第 3 章 NPU 开发技术分析

3.1 NPU 加速器建模设计

神经处理单元(Neural Processing Unit,NPU)是一种专门用于进行深度学习计算的芯片。它是近年来人工智能领域的热门技术之一,被广泛地应用于各种人工智能应用中,如自动驾驶、人脸识别、智能语音等领域。

3.1.1 NPU 加速器建模概述

对 NPU 进行建模的主要目的是快速评估不同算法、硬件、数据流下的时延、功耗开销,因此在架构的设计初期,可以评估性能瓶颈,方便优化架构。在架构和数据流确定后,建模可以快速评估网络在加速器上的执行效率。对其建模可以从算法维度和硬件维度进行。

(1) 算法维度:以一定的方式表示需要加速的网络,如一些中间件描述,主要包括算子的类型、网络的层数及操作数精度等信息,这一部分可以由自定义的网络描述文件表示,也可以由编译器解析网络文件生成,其目的在于定量地描述工作负载。

(2) 硬件维度:包括计算资源和存储资源两部分,不同的 NPU 具有不同数量的计算资源,加速不同算子的并行度,也会有所区别。不同 NPU 的存储层次结构也有很大区别,权重、特征是否共用同一片缓存,或者有各自独立的缓存,每一级存储的容量、带宽及相互之间的连接关系都是设计空间的一部分。

建模分析的主要问题是功耗、时延及访存。对功耗而言,通常具有统一的分析方法,通过每个操作的功耗,例如一次乘法,加载一个数据等需要的功耗,以及总的操作次数相乘后累加,就可以估算出整体的功耗。时延可以通过 RTL/FPGA 等精确的仿真得到,适用于架构与数据流确定的情况。另外有一些基于数学方法分析的建模方法,并不会实际执行网络,而是根据网络参数、硬件参数进行数学推导,估算时延。对访存的估计与需求有关,有些建模方法只关注于对 DRAM 的访问,有些则会同时考虑到片上不同存储单元间的数据移动。

建模越精确,其越贴近特定的架构,更有利于在特定硬件上评估算法的计算效率。建模越模糊,越具有普适性,有利于在加速器设计初期进行设计空间的探索。

3.1.2 加速器架构的设计空间探索

以下以最近的一篇论文为例，来分析加速器架构的设计空间探索，*DeFiNES*：*Enabling Fast Exploration of the Depth-first Scheduling Space for DNN Accelerators through Analytical Modeling* 中考虑了 PE 利用和内存层级结构，对于 PE 的利用，其主要是 PE 间的数据交互、排列和连接方式，并没有太大的探索空间，即计算和数据搬移的共同代价，在进行一些架构分析时，多集中于开关芯片，对内存层级关注较低，这里在深度优先的搜索加速器架构空间中，使用成本模型(Cost Model)原理找到最优架构模型。从以下 3 个层面分析：

(1) 逐次计算分块大小(会影响层间的输入和输出，位于哪个内存层级上，以及权重如何复用的问题，即权重访问的频次)。

(2) 数据复用模式(选择缓存已计算的数据，还是完全重新选择数据)。

(3) 层间融合(将层间存储在已知存储区内，完成上一层的输出，送入下一层的输入)。

依据以上 3 方面来探讨搜索空间与减少高层存储访问，需要一定的权衡折中。

成本模型一般考虑延迟和功耗两部分，尚未解决的问题是论文主要针对卷积进行的分析，以 Transform 为代表的大模型，计算模式则完成不同。在卷积计算中，权重复用、特征映射的滑动、感受野计算区域变化等，与 Transform 差距较大。

优点是详细的内存层级分布的探讨和不同容量层级的内存分布，值得借鉴；缺点是成本模型并未真正提及，对于卷积并没有关注到深度和点两种常见版本，对于 Transform 新的计算模式并未涉及。

1. 从单层一次调度到逐层调度，再到深度优先调度

层级从单层一次调度到逐层调度，再到深度优先调度，以保持较低的激活率内存级别，如图 3-1 所示。其中，L 表示神经网络层；T 表示分块；LB 表示本地缓冲区(小型片上存储器)；GB 表示全局缓冲区(较大的片上存储器)。

2. 设计空间的第 1 个轴：分块尺寸

DF 设计空间的第 1 个轴：分块尺寸，可以进行图层标注，如图 3-2 所示。

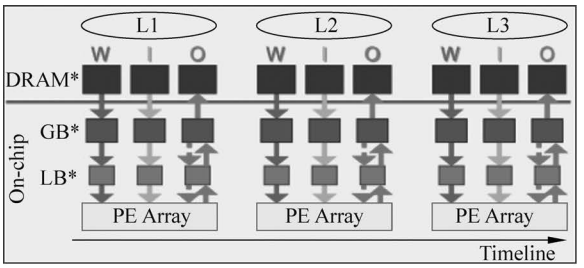
其中，K 表示输出通道；C 表示输入通道；OX 和 OY 是特性映射空间维度；FX 和 FY 是权重空间维度。

从图 3-2 可以看到，卷积的计算特点、权重给输入空间滑动带来了 3 个结果：

(1) 支持逐个模块的计算，即扩展跨层分块计算。

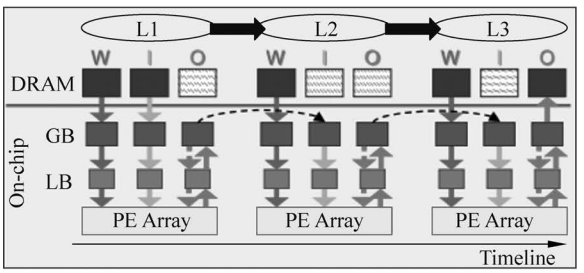
(2) 数据的生产者消费者模式，即步长与内核大小差异引起的数据复用，以及层间连接的数据交付。

(3) 计算模式导致的存储结构与权重在层内的复用，而分块大小影响了计算时与权重的访问频次。基于融合层感受野的影响(卷积的计算结构)，较大的分块大小带来了较好的计算效率，对比图 3-3，可以看到 tile_size=4×4，最上层的输入为 10×10，tile_size 为 1×1，输入为 7×7。



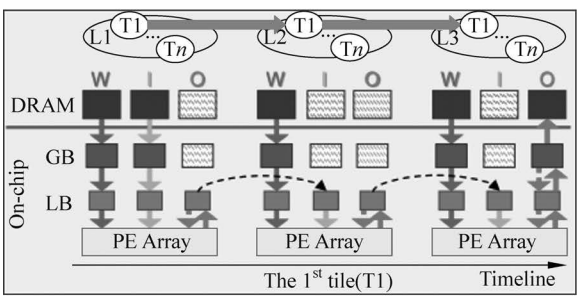
在图(a)单层(SL)的示例中，对所有图层，将每个层视为分离的工作负载：
(1) DRAM的输入(I)。
(2) DRAM的输出(O)。
(3) DRAM的重量(W)。
这里假定所有内存级别由W/I/O共享。

(a) 单层(SL)的示例



在图(b)逐层(LBL)的示例中，除最后一层外，考虑层之间的过渡：
(1) 如果合适，则层的输出可以留在芯片上作为下一层的输入(跳过DRAM)。
(2) 计算所有图层DRAM的权重。

(b) 逐层(LBL)的示例



在图(c)深度优先(DF)/层融合的示例中，将每一层分割成小块，并首先跨层深度处理每一块分块。对于T1到T*n*分块中的L1到L3层，计算层平铺：
(1) 如果合适，则层分块的输出可以保持在底层芯片上，成为下一层分块的输入(跳过DRAM和GB)。
(2) 计算DRAM中第1个分块(T1)的权重。

(c) 深度优先(DF)/层融合的示例

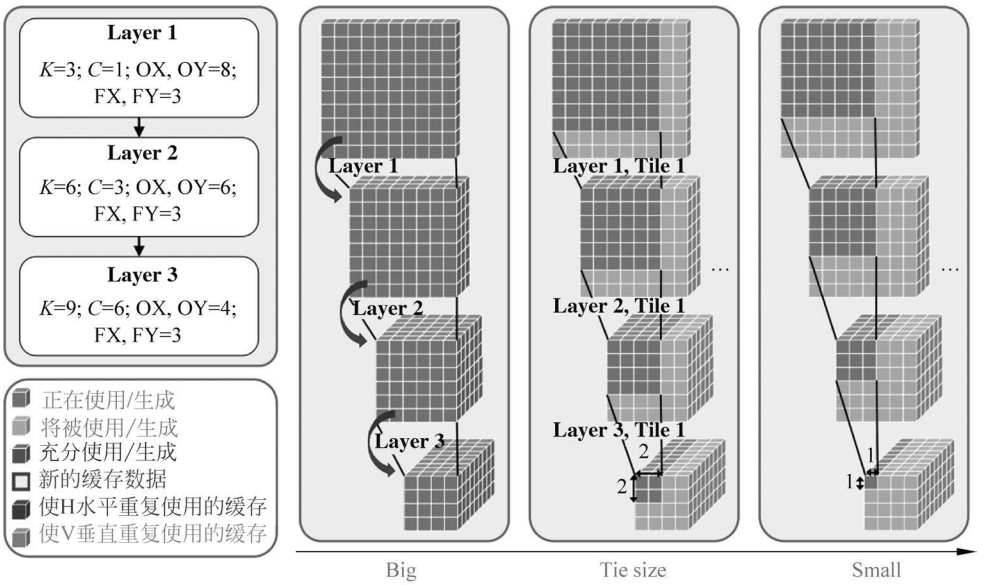
图 3-1　层级调度

3. 设计空间的第 2 个轴：重叠存储模式

DF 设计空间的第 2 个轴：重叠存储模式如图 3-3 所示。

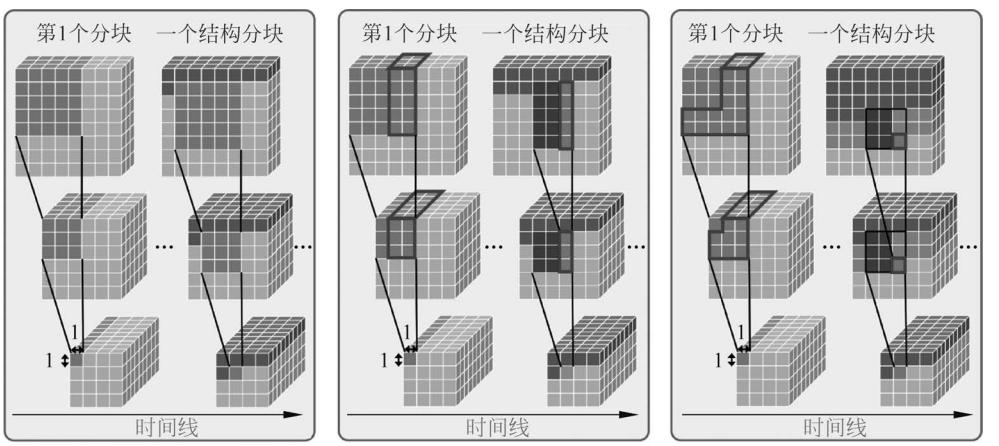
工作量为图 3-2(a)中的第 2 层和第 3 层。

紫色的表示计算已经生成的数据，对于图 3-2(a)为完全每次都从第 1 层开始重新计算的模式，表示最后一层生成一个 $1\times1\times c$ 绿色的数据，倒数第 2 层需要提供 $3\times3\times c$ 绿色数据，第 1 层需要提供绿色 $5\times5\times c$ 绿色数据，因为其属于完全重新计算，即没有数据复用，所以可以看到底层的(c)个数据需求，前两层分别需要用到 $9c$ 个和 $25c$ 个。图 3-2(b)属于 H 缓存、V 重新计算，即水平方向缓存、垂直方向计算，在最后一行中生成一个 $1\times1\times c$ 绿色数据，对应上面一层需要 $3\times3\times c$ 的数据块运算，其中需要复用缓存的红色 $2\times3\times c$ 的数据，增加了绿色 $1\times3\times c$ 的数据，新加入的 1×3 的绿色数据会被设置成 $1\times3\times c$ 的新缓存数据，作为下一次的领域计算的缓存数据，如图 3-3 蓝色框所示，图 3-2(b)中新读入的 $1\times3\times c$



(a) 工作量和图例　(b) LBL(1块/层)　(c) 分块尺寸2×2　(d) 分块尺寸1×1

图 3-2　DF 设计空间的第 1 个轴：分块尺寸(见彩插)



(a) 完全重新计算　(b) H缓存、V重新计算　(c) 完全缓存

图 3-3　DF 设计空间的第 2 个轴：重叠存储模式(见彩插)

数据，则对应最上层需要新缓存 $1\times5\times c$ 的数据，图 3-2(c)同理。

4. 分块尺寸(第 1 轴)和融合深度(第 3 轴)

分块尺寸(第 1 轴)和融合深度(第 3 轴)的影响，如图 3-4 所示。

ST 表示融合层栈。当一层一个栈时，每层的权重比较小，则将其放置在 LB 中，因为其栈很浅，所以每层的栈之间的 I/O 都会被写到最慢的 DRAM 中，而其每个栈的 I/O(上一层的输出传递到下一层的输入)，也只能在低级别存储中传递，如图 3-4(a)所示。

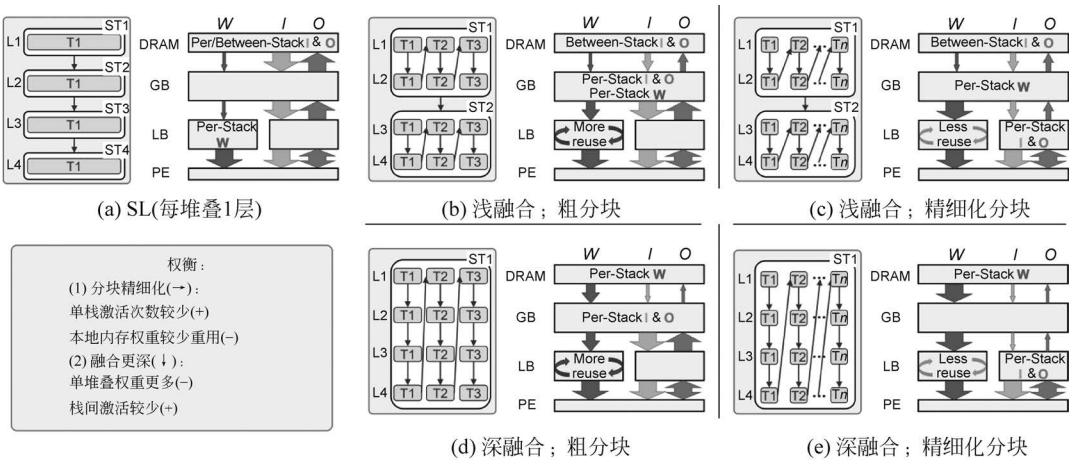


图 3-4　分块尺寸(第 1 轴)和融合深度(第 3 轴)的影响

图 3-4(b)与图 3-4(c)有相同之处，即都进行了融合，对比图 3-4(a)来讲，因为层融合了，栈变深，多层间累计的权重变大，权重从原本的 LB，被迫放在了 GB 中。对于图 3-4(b)与图 3-4(c)，分块粒度变细后，其相应的执行次数变多，权重访问频次变高，图 3-4(c)重新使用的权重变少。

对于与图 3-4(a)比较，因为进行融合了，所以上一层的输出传递到下一层的输入，对于每个栈的 I/O，因为图 3-4(a)为单层执行，即每次结果需要放回 DRAM，所以栈之间的 I/O 被放在 DRAM，因为只有一层，所以其输出 I/O 和栈之间是一样的，从图 3-4(a)中的 DRAM 移动到了 GB 中或者 LB 中。因为分块粒度变小，所以图 3-4(c)的每个栈位于 LB 中，而图 3-4(b)的每个栈位于 GB 中，对于栈之间，无论是图 3-4(b)还是图 3-4(c)都被写入最外层 DRAM 中。对于图 3-4(b)和图 3-4(c)而言，融合层比较浅，对比可以看出，分块越细，每层的特征映射越小，每个栈的 I/O 越容易被放到高速缓存中。在图 3-4(c)中，每个栈的输入特征映射与输出特征映射集中在最底层的 LB 上，而在图 3-4(b)中，则放在 GB 中，即细分导致每栈激活次数较少，但是同时也带来了缺点，多个分块则意味着更多次的访问权重，即细分使较少的本地内存权重被重用，从图 3-4(c)中可以看到，*W* 较少被使用。

对比图 3-4(b)与图 3-4(d)可以看到，融合层数越多，即融合越深，每个栈包含的层数就越多，一个栈包含多层的权重也就越多，因此每栈权重越多。对应图 3-4(b)，权重可以在 GB 中。在图 3-4(d)与图 3-4(e)中，权重数据量较多地集中在了 DRAM 上。融合更深的好处是，这些栈中逐层之间的激活，在高速存储中完成了交换(上一层的输出是下一层的输入)，图 3-4(d)中的 DRAM 没有栈之间的 I/O，I/O 集中在下面的高速层，即栈间激活较少。

5. DeFiNES 概述

DeFiNES 的概述如图 3-5 所示。

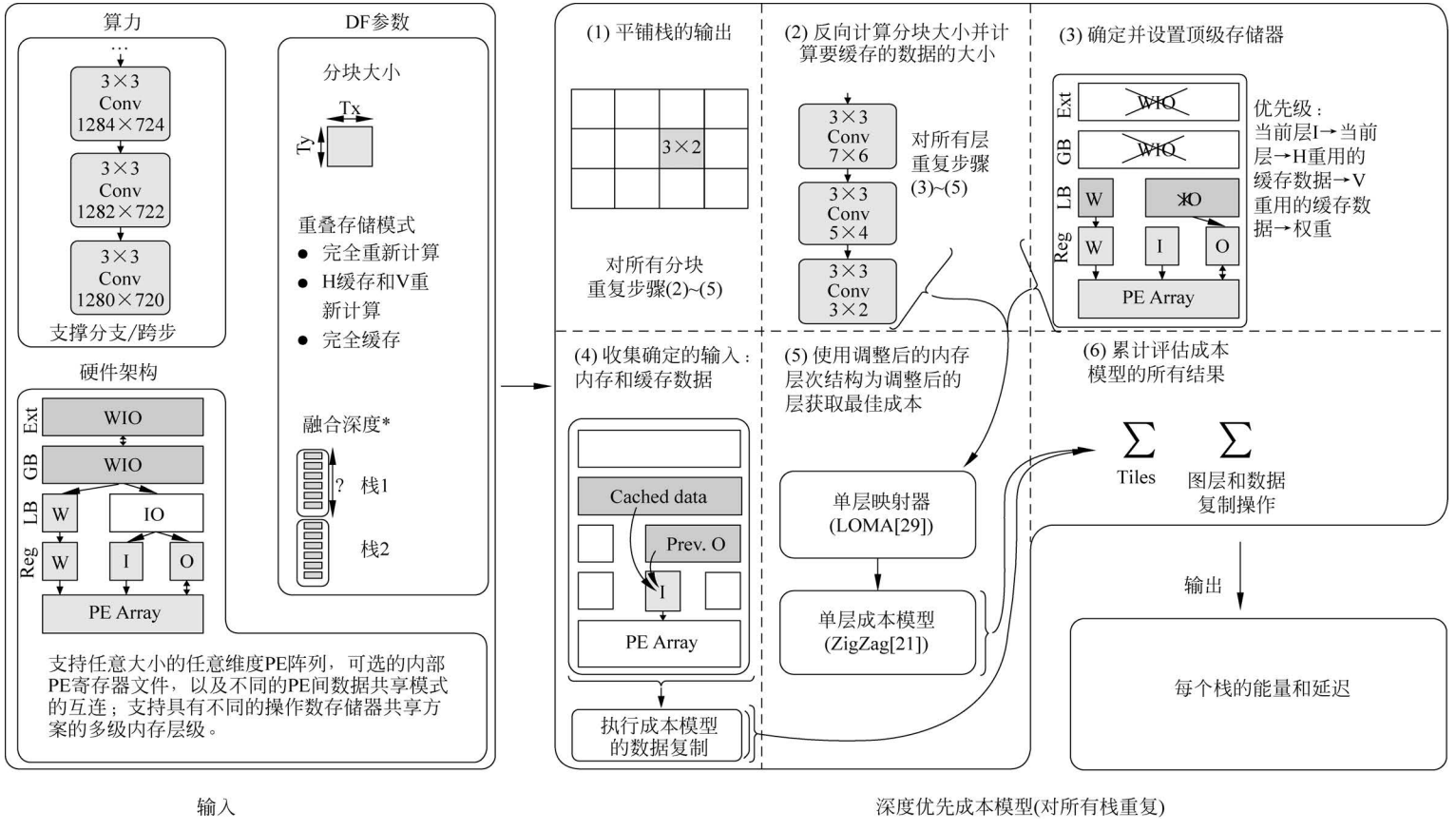


图 3-5　DeFiNES 概述(＊为可选输入，可自动设置)

