

学校代码： 10246

復旦大學

博士 学位 论 文 (学术学位)

高能效近似乘法器设计及综合研究

High-Efficient Approximate Multiplier Design and Synthesis

编 号： 19112020039

专 业： 微电子学与固体电子学

院 系： 微电子学院

完 成 日 期： 2024 年 03 月 20 日

目 录

插图目录	v
表格目录	ix
摘要	xi
Abstract	xiii
第 1 章 绪论	1
1.1 研究背景与意义	1
1.1.1 半导体工艺的发展	1
1.1.2 计算机体系结构的发展	2
1.1.3 后摩尔时代的技术路线	6
1.1.4 近似计算的优势	7
1.2 本文主要工作及组织结构	8
第 2 章 乘法器概述	11
2.1 精确乘法器	11
2.1.1 部分积的生成	12
2.1.2 部分积的累加	20
2.1.3 最终相加	24
2.2 对数乘法器	31
2.3 近似电路的误差指标	34
2.4 本章小结	35
第 3 章 考虑输入分布和极性的 ASIC 近似乘法器设计	37
3.1 研究背景	37
3.2 国内外研究现状	37
3.2.1 手工设计	37
3.2.2 数学转换近似	40
3.2.3 自动化方法	41

3.2.4 近似电路综合	43
3.3 研究动机	44
3.3.1 输入分布对近似乘法器精度的影响	44
3.3.2 输入极性对近似乘法器精度的影响	45
3.4 研究内容与创新点	47
3.5 研究方法	48
3.5.1 无符号乘法器	48
3.5.2 有符号乘法器	49
3.5.3 自动化求解	50
3.5.4 求解过程	52
3.5.5 基于 8 比特无符号数量化的 DNN 推断精度评估工具	55
3.6 实验结果	57
3.6.1 均匀分布下的 8 比特无符号乘法器	57
3.6.2 基于 8 比特无符号数的不同规模的 DNN 应用	59
3.6.3 基于 16 比特补码有符号定点数的自适应 FIR 滤波器	68
3.6.4 半正态分布下的无符号 32 比特乘法器	71
3.7 本章小节	72
第 4 章 面向 FPGA 的基于贝叶斯优化的自动化近似乘法器生成器	75
4.1 研究背景与现状	75
4.2 研究动机	77
4.3 研究内容与创新点	78
4.4 研究方法	78
4.4.1 半加器阵列	78
4.4.2 4 种精确半加器的简化方法	79
4.4.3 贝叶斯优化	80
4.4.4 误差分析	80
4.4.5 价格计算	80
4.4.6 优化流程	81
4.5 实验结果	82
4.6 本章小节	84
第 5 章 基于乘法器库的近似逻辑综合	85
5.1 基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架	85
5.1.1 研究背景	85
5.1.2 国内外研究现状	92

5.1.3 研究动机	98
5.1.4 研究内容与创新点	98
5.1.5 研究方法	99
5.1.6 实验结果	100
5.2 基于近似乘法器库面向 DNN 加速器的近似逻辑综合	102
5.2.1 研究背景	102
5.2.2 研究内容	102
5.3 本章小节	103
第 6 章 总结与展望	105
6.1 总结	105
6.2 展望	106
附录 A 不同电路在不同优化方法下的面积和延迟数据	107
参考文献	111
攻读学位期间研究成果	131

插图目录

1-1 近 50 年处理器发展趋势图	3
1-2 Transformer 类模型训练所需的运算量	5
2-1 无符号乘法器中部分积生成及累加结果示意图 (6×5)	13
2-2 补码乘法器的符号位扩展示例：左：操作数符号位扩展，右：部分积符号位扩展。	14
2-3 5×5 补码乘法器的部分积阵列示意图，红色部分积的权重为负	15
2-4 基于 Baugh-Wooley 算法设计的 5×5 补码乘法器部分积阵列示意图	17
2-5 16×16 无符号乘法的基 4 布斯算法部分积符号位扩展优化方法	19
2-6 16×16 补码乘法的基 4 布斯算法部分积符号位扩展优化方法， E 表示布斯码值符号 S 和被乘数符号的同或	20
2-7 4×4 无符号数乘法部分积及对应的阵列累加电路示意图	21
2-8 4 操作数 8 比特位宽，最后通过超前进位加法器相加的 CSA 结构图	21
2-9 利用华莱士树加法器对 8×8 无符号数乘法部分积进行累加的示意图	23
2-10 利用达达树加法器对 8×8 无符号数乘法部分积进行累加的示意图	24
2-11 一个 4 比特 CSKA 的结构示意图，FA 代表全加器	26
2-12 一个固定大小分块 CSKA 的例子：通过级联 4 个 4 比特 CSKA 实现 16 比特加法	27
2-13 一个 4 比特 CSEA 的结构图，FA 代表全加器	27
2-14 一个 16 比特线性进位选择加法器示意图	28
2-15 16 比特平方根进位选择加法器示意图	28
2-16 合并两个相邻或部分重叠的加法块 B' 、 B'' 的进位信号	30
2-17 标准 CLA 和标准 RCA 的 PPC 树状图	31
2-18 IEEE 754 单精度浮点数标准	34
3-1 近似全加器单元和近似乘法器结构示意图	38
3-2 利用 3 输入或门对部分积阵列进行压缩	39
3-3 DRUM 的原理的一个示例：通过一个小位宽乘法器对操作数的部分比特执行乘法	39

3-4 论文所提出的近似浮点数乘法器架构图和不同划分等级下的误差分布	42
3-5 一个具有 5 比特输入、2 比特输出的组合逻辑门级网表及其对应的 CGP 表示	42
3-6 实现近似电路综合的两种常见方式	43
3-7 采用 8 比特位宽量化的 LeNet 网络在 MNIST 数据集上训练后 FC1 层的输入和权重的数据分布直方图	45
3-8 近似乘法器 $f^{(1)}$ 和 $f^{(2)}$ 的误差分布图, 这里的误差是指误差距离 ED 的平方	45
3-9 基于 LeNet 和 MNIST 得到的 Evoapprox8b 中全部 500 个乘法器在 $P = 0$ 和 $P = 1$ 的情况下的精度散点图	46
3-10 Evoapprox8b 中全部 500 个乘法器不对称程度统计直方图	47
3-11 一个利用 AND、OR、XOR、shift 操作对 4×4 无符号乘法器部分积进行压缩的例子	48
3-12 一个利用 AND、OR、XOR、shift 操作对 4×4 改进的 Baugh-Wooley 乘法器的部分积进行压缩的例子	49
3-13 2×2 无符号乘法器部分积阵列及 $S = 0$ 时的分簇情况	51
3-14 均匀分布下 8 比特无符号乘法器在不同 l 和 λ 下得到的不同解对应的压缩项的总数 T 、乘法器的功耗延迟面积积 PDA 以及平均绝对误差 MAE	53
3-15 LeNet 在评估工具中的 DAG 表示	55
3-16 采用伪量化方法的基于噪声训练的 DNN 计算流图	55
3-17 不同噪声幅值训练并近似后的 AlexNet 神经网络在 CIFAR-10 推理数据集上的精度	56
3-18 不同 l 和 λ 取值下生成的近似乘法器的 MAE 和 PDA 散点图比较	58
3-19 生成的近似乘法器与国际前沿工作进行 MAE 和 PDA 比较	58
3-20 基于 8 比特位宽量化的 LeNet 网络在 MNIST 推理数据集上的输入和权重数据直方图	59
3-21 不同乘法器在 LeNet 和 MNIST 上的精度和 2GHz 时钟频率下的 PDA 散点图	60
3-22 基于 LeNet 和 MNIST 的不同乘法器的功耗、延迟、面积、PDA 和 APDA, 以 DesignW 为标准进行归一化	61
3-23 不同 DNN 加速器在多个时钟频率约束下基于不同乘法器得到的功耗和 PDA 指标	62

3-24 基于 8 比特位宽量化的 AlexNet 网络在 CIFAR-10 推理数据集上的输入和权重数据直方图	63
3-25 不同乘法器在 AlexNet 和 CIFAR-10 上的精度和 2GHz 时钟频率下的 PDA 散点图	64
3-26 基于 AlexNet 和 CIFAR-10 的乘法器的功耗、延迟、面积、PDA 和 APDA，以 DesignW 为标准进行归一化	65
3-27 基于 8 比特位宽量化的 VGG16 网络在 CIFAR-10 推理数据集上的输入和权重数据直方图	66
3-28 不同乘法器在 AlexNet 和 CIFAR-10 上的精度和 2GHz 时钟频率下的 PDA 散点图	67
3-29 基于 VGG16 和 CIFAR-10 的不同乘法器的功耗、延迟、面积、PDA 和 APDA，以 DesignW 为标准进行归一化	67
3-30 一个自适应 FIR 滤波器的结构图	68
3-31 滤波器在精确乘法下权重的数据分布	69
3-32 不同乘法器的 PDA 和 PSNR 对比散点图	70
3-33 不同乘法器在 200MHz 时钟频率约束下的的功耗、延迟、面积、PDA 和 LPDA，以 DesignW 为标准进行归一化	71
3-34 基于平均值为 0、标准差为 2^{30} 的正态分布随机生成的 2^{20} 对大于 0 的 32 比特输入数据直方图	71
3-35 生成的近似乘法器与 EvoLite 中的 32 位无符号乘法器 (Evo32) 在 1.5GHz 时钟频率的约束下进行 MAE 和 PDA 比较	72
 4-1 一个典型的拥有 6 个输入的 LUT 结构图	76
4-2 一种 2 比特位宽的 FPGA 进位链结构示意图	76
4-3 Evo8 中的所有乘法器在 ASIC 和 FPGA 下的 PDA 提升	77
4-4 4×4 无符号乘法器的部分积阵列，共 16 个比特	79
4-5 利用搜索到的半加器阵列对部分积进行压缩后的结果	79
4-6 整体流程	81
4-7 不同乘法器的 PDA 和 MM' 对比图	82
 5-1 传统逻辑综合流程图	86
5-2 函数 $x_2(x_1 + x_3)$ 的两种不同的 AIG 实现	86
5-3 函数 $\overline{x_1 + x_2} \oplus x_3 \cdot \overline{x_4}$ 的 AIG 和 XAIG，圆圈代表 AND 节点，六边形代表 XOR 节点，虚线边代表取反操作	88
5-4 函数 $\langle x_1, x_2, (x_3 \oplus x_4) \rangle$ 分别在不同 DAG 中的表示，从左到右依次为：AIG、MIG、XMG，虚线代表取反操作	88

5-5 节点 z 的一个锥 {z,x,a,d,c,b,e} 和两个割 cut1 与 cut2	89
5-6 不同节点的最大无扇出锥	90
5-7 超图划分问题的定义	90
5-8 基于强化学习的 DRiLLS 序列优化方法架构图	92
5-9 不同序列探索方法在迭代 200 次后的 LUT 映射结果	94
5-10 基于 AIG 和 GCN 的序列质量预测器	95
5-11 Bulls-Eye 整体框架图	95
5-12 LSOOracle 流程图	96
5-13 将卡诺图转变为 KMImage 的示例	97
5-14 本文提出的逻辑优化框架流程图	99
5-15 XWYF 乘法器的评估结果以及基于 XWYF 实现的卷积加速器的 评估结果	102

表格目录

2-1 基 4 布斯编码表	18
3-1 采用不同近似乘法器近似后的 LeNet 网络在 MNIST 数据集的精度	60
3-2 采用不同近似乘法器近似后的 AlexNet 网络在 CIFAR-10 数据集的精度	63
3-3 采用不同近似乘法器近似后的 VGG16 网络在 CIFAR-10 数据集的精度	66
4-1 根据 MM' 对乘法器进行分组后每组的最佳 PDA 值比较	83
5-1 DRiLLS 中不同效果的优化序列对应的奖励情况	93
5-2 实验结果	93
5-3 基于不同方法得到的面积、延迟和 ADP 的平均百分比提升	101

摘要

随着人工智能的不断发展，计算需求急剧增加，需要海量的算力进行支撑，带来大量的能源消耗。同时，在可穿戴设备、便携设备和数据中心等场景，集成电路面临的功耗问题同样严峻，人们需要寻找新的方法降低系统功耗，提高芯片能效。近似计算是一种新兴的计算范式，允许系统在可接受的误差范围内返回结果，与容错应用结合，在满足精度需求的前提下能够提高计算效率，降低芯片能耗。因此，在数字信号处理、机器学习等场景中，近似计算得到了工业界和学术界的广泛关注。

近似电路设计是近似计算的一个分支，是指通过对电路中的精确算术单元引入近似，达到降低硬件开销的目的。针对近似乘法器，本文进行了多方面的研究，包括：

(1) 提出并开源了一个考虑数据分布和输入极性的面向 ASIC 的高质量自动化近似乘法器生成方法，该方法在对部分积进行累加求和之前，引入与、或、异或和移位操作对部分积进行压缩，降低部分积阵列的规模，减轻后续的累加压力。基于改进的 Baugh-Wooley 算法，方法经过扩展后实现了对补码有符号乘法器的支持。为了能够自动化求解，将寻找较优压缩操作的问题建模成数学问题，利用混合整数遗传算法进行搜索。对方法进行了大量的实验，结果表明，均匀分布下生成的 8 比特无符号乘法器大幅优于国际前沿工作，针对采用 8 比特无符号数量化的 LeNet、AlexNet 和 VGG16 生成的乘法器在精度损失不超过 0.01% 的情况下实现了 26.4%-47.6% 的性能收益，面向自适应滤波器生成的 16 位补码有符号乘法器在 PSNR 损失可以忽略的情况下实现了 27.1% 的提升，基于 32 比特半正态分布的实验结果表明提出的方法对大位宽乘法器同样有效。

(2) 面向 FPGA，提出并开源了一个基于贝叶斯优化的自动化近似乘法器生成方法，该方法假设乘法器的部分积在生成后、累加前存在一次由半加器阵列进行的压缩操作，利用贝叶斯优化基于提出的 4 种半加器简化方法对半加器阵列进行优化，保留后续累加过程中部分积的粗粒度加法，使其能够被 EDA 工具轻易地识别并映射到 FPGA 的快速进位链。与国际前沿工作中的 1167 个近似乘法器相比，生成的乘法器能够形成 Pareto 前沿，性能平均提高了 28.70%-38.47%。

(3) 提出并开源了一个基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架，该框架首先利用 Yosys 对电路进行读入和解析，接着将电路中的组合

逻辑提取出来，利用“自然划分”和 MFFC 超图划分将提取的组合逻辑分割成多个子电路，对所有的子电路利用提出的强化学习序列优化方法并行地进行探索，最后由商业综合工具评估结果。基于超过 150 个电路的实验结果表明，提出的方法与 ABC resyn2 相比，面积延迟积平均提高了 5.17%。将框架与近似乘法器库结合，对基于不同近似乘法器实现的 DNN 硬件加速器进行了探究，结果显示近似乘法器的单独硬件成本提升与对应加速器的硬件成本提升存在一定偏差，但处于帕累托前沿的乘法器对应的加速器的硬件开销仍处于帕累托前沿，在实际使用中可对库中的帕累托前沿乘法器进行探索以确定最佳硬件实现。

关键词：近似计算；近似乘法器；逻辑综合

中图分类号：TN4

Abstract

With the continuous development of artificial intelligence, there has been a sharp increase in computational demands, requiring massive computing power and resulting in significant energy consumption. At the same time, power consumption has become a pressing issue for integrated circuits in scenarios such as wearable devices, portable devices, and data centers. People are seeking new methods to reduce system power consumption and improve chip efficiency. Approximate computing is an emerging computing paradigm that allows systems to return results within an acceptable range of error. When combined with fault-tolerant applications, it can improve computational efficiency and reduce chip power consumption while meeting accuracy requirements. As a result, approximate computing has gained widespread attention in the industry and academia, especially in fields like digital signal processing and machine learning.

Approximate circuit design is a branch of approximate computing that aims to reduce hardware costs by introducing approximation into precise arithmetic units. This thesis focuses on the approximate multiplier design and includes the following aspects:

(1) This thesis proposes an open-source high-quality automated approximation multiplier generation method for ASICs that considers data distribution and input polarity. The method uses AND, OR, XOR, and shift operations to compress partial products before accumulation, reducing the size of the partial product array and alleviating the accumulation pressure. Based on the improved Baugh-Wooley algorithm, the method can generate two's complement signed multipliers. To automatically search the optimal compression operations, the problem of finding the best compression operations is defined as a mathematical problem, then a mixed integer genetic algorithm is used to solve the problem. Extensive experiments show that the generated 8-bit unsigned multipliers under uniform distribution outperform the state-of-the-art works. For the multipliers generated for LeNet, AlexNet, and VGG16 using 8-bit unsigned quantization, the performance gains range from 26.4% to 47.6% while maintaining accuracy loss below 0.01%. The 16-bit two's complement signed multiplier generated for adaptive filters achieves a 27.1% hardware improvement with negligible PSNR loss. The experimental results based on a 32-bit half-normal distribution demonstrate the effectiveness of the method for large-bit-width multipliers.

(2) This thesis proposes an open-source automated approximate multiplier generation method based on Bayesian optimization for FPGA. The method assumes that there is a compression process performed by a half-adder array on the partial products after generation and before accumulation. The half-adder array is optimized using the Bayesian optimization algorithm based on four proposed half-adder simplification methods. Then the method preserves the coarse-grained additions in the subsequent accumulation, which can be easily mapped to the FPGA's fast carry chains by EDA tools. Compared to 1167 state-of-the-art approximate multipliers, the generated multipliers form a Pareto front with an average improvement of 28.70% to 38.47%.

(3) This thesis proposes an open-source end-to-end reinforcement learning logic optimization framework based on adaptive MFFC hypergraph partitioning. The framework uses Yosys to parse verilog and extract the combinational logic from the circuit, which then be divided into multiple sub-circuits using "natural partitioning" and MFFC hypergraph partitioning. It explores all sub-circuits in parallel using the proposed reinforcement learning sequence optimization method and evaluates the results using the commercial synthesis tool. Based on more than 150 benchmarks, experimental results show that the proposed method achieves an average improvement of 5.17% in area-delay product compared to ABC resyn2. Combining the framework with approximate multiplier libraries, exploration was conducted on DNN hardware accelerators based on different approximate multipliers. The results indicate that the increase in hardware cost for individual approximate multipliers deviates from the corresponding accelerator's hardware cost increase. However, the hardware overhead of accelerators corresponding to multipliers at the Pareto front still remains at the Pareto front. Thus exploring Pareto-front multipliers in the library can achieve the optimal hardware implementation.

Keywords: Approximate Computing; Approximate Multiplier; Logic Synthesis

CLC code: TN4

第 1 章 绪论

1.1 研究背景与意义

1.1.1 半导体工艺的发展

1947 年，贝尔实验室发明了世界上第一个晶体管，时任全球联通公司中心实验室职员的杰克·基尔比（Jack Kilby）对此产生了浓厚兴趣，并于 1958 年在德州仪器创造了世界上第一个采用飞线连接的锗基底扩散工艺集成电路。紧接着，仙童半导体的罗伯特·诺伊斯（Robert Norton Noyce）在 1959 年发明了更具有实用价值的、能够进行大规模量产的、基于导线结构的硅基底平面工艺集成电路。此后，在短短的半个世纪内，集成电路无处不在。

作为第三次工业革命的起源，集成电路的发明和应用使人类社会从工业时代迈入了信息时代，极大地解放了生产力，推动了人类社会的发展。早在集成电路发明早期，英特尔（Intel）的创始人之一戈登·摩尔（Gordon Earle Moore）就预言半导体行业将会迅猛发展，于 1965 年提出了著名的摩尔定律（Moore's law）^[1]：集成电路上可容纳的晶体管数目，约每隔一年翻一番（1975 年修正为两年^[2]）。不久后，罗伯特·登纳德（Robert Heath Dennard）发现，晶体管所消耗的电压和电流，会随着晶体管的尺寸做相同比例的减少，功率密度保持不变，这便是著名的登纳德缩放定律（Dennard scaling）^[3]。登纳德定律表明，由于单位面积的晶体管的功耗维持稳定，而计算能力会随着每一代工艺节点而提升，芯片将会越来越节能。

近现代的数十年间，半导体制造商一直遵循着摩尔定律，不断缩小晶体管的尺寸，改进晶体管的制造方法。在传统的硅平面工艺被发明 40 年后，晶体管的栅极长度已经缩小到了 100 纳米（Nanometer, nm）以下^[4]，由于短沟道效应（Short-channel effects）的影响^[5]和工作电压的下降，载流子的界面散射加剧，迁移率降低，器件的驱动电流减小，响应速度变差。为了改善晶体管的开关特性，工业届各家厂商于 2003 年-2005 年在 90nm-65nm 节点陆续引入了锗应变调控方法^[6-7]，实现了迁移率的大幅提升。之后，晶体管中电子的量子隧穿效应引起的漏电流问题取代了性能问题，成为了首要考虑事项，为了降低发热，高介电常数金属栅极技术（High-k Metal Gate, HKMG）被 Intel 于 2007 年在 45nm 工艺节点采用^[8]，改进后被再次应用在 32nm 节点^[9]。后来，为了进一步降低功耗，半导体

制造厂商于 2012 年左右陆续在 22nm 及以下制程全面转向由加州大学伯克利分校胡正明教授研发的鳍式场效应晶体管 (Fin Field-Effect Transistor, FinFET)^[10]，延续了摩尔定律。然而，随着新工艺节点的不断推出，最新的量产工艺已经由台积电 (Taiwan Semiconductor Manufacturing Company, TSMC) 推进到了 3nm，FinFET 晶体管几乎达到了物理极限，摩尔定律陷入停滞。

1.1.2 计算机体系结构的发展

一方面，半导体制造厂商在不断地更新工艺制程以提高晶体管的密度；另一方面，芯片设计厂商也持续地从体系结构层面进行优化，以更好地利用额外增多的晶体管，改善芯片的性能、功耗和面积 (Performance-Power-Area, PPA)。在登纳德定律的指导下，早期的集成电路设计厂商孜孜不倦地提高芯片的时钟频率，实现性能更高的单核处理器。英特尔甚至在 2000 年就豪言要在 2011 年将其生产的中央处理器 (Central Processing Unit, CPU) 推进至 10GHz (Gigahertz)。然而，随着晶体管尺寸的持续缩小，电子的量子隧穿效应 (Quantum tunneling effect) 开始显露，晶体管的漏电流不断增加，功耗不减反增，登纳德定律开始失效，人们无法再简单地通过增加芯片的时钟频率来提高单核芯片的性能。同时，人们意识到，更高的时钟频率并不一定会带来芯片性能的增强。在 1986 年-2002 年左右，伴随着主频提升的指令级并行 (Instruction Level Parallelism, ILP) 技术是提高处理器性能的主要方法。然而，过深的流水线会导致分支预测 (Branch prediction) 出错时需要花费巨大的代价来恢复状态，平白浪费许多能量，带来不可忽视的性能损失^[11]。同时，虽然单核处理器性能每年以 50% 的速度进行提升，但内存性能的提升每年仅约 7%，这导致了冯·诺依曼结构 (Von Neumann architecture)^[12] 下严重的内存墙问题 (The memory wall problem)^[13]，考虑到内存访问所需的时间远大于 CPU 的计算时间，盲目提升处理器主频的作用非常有限。另外，随着互联网的快速发展，应用的类型从传统的计算密集型向数据密集型转变，这一方面使控制流变得不规则，导致难以有效利用 ILP 技术提升性能，另一方面导致了大量的数据搬移，加剧了内存墙问题带来的负面影响。最后，芯片互连线延迟所占比例的持续上升和设计复杂度的不断增加也迫使研究人员停止开发更高速的单核处理器，转而将目光朝向多核架构的研究^[14]。如图1-1所示^[15]，以 2005 年英特尔放弃研发 4GHz 的奔腾四 (Pentium IV) 处理器为标志，多核设计开始兴起^[16]。

多核架构的处理器拥有多个核心，能够同时运行多个任务，或者并行处理一个任务，大大缩短软件的运行时间。表面上看，不断增加芯片的核心数便能不断提高其处理能力，然而，多核处理器的运算能力并不能随着核数无限提升，原因如下：(1) 由于功耗墙 (The power wall) 的存在，即使制造出一个拥有许多个

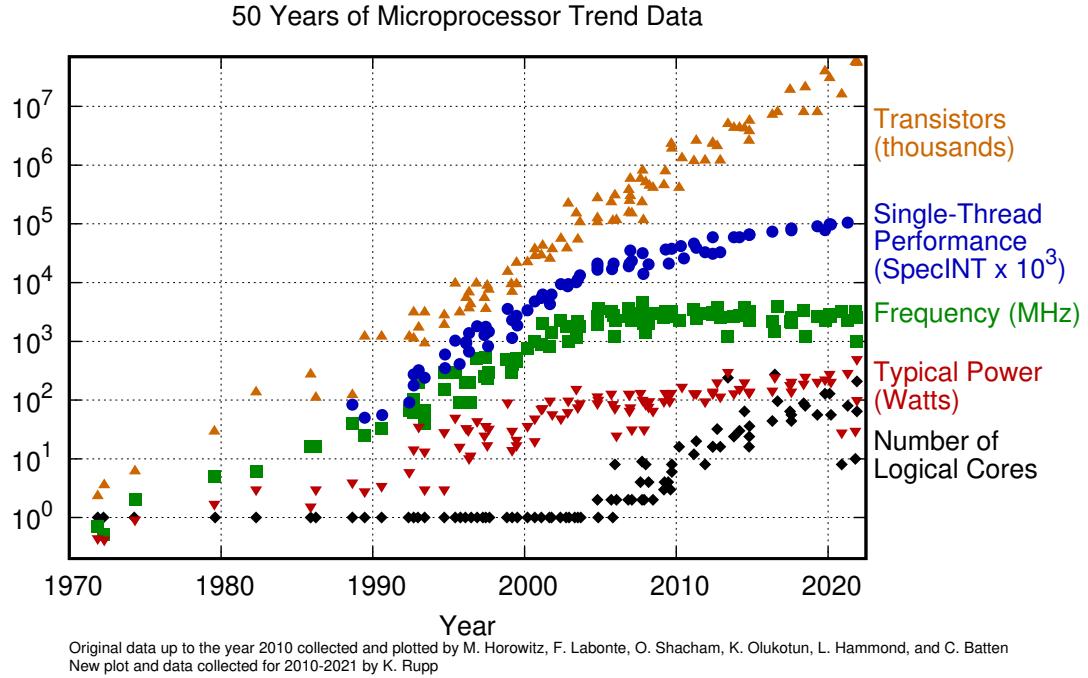


图 1-1 近 50 年处理器发展趋势图

核心的芯片，也无法允许所有核心同时运行^[17]，这部分不工作的晶体管被称为“暗硅（Dark silicon）”；（2）在阿姆达尔定律中^[18]，任务在多核处理器下的理论加速比为：

$$\text{加速比} = \frac{W_s + W_p}{W_s + \frac{W_p}{p}} \quad (1.1)$$

式中 W_s 和 W_p 为任务规模中的串行分量（不能被并行的部分）和并行分量（可以被并行的部分），可见加速比上限由任务中不能被并行处理的部分决定，若程序没有并行分量，那么不论使用多少核的处理器，任务都无法被加速；（3）由于内存提升的速度远小于处理器提升的速度，导致内存墙问题越来越严重，对某些应用来讲，盲目堆砌多核，不但不能加速任务的处理，反而导致了性能的下降^[19]。

为了解决多核架构遇到的问题，软件和硬件人员分别从两方面入手进行优化。一方面，与多核处理器配套的软件如操作系统和编译器等工具开始充分发展，尽可能利用多核架构的优点，提高任务的运行速度；另一方面，计算机体系结构人员开始采用不同的方法来解决“暗硅”问题和内存墙问题。对于内存墙问题，研究人员使用多级缓存（Multi-level caches）结构和更先进的分支预测方法来增加 cache 的命中率；对于“暗硅”问题，学术界和工业界提出了 3 个方法来充分利用未工作的晶体管^[20]：（1）低速多核。把处理器中每个核的运行频率

限制在一个较低的水平，充分利用处理器的并行性，提高运算能力，例如英特尔设计的采用太阳能供电的 x86 多核处理器^[21]；(2) 自动超频。允许多个核心在短时间内达到很高的频率用来处理高计算量的任务，之后迅速将每个核心的频率降低来减缓发热，常用于由电池供电的对功耗有严格要求的芯片；(3) 专用集成电路（Application-Specific Integrated Circuit, ASIC）。利用 ASIC 性能高、发热低的优点，把“暗硅”部分设计成专用加速器，将 CPU 和 ASIC 结合，提高整体的处理能力。为了得到能效更高的处理器，设计人员往往结合多种方法进行优化，比如英特尔的睿频技术（Turbo Boost Technology）^[22]，ARM 的大小核架构（big.LITTLE）^[23]等。发展到现代，广义的处理器已经不仅仅是一个只有传统运算核心的芯片，而是成为了一个包含许多专用处理单元如 GPU（Graphics Processing Unit）、NPU（Neural Processing Units）、ISP（Image Signal Processor）、嵌入式 FPGA（embedded Field Programmable Gate Array, eFPGA）、DSP（Digital Signal Processing）等模块的复杂异构片上系统（System on Chip, SoC）。

随着机器学习（Machine Learning, ML）的飞速发展，人工智能（Artificial Intelligence, AI）模型对算力的需求激增。在 2012 年之前，训练一个 ML 系统所需的算力大约每 17 到 29 个月翻一番^[24]，增长率和摩尔定律保持一致。然而，2012 年杰弗里·辛顿（Geoffrey Everest Hinton）的学生亚历克斯·克里泽夫斯基（Alex Krizhevsky）设计了 AlexNet^[25]这一 8 层卷积神经网络（Convolutional Neural Network, CNN），利用 GPU 夺得了 ImageNet LSVRC（Large Scale Visual Recognition Challenge）竞赛的冠军，并大幅领先第二名 10.8 个百分点，掀起了 CNN 研究的热潮，深度学习（Deep Learning）迎来了大爆发。伴随着互联网的快速发展和全球社会数字化转型带来的海量数据，深度学习常规模型（Regular-scale Model）训练一次所需的算力大约每 4-9 个月翻一番^[24]。同时，随着 2016 年以谷歌（Google）的 AlphaGo^[26]战胜韩国棋手李世石为代表，大规模模型（Large-Scale Model）开始引领人工智能的潮流，计算量大约每 9 到 10 个月翻一番^[24]。如此巨大的算力需求和不断改变的算法模式是传统的运算芯片所无能为力的，于是各种领域专用架构（Domain-Specific Architecture, DSA）竞相涌现，牺牲了部分的通用性，实现了高能效计算，如寒武纪的 DianNao^[27]、Google 的 TPU（Tensor Processing Unit）^[28]、GraphCore 的 IPU（Intelligence Processing Unit）^[29]、华为的 DaVinci 架构^[30]、百度的昆仑芯片^[31]等，计算机体系结构迎来了新黄金时代^[11]。与 ASIC 相比，DSA 的通用性更强，能够适应日新月异的 AI 算法。同时，CPU、GPU 和 FPGA 架构也不断革新，引入了各种针对 ML 应用的专用处理单元，如 Intel 至强（Xeon）处理器中的 AI 加速单元^[32]、英伟达（Nvidia）GPU 中的张量计算核心（Tensor Core）^[33]、赛灵思（Xilinx）FPGA 中的 AI 引擎（AI Engine, AIE）^[34]。

与计算机视觉 (Computer Vision, CV) 领域不同，在自然语言处理 (Natural Language Processing, NLP) 方面，纯粹由注意力机制构成的 Transformer 模型^[35]取代了 CNN，占据了主导地位。在 Google 提出 Transformer 模型一年后，大规模语言模型 (Large Language Model, LLM) 开始出现，以 OpenAI 旗下的 GPT (Generative Pre-trained Transformer) 为代表，LLM 的大小一直在飞速增长：2018 年发布的 GPT-1 参数量为 1.17 亿，数据量为 5GB (Gigabyte)；2019 年发布的 GPT-2 参数量为 15 亿，数据量为 40GB；2020 年发布的 GPT-3 参数量为 1750 亿，数据量增加到 45TB (Terabyte)。研究表明，Transformer 类模型训练所需的运算量每 2 年增加 750 倍，远超摩尔定律的演进速率，如图1-2所示^[36]。

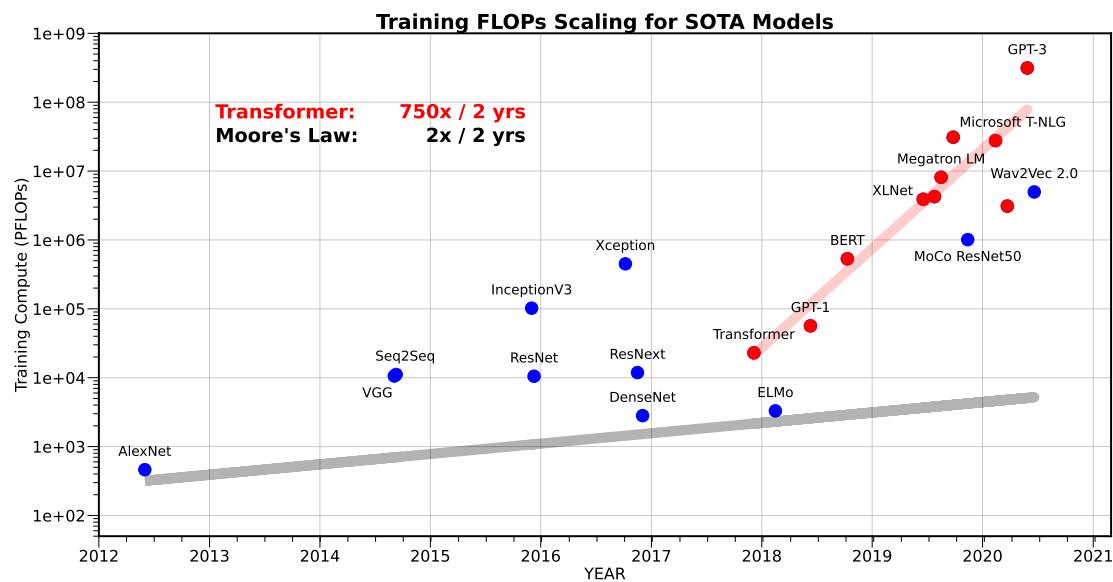


图 1-2 Transformer 类模型训练所需的运算量

如此巨大的计算需求，需要海量的算力进行支撑。目前业界通用的办法是利用并行化技术在多个 GPU 或 DSA 上进行训练，以求在合理的时间内获得想要的结果。然而，这会伴随着资源的大量消耗。例如，训练一个 GPT-3 模型需要 355 个 GPU 年（一块 GPU 运行 355 年的运算量），花费 460 万美元，耗电 1287 兆瓦时 (Megawatt Hour, MWh)，大约相当于 120 个家庭 1 年的用电量^[37-38]。并且，训练阶段的能耗通常只占模型整个生命周期的 40%^[39]，ChatGPT 的火热导致这一占比对 GPT 类应用更低，这产生了大量碳排放，对环境造成了很大的负担。为了降低能耗，研究人员从算法和硬件两方面来对 AI 应用进行优化。一方面，高效的 ML 模型架构可以在更少计算量的情况下实现更高的精度，减少资源的消耗；另一方面，采用专门用于 AI 训练和推断的芯片能够提高系统的能效，实现绿色计算 (Green Computing)^[40]。但是，目前的研究表明，LLM 模型的规模越大，往往 NLP 任务的效果越好，^[41]，这意味着模型的精简程度十分有限。同

时，芯片的能效提升远远跟不上模型的规模增长，人们迫切需要新的方法在提升算力的同时降低资源消耗。

1.1.3 后摩尔时代的技术路线

以生成式 AI (Generative AI) 为代表的人工智能等应用的发展，对半导体材料和器件提出了更高的要求。当前，随着硅晶体管的演进接近物理极限，不仅特征尺寸的缩小越来越困难，迭代产生的工艺红利也消失殆尽，硅基电子技术临近生命周期极限。为了探索集成电路领域新的发展规律，持续提高芯片能效，学术界和工业界提出了多个发展方向，这里列举几个典型的技术路线：

(1) More Moore

“More Moore”即“深度摩尔”，其基本思路是延续摩尔定律的发展，在兼顾性能和功耗的同时，继续缩小晶体管的尺寸^[42]。随着 FinFET 的漏电越发严重，对沟道拥有更强控制能力的全环栅晶体管（Gate All Around FET, GAAFET）将成为未来的主流^[43]。

(2) More than Moore

“More than Moore”即“超越摩尔”，侧重于功能的多样化，由应用需求驱动，通过先进封装技术实现异质集成系统^[44]。与不断优化晶体管的“More Moore”路线不同，“More than Moore”从需求端出发，以系统应用为起点，尝试在提高芯片集成度和能效的同时降低芯片制造的成本。在 SoC 中，除了逻辑（Logic）和存储（Dynamic Random Access Memory, DRAM）部分以外，模拟（Analog）、射频（Radio Frequency, RF）等模块往往并不能随着工艺的迭代获得显著地性能改善，甚至可能会变差。因此，数字（Digital）部分可由先进工艺实现，而其余部分可选择更合适的工艺进行流片，最后不同模块通过先进封装技术组合在一起，模块间通过高速接口进行通讯，实现整体的能效提升。

(3) Beyond Complementary Metal Oxide Semiconductor (CMOS)

前面两种路线仍然是基于硅基集成电路进行拓展，“Beyond CMOS”是指利用 CMOS 之外的新器件、新材料来制造晶体管，提高芯片的能效^[45]。与 CMOS 相比，这类新器件往往具有更高的密度、更强的性能、更低的功耗，但可能还无法大规模制造或制造成本不能接受。目前，该方向是学术界和工业界研究的热点之一，各种新型方案百花齐放，比如低功耗的隧穿场效应晶体管（Tunneling FET, TFET）^[46]、与 CMOS 工艺兼容的单电子晶体管（Single Electron Transistor, SET）^[47]、具有高迁移率的石墨烯晶体管（Graphene Transistor）^[48]、适合 RF 电路的碳纳米管场效应晶体管（Carbon Nanotube FET, CNFET）^[49]等。但是，这一方向的绝大多数成果还未走出实验室，仍处于初期的前瞻性研究阶段，距离商业化较远。

1.1.4 近似计算的优势

随着人工智能的不断发展，计算需求急剧增加，带来大量的能源消耗。同时，在可穿戴设备、便携设备和数据中心等场景，集成电路面临的功耗问题同样严峻，人们需要寻找新的芯片设计方法以同时满足高性能和低功耗的严苛要求。在实际生活中，许多应用具有错误容忍的特性，这类应用被称为容错应用 (Error-tolerant applications)。一个典型的例子是，当观看视频时，由于感知的限制，即使视频中某些帧出错甚至丢失了，人类很可能也察觉不到。类似地，即使搜索引擎返回的结果没那么精确，查询者也可以接受。近似计算 (Approximate computing) 是一种新型的计算范式 (Paradigm)，与精确计算 (Exact computing) 相比，它可能返回不准确的结果。与容错应用结合，近似计算可以在满足精度需求的前提下节省大量能源，达到降低功耗、提高能效的目的。因此，在数字信号处理、机器学习等场景中，近似计算得到了工业界和学术界的广泛关注^[50-51]。

目前，有关近似计算的研究主要集中在四个层面：

(1) 软件层近似 (Software-level approximation)

软件层的近似有多种实现方式，比如在循环中跳过一些迭代来更快地获得计算结果，或者根据条件语句进行判断，从而跳过某些任务的执行来减少程序运行时间。另外，许多启发式算法 (Heuristic algorithm) 如模拟退火 (Simulated annealing) 和遗传算法 (Genetic algorithm) 常常需要在一定的时间内获得次优解 (Sub-optimal solution)，也属于软件层近似的一种。

(2) 近似电路 (Approximate circuits) :

通过对加法器 (Adder)^[52]、乘法器 (Multiplier)^[53]、除法器 (Divider)^[54] 等算术运算单元 (Arithmetic units) 引入近似，获得能效的提升，被称为电路级近似。电路级近似的实现方式大体上可以分为两类：电压调节 (Voltage scaling) 和功能近似 (Functional approximation)^[55]。其中，电压调节是通过降低模块的工作电压但不降低频率来减少电路的功耗。然而，这一般会产生时序错误 (Timing error)，带来难以控制的计算误差^[56]。功能近似通常聚焦在电路结构或门级网表 (Gate-level netlist) 的简化上，与电压调节相比，功能近似的方法带来的误差易于控制，也是目前近似计算研究最为深入的方向^[57]。

(3) 近似存储和近似内存 (Approximate storage and memory) :

与存储精确数据相比，存储近似后的数据能够改善数据读取的延迟，降低数据搬移的能耗。例如，通过舍弃浮点数 (Floating-point number) 低有效位 (the Least Significant Bit, LSB)，可以减少数据存储所需要的位宽 (Bit width)，提高存储密度。在基于 Flash 的固态硬盘 (Solid State Drive, SSD) 中引入近似计算可以提高 SSD 的读取性能^[58]。对于内存或 cache 来说，降低 DRAM 的刷新率^[59] 或静态随机存储器 (Static Random-Access Memory, SRAM)^[60] 的供电电压也可以达

到节省功耗的目的。

(4) 近似系统 (Approximate system):

对不同子模块如传感器、内存、处理器、通信接口等进行协同优化的方法被称为近似系统，与单独优化各个组件相比，近似系统能够取得更好的效果^[61]。

1.2 本文主要工作及组织结构

本文的工作主要集中在近似电路中定点数 (Fixed-point number) 乘法器的设计及应用上，包含以下三个方面的研究：(1) 基于白盒优化的考虑输入分布 (Input distribution) 和极性 (Polarity) 的面向 ASIC 的自动化近似乘法器设计方法；(2) 基于黑盒优化的面向 FPGA 的自动化近似乘法器设计方法；(3) 基于生成的近似乘法器库进行近似逻辑综合 (Approximate Logic Synthesis, ALS)^[55] 的研究。具体工作如下：

(1) 面向 ASIC，提出并开源了一个白盒优化的考虑输入分布和极性的高质量自动化近似乘法器生成方法，该方法在对部分积进行累加求和之前，引入与 (AND)、或 (OR)、异或 (XOR) 和移位 (Shift) 操作，降低部分积的个数，实现能效的提升。具体来说，基于应用驱动，统计应用中乘法器的输入数据分布，并在考虑极性的情况下对部分积进行压缩，减轻后续的累加负担。为了能够自动化处理，将寻找较优压缩操作的问题建模成数学问题，并用 MATLAB 进行求解。基于改进的 Baugh-Wooley 算法^[62-64]，方法经过扩展后实现了对补码有符号乘法器的支持。基于位宽为 8×8 无符号乘法的三个不同规模的神经网络包括 LeNet、AlexNet 和 VGG16 以及位宽为 16×16 有符号定点数乘法的有限冲击响应 (Finite Impulse Response, FIR) 滤波器的实验结果表明，与国际前沿工作相比，生成的近似乘法器在几乎没有精度损失的前提下，功耗延迟面积积 (Product of Power, Delay, and Area, PDA) 提升了 26.4%-27.1%。

(2) 面向 FPGA，设计并开源了一个基于黑盒优化的近似乘法器生成器，该方法假设乘法器的部分积在生成后、累加前存在一次由半加器阵列进行的压缩操作，针对半加器，提出删减 (Eliminate)、或之和 (OR Sum)、直接进位 (Direct Cout)、精确 (Exact) 四种简化方法，利用贝叶斯优化 (Bayesian optimization) 和详细设计的能够同时考虑硬件 PPA 和软件精度的目标函数 (cost function) 对半加器的优化空间进行探索，生成高质量的近似乘法器集合。与国际前沿工作中 1167 个近似乘法器相比，生成的乘法器综合指标平均提升 28.70%-38.47%，且处于帕累托前沿 (Pareto front)。

(3) 基于前面两种自动化方法得到的 ASIC 和 FPGA 乘法器库，首先针对传统逻辑综合，提出并开源了一个基于 MFFC (Maximum Fanout-Free Cone) 自适应超图划分的强化学习 (Reinforcement learning) 逻辑优化序列探索框架。基

于超过 150 个电路的实验结果表明，面积延迟积（Area Delay Product, ADP）比 ABC^[65] resyn2 平均提高了 5.17%；然后将框架与近似乘法器库结合，对基于不同近似乘法器实现的 DNN 加速器进行了研究，结果显示近似乘法器的单独硬件成本提升与对应加速器的硬件成本提升存在一定偏差。

本文共有六个章节，各章节的组织结构安排如下：

第一章，绪论。首先介绍了自集成电路发明以来半导体工艺和计算机体系结构的发展，之后分析了近似计算结合容错应用具有的优势，引出本文的研究目的，同时简述了本文的主要工作和创新点。

第二章，乘法器概述。首先介绍了精确定点数乘法器的运算过程及不同的实现方法，以及采用 Mitchell 近似的对数乘法器^[66]；之后阐述了目前主流的衡量近似电路（主要是算术单元）误差的指标，这些指标同样适用于近似乘法器。

第三章，ASIC 近似乘法器设计。首先分析了国内外有关 ASIC 近似乘法器的实现方法，主要分为四大类：手工设计（Manual design）、数学转换近似（Mathematical transformation approximation）、自动化方法（Automated method）、近似电路综合（Approximate circuit synthesis）；接着介绍了基于白盒优化的考虑数据分布和输入极性的自动化近似乘法器设计方法，并与国际前沿工作进行对比。

第四章，FPGA 近似乘法器设计。首先介绍了学术界提出的面向 FPGA 领域的多种近似乘法器设计方法，主要是通过手工修改查找表（LookUp Table, LUT）编码的方式实现；接着提出了基于黑盒优化的自动化近似乘法器生成器，并与国际前沿工作进行误差和硬件开销比较。

