计 算 机 研 究 与 发 展
Journal of Computer Research and Development
https://crad.ict.ac.cn
62(4): 1003−1016, 2025

基于访存图案变形的 CGRA 存储划分优化

潘德财　牟　迪　尚家兴　刘大江
（重庆大学计算机学院　重庆　401331）
（pandc@cqu.edu.cn）

Memory Partitioning Optimization of CGRA Based on Access Pattern Morphing

Pan Decai, Mou Di, Shang Jiaxing, and Liu Dajiang
（*College of Computer Science*, *Chongqing University*, *Chongqing* 401331）

Abstract　With run-time configurable hardware, coarse-grained reconfigurable array (CGRA) is a potential platform to provide both program flexibility and energy efficiency for data-intensive applications. To exploit the access parallelism of the multi-bank memory, memory partitioning is usually introduced to CGRAs. However, existing work for memory partitioning on CGRAs either achieves the optimal partitioning solution with expensive addressing overheads or achieves area-and-energy efficient hardware at the sacrifice of more bank consumption. To this end, we propose an efficient memory partitioning approach for loop pipelining on CGRA via access pattern morphing. By performing a memory partitioning and scheduling co-optimization on multi-dimensional arrays, a memory partition-friendly access pattern is formed in the data domain such that it can be partitioned with a minimized number of all-one partitioning hyperplanes, resulting in both optimized partition factor and reduced addressing overhead. To solve the partitioning problem, firstly, we propose a backtracking-based scheduling algorithm to find the partition-friendly pattern with minimized initiation interval. Then, based on the partitioning result, we also propose an energy-area-efficient CGRA architecture by simplifying the address generators in load-store units. The experimental results show that our approach can achieve 1.25 times energy efficiency while keeping a moderate compilation time, as compared with the state-of-the-art method.

Key words　CGRA; memory partitioning; software/hardware codesign; operator scheduling; access pattern

摘　要　由于兼具高灵活性和高能效的特征，粗粒度可重构阵列（coarse-grained reconfigurable array, CGRA）是一种具有潜力的领域定制加速器架构. 为了利用多 bank 存储器的访问并行性，通常会在 CGRA 中引入存储器划分. 然而，在 CGRA 上进行存储划分工作要么以昂贵的寻址开销为代价实现最佳分区解决方案，要么以更多的存储 bank 消耗为代价来减少面积和功耗开销. 为此，提出了一种通过访存图案变形来实现面向 CGRA 的存储划分方法. 通过对包含多维数组的应用进行存储划分和算子调度协同优化，形成了存储划分友好的访存图案，从而可以用全“1”超平面对其进行存储划分，进而优化了划分结果并减少了访存地址计算开销. 基于全“1”超平面的划分策略，还提出了一种可精简地址生成单元的高能效 CGRA 架构. 实验结果表明，与最先进的方法相比，该方法可以实现 1.25 倍的能效提升.

关键词　粗粒度可重构阵列；存储划分；软硬件联合设计；算子调度；访存图案

收稿日期：2024-02-02；**修回日期**：2024-08-12
基金项目：国家自然科学基金项目（62274019）；重庆市自然科学基金项目（CSTB2022NSCQ-MSX1017）
This work was supported by the National Natural Science Foundation of China (62274019) and the Natural Science Foundation of Chongqing City (CSTB2022NSCQ-MSX1017).
通信作者：刘大江（liudj@cqu.edu.cn）

中图法分类号 TP391

DOI: 10.7544/issn1000-1239.202440079 CSTR: 32373.14.issn1000-1239.202440079

粗粒度可重构阵列(coarse-grained reconfigurable array, CGRA)是一种可空域并行、时域扩展的新型领域定制加速器架构. 通过组织不同粒度和不同功能的计算资源，具有运行时可配置数据通路的特征. 在运行时，CGRA 可根据数据流的特点，让可配置的硬件资源互连形成相对固定的数据通路，以接近“专用电路”的方式进行计算；当算法和应用变换时，CGRA 可通过重复配置，形成不同的数据通路去执行不同的任务. 因此，CGRA 是一种很有潜力的计算平台，可以在灵活性、性能和功耗之间提供良好的平衡. 由于 CGRA 具有粗粒度处理单元(processing element, PE)和稀疏的 PE 间互连，与细粒度现场可编程门阵列(field programable gate array, FPGA)相比，CGRA 的配置开销要小得多. 因此 CGRA 可以为广泛的应用提供高能效的解决方案[1-3].

如图 1 所示，一个典型的 CGRA 通常由主控制器、多 bank 存储器，DMA(direct memory access)和 2D mesh 网络连接的处理单元阵列(processing element array, PEA)组成. 每个 PE 主要由支持定点操作的功能单元(function unit, FU)、本地寄存器堆(local register file, LRF)、输出寄存器和配置缓存组成. 通过 2 组多路复用器(multiplexer, MUX)，每个 PE 可以为其 FU 选择来自邻居 PE 或访存单元(load-store unit, LSU)的数据. PE 和 PE 之间、PE 和 LSU 之间往往通过 mesh 网络互连，实现局部数据的交换. 通过 LSU，PE 可以和多 bank 存储器进行数据交换. LSU 通过全互连的交叉选择开关，可以访问多 bank 存储器中的任何一个 bank. 如果 PE 之间发生访存冲突，即不同的 PE 需要同时访问数据存储器中的同一个 bank，则会发生数据访存冲突. 冲突的访存请求将会被分配到多个时钟周期，这将严重影响 CGRA 的流水执行性能.

类似于图 1 的 CGRA 结构我们也称为访存-计算解耦的 CGRA 架构. 因为在这种架构中，存在专用的 LSU 来确定内存访问的地址. 不需要在 PE 阵列上进行访问地址的计算，这使得这种架构可以进一步提升 PE 执行计算任务的效率. 这种访存-计算解耦的架构已经在许多加速器架构[4-5]上得到了应用，并且在性能上取得了不错的提升. 在 CGRA 上，这种访存-计算解耦也已经开展了多项研究[6-8]. 因此本文将侧重于访存-计算解耦的 CGRA 目标架构[9].

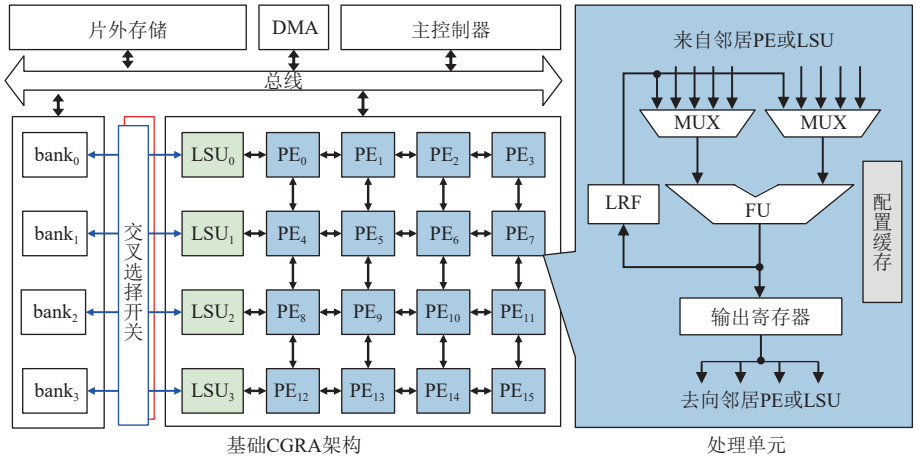


Fig. 1 The basic architecture of CGRA

图 1 CGRA 基础架构

为了加速执行 CGRA 上的循环内核，通常会使用模调度[10]来使循环内核对应的数据流图(data-flow graph, DFG)以流水的方式执行. 通过最小化迭代间的启动间隔(initiation interval, *II*)，改善循环执行的性能. 但是较小的启动间隔会引起更多的并发内存访问操作. 当循环内核包含具有对单个数组的多个引用时，对该数组的内存访问操作数量很容易超过 bank 的端口数量(通常每个 bank 为单端口或者双端口). 在多 bank 结构中，甚至会超过 bank 的数量，这将导致访存冲突，造成性能损失.

为了打破 bank 端口的限制，存储划分通常将原始数组拆分存放到多个 bank 中. 常用的存储划分方

法可以分为 3 类[11]：块划分、循环划分以及块循环划分. 例如：文献 [12] 提出了一种基于线性变换（linear transformation based, LTB）的多维数组存储划分方法. 但是这种方法仅限于循环划分的方式，在某些情况下只能找到次优解. 并且由于其变换公式中包含较多的除法、乘法等复杂计算，导致其访存地址计算的开销较大. 为了简化访存地址计算的复杂性，文献 [13] 提出了一种访存-计算解耦的 CGRA 存储划分方法（CASCADE），该方法支持将 1 维数据划分到多 bank 中. 对于多维数据而言，CASCADE 首先通过专用硬件将其展平为 1 维数组，然后对展平的数组进行块循环划分. 虽然这种方法简化了地址计算的复杂性，但是数组展平之后的划分结果具有不稳定性和次优性，其结果取决于原始数据的大小和尺寸. 对于不同的数组大小，会生成不同的划分方案，其中许多方案是次优的.

考虑到现有的方法无法在地址计算的复杂度和划分质量之间取得较好的平衡，我们提出了一种基于访存图案变形的粗粒度可重构计算架构存储划分优化方法. 本文的主要贡献有 2 点：

1）提出了一种基于访存图案变形的存储划分算法，该算法仅需要全“1”系数超平面来进行存储划分，并能在访问地址生成开销和划分结果质量之间取得良好的平衡.