因为第 1 行/列中的块还没有可用的缓存数据，同样，最后一列/行中的块也不必为它们的邻居存储重叠数据，因此不是所有的分块都是相同的。

6. 分块大小与分块类型计数

不同分块大小和重叠存储模式的分块类型计数如图 3-6 所示。

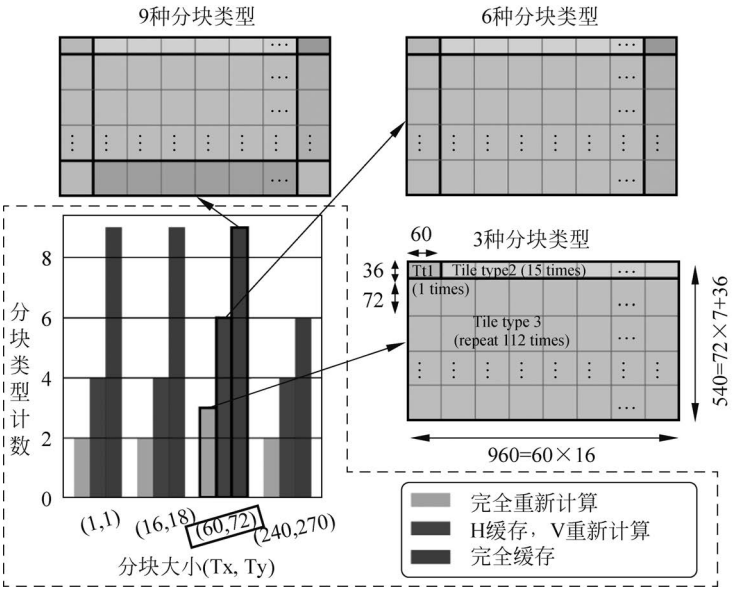


图 3-6　不同分块大小和重叠存储模式的分块类型计数(见彩插)

本例中使用的工作负载是 FSR-CNN，其最终输出特性映射的空间维度为 960×540。进一步使用 3 分块类型示例，如图 3-9 和图 3-10 所示。

7. 不同重叠存储模式所需的数据存储

不同重叠存储模式所需的数据存储可以利用高速缓存，以便进行邻居数据缓存，如图 3-7 所示。

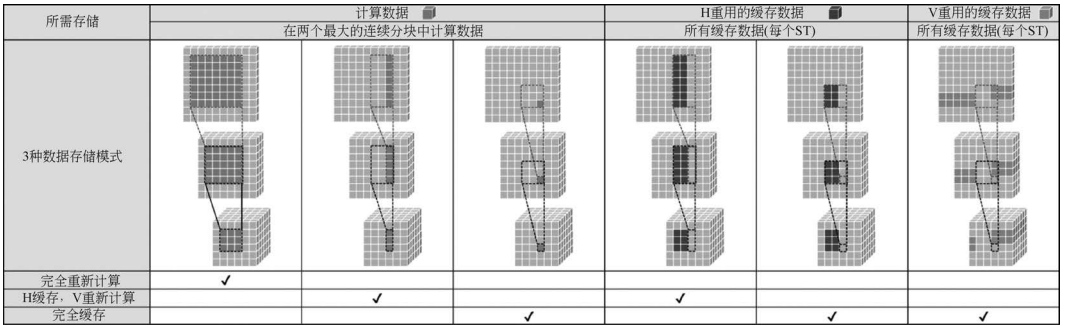


图 3-7　不同重叠存储模式所需的数据存储(见彩插)

8. DeFiNES 对分支的处理

反向计算分块大小对于每层中的每个分块计算要缓存的数据的大小。

根据栈中最后一个输出特征图的分块大小，计算最后一层输入所需的分块大小。接下来，计算上一层的待计算平铺大小。在没有缓存可重用的情况下，这只是等于最后一层输入所需的分块大小，然而，对于跨分块重用的缓存，并不是所有这些特性都需要计算，因为一些特性可以从缓存的数据中提取，如图 3-3(b)和图 3-3(c)所示。对栈中的所有层重复该过程，并且由此确定栈中的每个层的输入分块大小和计算输出分块大小。

在这个反向计算过程中，算法还跟踪每种类型的数据(包括早期或未来重叠分块)应该缓存多少，如图 3-7 所示。对分支情况的处理如图 3-8 所示，左分支和右分支需要缓存来自要素图中不同位置的要素。在这种情况下，通过组合到缓存区域的所有最外侧边缘，用以设置要缓存的特征的整个区域，以便所有分支始终具有在重叠缓存模式下操作的缓存特征，使 FM1 的中间组合可视化。

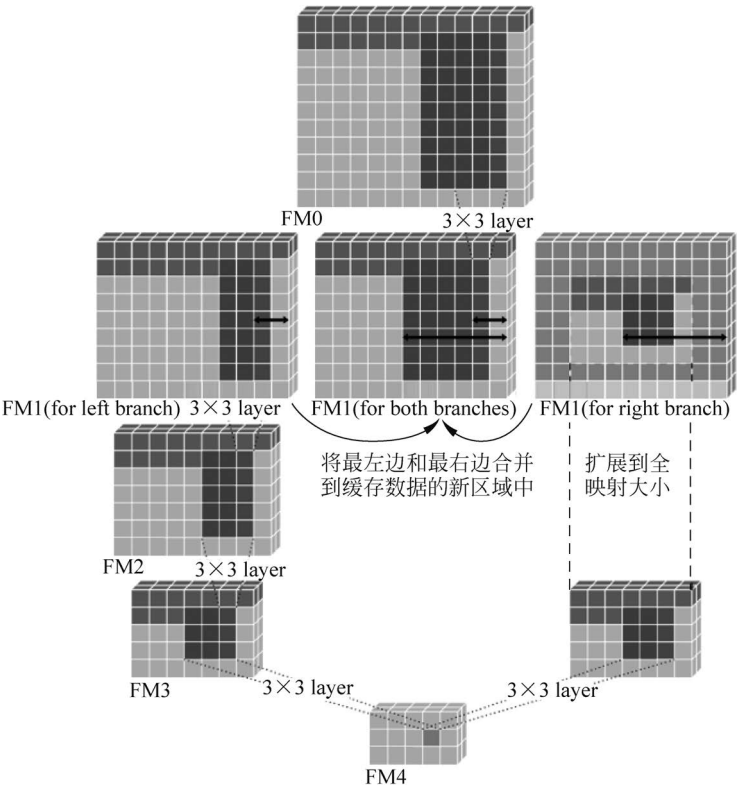


图 3-8　DeFiNES 对分支的处理(见彩插)

灰色像素对右侧分支没有贡献，FM 表示功能图。

9. 内存排列分布

内存排列分布如图 3-9 所示。

从图 3-6 可以看出，DF 调度取自 3 分块类型示例。HW 架构是表 3-1 中的 Idx2。

(1) 对于权重，第一分块的所有层从 DRAM 获取权重，而其他层分块组合从 LB 获取权重。

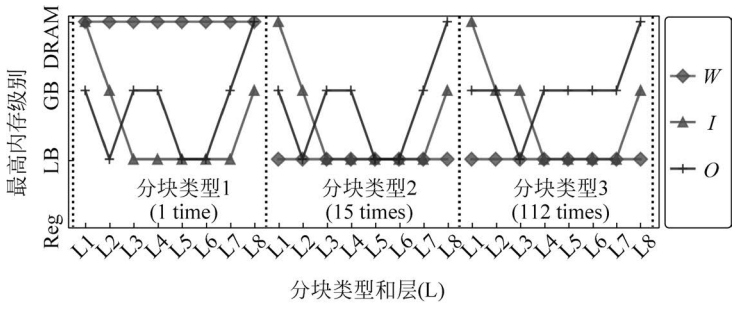


图 3-9　内存排列分布(见彩插)

(2) 对于输入和输出,所有分块的第 1 层从 DRAM 获得输入,所有分块最后一层写入。

从图 3-9 可以看到,图像被分割成了 3 种分块,分块 1 的数目为 1 个,其权重首次需要从 DRAM 逐层搬移进来。在分块类型 1 中,其权重位于 DRAM 中,在其后的 type2(15 个)和 type3(112 个)中权重是完全复用的,所以一直在 LB 结构中,对应于 5_step3 图中 W 位于 LB 中,而 I 因为有存储计算时的缓存存在,前一层生成的输出可能部分被缓存起来,或者使用更低的内存级别来作为下一层的输入。在每种类型切换时会从 DRAM 中读取一次数据,因此,数据从 DRAM 中读入后,以后每次使用时,一部分是新数据;另一部分是来源于缓存的数据。在 type2 和 type3 中,数据基本在 LB 中,而对于 O 只有类型切换和最终结果,并会写到 DRAM 中,如图 3-5 所示,从 step4 可以看到,预先的输出特征映射,在内存层级中更优先于高速缓存数据。

10. 激活数据大小的可视化

图 3-9 中的分块类型 2 和 3 中激活数据大小的可视化,如图 3-10 所示。

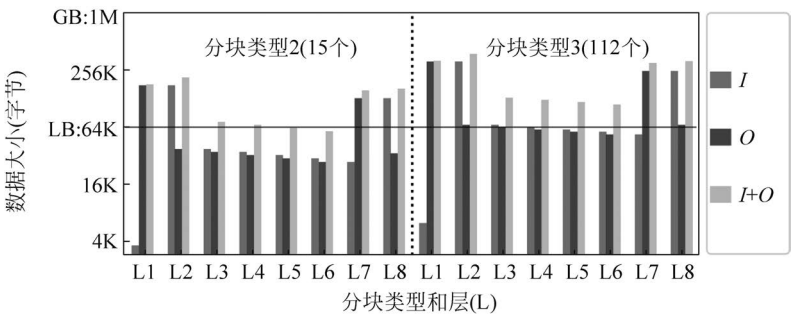


图 3-10　图 3-9 中分块类型 2 和 3 中激活数据大小的可视化(见彩插)

如图 3-9 和图 3-10 所示,LB 和 GB 的容量在 y 轴上标出,可得到如下信息:

(1) 当总激活大小($I+O$)可以放入 LB(例如,分块类型 2-L6)时,LB 是 I/O 的顶部存储器。

(2) 当总激活大小($I+O$)不能容纳 LB,而 I 或 O 可以容纳(例如,分块类型 3-L6)时,I 优先使用 LB 作为其最高内存级别,而 O 被推到 GB。

11. 芯片测量结果比较

对 DeFiNES 的结果与 DepFiN 芯片的测量结果进行比较,如图 3-11 所示。

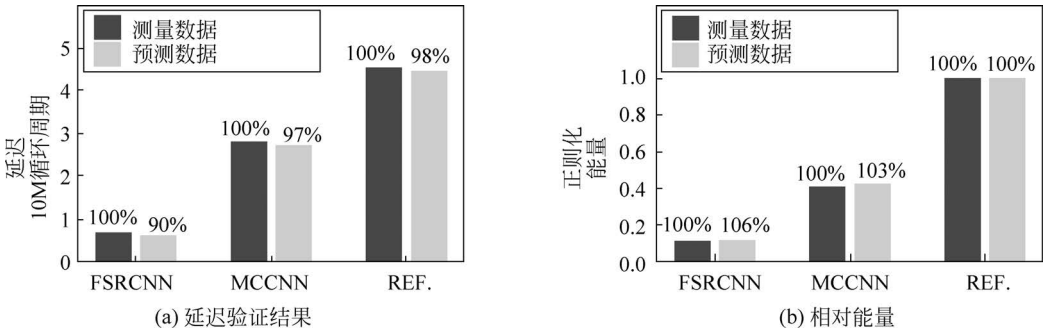


图 3-11 对 DeFiNES 的结果与 DepFiN 芯片的测量结果进行比较

首先，延迟的验证结果如图 3-11(a)所示，这表明 DeFiNES 的预测在第 2 网络和第 3 网络的 3%以内匹配。对于第 1 个网络 FSR-CNN，误差略高，仅为 10%。这是因为 DepFiN 的控制微处理器，由于 FSR-CNN 中发现的内核非常小，无法完全支持频繁的层切换，从而导致停滞。该控制流量限制未在 DeFiNES 中建模，其次，端到端匹配能源更具挑战性，因为它对几种精细的设计和布局非常敏感。

(1) 稀疏性：DepFiN 使用它来控制门控关闭逻辑活动以节省功耗。

(2) 地址和路线：导致数据传输要对比存储器读取/写入成本，并且还包括稀疏性相关效应。

(3) 工艺、电压和温度(PVT)变化。

尽管这些方面阻碍了准确预测绝对能源消耗，但是调度优化是相对建模精度最重要的能够选择的最佳选项。输出 3 个网络每次推理的相对能量如图 3-11(b)所示，归一化到参考网络的推理能量，以抵消排除 PVT 方面的影响，权重最大的是 MAC 的单位成本和每次访问能量中的 DeFiNES。

12. 5 种硬件架构

5 种硬件架构、DF-友好变体和(b)案例研究中使用的 5 种 DNN 工作负载，见表 3-1。

从表 3-1 中可以看到，当架构进行 DF 手动改造时，对于 Meta 原型 DF 处理器，本地缓存减少了权重的分配，增大了 I&O 的数量，并对 I&O 进行复用，增加 I&O 有助于融合的层次更深，支持更大一点的分块大小。对于类似 TPU 的 DF，减少了 Mac 组的寄存器的数目，增加了本地缓存中的 I&O；对于类似边缘 TPU 的 DF 处理器中，处理方式和 Meta 原型 DF 类似，即都减少了权重的 LB 的分配，增大了 I&O，可以用来在层间 I/O 传递。

13. Meta 原型 DF 架构和 FSR-CNN 作为工作负载

给定一个工作负载和架构，依据深度优先策略，使用 Meta 原型 DF 架构和 FSR-CNN 作为工作负载，如图 3-12 所示。

使用 FSR-CNN 与 Meta 原型 DF 分别作为目标工作负载和硬件架构。对于 3 个 DF 影响因子，包括分块大小、重叠存储模式、融合深度，其第 3 个轴融合深度固定在整个 DNN 上，因为 FSR-CNN 的总权重很小(15. 6KB)，因此所有权值都适合 Meta 原型 DF 架构的权重，可以全部放进片上的本地缓存(该架构的权重使用的本地缓存是 32KB)，因此不把整个 DNN 融合成一个栈是没有好处的。

表 3-1　10 种硬件架构及 5 种 DNN 工作负载

Idx	硬件架构	空间展开（1024 个 MAC）	每个 MAC 或 MAC 组的注册表信息	本地缓冲区	二级 LB	全局缓冲区（最大：2MB）
1	Meta-proto-like	K 32|C 2|OX 4|OY 4	W:1B;O:2B	W:64KB;I:32KB	/	W:1MB;I&O:1MB
2	Meta-proto-like DF			W:32KB;I&O:64KB	/	
3	TPU-like	K 32|C 32	W:128B;O:1KB	/	/	I&O:2MB
4	TPU-like DF		W:64B;O:1KB	I&O:64KB	/	W:1MB;I&O:1MB
5	Edge-TPU-like	K 8|C 8|OX 4|OY 4	W:1B;O:2B	W:32KB	/	I&O:2MB
6	Edge-TPU-like DF			W:16KB;I&O:16KB	/	W:1MB;I&O:1MB
7	Ascend-like	K 16|C 16|OX 2|OY 2	W:1B;O:2B	W:64KB;I:64KB;O:256K	/	W:1MB;I&O:1MB
8	Ascend-like DF			W:64KB;I&O:64KB	I&O:256KB	
9	Tesla-NPU-like	K 32|OX 8|OY 4	W:1B;O:4B	W:1KB;I:1KB	/	W:1MB;I&O:1MB
10	Tesla-NPU-like DF			W:1KB;I:1KB	W:64KB;I&O:64KB	W:1MB;I&O:896KB

（a）10 个硬件架构（5 个基线设计及其 DF-友好变体）

Idx	工作量	平均值/最大特征图	总权重
1	FSRCNN	10.9MB/28.5MB	15.6KB
2	DMCNN-VD	24.1MB/26.7MB	651.3KB
3	MCCNN	21.8MB/29.1MB	108.6KB
4	MobileNetV1	760KB/3.8MB	4MB
5	ResNet18	895KB/5.9MB	11MB

（b）5 种 DNN 工作负载

1. Meta-proto-like：Meta 原型样处理器
2. Meta-proto-like DF：Meta 原型样 DF 处理器
3. TPU-like：类似 TPU 处理器
4. TPU-like DF：类似 TPU 的 DF 处理器
5. Edge-TPU-like：边缘 TPU 风格处理器
6. Edge-TPU-like DF：类似边缘 TPU 的 DF 处理器
7. Ascend-like：类似华为升腾处理器
8. Ascend-like DF：类似华为升腾的 DF 处理器
9. Tesla-NPU-like：类似特斯拉 NPU 处理器
10. Tesla-NPU-like DF：类似特斯拉 NPU 的 DF 处理器

