开销和系统更优性能的权衡；2)在Puzzle中，并未针对计算图进行算子融合优化，且没有进行设计流水线并行的支持，在部分场景下浪费了算子间并行带来的优化空间. 未来我们将在Puzzle的计算图模块中加入图优化，进一步利用算子间并行，提升框架性能.

作者贡献声明：王梦迪设计并实现算法、撰写论文内容；王颖对论文选题、组织结构和论文写作提供了关键性的指导意见；刘成针对框架的模块和算法设计提出了关键性意见；常开颜、高成思协助设计了实验方案，并进行了实验对比分析；韩银和李华伟对论文的选题提供了重要的指导意见；张磊对论文选题进行了指导，并审阅与修订论文.

参 考 文 献

[1] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition[C/OL] //Proc of the 3rd Int Conf on Learning Representations. 2015[2023-04-01].https://arxiv.org/pdf/1409.1556.pdf

[2] He Kaiming, Zhang Xiangyu, Ren Shaoqing, et al. Deep residual learning for image recognition [C] //Proc of the 2016 IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2016: 770–778

[3] Liu Wei, Anguelov D, Erhan D, et al. SSD: Single shot multibox detector [C] //Proc of the European Conf on Computer Vision. Berlin: Springer, 2016: 21–37

[4] Ren Shaoqing, He Kaiming, Girshick R, et al. Faster R-CNN: Towards real-time object detection with region proposal networks [C] //Advances in Neural Information Processing Systems. New York: Curran Associates, Inc. , 2015: 91–99

[5] Hochreiter S, Schmidhuber J. Long short-term memory[J]. Neural Computation, 1997, 9(8): 1735−1780

[6] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need [C] //Advances in Neural Information Processing Systems. New York: Curran Associates, Inc. , 2017: 5998–6008

[7] Devlin J, Chang Mingwei, Lee K, et al. BERT: Pre-training of deep bidirectional transformers for language understanding [C] //Proc of the 2019 Conf of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Minneapolis, Stroudsburg, PA: Association for Computational Linguistics, 2019: 4171–4186

[8] NVIDIA. DGX-2 : AI servers for solving complex AI challenges [EB/OL]. 2018[2023-04-01]. https://www.nvidia.com/en-us/data-center/dgx-2/

[9] Google. Edge TPU - run inference at the edge [EB/OL]. 2018[2023-04-01].https://cloud.google.com/edge-tpu

[10] Huawei. Ascend 310 AI processor[EB/OL]. 2018[2023-04-01].https://e.huawei.com/eu/products/cloud-computing-dc/atlas/ascend-310

[11] Chen Tianshi, Du Zidong, Sun Ninghui, et al. DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning [C] //Proc of the 19th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM , 2014: 269–284

[12] NVIDIA. NVDLA primer—NVDLA documentation [EB/OL].

2017[2023-04-01]. http://nvdla.org/primer.html

[13] Google. Cloud TPU [EB/OL]. 2018[2023-04-01].https://cloud.google.com/tpu

[14] Tesla. Artificial intelligence & autopilot [EB/OL]. 2019[2023-04-01].https://www.tesla.com/AI

[15] Cerebras. Cerebras CS-2 - system [EB/OL]. 2021[2023-04-01].https://www.cerebras.net/product-system/

[16] Bao Yungang, Chang Yisong, Han Yinhe, et al. Agile design of processor chips: Issues and challenges[J]. Journal of Computer Research and Development, 2021, 58(6): 1131−1145 (in Chinese)
(包云岗, 常轶松, 韩银和, 等. 处理器芯片敏捷设计方法: 问题与挑战[J]. 计算机研究与发展, 2021, 58(6): 1131−1145)

[17] Kim J, Krishna C V C, Rahman N M, et al. Silicon vs. organic interposer: PPA and reliability tradeoffs in heterogeneous 2.5D chiplet integration [C] //Proc of the 2020 IEEE 38th Int Conf on Computer Design. Piscataway, NJ: IEEE, 2020: 80−87

[18] Li Tao, Hou Jie, Yan Jinli, et al. Chiplet heterogeneous integration technology—Status and challenges[J]. Electronics, 2020, 9(4): Article No.670

[19] Bharadwaj S, Yin Jieming, Beckmann B, et al. Kite: A family of heterogeneous interposer topologies enabled via accurate interconnect modeling [C] //Proc of the 2020 57th ACM/IEEE Design Automation Conf. Piscataway, NJ: IEEE, 2020: 1–6