2）实现了一种高面积效率和高能量效率的 CGRA 架构，以精简的地址生成单元来支持上述基于访存图案变形的存储划分结果.

实验结果表明，与目前最先进的架构相比，我们的架构具有平均 1.25 倍的能量效率提升.

1 相关工作

如引言所述，在多 bank 存储结构中，单个 bank 存在端口的限制. 因此为了打破这种限制，存储划分通常选择将同一个数组中的元素存放到多个 bank 中，以实现对同一个数组的并行数据访问. 在 FPGA 的高层次综合[14] 中，存储划分已经得到了比较好的研究. 例如：文献 [15] 中的工作尝试在同一个循环迭代中以循环划分的方式将一个数组的多个内存访问划分到多个 bank 中. 文献 [16] 提出了对循环体内核中不同迭代的内存访问进行调度，以找到接近最优的存储划分方式. 尽管这些工作为循环流水中的高效存储划分迈出了第一步，但它们仅限于 1 维数据. 然而实际应用程序中通常包含具有多维数据的嵌套循环. 在文献 [17] 中，一种通用存储划分（generalized memory partitioning, GMP）方法被提出用来支持多维数组的块循环划分，以便在大多数情况下都能找到最优值，但是这种方法具有较大的地址计算开销. 在文献 [18] 提出通过数据重用的方法来消除一些对内存的访问操作，减少并行访存数量，但是这种方法会消耗额外的寄存器资源. 文献 [19] 提出了基于晶格的存储划分方法，具有较少的硬件和能量开销. 文献 [20] 提出了一种存储划分算法来支持仿射和非仿射访存图案. 文献 [21] 提出了一种基于图论的存储划分（GMB）方法来实现最优 bank 数量. 现有的这些方法，包括基于超平面的和基于图论的，都是针对给定形状的访存图案来寻找最优的 bank 数量，失去了同时最小化 bank 数量和硬件开销的机会.

在 CGRA 上，存储划分变得更具有挑战性，这是因为在 CGRA 上 bank 的数量十分有限（例如 4~8 个），同时复杂的地址生成对 CGRA 不太友好. 为此，迫切地需要一种能够以较低的地址生成代价来实现（近）全局最优（即最小化 bank 的数量）的划分策略. 为了能够解决上述挑战，已经存在多种关于 CGRA 的存储划分的方案. 文献 [22–23] 考虑存储管理和 CGRA 映射 2 方面因素下，提出了一种无冲突的循环内核映射算法. 文献 [24] 提出了基于 LTB 存储划分和模调度的无冲突的循环映射算法. 然而该方法由于仅考虑了迭代内的内存访问并且忽略了块划分，导致可能无法找到最优解. 此外，如果基于 LTB 算法来实现地址生成单元，那么芯片面积和功耗开销会显著增大. 为了简化地址生成的复杂性，文献 [13] 提出了 CASCADE. 尽管这种方法很大程度上简化了地址生成，但是由于受到原数组尺寸大小的影响，导致划分结果在多数情况下并非最优解，甚至在某些情况下，解的质量十分糟糕.

从上面的分析中，我们注意到现有的 CGRA 存储划分工作无法在地址生成开销和划分质量之间取得良好的平衡. 为了能在控制地址生成的开销条件下，进一步提高划分结果的质量，本文提出了一种基于访存图案变形的 CGRA 存储划分优化方法. 由于访存图案的形状对划分结果至关重要，并且它与循环流水线中内存访问操作的调度密切相关，因此本文方法试图形成一种划分友好的访存图案，以在地址计算开销和划分质量之间达到良好的平衡. 尽管在这之前已经有人研究过存储划分和调度协同优化，但是他们仅限于 1 维数据或专注于迭代内的内存访问. 而本文的方法可以针对 2 维数组进行算子调度和

内存划分的协同优化.

2 存储划分的相关定义

在本节中我们主要介绍存储划分方法中的一些概念和定义.

在存储划分中, 通常用向量的形式来表示数据在数组中的位置. 例如, 对于一个 2 维数组, 其数据元素的位置就可以用一个 2 维向量(x_0,x_1)来表示. 在正式描述我们的算法之前, 我们先给出一些基本的符号解释和定义.

定义 1. 数据空间[25]. 给定一个 n 维数组 $\boldsymbol{A}$, 该数组的数据空间 M 被定义为所有存储元素的集合. M 中数据元素$\boldsymbol{\phi}$的坐标可以表示为$\boldsymbol{\phi}=(x_0,x_1,\cdots,x_{n-1})^{\mathrm{T}}$, 对于$0\leqslant i\leqslant n$都有$x_i\in[0,d_i-1]$, 其中$d_i$表示数组在第 i 维度的宽度.

定义 2. 访存图案[25]. 由数组 $\boldsymbol{A}$ 的 m 个相邻数据元素组成的访存图案定义为$P=\{\boldsymbol{\phi}_1,\boldsymbol{\phi}_2,\cdots,\boldsymbol{\phi}_m\}$, 其中$\boldsymbol{\phi}_i=(x_0^i,x_1^i,\cdots,x_{n-1}^i)^{\mathrm{T}}$.

定义 3. 存储划分[25]. 数组的存储划分表示为一对映射函数$(f(\boldsymbol{\phi}),g(\boldsymbol{\phi}))$, 其中 $f(\boldsymbol{\phi})$计算数据映射到 bank 号, $g(\boldsymbol{\phi})$ 用于计算对应的 bank 内偏移地址.

定义 4. 有效存储划分. 数据空间为 M 的数组的有效存储划分是指找到一对映射函数$(f(\boldsymbol{\phi}),g(\boldsymbol{\phi}))$, 使得对于任意的 $\boldsymbol{\phi}_1,\boldsymbol{\phi}_2\in M$, 式(1)成立.

$$\boldsymbol{\phi}_1\neq\boldsymbol{\phi}_2\iff(f(\boldsymbol{\phi}_1),g(\boldsymbol{\phi}_1))\neq(f(\boldsymbol{\phi}_2),g(\boldsymbol{\phi}_2)).\tag{1}$$

定义 5. 基于线性变化的存储划分(LTB). LTB 是仅支持循环划分的存储划分方法. 给定一个 n 维数组的访问索引$\boldsymbol{\phi}=(x_0,x_1,\cdots,x_{n-1})^{\mathrm{T}}$, LTB 执行线性变换$\boldsymbol{\phi}\to\boldsymbol{\alpha}\cdot\boldsymbol{\phi}$, 其中$\boldsymbol{\alpha}=(\alpha_0,\alpha_1,\cdots,\alpha_{n-1})$且$\alpha_i\in\mathbb{Z}$. 假设 N 是表示 bank 数量的划分参数, 则对于 2 维数组该映射函数可以表示为

$$f(\boldsymbol{\phi})=(\boldsymbol{\alpha}\cdot\boldsymbol{\phi})\bmod N=(\alpha_0x_0+\alpha_1x_1)\bmod N,\tag{2}$$

$$g(\boldsymbol{\phi})=\left\lceil\frac{d_1}{N}\right\rceil x_0+\left\lfloor\frac{x_1}{N}\right\rfloor.\tag{3}$$

定义 6. 通用存储划分(GMP). GMP 是同时支持循环划分和块划分的存储划分方法. 假设 N 和 B 分别是划分参数和分块大小参数, 则 GMP 在 2 维数组上的映射函数可以描述为

$$f(\boldsymbol{\phi})=\left\lfloor\frac{\boldsymbol{\alpha}\cdot\boldsymbol{\phi}}{B}\right\rfloor\bmod N=\left\lfloor\frac{\alpha_0x_0+\alpha_1x_1}{B}\right\rfloor\bmod N,\tag{4}$$

$$g(\boldsymbol{\phi})=\left\lceil\frac{d_1}{N\times B}\right\rceil\times B\times x_0+\left\lfloor\frac{x_1}{N\times B}\right\rfloor\times B+x_1\bmod B.\tag{5}$$

定义 7. 基于扁平化存储划分(flatten-based memory partition, FMP). FMP 首先将一个 d 维的数据数组展平为 1 维数据数组, 然后对 1 维数据数组执行块循环划分. 对于一个 2 维数据数组, 其展平函数被描述为$x=d_1x_0+x_1$. 在 2 维数组上, FMP 的映射函数可以描述为

$$f(\boldsymbol{\phi})=\left\lfloor\frac{x}{B}\right\rfloor\bmod N,\tag{6}$$

$$g(\boldsymbol{\phi})=\left\lceil\frac{x}{N\times B}\right\rceil\times B+x\bmod B.\tag{7}$$

基于以上这些不同的策略, 存储划分的主要任务是对$(\boldsymbol{\alpha},N)$, $(\boldsymbol{\alpha},N,B)$, 或者$(N,B)$进行空间上的探索, 以在使用的 bank 数量、存储开销和寻址复杂度之间进行权衡. 由于 GMP 在上述 3 种划分策略中具有最大的搜索空间, 因此可以在大多数情况下达到最优解.

3 动机示例及问题定义

在本节中, 我们以一个例子来演示所提出的方法和其他方法的不同, 并给出在 CGRA 存储划分中需要解决的问题.