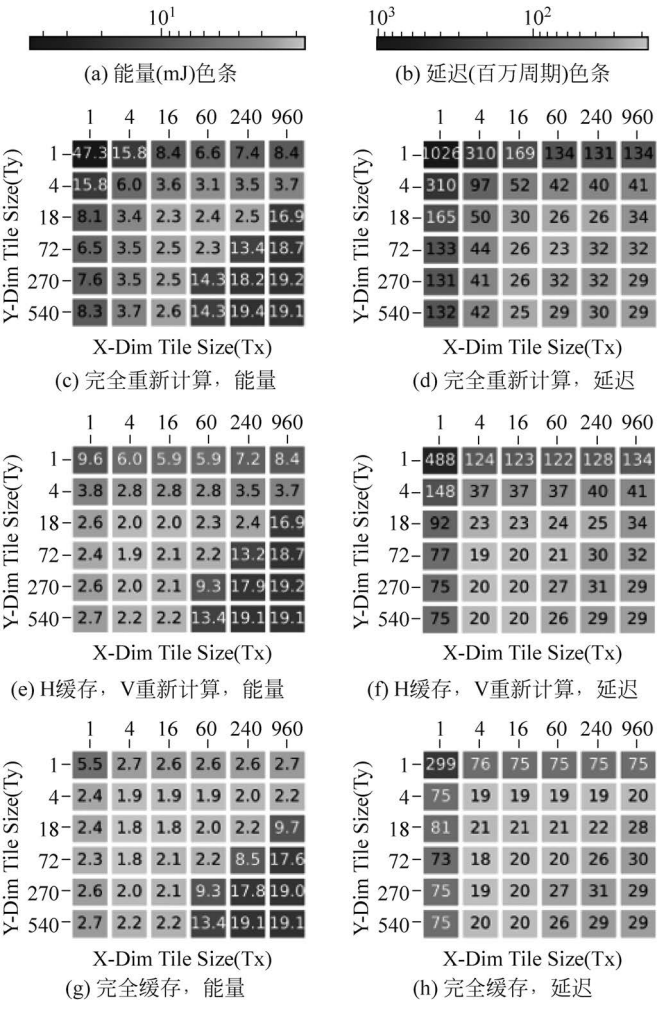


图 3-12　Meta 原型 DF 架构

不同计算模式下，它们的能量和延迟数(分别为 19.1 和 29)是相同的，因为此时的分块大小为 960×540，即全图，因此不存在分块，即已被转换为 LBL，因为不同的重叠存储模式对 LBL 没有影响。这种处理具有不同 DF 策略 FSR-CNN 的总能量和延迟。以下几个知识点需要说明：

(1) 考虑相同重叠存储模式下，即同一张图内比较，发现不同的分块尺寸，分块尺寸太小和太大都是次优的。分块尺寸过大会导致访问一些非常慢的存储层级，分块尺寸过小会导致大量访问权重，太大太小都不是最好的选择，最好是在中间分块尺寸。

(2) 考虑不同重叠存储模式下，在相同的分块大小的情况下，即对同一相对坐标下的步长进行比较，在大多数情况下，能耗顺序为完全缓存<(H 缓存，V 重新计算)<完全重新计算。这个也易于解释，适当的存储结构减少了大量的重复计算。

(3) 不同的分块大小和模式会严重影响能量和延迟。

(4) 完全重新计算比完全缓存,倾向于优先更大的分块大小。完全缓存倾向于小一些的分块。

14. 不同测向策略的 MAC 操作计数

分析器对角线对应的分块组合,其计算量如图 3-13 所示。图 3-13 和图 3-14 是从图 3-12 取出了所有对角调度特征点,并分别为贡献的层次结构(LB、GB 和 DRAM)中的每个内存级别,绘制了 MAC 操作计数和内存访问计数(以数据元素的数量为单位)。

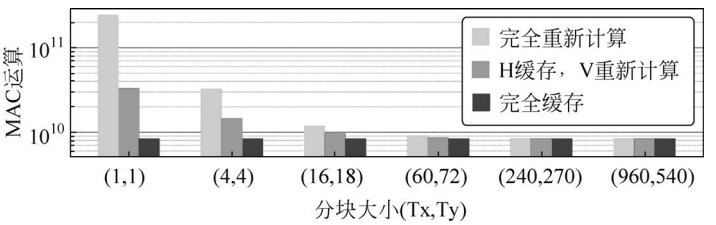


图 3-13　不同测向策略的 MAC 操作计数

从图 3-13 可以看到分块越小,计算量反而越大,因为数据复用比例较低,所以在完全重计算对应的 1×1 下,计算量最大。对于分块 1×1,顶层需要的计算为 7×7,而对于分块 2×2,需要的计算仅为 8×8,因此,分块大小并不是与计算量呈线性关系的,分块长和宽增大一倍,计算量并没有相应地增大,而是小于其增速。

因此,分块越小,计算量越大,对于 3 种存储模式都满足这个规律,只是完全缓存影响很小,统观全图,总的情况下满足完全重新计算>(H 缓存,V 重新计算)>完全缓存。随着分块大小的增大,高速缓存的数据(用于减少重复计算量)也会增加。通常为内核宽度－1 列(H 缓存,V 重新计算),或者(内核宽度×内核宽度－1)个(对应于完全缓存),但是相对于大的内核大小而言,缓存起来的数据占的比例在减小,因此,缓存数据的作用在降低,对于上一层输出的输出特征映射,比缓存数据更优先占用较快存储,随着分块大小的增大,例如,增加到 960×540 时,缓存也就没意义了(转换成了 LBL),因此,计算量也就一样了。

15. 不同内存级别的不同数据类型的内存访问

不同内存级别的不同数据类型的内存访问,用于元原型 DF 架构处理,具有不同 DF 策略的 FSR-CNN,如图 3-14 所示。

层的激活呈现了两个明显的趋势,DRAM 和 GB 访问不太依赖于所使用的模式,如图 3-14(a)所示。

对于层的权重,所有分块大小都具有相同的 DRAM 和 GB 访问,这是合理的,因为 FSR-CNN 的所有权重都可以适应权重 LB,如图 3-14(b)所示。

由数据复制操作引起的内存访问如图 3-14(c)所示,总的内存访问如图 3-14(d)所示。

这种结构用于具有不同 DF 策略的 FSR-CNN 的元原型 DF 架构处理。

16. 不同的工作负载导致不同的最佳解决方案

总的能量和延迟的可视化如图 3-15 所示,帮助读者更好地理解图 3-12 所示的热图。

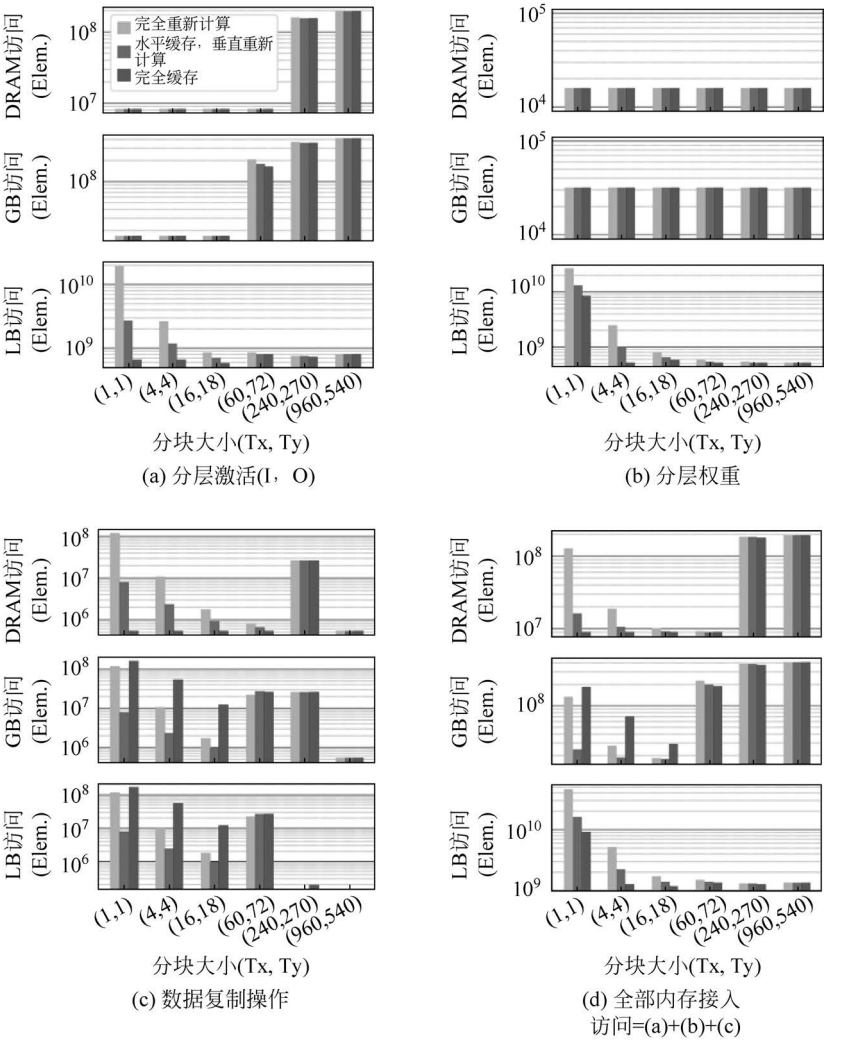


图 3-14　不同内存级别的不同数据类型的内存访问

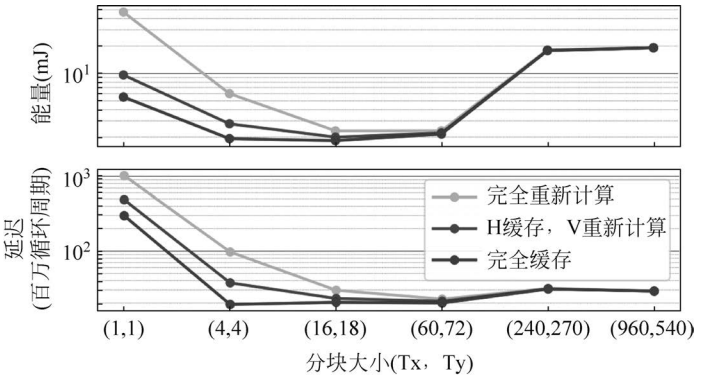


图 3-15　图 3-14 中设计点的总能量和延迟

不同的工作负载导致不同的最佳解决方案(在类似元原型的 DF 硬件上的所有结果),如图 3-16 所示。

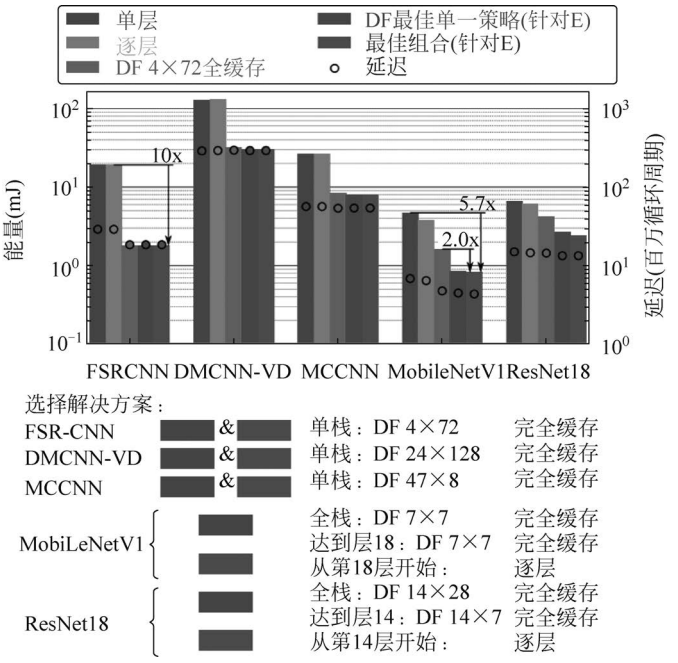


图 3-16　不同的工作负载导致不同的最佳解决方案(见彩插)

将深度优先应用于多个工作负载来分析其各自的性能,这里使用了如下的 5 种策略与 5 个网络,其中 FSR-CNN、DMCNN-VD、MCCNN 属于激活占主要的类的工作负载,而 MobiLeNetV1、ResNet18 则属于权重占主导的工作负载。

(1) 单层:一次对一个层进行完整评估,特征图始终存储到层之间的 DRAM 中,并从 DRAM 中提取。

(2) 逐层:逐层一次一个地被完全评估,这意味着没有分块,中间特征图被传递到它们所适应的最低内存级别的下一层。

(3) 完全缓存 d 采用 4×72 分块的 DF,这是案例研究中发现的最好的方案。

(4) 对所有融合层堆叠使用单一策略的最佳策略。

(5) 最佳组合,其中不同的栈,可以使用不同的 DF 策略。

这种解决方案在类似元原型的 DF 硬件上输出所有结果。FSR-CNN、DMCNN-VD、MCCNN 属于激活占主要的类的工作负载,这一类的特点是权重比较少,适合多层进行融合,权重通常多位于本地 SRAM 中,并且适合在进行层融合时栈的层次比较深。这 3 个网络的权重都在几十到几百千字节,适合融合深度与单个通道,因此对于单层是一种比较差的选择。这一类的另一个特点是激活占主要部分,适合对激活作分块处理。对于逐层也是一种比较差的选择,对应于 FSR-CNN 和 DMCNN-VD,MCCNN 使用单层/逐层时效果都比

较差。相应地，这几种网络更适合选择完全缓存的分块，其多层融合的单一栈，对应于这3种网络，绿色DF 4×72完全缓存、红色DF最佳单一策略和紫色的最佳组合的效果都比较好。

17. 逐层应用或最佳DF调度策略

现在来看单一策略对不同网络的影响，单层对应图3-16中的蓝色条，即每层作为一个栈；逐层对应黄色条，无论哪种网络，其能量和延迟都比较高。因为其前者需要访问最慢的DRAM，后者无法分块，也需要访问较慢的存储结构。再来看DF最佳单一策略和最佳组合的两种情况下（红色条和紫色条），对前3个网络都选择了比较合适的单一栈，即所有的层总体融合一个单一栈，其分块选取了较为合适的4×72/24×128/47×8。数据存储都选择了数据存储性能为完全重新计算> H缓存，V重新计算>完全缓存。这3个网络都取得了不错的延迟和能量，绿色框DF分块采取相互适配的大小4×72，4×72模式也是对激活类型占优的网络找到的最优解，数据存储选取为完全缓存d。对于FSR-CNN、DMCNN-VD、MCCNN，其DF 4×72完全缓存，DF为最佳单一策略，最佳组合占优，而单层/逐层不占优。

对于MobiLeNetV1、ResNet18这样的模型文件，则属于权重占主导的工作负载，这一类的特点是激活相对比较少，因此，在红色的DF最佳单一策略和紫色的最佳组合中，栈采取了不同的方式。以MobiLeNetV1为例子，DF最佳单个策略（一个单一的策略用于所有的融合层栈）中每个栈选取了7×7的分块大小，数据存储为完全缓存d，而对于组合模式（不同的栈可以使用不同的DF策略），则不同层使用不同的模式，前面18层使用一个固定的分块大小，使用完全缓存，而对于后面的层，以MobiLeNetV1为例子，而MobiLeNetV1和ResNet18的权重占主导地位（特征图较小，并在各层之间逐渐减少），在18层后，特征映射已经变得比较小，即不再对特征映射进行分块处理，所以使用了逐层，总体来讲，更细粒度调度策略的紫色的最佳组合的效果最好。

通过逐层应用或最佳DF调度策略，产生不同硬件架构的能量和延迟（5个工作负载的几何平均值）如图3-17所示。

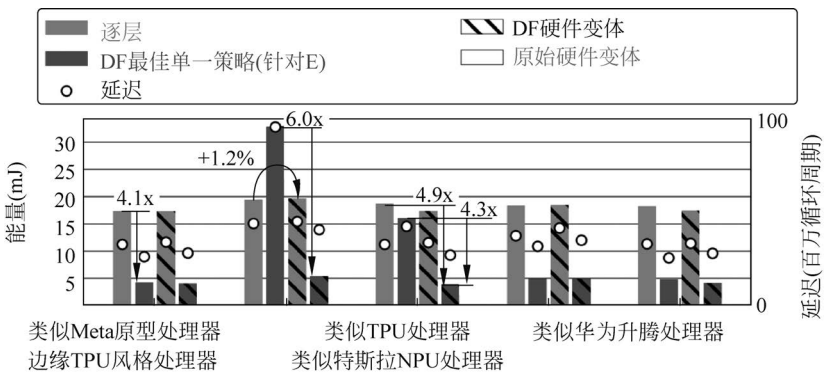


图3-17 不同硬件架构的能量和延迟（5个工作负载的几何平均值）（见彩插）

除了类 TPU，DF 在所有加速器架构上都优于 LBL，包括未调整的默认加速器，其最大增益为 4.1。由于缺乏片上权重缓冲器，类 TPU 对 DF 调度的支持较差。在 DF-友好变体中，添加了这样的缓冲区，DF 显著优于 LBL，这表明了在设计时考虑 DF 兼容性的重要性。DF-友好和默认型变体之间的总体比较进一步支持了这一发现，这表明 DF-友好变体在使用 DF 时，至少与默认值一样好，类 TPU 和类边缘 TPU 硬件的大增益，分别为 6.0 和 4.3，使用 LBL 时最大误差为 1.2%。

18. 相关 DF 建模框架比较与评估

相关 DF 建模框架比较如图 3-18 所示。

DF建模框架	重叠存储模式 ①	②	③	芯片数据流量模型	支持多级存储跳转	模型权重流量	优化目标
DNNVM	✗	✓	✗	✓	✗	✓	La
Efficient-S	✓	✗	✗	✓	✗	✗	La
LBDF	✓	✗	✓	✗	✗	✗	DRAM
ConvFusion	✓	✗	✓	✗	✗	✓	DRAM
Optimus	✓	✗	✓	✗	✗	✓	DRAM
DNNFuser	✓	✗	✗	✓	✗	✓	DRAM, Mem
DeFiNES(自研)	✓	✓	✓	✓	✓	✓	**En, La**
可视化每个因素的影响	图与案例研究						

1　重叠存储模式(✓ 支持/✗ 无支持)
①完全重新计算；②H缓存，V重新计算；③完全缓存。

2 ✓ 芯片上数据流量模型　⇔ ✗ 仅限型号/优化。DRAM存取。
3 ✓ 支持多级存储跳转　⇔ ✗ 仅支持DRAM跳转。
4 ✓ 模型权重流量　⇔ ✗ 仅限型号/优化，激活流量。

5　优化目标：DRAM：DRAM访问；Mem：片上存储器用法；
La：总延迟；En：总能量。

图 3-18　相关 DF 建模框架比较

前文已经讲解了几种 DF 建模和探索框架，如 DNNVM、Efficient-S、LBDF、ConvFusion、Optimus 和 DNNFuser。图 3-18 所示的这些框架，有助于在给定硬件架构和 DNN 工作负载的情况下，对 DF 调度进行建模和优化。在优化部分，引入了许多创新的搜索算法，如 DNNVM 中的启发式子图同构算法、基于 DAG 的搜索算法。

硬件优化中的感知算子融合算法，以及 DNNFuser 中的基于转换器的映射器，在建模部分都缺少一些重要因素。

图 3-18 中不同因素的实验如图 3-19 所示。将 FSR-CNN 映射到两个硬件平台上的实验结果如图 3-19(a)所示。只有在考虑整个系统时，才能获得最佳测向解决方案(橙色条)。所找到的解决方案的参数显示，当对整体能量(橙色)进行优化时，与仅对 DRAM(红色)进行优化相比，该框架发现了更小的瓦片大小。