[20] Banijamali B, Chiu C, Hsieh C, et al. Reliability evaluation of a cowos-enabled 3D IC package [C] //Proc of the 2013 IEEE 63rd Electronic Components and Technology Conf. Piscataway, NJ: IEEE, 2013: 35–40

[21] Gomes W, Khushu S, Ingerly D B, et al. 8.1 lakefield and mobility compute: A 3D stacked 10nm and 22FLL hybrid processor system in 12×12mm^2, 1mm package-on-package [C] //Proc of the 2020 IEEE Int Solid- State Circuits Conf. Piscataway, NJ: IEEE, 2020: 144−146

[22] Singh T, Rangarajan S, John D, et al. 2.1 zen 2: The AMD 7nm energy-efficient high-performance x86−64 microprocessor core [C] //Proc of the 2020 IEEE Int Solid- State Circuits Conf. Piscataway, NJ: IEEE, 2020: 42−44

[23] Gomes W, Koker A, Stover P, et al. Ponte Vecchio: A multi-tile 3D stacked processor for exascale computing [C] //Proc of the 2022 IEEE Int Solid- State Circuits Conf. Piscataway, NJ: IEEE, 2022: 42−44

[24] Arunkumar A, Bolotin E, Cho B, et al. MCM-GPU: Multi-chip-module gGPU for continued performance scalability [J].ACM SIGARCH Computer Architecture News, 2017, 45(2): 320–332

[25] Hwang R, Kim T, Kwon Y, et al. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations [C] //Proc of the 2020 ACM/IEEE 47th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2020: 968–981

[26] Shao Y S, Clemons J, Venkatesan R, et al. Simba: Scaling deep-learning inference with multi-chip-module-based architecture [C] //Proc of the 52nd Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2019: 14–27

[27] Pal S, Liu Jingyang, Alam I, et al. Designing a 2048-chiplet, 14336-core waferscale processor [C] //Proc of the 2021 58th ACM/IEEE Design Automation Conf. Piscataway, NJ: IEEE, 2021: 1183–1188

[28] Naffziger S, Lepak K, Paraschou M, et al. 2.2 AMD chiplet architecture for high-performance server and desktop products [C] // Proc of the 2020 IEEE Int Solid-State Circuits Conf. Piscataway, NJ: IEEE, 2020: 44−45

[29] AMD. UCIe [EB/OL]. 2022[2023-04-01].https://www.uciexpress.org

[30] Vinnakota B, Agarwal I, Drucker K, et al. The open domain-specific architecture: 1[J]. IEEE Micro, 2021, 41(1): 30−36

[31] Tan Zhanhong, Cai Hongyu, Dong Runpeng, et al. NN-Baton: DNN workload orchestration and chiplet granularity exploration for multichip accelerators [C] //Proc of the 2021 ACM/IEEE 48th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2021: 1013–1026

[32] Gao Mingyu, Yang Xuan, Pu Jing, et al. TANGRAM: Optimized coarse-grained dataflow for scalable nn accelerators [C] //Proc of the 24th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM , 2019: 807–820

[33] Venkataramani S, Dubey P, Raghunathan A, et al. ScaleDeep: A scalable compute architecture for learning and evaluating deep networks [C] //Proc of the 44th Annual Int Symp on Computer Architecture. New York: ACM , 2017: 13–26

[34] Song Linghao, Mao Jiachen, Zhuo Youwei, et al. HyPar: Towards hybrid parallelism for deep learning accelerator array [C] //Proc of the 2019 IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2019: 56–68

[35] Choi Y, Rhu M. PREMA: A predictive multi-task scheduling algorithm for preemptible neural processing units [C] //Proc of the 2020 IEEE Int Symp on High Performance Computer Architecture. Piscataway, NJ: IEEE, 2020: 220–233

[36] Kwon H, Lai Liangzhen, Pellauer M, et al. Heterogeneous dataflow accelerators for multi-DNN workloads [C] // Proc of the 2021 IEEE Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2021: 71−83

[37] Baek E, Kwon D, Kim J. A multi-neural network acceleration architecture [C] //Proc of the 2020 ACM/IEEE 47th Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2020: 940–953

[38] Liu Zihan, Leng Jingwen, Zhang Zhihui, et al. VELTAIR: Towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling [C] //Proc of the 27th ACM Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2022: 388–401

[39] Ghodrati S, Ahn B H, Kyung Kim J, et al. Planaria: Dynamic architecture fission for spatial multi-tenant acceleration of deep neural networks [C] // Proc of the 2020 53rd Annual IEEE/ACM Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2020: 681–697

[40] Oh Y H, Kim S, Jin Y, et al. Layerweaver: Maximizing resource

utilization of neural processing units via layer-wise scheduling [C] //Proc of the 2021 IEEE Int Symp on High-Performance Computer Architecture. Piscataway, NJ: IEEE, 2021: 584–597