3.1 动机示例

在图 2(d)中展示了 2 维去噪算法的循环内核, 该算法包括对数组 $\boldsymbol{C}$ 的 4 个取数据操作(L1, L2, L3, L4), 其中d_0和d_1分别表示数组 $\boldsymbol{C}$ 第 0 维和第 1 维的宽度. 图 2(a)中展示了该循环内核对应的原始 DFG 在启动间隔$II=1$的情况下流水线执行的场景, 可以看到在同一个时刻, 会同时执行同一迭代中的 4 个取数操作. 图 2(b)表示数据空间中这种执行方式下的数组访存图案, 这种访存图案用 GMP 方法进行求解, 可以用 4 个 bank 进行划分, 其中参数$\boldsymbol{\alpha}=(1,3)$, $B=2$. 如图 2(c)所示, 为了实现 GMP 方法对应的划分函数$(f(\boldsymbol{\phi}),g(\boldsymbol{\phi}))$, 生成了一个具有 11 个节点的 DFG, 包含 4 个乘法、2 个除法、2 个模运算和 6 个常数. 这样的 DFG 对应的电路将会比较复杂, 同时也会导致较大的配置缓冲区. 即使将 N 和 B 的取值限制为 2 的 n 次幂(一些乘法、除法和模运算可以被简化为移位运算和与运算), 但仍然有 3 个乘法无法消除. 因此, GMP 是以复杂的地址生成为代价来得到最优或者接近最优的划分结果.

对于 CASCADE[13] 中的方法, 如图 2(e)所示, 它首先将 2 维数组平坦化为 1 维数组, 然后对 1 维数组进行块循环划分. 在这个例子中, 其划分参数 $B=2$, $N=5$, 即需要 5 个 bank. 如图 2(f)所示, 为了实现其对应的划分函数, 我们生成了一个包含 8 个节点的 DFG, 其包括 2 个乘法、2 个除法和 2 个模运算. 如果

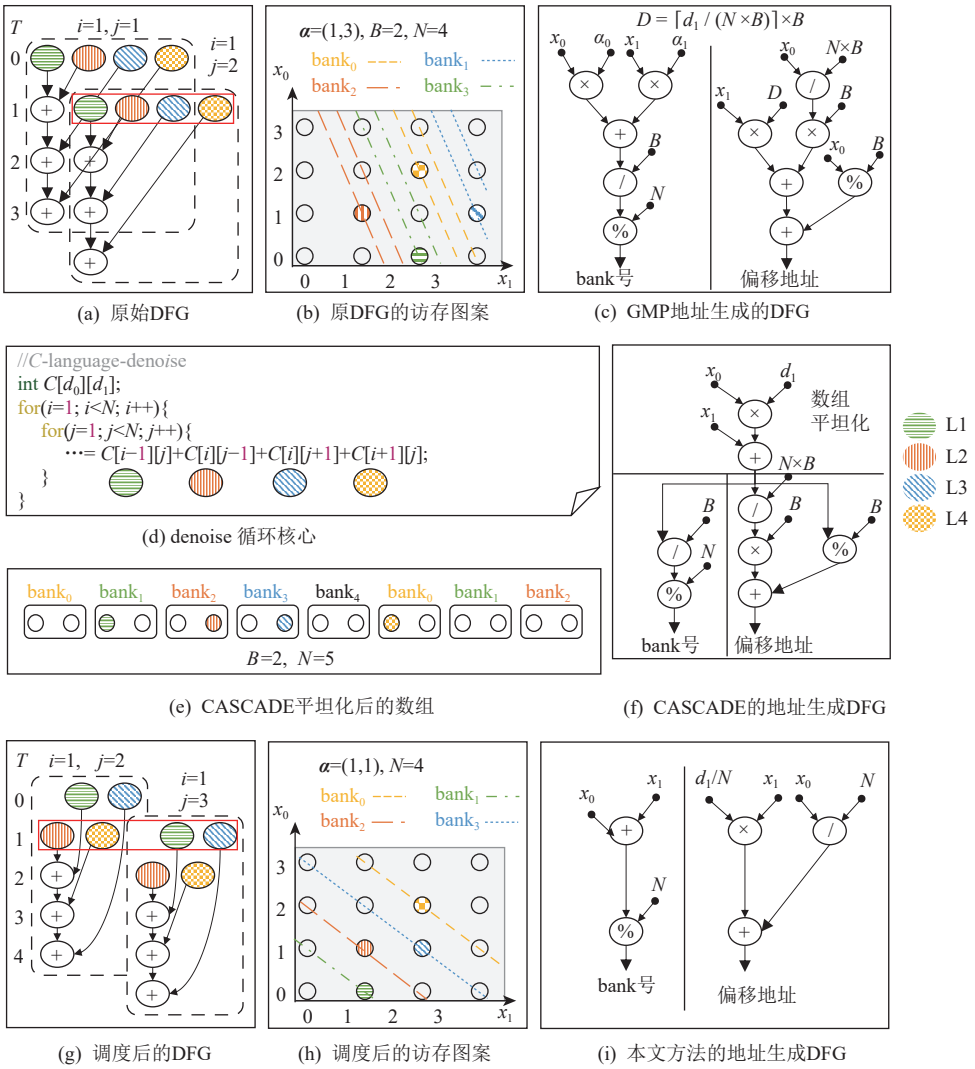


Fig. 2 The motivating example

图 2 动机示例

使用累加器[13]实现数组平坦化，并且将划分参数的取值都限制为 2 的 n 次幂，则所有的复杂运算（乘法、除法和模运算）都可以被简化为移位运算和与运算. 因此这种平坦化的方法以较低的地址生成开销实现了次优的划分结果.

以上 2 种方法都无法在较低的地址生成开销的前提下，达到最优的划分方案，所以我们引入了基于访存图案变形的存储划分方法. 如图 2(g)所示，DFG 被重新调度了（L2 和 L4 的执行时间步相较之前延后 1 个单位），来自不同迭代的访存操作将会同时执行. 重新调度之后，在数组的数据空间中，其访存图案如图 2(h)所示变成了阶梯图案，因此当 $\boldsymbol{\alpha}=(1,1)$ 时，执行循环划分可以达到 bank 数量为 4 的最优划分. 当 $\boldsymbol{\alpha}$ 固定为(1,1)时，其地址生成的 DFG 如图 2(i)所示，只需要进行 5 次操作，其中包括 1 个乘法、1 个除法和 1 个模运算. 同样地，如果将划分参数 N 的取值限制为 2 的 n 次幂，那么就只会存在 1 个复杂运算（乘法）. 这种方法相比 GMP 和 CASCADE 而言，既保证了划分结果的质量又减小了地址生成的开销. 因此基于访存图案变形的存储划分方法可以在适中的地址生成开销情况下找到最优或接近最优的划分结果.

3.2 问题定义

从 3.1 节的例子中，我们可以看到虽然 GMP 可以达到最优的解，但是其地址生成比较复杂，包含多个除法、乘法、加法运算. 而 CASCADE 虽然减小了地址生成的复杂度，但是其划分结果的质量是次优的. 这里我们将给出本文方法中关于阶梯图案及算法中重要变量的定义，并证明阶梯图案可以实现最佳划分结果，并在最后对 CGRA 存储划分问题进行定义.

定义 8. 全“1”系数超平面. 如果 LTB 中的划分

超平面的变换系数构成全“1”向量, 即$\boldsymbol{\alpha}=(1,1,\cdots,1)$, 则称之为全“1”超平面. 如图 2(h)所示, 法向量为(1,1)的虚线表示全“1”超平面. 在启动间隔为 II 的流水线执行时, 如果全“1”超平面恰好覆盖该访存图案的 II 个数据元素, 则称完全全“1”超平面. 例如在图 2(h)中, 我们令 $\boldsymbol{\alpha}=(1,1)$, 在数据空间中, 得到了若干条斜率为–1 的直线, 这些斜率为–1 的直线就组成了全“1”系数超平面.

定义 9. 阶梯图案. 给定一个由一组连续的全“1”超平面 H 覆盖的访存图案 P, 对于任意的$\boldsymbol{h}\in H$, 如果 $\boldsymbol{h}$ 穿越不超过 II 个数据元素, 则称 P 为阶梯图案. 例如在图 2(h)中, L1, L2, L3, L4 在数据空间可以被全“1”超平面覆盖, 对于每一个超平面(虚线)都只穿越了访存图案中的 1 个数据元素, 因此图 2(h)中的访存图案为可视为一个阶梯图案.

定义 10. 严格阶梯图案. 给定一个由一组连续的全“1”超平面 H 覆盖的访存图案 P, 对于除最后 1 个之外的任意$\boldsymbol{h}\in H$, 如果 $\boldsymbol{h}$ 正好穿越 II 个数据元素, 则称 P 为严格阶梯图案. 例如在图 2(h)中, 通过将 L2 和 L4 提前一步执行, 数据空间中对应的 2 个元素(竖线纹理和点状纹理)各自向右移动 1 个单位. 在 II=1 时, 4 个全“1”超平面恰好都只穿越该访存图案的 1 个元素, 此时的访存图案就称为严格阶梯图案.

定理 1. 如果 P 是一个由数组中 m 个元素构成的严格阶梯访存图案, 其对应的循环流水启动间隔为 II. 使用全“1”超平面对该访存图案进行循环划分, 可以达到最小的 bank 数量.

证明. 如定义 10, 一组连续的全“1”超平面中除最后 1 个以外, 其他超平面能够覆盖整个图案 P, 且恰好穿越 II 个元素. 因此, 需要$\lceil m/II\rceil$个连续的全“1”超平面来覆盖该图案, 即仅需要最小的 bank 数($\lceil m/II\rceil$)来实现无冲突的存储划分. 证毕.