测试的工作负载硬件组合，观察到 17%～18%的能量增益，如图 3-19(b)所示。这可能对整个系统的效率有害，如图 3-19(c)所示。在 SL 情况下，由激活的内存访问引起的能量

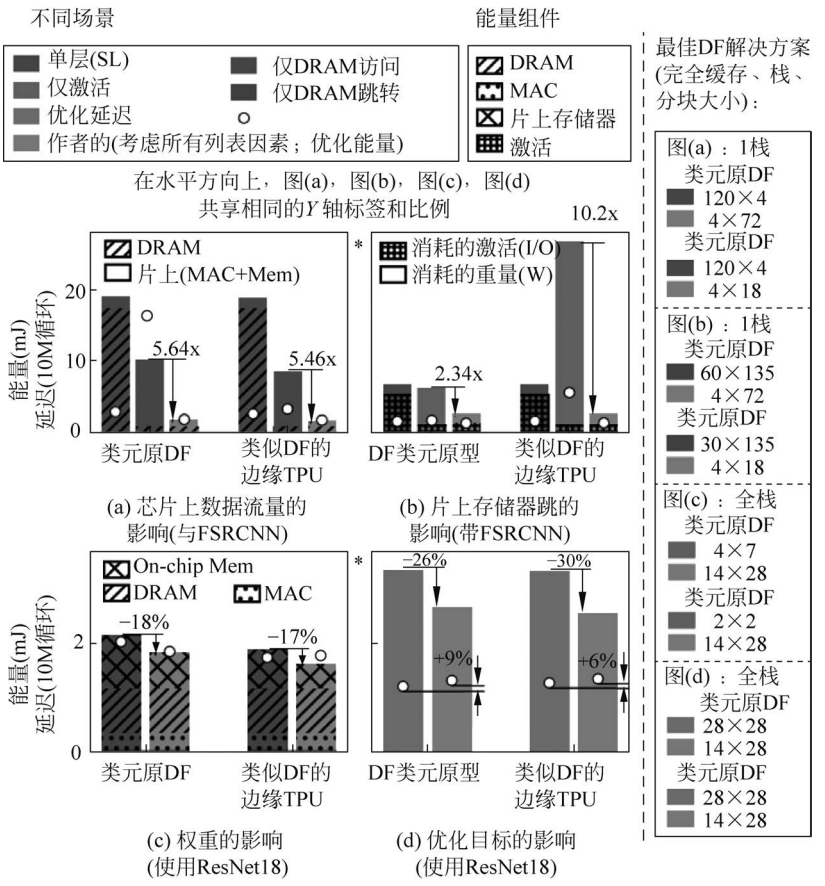


图 3-19　评估表不同因素的实验(见彩插)

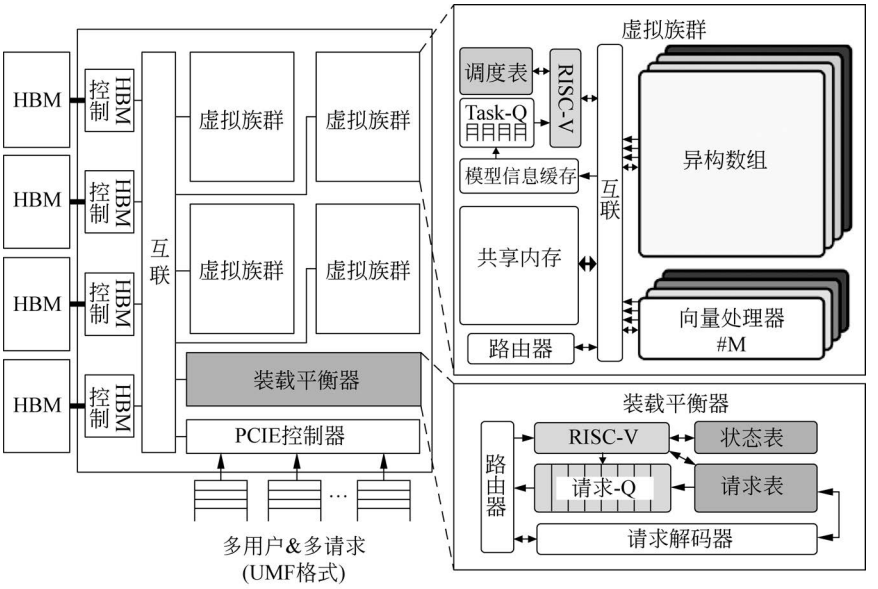
部分(用方形阴影突出显示)，贡献了大部分能量。粉色/橙色条分别是延迟/能量优化 DF 调度的能量(对应的点是延迟)，如图 3-19(d)所示。

3.2　异构系统：向量体系结构

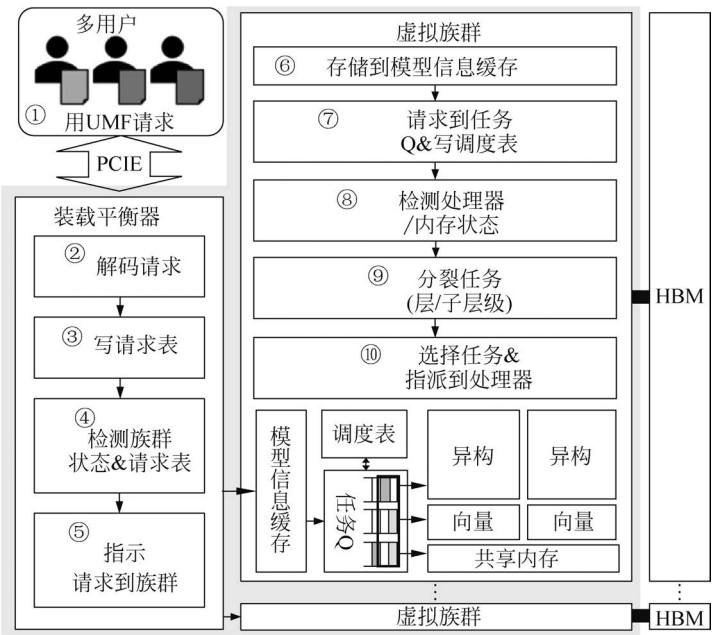
3.2.1　异构稀疏向量加速器的总体架构

异构稀疏向量(HSV)加速器的总体架构如图 3-20(a)所示，包括负载均衡器、稀疏向量(SV)集群、高带宽存储器(HBM)控制器和片上互连。在这个体系结构中，主机 CPU 对传入的机器学习进行编码。

将来自多个用户的推理请求转换为 UMF 分组，并通过 PCI Express(PCIE)将其发送到加速器，然后由负载均衡器对 UMF 分组进行解码，并向可用的 SV 集群执行高级工作负载分配。每个 SV 集群独立地执行分配的机器学习推理请求，并在完成任何一个请求时向负载均衡器发送信号。负载均衡器和 SV 集群可以通过完全连接的互连访问外部 HBM。



(a) 异构架构框图



(b) 用户请求处理和调度流程

图 3-20　异构架构框图与用户请求处理和调度流程

HSV 架构的这些组件被设计为遵循用户请求处理流程，并为数据中心中多个 DNN-ML 工作负载的多用户请求提供高效的硬件。

在 HSV 加速器中，用户请求的整体处理流程如图 3-20(b)所示。

(1) 主机 CPU 接收来自用户的各种 ML 推理工作负载，并通过 PCIE 将其发送到 UMF 中的加速器。

(2) 负载均衡器中的 UMF 解码器，对 UMF 报头进行解码以获得用户描述。

(3) 将模型写入请求表。

(4) 检查每个 SV 集群状态和请求表。

(5) 负载均衡器将 ML 推理请求分配给可用的 SV 集群并更新状态表。

(6) 一旦分配的请求进入 SV 集群，就被解释为分层任务，并存储在模型信息缓冲器中。该缓冲器存储着 ML 模型信息，该信息用于估计执行时间、外部存储器访问、需要获取的数据大小。

(7) 分层任务排队到任务队列，并将它们的信息(例如，估计值、事务 ID 等)写入调度表。该表包括用于调度的信息，例如层依赖性和每个处理器的状态。

(8) 检查底层处理器和存储器的状态和可用资源。

(9) SV 集群中的调度器可以将请求划分为子层任务。

(10) 任务队列中的这些任务被选择并分配给处理器，以最大限度地利用计算资源和外部存储器带宽。

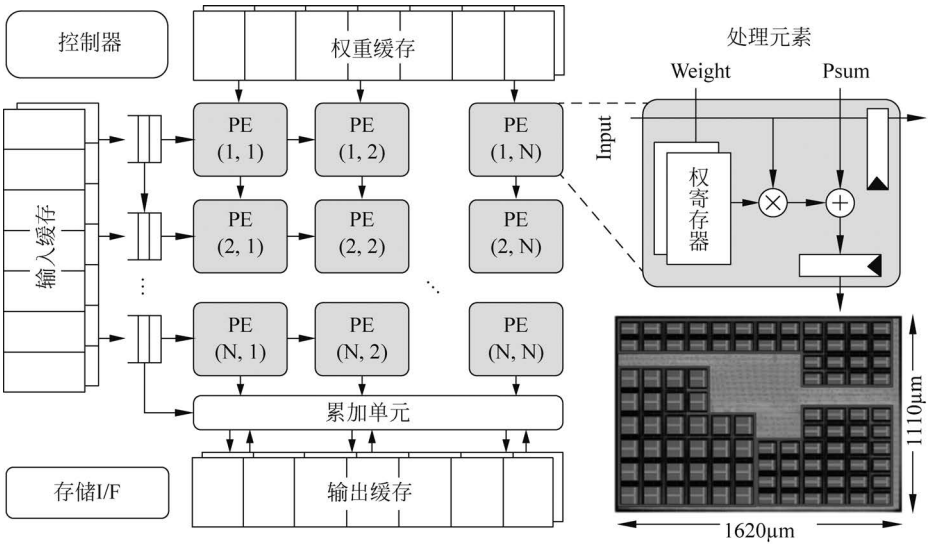
3.2.2 稀疏矩阵

稀疏矩阵具有典型的二维稀疏设计，可实现高通量矩阵乘法和三维卷积运算，由控制器、处理元件(PE)的二维阵列，以及用于输入、权重和输出数据的片上缓冲器组成，如图 3-21(a)所示。控制器负责处理器中的整体数据移动，为输入和权重缓冲器生成地址，以将数据馈送到 PE 阵列。使用权重固定数据流，它将权重从缓冲器预加载到 PE。对于矩阵乘法，权重矩阵的每行被映射到 PE 阵列的每列。对于三维卷积运算，每个三维权重核被展平，并映射到 PE 阵列的每列。通过这种权重映射，输入向量的元素从左起穿过 PE 阵列的多行进行馈送。

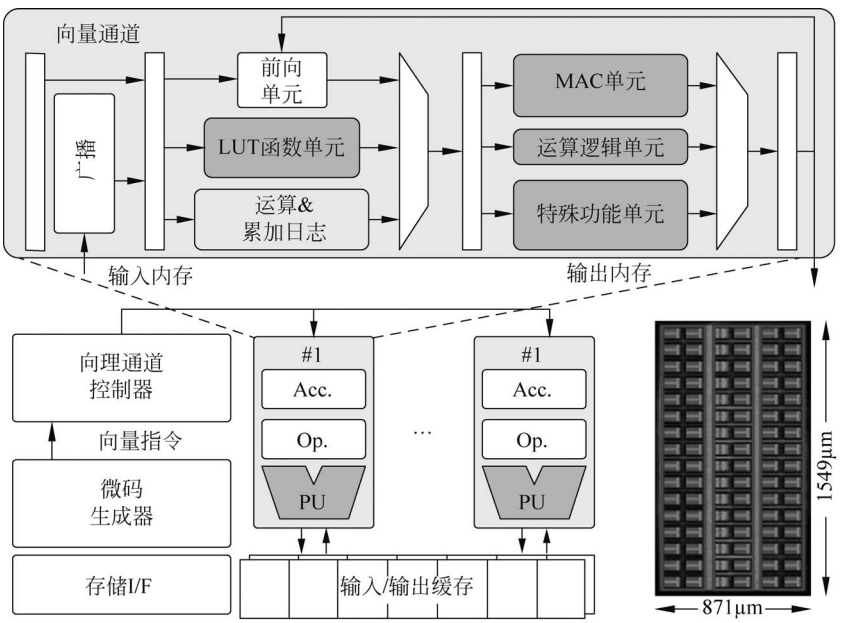
从最上面一行开始，每个元素都有一个周期延迟。在每个 PE 中，输入与存储的权重相乘，并与上述 PE 的部分和相加。在单周期延迟的情况下，部分和不断地被累积到阵列的底部。累加单元存储来自阵列的中间部分，并通过用于大矩阵/卷积运算的多次迭代来累加它们。将最终结果存储到输出缓冲器中。为了提高 PE 阵列的利用率，对 PE 的片上存储器和内部寄存器都使用了双重缓冲。输入缓冲器预取下一个输入数据，输出缓冲器写入先前的结果，以保持 PE 阵列在不停滞的情况下运行。同样，每个 PE 在处理当前权重时，加载后续权重。通过交替读取寄存器，它可以无缝地利用 MAC 单元。

在 28nm CMOS 工艺中实现了稀疏矩阵。对于具有一对 16×2KB 输入/权重和 16×4KB 输出缓冲器的 16×16 PE 阵列，稀疏矩阵占用 1620μm×1110μm 的芯片面积，经过后期放置和路线模拟，可达到 800MHz 的工作频率。

向量处理器的框图和布局，如图 3-21(b)所示。它是一个具有多个向量通道的有序单



(a) 稀疏矩阵



(b) 向量处理器

图 3-21　使用 28nm 标准单元格库的稀疏矩阵与向量处理器

指令多数据(SIMD)处理器。它主要用于在 DNN 模型中运行非矩阵运算，如池化、非线性激活和元素向量运算。由于向量处理器还可以通过程序运行矩阵乘法或三维卷积运算，因此在任务调度中提供了更多选项。向量处理器由微码生成器、向量通道控制器、多个向量通道和用于直接存储器存取(DMA)的输入/输出缓冲器组成。

3.2.3 示例：异构感知调度算法

异构感知调度示例显示了单个 SV 集群中的调度场景，如图 3-22 所示。方框的颜色表示每个请求，方框中的数字表示请求中任务的顺序。橙色图案框表示处理器不执行任何计算的空闲时间。上面的调度显示了 RR 的调度结果，下面的调度显示了无法获取请求 2 的任务，并且在现有场景中，向量处理器等待请求 3 的第 3 个任务结束。在 HAS 中，通过将请求 3 的第 3 个任务划分为子任务，减少了每个子任务的内存容量需求。每当子任务完成时都会从共享内存中刷新参数，以减少片上内存资源开销。通过将请求 3 的第 4 个任务，分配给向量处理器，而不是稀疏矩阵，HAS 减少了稀疏矩阵的计算负载和总计算时间。

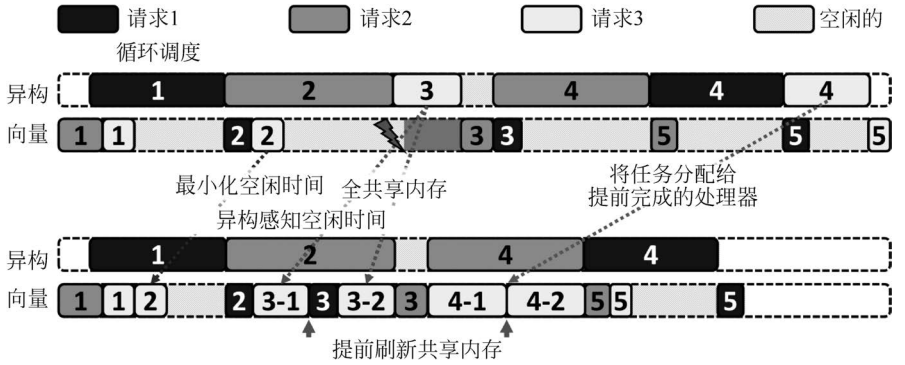


图 3-22 异构感知调度算法示例(见彩插)

3.2.4 外部内存访问调度

外部存储器访问调度算法在任务到达时，基于共享存储器的动态分析，调度从外部存储器读取和写入参数与激活。为了调度外部存储器访问，调度器参考调度表 S 和候选任务 T(需要服务的任务)，用以计算参数和激活函数的准备时间 T。首先，调度器计算参数 p 的数据大小，并使用候选任务 T 的信息激活 a 大小，然后它获得最后一次从外部存储器 T 提取它的时间。如果所需的参数是预先调度的，并且存在于共享内存中，则处理器使用该值，而无须进行不必要的外部内存访问。

外部内存访问调度算法的伪代码如下：

```
//第 3 章/exter mem sched algo.c
算法 2: 外部内存访问调度
输入: 调度表 S,候选任务 T
输出: 激活和参数准备时间 t
p 大小←getParamSize(T)
a 大小←getActSize(T)
t 存储←getLastMemFetchEndTime(S)
如果参数存在于共享存储器中,则
    t, F←getParamReadyTime(S, T), 0
否则
```

```
    t, F ← t 存储, p 大小
如果激活需要从外部存储器读取,则
    t←t 存储
如果 t = t 存储,则
    R←SM 大小 - getUsedMemSize(S)
    fetchParam(T, t, R, F)
    如果未完成 FetchParam(F),则
        对于调度 T ∈ S,do
            如果 t ≤ getEndTime(sched T),则
                updateTime(sched T, t)
                如果激活需要写入外部存储器,则
                    t ← t + getActWriteTime(sched T)
                如果没有任务使用调度 T 的参数,则
                    flushSM(sched T, R)
                fetchParam(T, t, R, F)
                如果 finishFetchParam(F),则
                    break
            end
        t ← t + getActReadTime(a size)
```

3.2.5　仿真框架

为了评估所提出的可扩展异构体系结构和调度算法，用 Python 构建了一个仿真框架，其中包括 DNN 模型描述转换器、循环模拟器、性能分析器和时间线可视化器。使用仿真框架进行了设计空间探索，以找出数据中心 DNN 工作负载的最佳配置和调度算法。模拟框架的高级概述，如图 3-23 所示。仿真框架采用 4 种不同的信息作为输入：ONNX 格式的 DNN 模型描述、调度算法、硬件架构配置和物理硬件信息（频域、面积和功率）。