第五章，近似逻辑综合。首先介绍了传统的逻辑综合，设计并实现了一个基于 MFFC（Maximum Fanout-Free Cone）自适应超图划分的强化学习逻辑优化序列探索框架，并与已有的序列优化方法进行对比；之后与近似乘法器库结合，探究不同近似乘法器对 DNN 硬件加速器带来的影响。

第六章，总结与展望。该章节总结了本文的主要研究工作和成果，分析了工作中存在的局限性，并对未来进一步的探索方向进行了展望。

第 2 章 乘法器概述

乘法 (Multiplication) 是科学计算中十分常见的一种操作，也是许多 AI 应用中最消耗资源的部分^[67]。大多数近似乘法器是基于精确乘法器优化得到的，在介绍有关近似乘法器的内容之前，对精确乘法器的实现方法做一个回顾是必要的。与浮点数相比，定点数消耗的存储资源更少，在硬件上更容易实现，运算效率也更高，本文的研究集中在定点数乘法器。

2.1 精确乘法器

在计算机发展的早期，由于芯片集成度较低，并没有专门用来直接完成乘法的硬件，而是将其拆分为逻辑与、加法和移位，利用算术逻辑单元（Arithmetic Logic Unit, ALU）来实现，这种乘法方式被称为移位加^[16]。ALU 在一个时钟周期内只能对一个部分积进行加法运算，速度较慢。随着摩尔定律的不断发展，集成电路可容纳的晶体管越来越多，现代处理器中已经有专门的硬件来完成乘法，速度大大提高。

计算机中的数值类型分为无符号数 (Unsigned number) 和有符号数 (Signed number)，不论基于哪种数值类型设计的精确乘法器，其运算过程均可以看作以下三个步骤：部分积的生成、部分积的累加、以及最终相加。对部分积的生成来说，与无符号数相比，采用补码 (Two's complement) 表示的有符号数相乘时需要对部分积进行符号位扩展 (Sign extension)，并对最后一个部分积执行减法。这一方面增加了部分积的规模，不利于后续的累加操作；另一方面需要硬件支持减法，增加了设计的复杂度。安德鲁·唐纳德·布斯 (Andrew Donald Booth) 于 1950 年提出了用于二进制补码有符号数相乘的布斯算法^[68]，该算法把符号位和数值位统一进行编码，通过移位操作跳过对乘数 (Multiplier) 中连续的 1 进行计算，提高乘法的运算速度，但在某些特殊情况下反而会增加部分积的操作次数^①。经过改进，基 4 的布斯编码 (Radix-4 Booth encoding) 在任何情况下都能将部分积的个数降低一半^[69-70]，提高了电路的性能，得到了广泛的应用。Baugh 和 Wooley 于 1973 年提出了 Baugh-Wooley 算法^[62]，将补码乘法中所有部分积的权重转换为正数，避免符号位扩展和减法操作，有利于硬件实现。与生成相比，部分积的累加方式复杂，优化空间大。因此，采用不同加法结构的累加阵列

^① <https://www.quora.com/How-does-Booths-algorithm-work/answer/Raymond-Paseman>

如华莱士树 (Wallace tree)^[71]、达达树 (Dadda tree)^[72]等被广泛研究。

2.1.1 部分积的生成

无符号乘法器

一般地，一个任意的整数位宽为 n 、小数位宽为 m 的无符号 R 进制定点数 (R 是正整数, $R \geq 2$) :

$$A = a_{n-1}a_{n-2} \cdots a_1a_0.a_{-1}a_{-2} \cdots a_{-m+1}a_{-m} \quad (2.1)$$

其十进制值为:

$$\begin{aligned} V(A) &= a_{n-1}R^{n-1} + a_{n-2}R^{n-2} + \cdots + a_1R^1 + a_0R^0 + \\ &\quad a_{-1}R^{-1} + a_{-2}R^{-2} + \cdots + a_{-m+1}R^{-m+1} + a_{-m}R^{-m} \\ &= \sum_{i=-m}^{n-1} a_i R^i \geq 0 \end{aligned} \quad (2.2)$$

其中 R 被称为基数, $R = 2$ 时便表示二进制; a_i 是自然数且 $a_i \in [0, R - 1]$; n, m 是自然数, $n + m \geq 1$, $n = 0$ 和 $m = 0$ 时分别表示无符号纯小数和无符号整数。

计算机采用二进制进行计数, 无符号二进制定点数 (0 及正数) 相乘的运算器件, 部分积是通过被乘数 (Multiplicand) 和乘数 (Multiplier) 的逻辑与 (AND) 得到的, 每个部分积的权重不同。但对电路来讲, 硬件上并不存在小数点的概念, 而是将数字视为整数直接相乘, 最后计算缩放倍数。图2-1展示了两个无符号二进制整数 (6×5) 的部分积生成及累加结果示意图。

补码有符号乘法器

无符号数不能表示负数, 为了解决二进制下这个问题, 研究人员引入了原码 (True form)、反码 (1's complement) 和补码 (2's complement) 来表示数据。与无符号数相比, 原码在最高位额外增加了一位符号位用来区分正负, 0 表示正数, 1 表示负数; 在反码中, 正数的反码就是其原码, 负数的反码是将原码中, 除符号位外, 每一位按位取反; 在补码中, 正数的补码同样是其原码, 负数的补码是其反码加一。与原码和反码相比, 补码避免了加减法不统一和存在两个零值的缺点, 简化了硬件电路设计的复杂度, 因此现代计算机底层采用补码的编码方式对数据进行存储和运算。若式(2.1)为补码, $R = 2$ 时, 其最高位权重为 -2^{n-1} , 式(2.2)变为:

$$V(A) = -a_{n-1}R^{n-1} + a_{n-2}R^{n-2} + \cdots + a_1R^1 + a_0R^0 +$$

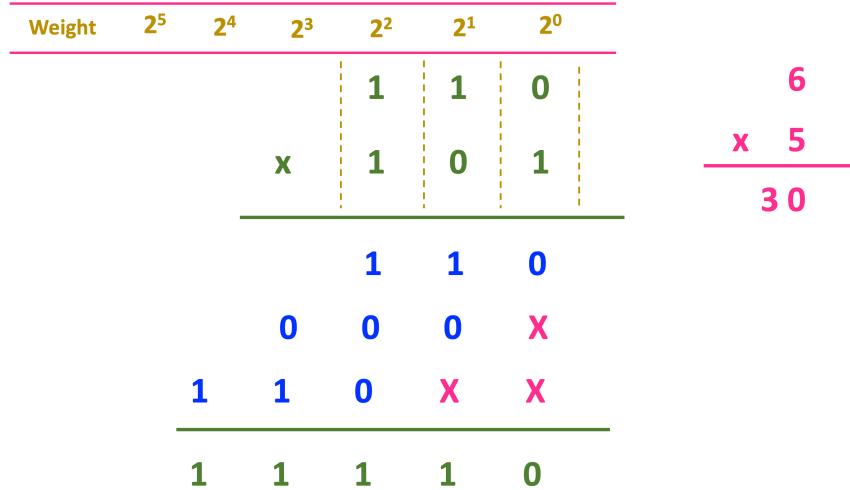


图 2-1 无符号乘法器中部分积生成及累加结果示意图 (6×5)

$$\begin{aligned}
 & a_{-1}R^{-1} + a_{-2}R^{-2} + \cdots + a_{-m+1}R^{-m+1} + a_{-m}R^{-m} \\
 &= -a_{n-1}2^{n-1} + a_{n-2}2^{n-2} + \cdots + a_12^1 + a_02^0 + \\
 & \quad a_{-1}2^{-1} + a_{-2}2^{-2} + \cdots + a_{-m+1}2^{-m+1} + a_{-m}2^{-m} \\
 &= -a_{n-1}2^{n-1} + \sum_{i=-m}^{n-2} a_i2^i
 \end{aligned} \tag{2.3}$$

这里 $n+m \geq 2$ (引入了一位符号位), $n=1$ 和 $m=0$ 时分别表示补码纯小数和补码整数。若不局限于二进制, 式(2.3)不一定成立^①, 即 R 进制下, 补码的最高位权重不一定是 $-R^{n-1}$ 。另外, 一般地, 对于一个 N 位 R 进制定点数 (N 是正整数, $N \geq 2$), 忽略小数点, 不同编码方式能够表示的数值范围为:

$$\text{无符号数: } [0, R^N - 1], \tag{2.4}$$

$$\text{原码与反码: } [-\lfloor \frac{R^N - 1}{2} \rfloor, \lceil \frac{R^N - 1}{2} \rceil], \tag{2.5}$$

$$\text{补码: } [-\lceil \frac{R^N - 1}{2} \rceil, \lfloor \frac{R^N - 1}{2} \rfloor]. \tag{2.6}$$

式中 $\lfloor \cdot \rfloor$ 和 $\lceil \cdot \rceil$ 分别表示向下取整和向上取整。例如, $R=2$, $N=8$ 时, 原码和反码的表示范围为 $[-127, 127]$, 补码的表示范围为 $[-128, 127]$ 。 $R=3$, $N=4$ 时, 原码、反码和补码的表示范围均为 $[-40, 40]$ 。

对于补码有符号二进制乘法器 (Signed binary multiplier), 目前常见的部分积生成方法有: 符号位扩展、改进的 Baugh-Wooley 算法^[62-64]、以及基 4 的布斯编码^[68-70]。下面分别进行介绍:

① <https://blog.csdn.net/mydreamongo/article/details/8863502>

(1) 符号位扩展。

按照实现细节分类，符号位扩展方法分为两种：一种是操作数（Operand）符号位扩展，优点是硬件实现不需要支持减法，缺点是部分积的规模巨大，累加电路非常复杂；另一种是部分积符号位扩展，优点是不需要修改操作数，部分积的规模适中，缺点是需要对最后一个部分积执行减法。具体细节如下：

- 操作数符号位扩展。首先根据乘数和被乘数确定乘积所需要的位宽，然后将两个操作数的位宽扩展到与乘积的位宽一致，扩展方法为高位符号位扩展，即正数进行 0 扩展，负数进行 1 扩展；之后仿照无符号数二进制乘法通过逻辑与（AND）得到部分积；最后进行累加求和，注意求和后的结果应根据乘积的正确位宽进行截断。
 - 部分积符号位扩展。同样先确定乘积所需要的位宽，但不修改操作数，通过逻辑与（AND）得到部分积；然后对部分积进行高位符号位扩展（正数 0 扩展，负数 1 扩展），将每个部分积的位宽扩展到乘积位宽；最后对部分积进行累加，但对最后一个部分积执行减法。注意对于二进制补码来说， $-[A]_{\text{补}} = [-A]_{\text{补}}$ ，即可以通过“按位取反加 1，符号位进位丢掉”求补码的相反数，将减法转换为加法。

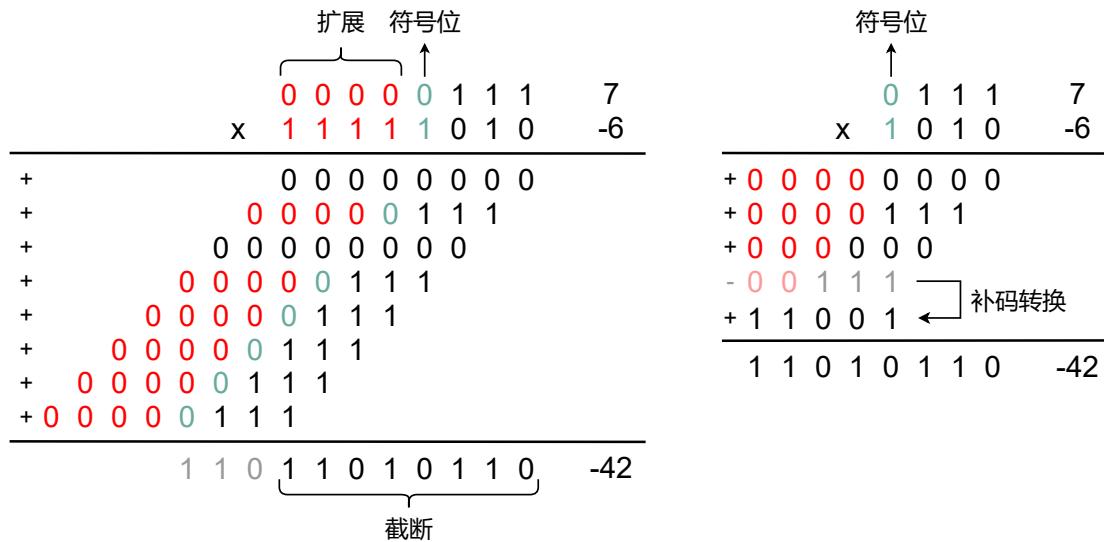


图 2-2 补码乘法器的符号位扩展示例：左：操作数符号位扩展，右：部分积符号位扩展。

图2-2举例说明了两种符号位扩展方法的不同之处。可以看到，与操作数扩展相比，部分积扩展需要的加法更少，对应的硬件实现也更具有优势。然而，与无符号乘法相比，符号位扩展总会增大部分积的规模，使电路设计更复杂。有没有

有办法能够将其降低到和无符号乘法同一个水平？改进的 Baugh-Wooley 算法是一种解决方案^[62-64]。

(2) 改进的 Baugh-Wooley 算法

Baugh-Wooley 算法是由 Baugh 和 Wooley 于 1973 年提出的用于二进制补码相乘的算法^[62]，该算法对权重为负的部分积进行修正，避免了符号位扩展，原理如下：

设 $m = 0$ ，由式(2.3)得两个 n 比特整数 X 和 Y 的十进制值：

$$V(X) = -x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i \quad V(Y) = -y_{n-1}2^{n-1} + \sum_{i=0}^{n-2} y_i 2^i \quad (2.7)$$

其乘积 P ：

$$\begin{aligned} V(P) &= V(X)V(Y) \\ &= (-x_{n-1}2^{n-1} + \sum_{i=0}^{n-2} x_i 2^i) \times (-y_{n-1}2^{n-1} + \sum_{i=0}^{n-2} y_i 2^i) \\ &= x_{n-1}y_{n-1}2^{2n-2} + \sum_{i=0}^{n-2} x_i 2^i \sum_{j=0}^{n-2} y_j 2^j - x_{n-1}2^{n-1} \sum_{i=0}^{n-2} y_i 2^i - y_{n-1}2^{n-1} \sum_{i=0}^{n-2} x_i 2^i \\ &= x_{n-1}y_{n-1}2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} x_i y_j 2^{i+j} - 2^{n-1} \sum_{i=0}^{n-2} x_{n-1} y_i 2^i - 2^{n-1} \sum_{i=0}^{n-2} y_{n-1} x_i 2^i \quad (2.8) \end{aligned}$$

$n = 5$ 时，式(2.8)对应的部分积阵列如图2-3所示，其中红色部分积的权重为负数，对应式(2.8)中的后两项：

x_4	x_3	x_2	x_1	x_0
y_4	y_3	y_2	y_1	y_0
x_4y_0	x_3y_0	x_2y_0	x_1y_0	x_0y_0
x_4y_1	x_3y_1	x_2y_1	x_1y_1	x_0y_1
x_4y_2	x_3y_2	x_2y_2	x_1y_2	x_0y_2
x_4y_3	x_3y_3	x_2y_3	x_1y_3	x_0y_3
x_4y_4	x_3y_4	x_2y_4	x_1y_4	x_0y_4
p_9	p_8	p_7	p_6	p_5
			p_4	p_3
			p_2	p_1
			p_0	

图 2-3 5×5 补码乘法器的部分积阵列示意图，红色部分积的权重为负

对于任一补码（不失一般性，假设是式(2.7)中的 X ）：

$$-V(X) = -\overline{x_{n-1}}2^{n-1} + \sum_{i=0}^{n-2} \overline{x_i} 2^i + 1 \quad (\text{按位取反加 } 1, \text{ 符号位进位丢掉}) \quad (2.9)$$

式(2.8)中的后两项变为:

$$\begin{aligned}
 & -2^{n-1}(-0 \cdot 2^n + 0 \cdot 2^{n-1} + \sum_{i=0}^{n-2} \textcolor{red}{x}_{n-1} \textcolor{blue}{y}_i 2^i) - 2^{n-1}(-0 \cdot 2^n + 0 \cdot 2^{n-1} + \sum_{i=0}^{n-2} \textcolor{blue}{y}_{n-1} \textcolor{magenta}{x}_i 2^i) \\
 & = +2^{n-1}(-1 \cdot 2^n + 1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{\textcolor{red}{x}_{n-1} \textcolor{blue}{y}_i 2^i} + 1) + 2^{n-1}(-1 \cdot 2^n + 1 \cdot 2^{n-1} + \sum_{i=0}^{n-2} \overline{\textcolor{blue}{y}_{n-1} \textcolor{magenta}{x}_i 2^i} + 1)
 \end{aligned} \tag{2.10}$$

为了避免出现与非门 (NAND)，注意到式(2.10)第一项为:

$$\begin{cases} 0, & x_{n-1} = 0, \\ +2^{n-1}(-2^n + 2^{n-1} + \sum_{i=0}^{n-2} \overline{\textcolor{blue}{y}_i 2^i} + 1), & x_{n-1} = 1. \end{cases} \tag{2.11}$$

即:

$$+2^{n-1}(-2^n + 2^{n-1} + \overline{x_{n-1}} 2^{n-1} + x_{n-1} + \sum_{i=0}^{n-2} x_{n-1} \overline{\textcolor{blue}{y}_i 2^i}) \tag{2.12}$$

同理式(2.10)第二项为:

$$\begin{cases} 0, & y_{n-1} = 0, \\ +2^{n-1}(-2^n + 2^{n-1} + \sum_{i=0}^{n-2} \overline{\textcolor{magenta}{x}_i 2^i} + 1), & y_{n-1} = 1. \end{cases} \tag{2.13}$$

即:

$$+2^{n-1}(-2^n + 2^{n-1} + \overline{y_{n-1}} 2^{n-1} + y_{n-1} + \sum_{i=0}^{n-2} y_{n-1} \overline{\textcolor{magenta}{x}_i 2^i}) \tag{2.14}$$

结合式(2.8)、式(2.10)、式(2.12)和式(2.14)， $V(P)$ 变为:

$$\begin{aligned}
 V(P) &= \textcolor{red}{x}_{n-1} \textcolor{blue}{y}_{n-1} 2^{2n-2} + \sum_{i=0}^{n-2} \sum_{j=0}^{n-2} \textcolor{red}{x}_i \textcolor{blue}{y}_j 2^{i+j} \\
 &+ 2^{n-1}(-2^n + 2^{n-1} + \overline{x_{n-1}} 2^{n-1} + x_{n-1} + \sum_{i=0}^{n-2} x_{n-1} \overline{\textcolor{blue}{y}_i 2^i}) \\
 &+ 2^{n-1}(-2^n + 2^{n-1} + \overline{y_{n-1}} 2^{n-1} + y_{n-1} + \sum_{i=0}^{n-2} y_{n-1} \overline{\textcolor{magenta}{x}_i 2^i})
 \end{aligned} \tag{2.15}$$

式(2.15)被称为 Baugh-Wooley 算法， $n = 5$ 时，该算法对应的部分积阵列如图2-4(a)所示，所有比特权重均为正值（除了最高位的 1）。与无符号乘法器相比，不论 n 多大，由式(2.15)得到的部分积阵列只会多 5 个比特。然而，原始的 Baugh-Wooley 方法会导致部分积阵列增加两层，不利于后面的累加，注意到式(2.10)可

变为：

$$+2^{n-1} \sum_{i=0}^{n-2} \overline{x_{n-1}y_i} 2^i + 2^{n-1} \sum_{i=0}^{n-2} \overline{y_{n-1}x_i} 2^i + 1 \cdot 2^n - 1 \cdot 2^{2n-1} \quad (2.16)$$

Hatamian 等人根据式(2.16)对部分积进行了重新排列^[63]，得到了2-4(b)，被称为改进的 Baugh-Wooley 算法^①。改进后的 Baugh-Wooley 方法只在部分积引入两个 1，不增加部分积阵列的层数，得到了广泛的应用^[64]。

	x_4	x_3	x_2	x_1	x_0		x_4	x_3	x_2	x_1	x_0	
	y_4	y_3	y_2	y_1	y_0		y_4	y_3	y_2	y_1	y_0	
	$\overline{x_4y_0}$	x_3y_0	x_2y_0	x_1y_0	x_0y_0		$\overline{x_4y_0}$	x_3y_0	x_2y_0	x_1y_0	x_0y_0	
	$\overline{x_4y_1}$	x_3y_1	x_2y_1	x_1y_1	x_0y_1		$\overline{x_4y_1}$	x_3y_1	x_2y_1	x_1y_1	x_0y_1	
	$\overline{x_4y_2}$	x_3y_2	x_2y_2	x_1y_2	x_0y_2		$\overline{x_4y_2}$	x_3y_2	x_2y_2	x_1y_2	x_0y_2	
	$\overline{x_4y_3}$	x_3y_3	x_2y_3	x_1y_3	x_0y_3		$\overline{x_4y_3}$	x_3y_3	x_2y_3	x_1y_3	x_0y_3	
	$\overline{x_4y_4}$	x_3y_4	$\overline{x_2y_4}$	$\overline{x_1y_4}$	$\overline{x_0y_4}$		$\overline{x_4y_4}$	x_3y_4	$\overline{x_2y_4}$	$\overline{x_1y_4}$	$\overline{x_0y_4}$	
1	$\overline{y_4}$						1					
	p_9	p_8	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0		
						y_4						

(a) 原始的 Baugh-Wooley 乘法器部分积阵列

(b) 改进的 Baugh-Wooley 乘法器部分积阵列

图 2-4 基于 Baugh-Wooley 算法设计的 5×5 补码乘法器部分积阵列示意图

(3) 基 4 的布斯编码

与 Baugh-Wooley 算法不同，布斯编码的意义在于能够减少乘法器中部分积的个数（行数），且基数越高效果越明显，比如基 4 和基 8 的布斯编码分别能够将部分积的个数降低一半和三分之二^[69-70]，大大减轻后续累加的压力。然而，高基的布斯编码电路实现复杂，目前最常用的基数是 4。原理如下：

对于补码有符号乘法，假设 n 为偶数， $y_{-1} = 0$ ，式(2.7)中的 $V(Y)$ 变为：

$$\begin{aligned} V(Y) &= -y_{n-1}2^{n-1} + y_{n-2}2^{n-2} + y_{n-3}2^{n-3} + y_{n-4}2^{n-4} + y_{n-5}2^{n-5} + \dots + \\ &\quad y_52^5 + y_42^4 + y_32^3 + y_22^2 + y_12^1 + y_02^0 + \textcolor{red}{y_{-1}2^{-1}} \\ &= (-2y_{n-1} + y_{n-2} + y_{n-3})2^{n-2} + (-2y_{n-3} + y_{n-4} + y_{n-5})2^{n-4} + \dots + \\ &\quad (-2y_5 + y_4 + y_3)2^4 + (-2y_3 + y_2 + y_1)2^2 + (-2y_1 + y_0 + \textcolor{red}{y_{-1}})2^0 \end{aligned} \quad (2.17)$$

式(2.8)变为：

$$\begin{aligned} V(P) &= V(X)V(Y) \\ &= V(X)(-2y_{n-1} + y_{n-2} + y_{n-3})2^{n-2} + \\ &\quad V(X)(-2y_{n-3} + y_{n-4} + y_{n-5})2^{n-4} + \dots + \\ &\quad V(X)(-2y_5 + y_4 + y_3)2^4 + \\ &\quad V(X)(-2y_3 + y_2 + y_1)2^2 + \end{aligned}$$

① <https://zhuanlan.zhihu.com/p/343133392>

表 2-1 基 4 布斯编码表

y_{i+1}	y_i	y_{i-1}	$-2y_{i+1} + y_i + y_{i-1}$	部分积操作
0	0	0	0	+0
0	0	1	1	$+[X]_{\text{补}}$
0	1	0	1	$+[X]_{\text{补}}$
0	1	1	2	$+2[X]_{\text{补}}$
1	0	0	-2	$-2[X]_{\text{补}}$
1	0	1	-1	$-[X]_{\text{补}}$
1	1	0	-1	$-[X]_{\text{补}}$
1	1	1	0	+0

$$V(X)(-2y_1 + y_0 + \textcolor{red}{y}_{-1})2^0 \quad (2.18)$$

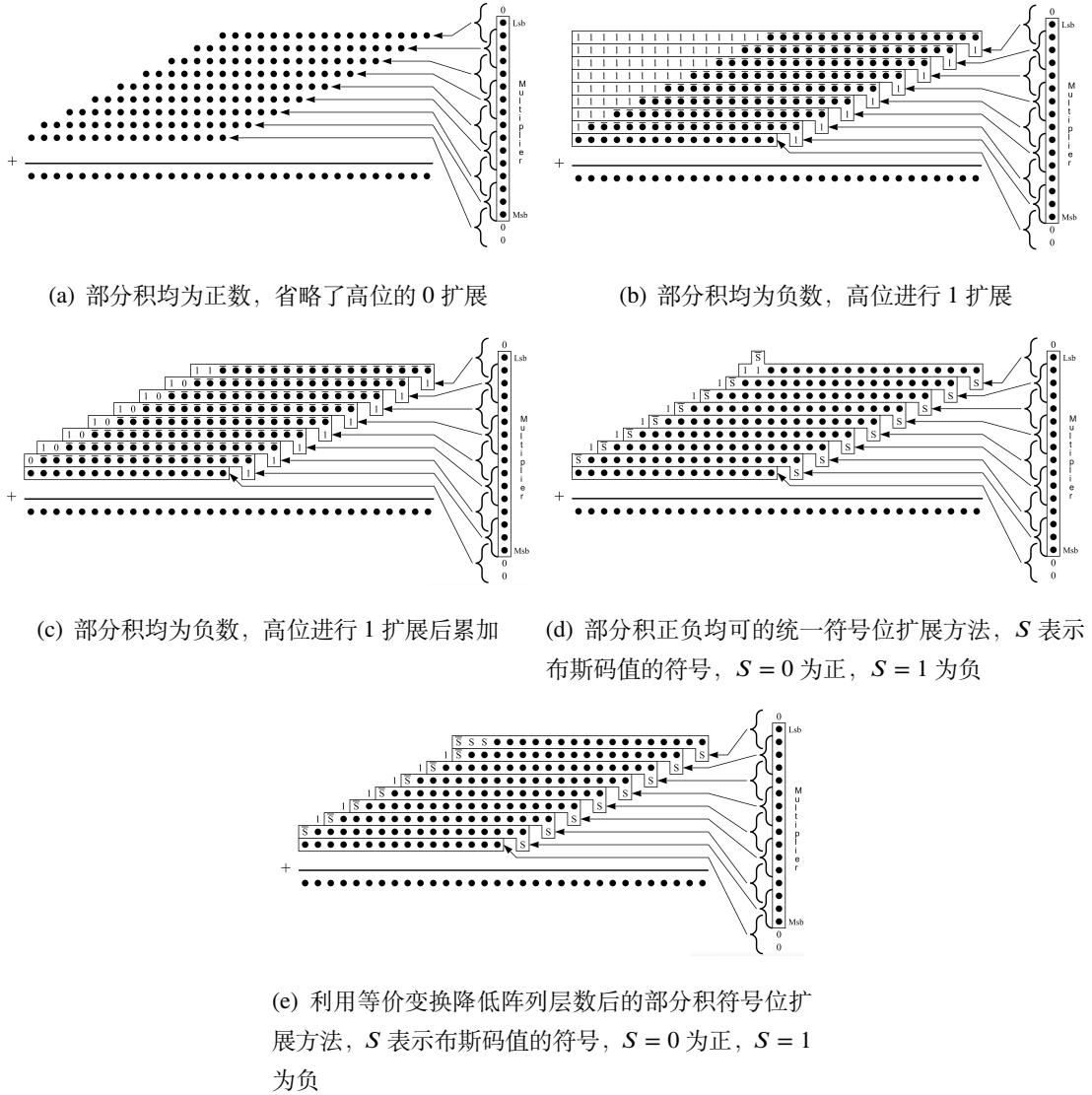
式(2.18)是基 4 的布斯编码算法公式，其中 X 是被乘数， Y 是乘数，对应的编码规则及部分积操作如表2-1所示。该算法在进行前需要在乘数的最右侧隐含地补一个 0，之后从最低有效位开始每次扫描 3 位乘数生成部分积，共 $\frac{n}{2}$ 个，然后对部分积进行符号位扩展、累加并最终相加。由表2-1可以看出，基 4 的布斯算法只涉及加法、减法和移位操作，硬件实现友好。需要注意的是，式(2.18)表示的基于补码乘法的基 4 布斯算法仅适用于 n 是偶数的情况，若 n 是奇数，先对乘数进行一位符号位扩展，将位宽变为偶数，之后再进行编码，部分积总数为 $\frac{n+1}{2}$ 个。布斯编码得到的部分积仍然需要符号位扩展，一个直接的方法是首先根据被乘数和乘数确定乘积位宽，然后直接通过高位符号位扩展将每个部分积的位宽增加到乘积位宽，但这种方法产生的部分积规模较大，累加电路复杂，可采用改进的符号位扩展方法对其进行优化。

另外，布斯算法也可以用于无符号数乘法^①，为了支持布斯编码中需要的减法操作，部分积也应采用补码格式。若基数取 4，编码形式仍然是 $-2y_{i+1} + y_i + y_{i-1}$ ，与补码乘法的基 4 布斯算法的区别在于：(a) n 是偶数时需要添加的不仅是 $y_{-1} = 0$ ，还有 $y_{n+1} = y_n = 0$ ，此时 $\{y_{n+1}, y_n, y_{n-1}\}$ 编码得到的部分积一定是 0 或正数，部分积总数为 $\frac{n}{2} + 1$ 个；(b) n 是奇数时需要添加的是 $y_n = y_{-1} = 0$ ，部分积总数为 $\frac{n+1}{2}$ 个；(c) 优化后的符号位扩展方法实现细节略微不同。

下面具体讲解基 4 布斯算法在无符号数乘法和补码有符号数乘法中部分积符号位扩展方法优化细节的不同^[73]：

对于 16×16 无符号数乘法，假设每个部分积是非负数，高位应进行 0 扩展，

^① <https://picture.iczhiku.com/resource/eetop/whKDFaTwqTZOkCxn.pdf>

图 2-5 16×16 无符号乘法的基 4 布斯算法部分积符号位扩展优化方法

0 可省略, 省略后的部分积阵列如图2-5(a)所示, 部分积总数为 $8 + 1 = 9$ 个。除了最下面的那个部分积之外, 每个部分积的位宽均为 17 比特。不考虑最下面那个部分积 (该部分积永远是非负数), 图2-5(b)展示了所有部分积均为负数时的符号位扩展情况, 即高位进行 1 扩展, 在对扩展产生的左上角大量的 1 进行累加后的部分积阵列如图2-5(c)所示。若图2-5(c)中存在部分积为正值, 则需要对部分积的符号位进行修正, 将高位的 1 扩展变回为 0 扩展, 方法如图2-5(d)所示, 引入 S 代表布斯码值的符号, $S = 0$ 表示布斯码值为正, $S = 1$ 表示布斯码值为负, 达到对部分积进行统一符号位扩展的效果。最后, 通过等价变换将2-5(d)中最上面的 \bar{S} 合并在部分积中, 降低阵列的层数, 如图2-5(e)所示, 得到最终的常用的符号位扩展方法。

对布斯算法来讲, 当乘数的位宽为偶数时, 与同位宽的无符号乘法相比, 补

码有符号数乘法的部分积个数会少一个。另外，在符号位扩展方面的区别是，在假设部分积均为负数而实际可能为正数、将大量的1累加后、需要对部分积的符号位进行修正时，不能再单一的引入布斯码值的符号，而是要引入布斯码值符号 S 和被乘数符号的同或（Exclusive-NOR）进行修正。图2-6展示了 16×16 补码乘法的基4布斯算法部分积符号位扩展优化方法的示意图。

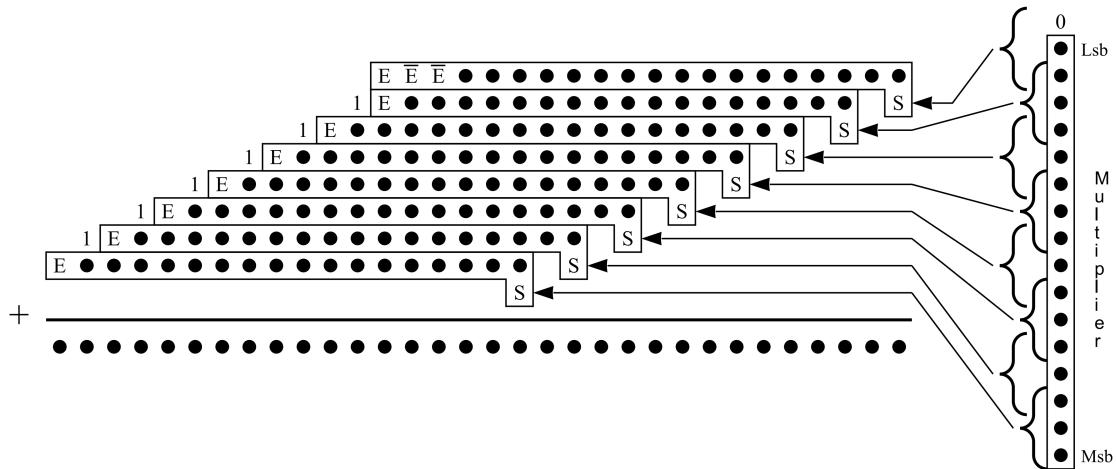


图 2-6 16×16 补码乘法的基4布斯算法部分积符号位扩展优化方法， E 表示布斯码值符号 S 和被乘数符号的同或

2.1.2 部分积的累加

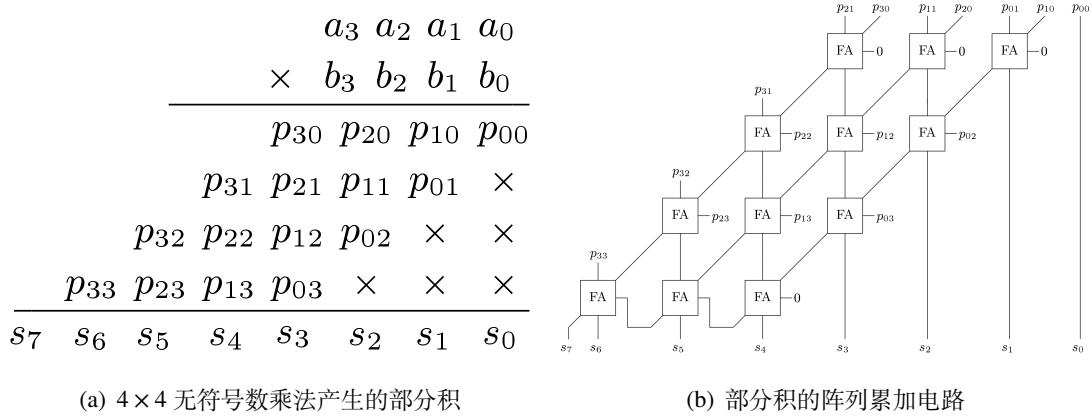
部分积产生后需要进行累加，这种累加本质上是一种多操作数的加法。一个直接的累加方法是用许多全加器（Full Adder, FA）形成阵列，被称为阵列累加，更为先进的方法是以进位保留或树结构的形式完成的。

阵列累加

图2-7展示了一个 4×4 无符号数乘法部分积的生成及阵列累加电路示意图，部分积的移位操作通过布线即可完成，整个结构可以被很轻易的压缩成一个矩形，使得版图非常紧凑。阵列累加可以直接生成最终结果，并不需要最终相加这一步操作。然而，阵列结构中部分积的累加是通过逐位进位实现的，关键路径较长，性能较差。

进位保留加法器

进位保留加法器（Carry Save Adder, CSA）可以高效的对多个（通常是3个及以上）二进制数进行求和，通常用于乘法器中部分积的累加。在二进制中，两个或三个比特相加产生的进位不会超过1，基于这个发现，CSA的基本思想是通

图 2-7 4×4 无符号数乘法部分积及对应的阵列累加电路示意图

过全加器将进位信号和求和信号保存下来，不断累加，直到最后通过一个向量合并加法器得到最终结果。图2-8展示了一个 4 操作数 8 比特位宽，最后通过超前进位加法器进行向量合并的 CSA 结构图。从图中可以看到，4 个 8 比特位宽的

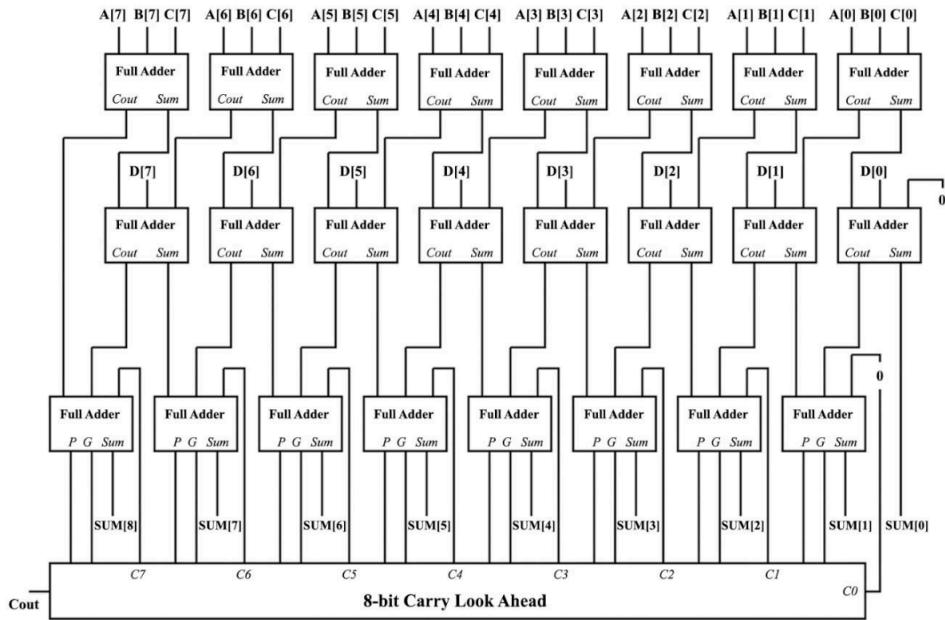


图 2-8 4 操作数 8 比特位宽，最后通过超前进位加法器相加的 CSA 结构图

操作数经过两级全加器的压缩变成了 2 个操作数，其中第一级全加器产生了 A 、 B 和 C 的进位及求和，之后和 D 一起被第二级全加器进行压缩，最后送给向量合并加法器进行最终求和。与阵列累加的方式相比，CSA 累加部分积的速度更快，性能更高。注意这里的 CSA 并不会对操作数进行分组后并行累加，而是逐级累加^[74]，这点与接下来讲的树形加法器有本质区别^①。

① https://blog.csdn.net/qq_26707507/article/details/106146612

树形加法器

不论是阵列累加还是 CSA 累加本质上都是通过排列全加器实现的，事实上，全加器也可以安排为树形，这样既能减少累加电路所需的全加器的数量，还能降低关键路径延迟。树形加法器的实现可以看作是并行的进位保留加法器，累加效率更高。常见的树形加法器包括华莱士树^[71]和达达树^[72]。

(1) 华莱士树

华莱士树 (Wallace tree) 加法器^[71]是由澳大利亚计算机科学家克里斯·华莱士 (Christopher Stewart Wallace) 于 1964 年设计的，被广泛用于乘法器中部分积的高效快速累加。其基本步骤如下：(a) 假设所有部分积都是同时生成的，第一步是将部分积每 3 个分为一组（不足 3 个保持），并将每组部分积的个数通过全加器和半加器压缩为 2 个；(b) 重复对部分积进行分组并压缩，直到只剩下两个部分积；(c) 相加最后两个部分积得到最终结果。图2-9展示了一个利用华莱士树加法器对 8×8 无符号数相乘产生的部分积进行累加的过程示意图，每个点表示一个比特，每个圈代表一个全加器或半加器（包含两个点的圈代表半加器，包含三个点的圈代表全加器），一共分组并压缩了 4 次，消耗了 15 个半加器和 38 个全加器，将 8 个部分积变成了 2 个部分积。与阵列累加方法和进位保留加法器相比，华莱士树加法器的速度很快，且位宽越大越明显，但它的缺点是电路结构非常不规则，难以获得高质量的版图设计。改进的华莱士树方法能够使用更少的半加器，降低设计的复杂度^[75]。全加器本质上是一个 3:2 压缩器，能够将乘法器中每组部分积的数目减少至三分之二。在进一步地设计 4:2 甚至更高比例的压缩器之后，结合华莱士树方法可以得到性能更高的乘法器^[76]。

(2) 达达树

达达树 (Dadda tree) 加法器是由计算机科学家 Luigi Dadda 于 1965 年发明的一种树形加法结构^[72]，与华莱士树方法类似，达达树也是采用全加器和半加器对部分积进行压缩，直到最后只剩下两个部分积。与华莱士树相比，达达树使用了数量最少的全加器以及半加器重构了树，且树的级数（深度）不变，在节省了硬件资源的情况下保证了相似的速度。但是，达达树产生的最后两个部分积的位宽可能会稍大。具体步骤如下^①：

- 部分积的累加过程由一个正整数序列 d_j 来控制： $d_1 = 2, d_{j+1} = \lfloor 1.5 \rfloor d_j$ ， $\lfloor \cdot \rfloor$ 表示向下取整；
- 初始的 j 应尽可能大并满足 $d_j < n$ ， n 是乘数的位宽（部分积阵列的高度）；
- j 逐步递减，且任意 j 下对部分积阵列从最低权重列开始遍历：(a) 若列

^① https://en.wikipedia.org/wiki/Dadda_multiplier

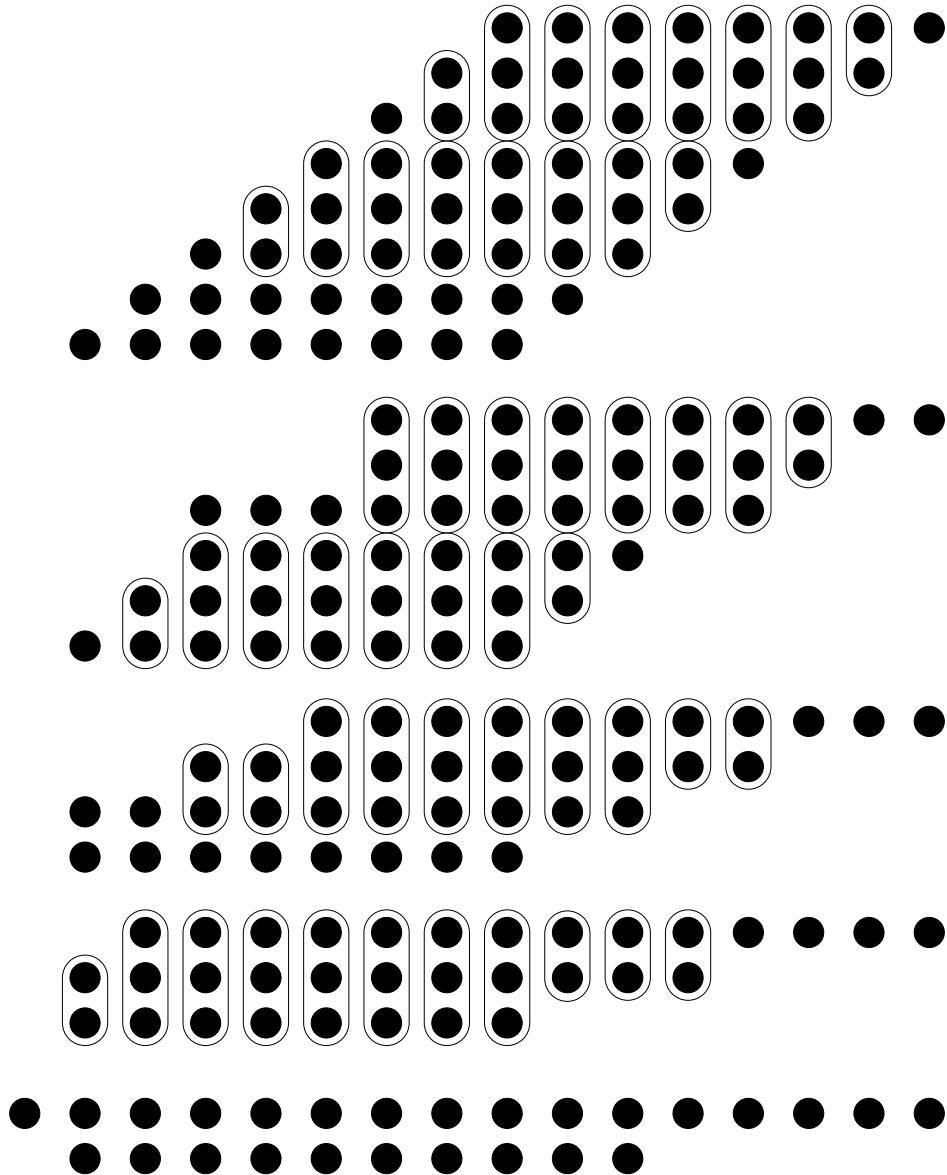


图 2-9 利用华莱士树加法器对 8×8 无符号数乘法部分积进行累加的示意图

高度小于或等于 d_j , 跳过; (b) 若列高度等于 $d_j + 1$, 运用半加器在该列最上面的两个比特, 产生进位及求和; (c) 若列高度大于 $d_j + 1$, 运用全加器在该列最上面的三个比特, 产生进位及求和, 并对该列重复 (a)、(b) 和 (c) 操作。注意列的高度可能需要考虑到前一列的进位;