由于全“1”超平面的系数为常数 1, 并且不涉及块划分(B=1), 其存储划分的映射函数可以简化为

$$f(\boldsymbol{\phi})=(1,1)\cdot(x_0,x_1)\bmod N=(x_0+x_1)\bmod N, \tag{8}$$

$$g(\boldsymbol{\phi})=\left\lceil\frac{d_1}{N}\right\rceil x_0+\left\lfloor\frac{x_1}{N}\right\rfloor. \tag{9}$$

定义 11. 转移矩阵. 转移矩阵 $\boldsymbol{TranA}$ 的元素 $A_{i,j}$ 表示访存图案中的第 i 个数据元素在移动 $j-m$ 步之后($j\in[0,2m]$, m 为访存图案大小), 经过全“1”超平面的划分, 该数据元素被分配的 bank 序号. 由于 j 的取值范围在 [0, 2m], 因此该转移矩阵的元素不仅能够表示数据元素的向右移动($j-m>0$), 还能表示数据元素的向左移动($j-m<0$). 为减少搜索空间, 我们约定数据元素向左移动不能超过原始访存图案中最左侧的元素点. 对于不满足该约定的移动, 其对应转移矩阵元素取值为–1.

针对如图 2(b)中的原始访存图案, 我们可以得到如下矩阵:

$$\boldsymbol{TranA}=\begin{pmatrix}-1&-1&-1&1&2&3&0&1&2\\-1&-1&-1&-1&2&3&0&1&2\\-1&-1&2&3&0&1&2&3&0\\-1&-1&-1&3&0&1&2&3&0\end{pmatrix}.$$

由于 L2 为该图案中的最左侧元素, 其他几个元素向左移动后的位置不能超过 L2 的位置. 由于该限制, L1 最多左移 1 步, 转移矩阵中第 0 行的第 1 个非“–1”值出现在 j=3($j-m$=3–4=–1)的列中. 当 L1 左移 1 步时, 会被划分到第 1 个 bank, 因此 $TranA$[0][3]取值为 1.

如动机示例所示, 对内存访问操作的算子进行调度可以实现访存图案变形, 并且通过严格阶梯图案可以达到最优存储划分. 因此我们将给出基于访存图案变形的存储划分问题的定义:

问题 1. 给定循环内核的流水执行启动间隔 II, 其访存图案 P 包含 m 个元素, 通过对该循环 DFG 的算子调度来实现访存图案变形使得:

1)在同一个全“1”超平面上的访存算子必须被调度到不同的控制步骤;

2)在调度形成的新访存图案中, 最大化能够覆盖 II 个访存算子的全“1”超平面的数量.

对于第 1 项而言, 由于同一个超平面上的元素将会存放在同一个 bank 中, 因此需要将这些元素的访问操作分开调度, 从而避免访存冲突. 对于第 2 项来说, 越最大化覆盖 II 个元素的全“1”超平面的数量意味着全“1”超平面的数量就越少, 即所需要的 bank 数量越少.

4 基于回溯的访存图案变形算法

由 3.1 节的动机示例可知, 访存算子的调度可以改变访存图案. 本节详细介绍通过算子调度算法来改变访存图案实现存储划分. 该算法分为 2 个步骤, 首先根据定义 11 计算出该访存图案的转移矩阵, 然后据此转移矩阵进行回溯搜索得到最终的调度结果.

如算法 1 所示, 我们首先对各个变量进行初始化, 然后计算出访存图案 P 的转移矩阵 $\boldsymbol{TranA}$. 在得到转移矩阵之后, 通过 while 循环来寻找在当前 II 下的解, 如果当前 II 下无法找到可用的解, 那么就增大 II 继

续寻找，直到找到解. 在算法 1 的行⑩~⑯，我们根据访存图案中每个元素的移动距离、最小距离（*minDis*）和 *II* 来确定该元素对应的控制步. 确定了所有的访存算子的控制步之后，利用整数线性规划[26]对剩下的非访存算子进行调度，从而生成最终的调度结果 ***CS***.

算法 1. 基于回溯的访存图案变形算法.

输入：访存图案 *P*，访问图案元素个数 *M*，bank 最大数量 *N*，DFG 算子集合 *OP*；

输出：使用 bank 数量 N_{used}，调度结果 ***CS***.

① ***bankInfo*** ←$N\times M$ *zeros*;
② *minDis*,*selected_values* ← 0;
③ *dis*[*N*] ← *INT_MIN*;
④ ***TranA*** = *get_transfer_matrix*(*P*);
⑤ *II*= max(⌈|*P*|⌉/*N*,⌈|*OP*|⌉/*PEA*);
⑥ while(! *find_solution*(*row*=0,*II*))
⑦ *II* = *II* + 1;
⑧ end while
⑨ *minDis* = min(*dis*);
⑩ for each *bank_i* in ***bankInfo***
⑪ *step* ← 0;
⑫ for each *p* in *bank_i*
⑬ *CS*[*p*] ← (*dis*[*p*]−*minDis*) *II*+*step*;
⑭ *step* ← *step* + 1;
⑮ end for
⑯ end for
⑰ *ILPScheduling*(*otherOperators*);
⑱ N_{used}←*getUseBanks*(***bankInfo***).

算法 2 通过回溯的方式来寻找转移矩阵 ***TranA*** 中符合条件的值. 首先从转移矩阵的第 0 行开始（即访存图案中的第 1 个元素），在第 0 行中搜索值不为 −1 且不会造成启动间隔大于给定 *II* 的值. 当在第 0 行中找到符合条件的值后，就更新 bank 分配列表和元素移动信息列表，再遍历 ***TranA*** 的下一行. 如果下一行找不到满足要求的值，那么就回溯到之前的状态. 当遍历完转移矩阵的所有行之后，就可以得到满足要求的 bank 分配列表和元素移动信息.

算法 2. 函数 *find_solution*.

输入：当前待处理行 *row*，启动间隔 *II*；

输出：bank 分配信息 ***bankInfo***，元素移动距离 *dis*.

① if *row* == *M*
② return *selected_values* == *M*;
③ for *col* from 0 to 2*M*
④ *value* ← *TranA*[*row*][*col*];
⑤ if *value* ≠−1 && ***bankInfo***[*value*]. *size* < *II*
⑥ ***bankInfo***[*value*]. *add*(*row*);
⑦ *dis[row]* ← *col*−*M*;
⑧ if (*find_solution*(*row*+1, *II*))
⑨ return true;
⑩ end if
⑪ ***bankInfo***[*value*]. *remove*(*row*);
⑫ *dis*[*row*] ← *INT_MIN*;
⑬ end if
⑭ end for
⑮ end if
⑯ return false.

5 硬件架构以及编译流程

在本节中，我们首先介绍了适配以上存储划分结果的高效 CGRA 架构，然后再给出了总体的编译流程.

5.1 高效 CGRA 架构

在传统 CGRA 的基础之上，我们首先为所有的 LSU 单元添加了共享的时钟分频器（divider, DIV），然后对每个 LSU 单元的内部结构进行了精简化设计. 由于在模调度中仅在每 *II* 个周期执行 1 次特定的存储器访问操作，因此 DIV 模块仅在每 *II* 个周期向特定 LSU 产生使能信号. 如图 3（d）所示，时钟分频器由累加器（Acc）和 II_{max} 个比较器组成，其中 II_{max} 表示硬件允许的最大循环流水 *II*. 由于 LSU 上的存储器访问操作在高度为 *II* 的时间窗口中任意一个时间步执行，因此比较器可以用于为工作在不同控制步的 LSU 生成 *II* 个使能信号.

在图 3（b）中列出了精简的 LSU 的结构. 它主要由加载寄存器（load register, LDR）、存储寄存器（store register, STR）、地址生成器（address generator, AG）和配置缓冲区组成. 精简之后的 AG 如图 3（e）所示，AG 首先接收来自时钟分频器的信号来启用累加器ACC_{x_1}，通过给定步长 $stride_1$ 和初始值 $start_1$ 来生成内存访问坐标的低维分量 x_1. 如果 x_1 达到最大值，则ACC_{x_1}将重置为 $start_1$. 同时，启用另一个累加器ACC_{x_0}，使用步长 $stride_0$ 和初始值 $start_0$ 生成内存访问坐标的高维分量 x_0. 然后，根据式（8）（9），使用 2 个加法器、1 个乘法器、1 个移位器和 1 个与门来计算产生 bank 号和偏移地址. 为了简化电路，bank 数量 *N* 的取值也仅限于 2 的 *n* 次幂. 因此，除法操作被简化为右移位器，求模运算简化为与运算. 在这样的架构实现中，由于存储划分超平面的系数为常数 1，其地址生成的开销被大幅度降低了.