[41] Zhao Yongwei, Du Zidong, Guo Qi, et al. Cambricon-F: Machine learning computers with fractal von neumann architecture [C] //Proc of the 46th Int Symp on Computer Architecture. New York: ACM, 2019: 788–801

[42] Yang Xuan, Gao Mingyu, Liu Qiaoyi, et al. Interstellar: Using halide's scheduling language to analyze dnn accelerators [C] //Proc of the 25th Int Conf on Architectural Support for Programming Languages and Operating Systems. New York: ACM, 2020: 369–383

[43] Chen Tianqi, Moreau T, Jiang Ziheng, et al. TVM: An automated end-to-end optimizing compiler for deep learning [C] //Proc of the 13th USENIX Conf on Operating Systems Design and Implementation. Berkeley, CA: USENIX Association, 2018: 579–594

[44] Chen Y, Emer J, Sze V. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks [C] //Proc of the 2016 ACM/IEEE 43rd Annual Int Symp on Computer Architecture. Piscataway, NJ: IEEE, 2016: 367–379

[45] Samajdar A, Zhu Y, Whatmough P, et al. SCALE-Sim: Systolic CNN accelerator simulator [J]. arXiv preprint, arXiv: 1811.02883, 2019

[46] Jiang N, Balfour J, Becker D U, et al. A detailed and flexible cycle-accurate network-on-chip simulator [C] //Proc of the 2013 IEEE Int Symp on Performance Analysis of Systems and Software (ISPASS). Piscataway, NJ: IEEE, 2013: 86–96

[47] Krizhevsky A, Sutskever I, Hinton G E. ImageNet classification with deep convolutional neural networks [C] //Proc of the 2012 Advances in Neural Information Processing Systems. New York: Curran Associates, Inc. , 2012: 84–90

[48] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions [C] //Proc of the 2015 IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2015: 1–9

[49] Howard A G, Zhu Menglong, Chen Bo, et al. MobileNets: Efficient convolutional neural networks for mobile vision applications [J]. arXiv preprint, arXiv: 1704.04861, 2017

[50] Redmon J, Farhadi A. YOLO9000: Better, faster, stronger [C] //Proc of the IEEE Conf on Computer Vision and Pattern Recognition. Piscataway, NJ: IEEE, 2017: 7263–7271

[51] Adarsh P, Rathi P, Kumar M. YOLO v3-tiny: Object detection and recognition using one stage improved model [C] //Proc of the 2020 6th Int Conf on Advanced Computing and Communication Systems. Piscataway, NJ: IEEE, 2020: 687–694

Wang Mengdi, born in 1995. PhD candidate. Her main research interests include accelerator design and multi-core system.

王梦迪, 1995 年生. 博士研究生. 主要研究方向为加速器设计和多核系统.

Wang Ying, born in 1985. PhD, associate professor. Senior member of CCF. His main research interests include novel EDA, and processor and memory architecture.

王　颖, 1985 年生. 博士, 副研究员. CCF 高级会员. 主要研究方向为新型 EDA、处理器与存储系统体系结构.

Liu Cheng, born in 1984. PhD, associate professor. Member of CCF. His main research interests include specific accelerator design and fault-tolerant computing.

刘　成, 1984 年生. 博士, 副研究员. CCF 会员. 主要研究方向为专用加速器设计和容错计算.

Chang Kaiyan, born in 1999. PhD candidate. Student member of CCF. His main research interests include compiler and programming language.

常开颜, 1999 年生. 博士研究生. CCF 学生会员. 主要研究方向为编译器和编程语言.

Gao Chengsi, born in 1995. PhD candidate. His main research interests include computer architecture and neural network processors.

高成思, 1995 年生. 博士研究生. 主要研究方向为计算机体系结构和神经网络处理器.

Han Yinhe, born in 1980. PhD, professor. Distinguished member of CCF. His main research interests include computer architecture and processor chips.

韩银和, 1980 年生. 博士, 研究员. CCF 杰出会员. 主要研究方向为计算机体系结构和处理器芯片.

Li Huawei, born in 1974. PhD, professor. Fellow of CCF. Her main research interest includes electronic design automation of digital circuits.

李华伟, 1974 年生. 博士, 研究员. CCF 会士. 主要研究方向为数字电路电子设计自动化.

Zhang Lei, born in 1981. PhD, professor. Member of CCF. His main research interests include chip and computing system.

张　磊, 1981 年生. 博士, 正高级工程师. CCF 会员. 主要研究方向为芯片和计算系统.