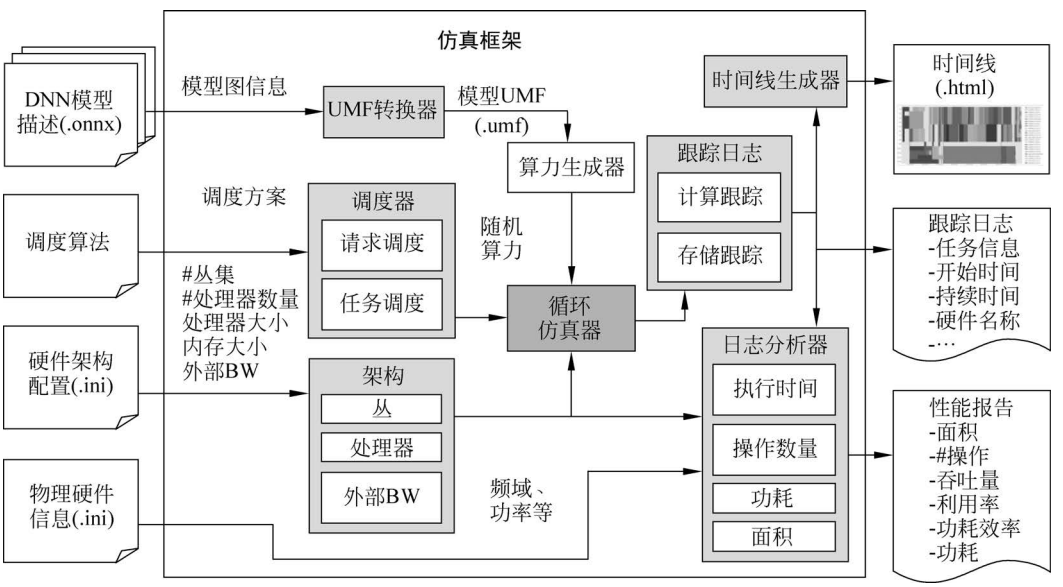


图 3-23　仿真框架概述

（1）UMF 转换器将模型信息从 ONNX 格式转换为 UMF 格式。

（2）调度算法和硬件配置文件确定调度方案和硬件架构（集群的数量、处理器的数量和大小、共享内存的大小等）。

（3）循环模拟器评估架构和算法，并将计算和内存跟踪记录在日志中。

（4）该框架基于跟踪日志和物理硬件信息，生成时间轴可视化和性能报告。

3.2.6 位片跳转架构与数据管理方面的硬件挑战

零跳转方法需要一个零数据跳转单元，该单元在没有任何延迟的情况下，预取非零输入并加载相应的权重，然而，如图 3-24 所示，由于与 8 位固定位宽 MAC 单元相比，4 位位片结构采用了×4 个 MAC 单元，因此需要×4 个零数据跳转单元，这增加了面积和功耗。在数据管理中，稀疏数据压缩通过将稀疏数据编码为非零数据及其索引，用来减少数据事务的数量和内存占用，然而，在对数据进行分解后，非零索引的位宽相对于非零数据变大，这将使位宽减半并使数字加倍。结果使压缩比低于固定位宽数据。

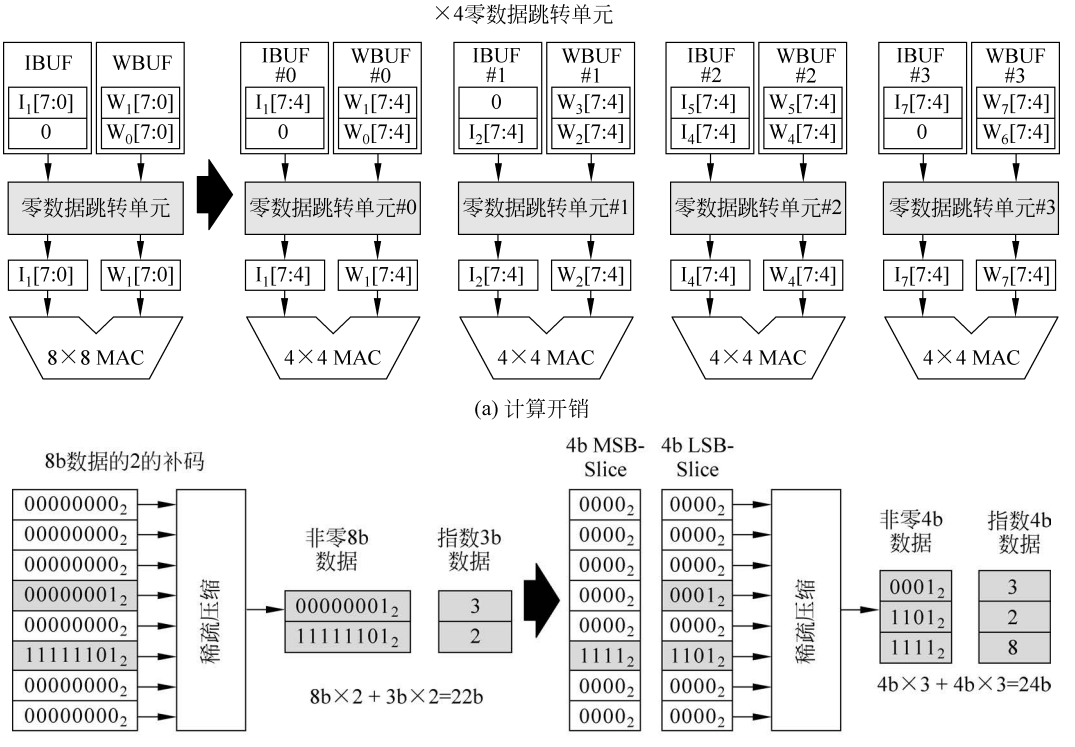
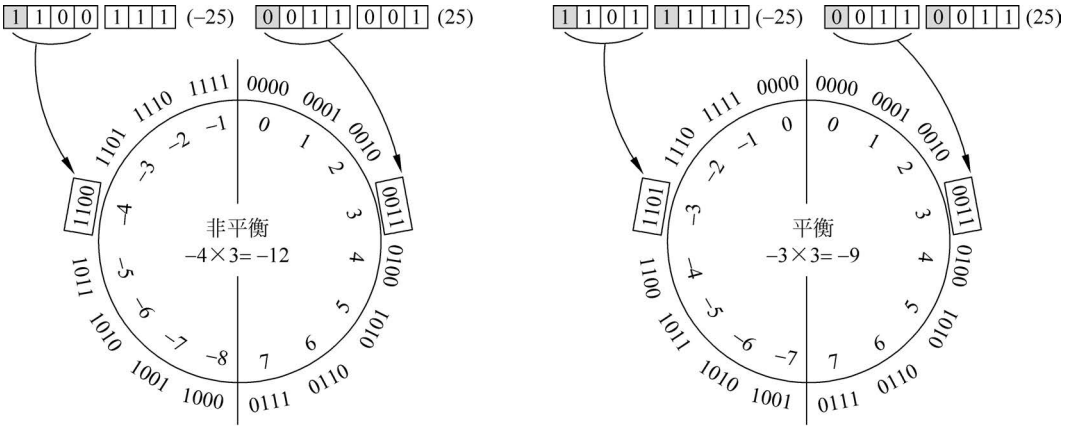


图 3-24 位片跳转架构

输出跳转架构不是利用稀疏的输入和权重数据，而是利用最大池化操作所呈现的输出稀疏性，如图 3-25 所示。特别地，基于三维点云的 DNN，利用大规模（例如 64-到-1）最大池化操作，用来提取最大特征，因此可以通过利用输出稀疏性来去除大量卷积操作，因此，输出跳

转架构，通过高位片的预计算来推测这些冗余计算，并跳转它们以实现高能效，然而，由于补数在正和负之间的不平衡，推测通常是失败的。例如，通过传统的位片分解，1100111_2(−25)和0011001_2(25)的位片的高阶，分别为1100_2(−4)和0011_2(3)。那么，(−25)×(−25)等于9。

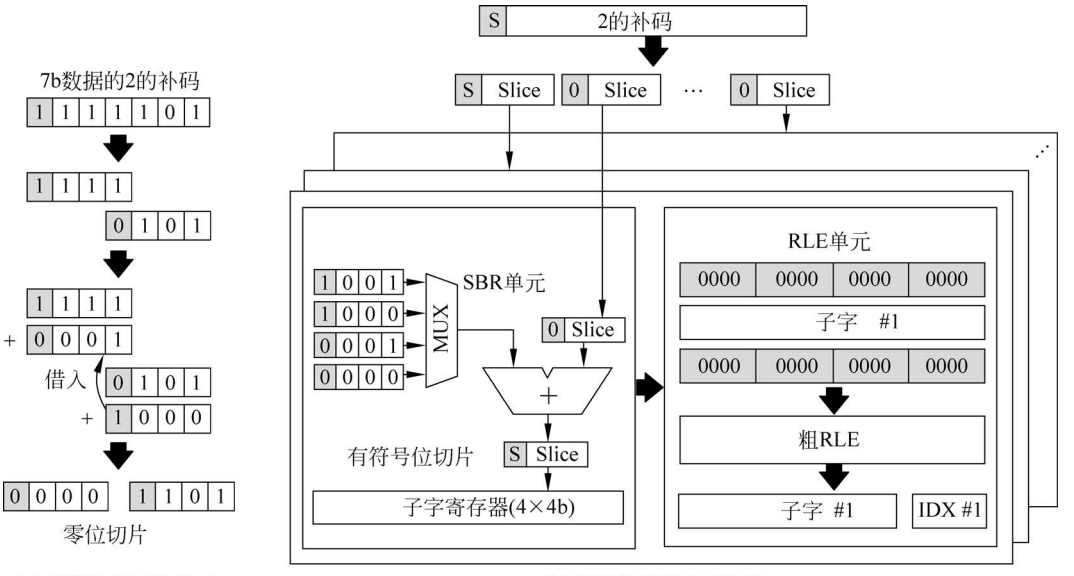


(a) 转换方法　　(b) 有符号位切片法

图 3-25　正数和负数位片中的不平衡及带符号位片的平衡

3.2.7　有符号位片表示及其编码单元

传统的位片表示，将2的补码定点数据，分解为作为有符号位片的MSB位片和作为无符号位片(Unsigned Bit slice)的较低片。在这项工作中，SBR将符号位添加到每个无符号位片，用以产生有符号位片，并且如果数据为负值，则SBR通过从其低位片借用来添加1个值，如图3-26(a)所示。例如，2的补码数据的1111101_2被分解为有符号位片的1111_2和无



(a) 有符号位切片表示　　(b) SBR单元和RLE单元

图 3-26　有符号比特切片的概念

符号位片 101_2，然后 SBR 通过 0101_2 支持，使 101_2 变为 0101_2，并将 1 加为 1111_2。最后，它们变成 0000_2 和 1101_2。结果，SBR 使大部分 1111_2 位片在小负值处变为 0000_2 位片，并且有符号位片的稀疏性显著增加。

SBR 单元和游程长度编码(RLE)单元，如图 3-26(b)所示。SBR 单元在标记位片结构中的最大矩阵处理(MPU)核处理之前，对数据执行 SBR。它接收定点数据的位宽，并通过考虑位片的顺序和数据的符号来选择要相加的值。例如，位片的中间阶，可以从位片的较低阶借用 12，并将 1000_2 借给位片的较高阶。另外，MSB 位片仅借用 12，LSB 位片仅借出 1000_2。编码比特片的每个顺序被收集到子字(16b)寄存器。在收集了 4 个 4 位有符号位片之后，如果它们都为 0，则将它们发送到 RLE 单元并进行压缩，然后只有非零子字数据被传输到 MPU 核心。

3.2.8 用于输入和输出跳转的零数据跳转单元

带有零数据跳转单元的 PE 单元的数据路径，如图 3-27 所示，即使在非 ReLU 激活函数中，SBR 也会生成大量的 4 位零位切片，并且可以去除大量的计算。为了最小化细粒度零位片跳转单元的开销，有符号位片架构将 4 个空间上相邻的 4 位输入位片处理为子字数据，并且跳转零子字数据。PE 从输入缓冲器(IBUF)获取非零子字数据，并从索引缓冲器(IDXBUF)中获取其 RLE 索引。IBUF 将输入的子字数据广播到 4 个 MAC 阵列，并且 MAC 阵列通过将子字数据分割到 4 个 4 比特切片来将子字分配给 4 个带符号的 MAC 单元。同时，MAC 阵列通过零数据跳转单元处的 RLE 索引，计算权重缓冲器(WBUF)的下一个地址，用来加载相应的权重位片。

3.2.9 片上异构网络

采用异构 NoC 进行 DNN 工作负载的高效分配，如图 3-27 所示。对于输入、权重和输出数据传输，使用双向 Bi-NoC。数据管理单元(DMU)核心将输入和权重数据提供给矩阵处理单元(MPU)核心，并且 MPU 核心通过 Bi-NoC 将卷积输出传输给 DMU 核心。Bi-NoC 还灵活地将权重数据传输到 IBUF，并将输入数据传输到 WBUF，用以进行混合跳转。在通过路由器接收到数据后，NoC 进行交换机单播、多播，并将数据广播到具有数据可重用性的 PE 阵列。

例如，输入比特片 I[3:0]被多播到 PE#0 和 PE#1，权重比特片 W[3:0]，则被多播给 PE#1 和 PE#3，如图 3-28(a)所示，然后通过分配不同的权重输出通道，输入位片 I[3:0]被复用到图中的 4 个 PE，如图 3-28(b)所示。权重位片 W[3:0]被广播到 3 个 PE 阵列，用以计算图中输入位片的不同阶数，如图 3-28(c)所示。通过单播图中 3×3 权重的不同空间权重，将输入数据与 3 个 PE 阵列共享，如图 3-28(d)所示，因此，在有符号位片结构中利用了工作负载分配的各种组合，通过 Bi-NoC 实现了数据的可重用性。

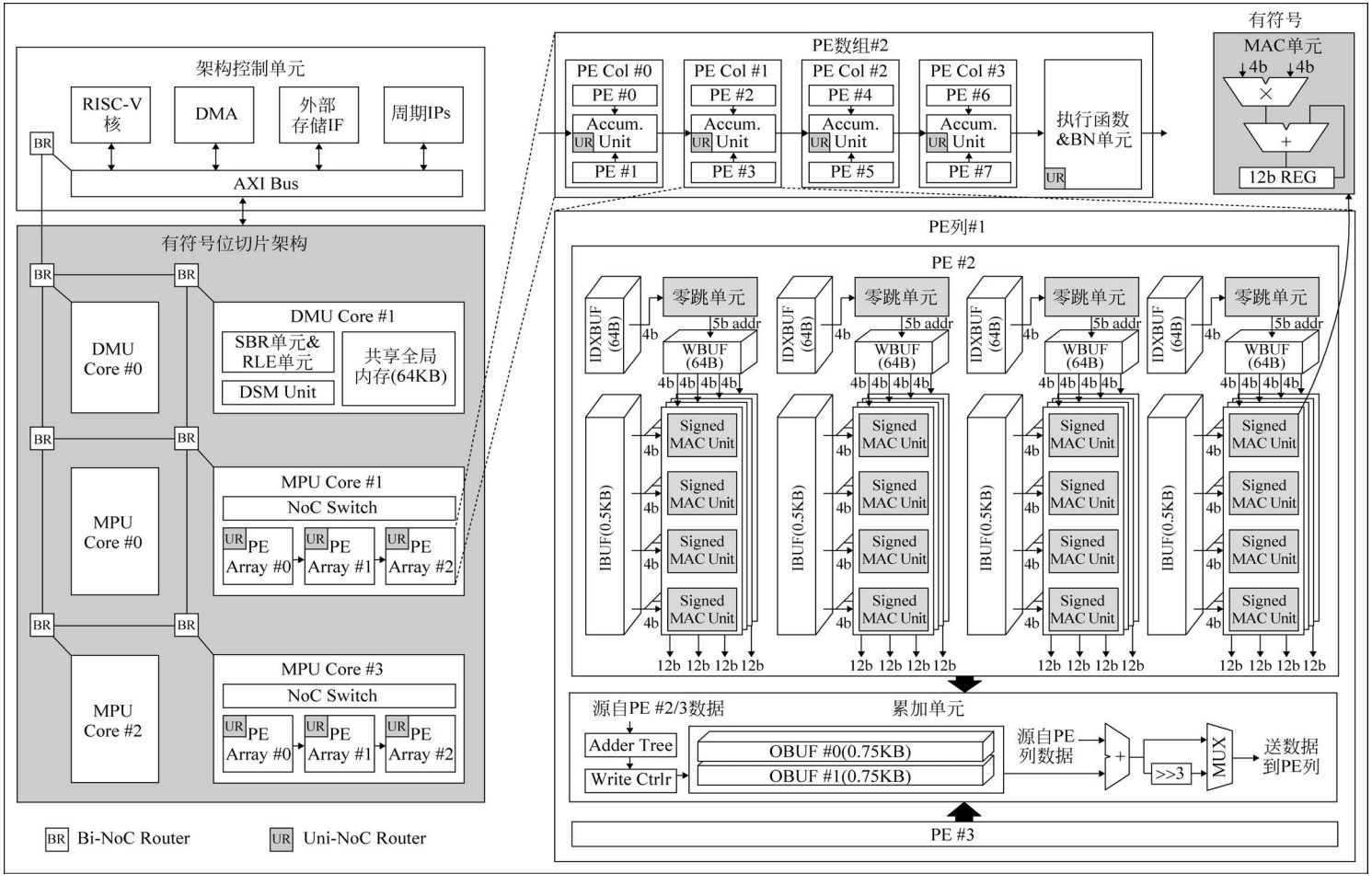


图 3-27　带符号位片结构的总体结构及其控制单元

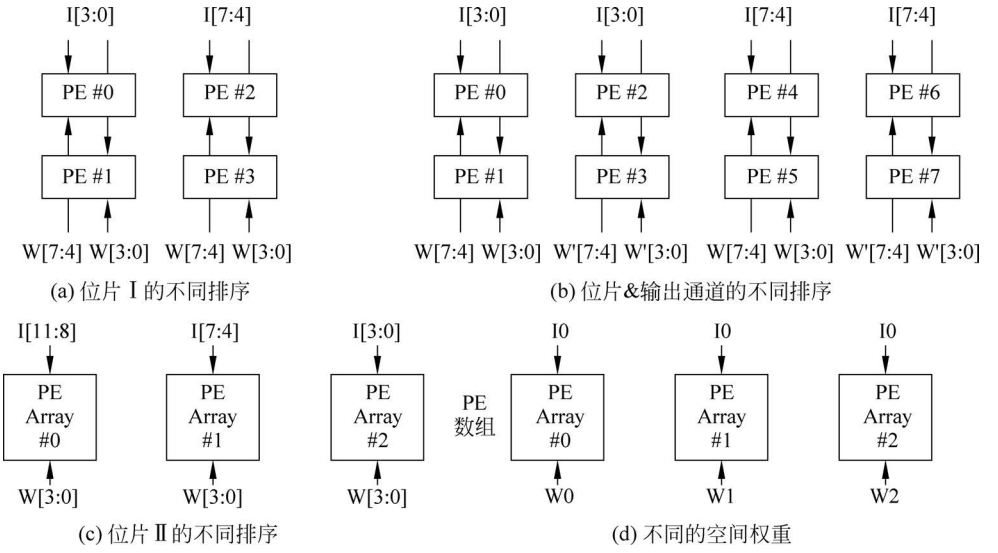


图 3-28　利用数据可重用性的工作负载分配

3.2.10　指令集体系结构

CPU 的指令(例如 RISC-V ISA)与由大量迭代 MAC 操作组成的 DNN 执行不兼容。由于 CPU 忙于控制移动 SoC 中的其他硬件单元(例如 GPU、I/O),因此必须以 CPU 的最少参与来控制 NPU,因此,提供了用于有符号位片结构的专用 ISA 和分层指令解码器,如图 3-29 所示。带符号位片结构从 RISC-V 核心(1)获取指令,顶部指令解码器读取指令的高位 7 位,并将操作码和操作数发送到 DMU 核心,或 MPU 核心的目标地址(2),然后由目标单元内的指令解码器对剩余指令的 4 位操作码和 16 位操作数进行解码,并配置和激活内核(3)。