- 累加结束时 $j = 1$, $d_j = 2$ (部分积只有两行), 之后使用一个向量合并加法器求得最终结果。

图2-10展示了一个运用达达树对 8×8 无符号数乘法生成的部分积进行压缩的过程示意图, 该树的深度与图2-9中的华莱士树的深度相同, 但使用了更少的全加器和半加器: 达达树使用了 35 个全加器、7 个半加器, 华莱士树使用了 38

个全加器、15个半加器。

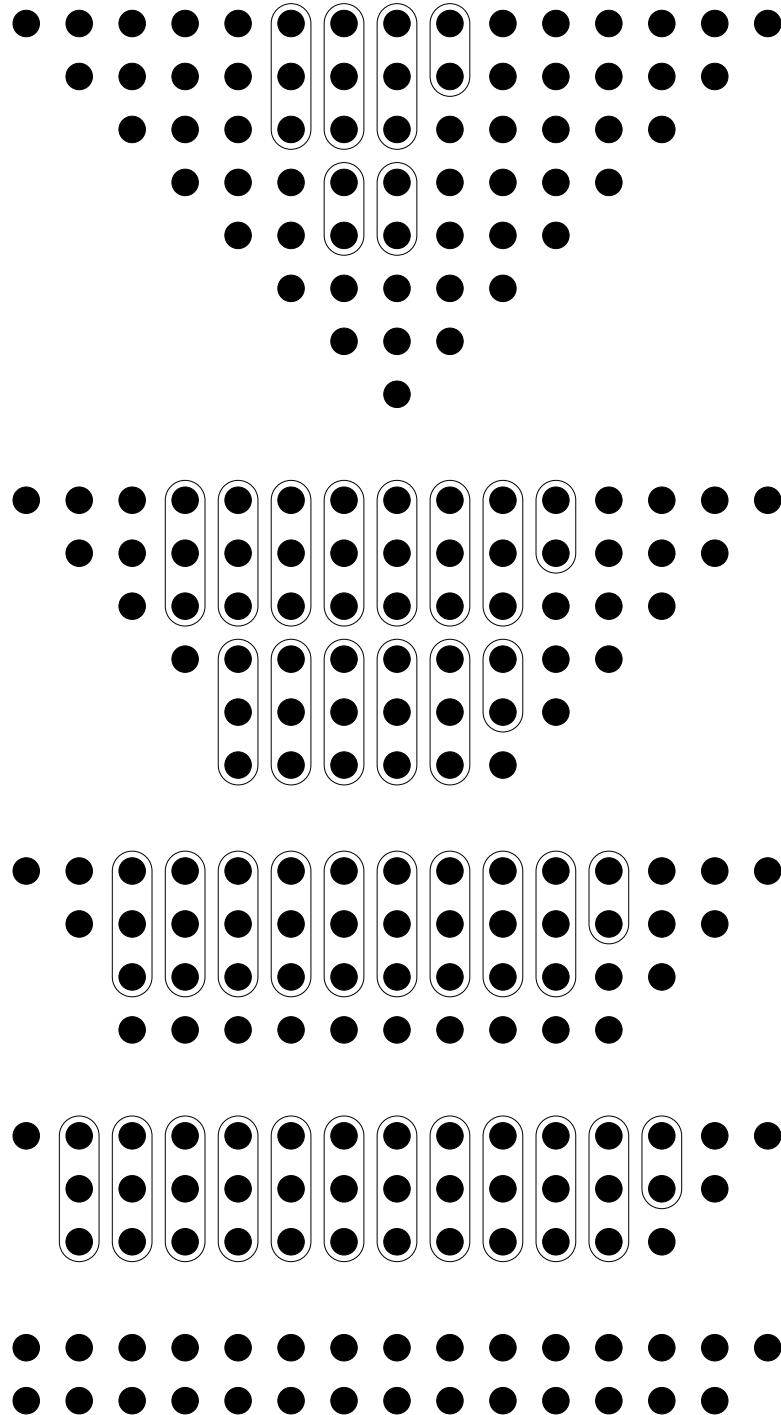


图 2-10 利用达达树加法器对 8×8 无符号数乘法部分积进行累加的示意图

2.1.3 最终相加

除了阵列累加方式以外，进位保存加法器、树形加法器等高速的部分积求和电路通常在最后都需要一个向量合并加法器（Vector-Merging Adder，VMA）进行最终两个部分积的相加，目前常见的 VMA 有以下几种结构：

行波进位加法器

行波进位加法器 (Ripple-Carry Adder, RCA) 又被称为逐级进位加法器，是由一系列全加器级联而成，优点是面积小、占用资源少，缺点是速度慢、效率低。在最好情况下，任何位宽的 RCA 都不需要传递进位信号也可以得到正确结果；但在最坏情况下，得到最终结果的延迟会随着位宽的增加而线性增大，从而限制了系统的运算速度。

超前进位加法器

当加法器的位宽较大时，由于 RCA 在最坏情况下每一级全加器的计算必须等待前一级的进位输出，导致其关键路径较长，效率较低。超前进位加法器 (Carry-Lookahead Adder) 的思想是并行计算每一级全加器的进位输出，本质上是数学公式推导的结果，原理如下：

假设 RCA 中第 i 级全加器的输入为 a_i, b_i, c_i ，进位输出为 c_{i+1} ，设 $p_i = a_i \oplus b_i$ ， $g_i = a_i b_i$ ，有：

$$\begin{aligned} c_{i+1} &= a_i b_i + c_i (a_i \oplus b_i) \\ &= g_i + c_i p_i \end{aligned} \tag{2.19}$$

若 $p_i = 1$ ，则 $g_i = 0$ ， $c_{i+1} = c_i$ ；若 $p_i = 0$ ，则 $c_{i+1} = g_i$ 。因此 p_i 和 g_i 分别被称为第 i 级加法进位的传播信号和生成信号。对 $c_i, c_{i-2}, c_{i-3}, \dots, c_1$ 使用式(2.19)，可将对 c_{i+1} 的求解转换为输入数的逻辑与 (AND) 和逻辑异或 (XOR) 操作（假设 $c_0 = 0$ ），避免了 RCA 中的进位依赖问题，实现了效率的提升。与 RCA 相比，CLA 的关键路径短，速度快，进位链计算依赖少。但在加法器位宽较大时，CLA 中高位的进位输出表达式涉及的变量较多，存在较大扇入扇出的问题；同时，组合逻辑电路的输入信号过多也会引起竞争冒险 (Race hazard)，产生毛刺 (Glitch)，影响系统的稳定性。所以在加法器位宽较大时，要想利用 CLA 提高进位效率，需要先对操作数进行划分，对每个部分实行 CLA，CLA 块间再通过级联或嵌套等方式进行连接^①，以避免大输入位宽逻辑门的产生。其中，采用级联对各 CLA 块进行连接的方式被称为分块 CLA，采用嵌套对各 CLA 块进行连接的方式被称为分级 CLA。灵活地对操作数进行划分并嵌套，能够得到许多不同的 CLA 结构，可在面积和速度之间进行权衡，存在一套能够简洁表示各种分级超前进位结构的符号体系，这一点会在后面的并行前缀加法器中详细讲述。最后需要注意的是，基于 CLA 方法实现的加法器的面积和复杂度通常会比同等位宽的 RCA 大。

^① <https://zhuanlan.zhihu.com/p/378267920>

进位旁路加法器

对(2.19)来说，一个 n 位 RCA 的最坏情况发生在 $p_{n-1}, p_{n-2}, \dots, p_0$ 均为 1 的时候（即 $p_{n-1}p_{n-2} \cdots p_0 = 1$ ），注意此时 $g_{n-1}, g_{n-2}, \dots, g_0$ 均为 0，式(2.19)变为：

$$c_{i+1} = c_i \quad (2.20)$$

进位旁路加法器（Carry-Skip Adder，为了与进位保存加法器区分，这里缩写为 CSKA，也叫 Carry-bypass adder）的思想便是加速该情况下进位链的传播。一个 n 位的 CSKA 包括一个 n 位的 RCA、一个 n 位的多输入与门（AND）、以及一个二选一的多路选择器（Multiplexer，MUX），图2-11展示了一个 4 比特 CSKA 的结构图， $p_0 \sim p_3$ 信号通过与门连接到 MUX 上作为选择信号，当 $p_0p_1p_2p_3 = 1$ 时， c_0 通过 MUX 直接输出，中间的 RCA 被旁路掉，大大减少了延迟。但是，与

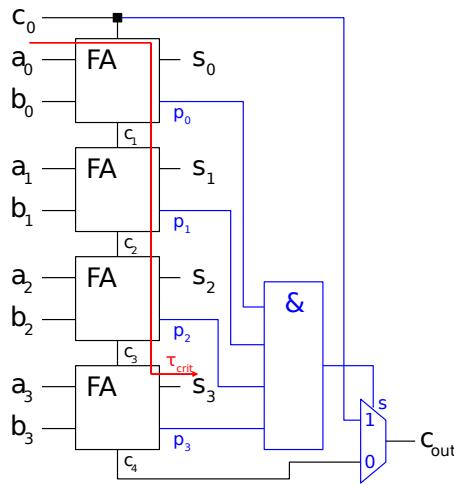


图 2-11 一个 4 比特 CSKA 的结构示意图，FA 代表全加器

RCA 相比，CSKA 并没有明显的性能改进（如图2-11中的关键路径 τ_{crit} 同样需要经过 4 个全加器），当位宽较大时，通过分块 CSKA（Block CSKA，BCSKA）的方法可取得较为显著的速度增益。若一个 n 比特的分块 CSKA 有 m 块 CSKA，每块 CSKA 的位宽均为 $\frac{n}{m}$ ，则该分块 CSKA 被称为固定大小分块 CSKA^①。

图2-12展示了一个通过级联 4 个 4 比特 CSKA 实现 16 比特加法的固定大小分块 CSKA 的结构图，其关键路径（红色线条 T_{critical} ）延迟包括头尾两个 4 比特 RCA 的延迟和中间两个旁路逻辑中 MUX 的延迟。从静态时序分析（Static Timing Analysis, STA）的角度看，图2-12的关键路径延迟比一个 16 比特的 RCA 更差，但 STA 得到的关键路径是伪路径，电路实际运行过程中并不会发生。固定大小分块 CSKA 真实的关键路径可通过以下过程来理解：输入同时到来后，每块

^① https://en.wikipedia.org/wiki/Carry-skip_adder

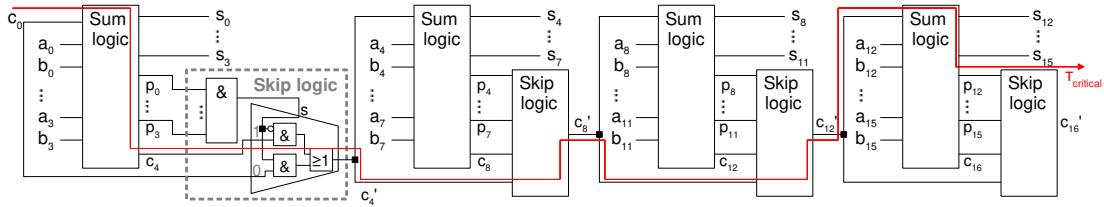


图 2-12 一个固定大小分块 CSKA 的例子：通过级联 4 个 4 比特 CSKA 实现 16 比特加法

CSKA 很快地被确定为是否处于旁路状态；之后所有的 CSKA 同时计算，假设首块 CSKA 没有被旁路，当首块 CSKA 的进位输出得到后，后面所有的 CSKA 要么处于旁路状态、要么也已计算完毕。因此固定大小分块 CSKA 的最坏情况是，进位信号需要通过首尾两个 RCA 和中间全部处于旁路状态的 CSKA 中的 MUX。通过调整块的大小和层级，分块 CSKA 的性能可得到进一步的优化。最后需要注意的是，与其他快速加法器如 CLA 不同，分块 CSKA 的性能仅在输入是某些情况下会获得提高，即速度的提高是概率性的。

进位选择加法器

另一种避免出现 RCA 中最坏情况下逐级进位的方法是预先考虑进位输入的两种可能的值（0 和 1），并提前计算针对这两种可能性的结果，一旦进位输入的值确定，正确的结果可以通过一个简单的 MUX 选出，这一设想的实现被称为进位选择加法器（Carry-Select Adder，为了与进位保存加法器区分，这里缩写为 CSEA）。CSEA 通常只包括 RCA 和 MUX，一个 4 比特位宽的 CSEA 的结构如图2-13所示，由于一个 RCA 的进位为 0，而另一个 RCA 的进位为 1，因此可通过实际进位输入决定哪个 RCA 的结果作为输出。与同等位宽的 RCA 相比，除了 MUX 以外，CSEA 消耗了两倍数量的全加器，是面积换性能的典型代表。

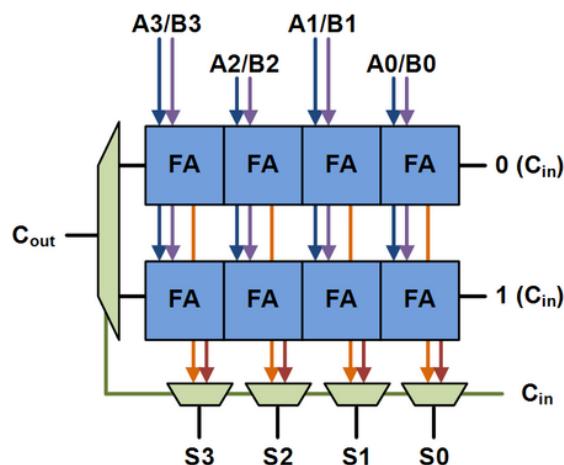


图 2-13 一个 4 比特 CSEA 的结构图，FA 代表全加器

完整的 CSEA 加法器需要对多个小的 CSEA 进行级联，并额外引入一个 RCA。若每个小 CSEA 的位宽相同，该加法器被称为线性 (Linear) CSEA；若每个小 CSEA 的位宽不同，该加法器被称为可变大小 (Variable-sized) CSEA，一种特殊的可变大小 CSEA 是平方根 (Square-root) CSEA。

(1) 线性进位选择加法器

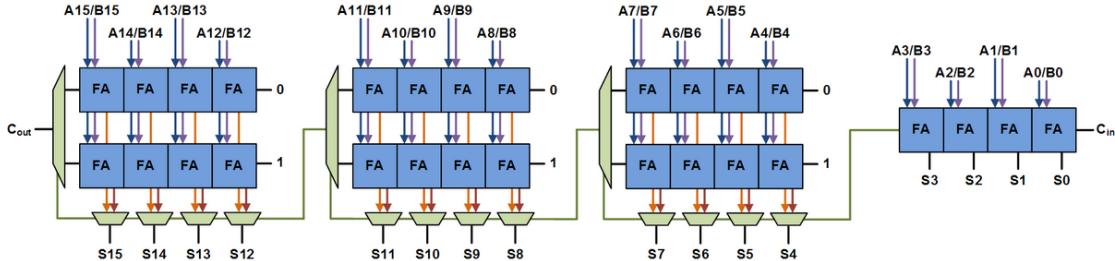


图 2-14 一个 16 比特线性进位选择加法器示意图

图2-14展示了16比特线性CSEA的结构示意图，该加法器由一个4比特RCA和三个4比特CSEA组成，其关键路径包括初始的4比特RCA和后续三个MUX，可通过以下过程理解：当初始的RCA计算完成时，后面所有的小CSEA中的RCA也均计算完成，只需等待真正的进位输入信号进行选择即可。通常来讲，对于一个 n 比特线性CSEA，每个小CSEA的位宽取 $\lfloor \sqrt{n} \rfloor$ 性能最好^①，这里 $\lfloor \cdot \rfloor$ 代表向下取整。

(2) 平方根进位选择加法器

若级联的每个小CSEA的位宽不同，且数值大小从低位到高位依次为2、3、4、…，初始RCA的位宽为2，则该可变大小CSEA被称为平方根CSEA，如图2-15所示，关键路径为初始的2位宽的RCA和后续的3个MUX。平方根CSEA

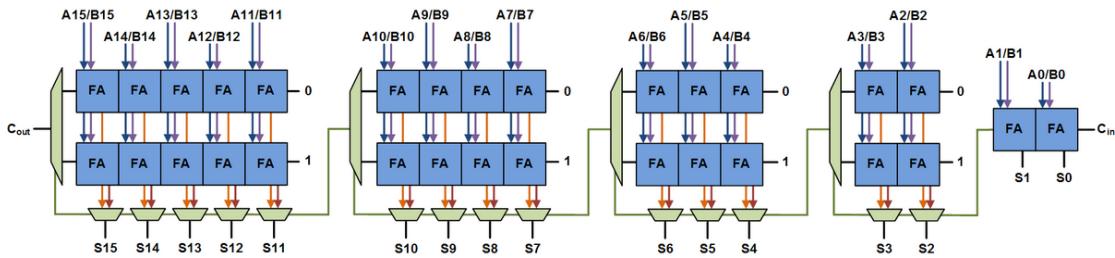


图 2-15 16 比特平方根进位选择加法器示意图

假设全加器和MUX的延迟相等，避免了线性CSEA中高位CSEA计算完成后等待进位输入信号选择的缺点，不论位宽多大，其关键路径总是初始的2位宽RCA加上后续的MUX，与线性CSEA相比取得了显著的性能提升。不过，一般

^① https://en.wikipedia.org/wiki/Carry-select_adder

情况下全加器和 MUX 的延迟不相等，因此需要根据实际情况决定使用哪种结构的 CSEA。

并行前缀加法器^[77]

由式(2.19)可得：

$$\begin{aligned}
 c_{i+1} &= \textcolor{blue}{g}_i + \textcolor{blue}{p}_i c_i \\
 &= g_i + p_i(\textcolor{red}{g}_{i-1} + \textcolor{red}{p}_{i-1} c_{i-1}) \\
 &= (\textcolor{blue}{g}_i + \textcolor{blue}{p}_i \textcolor{blue}{g}_{i-1}) + (\textcolor{blue}{p}_i \textcolor{blue}{p}_{i-1}) c_{i-1} \\
 &= (g_i + p_i g_{i-1}) + (p_i p_{i-1})(\textcolor{red}{g}_{i-2} + \textcolor{red}{p}_{i-2} c_{i-2}) \\
 &= (\textcolor{blue}{g}_i + \textcolor{blue}{p}_i \textcolor{blue}{g}_{i-1} + \textcolor{blue}{p}_i \textcolor{blue}{p}_{i-1} \textcolor{blue}{g}_{i-2}) + (\textcolor{blue}{p}_i \textcolor{blue}{p}_{i-1} \textcolor{blue}{p}_{i-2}) c_{i-2} \\
 &= \dots
 \end{aligned} \tag{2.21}$$

可以看到，若将 p 和 g 的定义从单比特拓展到连续的多比特，设 i, j 均是整数且 $0 \leq i < j$ ，有：

$$\begin{aligned}
 g_{[i,j]} &= g_j + p_j g_{j-1} + p_j p_{j-1} g_{j-2} + \dots + (p_j p_{j-1} p_{j-2} \dots p_{i+1}) g_i \\
 p_{[i,j]} &= p_j p_{j-1} p_{j-2} \dots p_i \\
 c_{j+1} &= g_{[i,j]} + p_{[i,j]} c_i
 \end{aligned} \tag{2.22}$$

即对加法块 $[i, j]$ 来讲同样存在进位的生成信号 $g_{[i,j]}$ 和传播信号 $p_{[i,j]}$ ，考虑到两者总是成对出现，可将其简写为二元对 $(g_{[i,j]}, p_{[i,j]})$ 。对两个相邻、部分重叠或完全重叠的加法块， $(g_{[i,j]}, p_{[i,j]})$ 有如下性质：

相邻：假设 k 是整数且 $i < k < j$ ，则块 $[i, k - 1]$ 与块 $[k, j]$ 相邻：

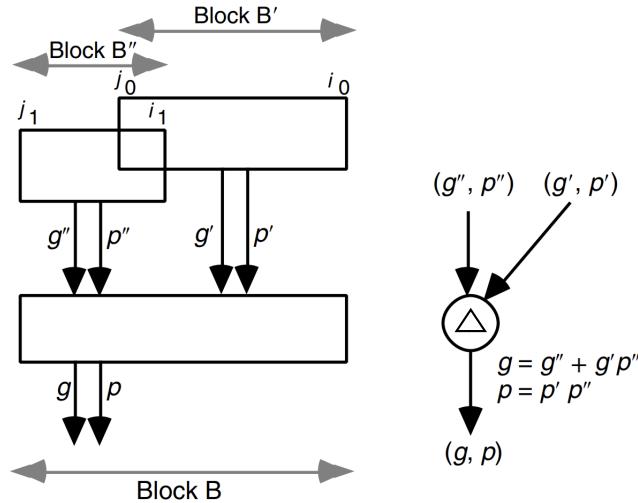
$$\begin{aligned}
 g_{[i,j]} &= g_{[k,j]} + g_{[i,k-1]} p_{[k,j]} \\
 p_{[i,j]} &= p_{[i,k-1]} p_{[k,j]}
 \end{aligned} \tag{2.23}$$

部分重叠：假设 h 是整数且 $i < k < h < j$ ，则块 $[i, h]$ 与块 $[k, j]$ 重叠：

$$\begin{aligned}
 g_{[i,j]} &= g_{[k,j]} + g_{[i,h]} p_{[k,j]} \\
 p_{[i,j]} &= p_{[i,h]} p_{[k,j]}
 \end{aligned} \tag{2.24}$$

完全重叠：

$$\begin{aligned}
 g_{[i,j]} &= g_{[i,j]} + g_{[i,j]} p_{[i,j]} \\
 p_{[i,j]} &= p_{[i,j]} p_{[i,j]}
 \end{aligned} \tag{2.25}$$

图 2-16 合并两个相邻或部分重叠的加法块 B' 、 B'' 的进位信号

如图2-16所示，假设两个加法块 B' 和 B'' 相邻或重叠，则可以通过合并两对进位信号 (g', p') 和 (g'', p'') 来得到块 B 的进位信号 (g, p) ，假设该合并操作符号为 Δ ，有：

$$(g, p) = (g', p') \Delta (g'', p'') \quad (2.26)$$

其中

$$\begin{aligned} g &= g'' + g' p'' \\ p &= p' p'' \end{aligned} \quad (2.27)$$

注意 Δ 满足结合律，不满足交换律：

$$\begin{aligned} ((g', p') \Delta (g'', p'')) \Delta (g''', p''') &\equiv (g', p') \Delta ((g'', p'') \Delta (g''', p''')) \\ (g', p') \Delta (g'', p'') &\not\equiv (g'', p'') \Delta (g', p') \end{aligned} \quad (2.28)$$

这里假设块 B''' 与块 B'' 相邻或重叠，进位信号为 (g''', p''') 。

基于此，可以定性地描述出进位问题：假设加法器的位宽为 n ，给定 (g_0, p_0) 、 (g_1, p_1) 、 (g_2, p_2) 、…、 (g_{n-1}, p_{n-1}) ，通过并行地计算

$$(g_0, p_0) \Delta (g_1, p_1) \Delta (g_2, p_2) \Delta \cdots \Delta (g_{n-1}, p_{n-1}) \quad (2.29)$$

能够得到所有的 $(g_{[0,0]}, p_{[0,0]})$ 、 $(g_{[0,1]}, p_{[0,1]})$ 、 $(g_{[0,2]}, p_{[0,2]})$ 、…、 $(g_{[0,n-1]}, p_{[0,n-1]})$ ，即 c_1 、 c_2 、 c_3 、…、 c_n ，且不同的并行方法对应不同的进位逻辑硬件实现，具有不同的面积和性能，称为并行前缀进位（Parallel Prefix Carry, PPC），得到的加法器被称为并行前缀加法器。PPC 可由树状图进行表示，图2-17展示了标准超前进位加法器 CLA 和标准行波进位加法器 RCA 的 PPC 树状图。可以看到，CLA

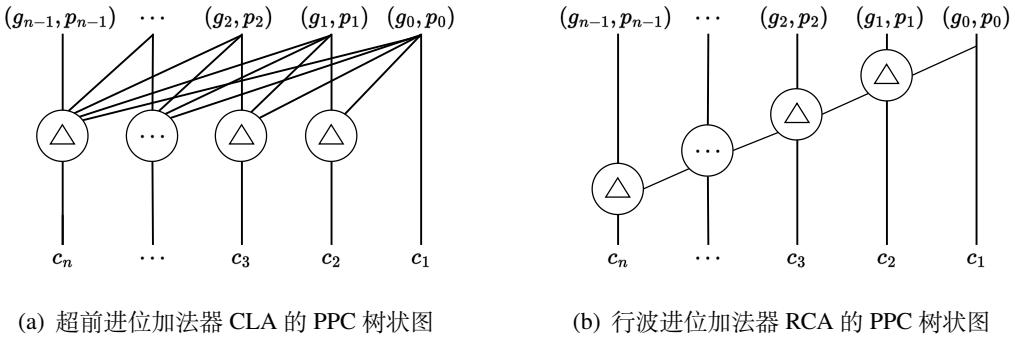


图 2-17 标准 CLA 和标准 RCA 的 PPC 树状图

的 PPC 树状图只有一级，关键路径短，但计算量大；CLA 的 PPC 树状图有 n 级，关键路径长，但使用了大量的中间节点，计算量小。因此，并行前缀加法器的核心在于如何权衡面积和延迟，这为并行前缀加法器的设计带来了非常大的灵活性^[78]：比如 1960 年的 Sklansky 加法器^[79]，特点是级数少、扇出高；1973 年的 Kogge-Stone 加法器^[80]，优点是逻辑级数和扇出都非常小，缺点节点数量非常多，布线拥塞度高；1980 年的 Ladner-Fisher 加法器^[81]，其结构与 Sklansky 加法器^[79]一致，原因是 Ladner-Fischer 形式化了进位问题的求解，Sklansky 加法器是其中一种方案；1982 年的 Brent-Kung 加法器^[82]，优点是扇出非常小，节点也较少，缺点是逻辑级数较多，即通过增加额外的逻辑级数来缓解扇出压力；1987 年的 Han-Carlson 加法器^[83]，混合了 Brent-Kung 结构和 Kogge-Stone 结构，更有利干硬件实现；1999 年 Knowles 提出可以从逻辑深度、布线拥塞度和面积三个方面对进位网络的设计进行权衡^[84]；2003 年 Harris 提出了一个有趣的 3-D 结构图来对已有的并行前缀加法器进行分类^[85]，同时提出了一个新的进位结构；其余的前缀加法器包括 Ling 型加法器^[86]和具有多扇入节点的 Beaumont-Smith 加法器^[87]。现代 EDA (Electronic Design Automation) 综合工具也往往根据用户约束来实现具有并行前缀结构的加法器。由于大位宽下并行前缀网络的搜索空间巨大，英伟达利用强化学习的方法设计出了面积更小、速度更快的加法器^[88]；也有工作在观察到 EDA 流程中逻辑综合和物理综合的不统一后^[89]，通过图神经网络的方式来进行优化^[90]。

2.2 对数乘法器

1962 年，John N. Mitchell 提出了一个相当有趣的算法^[66]：在二进制下，可以通过加法来近似得到两个无符号非零定点数的乘积，最大误差不超过 $\frac{1}{9}$ ，且编程实现非常简洁，原理如下：

对于式(2.1), 假设 $R = 2$ 、 $m = 0$, 两个 n 比特二进制无符号整数 X 和 Y :

$$X = x_{n-1}x_{n-2}x_{n-3} \cdots x_1x_0 \quad Y = y_{n-1}y_{n-2}y_{n-3} \cdots y_1y_0 \quad (2.30)$$

若 X 和 Y 均非零 (正整数), 有:

$$\begin{aligned} x_{n-1} + x_{n-2} + x_{n-3} + \cdots + x_1 + x_0 &= 1 \\ y_{n-1} + y_{n-2} + y_{n-3} + \cdots + y_1 + y_0 &= 1 \end{aligned} \quad (2.31)$$

则 X 和 Y 的十进制值为:

$$V(X) = \sum_{i=0}^{k_X} x_i 2^i \quad V(Y) = \sum_{i=0}^{k_Y} y_i 2^i \quad (2.32)$$

其中 k_X 和 k_Y 分别代表 X 和 Y 中最高位的 1 的权重值^[91], 乘积 P 的十进制值:

$$\begin{aligned} V(P) &= V(X)V(Y) \\ &= \sum_{i=0}^{k_X} x_i 2^i \times \sum_{i=0}^{k_Y} y_i 2^i \\ &= 2^{k_X} \left(1 + \sum_{i=0}^{k_X-1} x_i 2^{i-k_X} \right) \times 2^{k_Y} \left(1 + \sum_{i=0}^{k_Y-1} y_i 2^{i-k_Y} \right) \\ &= 2^{k_X+k_Y} \left(1 + \sum_{i=0}^{k_X-1} x_i 2^{i-k_X} \right) \left(1 + \sum_{i=0}^{k_Y-1} y_i 2^{i-k_Y} \right) \\ &= 2^{k_X+k_Y} (1 + x')(1 + y') \end{aligned} \quad (2.33)$$

其中

$$x' = \sum_{i=0}^{k_X-1} x_i 2^{i-k_X} \quad y' = \sum_{i=0}^{k_Y-1} y_i 2^{i-k_Y} \quad (2.34)$$

则:

$$\log_2(V(P)) = k_X + k_Y + \log_2(1 + x') + \log_2(1 + y') \quad (2.35)$$

易知 $x, y \in [0, 1)$, 在这里, Mitchell 做了一个相当果断而美妙的近似, 那就是利用 $\log_2(1 + x') \approx x'$ 和 $\log_2(1 + y') \approx y'$ 将式(2.35)变为:

$$\log_2(V(P)) \approx k_X + k_Y + x' + y' \quad (2.36)$$

则乘积 P :

$$V(P) \approx 2^{(k_X+k_Y)} \cdot 2^{(x'+y')} \quad (2.37)$$

这里再做一次近似：

$$x' + y' \approx \begin{cases} \log_2(1 + x' + y'), & x' + y' < 1, \\ 1 + \log_2(x' + y'), & x' + y' \geq 1. \end{cases} \quad (2.38)$$

则式(2.37)变为：

$$V(P) \approx \begin{cases} 2^{(k_x+k_y)}(1 + x' + y'), & x' + y' < 1, \\ 2^{(k_x+k_y+1)}(x' + y'), & x' + y' \geq 1. \end{cases} \quad (2.39)$$

式(2.39)即为 Mitchell 对数乘法器的最终形式，只需要加法、移位和检测电路即可完成两数相乘，可以证明 Mitchell 算法的最大误差不超过精确值的 $\frac{1}{9}$ 。

在 C++ 中，Mitchell 算法有一个非常简单的实现^①，单精度 (Single precision) 浮点数运算的代码如算法1所示^[92]，原理如下：

Algorithm 1: Mitchell 算法在 C++ 中的简单实现

```

1 float int mitchell_mul (const float a, const float b) {
2     int c = *(int*)&a + *(int*)&b - 0x3f800000;
3     return *(float*)&c;
4 }
```

C++ 中 int 类型通常是 32 位，同时 IEEE 754 标准规定采用 32 个比特表示单精度浮点数，如图2-18所示，其中最高位表示正负，之后 8 位表示科学计数法的指数，最后 23 位表示科学计数法的小数，注意指数部分需要加上偏移量 127。对于两个单精度浮点数 a 和 b，*(int*)&a 和 *(int*)&b 其实就是把 a 和 b 对应的二进制表示拿出来，当作普通的 int 类型将两者相加，对应式(2.36)，这时指数部分还多出一个偏移量，所以要减去这个偏移量，由于偏移量是 127，并且后面还有 23 位，所以要减去常数 127×2^{23} ，十六进制为 0x3f800000，最后将结果恢复为单精度浮点数。对双精度 (Double precision) 浮点数来说，在引入 64 比特位宽的 long int 类型和正确的偏移量 ($1023 \times 2^{52} = 0x3feffffb9f6e8000$) 之后，算法1同样适用（已验证）。

Mitchell 算法能够将乘法近似转化为加法实现，这对需要大量乘法并对误差有容忍性的神经网络应用带来了计算量优化的可能，比如有篇 NeurIPS 2020 的工作便在“ImageNet+ResNet50”中直接将神经网络中的乘法换成 Mitchell 近似的加法形式，准确率只有轻微的下降，甚至可能不下降^[92]，而这也并不是 Mitchell 近似在深度学习中第一次被讨论^[93-94]。

^① <https://kexue.fm/archives/7991>

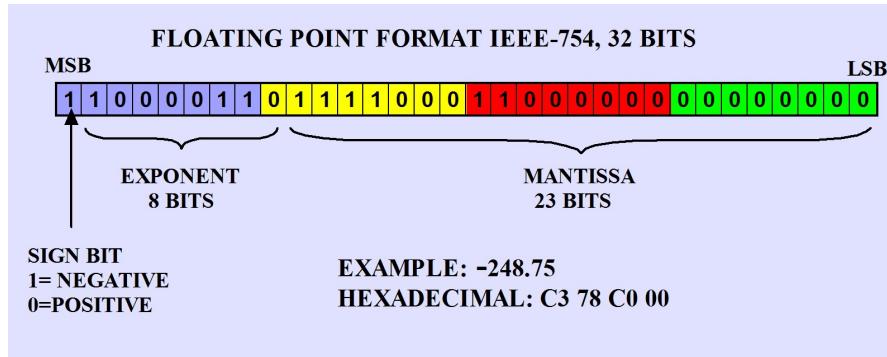


图 2-18 IEEE 754 单精度浮点数标准

2.3 近似电路的误差指标

近似电路通常是指近似的算术单元，如加法器、减法器、乘法器、除法器等，这些近似模块在计算中可能会产生误差，用来衡量误差的最基本的两个指标分别是误差率（Error Rate, ER）和误差距离（Error Distance, ED)^[57]。ER 表示产生错误结果的概率，ED 代表近似结果和精确结果之间的算术差异。假设在某输入下近似电路的结果是 M' ，精确电路的结果是 M ，那么该输入下误差距离为：

$$ED = |M' - M| \quad (2.40)$$

相对误差距离 (Relative ED, RED) 表示近似结果与准确结果的相对算术差异，则

$$RED = \frac{ED}{M} \quad (2.41)$$

ED 和 RED 是衡量近似电路在某输入下误差的两个重要指标。

当考虑所有的输入情况时，可由平均误差距离（Mean ED，MED）和平均相对误差距离（Mean RED，MRED）来衡量近似电路和精确电路之间的整体算术差异，MED 和 MRED 被定义为：

$$MED = \sum_{i=1}^N ED_i \cdot P(ED_i) \quad (2.42)$$

$$MRED = \sum_{i=1}^N RED_i \cdot P(RED_i) \quad (2.43)$$

其中 N 是所有输入情况的总数, ED_i 和 RED_i 分别代表输入是第 i 种情况下的误差距离 ED 和相对误差距离 RED, $P(ED_i)$ 和 $P(RED_i)$ 分别代表 ED_i 和 RED_i 发生的概率, 即输入取第 i 种情况的概率。MED 也被叫做平均绝对误差 (Mean Absolute Error, MAE)。归一化平均误差距离 (Normalized MED, NMED) 被定义为 MED 除以精确电路在所有输入情况下的最大值, 常被用来比较同一近似设计方法在不同输入位宽下的误差表现。

均方误差 (Mean Squared Error, MSE) 和均方根误差 (Root MSE, RMSE) 也被广泛用于衡量近似电路和精确电路之间的算术误差幅度，它们被定义为：

$$MSE = \sum_{i=1}^N ED_i^2 \cdot P(ED_i) \quad (2.44)$$

$$RMSE = \sqrt{MSE} \quad (2.45)$$

另外，平均误差被定义为所有可能输入情况下 $M' - M$ 的平均值，归一化平均误差被定义为平均误差除以精确电路在所有输入情况下的最大值。最后，最坏情况误差 (Worst Case Error, WCE) 反映了可能的最大 ED。

以上提到的衡量近似电路误差的指标均适用于近似乘法器。

2.4 本章小结

本章首先介绍了精确定点数乘法器的三个运算过程：部分积的生成、累加、最终相加，以及针对每个过程不同的实现方法。其中，无符号数相乘的部分积可由与门直接产生，补码有符号数乘法的部分积根据设计方法的不同有多种生成办法，目前使用最广泛的产生方式是基于 Baugh-Wooley 算法和基 4 的布斯算法；对部分积的累加来讲，通过将全加器排列为树形，并采用进位保存的思想可得到较为高速的累加阵列；累加结束后的部分积数量减少为 2 个，需要一个多位宽的向量加法器完成最终相加，可根据需求选择行波进位加法器、超前进位加法器、进位选择加法器、并行前缀加法器等不同结构完成最终结果的计算。本章紧接着介绍了由 Mitchell 发明的对数乘法器，该乘法器可将乘法转变为加法，大大降低了计算量，误差最大不超过 $\frac{1}{9}$ ；最后介绍了用来衡量近似电路误差的不同指标，这些指标适用于常见的近似算术单元。

第3章 考虑输入分布和极性的 ASIC 近似乘法器设计

3.1 研究背景

近年来，随着物联网设备（Internet of Things, IoT）的快速发展及其严格的资源限制，在IoT设备中部署如图像处理（Image processing）、数据挖掘（Data mining）、多媒体技术（Multimedia technology）、深度学习等许多计算密集型应用成为了一个严峻的挑战^[95]。幸运地是，这类应用往往具有容错性，采用近似乘法器对被频繁调用的乘法操作在硬件上进行优化，一方面能够提升处理速度、减少资源消耗，另一方面又不会带来明显的输出质量的下降。

3.2 国内外研究现状

在数字电路系统中，乘法器作为一个能够实现两个数相乘的运算电路，已经被研究了几十年，其性能和功耗依赖于设计的电路结构（见2.1）。为了结合容错类应用进一步提高效率，人们提出了近似乘法器的概念。与精确乘法器相比，近似乘法器（Approximate multiplier）通常具有更小的面积、更低的延迟、更优的功耗，但在某些情况下会输出不正确的结果。目前有关近似乘法器在功能近似层面（见1.1.4有关功能近似的定义）的设计方法大致可以分为四类：手工设计、数学转换近似、自动化方法和近似电路综合，下面分别进行介绍。

3.2.1 手工设计

手动化简乘法器的电路或门级网表的方法被称为手工设计方法，这种方法的优点在于通常能在不同位宽的乘法器之间迁移，缺点是费时费力，且无法灵活地根据应用的错误容忍程度进行调整，效果不好。

较为经典的一篇有关手工设计近似乘法器的工作是由Kulkarni等人^[96]于2011年发表的，他们观察到 2×2 精确乘法器只有在输入是 3×3 的时候才需要4个比特表示输出，于是修改了卡诺图（Karnaugh map），将 $3 \times 3 = 9$ 变为了 $3 \times 3 = 7$ ，输出从原先需要4个比特减少到了3个比特，乘法器的门数降低了37.5%，更有利硬件实现。与精确的 2×2 乘法器相比，该近似乘法器在16种可能的输入情况中，只有一种情况输出是错误的，误差率ER是 $\frac{1}{16}$ ，由式(2.40)得

误差距离 ED 为 $|7 - 9| = 2$ 。经过 EDA 软件评估，近似后的 2×2 乘法器的面积相比精确版本减少了一半，并且拥有更短的关键路径。同时，由于近似后乘法器的开关电容更小，在相同工作频率下的动态功耗也更低。更大位宽的乘法器可通过拆分、由近似的 2×2 乘法器搭建起来，且搭建时可对高位输入引入精确乘法来降低误差。

文献^[97]对行波进位加法器 RCA 中的全加器进行了重新设计，将进位输入信号替换了前一级全加器的非进位输入，使进位链最多只能传播两位，优化了关键路径，降低了计算延迟，同时将截断进位链产生的误差保存了下来。优化后的全加器被排列成树形用于乘法器中部分积的累加及最终相加，并在最终相加后根据需要引入不同等级的误差补偿，提高乘法器的精度。精确的误差补偿需要考虑所有全加器保存的误差并相加，为了降低设计复杂度，可以仅考虑最高几位全加器保存的结果，并通过或 (OR) 操作运算后加到最终的输出上。图3-1(a)代表论文提出的考虑两级输入的近似全加器电路图，其中 S_i 和 E_i 分别代表求和的输出和保存的误差。图3-1(b)展示了基于重新设计的近似全加器单元得到的近似乘法器的结构示意图，其中 A1-A7 代表树形的近似全加器阵列，一方面对部分积进行累加，一方面保留产生的误差，最后的补偿步骤选择最高 4 位全加器的结果进行运算。需要注意的是，因为保存的误差是通过或操作之后相加到结果上，所以即使考虑所有全加器保存的误差也无法将误差率降到 0 (正确的做法不是或操作而是相加)。

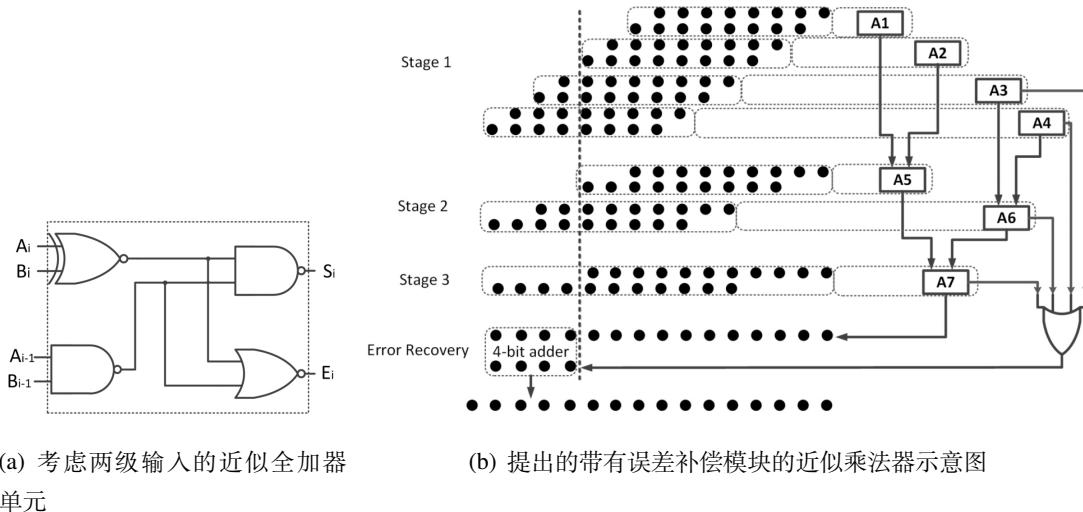


图 3-1 近似全加器单元和近似乘法器结构示意图

文献^[98]在部分积生成后、累加前，先通过逻辑或 (OR) 操作有选择地对部分积进行了一次压缩，大大降低了部分积阵列的规模，减轻了后续的累加压力。除了 2 输入的或操作之外，更大输入的或操作也可以被使用，好处是部分积规模

下降的更多，坏处是引入的误差更大。图3-2展示了利用3输入或门对部分积阵列进行压缩的过程，首先对部分积进行分组，每3个部分积一组，不足3个的可保持或用更小输入的或门，同时避免操作高位部分积以减少误差，压缩后的部分积个数由8个减少为3个，降低了62.5%。该论文提出的方法适用于不同位宽的乘法器，通过Synopsys Design Compiler综合工具对基于此方法设计的128比特的近似乘法器进行评估发现，与精确乘法器相比，面积减少了45%，关键路径减少了65%。

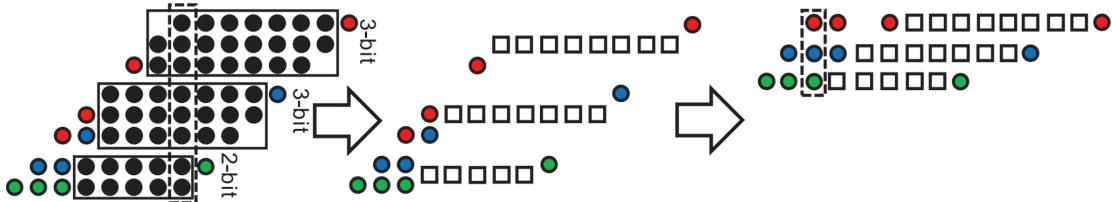


图 3-2 利用 3 输入或门对部分积阵列进行压缩

文献^[99]提出了一个用于两个无符号数相乘的动态无偏可配置乘法器DRUM，其基本思想是通过动态识别乘数和被乘数中权重最大的1的位置、挑选出以该位置开头的几个连续比特并对尾部数据进行四舍五入之后，利用一个小位宽的乘法器完成核心运算。整个电路包括检测器、四舍五入编码器、小位宽乘法器以及最后的移位器，可通过配置小乘法器的位宽来实现不同的精度。同时，该乘法器是无偏的，即平均误差为0（见2.3有关平均误差的定义）；该乘法器也是对称的，即对任意输入，交换乘数和被乘数后结果一致，没有极性。图3-3展示了DRUM原理的一个示例，在对操作数中挑出的部分比特四舍五入并执行乘法后，通过移位操作得到最终的近似结果。

x	0001 0111 0 100 1101
	0000 0001 0 101 1 010
x	10111 1
	10101 1
	0111 1110 0101
	0000 0000 0001 1111 1001 0100 0000 0000

图 3-3 DRUM 的原理的一个示例：通过一个小位宽乘法器对操作数的部分比特执行乘法

文献^[91]对式(2.33)进行了变形：

$$V(P) = 2^{k_x+k_y} (1 + x') (1 + y')$$

$$\begin{aligned}
&= 2^{k_x+k_y} (1 + x' \cdot y' + x' + y') \\
&\approx 2^{k_x+k_y} (1 + x'_{App} y'_{App} + x'_T + y'_T)
\end{aligned} \tag{3.1}$$