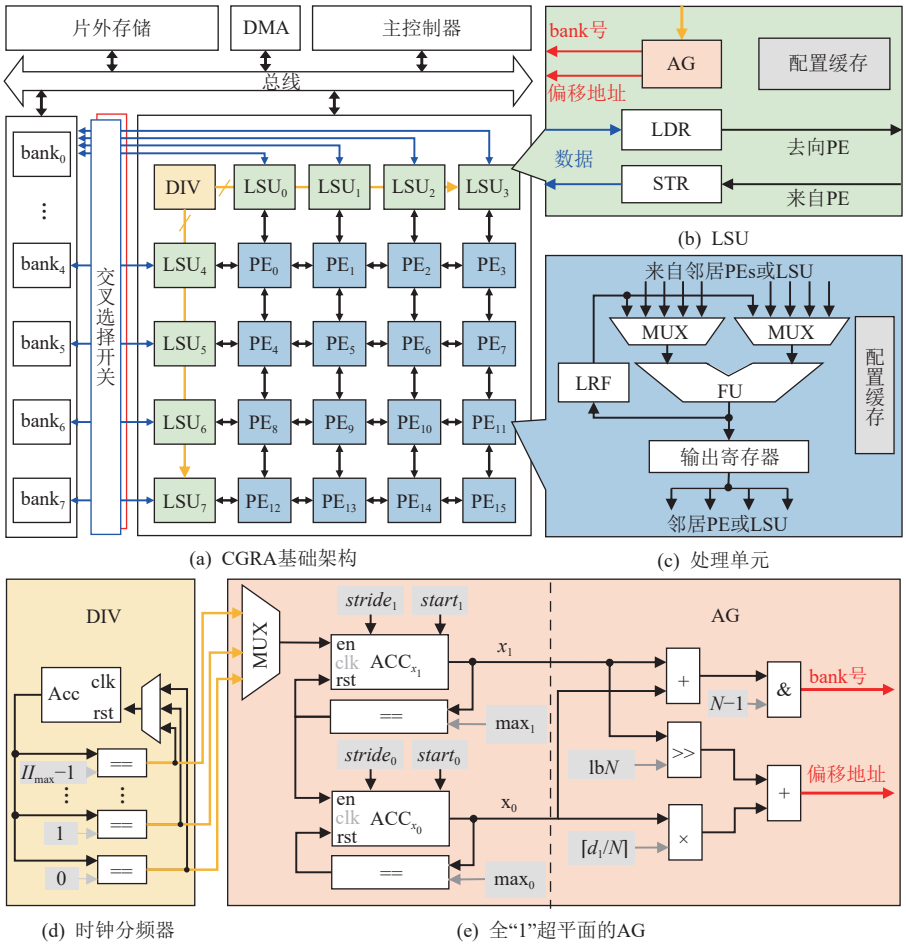


Fig. 3 Our proposed hardware architecture

图 3 本文提出的硬件架构

5.2 整体编译流程

图 4 展示了我们编译框架的总体流程. 首先, 使用 LLVM[27] 前端工具将输入循环内核转换为中间表式(IR); 然后基于 IR 生成 DFG, 其中记录着相应的访存图案; 接下来调用算法 1 执行访存图案变形, 并生成调度结果和划分结果; 最后基于调度结果执行基于最大团算法[10] 的 DFG 映射, 并生成配置信息.

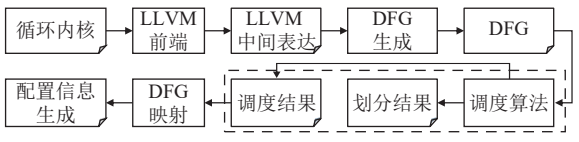


Fig. 4 Overall flow of compilation

图 4 整体编译流程

6 实验结果

在本节中, 为了评估我们提出的方法, 我们进行了一系列实验并比较了在硬件资源无约束情况和有约束情况下的结果. 在硬件资源无约束的情况下, 存储划分可以达到最优解. 在资源受限情况下, 存储划分只能在目标架构中找到次优解.

6.1 实验设置

由于 FMP[13] 和 GMP[17] 这 2 种存储划分方法与我们所提出的方法都是基于线性变换, 并且不像其他方法(例如基于图[21] 或者基于晶格[19] 的存储划分)需要额外的存储空间, 我们采用 FMP 和 GMP 作为主要的比较对象. FMP 将多维数组平坦化成 1 维数组进行存储划分, 并采用 CASCADE[13] 简化的 LSU 进行地址生成. 由于 FMP 方法对数组的宽度敏感, 因此在执行 FMP 时, 我们在第 1 维上提供了 d_1=30, 32, 34 这 3 种可变宽度. 为简单起见, 我们使用 FMP-30, FMP-32, FMP-34 分别表示 d_1=30, 32, 34 时 FMP 的实验结果. GMP 直接针对多维数组进行线性划分, 需要如图 2(c)所示的复杂地址生成单元来支持复杂参数的线性划分. 我们提出的基于访存图案变形的存储划分方法 PMM(access pattern-morphing-based memory partitioning)首先采用算法 1 进行存储划分, 并使用如图 3(e)所示的精简地址生成器来实现全"1"系数的

存储划分. 为了屏蔽硬件差异带来的影响, 我们还将算法 1 的存储划分的结果映射到 GMP 的目标硬件架构上(简称为 PMM*)进行比较.

为了公平比较, 所有架构都包括一个由 8 个 bank 组成的 4 KB 存储空间以及 4×4 的 PE 阵列, 每个 PE 中的 LRF 包含 4 个寄存器, 支持多周期数据依赖路由. 为了硬件的高效实现, 约定所有方法的划分参数 N 和块大小 B 的取值均为 2 的 n 次幂. 因此, 一些乘法、除法和模运算可以简化为移位器和逻辑门. 所有方法的相应编译均采用 RAMP[10] 进行后端映射, 该映射算法运行在频率为 2.1 GHz、内存为 64 GB 的 Intel 16 核 CPU 上.

实验中所有用例都来自真实应用[28](包括医学图像处理和 H.264 运动补偿等)中的 12 个循环内核. 如图 5 所示, 这些内核在 2 维数据中具有不同的访存图案, 其中访问元素的数量范围为 4~25.

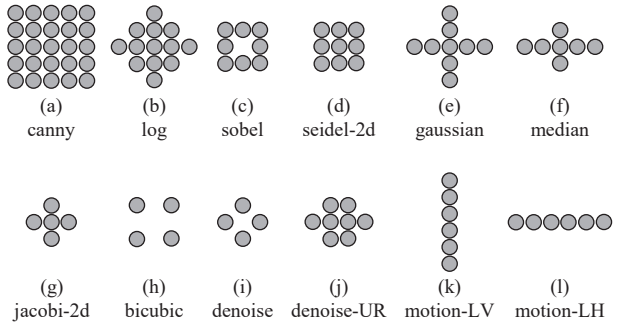


Fig. 5 Access patterns of the twelve benchmarks

图 5 12 个基准访存图案

6.2 无约束情况下的比较

为了验证 FMP, GMP, PMM*这 3 种方法的存储划分质量, 我们首先在无资源约束的情况下进行了实验, 其可用 bank 的数量是无限的, 参数 N 和 B 可以是任何正整数. 在这种情况下, 由于所有的存储器访问操作都可以在单个周期内执行, 它相当于 II=1 的完全流水执行.

图 6 展示了在无约束情况下各个方法占用的 bank 数量情况. 平均而言, FMP-30, FMP-32, FMP-34, GMP 和本文算法 PMM*分别使用 9.83, 10.33, 10.58, 9.00, 8.67 个 bank. 与 GMP 相比, PMM*可以在所有 12 个内核上达到最少的 bank 数量, 而 GMP 无法在 sobel 和 median 这 2 个例子中上求得最优划分结果. 在这 2 个例子中, 访存图案的几何形状是造成 GMP 失败的主要原因. 对于有 sobel 而言(具有 8 个访问元素, N 初始化为 8), 按照文献 [17] 中的方法遍历整个搜索空间后, GMP 仍然无法找到无冲突的解决方案. 于是, N 将会被增加到 9, 最终得到一个无冲突的解. 对于 median 这个用例, GMP 必须将 N 增加到 10, 以获得无冲突的解. 利用访存图案变形, PMM*可以形成存储划分友好的图案, 从而使这些内核占用的 bank 数量最少. 对于 FMP, 对于初始化的 N, 它在展平数组上的搜索空间非常有限, 并且容易出现访存冲突, 因此它更容易得到次优解. 从图 6 中我们还注意到 FMP 的划分结果对数据数组的维度宽度非常敏感, 而 GMP 和 PMM*对不同宽度的数据都很稳定.

此外, 为了能够得到正确的 bank 内地址, 在划分之后需要对数组进行填充, 将低维度的数组宽度扩充至 $N\times B$ 的整数倍. 图 7 展示了具有不同宽度(d_1)下 3 种方法的填充数据大小. 当 d_1=30 时, 平均而言, PMM*的填充大小分别是 FMP 和 GMP 的 86.7% 和 98.1%. 随着宽度的增加, PMM*仍然优于 FMP 和 GMP. 由于我们使用的是循环划分(即 B=1), 并且 N 可以达到最小值, 所以 PMM*更容易使得 $N\times B$ 的值较小. 因此, 平均而言, PMM*只需要较少的数据进行填充, 可以进一步减少内存的开销.