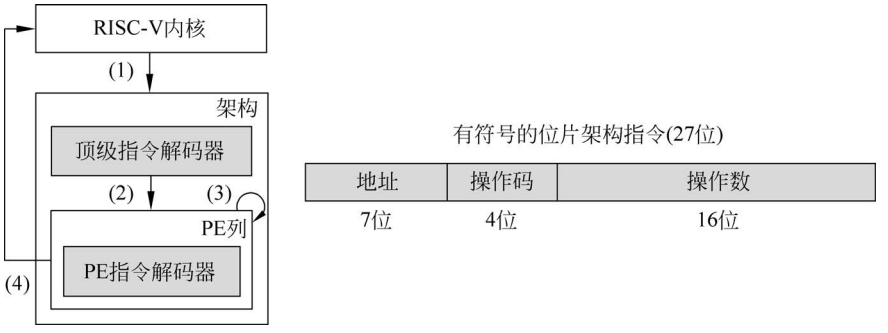


图 3-29　有符号位片结构 ISA 和分层指令解码器

3.2.11　广义深度学习的架构式编排、变换和布局

Violet(紫罗兰)芯片在逻辑上由并行处理元件、互联网络、存储系统 3 个主要组件组

成，如图 3-30 和图 3-31 所示。

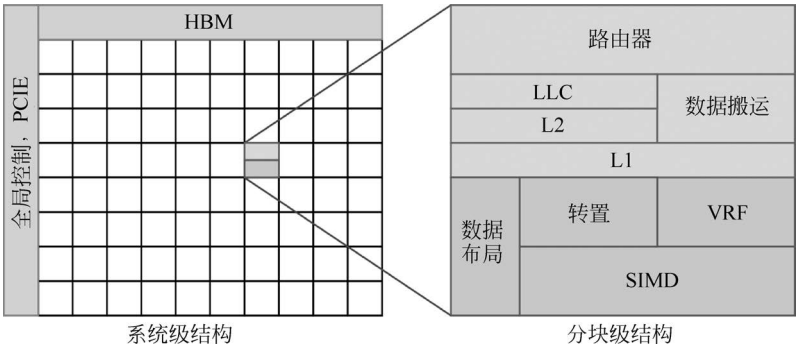


图 3-30　Violet 芯片架构概述

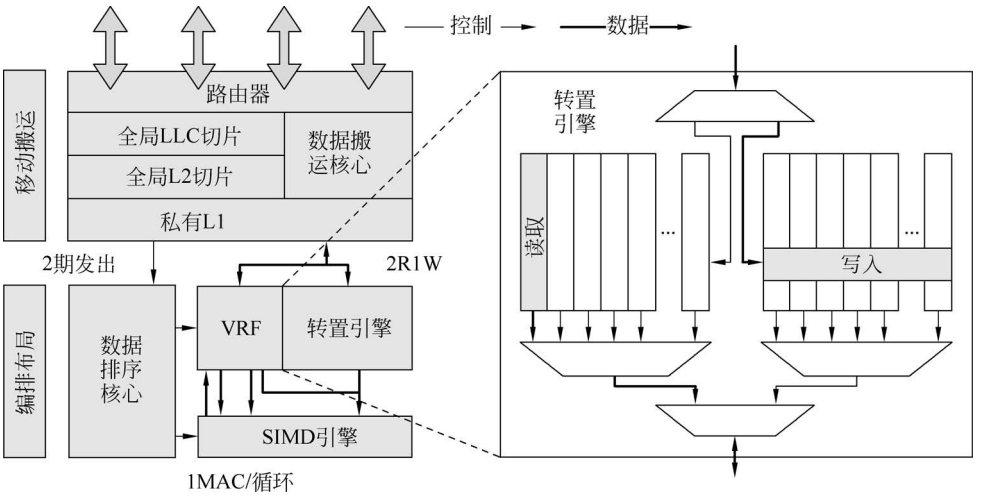


图 3-31　紫色分块的组织（见彩插）

在物理上，Violet 被划分为相同的分块，每个分块都包含一个数据编排核心，该核心与一个宽 SIMD 短向量数据路径耦合，该数据路径包括寄存器文件和组织为通道的算术单元。分块还包含与数据移动引擎相结合的 2D 网格 NoC 上的分布式存储器层次结构的切片。发现 ISA 的机制并不重要。该系统包括全局线程调度器，基于软件定义的工作位置，将工作传输到核心。它还包括主机接口控制器（类 PCIE 接口），以向运行 DL 栈的系统级部分的通用主机计算机提供高带宽、低时延通信。

最后，一个或多个存储器控制器和片上 PHY（从实现的角度来看，最好是 HBM）为 LLC 供电。LLC 的物理组织是直接的：通过静态地址映射分布在芯片上的切片。2D 网格互联网络在分块之间传输高速缓存线，并将高速缓存线传输到存储器和从存储器传输高速缓存行。

3.3 示例：NPU 开发

3.3.1 NPU 硬件概述

NPU 处理器专门为物联网人工智能而设计，用于加速神经网络的运算，解决传统芯片在神经网络运算时效率低下的问题。

在 GX8010 中，CPU 和 MCU 各有一个 NPU，MCU 中的 NPU 相对较小，习惯上称为 SNPU。NPU 处理器包括乘加、激活函数、二维数据运算、解压缩等模块。乘加模块用于计算矩阵乘加、卷积、点乘等功能，NPU 内部有 64 个 MAC，而 SNPU 有 32 个。激活函数模块采用最高 12 阶参数拟合的方式实现神经网络中的激活函数，NPU 内部有 6 个 MAC，而 SNPU 有 3 个。二维数据运算模块用于实现对一个平面的运算，如降采样、平面数据复制等，NPU 内部有一个 MAC，而 SNPU 有一个。

解压缩模块用于对权重数据进行解压。为了解决物联网设备中内存带宽小的特点，在 NPU 编译器中会对神经网络中的权重进行压缩，在几乎不影响精度的情况下，可以实现 6～10 倍的压缩效果。

3.3.2 gxDNN 概述

为了能将基于 TensorFlow 的模型用于 NPU 运行，需要使用 gxDNN 工具。

gxDNN 用于将用户生成的 TensorFlow 模型编译成可以被 NPU 硬件模块执行的指令，并提供了一套 API 让用户方便地运行 TensorFlow 模型。gxDNN 神经网络处理器的作用如图 3-32 所示。

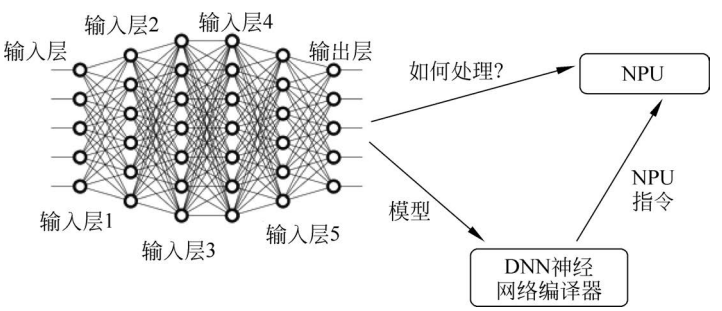


图 3-32 gxDNN 神经网络处理器的作用

gxDNN 包括 NPU 编译工具和 NPU 执行器。NPU 编译工具用于生成能够被 NPU 执行器执行的文件，该文件里包含了 NPU 执行指令和模型的描述信息。一般在 PC 上使用。NPU 执行器提供了一套 API，用于解析执行文件、加载模型、输入数据、运行模型、得到输出结果。它需要在有 NPU 硬件模块的机器上使用。

1. 使用 gxDNN 的步骤

gxDNN 神经网络处理器的作用如图 3-33 所示。

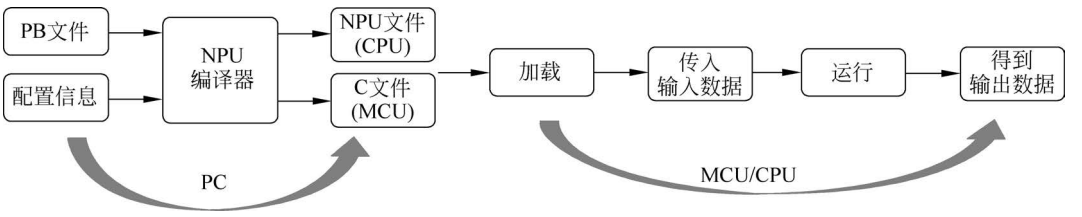


图 3-33　gxDNN 神经网络处理器的作用

(1) 在 PC 上生成 TensorFlow 的 Graph(PB 文件)和 Variable(校验点文件)后将两者合并为一个 PB 文件。

(2) 在 PC 上使用 gxDNN 的 NPU 编译器,将 PB 文件转换成能被 NPU 加载执行的指令文件,在 CPU 上该文件被称为 NPU 文件,在 MCU 上被称为 C 文件。

(3) 芯片端调用 NPU API,导入模型,传入输入数据,运行模型,得到输出数据。

2. NPU 编译器简介

TensorFlow 的运算流程基于图(Graph),图的结点称为算子,算子对 0 个或多个输入数据进行计算,生成 0 个或多个输出数据。NPU 编译器的主要工作就是把一个个算子转变成可以被 NPU 硬件模块执行的命令。

针对 CPU 和 MCU 的不同特性,生成的指令文件格式也不相同。

3. NPU API 简介

用户使用 NPU 编译器生成 NPU 指令文件后需要调用 NPU 执行器的 API,让模型运行起来。

由于 CPU 和 MCU 的系统不同,所以提供的 API 也不同。

4. NPU 性能测试

NPU 与 SNPU 时间与内存大小对比(1),见表 3-2。

表 3-2　NPU 与 SNPU 时间与内存大小对比(1)

编　译　器	时　　间	内 存 大 小
NPU	2.4ms	2.7MB
SNPU	5.8ms	3.4MB

NPU 200M 和 SNPU 120M 在 npu_compiler 1.0.14 版本下进行测试。

ASR 模型(FC320、LSTM400、LSTM400、LSTM400、LSTM400、FC211)。

NPU 与 SNPU 时间与内存大小对比(2),见表 3-3。

表 3-3　NPU 与 SNPU 时间与内存大小对比(2)

编　译　器	时　　间	内 存 大 小
NPU	7.4ms	4.6MB
SNPU	13.6ms	4.8MB

LeNet5(1×28×28→Conv2D 5×5×1×32 卷积核→ReLU→MaxPool→Conv2D。5×5×32×64 卷积→ReLU→MaxPool→FC 3136×1024 权重→FC 1024×10 权重)。

NPU 与 SNPU 时间与内存大小对比(3),见表 3-4。

表 3-4 NPU 与 SNPU 时间与内存大小对比(3)

编 译 器	时 间	内 存 大 小
NPU	221ms	35.1MB
SNPU	359ms	36.1MB

AlexNet(3×112×112→Conv2D 11×11×1×64 卷积核→ReLU→MaxPool→Conv2D 5×5×64×192 卷积核→ReLU→MaxPool→Conv2D 3×3×192×384 卷积核→ReLU→Conv2D 3×3×384×384 卷积核→ReLU→Conv2D 3×3×384×256 卷积核→ReLU→MaxPool)。

MobiLeNet V2 模型结构见表 3-5。

表 3-5 MobiLeNet V2 模型结构

输 入	算 子	*t*	*c*	*n*	*s*
160×160×3	Conv2D 1×1	-	32	1	2
80×80×32	瓶颈	1	16	1	1
80×80×16	瓶颈	6	24	2	2
40×40×24	瓶颈	6	32	3	2
20×20×32	瓶颈	6	64	4	2
10×10×64	瓶颈	6	96	3	1
10×10×64	瓶颈	6	160	3	2
5×5×160	瓶颈	6	320	1	1
5×5×320	Conv2D 1×1		1280	1	1
5×5×1280	平均池化 5×5			1	
5×5×1280	Conv2D 1×1		192		

说明:*t* 表示通道拓维因子(层内第 1 个卷积的输出通道除以输入通道),*c* 表示通道数,*n* 表示结构重复次数,*s* 表示步长。

NPU 与 SNPU 时间与内存大小对比(4),见表 3-6。

表 3-6 NPU 与 SNPU 时间与内存大小对比(4)

编 译 器	时 间	内 存 大 小
NPU	502ms	36.7MB
SNPU	1019ms	38.1MB

5. 编译器安装

NPU 编译器目前只支持在 Python 2 环境下安装和使用。

6. 安装 gxDNN 工具链

使用的命令如下:

```
pip install npu_compiler
```

7. 更新 gxDNN 工具链

使用的命令如下：

```
pip install -- upgrade npu_compiler
```

8. 查看工具链版本

安装或更新完成后，可以查看当前工具链的版本号，命令如下：

```
gxnpuc -- version
```

3.3.3　编译器使用

1. 工具链 gxnpuc 介绍

用于把模型文件编译成能在 NPU 上运行的 NPU 文件。使用的方法如下：

```
//第 3 章/gxnpuc tools.bat

usage: gxnpuc [ - h] [ - V] [ - L] [ - v] [ - m] [ - c CMD [CMD ...]] [config_filename]

NPU 编译器

位置参数:
  config_filename       配置文件

选择参数:
  - h, -- help          展示帮助信息与退出
  - V, -- version       展示程序版本号与退出
  - L, -- list          列出支持的算子
  - v, -- verbose       详细列出已处理的操作
  - m, -- meminfo       详细列出操作的内存信息
  - c CMD [CMD ...], -- cmd CMD [CMD ...]
                        使用命令行配置
```

2. 配置文件说明

配置文件的参数信息见表 3-7。

表 3-7　配置文件的参数信息

配　置　项	选　　项	说　　明
CORENAME	LEO	芯片型号
PB_FILE		包含校验点的 PB 文件
OUTPUT_FILE		编译后生成的文件名
NPU_UNIT	NPU32/NPU64	NPU 型号对应的 MAC 数(SNPU 选 NPU32，主 NPU 选 NPU64)
COMPRESS	true/false	是否启动压缩模式
COMPRESS_QUANT_BITS	4/5/6/7/8	量化压缩的 bit 数

续表

配　置　项	选　　项	说　　明
COMPRESS_TYPE	线性/高斯	线性压缩还是高斯压缩，线性压缩准确率更高，但压缩率不如高斯压缩
OUTPUT_TYPE	raw/c_code	Linux 环境下运行的模型选择 raw，VSP 下运行的模型选择 c_code
INPUT_OPS	op_name:[shape]...	设置输出的 OP 名与 shape
OUTPUT_OPS	[out_op_names,...]	设置输出的 OP 名列表
INPUT_DATA	Op_name:[data]...	有些占位符的数据在部署时是确定的，需要写明

3. gxnpudebug 调试工具

如果编译时配置文件中的 DEBUG_INFO_ENABLE 选项被设置为 true，编译出的 NPU 文件带上了调试信息，则此时可以使用调试工具 gxnpudebug 工具来处理该文件。使用的方法如下：

```
//第 3 章/gxnpudebug tools.bat
用法: gxnpudebug [ -h] [ -S] [ -P] file [file ...]

选择参数:
  -h, --help          展示帮助信息与退出
  -P, --print_info    打印调试信息
  -S, --strip         剥离调试信息
```

4. gxnpu_rebuild_ckpt 文件

对权重数据做量化或做 float16，并重新生成校验点文件，用于评估模型压缩后对结果的影响。使用的方法如下：

```
gxnpu_rebuild_ckpt [ -h] config_filename
```

5. 配置文件说明

配置文件参数信息见表 3-8。

表 3-8　配置文件参数信息

配　置　项	选　　项	说　　明
GRAPH_FILE		包含校验点的 PB 文件
OUTPUT_CKPT		输出校验点文件
MODE	quant/float16	目前只用到 float16
COMPRESS_TYPE	线性/高斯	压缩类型
MODE_BITS	4/5/6/7/8	量化比特数
NPU_UNIT	NPU32/NPU64	NPU 型号对应的 MAC 数(SNPU 选 NPU32，主 NPU 选 NPU64)
EXPECTED_OPS_NAME	input_name: Output_name...	设置输入的 OP 名与 shape

3.3.4　编译模型

1. 模型文件准备

(1) 准备 TensorFlow 生成的 PB 和校验点文件,或以 saved_model 方式生成的模型文件。

(2) 通过 TensorFlow 提供的 freeze_graph.py 脚本生成 Frozen PB 文件。

2. 编写配置文件

编写 YAML 配置文件,包括 PB 文件名、输出文件名、输出文件类型、是否压缩、压缩类型、输入节点名和维度信息、输出节点名等。

3. 编译

编译命令如下:

```
gxnpuc config.yaml
```

需要注意的是,NPU 工具链必须在安装有 TensorFlow 的环境下使用。

4. 优化模型

为了让模型更高效地运行在 NPU 处理器上,需要对模型做一些优化。

(1) 做卷积和降采样的数据格式需要用 NCHW 格式进行优化。

(2) 占位符的维度信息需要确定。

(3) 各算子的尺寸需要确定,即和尺寸有关的运算值需要确定。

(4) Softmax 不建议放在 NPU 中,因为 NPU 使用 FP16 数据格式,容易导致数据溢出。

5. CPU 中使用 NPU 或 SNPU

生成能在 CPU 上运行的模型文件,需要在编译模型的配置文件中指定。

OUTPUT_TYPE: raw 在 CPU 上内存资源相对丰富,模型指令以文件方式加载,能提高灵活性。NPU 处理器的输入/输出数据类型都是 float16,float16 和 float32 的转换 API 在内部完成,上层应用不需关心。

6. 调用 API 流程

调用 API 的流程如下:

(1) 打开 NPU 设备。

(2) 传入模型文件,得到模型任务。

(3) 获取任务的输入/输出信息。

(4) 将输入数据复制到模型内存中。

(5) 运行模型,得到输出数据。

(6) 释放模型任务。

(7) 关闭 NPU 设备。

7. MCU 中使用 SNPU

生成能在 MCU 上运行的模型文件,需要在编译模型的配置文件中指定。

OUTPUT_TYPE: c_code 在 MCU 上内存资源紧缺,生成的模型文件为 C 文件,能直

接参与编译。这样的优点是不需要解析模型文件，缺点是当需要换模型时得重新编译。另外，由于 MCU 效率较低，float16 和 float32 的转换不能在 MCU 上完成，必须由 DSP 或 ARM 来转换。