其中, x'_{App} 、 y'_{App} 、 x'_T 、 y'_T 均类似 DRUM^[99] 只对操作数取部分比特进行计算, 但与 DRUM 相比没有对尾部数据进行四舍五入而是直接丢弃, 这样原始乘法被转换成了一个小位宽乘法和两个加法, 减少了电路面积。同时, x'_{App} 、 x'_T (或 y'_{App} 、 y'_T) 的位宽可以不同, 配置空间更大, 能够满足更多种类的应用需求。

手工设计近似乘法器的方法众多, 相当大一部分工作着重于对部分积的累加进行优化, 这里只列举了几个比较典型和巧妙的示例, 接下来介绍通过数学转换近似来实现乘法。

3.2.2 数学转换近似

数学转换近似通常需要问题满足一定的数学特性, 通过将乘法转换为成本更低的操作来达到降低设计复杂度的目的, Mitchell 提出的对数乘法器^[66] (见2.2) 可以看作是通过此方法设计近似乘法器的一个例子。乘法是一个非线性操作, 文献^[100] 通过分段线性函数 (Piece-wise linear function) 来近似浮点数乘法中的尾数乘法器, 原理如下:

假设线性基空间为 $\{1, x, y, x^2, y^2\}$, 乘法器的输入是 x 和 y , 范围分别是 $[x_1, x_2]$ 和 $[y_1, y_2]$ ($x_2 > x_1 \geq 0$, $y_2 > y_1 \geq 0$), 则近似乘法器的输出 z_{approx} 为:

$$z_{approx} = k_0 + k_1 x + k_2 y + k_3 x^2 + k_4 y^2 \tag{3.2}$$

式中 $k_0 - k_4$ 均是待定常数。由(2.40)得误差距离 ED 的平方为:

$$ED^2 = |z_{approx} - xy|^2 \tag{3.3}$$

通过最小化(3.3)可以得到 $k_0 - k_4$ 的解析解为:

$$[k_0, k_1, k_2, k_3, k_4] = \left[-\frac{(x_1 + x_2)(y_1 + y_2)}{4}, \frac{y_1 + y_2}{2}, \frac{x_1 + x_2}{2}, 0, 0 \right] \tag{3.4}$$

即将精确乘法转换为了线性操作:

$$xy \approx z_{approx} = k_0 + k_1 x + k_2 y \tag{3.5}$$

其在基空间 $\{1, x, y, x^2, y^2\}$ 上误差距离 ED 的平方最小, 且有以下几个性质: (1) 对(3.5)来说, 当 $\{x, y\}$ 取 $\{x_1, y_1\}$ 、 $\{x_1, y_2\}$ 、 $\{x_2, y_1\}$ 、 $\{x_2, y_2\}$ 时近似乘法器的误差最大; (2) 所得到的近似乘法器是无偏的, 即平均误差为 0 (见2.3有关平均误差的定义):

$$\int_{x_1}^{x_2} \int_{y_1}^{y_2} (k_0 + k_1 x + k_2 y - xy) dx dy = 0 \tag{3.6}$$

(3) 若对精确乘法按照此论文所提出的方法进行分段线性近似，即将输入范围 $[x_1, x_2] \times [y_1, y_2]$ 划分为若干个子区域，每个子区域也将乘法转换成如(3.5)所示的线性操作，则每个子区域相等时整个乘法器的误差距离 ED 的平方最小，且最小为：

$$ED_{\min}^2 = \frac{(x_2 - x_1)^3(y_2 - y_1)^3}{144n^2} \quad (3.7)$$

n 是子区域的个数。因此可以通过不断划分面积相等的子区域来提高乘法器的精度。

对于浮点数来讲，尾数的取值范围为 $[1, 2)$ ，则在不分割的情况下，得到的近似乘法器为：

$$z_{approx}^0 = 1.5x + 1.5y - 2.25 \quad (3.8)$$

为了提高精度且方便硬件实现，对定义域进行划分，第 i 次划分后共生成 4^i 个子区域，每个子区域的面积相等。假设乘法器第 i 划分后的输出比第 $i-1$ 次划分后的输出差了 Δz_{approx}^i ，则：

$$\Delta z_{approx}^i = z_{approx}^i - z_{approx}^{i-1} \quad (3.9)$$

经过证明，对任意的 i ， Δz_{approx}^i 都可由一个异或 XOR、一个移位、2 个加法和几个反向逻辑实现，大大降低了误差补偿的成本，且第 i 次划分后乘法器的均方误差由式(2.44)得：

$$MSE = \frac{1}{9 \times 16^{i+1}} \times \frac{1}{2^{23}} \quad (3.10)$$

这里假设输入均匀分布。图3-4展示了论文提出的近似浮点数乘法器的整体结构图和不同划分等级下的误差分布，可以看到随着划分次数的增加，误差在指数级减小。

尽管该方法是针对浮点数乘法器提出的，但是经过简单的修改也可将其用于定点数乘法器。

3.2.3 自动化方法

将设计近似乘法器的问题建模成搜索问题，能够利用计算机在短时间内生成大量具有不同精度和不同性能的近似乘法器。在自动化方法中，由捷克布尔诺理工大学 (Brno University of Technology) 的可进化硬件 (Evolvable HardWare, EHW) 研究小组开发的、基于笛卡尔遗传规划 (Cartesian Genetic Programming, CGP)^[101-102] 的遗传算法 (Genetic algorithm) 取得了相当出色的效果。2016 年，EHW 的研究人员利用 CGP 设计了面向人工神经网络的近似乘法器^[103]，以精度下降小于 2.80% 的代价节省了 91% 的功耗。

CGP 起源于 Miller 等人于 1997 年开发的一种进化数字电路的方法^[104]，在 1999 年第一次出现^[105]，其通用形式于 2000 年被提出^[106]。在 CGP 中，一个电

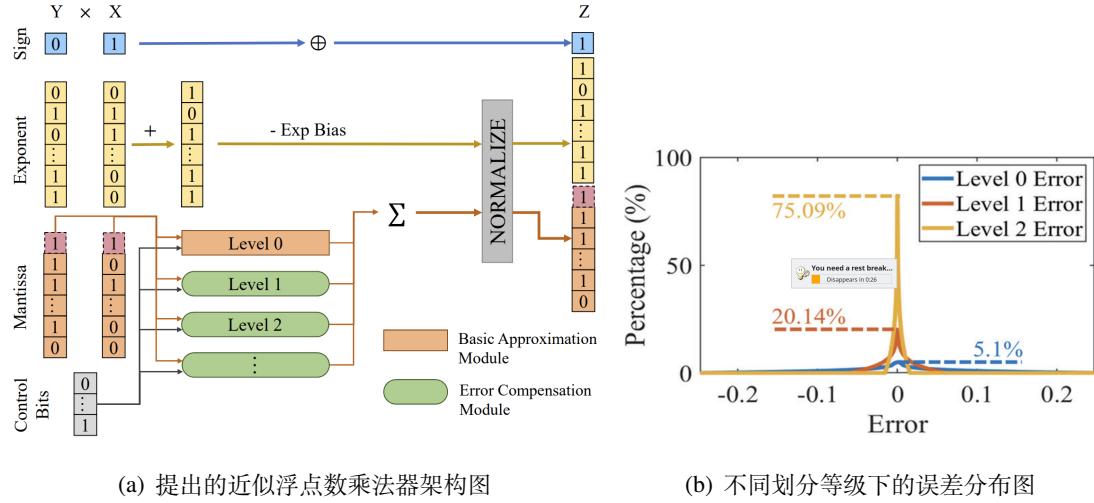


图 3-4 论文所提出的近似浮点数乘法器架构图和不同划分等级下的误差分布

路由器节点组成的二维有向无环图（Directed Acyclic Graph, DAG）进行表示（节点可以代表一个门或一个模块），每个节点由一串整数组成，分别代表该节点从哪个节点获得数据、节点对数据执行什么操作，输出没有被使用的节点可以被忽略。将所有的整数按照顺序排列，并在最后加上表示电路输出的节点编号便是电路对应的 CGP 形式。图3-5展示了具有 5 比特输入、2 比特输出的组合逻辑

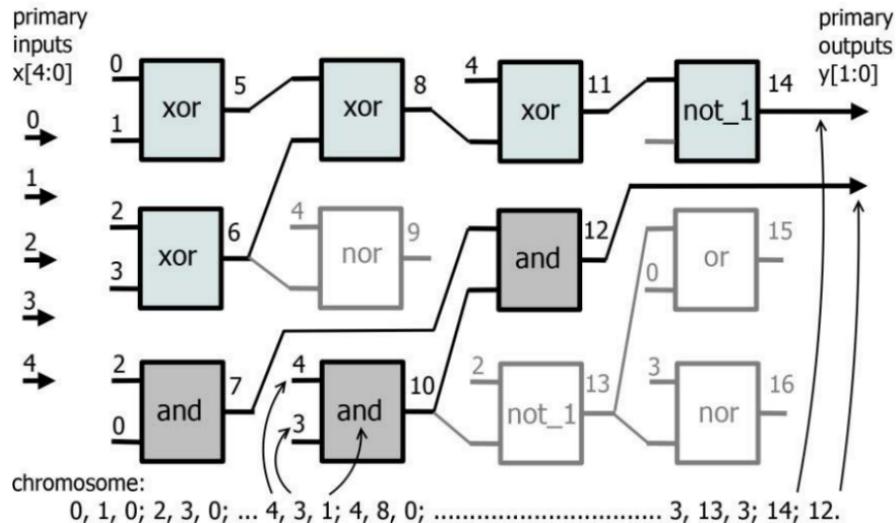


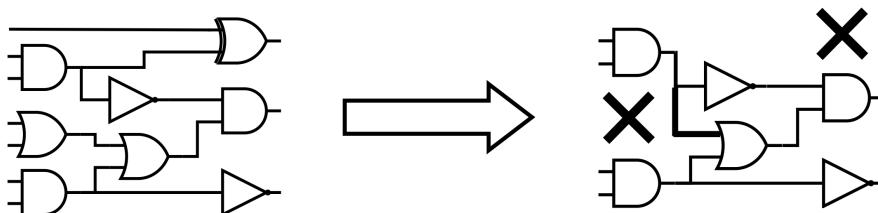
图 3-5 一个具有 5 比特输入、2 比特输出的组合逻辑门级网表及其对应的 CGP 表示

门级网表及其对应的 CGP 表示，可以看到有 4 个门没有被使用。容易想到，若将一个精确乘法器的电路结构转换成 CGP 形式，那么改变 CGP 中的整数序列便相当于改变了电路的功能，得到了近似乘法器。Mrazek 等人首先把 8 比特精确乘法器的不同电路实现结构转换成 CGP 形式，然后随机改变 CGP 中的一部分整数得到近似乘法器，挑选出精度、延迟和功耗都较好的电路作为新的基础电路，不

断迭代，在有限的时间内得到了均匀分布下误差较小、性能较高的 471 个近似乘法器，和 430 个近似加法器一起组成了一个近似算术单元库 EvoApprox8b^[107] 并开源了出来。考虑到许多应用中乘法器的输入数据分布并不是均匀的，在修改遗传算法中的目标函数后，CGP 也可以被用于生成特定分布下的高质量近似乘法器。^[108]。如果乘法器的位宽较大，CGP 等自动化方法生成的近似乘法器往往无法快速得到误差的边界，文献^[109]将近似等价性检查的形式化技术集成到基于 CGP 的搜索中去，能够将搜索推向可快速验证的近似电路，方法通过 ABC^[65] 工具实现，并对最大 32 比特位宽的乘法器进行了评估，在几小时内生成了一组高质量的 32 位近似乘法器。

3.2.4 近似电路综合

近似电路综合 (Approximate Circuit Synthesis)^[55] 是近似逻辑综合的一个细分方向，着重于近似算术单元的生成，属于近似电路设计方法中功能近似（见 1.1.4 有关功能近似的定义）的一种。给定一个精确电路的描述和约束（通常是误差），近似电路综合不需要知道具体的电路功能，能够自动生成满足精度的近似电路，可直接应用于各种算术单元，通用性更强。近似电路综合可以通过网表转换 (Netlist transformation) 或布尔重写 (Boolean rewriting) 的方式实现，图 3-6 举例展示了两种实现方式的区别，具体来讲，网表转换方式通过移除一些逻辑节点或用导线直接替换节点来对精确电路的门级网表进行简化，以达到减小面积和降低功耗的目的，而布尔重写方式则着重于修改抽象级别更高的函数真值表，使对应的布尔表达式更简洁紧凑。



(a) 网表转换方法实现近似电路，左：精确电路的门级网表，右：生成的近似电路的门级网表

i_n	\dots	i_0	o_1	o_0
0	0	...	0	0
0	0	...	1	1
...				...
1	1	...	0	1
1	1	...	1	0

i_n	\dots	i_0	o_1	o_0
0	0	...	0	0
0	0	...	1	0
...				...
1	1	...	0	1
1	1	...	1	1

(b) 布尔重写方法实现近似电路，左：精确电路的真值表，右：生成的近似电路的真值表

图 3-6 实现近似电路综合的两种常见方式

文献^[110]采用网表转换的方法，利用重替换(resubstitution)算法^[111]不断尝试修改电路的局部结构，每次修改后用仿真的方式确定误差，直到找到满足精度要求的电路后退出。该方法被 Meng 等人基于 ABC^[65]实现并命名为 ALSRAC，实验结果表明，ALSRAC 产生的近似电路的面积与国际前沿工作相比减小了 7%-18%。

3.3 研究动机

假设一个近似乘法器在输入为 x, y 时输出为 $f(x, y)$ ，由(2.40)得该输入下误差距离 ED 的平方为：

$$ED^2 = (xy - f(x, y))^2 \quad (3.11)$$

若将该近似乘法器应用于某一特定应用，且经统计该应用中乘法的两个输入分布为 p_1 和 p_2 ，由(2.44)和(3.11)得均方误差 MSE 为：

$$MSE = \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} (x_i y_j - f(x_i, y_j))^2 p(x'_i, y'_j) \quad (3.12)$$

$$p(x'_i, y'_j) = \begin{cases} p_1(x_i) \cdot p_2(y_j), & x'_i = x_i \text{ 且 } y'_j = y_j \\ p_1(y_j) \cdot p_2(x_i), & x'_i = y_j \text{ 且 } y'_j = x_i. \end{cases} \quad (3.13)$$

式中 $x_i \in \{x_0, x_1, \dots, x_{N-1}\}$, $y_j \in \{y_0, y_1, \dots, y_{M-1}\}$, 常数 N 和 M 分别是 x 和 y 的所有可能输入情况的总数，式(3.13)代表交换乘法器输入带来的影响。

3.3.1 输入分布对近似乘法器精度的影响

许多近似乘法器在设计时都有一个隐含的假设，即乘数和被乘数都是均匀分布的，然而很多应用并不满足该假设。例如，文献^[112]发现深度神经网络(Deep Neural Network, DNN)中权重的分布并不是均匀的，而是集中在某个值附近。为了证明不同数据分布对近似乘法器精度的影响，将文献^[100]提出的面向浮点数乘法的设计方法拓展到定点数领域，针对均匀分布和从 DNN 应用中提取的真实分布分别生成了 8 比特无符号整数近似乘法器 $f^{(1)}$ 和 $f^{(2)}$ 并进行误差比较。过程如下：

对于均匀分布，易得 $f^{(1)} = -16384 + 128x + 128y$ 。对于 DNN 应用，为了获得操作数的概率分布，对采用 8 比特位宽无符号整数量化(Quantization)的 LeNet 网络在 MNIST 数据集的 10000 张图片上进行训练^[113]，统计第一个全连接(First Fully-Connected, FC1)层的输入和权重的数据分布，如图3-7所示，可以看到输入值很多是 0 而权重值集中在 128 附近。将输入和权重分别用半高斯和高斯分布进行拟合，并利用文献^[100]的方法使式(3.12)最小，可以得到 $f^{(2)} = -1549 + 129x + 12y$ 。注意 $f^{(2)}$ 是非对称的， x 是特征输入， y 是权重。

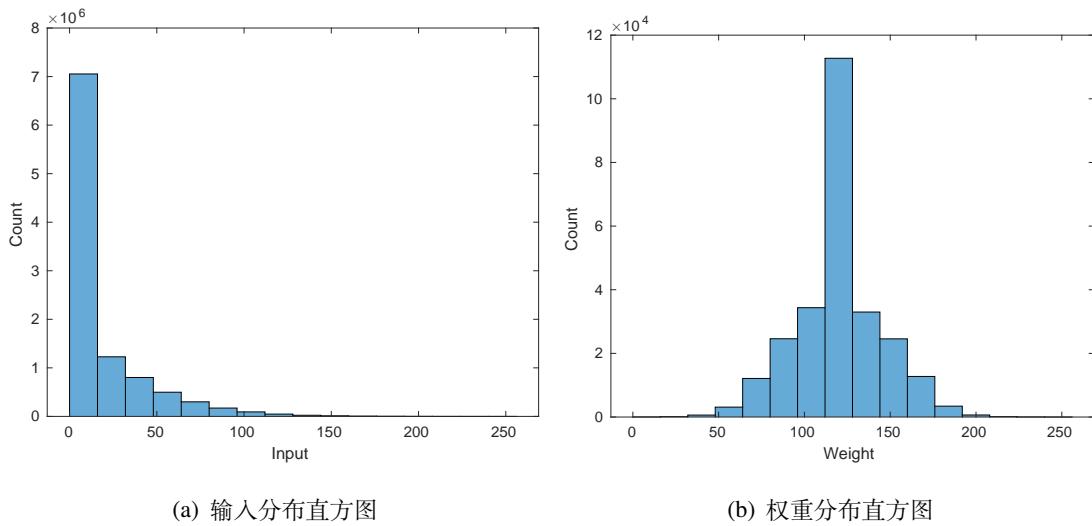


图 3-7 采用 8 比特位宽量化的 LeNet 网络在 MNIST 数据集上训练后 FC1 层的输入和权重的数据分布直方图

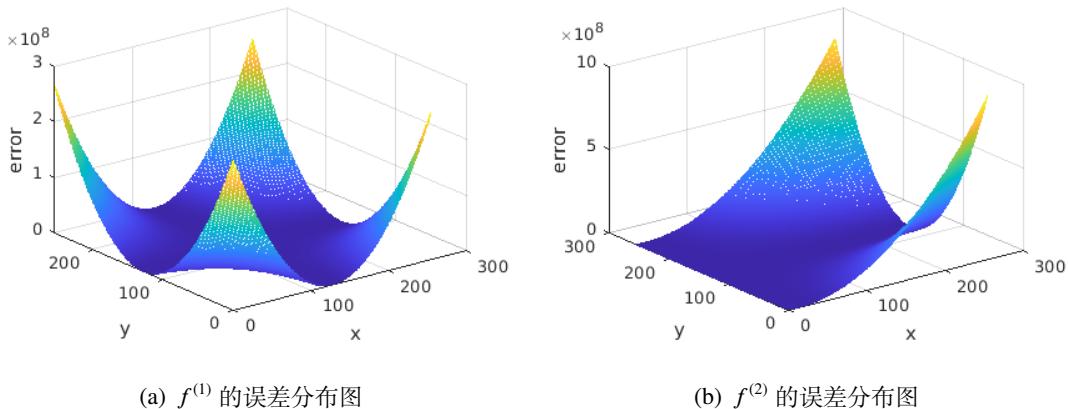


图 3-8 近似乘法器 $f^{(1)}$ 和 $f^{(2)}$ 的误差分布图, 这里的误差是指误差距离 ED 的平方

近似乘法器 $f^{(1)}$ 和 $f^{(2)}$ 的误差分布如图3-8所示，这里的误差是由式(3.11)计算得到的，可以看到 $f^{(2)}$ 在 $x = 0$ 、 $y = 128$ 附近的误差比 $f^{(1)}$ 小。把 FC1 中的精确乘法分别全部替换为 $f^{(1)}$ 和 $f^{(2)}$ ，重新对 LeNet 进行训练，并把每一次乘法产生的误差相加，得到 $f^{(1)}$ 的总误差为 3.12×10^{16} ， $f^{(2)}$ 的总误差为 4.77×10^{14} ，比 $f^{(1)}$ 小两个数量级，这充分说明了在设计近似乘法器时考虑真实的数据分布能够大大提高乘法器的精度。

3.3.2 输入极性对近似乘法器精度的影响

近似乘法器通常是不对称的，这意味着交换输入前后乘法器的输出结果不一致。为了展示输入极性对近似乘法器精度的影响，对基于 CGP 方法开发的包

含 500 个帕累托最优 (Pareto Optimality) 的近似乘法器库 Evoapprox8b^[107] 进行了研究，过程如下：

假设近似乘法器的输入分别是 x 和 y ，对于 DNN 或滤波器 (Filter) 应用，定义 $P = 0$ 代表 x 是输入、 y 是权重， $P = 1$ 代表 x 是权重、 y 是输入。基于 LeNet 网络和 MNIST 数据集^[113]，对 Evoapprox8b^[107] 中全部的 500 个近似乘法器进行 $P = 0$ 和 $P = 1$ 的精度评估，结果如图3-9所示。在图3-9中，每个点代表一个近

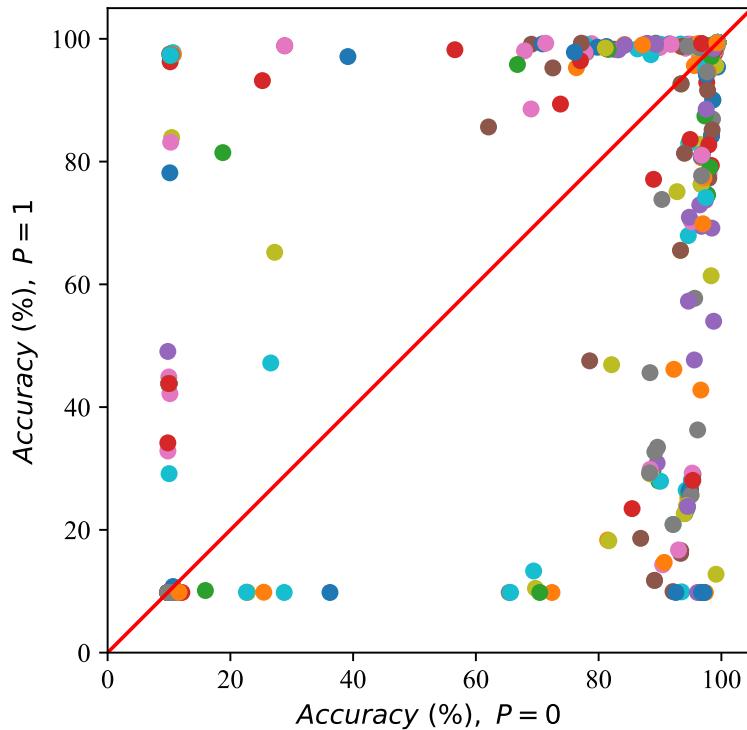


图 3-9 基于 LeNet 和 MNIST 得到的 Evoapprox8b 中全部 500 个乘法器在 $P = 0$ 和 $P = 1$ 的情况下的精度散点图

似乘法器，横轴表示 $P = 0$ 时的精度，纵轴表示 $P = 1$ 时的精度，落在红线上的点代表该乘法器在交换输入后 LeNet 的精度保持不变。可以看到，落在红线上的点很少，这意味着大多数近似乘法器在交换输入后会对精度产生影响。同时，点离红线的距离代表了乘法器不对称的程度，离得越远，乘法器在该 DNN 应用下的不对称程度越高，交换输入后对精度的影响越大。易知图3-9中点离红线的距离 D 为：

$$D = 100 \times |A_{P_0} - A_{P_1}| \quad (3.14)$$

式中 A_{P_0} 和 A_{P_1} 分别表示该点在 $P = 0$ 和 $P = 1$ 时的精度， $|A_{P_0} - A_{P_1}|$ 代表两个精度的差值的绝对值。图3-10展示了图3-9中全部的 500 个点与红线的距离直方图，统计结果表明超过 250 个点的 D 大于 2.8，这意味着 Evoapprox8b^[107] 中至少一半的乘法器在交换输入后 $|A_{P_0} - A_{P_1}|$ 大于 2.8%，值得注意的是所有乘法

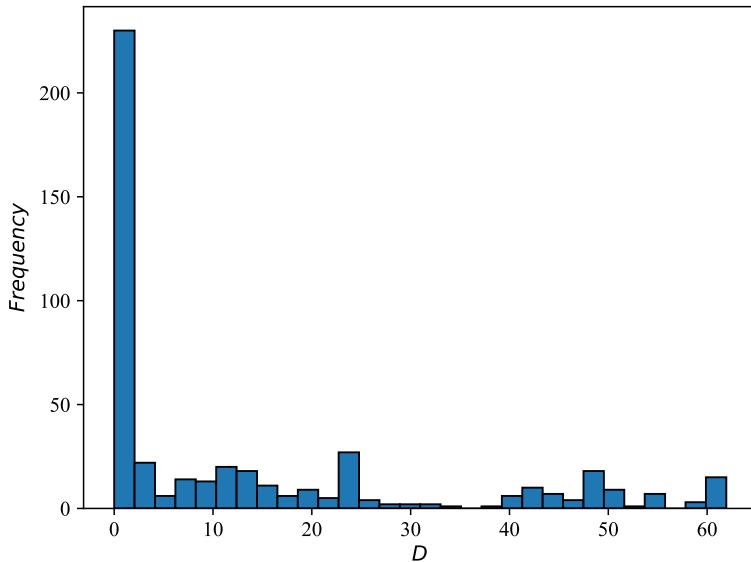


图 3-10 Evoapprox8b 中全部 500 个乘法器不对称程度统计直方图

器的精度评估均基于网络规模较小的 LeNet，当网络规模上升后，由于误差的累计，近似乘法器的不对称性导致的精度下降会更加严重，这充分说明了在设计近似乘法器时考虑输入极性的必要性。

3.4 研究内容与创新点

本文提出了一种新的自动化近似乘法器生成方法，与文献^[98]类似，该方法通过逻辑操作和移位操作在部分积生成后、累加前对部分积进行一次压缩，降低部分积阵列的规模，减轻后续的累加压力。同时，与文献^[98]只采用或操作不同，所提出的方法同时利用与、或、异或和移位操作对部分积进行压缩，降低乘法器的误差。最后，该方法将设计近似乘法器的问题建模成优化问题，利用计算机自动寻找在特定输入分布下的最优压缩操作，同时考虑输入极性。主要创新点如下：

- 为了方便测试不同近似乘法器在不同规模神经网络上的性能，本文提出并开源了一个基于 8 比特无符号数量化的 DNN 推断（Inference）精度评估工具，该工具通过查找表表示一个近似乘法器，支持 LeNet^[113]、AlexNet^[114]、VGG16^[115]三个不同规模的神经网络和 MNIST^[113]、CIFAR-10^[116]两个不同大小的数据集。
- 提出的设计方法可以生成任意分布下的无符号乘法器，针对 8 比特位宽的三个不同规模的神经网络的实验结果表明，与国际前沿工作相比，生成的近似乘法器在几乎没有任何精度损失的情况下，功耗延迟面积积（Product

of Power, Delay, and Area, 缩写作 PDA) 提升了 26.4%。同时，针对大规模神经网络设计的近似乘法器在面对小规模神经网络时表现出了一定的可迁移性。

- 利用改进的 Baugh-Wooley 算法^[62-64] (见2.1.1)，方法能够生成补码有符号乘法器，基于采用 16 比特位宽的有限冲击响应 (Finite Impulse Response, FIR) 滤波器的实验结果表明，与国际前沿工作相比，生成的近似乘法器在几乎没有任何精度损失的情况下，PDA 优化了 27.1%。

3.5 研究方法

受相关工作^[98,100,103]的启发，对乘法器的部分积在生成后、累加前同时通过多种操作对其进行压缩，降低部分积阵列的规模，减轻后续的累加压力，并把寻找最优压缩操作的问题建模成数学问题，利用计算机进行求解。

3.5.1 无符号乘法器

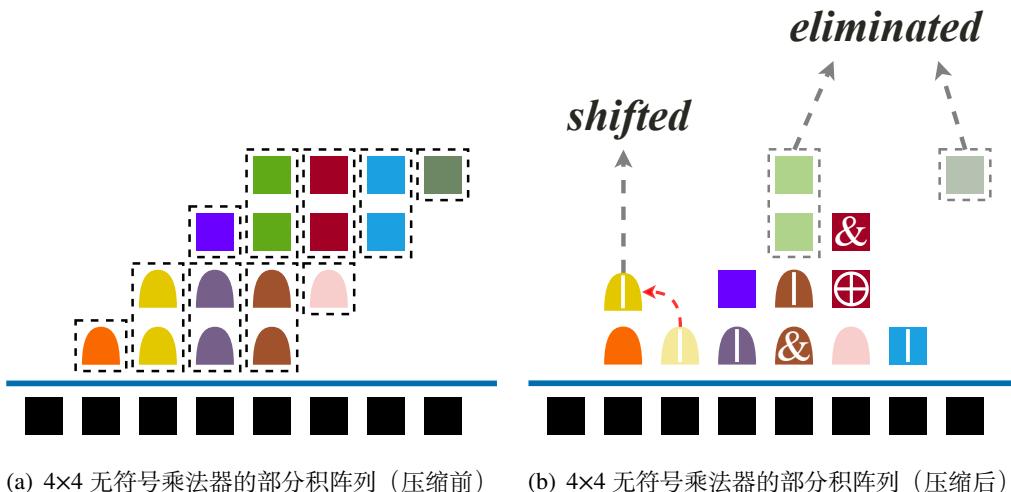


图 3-11 一个利用 AND、OR、XOR、shift 操作对 4×4 无符号乘法器部分积进行压缩的例子

图3-11展示了一个利用与、或、异或和移位共 4 种操作对 4×4 无符号乘法器部分积进行压缩的示例。在图3-11(a)中，蓝线上方不同颜色的正方形或半椭圆形代表部分积比特，组成部分积阵列，共由 4^2 个与门生成。首先，选择全部四行部分积，每两行一组对其权重相同的部分积比特进行分簇，共分为 10 个簇，每个簇由虚线矩形表示，包含 1 个或 2 个部分积比特；之后，利用与、或、异或和移位操作对簇内的部分积进行运算，生成运算后的部分积比特，可能的运算有 6 种：单独地与、或、异或三种逻辑操作，以及分别三种逻辑操作后左移一位。运算过程中有以下几点需要注意：(1) 簇可以直接消失；(2) 对只包含一个部分积比

特的簇（以下简称为单比特簇），不进行运算，只存在保留或消失两种情况；(3)对包含两个比特的簇（以下简称为双比特簇），若没消失，则该簇不同运算产生的部分积比特可能会同时存在。把双比特簇运算产生的部分积和单比特簇保留产生的部分积统称为压缩项（Compressed term）。图3-11(b)展示了对图3-11(a)中的部分积阵列进行压缩的一种可能结果，蓝线以上的所有形状代表压缩项，组成了新的部分积阵列。对于内有符号的压缩项，其符号代表了对图3-11(a)中与压缩项同颜色的双比特簇执行的逻辑操作，注意逻辑操作后可能会伴随移位，同时可以看到有两个簇消失了。

假设精确乘法器的部分积阵列有 g 行，每行有 h 个比特，选择 l 行部分积进行分簇，则未分簇的部分积比特的总数 S 为：

$$S = (g - l)h, \quad l \in \{0, 2, 4, \dots, g'\}, \quad (3.15)$$

$$g' = \begin{cases} g, & g \text{ 是偶数.} \\ g - 1, & g \text{ 是奇数.} \end{cases} \quad (3.16)$$

例如，对图3-11(a)有 $g = h = l = 4$, $S = 0$ 。

3.5.2 有符号乘法器

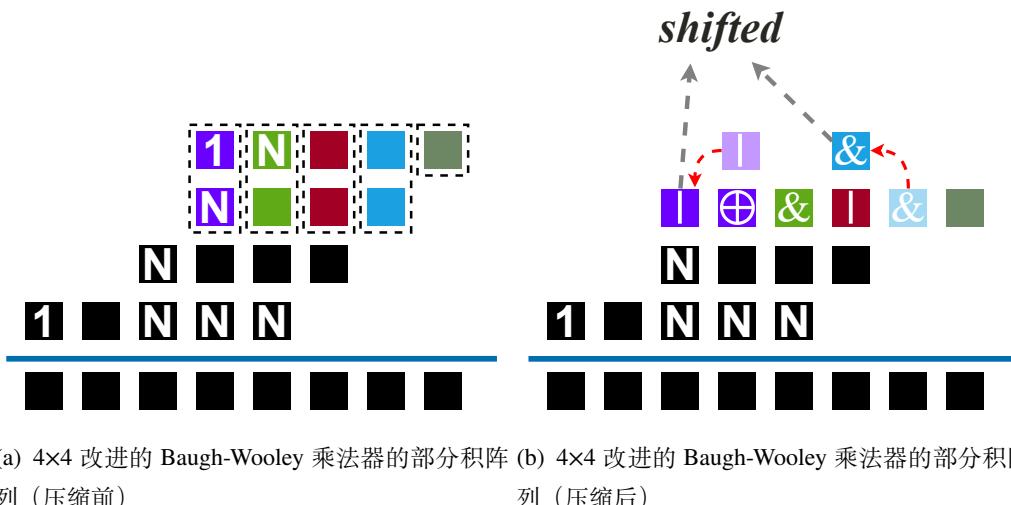


图 3-12 一个利用 AND、OR、XOR、shift 操作对 4×4 改进的 Baugh-Wooley 乘法器的部分积进行压缩的例子

两个部分积比特成双比特簇的前提是两者具有相同的权重值，补码有符号数直接相乘产生的部分积比特的权重值有正有负，无法直接进行分簇，改进的 Baugh-Wooley 算法^[62-64]（见2.1.1）能够将所有部分积比特的权重变为正值，顺

利实现分簇压缩。有符号乘法器部分积的压缩过程与无符号乘法器类似，图3-12展示了一个利用提出的4种操作对改进的Baugh-Wooley乘法器的部分积进行压缩的示例，与3-11相比有以下不同：

- 改进的Baugh-Wooley乘法器的部分积比特并不全部由与门生成，有些部分积比特由与非(NAND)门产生，如图3-12(a)所示，蓝线上方每个正方形代表一个部分积比特，内有“N”符号的正方形表示该比特由与非门得到。同时，对第一行部分积和最后一行部分积分别额外添加了一个常数“1”来保证结果的正确性，由内有“1”符号的正方形表示；
- 与图3-11(a)选择全部4行部分积进行分簇压缩不同，图3-12(a)只选择了前两行部分积进行压缩，后两行部分积保持不变。与选择全部的部分积进行压缩相比，这能够提高生成的近似乘法器的精度，即可以通过调整分簇的部分积的行数来生成具有不同质量的近似乘法器，注意最后一行部分积的常数“1”永远不参与压缩。
- 图3-11(b)表示全部的压缩项组成了新的部分积阵列，而图3-12(b)表示压缩项和原先未分簇的部分积一起构成了新的部分积阵列。

在不考虑额外添加的两个常数“1”的情况下，同样假设改进的Baugh-Wooley乘法器有 g 行部分积，每行有 h 个部分积比特，选择 l 行部分积进行分簇压缩，式(3.15)变为：

$$S = \begin{cases} gh + 2, & l = 0, \\ (g - l)h + 1, & l \in \{2, 4, \dots, g'\}, \end{cases} \quad (3.17)$$

式中 g' 如式(3.16)所示。对于图3-12(a)， $h = g = g' = 4$ ， $l = 2$ ， $S = 9$ 。

3.5.3 自动化求解

当输入为 x_i 、 y_j 时，本文提出的基于部分积压缩的近似乘法器的输出可以被公式化地表述为：

$$f(x_i, y_j | \theta) = \sum_{u=0}^{S-1} b_u + \sum_{k=0}^{Z-1} \theta_k L_k \quad (3.18)$$

式中 b_u 表示 S 个不分簇的部分积比特中的一个； $\theta_k \in \{0, 1\}$ ，代表是否存在一个压缩项； L_k 代表对一个双比特簇执行一种运算或对一个单比特簇进行保留； Z 是待求解的变量的数目，也代表压缩项总数的上限，例如，在无符号乘法器中，每两行部分积包含2个单比特簇和 $h-1$ 个双比特簇（如3-11(a)所示），因此，当选择 l 行部分积进行压缩时，无符号乘法器的 Z 为：

$$Z = [2 \times 1 + (h - 1) \times 6] \times \frac{l}{2} = (3h - 2)l \quad (3.19)$$

对于基于 Baugh-Wooley 算法的有符号乘法器，其部分积的第一行中有一个额外的常数“1”（如图3-12(a)所示），因此前两行部分积包含 1 个单比特簇和 h 个双比特簇， Z 变为：

$$Z = \begin{cases} 0, & l = 0, \\ (3h - 2)l + 5, & l \in \{2, 4, \dots, g'\}. \end{cases} \quad (3.20)$$

式中 g' 如式(3.16)所示，注意最后一行部分积的常数“1”永远不参与分簇。

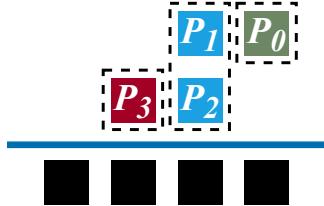


图 3-13 2×2 无符号乘法器部分积阵列及 $S = 0$ 时的分簇情况

图3-13展示了 2×2 无符号乘法器部分积阵列及 $S = 0$ 时的分簇情况，假设输入是 x_i 和 y_j ，式(3.18)变成了：

$$\begin{aligned} f(x_i, y_j | \theta) = & 0 + \theta_0 P_0 \cdot 2^0 \\ & + \theta_1(P_1 \& P_2) \cdot 2^1 + \theta_2(P_1 \& P_2) \cdot 2^2 \\ & + \theta_3(P_1 | P_2) \cdot 2^1 + \theta_4(P_1 | P_2) \cdot 2^2 \\ & + \theta_5(P_1 \oplus P_2) \cdot 2^1 + \theta_6(P_1 \oplus P_2) \cdot 2^2 \\ & + \theta_7 P_3 \cdot 2^2 \end{aligned} \quad (3.21)$$

对于式(3.18)，任意的 Z ，均存在 $\theta = \theta^e$ 使 $f(x_i, y_j | \theta^e) = x_i \times y_j$ 对任意的 $x_i \times y_j$ 成立，即对任意位宽的乘法器，不论选择几行部分积进行分簇压缩，精确乘法器都是所有可能压缩情况中的一个解。例如，假设 $\theta = [1, 0, 1, 0, 0, 1, 0, 1]$ ，则(3.21)代表一个精确的 2×2 无符号乘法器，相当于对图3-13中的部分积保留 P_0 和 P_3 的同时使用一个半加器对 P_1 和 P_2 进行累加。这表明所有可能的压缩情况构成了一个高质量的解空间，存在许多低误差的近似乘法器。

将式(3.18)与式(3.12)结合，以均方误差 MSE 为优化目标的求解问题可以被公式化地表示为：

$$\min_{\theta} \left\{ \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} \left[(x_i y_j - (\sum_{u=0}^{S-1} b_u + \sum_{k=0}^{Z-1} \theta_k L_k))^2 p(x'_i, y'_j) \right] \right\} \quad (3.22)$$

式(3.22)只考虑了误差，对近似乘法器来讲硬件成本也同样重要。从电路实现角度来看，部分积阵列的规模与乘法器的性能强相关，为了降低压缩后新部分积

阵列规模的大小，引入一个惩罚项 $Cont(\theta)$ 作为优化目标的一部分，定义为：

$$Cont(\theta) = \lambda T \quad (3.23)$$

$$T = \sum_{k=0}^{Z-1} \theta_k \quad (3.24)$$

其中 T 表示压缩项的总数， λ 是一个用于控制惩罚程度的常量， λ 越大，压缩项总数越少，新的部分积阵列规模越小，电路实现越简单，误差越大，反之则相反。加入惩罚项后式(3.22)变成了：

$$\min_{\theta} \left\{ \sum_{i=0}^{N-1} \sum_{j=0}^{M-1} [(x_i y_j - (\sum_{u=0}^{S-1} b_u + \sum_{k=0}^{Z-1} \theta_k L_k))^2 p(x'_i, y'_j)] + Cont(\theta) \right\} \quad (3.25)$$

即为最终的优化目标，式中 $p(x'_i, y'_j)$ 代表乘法器的极性，由(3.13)给出。可采用混合整数遗传算法（Mixed Integer Genetic Algorithm, MIGA）通过 MATLAB 或 Python 对(3.25)进行求解。

3.5.4 求解过程

求解过程包括以下三个步骤：(a) 基于用户想要的面积减少比例 R （与同位宽精确乘法器相比）来确定 l 的值，并参考(3.22)基于用户给定的输入分布生成具有不同输入极性的面向均方误差 MSE 的目标函数；(b) 给定 R ，找到一个合适的 λ 值，参考(3.25)生成最终的两个输入极性相反的优化目标；(c) 通过 MIGA 求解目标函数并生成近似乘法器。

根据用户想要的面积减少百分比 R 确定 l

在步骤 (a) 中，为了对任意的 R 确定一个合适的 l 的取值，对式(3.25)测试了均匀分布下 8 比特无符号乘法器在不同 l 和 λ 下得到的不同解对应的压缩项的总数 T （见式(3.24)）、乘法器的功耗延迟面积积 PDA 以及平均绝对误差 MAE（即平均误差距离 MED，见式2.42），结果如图3-14所示。可以看到，当

$$T \approx \frac{l-1}{2} h \quad (3.26)$$

时，对应的乘法器能在精度和硬件之间能取得一个较好的权衡，例如在图3-14(d)中， $\frac{l-1}{2} h = 28$ ， $T = 29$ 或30。假设乘法器的面积和部分积阵列的规模成正比，且对任意位宽的乘法器压缩后压缩项的总数 T 都应在 $\frac{l-1}{2} h$ 附近，那么有：

$$lh - ghR = \frac{l-1}{2} h \quad (3.27)$$

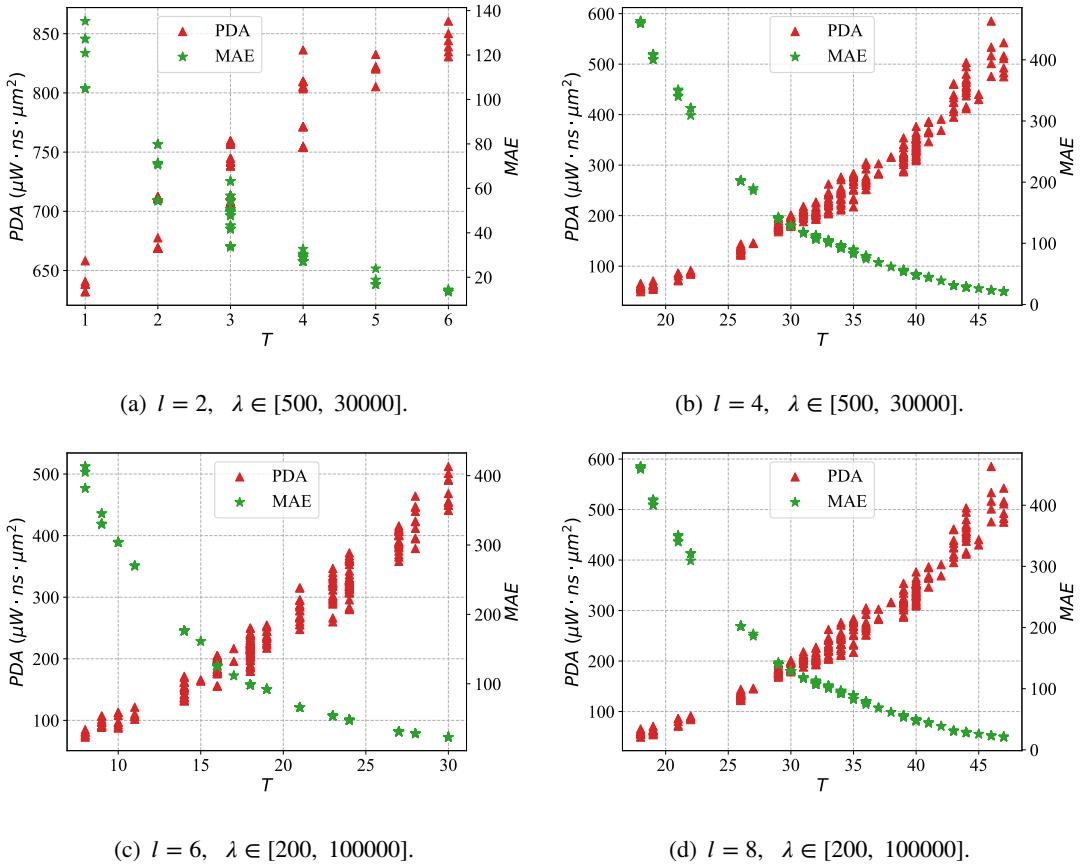


图 3-14 均匀分布下 8 比特无符号乘法器在不同 l 和 λ 下得到的不同解对应的压力项的总数 T 、乘法器的功耗延迟面积积 PDA 以及平均绝对误差 MAE

可得：

$$l \approx \min\{g', \text{even}(2gR - 1)\} \quad (3.28)$$

式中 $\text{even}()$ 表示向上取最近的偶数。换句话说， l 为 $\min\{g', \text{even}(2gR - 1)\}$ 时与合适的 λ 值一起能够在满足 R 的前提下产生高质量的解空间，实验结果表明式(3.28)能够指导生成高质量的近似乘法器。