6.3 硬件频率、面积、功耗的比较

为了比较硬件开销, 我们首先使用 Verilog HDL

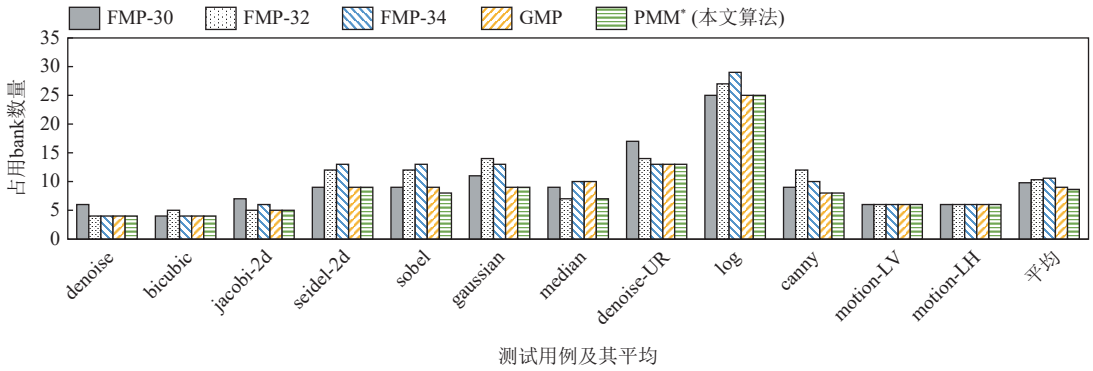
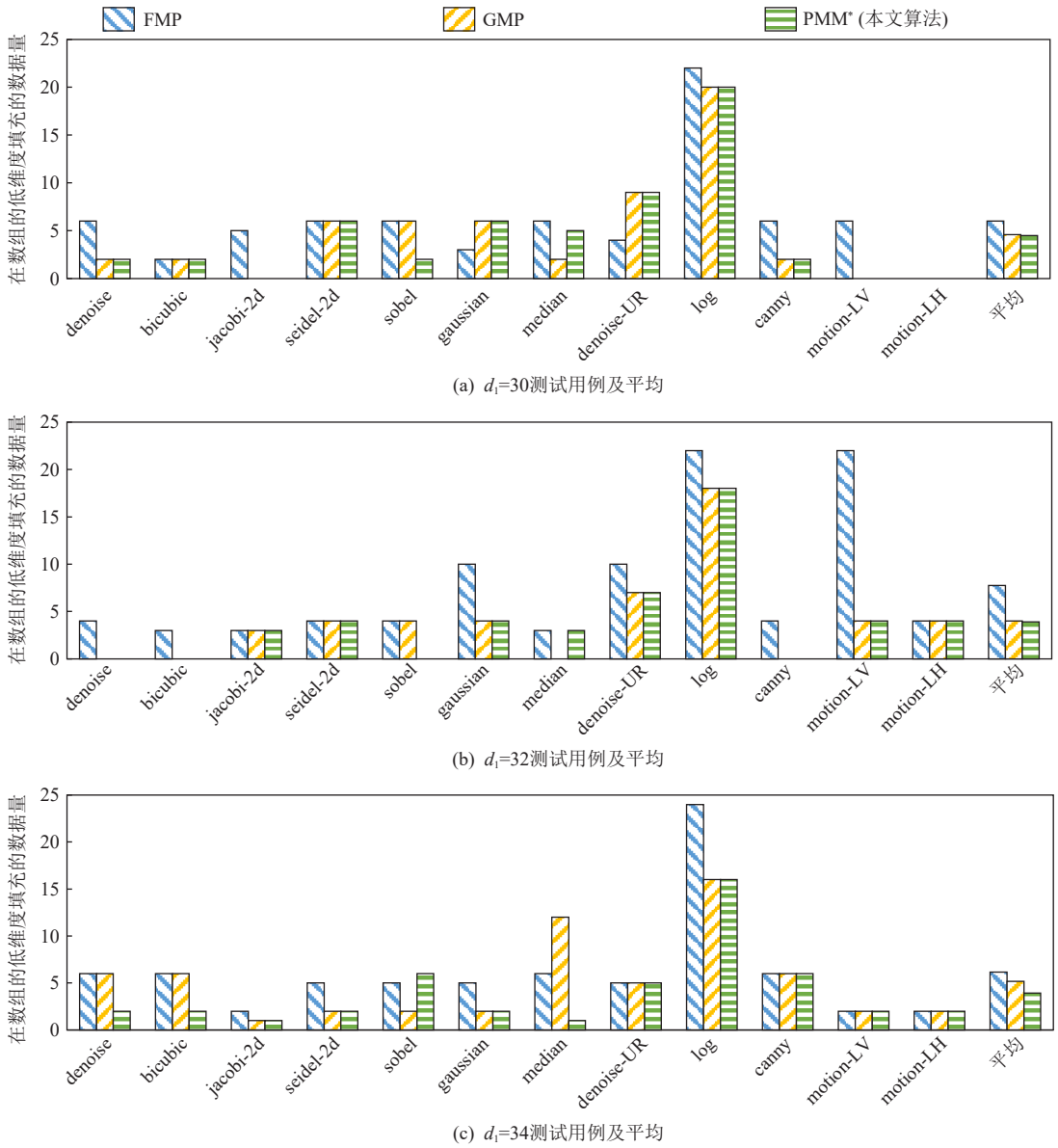


Fig. 6 Number of used banks in the unconstrained situation

图 6 无约束条件下的 bank 使用数量



(a) d_1=30测试用例及平均

(b) d_1=32测试用例及平均

(c) d_1=34测试用例及平均

Fig. 7 Data padding size comparison over different d_1 values

图 7 不同 d_1 取值下的数据填充大小比较

分别实现了 GMP 的访存-计算解耦的 CGRA 架构、面向 FMP 的 CASCADE 架构和本文提出的如图 3 所示的 CGRA 架构；然后使用 Synopsys Design Compiler 在 FreePDK 45 nm 工艺库上评估了它们的频率、面积和功耗；最后使用 Cadence innovus 完成布局布线. 这 3 种架构之间的主要区别在于 LSU 中的 AG 不同. 对于面向 GMP 的访存-计算解耦的 CGRA 架构，其 AG 实现根据式(4)(5).

如表 1 所示，CASCADE 和 CGRA 具有相同的频率，而 GMP 架构的频率较低. 这是因为 GMP 的 CGRA 架构中复杂的 AG 成为关键路径. 与 CASCADE 相比，CGRA 的面积为 0.549 mm^2，平均功耗为 120.523 mW，面积仅增加了 5.78%，功耗仅增加了 8.48%. 由于 GMP 的目标架构在 AG 中使用了更多的乘法器和用于超平面参数的配置缓存，GMP 的目标架构在面积和功耗方面都有更大的开销.

Table 1 Comparison of Frequency, Area and Power

表 1 频率、面积、功耗的比较

架构	频率/MHz	面积/mm^2	功耗/mW
GMP-解耦架构	452.49	0.607	137.128
CASCADE	462.96	0.519	111.101
PMM（本文架构）	462.96	0.549	120.523

6.4 资源有限情况下的比较

当将 12 个内核映射到 CGRA 时，存储划分受到最大 bank 数量($N\leqslant 8$)和 N 的取值为 2 的 n 次幂的约

束. 图 8 展示了 12 个内核的归一化的流水执行性能($II/II_{\min}$)和归一化的吞吐量(频率/II_{mapped})的比较，其中 $II_{\min}$ 是理论上能达到的最小 II，II_{mapped} 是实际映射成功的 II. 为了简单起见，该图仅显示了 d_1=30, 32, 34 时的 FMP 平均值. 如图 8 所示，相比其他 2 种方法，PMM*由于屏蔽硬件差异，在所有测试用例中均达到了最好的流水执行性能. 由于 FMP 需要更多的 bank 完成存储划分，在资源有限的情况下，其流水执行性能最差. 对于 GMP 而言，在大多数测试用例中，它都表现较好. 但是在 sobel 和 median 这 2 个测试用例中，由于其无法做到最优存储划分，因此需要更大的 II 才能在 8 个 bank 的限制下完成无冲突访存，从而造成其流水执行性能下降. 平均而言，PMM*相比于 FMP 和 GMP 在流水执行性能上分别可以获得平均 1.54 倍和 1.09 倍的加速比. 再考虑到算法和硬件的综合影响，PMM 的目标硬件相比于 GMP 的目标硬件具有更高的频率. 因此，如图 8 所示，经过算法和硬件的联合优化，相比于 FMP, GMP, PMM*, PMM 在吞吐量上分别获得平均 1.74 倍、1.15 倍、1.02 倍的加速比.