3.3.5 调用 API 流程与 MCU API 代码

调用 API 的流程如下：

(1) 打开 SNPU。

(2) 将输入数据复制到模型内存中。

(3) 运行模型，得到输出数据。

(4) 关闭 SNPU。

MCU API 的代码如下：

```
//第 3 章/mcu api.c
/* GXDNN
  * Copyright(C)1991 - 2017 NationalChip Co., Ltd
  *
  * gxdnn.h  NPU 任务装载与执行
  *
  */

#ifndef __GXDNN_H__
#define __GXDNN_H__

#ifdef __cplusplus
extern "C" {
#endif

/* ============================================================================
============================ */

typedef void* GxDnnDevice;
typedef void* GxDnnTask;

typedef enum {
    GXDNN_RESULT_SUCCESS = 0,
    GXDNN_RESULT_WRONG_PARAMETER,
    GXDNN_RESULT_MEMORY_NOT_ENOUGH,
    GXDNN_RESULT_DEVICE_ERROR,
    GXDNN_RESULT_FILE_NOT_FOUND,
    GXDNN_RESULT_UNSUPPORT,
    GXDNN_RESULT_UNKNOWN_ERROR,
    GXDNN_RESULT_OVERTIME,
} GxDnnResult;

/* ============================================================================
============================ */
```

```
/**
 * @brief   打开 NPU 设备
 * @param   [in]    devicePath      设备路径
 *          [out]   device          设备打开句柄
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 *                      GXDNN_RESULT_DEVICE_ERROR     设备错误
 * @remark  if devicePath is "/dev/gxnpu", 打开 NPU 设备
 *           devicePath is "/dev/gxsnpu", 打开 NPU 设备
 */
GxDnnResult GxDnnOpenDevice(const char * devicePath,
                    GxDnnDevice * device);

/* ============================================================================
============================ */

/**
 * @brief   关闭 NPU 设备
 * @param   [in]    device          设备打开句柄
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 *                      GXDNN_RESULT_DEVICE_ERROR     设备错误
 * @remark
 */
GxDnnResult GxDnnCloseDevice(GxDnnDevice device);

/* ============================================================================
============================ */

/**
 * @brief   从文件加载 NPU 任务(在 Linux/macOS 中)
 * @param   [in]    device          设备句柄
 *          [in]    taskPath        NPU 任务文件路径
 *          [out]   task            加载任务的句柄
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 *                      GXDNN_RESULT_MEMORY_NOT_ENOUGH 内存不够
 *                      GXDNN_RESULT_DEVICE_ERROR     设备错误
 *                      GXDNN_RESULT_FILE_NOT_FOUND   文件没找到
 *                      GXDNN_RESULT_UNSUPPORT        不支持 NPU 类型或者 NPU 任务版本
 * @remark
 */
GxDnnResult GxDnnCreateTaskFromFile(GxDnnDevice device,
                            const char * taskPath,
                            GxDnnTask * task);

/* ============================================================================
============================ */

/**
 * @brief   由内存装载 NPU 任务
 * @param   [in]    device          设备句柄
 *          [in]    taskBuffer      NPU 任务缓存指针
```

```
 *          [in]    bufferSize      NPU任务缓存大小
 *          [out]   task            装载任务句柄
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 *                      GXDNN_RESULT_MEMORY_NOT_ENOUGH 没有足够内存
 *                      GXDNN_RESULT_DEVICE_ERROR     设备错误
 *                      GXDNN_RESULT_UNSUPPORT        不支持NPU类型或者NPU任务版本
 * @remark
 */
GxDnnResult GxDnnCreateTaskFromBuffer(GxDnnDevice device,
                                      const unsigned char *taskBuffer,
                                      const int bufferSize,
                                      GxDnnTask *task);

/* ============================================================================
============================ */

/**
 * @brief   发布NPU任务
 * @param   [in]    task            装载任务句柄
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无错误成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 * @remark
 */
GxDnnResult GxDnnReleaseTask(GxDnnTask task);

/* ============================================================================
============================ */

#define MAX_NAME_SIZE 30
#define MAX_SHAPE_SIZE 10

typedef struct NpuIOInfo {
    int direction;                                  /* 0:输入; 1:输出 */
    char name[MAX_NAME_SIZE];                       /* IO名称 */
    int shape[MAX_SHAPE_SIZE];                      /* 形状大小 */
    unsigned int dimension;                         /* IO维数 */
    void *dataBuffer;                               /* 数据缓存 */
    int bufferSize;                                 /* 数据缓存大小 */
} GxDnnIOInfo;

/**
 * @brief   Get the IO Num of the loaded task
 * @param   [in]    task            the loaded task
 *          [out]   inputNum        Input number
 *          [out]   outputNum       Output Number
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 * @remark
 */
GxDnnResult GxDnnGetTaskIONum(GxDnnTask task,
                              int *inputNum,
```

```
                                int * outputNum);

/* ============================================================================
============================ */

/**
 * @brief  获取加载任务的 IO 信息
 * @param [in]      task             装载句柄
 *        [out]     inputInfo        输入参数列表
 *        [in]      inputInfoSize    输出信息列表缓冲区的大小
 *        [out]     outputInfo       输出信息列表
 *        [in]      outputInfoSize   输出信息列表缓冲区的大小
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS     无差错成功
 *                      GXDNN_ERR_BAD_PARAMETER     错误参数
 * @remark
 */
GxDnnResult GxDnnGetTaskIOInfo(GxDnnTask task,
                               GxDnnIOInfo * inputInfo,
                               int inputInfoSize,
                               GxDnnIOInfo * outputInfo,
                               int outputInfoSize);

/* ============================================================================
============================ */

typedef enum {
    GXDNN_EVENT_FINISH,
    GXDNN_EVENT_ABORT,
    GXDNN_EVENT_FAILURE
} GxDnnEvent;

/**
 * @brief   The event handler
 * @param   [in]    task            正在运行的任务
 *          [in]    event           事件类型
 *          [in]    userData        GxDnnRunTask 传递的 userData
 * @return  int     0               中断任务
 *                  not 0           继续任务
 */
typedef int( * GxDnnEventHandler)(GxDnnTask task, GxDnnEvent event, void * userData);

/* ============================================================================
============================ */

/**
 * @brief   Run task
 * @param   [in]    task            装载任务
 *          [in]    priority        任务优先级
 *          [in]    eventHandler    事件调用(参见备注)
 *          [in]    userData        将 void 数据传递给事件处理程序
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER     错误参数
```

```
 * @remark  if eventHandler == NULL, 直到完成或发生错误,函数才会返回
 *          如果任务正在运行,则该任务将首先停止
 */
GxDnnResult GxDnnRunTask(GxDnnTask task,
                         int priority,
                         GxDnnEventHandler eventHandler,
                         void * userData);

/* ============================================================================
============================ */

/**
 * @brief   Stop task                停止任务
 * @param   [in]    task             装载任务
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 * @remark  如果任务正在运行,则将调用事件处理程序
 */
GxDnnResult GxDnnStopTask(GxDnnTask task);

/* ============================================================================
============================ */

typedef struct NpuDevUtilInfo {
    float ratio;
} GxDnnDevUtilInfo;

/**
 * @brief   获取设备使用信息
 * @param   [in]   GxDnnDevice 设备
 *          [out]  GxDnnDevUtilInfo 信息
 * @return  GxDnnResult GXDNN_RESULT_SUCCESS          无差错成功
 *                      GXDNN_RESULT_WRONG_PARAMETER  错误参数
 *                      GXDNN_RESULT_DEVICE_ERROR     设备错误
 */
GxDnnResult GxDnnGetDeviceUtil(GxDnnDevice device, GxDnnDevUtilInfo * info);

/* ============================================================================
============================ */

#ifdef __cplusplus
} /* extern C */
#endif

#endif
/* 语音信号预处理
 * snpu.h: SNPU 设备驱动
 *
 */

#ifndef __SNPU_H__
#define __SNPU_H__
```

```
int SnpuInit(void);
int SnpuLoadFirmware(void);
int SnpuDone(void);

#ifdef CONFIG_GX8010NRE
int SnpuFloat32To16(unsigned int * in_data, unsigned short * out_data, int num, int exponent_
width);
int SnpuFloat16To32(unsigned short * in_data, unsigned int * out_data, int num, int exponent_
width);
#endif

typedef enum {
    SNPU_IDLE,
    SNPU_BUSY,
    SNPU_STALL,
} SNPU_STATE;

typedef int( * SNPU_CALLBACK)(SNPU_STATE state, void * private_data);

typedef struct {
    const char * version;                                    //model.c 版本
    void * ops;                                              //model.c 文件中的 ops_content 算子
    void * data;                                             //in model.c 文件中的 cpu_content
    void * input;                                            //model.c 文件中的输入
    void * output;                                           //model.c 文件中的输出
    void * cmd;                                              //model.c 文件中的 npu_content
    void * tmp_mem;                                          //model.c 文件中的 tmp_content
} SNPU_TASK;

int SnpuRunTask(SNPU_TASK * task, SNPU_CALLBACK callback, void * private_data);

SNPU_STATE SnpuGetState(void);

#endif //__SNPU_H__
```

3.3.6　NPU 使用示例

1. MNIST 示例

MNIST 是一个入门级的计算机视觉数据集，它的输入是像素为 28×28 的手写数字图片，输出是图片对应的 0～9 数字的概率。下面以 TensorFlow 自带的 MNIST 模型(TensorFlow v1.0)为例，说明 gxDNN 的使用。

2. 生成 NPU 文件

这个示例的 MNIST 计算模型非常简单，可以用一个公式来表示：$y=x\times W+b$(训练的过程中还会去计算 Softmax 函数，但由于正式使用时，只需获取结果中最大值的索引，而 Softmax 是个单调递增函数，因此省去这个函数不会对结果有影响)，其中 x 为输入数据，y 为输出数据，W 和 b 为训练的参数。训练的过程就是不断地通过计算出来的 y 和期望的 y 去调整 W 和 b 的过程。在 NPU 上，只需用到训练好的 W 和 b，而不需要训练的过程。

3. 生成校验点和 PB 文件

为了方便地获取输入节点和输出节点，给输入节点和输出节点取个名字，把 x 取名为 input_x，把 y 取名为 result，代码如下：

```
//第 3 章/input output.py
#mnist_softmax.py 第 40 行
x = tf.placeholder(tf.float32, [None, 784])
#修改为
x = tf.placeholder(tf.float32, [None, 784], name = "input_x")
#第 43 行
y = tf.matmul(x, W) + b
#修改为
y = tf.add(tf.matmul(x, W), b, name = "result")
#为了生成校验点和 PB 文件，在 main 函数的末尾添加代码
saver = tf.train.Saver()
saver.save(sess, "mnist.ckpt")
tf.train.write_graph(sess.graph_def, "./", "mnist.pb")
#运行程序后，当前路径下会生成 mnist.ckpt.* 和 mnist.pb 文件
```

4. 把校验点和 PB 文件合并成一个 PB 文件

使用 freeze_graph.py 脚本将 mnist.ckpt.* 和 mnist.pb 合并为一个 PB 文件。注意，不同 TensorFlow 版本的 freeze_graph.py 脚本可能不同。执行的命令如下：

```
//第 3 章/ckpt pb merge.py
python freeze_graph.py --input_graph = mnist.pb --input_checkpoint = ./mnist.ckpt
--output_graph = mnist_with_ckpt.pb --output_node_names = result
```

生成 mnist_with_ckpt.pb 文件，其中，--input_graph 后跟输入的 PB 名；--input_checkpoint 后跟输入校验点名；--output_graph 后跟合成的 PB 文件名；--output_node_names 后跟输出结点名称，如果有多个，则用逗号分隔。

执行完成后，在当前路径下会生成 mnist_with_ckpt.pb 文件。如果以 saved_model 方式保存模型，则执行完成后会在当前路径下生成 mnist_with_ckpt.pb 文件。

如果以 saved_model 方式保存模型，则使用的命令如下：

```
//第 3 章/model pb merge.py
python freeze_graph.py --input_saved_model_dir = ./saved_model_dir --output_graph = mnist_
with_ckpt.pb --output_node_names = result
```

5. 编辑 NPU 配置文件

编辑 NPU 配置文件 mnist_config.yaml，代码如下：

```
//第 3 章/mnist_config.yaml
CORENAME: LEO                    #芯片型号
PB_FILE: mnist_with_ckpt.pb      #输入的 PB 文件
OUTPUT_FILE: mnist.npu           #输出的 NPU 文件名
```

```
SECURE: false              #不开启内容保护
NPU_UNIT: NPU64            #NPU 设备类型
COMPRESS: true             #压缩模型
COMPRESS_QUANT_BITS: 8     #量化成 8 位 s
OUTPUT_TYPE: raw           #NPU 文件的类型
INPUT_OPS:
input_x: [1, 784]          #输入节点名称和数据维度,每运行一次输入数据为 1×784,即一张图
OUTPUT_OPS: [result]       #输出节点名称
```

6. 编译

生成 NPU 文件 mnist.npu,代码如下：

```
//第 3 章/mnist.npu
#使用 gxnpuc 工具编译,命令如下
gxnpuc mnist_config.yaml
#如果 gxnpuc 的版本在 1.0 之前,则可使用的命令如下
gxnpuc --config=./mnist_config.yaml
#生成 NPU 文件 mnist.npu
```

7. 执行 NPU 文件

NPU 文件生成后,需要调用 API,把模型部署到 GX8010 开发板上运行。

8. 调用 SDK 流程

调用 SDK 的流程如下：

(1) 打开 NPU 设备。

(2) 传入模型文件,得到模型任务。

(3) 获取任务的输入/输出信息。

(4) 将输入数据复制到模型内存中。

(5) 运行模型。

(6) 释放模型任务。

(7) 关闭 NPU 设备。

9. mnist 示例

示例代码可参考 https://github.com/NationalChip/gxDNN/blob/master/examples/mnist/execution/test_mnist.c。

程序要求用户输入一个保存有 28×28 像素值的二进制文件,输出识别的数字。images 的链接为 https://github.com/NationalChip/gxDNN/blob/master/examples/mnist/execution/images。

其中存放的是若干二进制测试文件,内容为 28×28 的已做归一化的像素值。

执行 make 命令生成可执行文件 test_mnist.elf。

把 mnist.npu、test_mnist.elf 和 images 目录放到 GX8010 开发板上,并运行,代码如下：

```
//第 3 章/test_mnist.txt
./test_mnist.elf images/image0
```

```
Digit: 7
./test_mnist.elf images/image1
Digit: 2
./test_mnist.elf images/image2
Digit: 1
./test_mnist.elf images/image3
Digit: 0
./test_mnist.elf images/image4
Digit: 4
./test_mnist.elf images/image5
Digit: 1
```

3.4 TPU2 机器学习集群

3.4.1 TPU2 概述

谷歌 I/O 上第 2 代 TensorFlow 处理单元(TPU2)如图 3-34 所示。谷歌称这一代为谷歌云 TPU,但除了提供一些彩色照片外,几乎没有提供有关 TPU2 芯片和使用它的系统的信息。图片确实比文字更能说明问题,因此在本节中,将深入研究照片,并根据图片和谷歌提供的少量细节进行讲述。

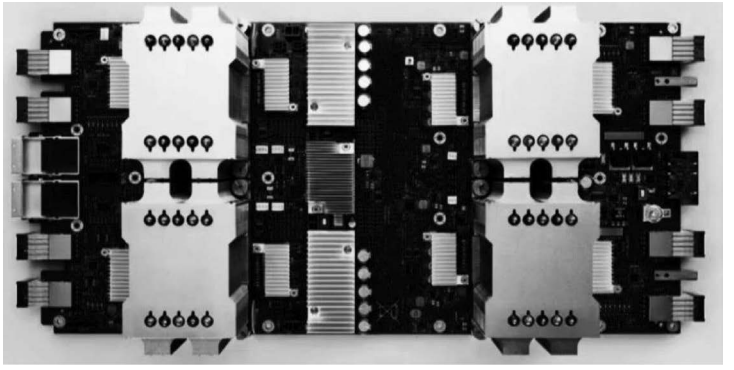


图 3-34 谷歌 I/O 上第 2 代 TensorFlow 处理单元(TPU2)

谷歌不太可能销售基于 TPU 的芯片、电路板或服务器——TPU2 是谷歌的专属内部产品。谷歌将只通过 TensorFlow Research Cloud(TRC)提供对 TPU2 硬件的直接访问,这是一个高度选择性的计划,旨在让研究人员分享 TPU2 可以加速的代码类型的发现,以及通过谷歌计算引擎云 TPU Alpha 程序,可以认为这也是高度选择性的,因为两条通往市场的途径共享一个注册页面。

谷歌专门设计 TPU2,以加速其面向消费者的核心软件(如搜索、地图、语音识别和自动驾驶汽车训练等研究项目)背后的专注深度学习的工作负载。

谷歌为深度学习推理和分类任务设计了最初的 TPU——运行已经在 GPU 上训练过的模型。TPU 是一种协处理器,通过两个 PCI 总线 3.0×8 边缘连接器连接到处理器主板(如

图 3-35 的左下角所示),总计 16GB/s 的双向带宽。TPU 的功耗高达 40W,完全符合 PCI 总线供电规格,为 92 位整数运算提供 8 Tera 运算(TOPS),为 23 位整数运算提供 16TOPS。相比之下,谷歌声称 TPU2 的峰值为每秒 45TFlops 浮点运算。

TPU 没有内置的调度功能,也无法虚拟化。它是一个直接连接到一个服务器主板的简单矩阵乘法协处理器。谷歌的第 1 代 TPU 卡如图 3-35 所示,A 是不带散热器,B 是带散热器。

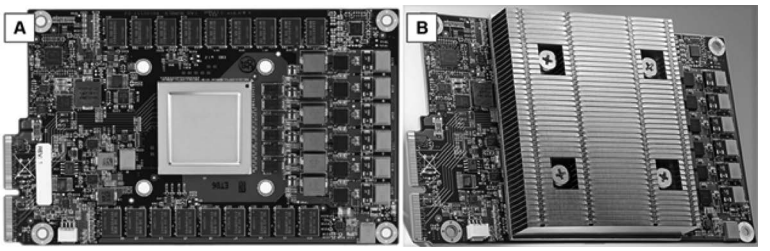


图 3-35　第 1 代 TPU 架构图

谷歌从未说过此主板的处理能力,或 PCI 总线吞吐量过载之前,它连接到一个服务器主板的 TPU 数量。协处理器需要主机处理器的大量支持,其形式包括任务设置和拆卸,以及管理进出每个 TPU 的数据传输带宽。协处理器只做一件事情,但它们被设计成可以很好地完成这一件事。

3.4.2　TPU2 设计方案

1. TPU2 图案标记的高级组织

谷歌将其 TPU2 设计用于四机架图案,称为吊舱。标记是一组相关工作负载的标准机架配置(从半机架到多个机架)。标记有助于大型数据中心所有者更轻松地进行购买、安装和部署,并降低成本。例如,Microsoft 的 Azure 栈标准半机架是一个图章。

四机架图案尺寸主要是基于谷歌使用的铜缆类型和全速运行的最大铜线长度,图案标记的高级组织如图 3-36 所示。

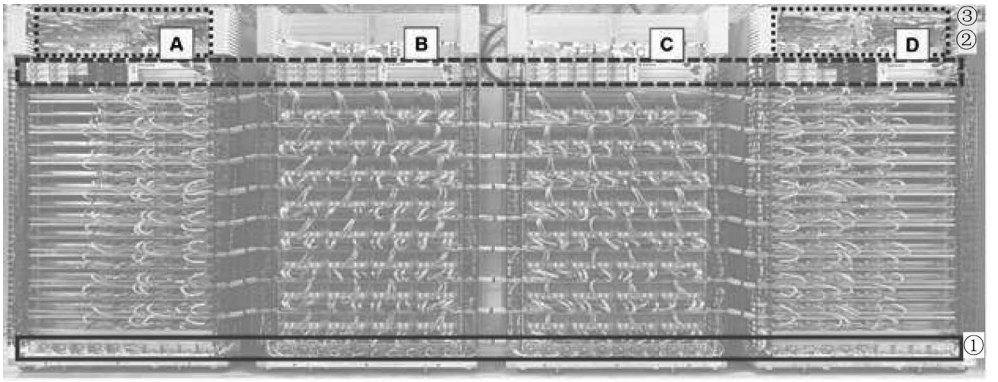


图 3-36　TPU2 图案标记的高级组织

注意到的第一件事是谷歌通过两根电缆将每个 TPU2 板连接到一个服务器处理器板。可能是谷歌将每个 TPU2 板连接到两个不同的处理器板，但即使是谷歌也不太可能想要弄乱该拓扑的安装、编程和调度复杂性。如果服务器主板和 TPU2 主板之间存在一对一连接，则要简单得多。