式(3.12)需要遍历乘法器所有可能的输入情况，这对小位宽乘法器来讲是可行的，测试结果在 Intel Xeon Platinum 8354H 处理器上对 8 比特位宽乘法器利用单线程遍历 (2^{16} 种输入情况) 仅需 15 秒，对 16 比特位宽乘法器利用 128 线程遍历 (2^{32} 种输入情况) 需 8 小时。注意遍历时间随着乘法器位宽的增加指数增长，当位宽大于 16 比特时，遍历变得不可接受。有两种办法解决该问题：(1) 如果实际应用中乘法器的输入只会取某些特定的值，并且遍历所有可能输入值的时间是可接受的，那么只需针对这些可能值进行计算；(2) 当无法遍历所有可能输入值时，通过随机抽样的方式仅考虑一部分可能输入值进行计算。注意不论哪种情况都应基于真实的数据分布进行求解。

根据用户想要的面积减少百分比 R 确定 λ

Algorithm 2: 给定 R 找到一个合适的 λ 值

Input: R : 用户想要的面积减少比例 R (与同位宽精确乘法器相比)。

Output: λ_R : 一个满足 R 并且能够生成高精度近似乘法器的 λ 的取值。

```

1 MSE: 均方误差, 由式(3.12)、式(3.18)、式(3.28)和  $R$  联合确定;
2  $Z$ : 由式(3.19)或式(3.20)联合式(3.28)、 $R$  确定;
3  $\lambda_R = 1$ ;  $T_{\lambda_R} = Z$ ; // 初始化
4 while  $T_{\lambda_R} > \max\{lh - gh \times R, 0\}$  do
5    $\theta^{\lambda_R} = \text{MIGA}(MSE + \lambda_R \sum_{k=0}^{Z-1} \theta_k)$ ; // 利用 MIGA 进行求解
6    $T_{\lambda_R} = \sum_{j=0}^{Z-1} \theta_j^{\lambda_R}$ ; // 压缩项总数
7    $\lambda_R = \lambda_R * 10$ 
8  $\lambda_R = \lambda_R / 20$ ;
9 return  $\lambda_R$ ;

```

给定 R , λ 的值可由算法2确定。在算法2中, 对 λ 的取值从 1 开始尝试, 若不满足条件直接将 λ 增大十倍, 直到得到的解的压缩项总数 T 满足

$$T \leq lh - gh \times R \quad (3.29)$$

即认为 λ 的值和式(3.28)的 l 一起满足 R 和(3.26), 能够生成高质量的近似乘法器。

需要注意的是, R 是一种软性约束, 即基于 R 按照式(3.28)和算法2得到的 l 和 λ 的值是指导性的, 可尝试后根据实际情况进行灵活调整。

误差分析

l 与 λ 的取值和近似乘法器的精度高度相关, 原因如下:

- λ 的大小影响生成的压缩项的总数, 精确乘法对应的压缩项数目是一个适中的值, 因此 λ 较小时, 目标函数由 MSE 主导, 有可能生成低误差的乘法器, 而 λ 过大时, 一定会降低乘法器的精度。
- 对于给定的 g 和 h , 不分簇的部分积比特总数 S 与 l 成反比关系 (见式(3.15)和式(3.17)), 小的 l 的取值导致大的 S , 能够在 λ 较大时也能产生低误差的近似乘法器, 但坏处是硬件提升上限低;

- 对式(3.25)的求解是一个 NP 难问题，尽管混合整数遗传算法 MIGA 提供了一个高效的解决方案，MIGA 的随机性则导致变量过多时求解效率的低下。在式(3.18)中，待求解的变量的数目 Z 的取值和 hl 成正比（见式(3.19)和式(3.20)），对于一个给定的 h ，大的 l 导致求解变量过多，MIGA 算法无法高效对问题进行求解。

3.5.5 基于 8 比特无符号数量化的 DNN 推断精度评估工具

本文提出并开源了一个面向近似乘法器的基于 8 比特无符号数量化的 DNN 推断精度评估工具，其中近似乘法器以查找表的形式表示。在该工具中，一个 DNN 通过有向无环图 DAG 进行表示，其中点代表 DNN 中的连接层，边代表数据流，当 DAG 中的一个节点被执行时，它的依赖项将被自动执行。图3-15展示了 LeNet 网络^[113]在评估工具中的 DAG 表示，一张图从 *Image* 节点输入，分类结果从 *FC2* 节点输出。

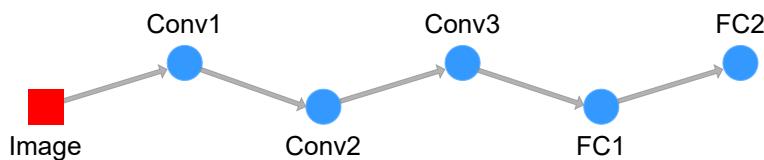


图 3-15 LeNet 在评估工具中的 DAG 表示

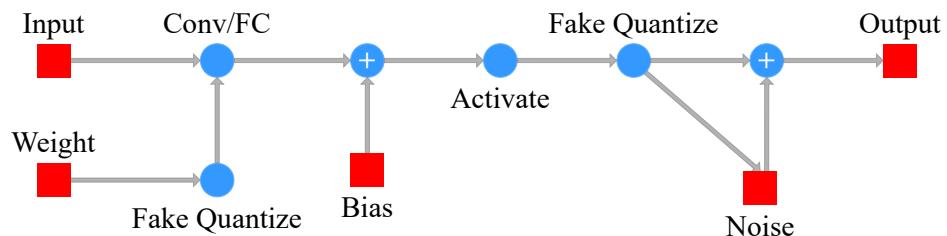


图 3-16 采用伪量化方法的基于噪声训练的 DNN 计算流图

为了减少由于量化造成的精度损失，对 DNN 采用了伪量化方法^[117]，如图3-16所示。伪量化操作应用于权重和激活函数（Activation function），由量化级别 t 和钳位范围 $[a, b]$ 两个参数组成，通过逐点应用量化函数 q 来进行， q 被定义为：

$$\text{clamp}(r, a, b) = \min(\max(r, a), b) \quad (3.30)$$

$$s(a, b, t) = \frac{b - a}{t - 1} \quad (3.31)$$

$$q(r, a, b, t) = a + \lfloor \frac{\text{clamp}(r, a, b) - a}{s(a, b, t)} \rfloor \quad (3.32)$$

其中 r 表示要量化的实数， $\lfloor \cdot \rfloor$ 表示四舍五入到最接近的整数。在 8 比特伪量化中， $n = 2^8 = 256$ ， $a = \min(r)$ ， $b = \max(r)$ 。对于批归一化（Batch Normalization, BN）的 DNN，在伪量化过程开始时将每个 BN 层与其前一层合并，以保持高精度的同时简化量化后的计算。合并操作可以通过以下等式来描述：

$$\hat{w} = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} w \quad (3.33)$$

其中 w 是原始权重张量， \hat{w} 是合并后的权重张量， γ 表示 BN 层的尺度参数， σ^2 表示激活方差， ϵ 是一个用于保持数值稳定性的常数。

另外，在伪量化^[117]过程中，采用了噪声训练（Noise training）技术来减少由于近似乘法器的引入造成的精度损失。图3-16展示了带有噪声训练技术的 DNN 计算流图，其中输出是激活加上噪声，用于模拟近似乘法引起的计算误差，噪声 $n(y)$ 根据激活 y 生成， $n(y)$ 被定义为：

$$n(y) = \alpha \times \text{rand}(-1.0, 1.0) \times |y| \quad (3.34)$$

其中 α 表示噪声幅度， $\text{rand}(-1.0, 1.0)$ 表示在 -1.0 到 1.0 之间随机取值。图 3-17 显示了基于不同噪声幅值 ($\alpha \in \{0, 0.2, 0.4, 0.6, 0.8\}$) 训练并近似后的 AlexNet 网络^[114]在 CIFAR-10^[116] 推理数据集上的精度。可以看到噪声幅度 α 为 0.8 时 AlexNet 实现了最高的精度，这表明了噪声训练技术的有效性。在本文中，如果不特殊说明，所有 DNN 的噪声幅度 α 取值均为 0.8。

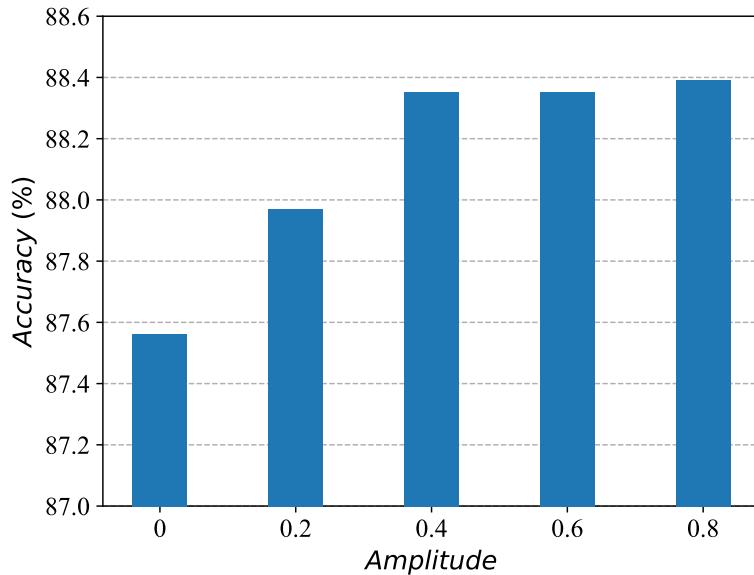


图 3-17 不同噪声幅值训练并近似后的 AlexNet 神经网络在 CIFAR-10 推理数据集上的精度

3.6 实验结果

为了详细评估所提出的方法的有效性，对不同应用进行了实验，并与国际前沿工作进行比较，具体步骤如下：首先确定乘法器的位宽，并提取输入数据分布，基于给定的面积减少比例 R 通过式(3.28)和算法2得到合适的 l 和 λ 的取值，然后根据式(3.25)生成两个输入极性相反的优化目标函数，注意均匀分布下不需要考虑极性；之后利用 MATLAB 混合整数遗传算法 MIGA 在一台拥有 72 核 256GB 内存的 Intel Xeon 服务器上分别对两个目标函数运行 48 小时进行求解，生成近似乘法器，挑出目标值最小的一批乘法器并与已有的工作进行比较。比较的乘法器包括 KMap^[96]、SDLC^[98]、CR^[97]、AC^[118]、OU^[100]、RoBA^[119]、DRUM^[99]、TOSAM^[91]、PPAM^[120]。其中，SDLC 采用 2 比特或门的压缩策略，以实现最高的精度；CR 的误差模块通过 6 比特和 7 比特两种位宽进行实现，分别命名为 C.6 和 C.7；OU 原本是面向浮点数乘法器设计的，将其修改为定点数乘法器并使用 1 次划分和 3 次划分方式进行误差补偿，分别命名为 L.1 和 L.3；DRUM 的参数 m 分别取 4,5,6,7 进行实现；TOSAM 中的可配置参数 h 和 t 取 $\{0, 1, 2\} \times \{1, 2, 3\}$ ；PPAM 中的可配置参数 i 和 j 取 $\{0, 1, 2\} \times \{1, 2, 3\}$ 。同时，基于 CGP 方法生成的两个近似算术单元库 Evoapprox8b (Evo8)^[107] 和 EvoApproveLib^{LITE} (EvoLite)^[121] 也参与比较。

所有乘法器的精度直接由应用的精度表示，注意对于没有标明输入的非对称乘法器在交换输入后也有一个精度，为保证公平，取最高的精度参与比较；硬件指标包括面积、延迟、功耗，均在一定的时钟频率约束下由 Synopsys Design Compiler (DC) S-2021.06-SP5 基于一个开源的 7nm 工艺库^[122]综合得到。对精确乘法器来讲，DC 会调用 DesignWare 库^[123]自动进行实现，实现的精确乘法器简称为 DesignW。

3.6.1 均匀分布下的 8 比特无符号乘法器

假设某一应用中乘法器的输入为 8 比特无符号数，且数据是均匀分布的（无需考虑输入极性），应用精度是乘法器的平均绝对误差 MAE（即平均误差距离 MED，见式2.42）。

为方便比较，忽略 R ，直接对 l 取 2,4,6,8 并对 λ 进行不同取值的尝试，生成的乘法器的 MAE 和 PDA 散点图如图3-18所示，其中 PDA 是在 2GHz 的时钟频率约束下得到的，DesignW 代表由 DesignWare 库^[123]实现的精确乘法器，可以看到 l 取值决定了近似乘法器 PDA 的提升上限 ($l = 6, 8$ 时不太明显)，但代价是误差较大。

图3-19展示了生成的近似乘法器与国际前沿工作进行 PDA 和 MAE 对比的散点图，时钟频率约束为 2GHz。在该图中，越靠近左下角的乘法器质量越好，

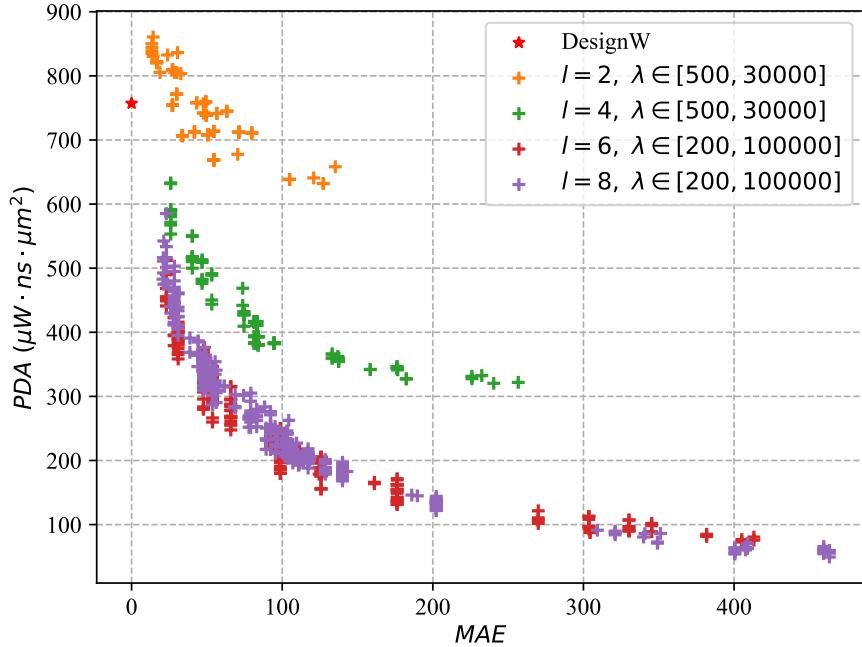
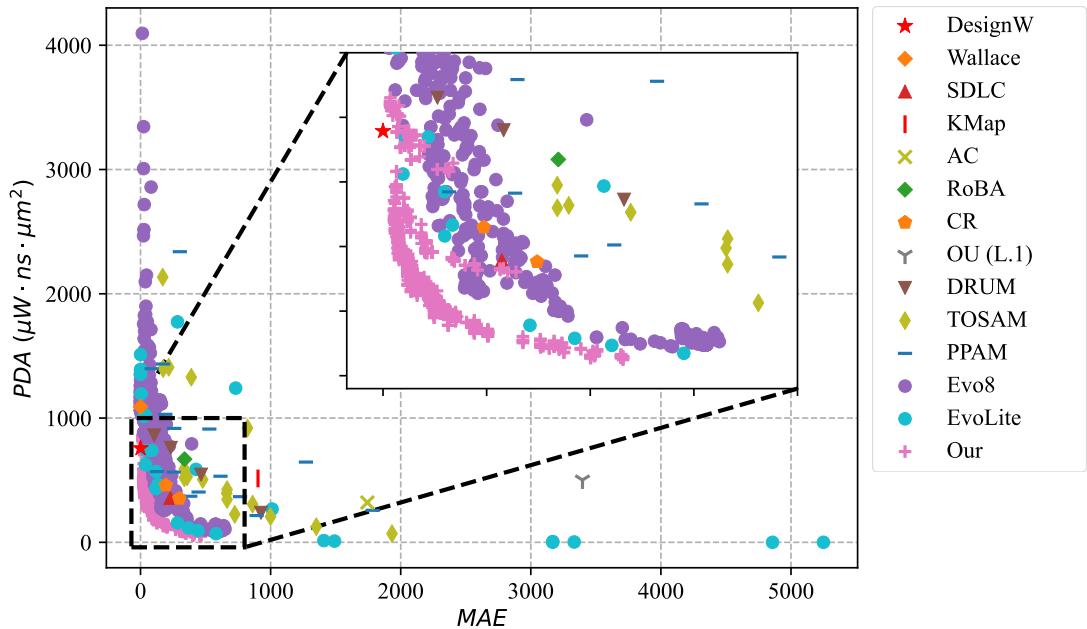
图 3-18 不同 l 和 λ 取值下生成的近似乘法器的 MAE 和 PDA 散点图比较

图 3-19 生成的近似乘法器与国际前沿工作进行 MAE 和 PDA 比较

可以看到本文提出的基于多种操作对部分积进行压缩的自动化方法取得了显著的优势，在 MAE 和 PDA 两方面都领先于绝大部分已有的近似乘法器，同时需要注意 DesignW 的 PDA 比许多近似乘法器要好。

3.6.2 基于 8 比特无符号数的不同规模的 DNN 应用

从 DNN 中提取乘法器的输入分布，并在某个特定的 R 下按照如前所述的方法生成近似乘法器，之后与已有的近似乘法器一起在考虑极性的情况下利用提出的 DNN 推断精度评估工具对近似后的神经网络进行精度评估，并利用 DC 进行综合以比较硬件性能。DNN 中乘法器的两个输入分别是特征输入（Feature input）和权重（Weight），两者的概率分布通常不同，因此按照本文提出的方法对 DNN 生成近似乘法器时需注意极性，设 XFYW 代表 $P = 0$ 的乘法器（ x 是特征输入、 y 是权重），XWYF 代表 $P = 1$ 的乘法器（ x 是权重、 y 是特征输入）。

为了对不同乘法器的质量进行统一比较，引入功耗、延迟、面积、相对精度损失 4 个指标的乘积（Product of relative accuracy loss and PDA，APDA）来衡量不同乘法器在某个 DNN 应用中的好坏，相对精度损失定义为：

$$\text{ceil\%}(A_{exact}) - A_{mul} \quad (3.35)$$

式中 A_{exact} 表示 DNN 在精确乘法器下的精度，ceil%() 表示百分比格式的向上取整，如 $\text{ceil\%}(88.39\%) = 89\%$ ， A_{mul} 表示 DNN 在测试乘法器下的精度，注意精确乘法器的相对精度损失可能不为 0。

LeNet 和 MNIST

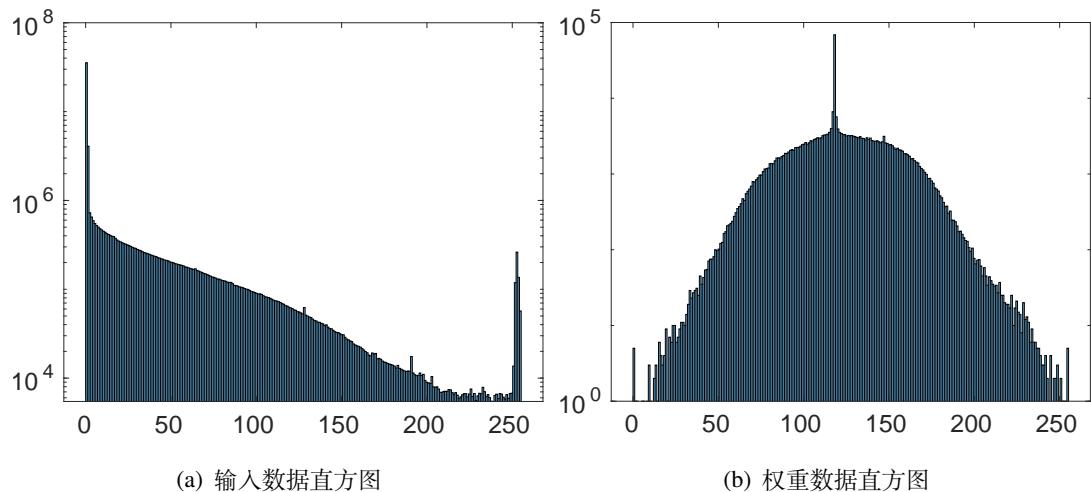


图 3-20 基于 8 比特位宽量化的 LeNet 网络在 MNIST 推理数据集上的输入和权重数据直方图

首先对基于 8 比特位宽量化的 LeNet 网络在 MNIST 推理数据集^[113]上的数据进行分析，提取了所有层的输入和权重数据，直方图分别如图3-20(a)和如3-20(b)所示，可以看到输入集中在 0 和 255 附近，权重集中在 128 附近。设 $R = 0.4$ ，

表3-1 采用不同近似乘法器近似后的LeNet网络在MNIST数据集的精度

Metric	KMap [96]	CR (C.6) [97]	CR (C.7) [97]	AC [118]	RoBA [119]	OU (L.1) [100]	OU (L.3) [100]	SDLC [98]	DRUM [99]	TOSAM [91]	PPAM [120]	Evo8 [107]	EvoLite [121]	Exact
Accuracy (%)	96.32	74.88 / 81.79	97.77 / 98.26	18.28	99.31	11.35	97.28	98.07 / 97.92	26.01 - 99.1	19.72 - 99.32	9.8 - 99.27	9.74 - 99.43	9.78 - 99.42	99.41

由(3.28)得 l 为 6, 即对 8×8 无符号乘法器的前 6 行部分积进行分簇压缩, 且需要考虑输入极性。在由算法2得 $P = 0$ 和 $P = 1$ 时的 λ 均为 5000 后, 与 $l = 6$ 一起根据式(3.25)生成近似乘法器。

表3-1展示了在考虑输入极性的情况下采用不同近似乘法器近似后的LeNet网络在MNIST数据集的精度, KMap、AC、RoBA、OU、DRUM、TOSAM、以及Evo8和EvoLite中的一些乘法器是对称的。Evo8中的三个乘法器mul8_98、mul8_108和mul8_154的精度最高, 达到99.43%, 但它们的硬件成本太高, 例如, 在2GHz的时钟频率约束下, mul8_108的PDA是DC从DesignWare库自动构建的精确乘法器DesignW的2倍。在EvoLite中, mul8u_ZFB精度最高, 为99.39%。为了直观的对不同乘法器的性能进行比较, 选择表3-1精度高于98%的乘法器(Evo8中精度高于99.2%的乘法器)与生成的乘法器进行比较, 结果如图3-21所示, PDA是基于2GHz的时钟频率约束下得到的, XFYW和XWYF分别代表基于 $P = 0$ 和 $P = 1$ 生成的近似乘法器。

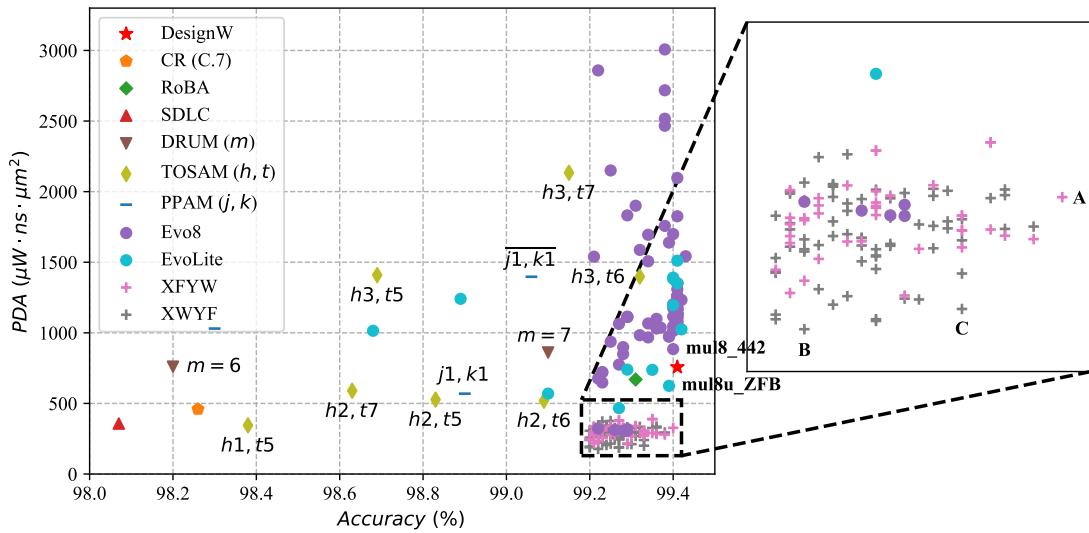


图3-21 不同乘法器在LeNet和MNIST上的精度和2GHz时钟频率下的PDA散点图

在图3-21中, $\overline{\text{PPAM}(j, k)}$ 表示 $\text{PPAM}(j, k)$ 的比较器版本, 通过引入一个比较器决定是否交换乘法器的两个输入数来实现。从图中可以看到, TOSAM的效果好于DRUM, 这是合理的, 因为TOSAM有一个额外的截断操作来减少误差。对

于 PPAM 来说，同样配置参数下有比较器的版本精度更高，但代价是消耗了更多的硬件资源。在基于本文的方法生成的乘法器 XFYW 和 XWYF 中，‘A’、‘B’ 和 ‘C’ 分别代表了生成的精度最高、硬件成本最低、以及精度硬件权衡最好的乘法器。乘法器 ‘A’ 的精度为 99.4%，仅比精确乘法器的精度低 0.01%。与 DesignW、Evo8 中的 mul8_442、EvoLite 中的 mul8u_ZFB 和 TOSAM(3,6) 相比，‘A’ 的 PDA 分别提升了 56.8%、63.0%、47.6% 和 76.6%。‘B’ 以小于 0.2% 的精度损失实现了图3-21所有的乘法器中最低的 PDA 值。与 SDLC、DRUM(6) 和 TOSAM(1,5) 相比，‘B’ 的 PDA 分别降低了 50.3%、76.6% 和 48.4%。与 RoBA 相比，乘法器 ‘C’ 的精度高了 0.02%，PDA 低了 70.0%。同时，极性相反的 XFYW 和 XWYF 乘法器并没有显现出明显的性能区别，这可能是由于 LeNet 网络结构简单、对误差的容忍性较大的原因。

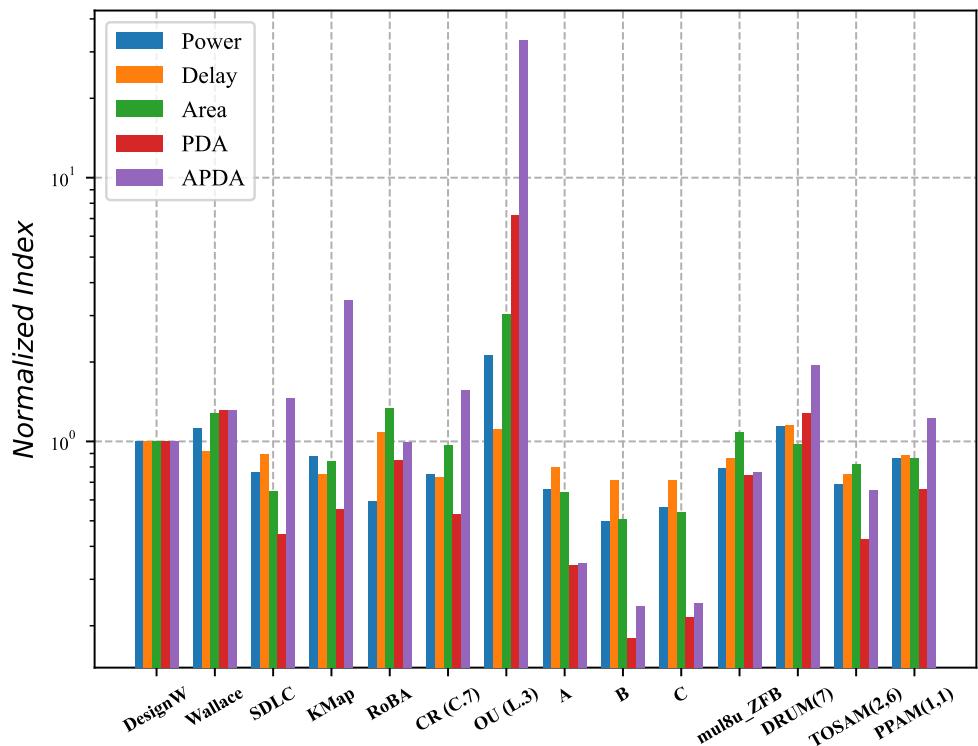
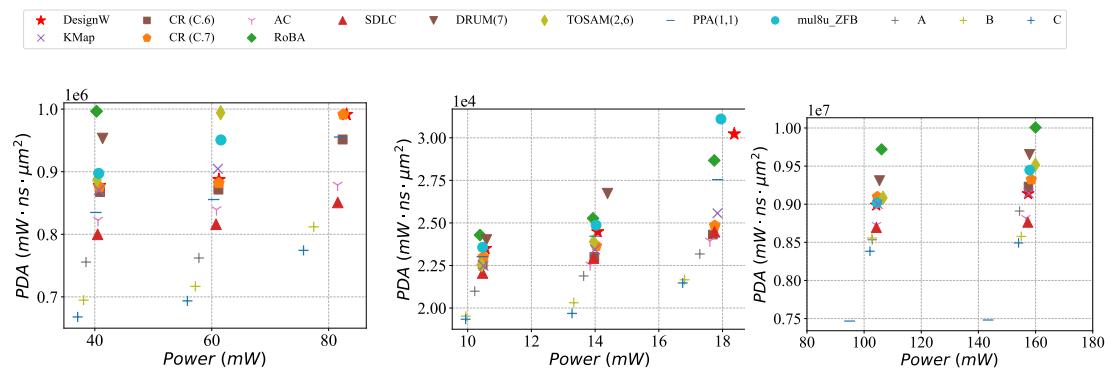


图 3-22 基于 LeNet 和 MNIST 的不同乘法器的功耗、延迟、面积、PDA 和 APDA，以 DesignW 为标准进行归一化

图3-22展示了以 DesignW 为标准进行归一化后的不同近似乘法器的功耗、延迟、面积、PDA 和 APDA 的比较，数据基于 0.5GHz 的时钟频率约束得到，其中 Wallace 是采用 Wallace 树结构对部分积进行累加的乘法器。可以看到，RoBA 是一种以面积换取功耗的设计；OU 的电路面积随着误差补偿级别的增加增长过；CR 乘法器的延迟极低，这符合它仅允许进位信号最多传播两个加法器的设计理念；EvoLite 中的 mul8u_ZFB 乘法器的面积略大于 DesignW，这暗示基于门级结

构对精确乘法器进行简化可能不是个好主意；DRUM(7) 的功耗和延迟都略高于 DesignW，这是因为需要检查的比特数为 7，引入检测器带来的成本超过了小位宽乘法器带来的受益；TOSAM(2,5) 的 APDA 比 PDA 高出一大截，这标明其硬件复杂度低，但精度损失高。同时可以观察到，乘法器 ‘A’、‘B’、‘C’ 在功耗、延迟、面积、PDA 和 APDA 方面优于所有乘法器。与 mul8u_ZFB 相比，‘A’ 的面积减少了 44.57%，延迟减少了 6.3%，功耗减少了 13.14%，PDA 减少了 54.4%，APDA 减少了 55.1%。与 RoBA 相比，‘C’ 在面积、延迟和功耗方面分别提升了 59.6%、34.0%、5.1%、60.3% 和 65.5%。



(a) 1GHz、1.5GHz 和 2GHz 下的 SA (b) 1.5GHz、2GHz 和 2.5GHz 的 SC (c) 1GHz 和 1.5GHz 下的 TASU

图 3-23 不同 DNN 加速器在多个时钟频率约束下基于不同乘法器得到的功耗和 PDA 指标

为了展示生成的近似乘法器对实际的 DNN 硬件加速器到底带来了多少提升，将不同的近似乘法器对三个 DNN 加速器中的精确乘法器进行替换并利用 DC 进行综合得到功耗、延迟和面积，考虑的加速器包括：(1) 针对 DoReFa-Net 网络^[124]设计的 TASU^[125]；(2) 针对 DNN 中卷积操作进行加速的 SC^[126]；(3) 谷歌的 TPU 采用的脉动阵列 (Systolic Array) 架构^[28]，简称为 SA。

图3-23展示了三个加速器基于不同近似乘法器在多个时钟频率约束下得到的功耗和 PDA 指标，其中 SA 的规模为 16×16 。可以看到，基于 ‘B’ 和 ‘C’ 的实现的 SA 和 SC 加速器在考虑的所有频率下都优于基于其他乘法器实现的版本。基于 PPAM(1,1) 的 TASU 加速器功耗和 PDA 较低，这可能是因为 PPAM 在 Verilog 编写时采用多位宽的“+”运算符来对乘法器的部分积进行累加，能够让 EDA 工具对其进行有效地优化。

AleNet 和 CIFAR-10

为了验证提出的方法对不同规模神经网络的有效性，对基于 CIFAR-10 数据集^[116]的 AlexNet 网络^[114]也进行了类似的实验，从 AlexNet 中提取的输入和

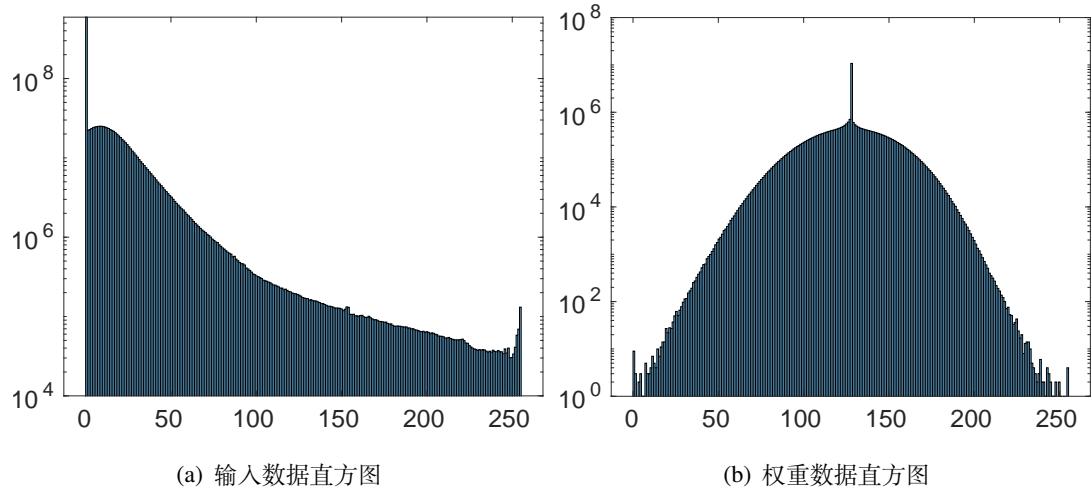


图 3-24 基于 8 比特位宽量化的 AlexNet 网络在 CIFAR-10 推理数据集上的输入和权重数据直方图

表 3-2 采用不同近似乘法器近似后的 AlexNet 网络在 CIFAR-10 数据集的精度

Metric	KMap [96]	CR (C.6) [97]	CR (C.7) [97]	AC [118]	RoBA [119]	OU (L.1) [100]	OU (L.3) [100]	SDLC [98]	DRUM [99]	TOSAM [91]	PPAM [120]	Evo8 [107]	EvoLite [121]	Exact
Accuracy (%)	49.67	10 / 10	10.01 / 10.02	10.06	8.92	10	61.21	10.29 / 24.94	10.0 - 87.5	9.09 - 86.81	10.0 - 85.86	9.85 - 88.54	10.0 - 88.47	88.39

权重数据直方图如图3-24所示，可以看到其概率分布与基于 MNIST 的 LeNet 类似^[113]，但集中程度更高。与 LeNet 相比，AlexNet 的网络深度更大，在采用相同近似乘法器时会导致更多的误差积累，引起较多的精度下降。考虑到 AlexNet 对误差的容忍性不如 LeNet，假设 $R = 0.3$ ，由式(3.28)的 $l = 4$ ，即取 8×8 无符号乘法器的前四行部分乘积进行压缩，同时由算法2得 $P = 0$ 和 $P = 1$ 时的 λ 均为 500，与 $l = 4$ 一起根据式(3.25)生成近似乘法器。

表3-2展示了在考虑输入极性的情况下采用不同近似乘法器近似后的 AlexNet 网络在 CIFAR-10 数据集的精度，可以看到与 LeNet 相比下降了很多。在表3-2中，OU(L.3) 的精度远高于 KMap、RoBA 和 SDLC，这主要是因为 OU(L.3) 采用了 3 次划分的设计方式，提供了足够的误差补偿。DRUM(7)、TOSAM(3,5)、PPAM(0,1) 和 PPAM(0,1) 精度分别为 87.50%、86.81%、85.63% 和 85.86%，但它们的在 2GHz 下的 PDA 平均比 DesignW 多了 18.5%。

图3-25展示了不同乘法器在基于 CIFAR-10^[116]数据集的 AlexNet 网络^[114]上评估得到的精度和利用 DC 在 2GHz 时钟频率约束下获得的 PDA 散点图，为了保证公平，除了标明输入的 XFYW 和 XWYF 之外，其余所有的非对称乘法器都是取 $P = 0$ 和 $P = 1$ 下精度较高的值进行比较。图3-25只显示了一



图 3-25 不同乘法器在 AlexNet 和 CIFAR-10 上的精度和 2GHz 时钟频率下的 PDA 散点图

部分乘法器的原因是只有这部分乘法器的精度和 PDA 在坐标表示的范围之内，其余的乘法器要么精度不够、要么 PDA 太差。对比的精确乘法器有两个版本，分别是 DesignW 和 Wallace。可以看到，虽然 Evo8 中有些乘法器的精度损失很低，但 PDA 比 DesignW 和 Wallace 还要高，硬件成本不可接受。EvoLite 中的 mul8u_ZFB 乘法器以精度损失小于 0.1% 的代价实现了比 DesignW 更好的 PDA 值，是一个不错的设计。

值得一提的是，基于本文的方法生成的近似乘法器 XFYW 和 XWYF 效果最好，由于 DNN 的特性，甚至存在近似后神经网络的精度更高的情况，这可能是因为 DNN 的冗余性。如图3-25中的乘法器 ‘E’、‘D’、‘F’ 所示，它们不仅拥有比 DesignW 更低的 PDA，还拥有比 DesignW 更高的精度。另一个有趣的现象是，基于 $P = 0$ 生成的许多 XFYW 乘法器的精度比基于 $P = 1$ 生成的 XWYF 乘法器要高，这有力地证明了在设计近似乘法器时考虑输入极性的重要性。在所有生成的乘法器 XFYW 和 XWYF 中，‘G’ 的 PDA 最低，与 DesignW 和 mul8u_ZFB 相比，‘G’ 的 PDA 分别改进了 45.8% 和 34.3%。选择 6 个生成的乘法器 ‘D’、‘E’、‘F’、‘G’、‘H’、‘T’ 与两个精确乘法器 DesignW、Wallace 和三个近似乘法器 mul8u_ZFB、DRUM(7)、TOSAM(3,5) 进行进一步地比较。

图3-26展示了 11 个乘法器在 2GHz 时钟频率约束下以 DesignW 为标准进行归一化后的功耗、延迟、面积、PDA 和 APDA 比较图。可以看到根据本文方法生

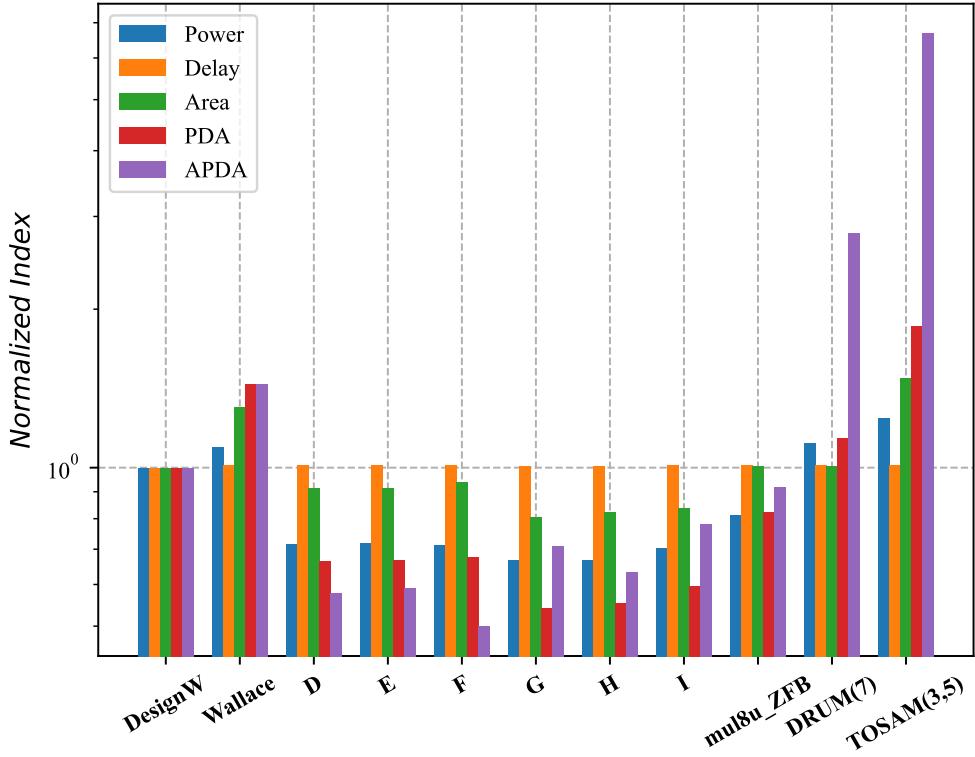


图 3-26 基于 AlexNet 和 CIFAR-10 的乘法器的功耗、延迟、面积、PDA 和 APDA, 以 DesignW 为标准进行归一化

成的 6 个乘法器拥有最优的 PDA 和 APDA, TOSAM(3,5) 的硬件成本过高, 且精度损失也较大。与 DesignW 相比, 6 个生成的乘法器 (‘D’-‘I’) 平均改进了 30.2% 的功耗、12.8% 的面积、38.4% 的 PDA 和 36.8% 的 APDA; 与 mul8u_ZFB 相比, 6 个生成的乘法器的乘法器平均提高了 13.9% 的功耗、13.3% 的面积、25.3% 的 PDA 和 31.3% 的 APDA。

将基于 AlexNet^[114]和 CIFAR-10^[116]生成的乘法器 (如图3-25中的 XFYW 和 XWYF 所示) 放在 LeNet 中通过 MNIST 进行评估, 大部分乘法器的精度都在 99% 以上, 也就是说, 按照本文的方法基于较大规模 DNN 生成的近似乘法器在面对较小规模 DNN 时表现出了一定的可迁移性。

VGG16 和 CIFAR-10

为了进一步证明提出的自动化近似乘法器设计方法在大规模神经网络下的适用性, 对基于 CIFAR-10 数据集^[116]的 VGG16 神经网络^[115]进行输入和权重的数据提取, 直方图如图3-27所示, 可以看到其分布表现出了与 LeNet 和 AlexNet 相似的特性。考虑到 VGG16 的网络层数比 LeNet 和 AlexNet 都大, 近似后的误差累计更多, 能够在 VGG16 上不引起大幅精度下降的近似乘法器可能与精

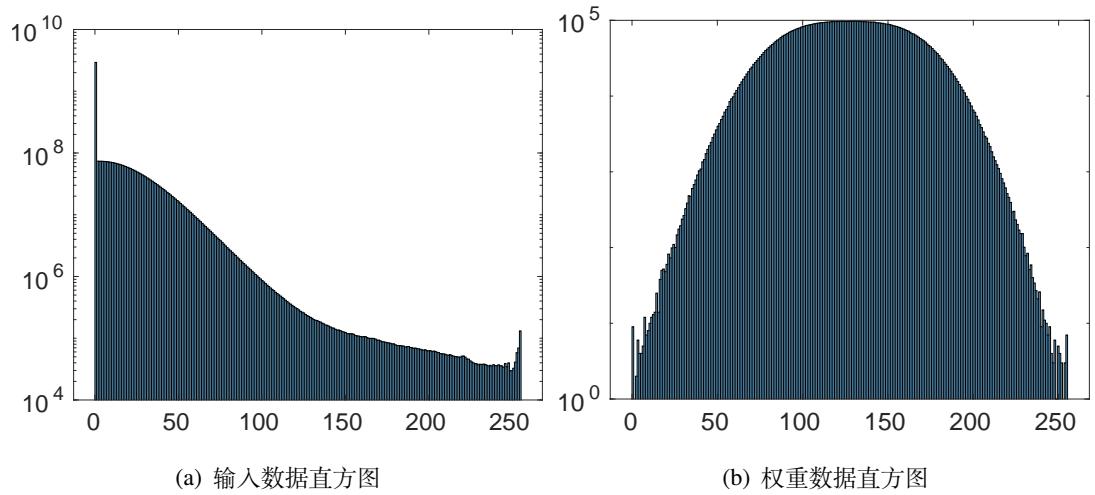


图 3-27 基于 8 比特位宽量化的 VGG16 网络在 CIFAR-10 推理数据集上的输入和权重数据直方图

确乘法器相比没有特别明显的硬件优势，假设 $R = 0.2$ ，由式(3.28)得 $l = 4$ ，根据算法2得 $P = 0$ 和 $P = 1$ 时 $\lambda = 5000$ ，这里对 λ 进行调整，同时考虑 $\lambda \in \{1000, 2000, 3000, 4000, 5000\}$ 生成近似乘法器。表3-3展示了在考虑输入极