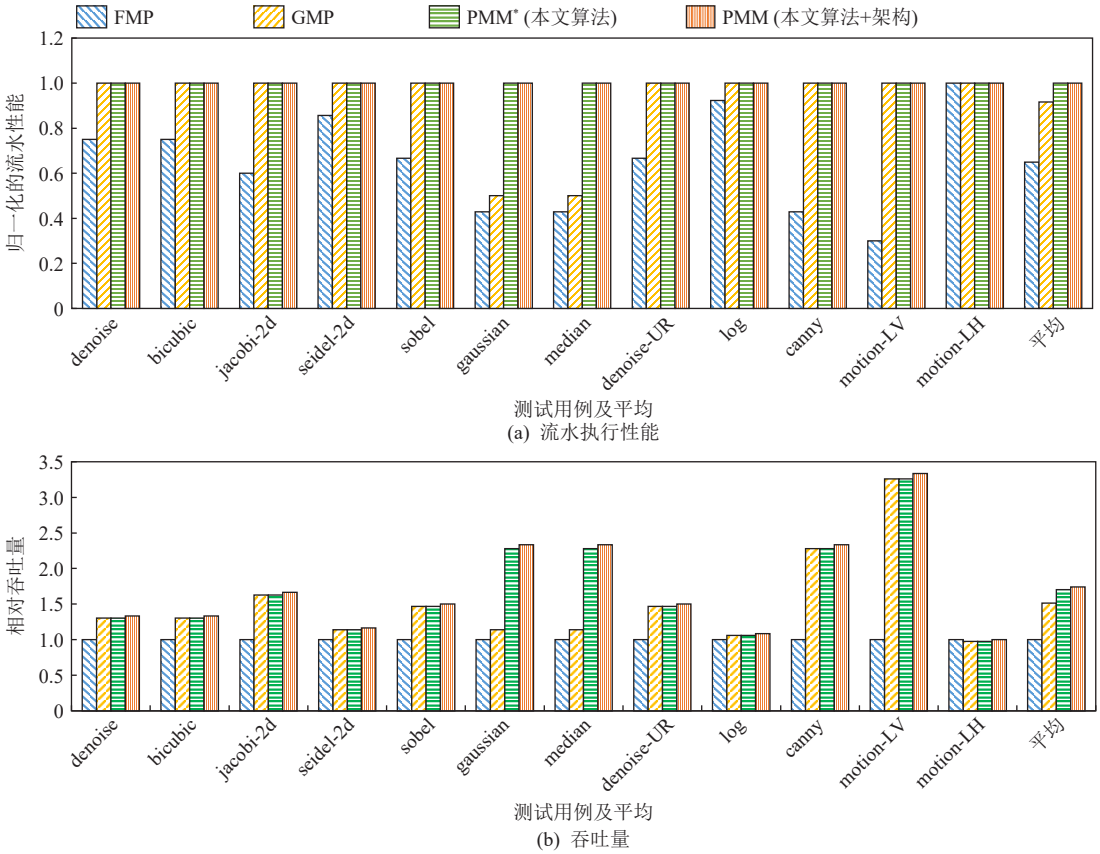


Fig. 8 Comparison of the mapped II and throughput

图 8 映射的 II 和吞吐量的比较

6.5 能量效率的比较

为了考虑包括功耗、频率、流水性能等因素的综合影响，我们进一步采用能效指标(吞吐量/功耗)进行比较. 如图 9 所示，与 FMP 和 GMP 相比，PMM 可以分别实现平均 1.67 倍和 1.25 倍的能效提升. 在 motion-LH 用例中，由于 FMP 的硬件功耗最低，且这 3 种方法在这个用例中的映射 II 均为 1. 因此，FMP 在 motion-LH 这一用例中达到了最高的能效. 在其他的用例中，由于 FMP 得到的映射 II 均高于其他 2 种方法，因此 FMP 的能效较低. 而对于 GMP 而言，虽然其映射 II 基本接近理论最小值，但是由于其硬件结构的能耗较大，因此其能效相对较低. 屏蔽硬件影响之后，我们发现 PMM*相比于 FMP 和 GMP 依然有平均 1.51 和 1.14 倍的能效加速. 因此，相比于 FMP 和 GMP, PMM*分别贡献了 76% 和 56% 力量，而硬件优化分别贡献了 24% 和 44% 的力量.

6.6 编译时间的比较

图 10 给出了 FMP, GMP, PMM 在存储划分算法上的编译运行时间比较. 其中 FMP 为 $d_1 = 32$时的编译时间. 由于不同测试用例的耗时差异较大，为方便

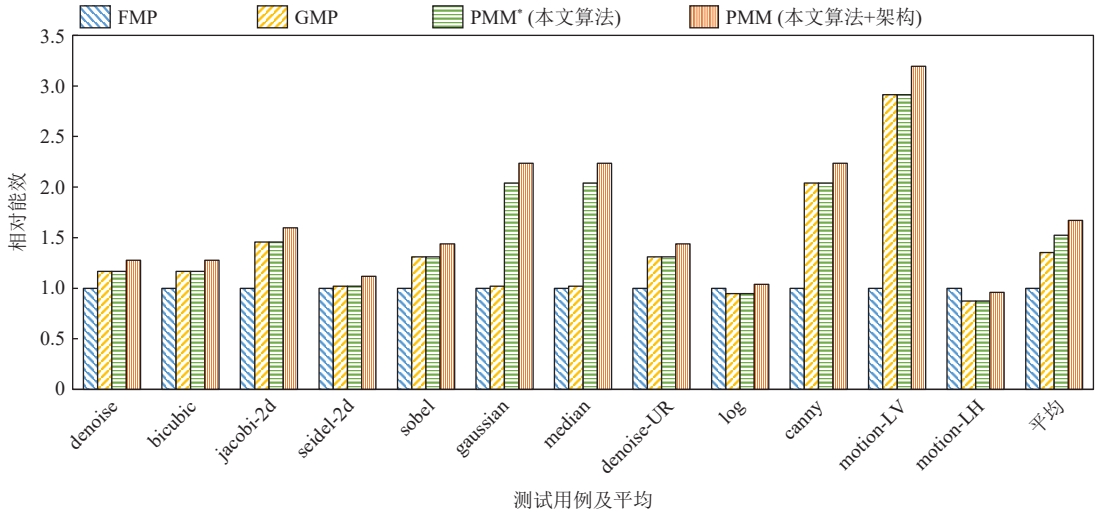


Fig. 9 Energy efficiency comparison

图 9 能效的比较

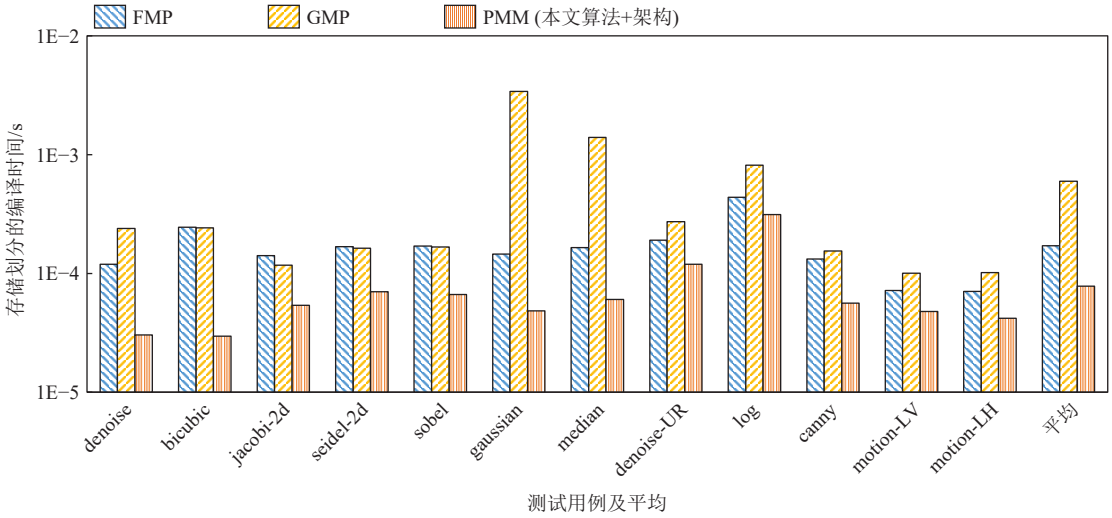


Fig. 10 Compilation time comparison of storage division (d_1=32)

图 10 存储划分编译时间的比较（d_1=32）

展示, 我们选取了对数坐标轴. 由于 PMM 在图案变形之后仅需全“1”超平面, 不需要穿越整个迭代空间去寻找划分超平面, 也不需要对数组进行平坦化, 所以用于超平面的搜索时间较少. 另外, 虽然我们的图案变形使用了回溯算法, 但是该算法仅运行在问题规模并不大的访存图案上. 因此, PMM 的平均编译时间仅为 GMP 和 FMP 的 13.07% 和 45.57%.

由于存储划分的复杂性远低于 DFG 后端映射[10], 因此不同方法在总编译时间上并没有明显的差距. 如图 11 所示, PMM 和 GMP 获得了几乎相同的编译时间. 这是因为这 2 种方法在所有内核都获得了相同的 *II*, 因此映射算法几乎具有相同的复杂度. 至于 FMP, 对于某些测试用例它的存储划分结果需要更大的 *II*. 由于更大的 *II* 下硬件资源约束更加宽松, FMP 的后端映射更容易找到合法的映射结果. 平均而言, PMM 在总体编译时间上相比于 FMP 和 GMP 具有平均 1.92 倍和 1.02 倍的时间消耗.