谷歌 TPU2 图案的含义如下：

(1) A 是 CPU 机架，B 是 TPU2 机架，C 是 TPU2 机架，D 是 CPU 机架。

(2) 实心箱(①)：机架不间断电源(UPS)。

(3) 虚线框(②)是电源。

(4) 虚线框(③)：机架网络交换机和架顶交换机。

TPU2 图案标记的 3 张不同照片如图 3-37 所示，这 3 张照片的配置和布线看起来都是一样的。TPU2 布线的花哨颜色和编码对这种比较有很大帮助。

图 3-37　TPU2 图案标记的 3 张不同照片

2. TPU2 板的俯视图

谷歌发布了 TPU2 板的俯视图和板前面板连接器的特写。4 个 TPU2 板象限中的每个连接器都共享电路板功率分配。4 个 TPU2 板象限也通过简单的网络交换机共享网络连接。看起来每个板象限都是一个单独的子系统，并且 4 个子系统在板上没有相互连接。TPU2 板的俯视图如图 3-38 所示。

(1) A 是 4 个带散热器的 TPU2 芯片。

(2) B 是每个 TPU25 两根带宽为 2GB/s 的 BlueLink 电缆。

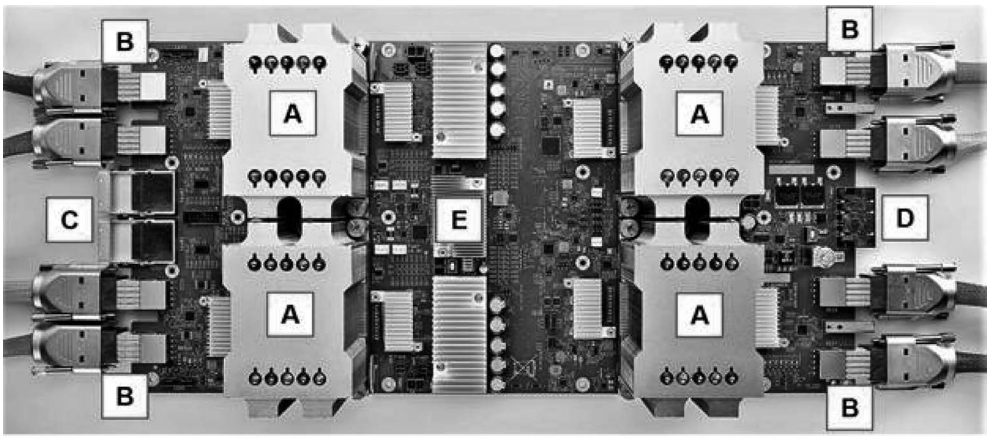


图 3-38　TPU2 板的俯视图

(3) C 是每块板两根全路径架构(OPA)电缆。

(4) D 是板背面电源连接器。

(5) E 很可能是网络交换机。

3. 前面板连接器

TPU2 前面板连接如图 3-39 所示。前面板连接看起来像一个 QSFP 网络连接器,两侧是 4 个以前从未见过的方形横截面连接器。IBM BlueLink 规范为每个方向定义了 200 个 16Gb/s 信号通道(总共 25 个通道),以实现最小 GB/s 配置(称为子链路)。谷歌是 OpenCAPI 的成员,也是 OpenPOWER Foundation 的创始成员,因此使用 BlueLink 是有道理的。

图 3-39　TPU2 前面板连接

电路板前部中央的两个连接器看起来像带有铜双绞线束的 QSFP 型连接器,而不是光纤。这提供了两种选择,即 10Gb/s 以太网,或 100Gb/s 英特尔全路径架构(OPA)。两个 100Gb/s OPA 链路可以组合在一起,形成 25GB/s 的总双向带宽,这与 BlueLink 速度相匹配,因此认为它是 Omni-Path。

这些铜缆,BlueLink 或 OPA 都不能以最大信号速率运行超过 3m 或 10 英尺。互连拓扑将连接 CPU 和 TPU2 板的互连拓扑绑定在一起,物理跨度为 3m。谷歌使用颜色编码的电缆,这使组装更容易,而不易发生布线错误。最前面连接器下方与电缆颜色匹配的贴纸如图 3-39 所示,颜色编码表明,谷歌计划更大规模地部署这些 TPU2 图案标记。

白色电缆很可能是 1Gb/s 以太网系统管理网络。没有看到谷歌可以将管理网络连接到照片中的 TPU2 板的方法,但是,根据白色电缆的路由,确实假设谷歌将管理网络从后面连接到处理器板。也许处理器板可通过其 OPA 连接管理和评估 TPU2 板的运行状况。

谷歌的 TPU2 机架图案具有双边对称性。翻转处理器机架 D 以将其与处理器机架 A

进行比较，如图 3-40 所示。这两个机架是相同但彼此镜像的。两个翻转机架 C 的 TPU2 机架的比较，很明显机架 B 和 C 也是彼此的镜像，如图 3-41 所示。

图 3-40　将两个 CPU 机架与机架 D 翻转进行比较

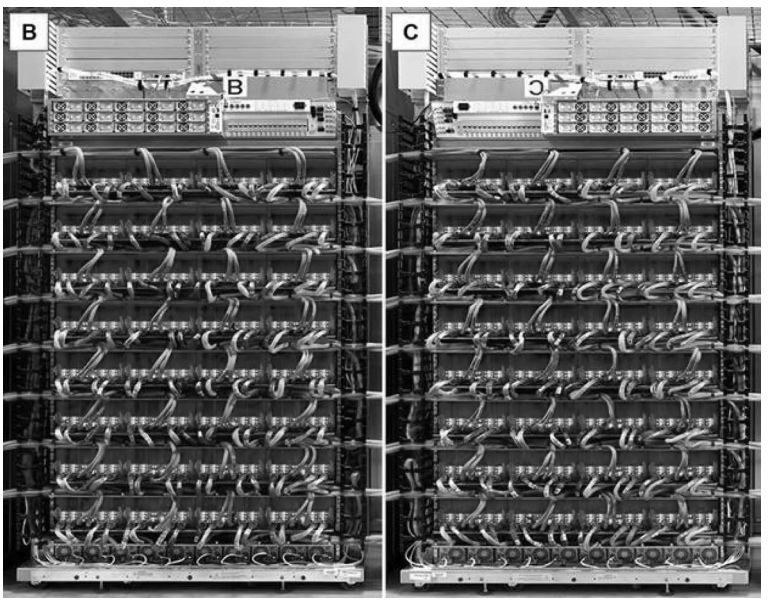


图 3-41　两个翻转机架 C 的 TPU2 机架的比较

谷歌的照片中没有足够的布线来确定确切的互连拓扑，但它确实看起来像某种超网状互连。

相信 CPU 主板是标准的英特尔 Xeon 双插槽主板，适合谷歌的 1.5 英寸服务器外形。它们是当前一代的主板，因为它们具有 OPA，所以可能是 Skylake 板。相信它们是双插槽主板，因为还没有听说很多单插槽主板正在通过英特尔供应链的任何部分发货。随着 AMD 采用那不勒斯 Epyc x86 服务器芯片和高通采用 Centriq ARM 服务器芯片强调单插槽配置，这种情况可能会发生变化。

4. TPU2 光纤带宽

谷歌使用两根 OPA 电缆将每个 CPU 板恰好连接到一个 TPU2 板，以实现 25GB/s 的总带宽。这种一对一的连接回答了 TPU2 的一个关键问题——谷歌设计的 TPU2 图案与 TPU2 芯片与至强插座的比例为 1∶2。也就是说，每个双插槽至强服务器需要 4 个 TPU2 芯片。

TPU2 加速器与处理器的这种紧密耦合与深度学习训练任务中，GPU 加速器典型的 4∶1 到 6∶1 比率大不相同。2∶1 的低比例表明谷歌保留了原始 TPU 中使用的设计理念：TPU 在精神上更接近 FPU(浮点单元)协处理器，而不是 GPU。该处理器仍在谷歌的 TPU2 架构中做大量工作，但它正在将所有矩阵数学卸载到 TPU2 上。

在 TPU2 图案中看不到任何存储。架空追逐中的大束蓝色光缆的用途，如图 3-42 所示。数据中心网络连接到 CPU 板，没有光纤电缆路由到机架 B 和 C，TPU2 板上也没有网络连接。

图 3-42　谷歌数据中心其余部分的大量光纤带宽

每个机架有 32 个计算单元，无论是 TPU2 还是 CPU，每个图案上有 64 块 CPU 板和 64 块 TPU 板，总共 128 个 CPU 芯片和 256 个 TPU2 芯片。

谷歌表示，TensorFlow Research Cloud(TRC)包含 1 个 TPU000 芯片，但略有下降。4 个图案标记包含 21 个 TPU024 芯片，因此，4 个图案是谷歌已经部署了多少 TPU2 芯片的下限。在输入/输出期间发布的照片中可以看到 3 个(可能是 4 个)图案标记。

目前尚不清楚处理器对和 TPU2 芯片如何跨标记联合，以便 TPU2 芯片可以在超网状网络中的链路之间有效地共享数据。几乎可以肯定，TRC 不能跨越 4 个图案(256 个 TPU2 芯片)中的一个以上的单个任务。最初的 TPU 是一个简单的协处理器，因此处理器处理所有数据流量。在此体系结构中，处理器通过数据中心网络从远程存储访问数据。

也没有描述图案记忆模型。TPU2 芯片是否可以跨 OPA 使用远程直接内存访问(RDMA)从处理器板上的内存加载自己的数据？似乎很有可能。处理器板似乎也可能在整个标记中执行相同的操作，从而创建一个大型共享内存池。该共享内存池不会像机器共享内存系统原型中的内存池那样快，但是对于 25GB/s 的连接，它不会很慢，而且可能仍然很大，在两位数的 TB 范围内(每个 DIMM 16GB，每个处理器包含 8 个 DIMM，每个板包含两个处理器，

64 块板共 16TB 内存)。

据推测,在图案标记上安排一个需要多个 TPU2 的任务如下:

(1) 处理器池应具有标记的超网状拓扑图,以及哪些 TPU2 芯片可用于运行任务。

(2) 处理器组可以联合对每个 TPU2 进行编程,用以在连接的 TPU2 芯片之间显式地连接网格。

(3) 每个处理器板将数据和指令加载到其配对的 TPU2 板上的 4 个 TPU2 芯片上,包括网状互连的流量控制。

(4) 处理器在互连的 TPU2 芯片之间同步启动任务。

(5) 任务完成后,处理器从 TPU2 芯片收集结果数据(该数据可能已通过 RDMA 存在于全局内存池中),并将 TPU2 芯片标记为可用于其他任务。

这种方法的优点是 TPU2 芯片不需要了解多任务处理、虚拟化或多租户——处理器的任务是处理整个标记中的所有内容。

这也意味着,如果谷歌提供云 TPU 实例作为谷歌云平台定制机器类型 IaaS 的一部分,则该实例必须同时包含处理器和 TPU2 芯片。

希望工作负载可以跨标记缩放,并保留超网格的低时延和高吞吐量。虽然研究人员可以通过 TRC 访问 1024 个 TPU2 芯片中的一些,但跨图案标记扩展工作负载似乎是一个挑战。研究人员可能有能力连接多达 256 个 TPU2 芯片的集群,因为云 GPU 能连接到 32 个互联设备(通过微软的 Olympus HGX-1 项目设计)。

5. 功耗散热处理

谷歌的第 1 代 TPU 在负载下的功耗 40W,同时以 16TOPS 的速率执行 23 位整数矩阵乘法。对于 TPU2,谷歌将运行速度提高了一倍,达到 2TFLOPS,同时通过升级到 16 位浮点运算来增加计算复杂性。粗略的经验法则是,功耗至少翻了两倍——TPU2 的功耗至少为 160W,如果它除了将速度提高一倍并移动到 FP16 之外什么都不做。散热器尺寸暗示功耗要高得多,高达 200W。

TPU2 板在 TPU2 芯片顶部有巨大的散热器。它们是多年来见过的最高的风冷散热器。它们具有内部密封液体循环。将 TPU2 散热器与早前数据的最大可比散热器进行比较,如图 3-43 所示。

A 是 4 个 TPU2 主板侧视图,B 是双 IBM Power9 Zaius 主板,C 是双 IBM Power8 Minsky 主板,D 是双 Intel Xeon Facebook Yosemite 主板,E 是 NVIDIA P100 SMX2 模块,带有散热器和 Facebook Big Basin 主板。

这些散热器的尺寸暗示每个功耗超过 200W。很容易看出,它们比原始 TPU 上的 40W 散热器大得多。这些散热器填充了两个谷歌垂直 1.5 英寸的谷歌外形单元,因此它们几乎有 3 英寸高(谷歌机架单元高度为 1.5 英寸,比行业标准的 1.75 英寸 U 型高度略短)。可以肯定的是,每个 TPU2 芯片也有更多的内存,这有助于提高吞吐量并增加功耗。

此外,谷歌从为单个 TPU 芯片供电的 PCI 总线(PCI 总线插槽为 TPU 卡供电)转向 4 个 TPU2 板设计,共享双 OPA 端口和交换机,以及每个 TPU2 芯片两个专用的 BlueLink

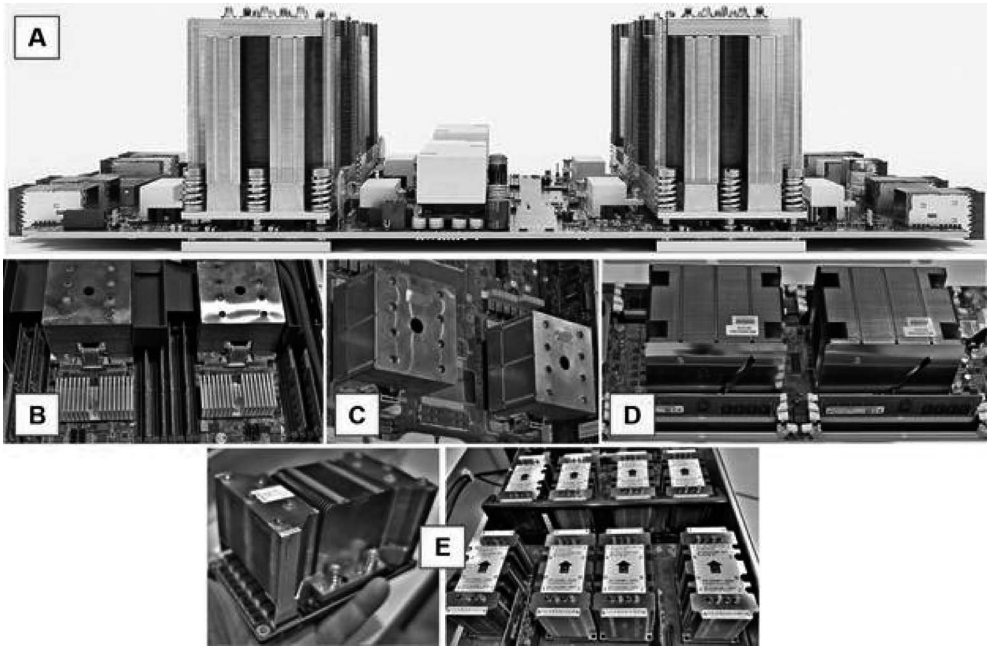


图 3-43　并行处理器中的散热器

端口。OPA 和 BlueLink 都增加了 TPU2 板级功耗。

谷歌的开放计算项目机架规格显示了 6kW、12kW 和 20kW 的供电曲线。20kW 的配电支持 90W 的 CPU 处理器插槽。可以估计，随着 Skylake 一代至强处理器和 TPU2 芯片处理大部分计算负载，机架 A 和 D 可能使用 20kW 的电源。

机架 B 和 C 是另一回事。30kW 的功率输送将为每个 TPU200 插座提供 2W 的电力。每个机架的功率为 36kW，可为每个 TPU250 插座提供 2W 的功率。36kW 是一种常见的高性能计算能力传输规范。可以认为，每个芯片 250W 的功耗也是谷歌为上面显示的巨大 TPU2 散热器付费的唯一原因，因此，单个 TPU2 图案标记的功率输出，可能在 100～112kW 的范围内，并且可能更接近较高的数字。

这意味着，TRC 在满负荷运行时消耗近半兆瓦的电力。虽然部署 4 个图案用于研究的成本很高，但这是一次性的资本支出，不会占用很多数据中心空间，然而，半兆瓦的电力是一笔巨大的运营费用，可以持续用于学术研究，即使对于谷歌这样规模的公司也是如此。如果 TRC 在一年内仍在运行，则表明谷歌正在认真地为其 TPU2 寻找新的用例。

一个 TPU2 图案包含 256 个 TPU2 芯片。每个 TPU45 芯片 2 TFlops 时，每个图案可产生总计 11.5 PFlops 的深度学习加速器性能。这令人印象深刻，即使它确实是 FP16 的峰值性能。深度学习训练通常需要更高的精度，因此 FP32 矩阵乘法性能可能是 FP16 性能的四分之一，或每个图案标记约 2.9 PFlops 和整个 TRC 的 11.5 FP32 PFlops。

在峰值性能下，这意味着整个标记中的 FP100 操作每瓦 115 GFlops 到 16 GFlops（不包括 CPU 性能贡献或位于标记外部的存储）。

在英特尔披露双插槽 Skylake 一代至强内核数量和功耗配置后，可以计算至强处理器的 FP16 和 FP32 性能，并将其添加到每瓦的总性能中。

目前还没有足够的关于谷歌 TPU2 图案行为的信息，无法可靠地将其与英伟达新一代 Volta 等商业加速器产品进行比较。这些架构差异太大，如果不在同一任务上对两种架构进行基准测试，就无法进行比较。比较峰值 FP16 的性能，就像仅根据处理器的频率比较，具有不同处理器、内存、存储和图形选项的两台 PC 的性能。

可以认为真正的竞争不是在芯片层面。挑战在于将计算加速器扩展到百万兆次级比例。英伟达通过 NVLink 迈出第 1 步，并追求与处理器更大的加速器独立性。英伟达将其软件基础设施和工作负载基础从单个 GPU 扩展到 GPU 集群。

谷歌选择将其原始 TPU 扩展为直接连接到处理器的协处理器。TPU2 还可以横向扩展为处理器的直接 2∶1 加速器，但是，TPU2 超网格编程模型似乎没有可以很好地缩放的工作负载，然而，谷歌正在寻找第三方帮助，以查找可与 TPU2 架构一起扩展的工作负载。