表 3-3 采用不同近似乘法器近似后的 VGG16 网络在 CIFAR-10 数据集的精度

Metric	KMap [96]	CR (C.6) [97]	CR (C.7) [97]	AC [118]	RoBA [119]	OU (L.1) [100]	OU (L.3) [100]	SDLC [98]	DRUM [99]	TOSAM [91]	PPAM [120]	Evo8 [107]	EvoLite [121]	Exact
Accuracy (%)	10	10.83 / 10.84	9.56 / 9.09	10.28	10.01	10.01	10	9.95 / 10	8.89 - 51.04	7.99 - 76.21	7.79 - 15.67	8.58 - 89.82	9.01 - 89.45	89.45

性的情况下采用不同近似乘法器近似后的 VGG16 网络^[115]在 CIFAR-10 数据集^[116]上的精度，可以看到与精确乘法器相比，大部分乘法器的精度下降都十分剧烈，比如 DRUM(7) 和 TOSAM(3,5) 的准确率分别只有 51.04% 和 76.21%。这表明与 LeNet 和 AlexNet 相比，VGG16 对误差的容忍程度更低。

图3-28展示了2个精确乘法器和多个近似乘法器在基于CIFAR-10^[116]数据集的VGG16网络^[114]上评估得到的精度和利用DC在2GHz时钟频率约束下获得的PDA散点图，基于本文提出的方法生成的XFYW乘法器具有从89.2%到89.54%的不同精度（有两个乘法器的精度高于精确乘法器），PDA优于图中出现的所有近似乘法器。虽然基于CGP方法设计的Evo8^[107]和EvoLite^[121]中部分乘法器精度尚可，但硬件成本显然比精确的DesignW乘法器高出数倍，这表明通过门级网表简化的办法在面向对误差容忍程度较低的应用时无法获得高性能的近似乘法器，违背了近似计算的初衷。另外，基于 $P=1$ 生成的XWYF乘法器没有出现在图3-28中，表明要么它们的准确率低于89.15%要么具有非常差的



图 3-28 不同乘法器在 AlexNet 和 CIFAR-10 上的精度和 2GHz 时钟频率下的 PDA 散点图

PDA，这再一次证明了在设计非对称近似乘法器时考虑输入极性的重要性。与 DesignW 相比，乘法器 ‘M’ 的 PDA 改进了 26.4%，且精度更高（89.50%）。选择 ‘J’、‘K’、‘L’ 和 ‘M’4 个生成的近似乘法器与两个精确的乘法器进行进一步地比较。图3-29展示了 2 个精确乘法器 DesignW、Wallace 和 4 个生成的近似乘法

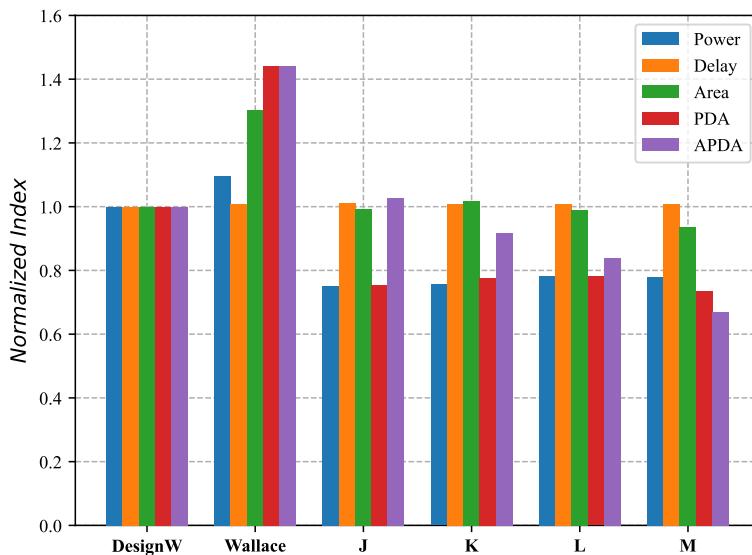


图 3-29 基于 VGG16 和 CIFAR-10 的不同乘法器的功耗、延迟、面积、PDA 和 APDA，以 DesignW 为标准进行归一化

器在 2GHz 时钟频率约束下以 DesignW 为标准进行归一化后的功耗、延迟、面积、PDA 和 APDA 比较图，可以看到与 DesignW 相比，4 个近似乘法器的延迟稍差，意味着高频性能不足，但面积和功耗领先。以乘法器 ‘M’ 为例，功耗、面积、PDA 和 APDA 分别比 DesignW 好了 22.1%、6.4%、26.4% 和 33.1%。

将图3-29中基于 VGG16 和 CIFAR-10 生成的 XFWY 近似乘法器分别放在 LeNet 和 AlexNet 中进行评估，其准确率分别为 99.38%-99.41% 和 88.18%-88.45%，注意 LeNet 和 AlexNet 在精确乘法下的准确率分别是 99.40% 和 88.39%，这再一次证明了针对大规模 DNN 设计的近似乘法器在面对小规模 DNN 时有可迁移性。

3.6.3 基于 16 比特补码有符号定点数的自适应 FIR 滤波器

为了证明提出的方法对有符号乘法器同样有效，对基于 16 比特补码有符号定点数的自适应 FIR 滤波器进行输入数据特性分析并生成乘法器。一个基本的

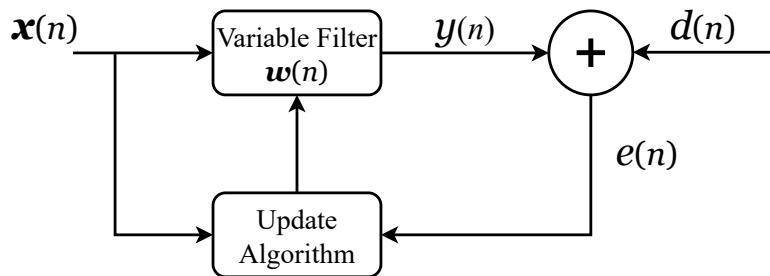


图 3-30 一个自适应 FIR 滤波器的结构图

自适应 FIR 滤波器的结构图如图3-30所示，其中 n 代表迭代次数， $\mathbf{x}(n)$ 是输入向量， $\mathbf{w}(n)$ 是权重向量，由最小均方 (Least mean squares, LMS) 算法通过负反馈回路进行调整， $y(n)$ 是输出信号， $d(n)$ 是参考信号， $e(n)$ 是误差。在 M 阶 FIR 滤波器中，假设第 n 次迭代时 $\mathbf{w}(n) = [w_0(n), w_1(n), \dots, w_{M-1}(n)]$ ， $\mathbf{x}(n) = [x(n), x(n-1), \dots, x(n-M+1)]^T$ ，输出信号 $y(n)$ 由下式计算：

$$y(n) = \mathbf{w}(n) \cdot \mathbf{x}(n) = \sum_0^{M-1} w_i(n) \cdot x(n-i) \quad (3.36)$$

误差 $e(n)$ 为：

$$e(n) = d(n) - y(n) \quad (3.37)$$

权重向量由 LMS 算法进行调整，第 $n+1$ 次迭代的第 i 阶重 $w_i(n+1)$ 为：

$$w_i(n+1) = w_i(n) + \mu \cdot e(n) \cdot x(n-i) \quad (3.38)$$

其中 $i = 0, 1, \dots, M-1$ ， μ 代表步长。

自适应 FIR 滤波器由误差计算和权值更新两个模块组成，若阶数为 M ，则总共需要 $2M$ 个乘法器。考虑一个基于 16 比特补码有符号数实现的 20 阶低通 (Low pass) 自适应 FIR 滤波器，LMS 算法的步长为 0.001，参考信号 $d(n)$ 是一个叠加了高频噪声的幅值为 5 的正弦信号：

$$d(n) = 5 * \sin\left(\frac{\pi n}{25}\right) + 0.5 * \sin\left(\frac{2\pi n}{5} + \frac{\pi}{4}\right) \quad (3.39)$$

其中 $n \in \{0, 1, \dots, 999\}$ ，输入信号与噪声有一个 $\frac{\pi}{4}$ 的相位差：

$$x(n) = \sin\left(\frac{2\pi n}{5}\right) \quad (3.40)$$

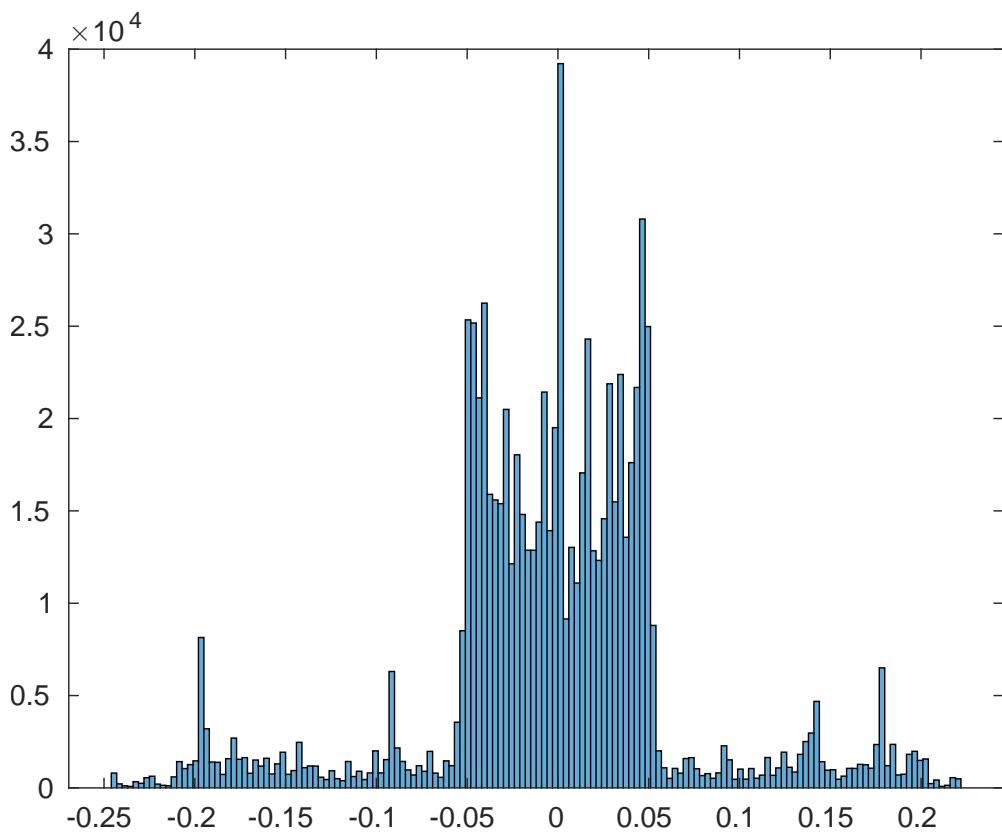


图 3-31 滤波器在精确乘法下权重的数据分布

首先在精确乘法下运行滤波器并得到对应的权重数据，结果如图3-31所示，可以看到分布是不均匀的。假设 $R = 0.2$ ，由式3.28得 $l = 6$ ，根据算法2得 $P = 0$ 和 $P = 1$ 下的 λ 均为 5000，基于式(3.25)生成近似乘法器并进行峰值信噪比 (Peak Signal-to-Noise Ratio, PSNR) 的评估。

图3-32展示了不同乘法器基于 1GHz 时钟频率约束的 PDA 和 PSNR 对比散点图，其中 DesignW 代表 DC 通过 DesignWare 库^[123]自动构建的 16 比特精确乘法器，XSYW 和 XWYS 分别代表基于 $P = 0$ (x 是输入， y 是权重) 和 $P = 1$ (x

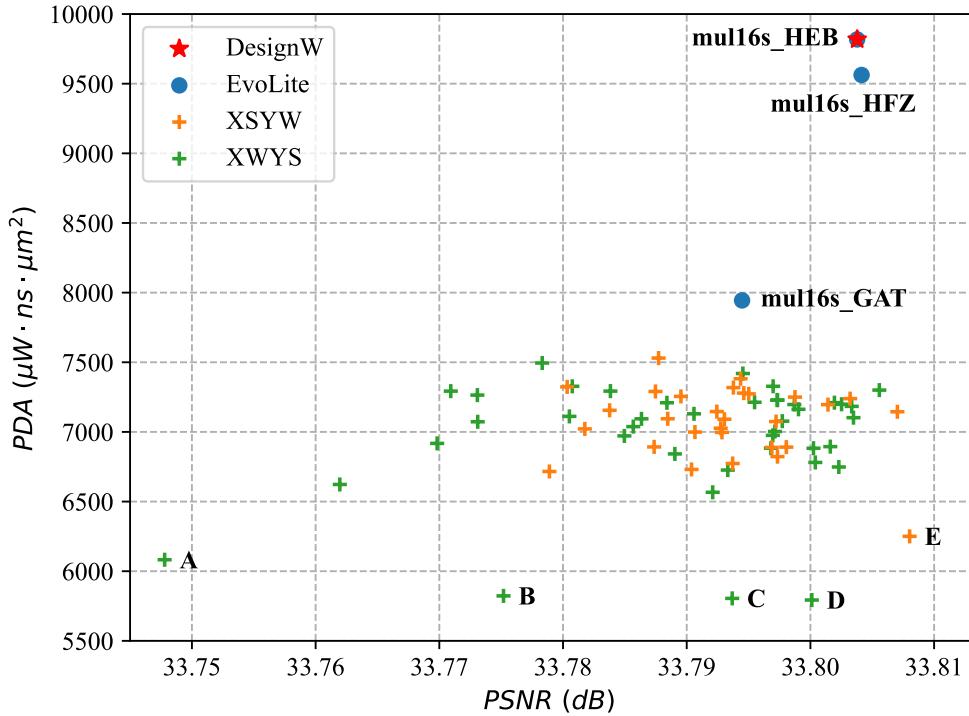


图 3-32 不同乘法器的 PDA 和 PSNR 对比散点图

是权重， y 是输入）生成的乘法器。可以看到按照本文方法设计的近似乘法器在几乎没有 PSNR 损失的情况下提供了更好的 PDA 值，与 DesignW 和 EvoLite 中的 mul16s_GAT 乘法器相比，5 个生成的乘法器 ‘A’、‘B’、‘C’、‘D’、‘E’ 的 PDA 平均分别提升了 41.0% 和 27.1%。

为了对不同乘法器的质量进行统一地比较，引 PDA 和相对峰值信噪比损失的乘积（Product of PDA relative PSNR loss，LPDA）来表征乘法器的好坏，相对 PSNR 损失被定义为：

$$[P_{exact}] - P_{mul} \quad (3.41)$$

其中 P_{exact} 代表滤波器在精确乘法器下的 PSNR， $[]$ 代表向上取整， P_{mul} 表示滤波器在测试乘法器下的 PSNR，注意精确乘法器的相对 PSNR 损失不为 0。

图3-33展示了不同乘法器在 200MHz 时钟频率约束下以 DesignW 为标准进行归一化之后的功耗、延迟、面积、PDA 和 LPDA 对比图，可以看到 mul16s_GAT 的功耗比 DesignW 低 20% 左右，但面积没有很大优势。与 mul16s_GAT 相比，根据本文的方法生成的乘法器 ‘A’-‘E’ 的 PDA 和 LPDA 平均提升了 12.5% 和 8.3%。

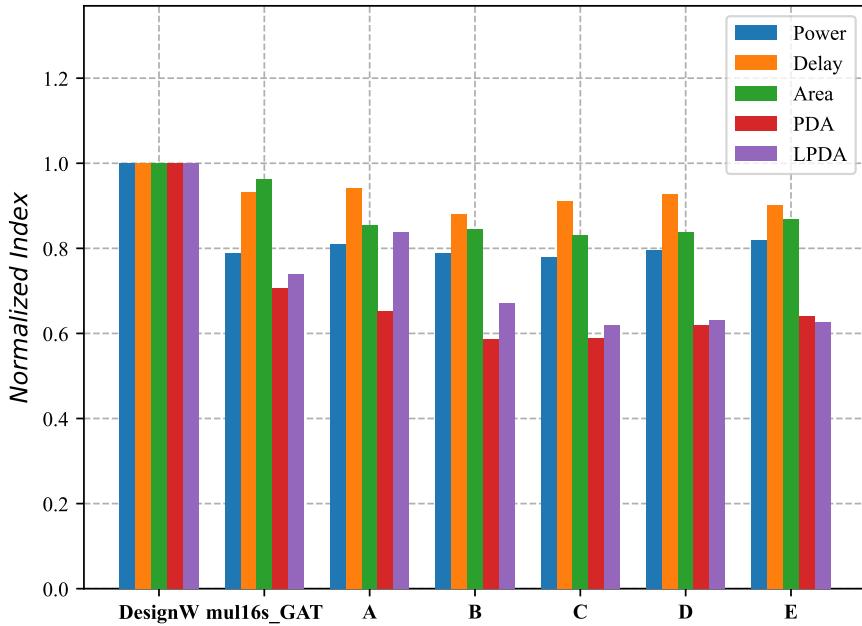


图 3-33 不同乘法器在 200MHz 时钟频率约束下的功耗、延迟、面积、PDA 和 LPDA，以 DesignW 为标准进行归一化

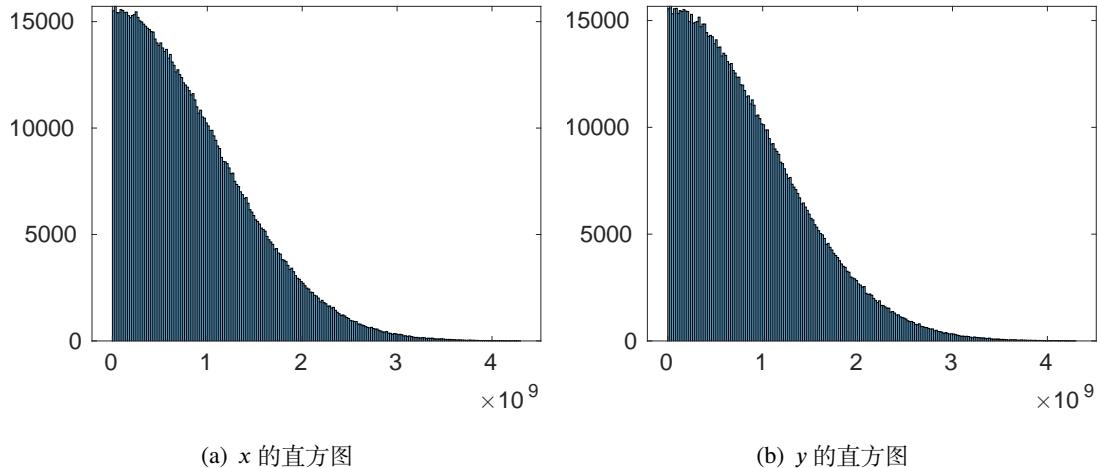


图 3-34 基于平均值为 0、标准差为 2^{30} 的正态分布随机生成的 2^{20} 对大于 0 的 32 比特输入数据直方图

3.6.4 半正态分布下的无符号 32 比特乘法器

为了验证方法在大位宽乘法器下的有效性，基于一个平均值 μ 为 0、标准差 σ 为 2^{30} 的正态分布 (Normal distribution) 随机生成了 2^{20} 对大于 0 的 32 比特输入数据，其直方图如图3-34所示。假设某一应用的乘法器也基于 32 比特的无符号数实现，且输入数据只从生成的 2^{20} 种情况中选择，精度由该 2^{20} 对输入下的平均绝对误差 MAE (即平均误差距离 MED，见式2.42) 决定。考虑到 x 和 y 分

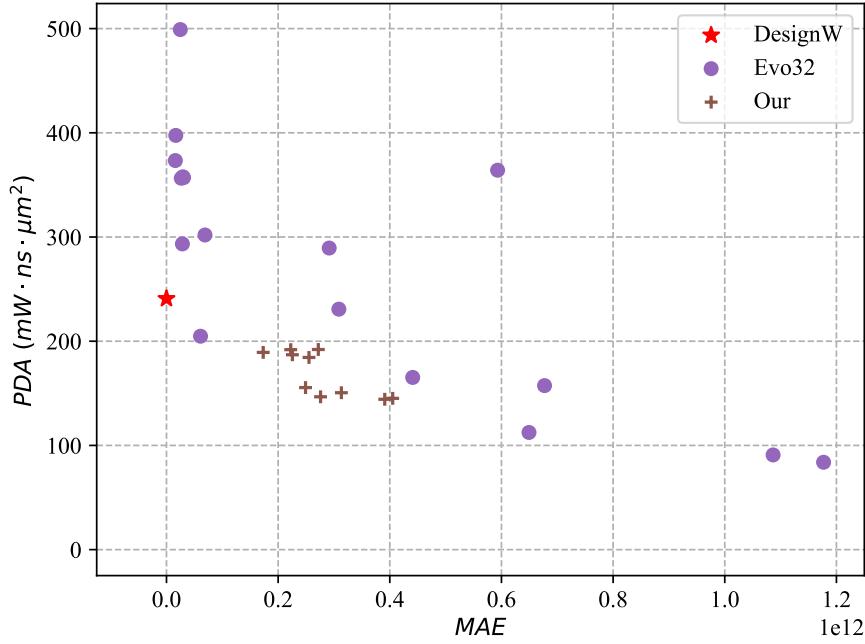


图 3-35 生成的近似乘法器与 EvoLite 中的 32 位无符号乘法器（Evo32）在 1.5GHz 时钟频率的约束下进行 MAE 和 PDA 比较

布相同，所以无需考虑极性，且输入情况可以遍历。假设 $R = 0.15$ ，由式(3.28)得 l 取 10，根据算法2确定 λ 为 500 后，求解(3.25)得到近似乘法器。在 1.5GHz 的时钟约束下与 EvoLite^[121]中的 32 比特无符号乘法器（简称为 Evo32）进行 MAE 和 PDA 的对比，结果如图3-35所示，可以看到生成的乘法器与 Evo32 和精确乘法器一起组成了帕累托前沿（Pareto front），证明了方法对大位宽乘法器的有效性。

3.7 本章小节

本章提出了一种由输入分布和极性驱动的自动化近似乘法器设计方法，该方法通过逻辑和移位操作对乘法器中的部分积在生成后、累加前进行压缩，以减轻后续的累加压力，达到降低电路设计复杂度的目的。

基于均匀分布下的 8 比特无符号乘法器的实验结果标明，该方法生成的乘法器大幅优于已有的工作。对 LeNet、AlexNet 和 VGG16 三种不同规模的采用 8 比特无符号数量化的 DNN 的实验结果表明，与最先进的近似乘法器相比，本文生成的乘法器在精度损失不超过 0.01% 的情况下实现了 26.4%-47.6% 的 PDA 收益。另外，AlexNet 和 VGG16 的结果证明了在设计非对称近似乘法器时考虑输入极性的重要性，也验证了基于提出的方法利用可迁移性为一类具有相似数据分布的应用生成一个乘法器的可行性。对补码有符号乘法器的有效性在 16 位自适应 LMS-FIR 滤波器中得到了证明，与国际前沿工作相比，生成的乘法器在

PSNR 损失可以忽略不计的情况下实现了 PDA 高达 27.1% 的提升。最后，基于 32 比特半正态分布的无符号数乘法器的实验结果表明，生成的乘法器与别的乘法器一起组成了帕累拖前沿，表明方法对大位宽乘法器同样有效。

第 4 章 面向 FPGA 的基于贝叶斯优化的自动化近似乘法器生成器

4.1 研究背景与现状

FPGA 应用中的算术操作通常由 DSP 模块实现，但 DSP 电路的面积只占整个 FPGA 芯片的 5%，且位置固定^[127]，这意味着某些需要大量乘法的应用比如 DNN 无法在 FPGA 上被有效的映射^[128]。相对应的，FPGA 中存在着丰富的查找表 LUT，能够和布线资源一起实现复杂的函数功能，某些充分考虑 LUT 特性的设计能够在 FPGA 上实现相当高的性能^[129]。然而，一块 FPGA 芯片的容量是有限的，对于大型设计来讲往往需要将其划分后部署到不同的 FPGA 芯片上，这会极大地降低电路的性能。同时 FPGA 中的布线资源昂贵，如果使用的 FPGA 资源过多，有可能会造成布线拥塞，也会对电路性能造成影响。所以有必要在 FPGA 中使用近似乘法器来提高乘法的效率，降低 FPGA 资源的使用量，提高电路性能。

目前已有的近似乘法器的工作大多是面向 ASIC 的^[53,96–100,107,118]，由于在 ASIC 中组合逻辑（Combinational logic）由逻辑门（Logic gate）和金属线（Metal wire）构成而在 FPGA 中由 LUT 和布线资源组成，因此基于 ASIC 实现的近似乘法器在 FPGA 上往往无法获得相同程度的硬件性能提升，需要专门开发面向 FPGA 结构的近似乘法器设计方法。

LUT 通常由多路选择器 MUX 和 SRAM 构成，根据输入个数的不同进行规模的区分，一个 n 输入的 LUT 包含 $2^n - 1$ 个 MUX 和 2^n 个 SRAM，能实现任意的 n 输入函数（共 2^{2^n} 个），同时可看作两个 $n - 1$ 输入 LUT 和一个二选一 MUX 的组合。目前最常用的 LUT 的输入个数为 6^[130]，简称为 LUT6。图4-1展示了一个由两个 LUT5 组成的 LUT6 示意图，被称为 LUT6_2。LUT6_2 共包含 64 个 SRAM，在编码后每个 SRAM 都有一个确定的值，被称为初始（Initial, INIT）值，不同的 INIT 值的组合能够实现不同的函数，一共有 2^{64} 种组合，能够实现任意的 6 输入函数。

虽然 LUT 能够实现特定输入数下的任意单输出函数，但对算术操作这种多输入多输出的运算效率并不高。考虑到加法使用最频繁，为了提高性能，FPGA 引入了进位链来降低加法的进位延迟，一种典型的 2 比特位宽的进位链结构如

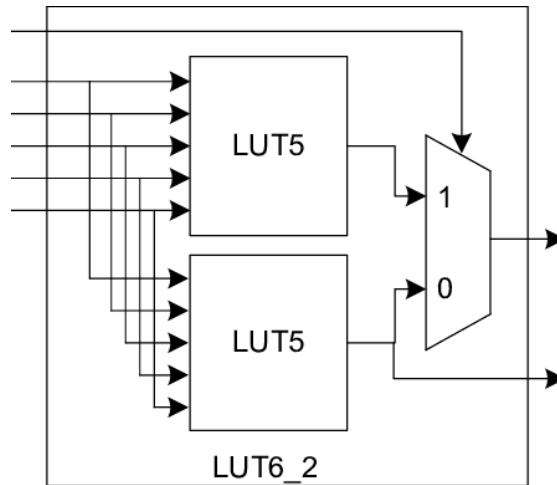


图 4-1 一个典型的拥有 6 个输入的 LUT 结构图

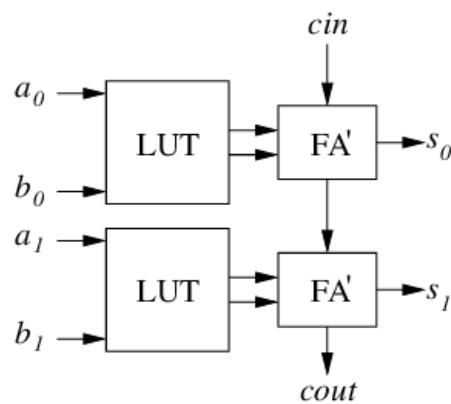


图 4-2 一种 2 比特位宽的 FPGA 进位链结构示意图

图4-2所示，由两个进位单元级联而成，每个进位单元包含一个 LUT 和一个伪全加器 FA'，每个 LUT 的结构如图4-1中的 LUT6_2 所示。根据式(2.19)得全加器的进位信号： $c_{i+1} = p_i c_i + \bar{p}_i g_i$ ，求和信号： $s_i = p_i \oplus g_i$ ，则可利用 LUT6_2 中的两个 LUT5 产生进位的传播信号和生成信号送给伪全加器 FA'，伪全加器 FA' 通过一个异或门 XOR 和一个二选一 MUX 进行求和及进位输出，MUX 可由延迟较低的传输管结构实现。通过级联更多的进位单元可实现更大位宽的进位链。

文献^[131]通过手动修改精确乘法器中用于部分积累加的 LUT 的 INIT 值，设计了几个拥有不同误差和不同硬件成本的近似乘法器，提出并开源了第一个面向 FPGA 的近似乘法器库 SMAApproxLib。类似地，文献^[132-134]也采用修改精确模式下 LUT 的 SRAM 编码来得到不同质量的近似乘法器。然而，这些手工设计方法通常非常耗时，并且生成的乘法器之间误差和硬件性能差距很大，无法根据应用的需求进行灵活地调整，因此需要针对 FPGA 提出一个新的自动化设计方法，能够在短时间内生成许多适合应用需要的高质量的基于 LUT 实现的软核

(Softcore) 近似乘法器。

4.2 研究动机

为了证明同一个近似乘法器在 ASIC 和 FPGA 下会表现出不同的性能差异，对面向 ASIC 根据 CGP 方法生成的近似乘法器库 Evo8^[107]进行 FPGA 评估，并将结果与 ASIC 下进行对比。

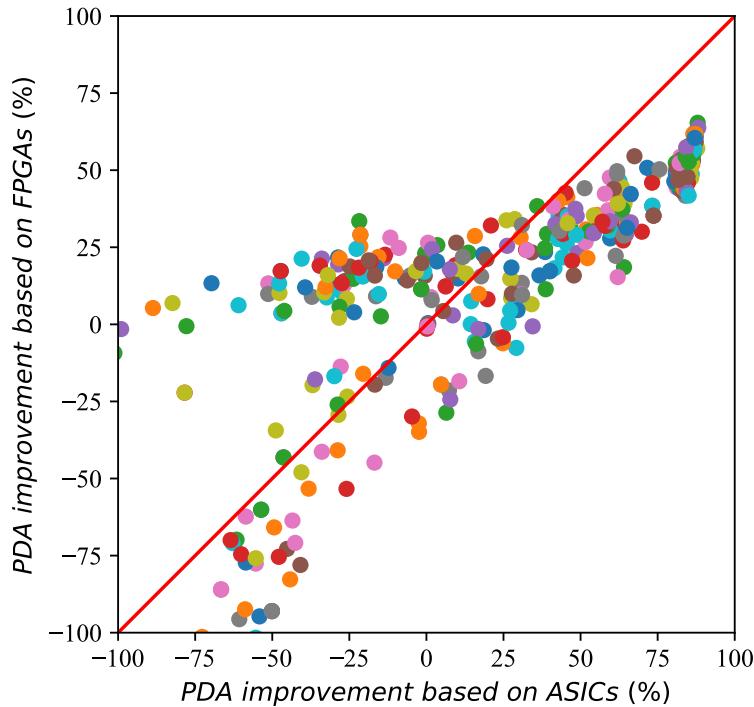


图 4-3 Evo8 中的所有乘法器在 ASIC 和 FPGA 下的 PDA 提升

图4-3展示了不同乘法器在 ASIC 和 FPGA 下分别与精确乘法器相比的 PDA 提升，其中 ASIC 的结果是通过 DC 基于 2GHz 的时钟频率约束在开源的 7nm 工艺库^[122]上进行综合得到的，FPGA 的结果是通过 Vivado 2023.1 在 Virtex Ultra-Scale+ 系列的器件 xcvu3p-ffvc1517-3-e 上基于 100MHz 时钟频率约束得到的，精确乘法器均由 Verilog 中的“*”操作符实现，在 DC 和 Vivado 中分别通过 DesignWare 库^[123]和 Xilinx Multiplier LogiCORE IP^[135]自动构建，PDA 提升由下式进行计算：

$$\frac{\text{PDA}_{ext} - \text{PDA}_{app}}{\text{PDA}_{ext}} * 100 \quad (4.1)$$

其中 PDA_{app} 和 PDA_{ext} 分别代表近似乘法器和精确乘法器的 PDA 值，需要注意的是在 FPGA 中乘法器的面积指标由占据的 LUT 个数表示。在图4-3中，每个点代表一个乘法器，横轴和纵轴分别代表在 ASIC 和 FPGA 下相比精确乘法器的 PDA 提升，由式(4.1)计算得到。落在红线上的点代表该乘法器在 ASIC 和 FPGA

实现了相同程度的硬件性能提升，可以看到许多乘法器并不在红线上，这意味着这些针对 ASIC 设计的乘法器在由 FPGA 映射后无法获得类似的收益，甚至有一些在 ASIC 下 PDA 收益为-25% 的近似乘法器在 FPGA 上却改进了 25%。这是因为 Evo8^[107] 中乘法器由 2 或 3 输入逻辑门构成，在 ASIC 设计中，可以利用综合工具对这些门级电路进行细粒度的控制和优化。然而，FPGA 使用可配置的 LUT 作为逻辑单元，这具有与逻辑门完全不同的特性。此外，在 FPGA 中，布线延迟约占关键路径延迟的 50%^[136]，这部分延迟的改进和 LUT 优化的关系较小。因此，在设计 FPGA 近似乘法器时也应同时考虑布线延迟的优化。

4.3 研究内容与创新点

本文提出并开源了一个面向 FPGA 的基于贝叶斯优化的自动化近似乘法器生成方法，该方法假设乘法器的部分积在累加前存在一次由半加器阵列进行的压缩操作，同时针对半加器提出了 4 种简化方法，利用贝叶斯算法对半加器阵列进行优化，同时保留压缩后累加过程中部分积的粗粒度加法，创新点如下：

- 设计了一种能够生成任意位宽下乘法器的半加器阵列压缩电路的自动化方法，同时根据半加器的两个同权重的输入确定每个半加器的权重，该权重能够反映半加器对输出结果的重要性，以及半加器简化后引入的误差大小；
- 提出了 4 种简化方法来优化半加器的电路，根据期望的面积（LUT 个数）减少比例和每个半加器的权重来决定半加器阵列中哪些半加器考虑被优化，形成一个高质量的搜索空间；
- 基于详细设计地能够同时反映误差和硬件成本的目标函数，利用并行贝叶斯优化对半加器阵列的优化空间进行搜索，压缩后的部分积通过多位宽的加法进行累加，能够被 EDA 工具（如 Vivado）有效地识别，并映射到 FPGA 中的进位链上；
- 与国际前沿工作中的 1167 个乘法器相比，生成的乘法器能够形成 Pareto 前沿，在硬件成本和误差的乘积上平均有 28.70%-38.47% 的改进。

4.4 研究方法

4.4.1 半加器阵列

图4-4展示了一个由 16 个 AND 门生成的 4×4 无符号乘法器的部分积阵列，以十六进制数进行标记，每个虚线框代表一个半加器，对两个部分积进行压缩，生成求和 *Sum* 与进位 *Cout*，共 6 个半加器，组成一个半加器阵列。半加器压缩阵列的输出和未被压缩的部分积 0、7、8、F 一起组成新的部分积阵列，对其进

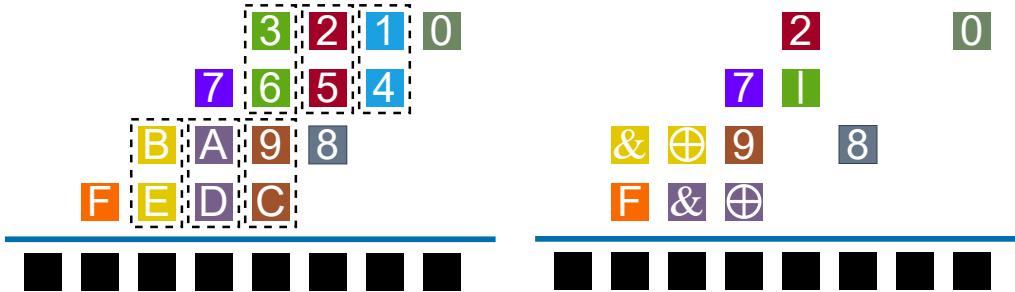


图 4-4 4×4 无符号乘法器的部分积阵列，共 16 个比特

图 4-5 利用搜索到的半加器阵列对部分积进行压缩后的结果

行累加求和后得到最终的结果。利用半加器阵列对部分积进行压缩后能够减少后续累加电路的压力，有利于乘法器的硬件实现。若对半加器阵列进行简化，则可以进一步降低硬件复杂度。假设一个乘法器有 N 行部分积，每行部分积包含 M 个比特，则精确乘法器的半加器压缩阵列需要的精确半加器的数量 S 为：

$$S = (M - 1) \times \lfloor \frac{N}{2} \rfloor \quad (4.2)$$

其中 $\lfloor \cdot \rfloor$ 表示向下取整。例如对图4-4来说 $S = 6$ 。未被半加器阵列压缩的部分积个数为：

$$N + (N \bmod 2) \times (M - 1) \quad (4.3)$$

4.4.2 4 种精确半加器的简化方法

本文提出 4 种精确半加器的简化方法，其中 3 种能够有效降低半加器的硬件成本：

- Eliminate：直接删除该半加器，其求和 Sum 和进位输出 $Cout$ 均接地；
- OR Sum ：求和输出 Sum 由半加器的两个输入进行或运算（OR）得到，进位输出 $Cout$ 直接接地；
- Direct $Cout$ ：将 $Cout$ 连接到两个输入之一，并将 Sum 接地；
- Exact：保持该精确半加器的结构。

在上述的 4 种方法中，“Eliminate” 和 “OR Sum ” 会使半加器的结果变小，而 “Direct $Cout$ ” 会使半加器的结果变大。不同简化方法的结合能够降低半加器阵列的误差，提高乘法器的精度。对压缩阵列中的一部分半加器考虑简化，则生成高质量近似乘法器的问题变成了寻找较优简化操作组合的问题，可利用贝叶斯算法进行搜索。

4.4.3 贝叶斯优化

贝叶斯优化^[137]是一种基于顺序模型（Sequential model-based）的无梯度（Gradient-free）优化方法，常用于优化评估昂贵的黑盒函数（Black-box function）。在贝叶斯优化中，代理模型（Surrogate model，比如高斯过程 Gaussian process）和获取函数（Acquisition function，比如预期改进 expected improvement）一起用于对目标函数进行建模，并决定对哪个点进行采样以在下一次迭代中进行评估。贝叶斯优化需要已知点来训练代理模型，因此通常利用随机搜索构建初始模型，每搜索并评估一个点后，算法对代理模型进行更新并通过最大化获取函数来生成下一个待评估的点。本文使用贝叶斯算法的变体 TPE（Tree-structured Parzen estimator）^[138]来对优化空间进行搜索。

4.4.4 误差分析

由于部分积的权重不同，对不同的半加器进行简化会带来不同大小的误差，对每个半加器引入一个权重，且大小与半加器两个输入比特的权重值一致，则图4-4中输入为部分积 1、4 和输入为部分积 B、E 的两个半加器的权重分别为 1 和 5。理论上讲，由于不同半加器具有不同的权重值，贝叶斯算法在对压缩阵列进行优化时会倾向对低权重的半加器进行电路简化，保持高权重半加器仍然是精确的，从而生成低误差高性能的近似乘法器。然而实际的贝叶斯优化算法并没有如此高效，尤其是面对搜索空间是分离的而不是连续的情况，因此在搜索前可预先选择一部分半加器不参与优化，以提高生成的乘法器的精度。假设乘法器的面积（即 LUT 个数）与压缩阵列中精确半加器的个数 S 成比例（见(4.2)），并使用 R 来表示用户想要的与精确乘法器相比的面积百分比减少，则参与优化的半加器的数量为 $\lfloor S \times R \rfloor$ ，预先保留的精确半加器的数量为 $S - \lfloor S \times R \rfloor$ ，其中 $\lfloor \cdot \rfloor$ 表示四舍五入到最近的整数。图4-5展示了 $R = 0.8$ 时利用搜索到的近似半加器阵列对原始部分积进行压缩后的结果，可以看到压缩后的新部分积阵列的比特总数为 11，与图4-4相比减少了 31.25%。另外需要注意的是，若压缩结果为图4-5，则图4-4中输入为部分积 B、E 和部分积 A、D 的两个半加器分别是预先保留的精确半加器和搜索得到的精确半加器。压缩后的部分积和未被压缩的部分积一起组成新的部分积阵列，通过 Verilog 中的“+”操作符进行累加，可被 FPGA 的 EDA 工具高效地识别并映射到进位链，实现高性能的近似乘法器。

4.4.5 价格计算

贝叶斯优化需要一个价格（cost）计算方法对每个点的好坏进行评估（cost 越小的点质量越高），以对目标函数进行建模和采样，cost 的计算方法决定了贝叶斯优化的搜索方向。例如，如果将 cost 设置为平均绝对误差 MAE（即平均误

差距离 MED, 见(2.42)), 那么贝叶斯优化将会忽略硬件成本开销, 只着重于寻找 MAE 最小的乘法器, 这会导致生成的乘法器性能很差。然而, 近似乘法器的质量是由硬件成本和误差共同决定的, 因此需要一个合适的计算方法对 $cost$ 进行定义。乘法器的硬件成本可以由功耗延迟面积积 PDA 描述, 而误差可以由 MAE 和均方误差 MSE 的乘积 (Product of MAE and MSE, MM) 表示, 一个简单的同时考虑硬件开销和误差的 $cost$ 定义方法是将其设置为 PDA 和 MM 的乘积。然而, 根据分析, 近似乘法器的 MAE 和 MSE 值随着 PDA 的降低呈指数级增长, 这意味着若将 $cost$ 设置为 $PDA \times MM$, 则 $cost$ 将由 MM 主导。为了使 $cost$ 真实地反映不同乘法器的质量, 使具有较小 $cost$ 的乘法器位于帕累托前沿, 将 $cost$ 设置为 PDAE, 其定义为:

$$PDAE = PDA \times \log_2(MM') \quad (4.4)$$

$$MM' = MAE \times MSE + 1 \quad (4.5)$$

根据式(2.42)、(2.44)、式(4.4)、式(4.5)可知精确乘法器的 $cost = PDAE = 0$ 。

4.4.6 优化流程

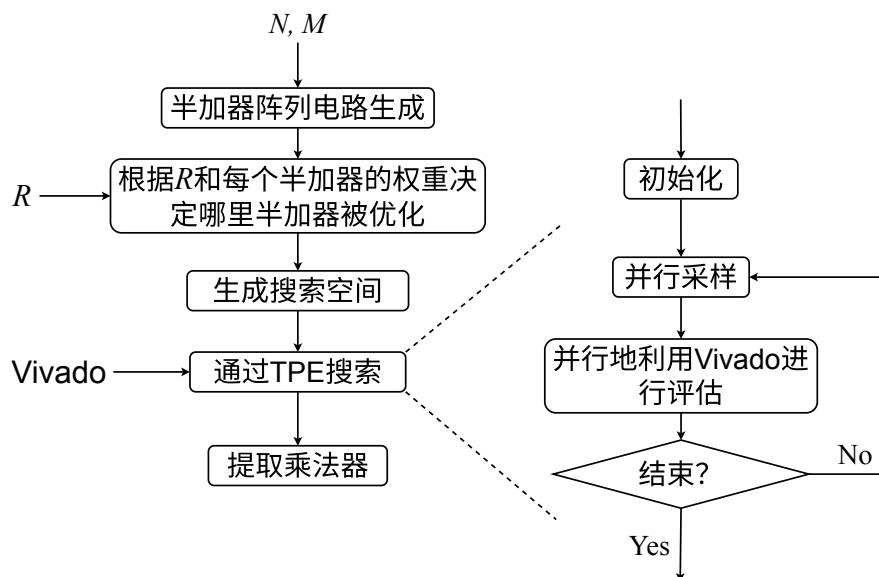


图 4-6 整体流程

图4-6展示了整体优化流程, 首先根据乘法器的位宽 N 和 M 生成精确模式下的半加器阵列电路并确定每个半加器的权重, 接着根据用户想要的面积减少比例 R 和每个半加器的权重决定哪些半加器被优化, 然后针对选定的半加器基于提出的 4 种简化方法生成优化空间, 并利用 TPE 并行地进行搜索, 在搜索时直接利用 Vivado 对生成的乘法器进行综合、布局、布线后硬件开销的评估, 能够

最大程度地考虑布线结果对乘法器质量的影响，注意决定TPE搜索方向的 $cost$ 计算方法如式(4.4)所示。达到最大迭代次数或时间要求后中止程序，提取 $cost$ 最小的多个乘法器与国际前沿工作进行对比。

4.5 实验结果

为了全面评估所提出的自动化方法，考虑无符号的 8×8 乘法器并将 R 设置为多个值： $R \in \{0.3, 0.4, 0.5, 0.6, 0.7\}$ ，对 R 的每个取值在60核Intel Xeon服务器上运行48小时之后提取 $cost$ 较小的一批乘法器合在一起与已有的工作进行比较。对比的乘法器除了3.6提到的面向ASIC设计的近似乘法器之外，也包括一个FPGA近似乘法器库ApproxFPGAs^[139]和通过手动修改LUT编码的方法生成的FPT22^[133]、CaCc^[132]、SMAproxLib^[131]、TCAD22^[134]，总共1167个近似乘法器。除此，两个分别由Xilinx LogiCORE IP^[135]构建的和采用华莱士树结构的精确乘法器（分别被命名为Xilinx Default IP和Wallace）也被考虑进行对比。

所有的乘法器均通过Synopsys VCS S-2021.09-SP2和Vivado 2023.1基于Xilinx Virtex UltraScale+系列的器件xcvu3p-ffvc1517-3-e进行仿真、综合、布局布线，提取数据后计算 MM' 和PDA。