7 结 论

本文提出了一种针对 2 维数组的存储划分和 CGRA 硬件的协同优化方法. 在算法上, 我们通过访存图案的变形不仅能找到(近)全局最优的划分结果, 还能简化用以存储划分的超平面的参数. 在硬件架构上, 我们根据简化的超平面参数设计了精简的地址生成单元. 实验结果表明本文方法不仅能取得较好的存储划分方案, 并且能够减少硬件开销, 最终能够提升整体能效. 目前本文方法只能针对 2 维数组进

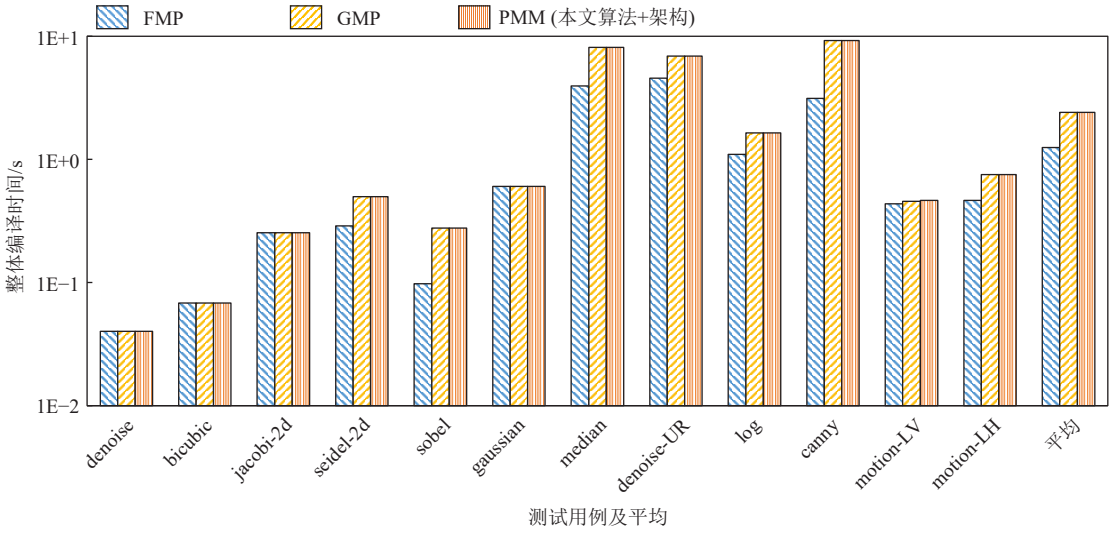


Fig. 11 Comparison of total compilation time

图 11 总体编译时间的比较

行存储划分. 未来, 我们将把存储划分方法扩展至高维数组.

作者贡献声明: 潘德财提出了求解算法、实验设计, 分析数据并撰写论文; 牟迪协助部分实验设计; 尚家兴对论文撰写提出指导意见; 刘大江提出研究思路并对论文的实验和撰写提出指导意见.

参 考 文 献

[1] Prabhakar R, Zhang Yaqi, Koeplinger D, et al. Plasticine: A reconfigurable architecture for parallel patterns[C]//Proc of the 44th Annual Int Symp on Computer Architecture. New York: ACM, 2017: 389−402

[2] Nguyen Q M, Sanchez D. Fifer: Practical acceleration of irregular applications on reconfigurable architectures[C]//Proc of the 54th Annual IEEE/ACM Int Symp on Microarchitecture. New York: ACM, 2021: 1064−1077

[3] Tan Chen, Agostini N B, Geng Tong, et al. DRIPS: Dynamic rebalancing of pipelined streaming applications on CGRAs[C]//Proc of the 28th Int Symp on High-Performance Computer Architecture (HPCA). Piscataway, NJ: IEEE, 2022: 304−316

[4] Charitopoulos G, Vatsolakis C, Chrysos G, et al. A decoupled access-execute architecture for reconfigurable accelerators[C]//Proc of the 15th ACM Int Conf on Computing Frontiers. New York: ACM, 2018: 244−247

[5] Ciricescu S, Essick R, Lucas B, et al. The reconfigurable streaming vector processor (RSVP™)[C]//Proc of the 36th Annual Int Symp on Microarchitecture. Piscataway, NJ: IEEE, 2003: 141−150

[6] Farahini N, Hemani A, Sohofi H, et al. Parallel distributed scalable runtime address generation scheme for a coarse grain reconfigurable computation and storage fabric[J]. Microprocessors and Microsystems, 2014, 38(8): 788−802

[7] Ho C H, Kim S J, Sankaralingam K. Efficient execution of memory access phases using dataflow specialization[C]//Proc of the 42nd Annual Int Symp on Computer Architecture. New York: ACM, 2015: 118−130

[8] Nowatzki T, Gangadhar V, Ardalani N, et al. Stream-dataflow acceleration[C]//Proc of the 44th Annual Int Symp on Computer Architecture. New York: ACM, 2017: 416−429

[9] Smith J E. Decoupled access/execute computer architectures[J]. ACM SIGARCH Computer Architecture News, 1982, 10(3): 112−119

[10] Dave S, Balasubramanian M, Shrivastava A. RAMP: Resource-aware mapping for CGRAs[C/OL]//Proc of the 55th Annual Design Automation Conf. New York: ACM, 2018[2023-12-01]. https://ieeexplore.ieee.org/document/8465892

[11] Chatterjee S, Gilbert J R, Long F J E, et al. Generating local addresses and communication sets for data-parallel programs[J]. ACM SIGPLAN Notices, 1993, 28(7): 149−158

[12] Wang Yuxing, Li Peng, Zhang Peng, et al. Memory partitioning for multidimensional arrays in high-level synthesis[C/OL]//Proc of the 50th Annual Design Automation Conf. Piscataway, NJ: IEEE, 2013[2023-12-01]. https://ieeexplore.ieee.org/document/6560605

[13] Wijerathne D, Li Zhaoying, Karunarathne M, et al. Cascade: High throughput data streaming via decoupled access-execute CGRA[J]. ACM Transactions on Embedded Computing Systems, 2019, 18(5s): 50: 1−50: 27

[14] Canis A, Choi J, Aldham M, et al. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems[J]. ACM Transactions on Embedded Computing Systems, 2013, 13(2): 24: 1−24: 27

[15] Cong J, Jiang Wei, Liu Bin, et al. Automatic memory partitioning and scheduling for throughput and power optimization[C]//Proc of the 2009 Int Conf on Computer-Aided Design. New York: IEEE/ACM, 2009: 697−704

[16] Li Peng, Wang Yuxin, Zhang Peng, et al. Memory partitioning and scheduling co-optimization in behavioral synthesis[C]//Proc of the 2012 Int Conf on Computer-Aided Design. New York: ACM, 2012: 488−495

[17] Wang Yuxin, Li Peng, Cong J. Theory and algorithm for generalized memory partitioning in high-level synthesis[C]//Proc of the 22nd ACM/SIGDA Int Symp on Field-Programmable Gate Arrays. New York: ACM, 2014: 199−208

[18] Liu Binbin, Yang Fan, Zhou Dian, et al. An efficient memory partitioning approach for multi-pattern data access in STT-RAM[C/OL]//Proc of the 2020 Int Symp on Circuits and Systems. Piscataway, NJ: IEEE, 2020[2023-12-01]. https://ieeexplore.ieee.org/document/9181278

[19] Escobedo J, Lin Mingjie. Tessellation-based multi-block memory mapping scheme for high-level synthesis with FPGA[C]//Proc of the 24th Int Conf on Field-Programmable Technology. Piscataway, NJ: IEEE, 2016: 125−132

[20] Zhou Yuan, Al-Hawaj K M, Zhang Zhiru. A new approach to automatic memory banking using trace-based address mining[C]//Proc of the 25th ACM/SIGDA Int Symp on Field-Programmable Gate Arrays. New York: ACM, 2017: 179−188

[21] Escobedo J, Lin Mingjie. Graph-theoretically optimal memory banking for stencil-based computing kernels[C]//Proc of the 26th ACM/SIGDA Int Symp on Field-Programmable Gate Arrays. New York: ACM, 2018: 199−208

[22] Yin Shouyi, Xie Zhicong, Meng Chenyue, et al. Memory partitioning for parallel multipattern data access in multiple data arrays[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2017, 37(2): 431−444

[23] Yin Shouyi, Yao Xianqing, Liu Dajiang, et al. Memory-aware loop mapping on coarse-grained reconfigurable architectures[J]. IEEE Transactions on Very Large Scale Integration Systems, 2015, 24(5): 1895−1908

[24] Yin Shouyi, Yao Xianqing, Lu Tianyi, et al. Conflict-free loop mapping for coarse-grained reconfigurable architecture with multi-bank memory[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 28(9): 2471−2485

[25] Yin Shouyi, Xie Zhicong, Meng Chenyue, et al. Multibank memory optimization for parallel data access in multiple data arrays[C/OL]//Proc of the 2016 IEEE/ACM Int Conf on Computer-Aided Design. Piscataway, NJ: IEEE, 2016[2023-12-01]. https://ieeexplore.ieee.org/document/7827609

[26] Hwang C T, Lee J H, Hsu Y C. A formal approach to the scheduling problem in high level synthesis[J]. IEEE Transactions on Computer-Aided Design, 1991, 10(4): 464−475

[27] Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation[C]//Proc of the 2004 Int Symp on Code Generation and Optimization. Piscataway, NJ: IEEE, 2004: 75−86

[28] Cong J, Sarkar V, Reinman G, et al. Customizable domain-specific computing[J]. IEEE Design & Test of Computers, 2010, 28(2): 6−15



Pan Decai, born in 1999. Master. His main research interests include reconfigurable computing and compiler optimization.

潘德财, 1999 年生. 硕士. 主要研究方向为可重构计算、编译器优化.



Mou Di, born in 1999. Master candidate. His main research interest includes high performance hardware accelerator in deep neural network.

牟　迪, 1999 年生. 硕士研究生. 主要研究方向为高性能神经网络硬件加速器.



Shang Jiaxing, born in 1987. PhD, professor, PhD supervisor. Member of CCF and IEEE. His main research interests include social network analysis and industry big data mining.

尚家兴, 1987 年生. 博士，教授，博士生导师. CCF 会员，IEEE 会员. 主要研究方向为社交网络分析、行业大数据挖掘.



Liu Dajiang, born in 1986. PhD, associate professor, PhD supervisor. Member of CCF and IEEE. His main research interests include reconfigurable computing, domain-specific accelerator, and compiler optimization.

刘大江, 1986 年生. 博士，副教授，博士生导师. CCF 会员，IEEE 会员. 主要研究方向为可重构计算、领域定制加速器、编译优化.