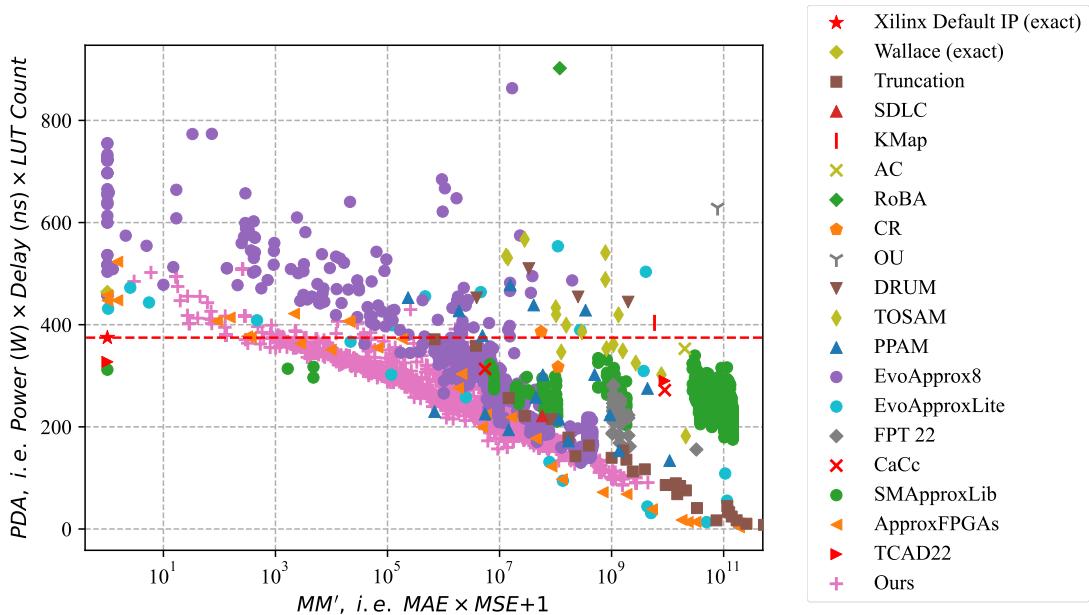


图4-7 不同乘法器的PDA和 MM' 对比图

图4-7展示了不同乘法器的PDA和 MM' 散点图，可以看到基于本文生成的乘法器与别的乘法器一起组成了帕累托前沿。SMAproxLib和TCAD22中存在两个乘法器是精确乘法器，硬件开销比Xilinx LogiCORE IP实现的乘法器更低。具体来讲，这两个乘法器消耗的LUT资源比Xilinx IP少20%左右，关键路径延

迟增加了约 5%，这与文献^[131]和文献^[134]中描述地一致。SMAApproxLib 中有三个近似乘法器以较小的误差 ($MM' \in [10^3, 10^4]$) 实现了较高的硬件性能。同时，从图中可以看出，FPT22、CaCc、SMAApproxLib 和 TCAD22 这些基于 LUT 编码方式手工设计的近似乘法器之间误差相差较大，无法根据需求灵活地提供不同质量的解决方案，尤其是 SMAApproxLib。尽管 EvoApprox8b^[107]库是面向 ASIC 的，但基于自动化方法生成的近似乘法器之间具有较小的质量差异，能够为具有不同约束的各种应用程序提供更多选择。最后，当 $MM' \in [10^8, 10^9]$ 时，可以看到 EvoApproveLite^[121]和 ApprovxFPGA^[139]的乘法器比其他乘法器质量更高，但这些乘法器可能并不实用，因为它们要求应用对误差具有较大的容忍度。

从生成的乘法器中提取的具有小 PDAE 值的近似乘法器都在帕累拖前沿，这证明了 PDAE 作为 *cost* 指导贝叶斯搜索的有效性。

表 4-1 根据 MM' 对乘法器进行分组后每组的最佳 PDA 值比较

	$MM' \in [10^3, 10^7]$		$MM' \in [10^3, 10^8]$		$MM' \in [10^4, 10^7]$		$MM' \in [10^4, 10^8]$	
<i>Group</i>	<i>Best PDAE</i>	<i>Imp. (%)</i>						
Truncation	7205.03	50.90	5488.09	35.53	7205.03	49.60	5488.09	33.83
SDLC ^[98]								
KMap ^[96]								
AC ^[118]								
RoBA ^[119]	5709.04	38.03	5709.04	38.03	5709.04	36.39	5709.04	36.39
CR ^[97]								
OU ^[100]								
DRUM ^[99]	9909.35	64.30	9909.35	64.30	9909.35	63.35	9909.35	63.35
TOSAM ^[91]	6247.27	43.37	6247.27	43.37	6247.27	41.87	6247.27	41.87
PPAM ^[120]	4461.53	20.70	4461.53	20.70	4461.53	18.61	4461.53	18.61
EvoApprox8b ^[107]	4857.81	27.17	4340.22	18.48	4857.81	25.25	4340.22	16.33
EvoApproveLite ^[121]	5088.03	30.46	3442.06	-2.79	5088.03	28.63	3442.06	-5.50
FPT22 ^[133]	5010.96	29.40	5010.96	29.40	5010.96	27.53	5010.96	27.53
CaCc ^[132]	7017.36	49.58	7017.36	49.58	7017.36	48.25	7017.36	48.25
SMAApproxLib ^[131]	3356.49	-5.41	3356.49	-5.41	6266.03	42.05	5801.66	37.41
ApprovxFPGAs ^[139]	4151.85	14.79	3220.87	-9.85	4429.49	18.02	3220.87	-12.74
TCAD22 ^[134]	9584.22	63.09	9584.22	63.09	9584.22	62.11	9584.22	62.11
Ours	3537.97	/	3537.97	/	3631.35	/	3631.35	/
Avg. Imp.	/	35.53	/	28.70	/	38.47	30.62	/

为了更直观地比较不同乘法器的质量，将不同乘法器按照 MM' 分组，并挑出每组中最佳的 PDAE 值进行对比。表4-1显示了分组后每组最佳的 PDAE 值，按照本文的方法生成的近似乘法器的 PDAE 比别的组平均提高了 28.70%-38.47%。如果将 MM' 限制在 $[10^4, 10^7]$ 的范围内，本文生成的乘法器在所有乘法器中质量最高。

4.6 本章小节

本章提出了一种面向 FPGA 的自动化近似乘法器设计方法，该方法假设乘法器的部分积在累加前存在一次由半加器阵列进行的压缩操作，利用贝叶斯算法提出的 4 种半加器简化方法对半加器阵列进行优化，同时保留压缩后累加过程中部分积的粗粒度加法，以使 FPGA 的 EDA 工具对其进行有效地识别，映射到进位链，提高电路性能。与国际前沿工作的 1167 个近似乘法器相比，基于本章的方法生成的乘法器能够形成 Pareto 前沿，比已有的乘法器平均提高了 28.70%-38.47%。尽管实验是基于 FPGA 和均匀分布进行的，但该方法可以很容易地拓展到 ASIC 领域和非均匀分布。

第 5 章 基于乘法器库的近似逻辑综合

广义的逻辑综合可分为传统逻辑综合、精确逻辑综合（Exact logic synthesis）和近似逻辑综合三类。其中，传统逻辑综合（通常简称为逻辑综合）在整个芯片设计流程中处于最上游的位置，是指将数字电路的高抽象级描述（通常是指寄存器传输级），在功能一致的前提下经过布尔函数化简、优化后，转换到的逻辑门级别的网表的过程，其性能对芯片最终的面积和延迟起决定性作用。精确逻辑综合（可简称为精确综合）要求在给定的约束下找到电路的最佳实现，比如基于给定的门的种类和数量上限对布尔网络进行映射^[140]，属于传统逻辑综合中的一个细分研究方向。近似逻辑综合包括两个方向，一个方向侧重于近似算术单元的生成，也被叫做近似电路综合；另一个方向更靠近传统的逻辑综合，即基于已有的近似库对大型设计进行优化和映射。本文着重于传统逻辑综合中大规模电路的序列探索，以及和近似乘法器库结合后针对 DNN 加速器的近似逻辑综合研究。

5.1 基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架

5.1.1 研究背景

Yosys 和 ABC

图5-1展示了传统逻辑综合的流程图，首先将用户设计的寄存器传输级电路读入并解析，然后进行一系列工艺无关的逻辑优化操作，最后进行工艺相关的映射，生成门级网表。在现代 EDA 工具中，解析后的电路的组合逻辑部分通常由一个有向无环图进行表示，被称为布尔网络^[141]，其中节点代表逻辑函数，边代表连接关系，之后的一系列的优化及映射操作都基于该图进行。在一个布尔网络中^[142-144]，一个节点的扇入（Fanin）和扇出（Fanout）分别指该节点的输入和输出节点；假设存在一条路径从节点 v 到节点 w ，则 v 是 w 的传递扇入（Transitive fanin）， w 是 v 的传递扇出（Transitive fanout）；网络的主要输入（Primary Inputs, PIs）指所有的无扇入节点，网络的主要输出（Primary Outputs, POs）指所有与外部相连的节点；一条路径的长度指经过的节点数目；节点的深度或级数指从网络的所有主要输入到该节点的所有路径中最长路径的长度；最大节点深度被



图 5-1 传统逻辑综合流程图



图 5-2 函数 $x_2(x_1 + x_3)$ 的两种不同的 AIG 实现

称为网络的深度。

AIG (And-Inverter Graph) 是目前被广泛用来对逻辑函数进行表示和优化的一种有向无环图^[145]，在 AIG 中，节点分为输入节点、输出节点和 2 输入的与门节点三种类型，边包括取反和不取反两种情况，输入节点没有输入边，输出节点可能有输出边。一个逻辑函数可由不同结构的 AIG 表示，图5-2展示了函数 $x_2(x_1 + x_3)$ 的两种不同的 AIG 实现。基于 AIG，由 Berkeley 大学研发的逻辑综合与验证工具 ABC^[65]在学术界和工业界取得了广泛的关注和赞誉。ABC 可以对 AIG 进行逻辑优化和工艺映射，逻辑优化的目标是最小化 AIG 的规模和深度，映射目标是最小化 LUT 或 ASIC 电路的面积和延迟。

在逻辑优化阶段，ABC 存在着许多不同的命令对 AIG 进行变换和化简，典型的命令包括 balance, refactor, rewrite 等，不同的优化命令会对 AIG 产生不同的优化效果，开发人员根据经验预先设定了一些优化脚本（如 resyn2）对不同的电路进行使用。然而，不同命令的组合及顺序对逻辑优化的效果影响很大，并且当优化过程进行到一定程度后，继续优化并不一定会对后续的映射操作起到正面效果^[146]，因此必须针对不同的电路提供不同的优化序列，以达到对不同电路都能实现良好提升的目的。为给定的布尔逻辑网络寻找较优逻辑优化命令组合的问题被称为序列探索。

ABC 能够利用 AIG 对布尔网络进行有效地表示及转换，但在电路解析方面能力不足。目前使用最广泛的硬件描述语言（Hardware description language）是 Verilog-2005，被大多数 EDA 工具支持。Yosys^[147]是一款开源的 Verilog 解析工具，支持绝大部分 Verilog-2005 的语法，与 ABC 结合能够完成从 Verilog 到门级网表的全流程工作。

用来表示布尔网络的不同 DAG 形式

(1) XAIG

在 AIG 中，一个 2 输入异或门至少需要 3 个节点才能表示，这导致 AIG 无法对异或密集型的电路进行高效地表示和优化，因此有工作提出了 XAIG (Xor-And-Inverter Graph)，在 AIG 中引入 2 输入的 XOR 节点，提高异或操作的表达效率^[148-150]。图5-3展示了函数 $\overline{x_1 + x_2} \oplus x_3 \cdot \overline{x_4}$ 的 AIG 和 XAIG，其中圆圈代表 AND 节点，六边形代表 XOR 节点，虚线边代表取反操作，可以看到 XOR 节点的引入能够降低图的规模。XAIG 也被称为 XAG (Xor-And Graph)。

(2) MIG

与 XAIG 的提出类似，有工作发现 AIG 对控制电路来说是一种高效的表示电路的 DAG 形式，但对算术电路效率较低，于是提出了适用于对算术电路进行表达和优化的 MIG (Majority-Inverter Graph)^[151]。在 MIG 中，除了输入节点外，



图 5-3 函数 $\overline{x_1 + x_2} \oplus x_3 \cdot \overline{x_4}$ 的 AIG 和 XAIG，圆圈代表 AND 节点，六边形代表 XOR 节点，虚线边代表取反操作

每个节点表示一个 3 输入的 Majority 门，用符号 $\langle \rangle$ 表示，定义为：

$$\langle xyz \rangle = xy + xz + yz = (x+y)(x+z)(y+z) \quad (5.1)$$

(3) XMG

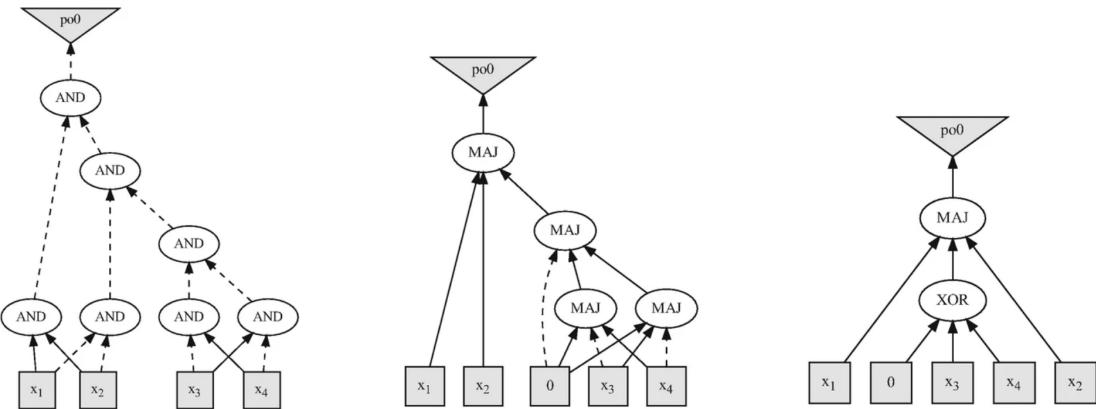


图 5-4 函数 $\langle x_1, x_2, (x_3 \oplus x_4) \rangle$ 分别在不同 DAG 中的表示，从左到右依次为：AIG、MIG、XMG，虚线代表取反操作

对 MIG 引入 3 输入的 XOR 节点，能够提高某些函数的表达效率，对应的 DAG 被称为 XMG (Xor-Majority Graph)^[152]。图5-4展示了函数 $\langle x_1, x_2, (x_3 \oplus x_4) \rangle$ 分别在 AIG、MIG 和 XMG 中的表示，虚线代表取反操作，可以看到 XMG 使用的节点数目最少^[153]。

MFFC

在一个布尔网络中，节点 v 的一个锥 (Cone) C_v 是指 v 和 v 的传递扇入节点的集合（不包括网络的主要输入），锥中任意节点到 v 的路径都在锥内， v 被

称为锥的根节点，易知 v 可以有多个锥^[142-143]。扇入节点数量小于等于 K 的锥被称为 K 可行锥 (K -feasible cone)^[154]。

节点 v 在锥 C_v 下的一个割 (Cut) 是 C_v 的一个划分 (X, \bar{X}) ，其中 \bar{X} 是 v 的一个锥。当 \bar{X} 是一个 K 可行锥时，割 (X, \bar{X}) 被称为 K 可行割 (K -feasible cut)^[154]。 \bar{X} 的扇入节点集合 L 满足以下两个性质：

- 任意一条从输入到节点 v 的路径至少会经过 L 中的一个元素；
- 对于 L 中的任何一个节点 l ，至少存在一条从输入到 v 的路径经过 l 且不经过 L 中的其他节点。

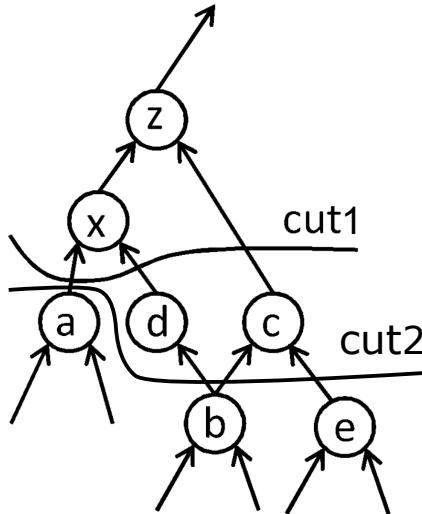


图 5-5 节点 z 的一个锥 $\{z, x, a, d, c, b, e\}$ 和两个割 cut1 与 cut2

图5-5展示了节点 z 的一个锥 $\{z, x, a, d, c, b, e\}$ 和两个割^[141]， cut1 和 cut2 均是 3 可行割。

若锥 C_v 内任意节点的扇出均在锥内，则称 C_v 为节点 v 的无扇出锥 (Fanout Free Cone, FFC)， v 的所有无扇出锥中最大的那个被称为 v 的最大无扇出锥 (Maximum Fanout Free Cone, MFFC)，记为 $MFFC_v$ ，易知 MFFC 有以下性质^[142-143,155]：

- 一个节点的 MFFC 有且只有一个；
- 若 $w \in MFFC_v$ ，则 $MFFC_w \subseteq MFFC_v$ ；
- 两个 MFFC 要么不相交，要么一个包含另一个；
- $MFFC_v$ 内节点的值只会影响到 v 和 v 的传递扇出。

图5-6展示了不同节点的 MFFC，以不同程度的阴影区域表示，可以看到满足上述性质。

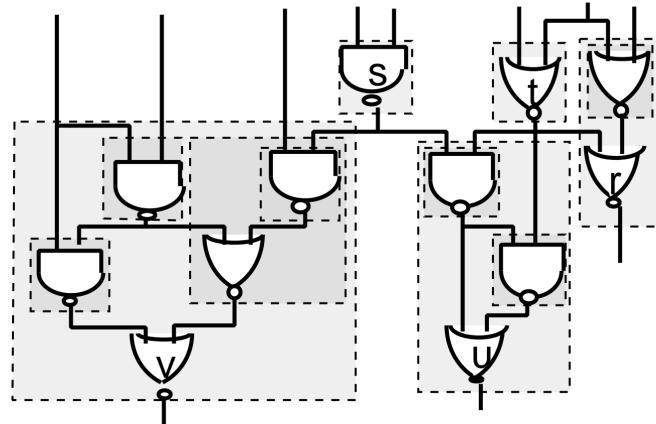


图 5-6 不同节点的最大无扇出锥

电路划分

当布尔网络的规模太大时，单个优化命令的运行时间变长，序列探索时一次迭代需要的时间显著增加，导致无法在一个可接受的时间内得到一个较优的命令组合，通过划分将大型网络分割成较小的子网络来并行地探索，能够大大减少运行时间。

(1) 超图划分

一个网络的 DAG 图既可以转换成普通图（一条边只连接两个顶点），也可以转换成超图（一条边可以连接超过两个顶点），考虑到电路中的输出往往都是多扇出的，超图能够更好地体现出连接性，因此将电路转换成超图进行划分是一个较好的选择^[156]。

Partition hypergraph $H = (V, E, c, \omega)$ into k disjoint blocks

$\Pi = \{V_1, \dots, V_k\}$ such that:

- blocks V_i are **roughly equal-sized**:

$$c(V_i) \leq (1 + \varepsilon) \lceil \frac{c(V)}{k} \rceil$$

imbalance parameter

- objective function is **minimized**:

cut-net:

$$\sum_{e \in \text{cut}} \omega(e) = 3$$

connectivity:

$$\sum_{e \in \text{cut}} (\lambda - 1) \omega(e) = 6$$

blocks connected by net e

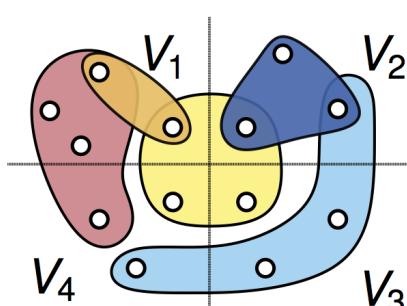


图 5-7 超图划分问题的定义

超图划分 (Hypergraph partitioning)^[157]能够将节点划分为 k 个大致相等的部

分，同时最小化基于边定义的目标函数，常见的目标函数有割边重要性和子图连通性，其中割边重要性是指被切割的超边的权重之和，而子图连通性则会同时考虑权重和连接的子图的数量。图5-7展示了超图划分问题的定义示意图。目前常见的超图划分算法是多层（Multilevel）方法，由三个阶段组成：(1) 粗化：对超图中连接紧密的点进行逐级合并以降低超图的规模；(2) 初始划分：粗化完成后得到一个大的超图并对其进行初始划分；(3) 细化：逐级划分并分解原先合并的点，每一级划分后使用局部搜索方法来调整边界点以最小化目标函数。

(2) 自然划分

通常来讲，EDA 工具对电路进行解析后生成布尔网络时会把寄存器与组合逻辑分开，将寄存器的输入变成组合逻辑的输出，寄存器的输出变成组合逻辑的输入。有工作发现，良好设计的电路中流水线（Pipeline）技术充分，能够将整个电路切分成比较均匀的组合逻辑块，转换成布尔网络后对应多个相互独立的 DAG。基于此发现，文献^[158]提出了一种“自然划分（Natural partitioning）”的方法，基于 ABC^[65]对 AIG 进行分析，将一个布尔网络分簇，且簇与簇之间没有连接，之后对所有的 AIG 簇并行地进行 LUT 映射，对 13 个大型电路的测试结果表明，映射速度平均提高了 5.76 倍，面积略微增加了 0.57%，延迟保持不变。

(3) 基于 MFFC 和带约束超图划分的有向无环划分

普通的超图划分并没有考虑 DAG 有向无环的特点，文献^[155]首先对一个网络进行遍历，利用 MFFC 缩小超图的规模，之后利用带约束的多层划分方法，提出了一个面向 FPGA 领域的有向无环划分方案，比普通的超图划分方法切割的边的数量更少、划分质量更高。

强化学习

强化学习是机器学习中的一个领域，强调一个智能体（Agent）如何基于环境（Environment）行动，以取得最大化的预期利益，是除了监督学习和非监督学习之外的第三种基本的机器学习方法。强化学习通过感知所处环境的状态（State）对动作（Action）的反应（Reward），来指导更好的动作，从而获得最大的收益（Return），以游戏为例，如果在游戏中采取某种策略可以取得较高的得分，那么就进一步“强化”这种策略，以期继续取得较好的结果。目前，强化学习在某些领域已经证明达到了人类水平，甚至优于人类，比如，由谷歌研发的电脑围棋软件 AlphaGo^[26]击败了韩国围棋冠军李世石。

强化学习基于马尔科夫决策过程（Markov decision process），即当前状态包含了对未来预测所需要的有用信息，过去信息对未来预测不重要，关注点在于“探索未知领域”和“利用已有知识”的平衡，其算法分为有模型学习（Model-Based）和免模型学习（Model-Free）两类，其中免模型学习更容易实现，迁移性也更好，

得到了广泛的研究。

5.1.2 国内外研究现状

序列探索

学术界提出了多种方法用来对逻辑综合中的不同命令的组合进行探索，包括基于强化学习的方法、利用贝叶斯优化进行搜索、以及基于图神经网络进行预测等方法，下面分别进行介绍。

(1) 基于强化学习的方法

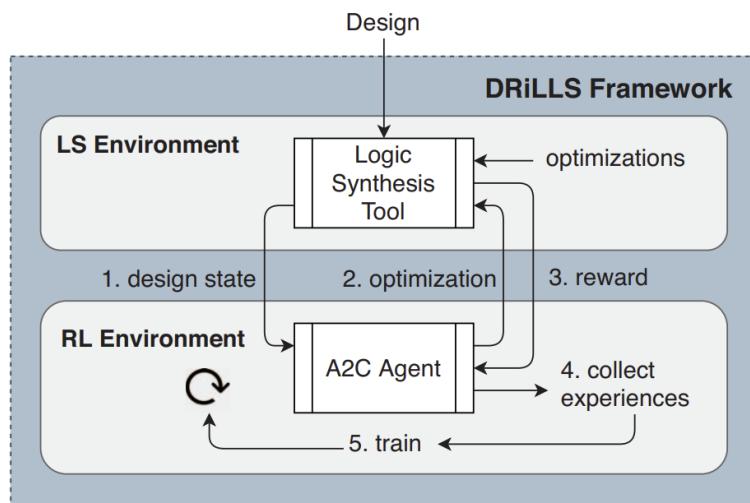


图 5-8 基于强化学习的 DRiLLS 序列优化方法架构图

假设 $\mathbb{A} = \{a_1, a_2, a_3, \dots, a_n\}$ 代表 n 个相互之间没有依赖的优化命令集合， k 表示优化序列长度，则可能的命令组合情况一共有 n^k 种。文献^[159]提出了一种基于强化学习的序列优化方法 DRiLLS，DRiLLS 利用 A2C（Advantage Actor Critic）代理来探索序列空间。图5-8给出了 DRiLLS 的架构图，其中逻辑综合环境由 Yosys 和 ABC 实现，强化学习环境由 A2C 作为代理，与综合环境进行交互学习。DRiLLS 将强化学习中的状态 *state* 定义为 ABC 中的 AIG 信息，包括输入输出数量、节点数、级数、寄存器数量、AIG 中边的条数、以及 AIG 中反向边的数量。同时，DRiLLS 中强化学习的奖励函数是一个同时考虑面积和延迟的多目标函数，对于满足延迟约束并且面积减少的优化序列，奖励最高（用+++表示）；对于不满足延迟约束并且面积增加的优化序列，奖励最低（用---表示）。具体的奖励标准如表5-1所示。

表5-2展示了基于 EPFL 电路集^[160-161]通过贪婪算法（Greedy algorithm）、手工设计的优化序列^[162]（Expert-crafted）、已有的最佳记录、以及 DRiLLS 利用 ABC^[65]在开源的 7nm 工艺库^[122]上得到的实验结果，其中 DRiLLS 的延迟约束

表 5-1 DRiLLS 中不同效果的优化序列对应的奖励情况

				Optimizing (Area)		
				Decr.	None	Incr.
Constraint (Delay)	Met			+++	0	-
	Not Met	Decr.		+++	++	+
		None		++	0	--
		Incr.		-	--	---

表 5-2 实验结果

Benchmark	Delay Constr. (ns)	Initial Design		Greedy			Expert-crafted			EPFL Best Size			DRiLLS		
		Area (um ²)	Delay (ns)	Area (um ²)	Delay (ns)	Impr. (%)									
Adder	2.00	867	2.02	1011	4.10	-16%	1772	1.82	-104%	1690	1.87	-94%	823	1.97	5%
B. Shifter	0.80	2499	1.03	2935	0.66	-17%	1534	0.77	38%	1040	0.77	58%	1400	0.77	43%
Divisor	75.00	12388	75.83	22439	79.14	-81%	21167	65.05	-70%	16031	74.91	-29%	13441	67.61	-8%
Hypotenuse	1000.00	176938	1774.32	236271	563.12	-33%	210828	525.34	-19%	169468	1503.88	4%	154227	995.95	12%
Log2	7.50	19633	7.63	30893	6.96	-57%	18451	7.45	6%	23999	10.12	-22%	17687	7.44	9%
Max	4.00	1427	4.48	3082	3.79	-115%	1440	3.93	-0.88%	1713	4.84	-20%	1037	3.76	27%
Multiplier	4.00	19617	3.83	25219	4.38	-28%	21094	3.70	-7%	19940	5.27	-1%	17797	3.96	9%
Sin	3.80	3893	3.65	5501	2.88	-41%	4421	2.19	-13%	4892	4.14	-25%	3050	3.76	21%
Square-root	170.00	11719	329.46	19233	93.71	-64%	16594	92.30	-41%	9934	169.46	15%	9002	167.47	23%
Square	2.20	11157	2.27	19776	3.96	-77%	16373	1.59	-46%	16838	4.06	-50%	12584	2.199	-12%
Avg. Area Imprv.		0.00%		-53.31%			-26.00%			-16.69%			13.19%		
Constraint Met		2/10		4/10			9/10			4/10			10/10		

是初始的未优化的电路直接映射得到的延迟。可以看出，DRiLLS 在不同电路上均满足了延迟约束，面积平均提高了 13.19%，这显示了 DRiLLS 的多目标奖励函数的有效性。

(2) 利用贝叶斯优化搜索

逻辑优化中的序列探索问题可以归结为黑盒优化问题，文献^[163]提出了 BOiLS，利用贝叶斯优化对优化命令的组合进行高效地探索。基于 ABC，对于一个给定的 AIG，BOiLS 的目标是在 n 个优化命令中找到一个长度为 K 的序列来优化电路，优化序列的好坏通过 ABC 中的 $if - K6$ 进行 LUT 映射后来评估，并用下面的公式进行计算：

$$QoR = \frac{Area(seq)}{Area(ref)} + \frac{Delay(seq)}{Delay(ref)} \quad (5.2)$$

其中 $Area(ref)$ 和 $Delay(ref)$ 分别代表对原始电路用 resyn2 进行优化后并进行 LUT 映射后的 LUT 个数和级数， $Area(seq)$ 和 $Delay(seq)$ 分别代表 BOiLS 找到的优化序列对原始电路进行优化后并映射后的 LUT 个数和级数。在 BOiLS 中，考虑的优化命令包括：rewrite, rewrite -z, refactor, refactor -z, resub, resub -z, balance, fraig, sopb, blut, dsdb，序列大小 $K=20$ 。

BOiLS 基于 EPFL 电路集^[160-161]进行测试，并对比了 6 个前沿工作，包括：基

	DRiLLS (PPO)	DRiLLS (A2C)	Graph-RL	GA	RS	Greedy	SBO	BOiLS	EPFL best (lvl)	EPFL best (count)
Adder	22.62	24.59	24.48	24.80	24.27	23.36	25.02	25.57	21.36	-55.76
Barrel Shifter	00.00	00.00								
Divisor	40.40	42.66	-	44.82	43.78	40.46	45.49	47.36	-59.52	14.04
Hypotenuse	00.81	00.89	-	01.66	01.75	-0.04	01.77	5.99	-68.80	01.62
Log2	07.02	07.48	-	07.96	07.77	04.70	09.01	08.70	06.25	-33.34
Max	29.28	30.49	31.51	31.97	30.76	28.14	31.04	31.77	35.61	-164.0
Multiplication	18.56	19.15	-	20.20	19.25	18.32	20.33	21.13	20.67	00.00
Sine	01.64	02.18	01.64	02.70	01.88	00.79	02.64	03.82	-23426.71	-26.21
Square-root	12.47	14.07	13.23	13.06	13.70	08.19	13.79	14.10	00.00	11.14
Square	36.65	37.77	37.88	38.01	37.78	36.56	38.27	38.90	38.88	-21.81
Average	16.94	17.93	-	18.52	18.09	16.05	18.77	19.74	-2343	-29.66

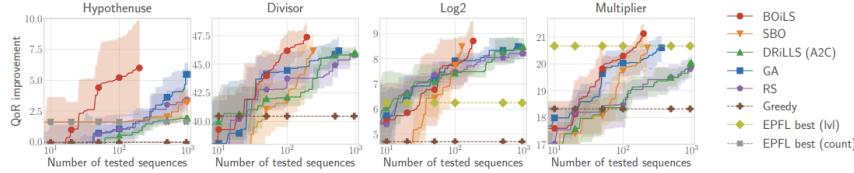


图 5-9 不同序列探索方法在迭代 200 次后的 LUT 映射结果

于强化学习方法的 DRiLLS^[159]，标准贝叶斯优化 (Standard Bayesian Optimization, SBO)，遗传算法 (Genetic Algorithm, GA)，随机搜索 (Random Search, RS)，以及公开的最佳记录。每种方法限制的迭代次数为 200，图5-9展示了不同方法的实验结果。可以看到，BOiLS 在 8/10 个电路中都取得了最优结果，SBO 在 log2 电路中取得了最优结果，BOiLS 紧随其后，显示了贝叶斯优化在逻辑综合序列探索中的巨大潜力。另外有趣的一点是，结果显示，基于强化学习的序列搜索方法看起来似乎并没有比随机搜索强多少。

(3) 基于图神经网络的方法

图卷积网络 (Graph Convolutional Network, GCN) 是一种用于处理图数据的深度学习模型，其目标是将传统的卷积神经网络和递归神经网络等深度学习方法扩展到图结构数据如社交网络、推荐系统、生物信息学等领域的数据处理，填补深度学习领域对图数据处理方式的空缺。

文献^[164]首先利用 GCN 基于 AIG 提出了一个序列质量预测器，总体结构如图5-10所示。预测器的输入包括 AIG 和优化命令序列 (图中的 Synthesis recipe)，两者分别通过 AIG 嵌入网络和序列嵌入网络进行嵌入，拼接后经过四层全连接层进行输出。之后，在基于小型电路集上对预测器进行训练后，面对新的大型电路，通过主动学习 (Active learning) 的方式对预测器进行微调，以提高对新电路的预测准确率。最后，基于微调后的预测器，利用模拟退火算法对新电路生成一个高质量的优化序列。整体框架命名为 Bulls-Eye，流程如图5-11所示。Bulls-Eye 的预测器在 44 个开源的电路设计构成的数据集上进行训练，实验证明，训练后的模型在新的大型电路的序列质量预测任务中表现良好，微调后的模型准确率进一步上升。



图 5-10 基于 AIG 和 GCN 的序列质量预测器

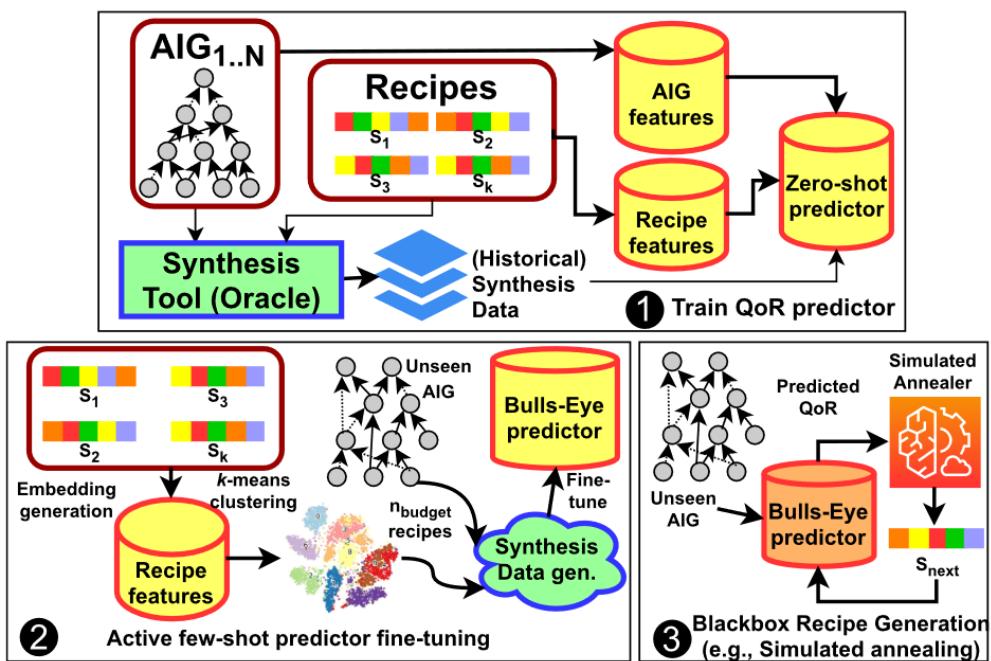


图 5-11 Bullseye 整体框架图

多种 DAG 联合优化

文献^[156]基于 MIG 在对算术电路的综合及映射中实现了比 AIG 更好的效果这一发现，提出了 LSOOracle，是第一个同时采用多种 DAG 形式对布尔逻辑电路进行表示和优化的异构逻辑综合框架。

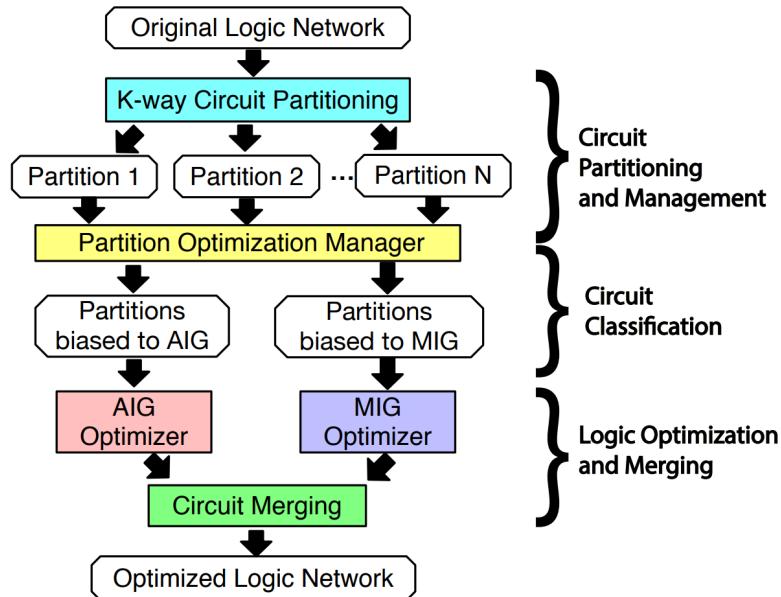


图 5-12 LSOOracle 流程图

图5-12展示了 LSOOracle 的工作流程图，输入电路首先被转换成 AIG，然后变成超图，超图由点和超边组成，一条超边可能会连接多个顶点。接着利用开源的超图划分工具 KaHyPar^[157]将超图分割成多个子图，子图之前存在松散的连接。之后，分类引擎会利用二维图片对子图进行表示，并利用神经网络对图片进行分类，对每个子图挑选一种最适合的 DAG 表示（AIG 或 MIG），并分别优化。具体来讲，设 $B = \{0, 1\}$ ，一个 n 输入的布尔函数是一个从 B^n 到 B 的映射： $f : B^n \rightarrow B$ 。因此，一个 n 输入的布尔函数可以由一个 n 维空间表示，卡诺图是一种利用二维空间完成 n 维空间映射的表示形式。受卡诺图启发，LSOracle 提出了一种用二维网格来表示逻辑函数的方法，命名为 KMImage (Karnaugh-Map Image)。图5-13展示了一个将卡诺图转换为 KMImage 的例子，在 KMImage 中，每个网格表示图片的一个像素点，卡诺图中函数值为 1 的网格在 KMImage 中以灰色像素表示，卡诺图中函数值为 0 的网格在 KMImage 中以白色表示黑色像素表示。这种表示方法类似于 MNIST^[113] 数据集中的图片表示方法。在 MNIST 中，每张图片包含 28×28 个像素点，由黑白两种颜色组成。通过将布尔函数转为二维图片，能够利用计算机视觉领域成熟的分类方法对 KMImage 的特征进行识别并分类，挑选出最适合布尔函数实现的 DAG 格式。一个 $N \times N$ 大小的 KMImage 可



图 5-13 将卡诺图转变为 KMImage 的示例

以表示任意一个输入数为 $2\log_2 N$ 的逻辑函数，除非逻辑函数是对称的，否则同一布尔函数在改变输入顺序后就会产生不同的 KMImage，有可能导致分类结果不一致。然而，一个逻辑函数只会对应一个最优的 DAG 实现，分类错误会导致电路性能的下降，这是 LSOOracle 考虑不周的地方。超图划分后的子图往往是多输出函数，KMImage 只能表示单输出函数，为了解决多输出函数子图的表示问题，LSOracle 采用了一种计分机制。具体来说，分类引擎基于子图和所有的输出节点得到多个 KMImage 并赋予不同的权重，基本原则是对那些拥有更多节点数目和更大节点深度的网络对应的 KMImage 给予更大的权重。在对每个 KMImage 进行分类之后，分别计算不同 DAG 的总得分，得分最高的 DAG 实现作为该子图的最终分类结果，计算公式如下所示：

$$score = \sum_{i=1}^m (W_{ni} * N_i + W_{di} * D_i) \quad (5.3)$$

式中 m 是子图的输出数，也是 KMImage 的个数； N_i 和 W_{ni} 分别是第 i 个 KMImage 对应的逻辑网络的节点数和节点数权重； D_i 和 W_{di} 分别是第 i 个 KMImage 对应的逻辑网络的深度和深度权重。LSOracle 的分类器采用大小为 256×256 的固定尺寸的 KMImage，可以表示不超过 16 个输入的逻辑函数。对于小于 16 个输入的逻辑函数，通过随机填充的方法将 KMImage 扩展到 256×256 大小。对于超过 16 个输入的逻辑函数，直接通过启发式算法对其进行分类：如果逻辑函数的逻辑深度超过所在子图逻辑深度的 40%，那么该逻辑函数则被分类到 MIG，否则被分类到 AIG，之后根据式(5.3)计算得分，最终得到该子图的分类结果。一旦一个子图分类完成，该子图立刻被转化成对应的 DAG 格式，由预先定好的优化脚本进行优化。当所有的子图优化完成后，合并所有优化后的子图并将整个电路转换成 MIG，进行最后的优化后输出。

LSOracle 的分类算法具有一定的局限性：(1) KMImage 具有固定尺寸，只

能表示不大于 16 个输入数的逻辑函数，对于小于 16 个输入数的布尔函数采用“随机补全”办法，这势必会引入不确定性，影响分类精度；(2) 划分后的子图往往是多输出的，然而 KMImage 只能表示单输出函数，无法直接表示多输出函数，当采用多个 KMImage 对子图进行表征时，多个 KMImage 对应的布尔网络之间存在重复的节点，这也会带来重复的计算，影响分类精度。解决这两个问题可以采用图神经网络 (Graph Neural Network, GNN) 对子图进行分类，GNN 不受电路规模和输出个数的影响，能够极大地提高分类效率。

5.1.3 研究动机

目前已有的序列优化方法^[159,163]在针对大规模电路进行探索时迭代速度慢，无法在有限地时间内得到高质量的输出结果，尽管有工作^[164]基于图神经网络在训练一个预测器后利用主动学习的方法能够在较短时间内应用到大网络上，但预测质量往往不如直接探索的方法，同时预测器的搜索空间是预先固定的，无法进行灵活调整。同时，这些方法都是将一个优化序列应用于整个网络，实际上网络的不同部分可能适用于不同的序列，比如电路的算术逻辑部分和控制逻辑部分有不同的特性^[151]。

文献^[156]利用超图划分的方法对大型的布尔网络进行分割，并对划分后的子网络选择适合的 DAG 进行优化，但由于划分后每个子网络的输入信号的到达时间可能不为 0，这导致在对子网络的关键路径进行优化时可能并不是真实的关键路径，同时超图划分的目标只是最小化割边权重和或子图连通性，并没有考虑节点处于不同子图可能会带来迥异的优化效果，最后不同 DAG 的优化引擎质量不一致，这些原因加起来导致合并后的网表经过评估后优势不大。因此需要针对大规模网络提出一个采用高质量划分的序列探索方法，既能充分利用现代 CPU 架构的多核特性提高效率，又能保证合并后优化质量的提升。

5.1.4 研究内容与创新点

本文提出并开源了一个基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架，该框架基于 Yosys^[147]和 ABC^[65]实现。首先利用 Yosys 对电路进行读入和解析，接着将电路中的组合逻辑提取出来。基于文献^[158]的发现，利用 ABC 将提取的组合逻辑转换成 AIG 进行分析和“自然划分”，得到多个子 AIG，子 AIG 之间没有任何连接。对“自然划分”后仍然较大的子 AIG，利用 MFFC 遍历将子 AIG 转换成以 MFFC 为节点的超图。之后利用开源的超图划分工具 KaHyPar^[157]采用多层划分的方法对 MFFC 超图进行划分，同时得到对应的子 AIG 的划分。划分完成后，将所有的子 AIG 转换成电路，对所有的子电路并行地利用提出的强化学习序列优化方法在给定的时间内进行搜索。最后用找到的



图 5-14 本文提出的逻辑优化框架流程图

高质量序列对子电路进行优化，优化完成后合并所有的子电路，和时序逻辑一起输出，并由商业综合工具评估优化结果，整体流程如图3-14(c)所示。主要创新点如下：

- 提出了一个开源的应用于大型电路的端到端强化学习逻辑优化框架，能够在对布尔网络进行分割后并行地利用强化学习方法进行序列探索，充分利用现代 CPU 的多核特性，提高探索效率；
- 与不分割相比，划分后能够对大网络的不同区域进行充分探索，避免了基于整个网络进行优化带来的局限性；
- 基于超过 150 个电路的商业综合工具的评估结果显示，与 ABC 的脚本 resyn2 相比，面积延迟积平均提高了 5.17%。

5.1.5 研究方法

(1) 自然划分

本文首先利用 Yosys 在对 Verilog 进行解析后，将电路中的组合逻辑和时序逻辑分开，其中，寄存器的输入会变成组合逻辑的输出，寄存器的输出会变成组合逻辑的输入。之后基于文献^[158]的发现，将提取出的组合逻辑通过 ABC 读入，转换成 AIG，并识别出 AIG 中所有互相之间没有连接的逻辑簇，按照簇将 AIG 进行分割，易知切割的边数为 0，此时分割得到的所有子 AIG 的主要输入要么来源于电路的输入、要么来源于寄存器的输出，因此输入延迟均为 0。

(2) 基于 MFFC 的超图划分

自然划分后，有可能会存在子 AIG 仍然过大的情况，这时对于过大的子 AIG，首先对所有的节点遍历并计算 MFFC，根据5.1.1的内容可知两个 MFFC 要么不相交、要么一个包含另一个，同时 MFFC 内的值只会影响到根节点和根节点的

传递扇出，因此每个 MFFC 要尽可能大，相当于在划分前将一些应该一起优化的 AIG 节点捆绑在一起，减少文献^[156]中直接对 AIG 超图划分带来的缺点。

之后对由 MFFC 为节点构成的超图采用开源的超图划分工具^[157]进行分割，为了保证分割后每个子图规模大致相等，MFFC 超图中每个节点都有一个权重值，大小为该 MFFC 对应的 AIG 的节点数。

(3) 提出的强化学习方法

DRiLLS^[159]在序列探索过程中每次运行一个优化命令都需要保存中间文件，效率较低。本文提出并开源了一个新的强化学习逻辑综合方法，该方法基于 OpenAI Gym^[165]和 Stable Baselines3^[166]实现，支持 ABC^[65]、Cirkit^[167]、iMAP^[168]三个学术界主流的逻辑综合工具，能够完成以逻辑优化、LUT 映射、ASIC 映射为目标的序列探索任务。

5.1.6 实验结果

对来自 OPDB^[169]、VTR^[170]、Koios^[171]、EPFL^[160-161]、IWLS05^[172]、QUIP^[173]等超过 150 个不同规模的电路进行广泛测试，对提取的组合逻辑网表分别用 ABC 的 resyn2、BOiLS^[163]、DRiLLS^[159]、LSOracle^[156]、以及本文提出的方法在 Intel Xeon 处理器利用 200 个 CPU 核进行优化，其中 BOiLS 和 DRiLLS 使用单线程运行，运行时间由下式确定：

$$\text{runtime} = \lceil \frac{N_p}{200} \rceil * 2h \quad (5.4)$$

其中 N_p 代表根据本文提出的方法在 MFFC 超图划分后的子图总数， $\lceil \rceil$ 表示向上取整。优化完成后和时序逻辑合并输出，利用 Synopsys Design Compiler (DC) S-2021.06-SP5 在一个开源的 7nm 工艺库^[122]进行综合，获得面积延迟积 (Area Delay Product, ADP) 进行比较。为了避免 DC 的优化影响结果，将电路的时钟频率约束设为 0，同时最小化面积 (set_max_area 0)，并使用“compile”命令进行映射。注意本文并没有考虑划分消耗的时间，对于一个拥有百万个 AIG 节点的组合逻辑网络来说，“自然划分”和 MFFC 超图划分所花费的时间总共在 10h 左右。

表5-3展示了以 ABC resyn2 为标准进行归一化的平均百分比提升，正数表示变差，负数表示改进，标红的数字代表每一行对应的指标中不同方法得到的最好结果。可以看到，LSOracle^[156]的效果最差，数据显示，与 ABC resyn2 相比，LSOracle 在最大的 10、20、30、40 个电路上面积和延迟的平均百分比没有任何提升，反而是变差的，这可能是因为尽管 LSOracle 使用了多种 DAG 对电路进行表示，但不同 DAG 的优化引擎效果不一致，导致网表合并后并没有明显的改进，同时不同 DAG 的节点实现成本不一样，比如 AIG 中每个节点代表一个 2 输

表 5-3 基于不同方法得到的面积、延迟和 ADP 的平均百分比提升

最大的 n 个电路		LSOracle ^[156]	BOiLS ^[163]	DRiLLS ^[159]	本文的方法
10	Area(%)	1.57	-10.69	-12.00	-11.13
	Delay(%)	12.10	0.69	-2.52	-0.23
	ADP(%)	14.35	-10.45	-14.11	-11.52
20	Area(%)	1.29	-5.80	-6.58	-6.55
	Delay(%)	13.61	-0.69	-1.74	-1.78
	ADP(%)	15.67	-6.72	-8.11	-8.37
30	Area(%)	5.03	-5.78	-6.19	-4.87
	Delay(%)	11.12	-0.08	0.09	-0.82
	ADP(%)	17.55	-5.95	-5.88	-5.73
40	Area(%)	4.10	-5.04	-5.47	-5.50
	Delay(%)	8.90	-0.55	-0.29	-1.49
	ADP(%)	14.08	-5.41	-5.46	-6.75
所有 (≥ 150)	Area(%)	2.08	-3.29	-3.61	-3.85
	Delay(%)	3.81	-0.64	-0.30	-1.70
	ADP(%)	6.16	-3.68	-3.73	-5.17

入与门，而 MIG 中每个节点代表一个 3 输入的 Majority 门，这导致 MIG 中节点的实现成本更高，无法简单通过比较同一电路在不同 DAG 实现下的节点数和级数来判断 DAG 映射后的好坏。与 ABC resyn2 相比，超过 150 个电路的测试结果显示，LSOracle 的面积、延迟、ADP 分别变差了 2.08%、3.81%、6.16%，且电路越大 LSOracle 的效果越差，比如最大的 10 个电路的结果显示，LSOracle 的 ADP 比 ABC resyn2 落后了 14.35%。BOiLS 和 DRiLLS 的结果是在没有划分的情况下直接对整个组合逻辑网表进行优化得到的，可以看到 DRiLLS 对大规模电路的探索效果更好，与 BOiLS 相比，DRiLLS 在最大的 10 个电路的面积、延迟、ADP 分别平均提升了 1.31%、3.21%、3.66%，且在最大的 20 个电路的表现也优于 BOiLS。从表5-3可以看出，本文提出的基于 MFFC 自适应超图划分的端到端强化学习逻辑优化方法在大电路的表现上不弱于 BOiLS，最大的 10 个电路的面积、延迟、ADP 相比 BOiLS 分别平均提升了 0.44%、0.92%、1.07%。基于全部的超过 150 个电路的实验结果显示，本文的方法效果最好，与 ABC resyn2、BOiLS、DRiLLS 相比 ADP 分别提升了 5.17%、1.49%、1.44%。

考虑到数据是基于大量的用例通过商业综合工具评估得到的，因此实验结果能够代表真实的应用场景，每个电路具体的数据见附录A。

5.2 基于近似乘法器库面向 DNN 加速器的近似逻辑综合

5.2.1 研究背景

即使生成的乘法器在经过评估后硬件开销较低，但基于该乘法器设计的DNN加速器可能硬件成本并不领先。例如，在图3-21中，PPAM(1,1)的PDA比XFYW和XWYF都要高，但在如图3-23(c)所示的基于不同乘法器的TASU^[125]实现中，基于PPAM(1,1)的设计比基于乘法器‘A’、‘B’、‘C’的PDA都要低，这是因为当近似乘法器与别的模块组合时，EDA工具对不同的电路能够进行不同的优化，因此有必要对基于不同乘法器的DNN加速器实现进行探究。

5.2.2 研究内容

研究方法

将提出的基于MFPC自适应超图划分的端到端强化学习逻辑优化框架与近似乘法器库结合，针对DNN硬件加速器，本文首先对图3-21中共68个XWYF近似乘法器通过DC在7nm工艺库^[122]上以2GHz时钟频率约束进行综合并获取PDA，找到位于帕累托前沿的乘法器；之后对基于不同乘法器的68个卷积操作加速单元SC^[126]利用提出的开源逻辑优化框架进行评估，获取ADP并比较结果。

实验结果

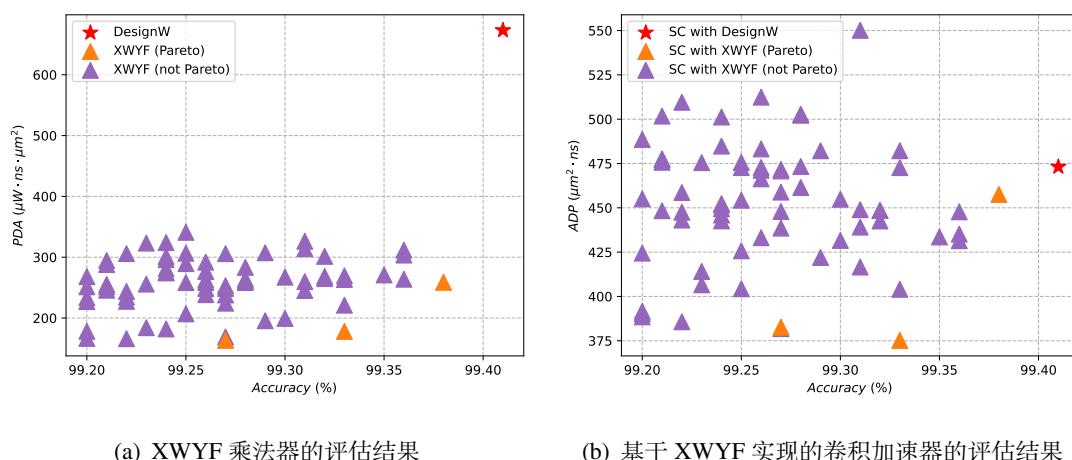


图5-15 XWYF乘法器的评估结果以及基于XWYF实现的卷积加速器的评估结果

图5-15展示了68个XWYF乘法器的评估结果以及基于XWYF实现的68个

卷积加速器 SC^[126]的评估结果，作为对比，DC 通过 DesignWare 库^[123]实现的精确乘法器 DesignW 也被纳入比较。图5-15(a)中共有 3 个乘法器位于帕累拖前沿，以黄色表示，对应的 SC 实现在图5-15(b)中也以黄色表示。一个有意思的现象是，虽然 DesignW 的单独评估结果显示其 PDA 较高，但基于 DesignW 实现的 SC 模块的硬件开销却优于相当一部分的 XWYF 乘法器，这说明了近似乘法器库在使用时不能简单地根据单独的硬件成本直接选择，而是应该进行多次尝试后决定。图5-15(b)表明，基于 3 个帕累拖最优的乘法器的 SC 实现仍然位于帕累拖前沿，因此当给定一个近似库进行综合时，可在库中位于帕累拖前沿的近似单元中进行一定次数的随机尝试，综合比较后选择最好的那个进行实现。

5.3 本章小节

本章首先介绍了基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架，该框架基于 Yosys 和 ABC 实现，首先利用 Yosys 对电路进行读入和解析，接着将电路中的组合逻辑提取出来并划分，划分完成后采用强化学习方法对所有的子电路并行地进行序列寻优，最后通过商业综合工具进行评估。基于超过 150 个电路的实验结果显示，与 ABC 的脚本 resyn2 相比，面积延迟积平均提高了 5.17%。之后将框架与近似乘法器库结合，对基于不同近似乘法器的 DNN 硬件加速器进行研究，结果显示乘法器和加速器之间的性能收益存在不匹配的现象，即基于硬件开销低的乘法器实现的加速器性能不一定高，因此近似库在使用时不能简单地根据乘法的硬件成本直接选择，而是应该进行多次尝试后决定。

第 6 章 总结与展望

6.1 总结

作为一种新兴的计算范式，近似计算允许系统在可接受的误差范围内返回结果，与容错应用结合，能够提高计算效率，降低芯片能耗。针对近似乘法器，本文提出并开源了三个研究工作：

- 面向 ASIC，基于数据分布和输入极性提出并开源了一种自动化近似乘法器设计方法，该方法利用逻辑操作和移位操作在部分积生成后、累加前对部分积进行一次压缩，降低部分积阵列的规模，减轻后续的累加压力。对提出的方法进行了广泛的测试评估，结果显示，基于均匀分布下 8 比特无符号数生成的近似乘法器与国际前沿工作相比取得显著进步，在平均误差距离 MAE 和硬件开销 PDA 两个指标均领先；同时，对提出的方法基于不同规模的 DNN 以及滤波器应用进行实验，结果显示生成的近似乘法器能够在几乎没有引起应用级精度下降的情况下将 PDA 降低 26.4%-27.1%；最后基于正态分布 32 比特无符号数的实验结果证明了方法对大位宽乘法器的有效性。
- 面向 FPGA，提出并开源了一个基于贝叶斯优化的自动化近似乘法器生成方法，该方法假设乘法器的部分积在生成后、累加前存在一次由半加器阵列进行的压缩操作，针对精确半加器提出了 4 种简化方案，利用贝叶斯算法对半加器阵列进行优化，同时保留压缩后累加过程中部分积的粗粒度加法。通过详细设计地能够同时反映误差和硬件开销的目标函数，与国际前沿工作中的 1167 个乘法器相比，生成的乘法器能够形成 Pareto 前沿，在硬件成本和误差的乘积上平均有 28.70%-38.47% 的改进。
- 提出并开源了一个基于 MFFC 自适应超图划分的端到端强化学习逻辑优化框架，首先利用 Yosys 对电路进行读入和解析，接着将电路中的组合逻辑提取出来，利用“自然划分”和 MFFC 超图划分将组合逻辑分割成多个子网表，对所有的子网表基于 ABC 并行地用提出的强化学习序列优化方法在给定的时间内进行探索，并由商业综合工具进行评估。基于超过 150 个电路的商业综合工具的评估结果显示，提出的方法处于国际前沿水平，与 ABC resyn2 相比，面积延迟积 ADP 平均提高了 5.17%。进一步与提出的近

似乘法器库结合，对 DNN 硬件加速器的近似实现进行了研究，结果显示近似乘法器的单独硬件成本与对应加速器的硬件成本存在一定偏差，因此在实际使用中可对库中的不同乘法器进行多次尝试，以实现更好的 PPA。

6.2 展望

本文提出的两种近似乘法器设计方法均基于部分积的压缩实现，如何结合部分积的生成过程进行优化需要做进一步地探讨。同时，对生成的近似乘法器添加可以旁路的误差补偿模块能够满足更多的应用场景，可调精度近似乘法器的自动化生成方法值得探索。最后，在近似逻辑综合的研究中，序列探索时考虑输入的到达时间能够对网表进行更好的优化，可对逻辑综合工具添加相应功能进行支持。

附录 A 不同电路在不同优化方法下的面积和延迟数据

	ABC resyn2		LSOracle ^[156]		BOiLS ^[163]		DRiLLS ^[159]		本文的方法	
	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)	Area (μm^2)	Delay (ps)
VTR/arm_core	156318.23	1034.19	151004.5	1056.51	152004.87	1041.54	161500.76	1031.03	146799.08	1022.63
OPDB/sparc_core	150133	678.49	156659.43	1376.45	112817.12	634.49	110688.54	675.17	140216.34	668.52
OPDB/sparc_ifu	82344.63	1418.87	82610.32	1623.1	80266.22	1366.24	77454.7	1378.32	78119.41	1397.56
QUIP/oc_aquarius	74737.33	1319.75	72260.9	1322.08	76614.06	1289.77	65852.52	1310.53	70277.92	1324.6
Koios/spmv	69036.24	486.8	68554.43	442.71	69442.06	518	67610.2	441.99	67428.36	514.55
VTR/mcml	64033.73	2422.24	65656.8	2516.05	63919.26	2408.69	63803.23	2477.33	62154.54	2412.85
Koios/attention_layer	58978.97	672.35	54964.41	655.77	52220.98	675.19	52565	662.03	52056.65	665.22
OPDB/I15_wrap	54529.61	694.74	54230.35	636.12	52146.39	705.92	52314.85	658.1	51671.43	659.52
OPDB/lsu	47143.15	517.17	52444.45	650.54	15886.88	564.46	15887.74	491.67	15565.55	533.04
Koios/bnn	46684.91	766.98	52291.84	703.69	47817.84	779.37	48742.38	750.25	47467.94	760.95
VTR/LU8PEEng	30131.14	4459.65	32038.3	4468.85	29928.95	4324.69	30648.11	4370.53	29467.1	4179.1
OPDB/fpga_bridge_rcv_32	26856.86	489.02	26317.4	492.81	26211.68	437.79	26362.59	455.97	25876.15	514.84
VTR/sv_chip2_hierarchy_no_mem	17329.58	1039.59	17958.9	1047.64	16616.14	1026.55	16746.82	1004.29	16591.72	1014.9
Koios/robot_maze	16794.64	368.82	16629.96	331.26	16365.57	368.82	16652.61	342.07	16222.98	321.65
OPDB/sparc_exu_wrap	15425.38	465.44	14959.43	574.57	15648.44	461.05	15623.78	469.13	15534.19	439.89
VTR/mkDelayWorker32B	13983.25	428.96	13891.52	434.21	13788.95	433.98	13791.44	482.77	13755.33	443.44
VTR/sv_chip1_hierarchy_no_mem	10483.11	506.19	10734.42	484.28	10837.47	499.5	10652.57	506.75	10626.01	503.2
VTR/system	8871.36	639.46	8888.65	636.27	8924.72	610.22	8848.41	633.89	8905.1	619.61
IWLS05/opencores/vga_enh_top	8781.15	322.06	8581.47	319.67	8464.84	346.26	8290.25	327.63	8429.19	330.72
VTR/bgm	8666.03	860	8759.63	874.37	8689.88	792.87	8410.6	821.83	8406.27	739.68
Koios/top	7834.3	664.41	7767.93	680.06	7699.67	674.58	7736.92	670.08	7741.44	669.85
QUIP/oc_video_compression_systems_det	7757.51	709.1	8111.51	705.42	8171.45	704.5	7897.9	714.94	7807.3	705.37
QUIP/oc_video_compression_systems_jpeg	7582.1	638.1	7734.78	631.2	7326.87	646.18	7237.85	643.14	7537.16	660.06
Koios/softmax	7088.1	574.7	7280.45	580.09	7098.39	584.58	7096.01	566.71	7117.66	554.47
QUIP/oc_des_des3perf	6363.16	266.97	6877.2	300.73	6084.95	291.12	6377.9	278.75	6089.83	263.97
OPDB/sparc_ffu_nospu_wrap	5834.26	431.75	5544.7	417.95	5788.55	449.04	5747.13	408.44	5683.58	415.36
EPFL/arithmetic/div/top	5749.54	13077.3	11437.69	16414.99	2905.78	12444.78	2856.34	12357.49	5364.57	14599.44
IWLS05/opencores/eth_top	5577.2	360.39	5512.76	422.38	5404.44	355.11	5520.66	461.11	5450.37	354.14
OPDB/tlu_nospu_wrap	5529.52	577.22	6330.68	610.14	5568.25	579.02	5558.19	639.39	5578.86	598.3
VTR/sv_chip0_hierarchy_no_mem	5308.69	227.37	5300.49	243.28	5261	228.25	5405.62	236.44	5343.03	232.42
VTR/paj_raygentop_hierarchy_no_mem	4919.35	611.17	4892.3	595.75	4711.31	624	4839.79	612.42	4588.54	614.16
OPDB/chip_bridge	4842.7	471.79	4820.1	480	4828.72	508.88	4875.42	505.32	4828.97	480.73
VTR/paj_boundtop_hierarchy_no_mem	4815.51	369.97	4719.6	397.61	4670.82	392.17	4671.87	369.47	4755.85	363.68
VTR/mkPktMerge	3807.71	240.87	3772.41	237.02	3786.4	231.31	3970.24	240.47	3771.12	229.37
EPFL/arithmetic/sqrt/top	3696.8	23481.82	3430.43	22816.8	2731.1	14585.37	3035.1	16686.45	2634.27	14402.77
IWLS05/faraday/frisc	3076.35	344.28	3022.62	356.66	2968.15	351.11	3014.66	354.17	2948.54	360.18
IWLS05/faraday/TOP	2839.47	666.02	2964.76	687.55	2838.23	681.02	2861.15	694.87	2347.89	660.13
EPFL/arithmetic/log2/top	2325.48	3292.23	2460.2	3336.55	2290.96	3259.56	2051.1	3311.99	2059.51	3331.18
QUIP/oc_fpu	2317.62	9028.21	2689.82	9471.38	2443.74	9272.64	2308.25	9109.36	2249.26	9004.45
IWLS05/opencores/fpu	2281.89	9158.31	2577.15	9600.33	2413.89	9194.28	2238.31	9066.59	2279.73	9439.21
VTR/or1200_flat	2070.42	929.24	2130.56	909.13	2100	918.18	2113.62	924.61	2086.46	925.56
QUIP/oc_aes_core_inv	2027.51	468.93	2158.66	478.1	1971.01	472.3	1982.25	478.06	1963.31	468.55
IWLS05/opencores/wb_connmax_top	1879	347.58	1855.04	356.91	1854.12	354.62	1875.96	337.57	1819.98	345.62
IWLS05/opencores/des/des	1840.64	265.01	2027.52	282.44	1815.81	265.39	1808.59	265.33	1788.25	267.05
IWLS05/opencores/pci_bridge32	1760.81	331.82	1758.99	324.41	1760.01	308.79	1774.05	308.96	1757.34	317.8

附录 A 不同电路在不同优化方法下的面积和延迟数据

QUIP/oc_des_perf_opt	1732.96	239.08	1881.01	244.64	1760.16	232.21	1737.82	240.92	1656.59	250.27
EPFL/arithmetic/multiplier/top	1695.89	1619.34	1781.89	1692.38	1726.97	1642.82	1799	1650.9	1636.31	1665.07
EPFL/random_control/mem_ctrl/mem_ctrl	1645.18	515.5	2005.52	624.04	1399.58	424.67	1530.43	449.96	1479.75	439.06
OPDB/picorv32	1520.33	344.63	1447.47	363.8	1528.82	341.08	1478.95	362.14	1455.89	366.51
VTR/RLE_BlobMerging	1519.1	806.13	1575.09	859.4	1439.22	783.26	1462.14	774.85	1474.36	842.01
OPDB/dynamic_node_top_wrap	1374.97	400.55	1405.89	417.44	1398.83	381.79	1389.21	401.73	1369.43	411.55
QUIP/oc_vga_lcd	1304.74	492.54	1317.9	484.88	1312.45	474.83	1316.34	496.62	1320.19	462.76
IWLS05/faraday/dma_top	1259.04	319.33	1293.14	333.87	1195.43	299.59	1193.85	323.54	1195.25	312.86
EPFL/arithmetic/square/top	1232.05	667.26	1398.19	674.22	1230.52	655.79	1383.5	662.29	1215.4	666.1
QUIP/oc_aes_core	1223.48	378.76	1336.02	386.81	1174.35	361.26	1133.52	359.47	1169.62	364.53
QUIP/oc_mem_ctrl	1208.17	406.68	1136.74	422.89	1111.35	398.25	1118.68	424.05	1104.67	386.17
IWLS05/opencores/aes_cipher_top	1200.75	377.16	1265.3	393.31	1140.68	363.2	1123.14	382.63	1155.84	374.91
VTR/diffeq_paj_convert	1197.88	1637.46	1289.53	1678.27	1152.49	1521.14	1130.69	1566.63	1138.8	1551.99
IWLS05/opencores/usb_top	1145.4	359.32	1137.37	443.38	1136.53	347.3	1134.09	358.14	1112.06	341.35
VTR/diffeq_f_systemC	1144.47	1422.12	1240.16	1453.87	1109.33	1363.86	1059.88	1406.98	1121.76	1403.46
IWLS05/opencores/ac97_top	1138.96	182.35	1162.89	195.36	1149.78	185.99	1133.07	187.77	1127.57	186.43
QUIP/oc_wb_dma	1099.78	356.02	1111.51	360.9	1130.8	355.21	1122.66	374.49	1117.25	361.49
IWLS05/iscas/s35932_bench	1076.21	169.44	1097.25	170.76	1021.45	163.52	1038.1	188.73	1036.49	167.8
IWLS05/iscas/s38417_bench	977.66	347.71	993.34	354.3	971.68	331.22	971.55	334.73	972.87	328.98
VTR/mkSMAAdapter4B	887.51	280.22	894.1	305.31	857.27	301	865.18	308.63	838.47	305.95
IWLS05/iscas/s38584_bench	841.62	219.47	841.24	209.69	843.19	211.99	818.94	228.29	835.62	219.14
OPDB/gng	795.34	600.95	807.5	583.32	784.97	580.01	746.39	595	785.51	564.24
QUIP/oc_ethernet	753.22	326.97	771.34	366.65	672.52	362.85	686.65	333.46	675.39	306.66
VTR/sha1	741.47	836.1	781.18	855.82	786.24	826.3	734.09	850.7	731.55	864.97
IWLS05/opencores/mc_top	687.14	360.01	672.52	392.63	680.22	377.51	683.2	379.57	676.85	366.03
EPFL/random_control/arbiter/top	679.78	191	428.52	207.17	649.23	196.13	607.77	199.84	691.24	194.49
IWLS05/opencores/aes	651.41	522.79	648.59	548.32	654.07	500	611.24	496.36	617.89	468.53
QUIP/barrel64	618.86	278.96	660.34	263.79	393.32	260.42	505.36	277.61	268.53	242.69
EPFL/random_control/voter/top	612.53	867.08	619.45	803.12	567.4	827.93	599.78	811.7	532.52	812.11
QUIP/oc_minirisc	501.33	382.46	491.84	419.13	506.17	386.36	500.36	370.22	512.53	385.32
EPFL/arithmetic/sin/top	490.22	1660.26	489.33	1625.4	479.7	1658.69	464.08	1653.62	462.38	1615.52
IWLS05/opencores/tv80s	467.51	508.23	506.32	489.44	430.97	506.64	440.53	505.73	436.32	497.55
QUIP/mux8_128bit	453.15	175.76	470.29	173.01	445.62	158.39	454.21	166.12	453.15	175.76
QUIP/fip_risc8	426.76	524.26	415.19	534.18	423.3	503.78	422.62	493.23	418.81	509.51
QUIP/mux64_16bit	422.81	208.73	432.37	231.71	428.8	198.81	429.38	226.94	434.89	239.25
Koios/reduction_layer	377.99	247.58	372.74	269.03	365.05	260.76	359.75	256.7	371.09	244.52
QUIP/oc_des_des3area	369.97	357.65	313.51	384.91	309.85	330.27	303.7	351.07	339.44	347.7
IWLS05/opencores/wb_dma_top	345.3	246.2	354.75	220.41	351.58	219.89	346.35	215.54	348.67	219.74
OPDB/dynamic_node_top_wrap_para	314.97	302.58	320.41	316.86	327.57	323.93	323.28	313.93	321.23	291.05
OPDB/sparc_ifu_esl	271.9	614.69	293.48	640.66	276.73	633.8	297.39	631.57	279.83	660.76
EPFL/arithmetic/bar/top	250.72	164.93	257.89	176.79	257.47	169.95	241.49	168.53	256.24	167.93
IWLS05/opencores/systemcdes/des	237.93	393.39	246.62	393.39	244.2	406.84	242.54	385.73	240.21	380.3
QUIP/oc_des_area_opt	231.27	306.63	263.11	270.26	235.16	300.61	246.69	286.06	241.33	268.36
QUIP/mux8_64bit	229.82	166.91	244.89	182.68	224.4	164.8	229.82	166.91	229.82	166.91
IWLS05/opencores/spi_top	226.6	420.72	242.33	421.39	217.3	394.29	210.58	407.88	229.43	385.52
QUIP/mux32_16bit	219.37	187.45	221.5	218.49	224.59	202.1	219.52	175.81	224.69	188.63
EPFL/arithmetic/max/top	200.01	475.64	269.67	499.71	196.23	474.47	206.37	503.86	202.18	484.03
QUIP/fip_cordic_rca	199.02	475.82	187.37	457.61	144.62	484.13	178.55	461.71	167.1	451.91
QUIP/barrel32	193.64	232.73	120.33	191.38	121.06	206.27	121.8	204.72	118.36	207.81
QUIP/fip_cordic_cla	157.35	455.6	166.75	449.44	153.59	404.7	159.84	420	160.22	392.92
QUIP/oc_video_compression_systems_huffman_dec	137.82	311.98	137.75	300.93	128.01	286.6	130.96	284.09	127.79	285.64
IWLS05/iscas/s13207_bench	137.69	156.11	139.62	154.08	133.73	180.7	136.95	159.48	135.55	157.23
EPFL/arithmetic/adder/top	129.56	260.59	135.48	254.11	155.32	263.97	188.04	240.38	155.89	263.73
IWLS05/iscas/s5378_bench	122.88	237.89	128.86	244.47	121.51	236.22	123.55	228.81	123.2	233.88
VTR/memset	111.38	246.91	112.03	239.64	113.75	245.05	114.92	238.59	113.94	245.62
QUIP/oc_video_compression_systems_huffman_enc	110.3	209.02	110.31	197.72	109.76	204.06	114.02	192.1	109.19	186.22
QUIP/oc_rtc	110.3	249.43	105.05	268.63	116.98	236.8	109.45	229.95	105.72	225.66
EPFL/random_control/dec/dec	107.53	76.52	107.53	76.52	107.53	76.52	107.53	76.52	107.53	76.52
EPFL/random_control/i2c/i2c	103.84	112.48	120.07	112.85	103.91	115.68	124.29	106.32	102.8	116.23
QUIP/os_sdram16	100.38	85.4	100.54	85.4	101	85.4	99.8	85.4	102.42	85.4
IWLS05/iscas/s9234_1_bench	99.49	231.14	102.23	230.72	104.57	225.6	100.5	230.03	100.5	234.06

QUIP/oc_i2c	94.17	221.69	98.84	229.67	92.76	230.21	92.36	234.14	90.91	226.58
IWLS05/opencores/i2c_master_top	91.81	233.92	98.08	225.86	91.59	225.53	92.61	223.47	92.41	224.56
QUIP/oc_sdram	90.13	183.33	87.09	182.91	86.5	194.88	86.08	189.23	86.07	178.82
IWLS05/opencores/simple_spi_top	79.9	186.75	83.51	197.93	81.75	171.91	83.83	187.52	80.7	177.4
QUIP/oc_gpio	74.53	137.16	75.92	136.08	70.61	135.4	68.09	171.76	74.08	138.73
IWLS05/iscas/s15850_bench	74.49	188.41	76.38	199.31	72.42	194.84	73.15	190.33	74.27	191.98
OPDB/fpu	73.7	172.8	47.41	159.1	25.33	124.54	14.73	125.76	26.01	125.15
IWLS05/opencores/sasc_top	70.12	169.53	71.56	165.44	66.12	168.37	63.92	164.24	63.26	160.75
VTR/sv_chip3_hierarchy_no_mem	67.67	195.9	66.47	188.06	63.85	191.8	63.77	193.92	65.52	186.42
EPFL/random_control/cavlc/top	67.18	118.11	61.59	123.2	60.52	125.17	56.56	122.77	65.16	119.98
IWLS05/iscas/s1423_bench	65.96	283.37	69.55	274.55	71.24	295.88	69.82	260.06	68.79	290.62
OPDB/sparc_ifu_esl_fsm	61.59	304.88	59.68	283.34	56.67	266.5	53.25	301.05	51.03	240.13
EPFL/random_control/priority/top	53.26	134.89	49.72	188.07	49.37	140.88	53.42	137.14	49.06	139.99
IWLS05/opencores/usb_phy	53.13	170.74	52.37	165.83	53.38	151.46	53.29	166.86	53.38	157.88
QUIP/barrel16	51.88	215.26	43.86	172.67	39.6	174.84	38.81	174.63	39	169.88
QUIP/xbar_16x16	43.89	104.47	43.89	104.47	48.08	116.74	43.89	104.47	44.25	136.22
IWLS05/iscas/s1494_bench	43.49	188.52	41.84	185.9	40.37	188.42	37.76	210.7	40.56	209.92
IWLS05/opencores/pcm_slv_top	41.7	146.94	41.06	138.16	43.16	156.93	43.08	156.93	41.6	158.97
IWLS05/iscas/s1488_bench	40.21	186	39.1	186.91	40.37	214.92	39.7	196.88	38.42	209.88
IWLS05/iscas/s1196_bench	37.73	194.92	37.22	196.92	40.33	186.15	36.57	184.54	36.68	179.02
IWLS05/iscas/s1238_bench	37.09	199.44	43.78	180.31	38.43	197.07	35.15	207.15	38.04	190.08
QUIP/barrel16a	33.91	176.32	32.06	177.48	34.82	166.47	32.76	165.6	35.63	170.62
IWLS05/iscas/s838_1_bench	31.7	174.8	31.94	175.52	32.5	210.38	31.65	179.44	32.78	199.61
QUIP/oc_fcmp	26.96	160.7	25.82	164.82	32.29	167.58	27.45	167.9	29.03	159.87
IWLS05/iscas/s820_bench	23.37	172.7	21.42	169.11	20.35	172.06	20.67	171.61	21.86	172.91
EPFL/random_control/ctrl/top	22.85	65.92	17.95	83.14	20.24	67.26	20	79.26	24.33	63.96
IWLS05/iscas/s832_bench	21.07	169.09	21	170.5	20.72	169.01	18.87	176.46	18.88	171.95
EPFL/random_control/int2float/top	20.43	99.62	19	103.36	18.59	100.87	20.89	106.3	18.9	100.94
IWLS05/iscas/s713_bench	19.64	146.15	19.26	141.32	17.61	148.98	15.53	152.37	16.66	145.32
IWLS05/iscas/s526_bench	19.46	148.61	18.39	153.03	18.95	155.35	17.88	151.2	18.41	148.35
IWLS05/iscas/s526n_bench	18.52	154.54	19.27	157.71	18.06	143.59	18.06	148.03	19.14	149.32
IWLS05/iscas/s382_bench	18.44	156.06	18.59	150.87	16.74	150.6	17.25	149.15	15.99	147.86
IWLS05/iscas/s641_bench	18.27	156.02	20.95	182.46	18.75	176.81	19.33	145	16.75	148.78
IWLS05/iscas/s444_bench	17.93	152.13	16.75	153.2	17.01	152.94	17.34	151.9	19.07	148.04
IWLS05/iscas/s510_bench	17.23	166.02	16.99	160.85	16.74	163.31	17.44	166.55	17.26	167.68
IWLS05/iscas/s400_bench	16.74	149.23	17.76	148.42	17.55	153.8	19.45	154.01	18.34	144.59
IWLS05/iscas/s420_1_bench	16.64	146.99	15.28	184.43	15.47	179.66	16.37	157.58	15.88	185.05
OPDB/sparc_ifu_esl_counter	15.57	189.6	12.58	203.02	14.75	208.77	13.63	196.49	14.52	195.84
OPDB/sparc_ifu_esl_htsm	15.43	115.13	10.69	116.65	12.83	114.84	9.99	120.39	10.63	77.5
IWLS05/iscas/s344_bench	13.91	159.39	15.06	162.78	12.93	164.58	13.81	159.59	13.94	157.13
IWLS05/iscas/s349_bench	13.79	158.8	15.8	155.55	14.43	148	13.78	153.82	13.25	151.57
EPFL/random_control/router/top	11.39	108.79	10.73	121.62	9.7	116.11	9.45	126.22	8.94	108.85
IWLS05/iscas/s298_bench	11.1	146.99	11.71	153.18	10.56	144.19	10.99	152.62	11.12	140.17
IWLS05/iscas/s386_bench	10.67	170.02	12.06	144.06	10.47	171.34	11.43	164.52	12.31	169.08
OPDB/sparc_ifu_esl_lfsr	10.35	117.2	12	108.6	11.88	118.26	11.88	118.26	11.88	118.26
OPDB/sparc_ifu_esl_stsm	8.51	91.91	12.83	83.66	10.21	85.83	7.76	86.38	7.73	90.76
OPDB/sparc_ifu_esl_shiftreg	8.05	73.36	8.05	73.36	8.05	73.36	8.05	73.36	8.05	73.36
IWLS05/iscas/s208_1_bench	7.87	158.08	7.64	162.01	7.76	164.14	8.06	155.69	8.11	157.34
QUIP/ts_mike_fsm	6.49	126.27	7.13	132.49	7.44	125.3	7.44	125.3	8	116.21
OPDB/sparc_ifu_esl_rtssm	4.67	67.2	5.85	73.44	5.66	65.92	5.66	65.92	5.66	65.92
IWLS05/iscas/s27_bench	2.1	100.2	2.1	100.2	2.1	100.2	2.1	100.2	2.1	100.2
OPDB/sparc_mul_top_nospu_wrap	1.2	71.08	1.2	71.08	1.2	71.08	1.2	71.08	1.2	71.08

参考文献

- [1] MOORE G E. Cramming more components onto integrated circuits[J]. Electronics, 1965, 38(8): 114.
- [2] MOORE G E. Progress in digital integrated electronics[C]//1975 International Electron Devices Meeting (IEDM): Vol. 21. 1975: 11-13.
- [3] DENNARD R, GAENSSLEN F, YU H N, et al. Design of ion-implanted mosfet's with very small physical dimensions[J/OL]. IEEE Journal of Solid-State Circuits, 1974, 9(5): 256-268. DOI: 10.1109/JSSC.1974.1050511.
- [4] MISTRY K, ARMSTRONG M, AUTH C, et al. Delaying forever: Uniaxial strained silicon transistors in a 90nm cmos technology[C/OL]//Digest of Technical Papers. 2004 Symposium on VLSI Technology, 2004. 2004: 50-51. DOI: 10.1109/VLSIT.2004.1345387.
- [5] KHANNA V K. Short-channel effects in mosfets[M/OL]. New Delhi: Springer India, 2016: 73-93. https://doi.org/10.1007/978-81-322-3625-2_5.
- [6] GHANI T, ARMSTRONG M, AUTH C, et al. A 90nm high volume manufacturing logic technology featuring novel 45nm gate length strained silicon cmos transistors[C/OL]//IEEE International Electron Devices Meeting 2003. 2003: 11.6.1-11.6.3. DOI: 10.1109/IEDM.2003.1269442.
- [7] BAI P, AUTH C, BALAKRISHNAN S, et al. A 65nm logic technology featuring 35nm gate lengths, enhanced channel strain, 8 cu interconnect layers, low-k ild and 0.57 /spl mu/m/sup 2/ sram cell[C/OL]//IEDM Technical Digest. IEEE International Electron Devices Meeting, 2004. 2004: 657-660. DOI: 10.1109/IEDM.2004.1419253.
- [8] MISTRY K, ALLEN C, AUTH C, et al. A 45nm logic technology with high-k+metal gate transistors, strained silicon, 9 cu interconnect layers, 193nm dry patterning, and 1002007 IEEE International Electron Devices Meeting. 2007: 247-250. DOI: 10.1109/IEDM.2007.4418914.

- [9] JAN C H, AGOSTINELLI M, BUEHLER M, et al. A 32nm soc platform technology with 2nd generation high-k/metal gate transistors optimized for ultra low power, high performance, and high density product applications[C/OL]// 2009 IEEE International Electron Devices Meeting (IEDM). 2009: 1-4. DOI: 10.1109/IEDM.2009.5424258.
- [10] HISAMOTO D, LEE W C, KEDZIERSKI J, et al. Finfet-a self-aligned double-gate mosfet scalable to 20 nm[J/OL]. IEEE Transactions on Electron Devices, 2000, 47(12): 2320-2325. DOI: 10.1109/16.887014.
- [11] HENNESSY J L, PATTERSON D A. A new golden age for computer architecture [J/OL]. Commun. ACM, 2019, 62(2): 48–60. <https://doi.org/10.1145/3282307>.
- [12] VON NEUMANN J. First draft of a report on the EDVAC[R]. University of Pennsylvania, 1945.
- [13] WULF W A, MCKEE S A. Hitting the memory wall: Implications of the obvious [J/OL]. SIGARCH Comput. Archit. News, 1995, 23(1): 20–24. <https://doi.org/10.1145/216585.216588>.
- [14] SUTTER H. The free lunch is over: A fundamental turn toward concurrency in software[J]. Dr. Dobb’s journal, 2005, 30(3): 202-210.
- [15] KARLRUPP. Microprocessor trend data[EB/OL]. <https://github.com/karlrupp/microprocessor-trend-data>.
- [16] 胡伟武. 计算机体系结构基础[M/OL]. 3 版. 机械工业出版社, 2021. <https://github.com/foxsen/archbase>.
- [17] ESMAEILZADEH H, BLEM E, AMANT R S, et al. Dark silicon and the end of multicore scaling[C]//2011 38th Annual International Symposium on Computer Architecture (ISCA). 2011: 365-376.
- [18] AMDAHL G M. Validity of the single processor approach to achieving large scale computing capabilities[C/OL]//AFIPS ’67 (Spring): Proceedings of the April 18-20, 1967, Spring Joint Computer Conference. New York, NY, USA: Association for Computing Machinery, 1967: 483–485. <https://doi.org/10.1145/1465482.1465560>.
- [19] MOORE S K. Multicore is bad news for supercomputers[J/OL]. IEEE Spectrum, 2008, 45(11): 15-15. DOI: 10.1109/MSPEC.2008.4659375.

- [20] TAYLOR M B. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse[C/OL]//DAC '12: Proceedings of the 49th Annual Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 2012: 1131–1136. <https://doi.org/10.1145/2228360.2228567>.
- [21] JAIN S, KHARE S, YADA S, et al. A 280mv-to-1.2v wide-operating-range ia-32 processor in 32nm cmos[C/OL]//2012 IEEE International Solid-State Circuits Conference. 2012: 66-68. DOI: 10.1109/ISSCC.2012.6176932.
- [22] Intel® turbo boost technology in Intel® core microarchitecture (Nehalem) based processors[R]. Intel Corporation, 2008.
- [23] big.little technology: The future of mobile[R]. Arm Corporation, 2013.
- [24] SEVILLA J, HEIM L, HO A, et al. Compute trends across three eras of machine learning[C/OL]//2022 International Joint Conference on Neural Networks (IJCNN). 2022: 1-8. DOI: 10.1109/IJCNN55064.2022.9891914.
- [25] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C/OL]//PEREIRA F, BURGES C, BOTTOU L, et al. Advances in Neural Information Processing Systems: Vol. 25. Curran Associates, Inc., 2012. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.
- [26] SILVER D, HUANG A, MADDISON C J, et al. Mastering the game of Go with deep neural networks and tree search[J/OL]. Nature, 2016, 529(7587): 484-489. DOI: 10.1038/nature16961.
- [27] CHEN T, DU Z, SUN N, et al. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning[C/OL]//ASPLOS '14: Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2014: 269–284. <https://doi.org/10.1145/2541940.2541967>.
- [28] JOUPPI N, KURIAN G, LI S, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings[C/OL]//ISCA '23: Proceedings of the 50th Annual International Symposium on Computer Architecture. New York, NY, USA: Association for Computing Machinery, 2023. <https://doi.org/10.1145/3579371.3589350>.

- [29] JIA Z, TILLMAN B, MAGGIONI M, et al. Dissecting the graphcore ipu architecture via microbenchmarking[A]. 2019. arXiv: 1912.03413.
- [30] LIAO H, TU J, XIA J, et al. Davinci: A scalable architecture for neural network computing[C/OL]//2019 IEEE Hot Chips 31 Symposium (HCS). 2019: 1-44. DOI: 10.1109/HOTCHIPS.2019.8875654.
- [31] OUYANG J, NOH M, WANG Y, et al. Baidu kunlun an ai processor for diversified workloads[C/OL]//2020 IEEE Hot Chips 32 Symposium (HCS). 2020: 1-18. DOI: 10.1109/HCS49909.2020.9220641.
- [32] Scalable end-to-end enterprise ai on 4th gen Intel® Xeon® scalable processors [R]. Intel Corporation, 2023.
- [33] Nvidia H100 tensor core GPU architecture v1.04[R]. Nvidia, 2023.
- [34] GAIIDE B, GAITONDE D, RAVISHANKAR C, et al. Xilinx adaptive compute acceleration platform: Versal™ architecture[C/OL]//FPGA '19: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: Association for Computing Machinery, 2019: 84–93. <https://doi.org/10.1145/3289602.3293906>.
- [35] VASWANI A, SHAZER N, PARMAR N, et al. Attention is all you need [C/OL]//GUYON I, LUXBURG U V, BENGIO S, et al. Advances in Neural Information Processing Systems: Vol. 30. Curran Associates, Inc., 2017. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fb053c1c4a845aa-Paper.pdf.
- [36] GHOLAMI A, YAO Z, KIM S, et al. Ai and memory wall[J]. RiseLab Medium Post, 2021.
- [37] PATTERSON D, GONZALEZ J, LE Q, et al. Carbon emissions and large neural network training[A]. 2021. arXiv: 2104.10350.
- [38] YOU J, CHUNG J W, CHOWDHURY M. Zeus: Understanding and optimizing GPU energy consumption of DNN training[C/OL]//20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). Boston, MA: USENIX Association, 2023: 119-139. <https://www.usenix.org/conference/nsdi23/presentation/you>.

- [39] PATTERSON D, GONZALEZ J, HöLZLE U, et al. The carbon footprint of machine learning training will plateau, then shrink[J/OL]. Computer, 2022, 55(7): 18-28. DOI: 10.1109/MC.2022.3148714.
- [40] ZHOU Y, LIN X, ZHANG X, et al. On the opportunities of green computing: A survey[A]. 2023. arXiv: 2311.00447.
- [41] NAVEED H, KHAN A U, QIU S, et al. A comprehensive overview of large language models[A]. 2023. arXiv: 2307.06435.
- [42] More moore[R]. International Roadmap for Devices and Systems (IRDS), 2022.
- [43] BAE G, BAE D I, KANG M, et al. 3nm gaa technology featuring multi-bridge-channel fet for low power and high performance applications[C/OL]//2018 IEEE International Electron Devices Meeting (IEDM). 2018: 28.7.1-28.7.4. DOI: 10.1109/IEDM.2018.8614629.
- [44] More than moore[R]. International Roadmap for Devices and Systems (IRDS), 2022.
- [45] Beyond cmos and emerging materials integration[R]. International Roadmap for Devices and Systems (IRDS), 2022.
- [46] BRITNELL L, GORBACHEV R V, JALIL R, et al. Field-effect tunneling transistor based on vertical graphene heterostructures[J/OL]. Science, 2012, 335(6071): 947-950. <https://www.science.org/doi/abs/10.1126/science.1218461>.
- [47] RAY V, SUBRAMANIAN R, BHADRACHALAM P, et al. Cmos-compatible fabrication of room-temperature single-electron devices[J/OL]. Nature Nanotechnology, 2008, 3: 603-608. DOI: <https://doi.org/10.1038/nnano.2008.267>.
- [48] SCHWIERZ F. Graphene transistors[J/OL]. Nature Nanotechnology, 2010, 5: 487-496. DOI: <https://doi.org/10.1038/nnano.2010.89>.
- [49] SCHROTER M, CLAUS M, SAKALAS P, et al. Carbon nanotube fet technology for radio-frequency electronics: State-of-the-art overview[J/OL]. IEEE Journal of the Electron Devices Society, 2013, 1(1): 9-20. DOI: 10.1109/JEDS.2013.2244641.

- [50] LIU W, LOMBARDI F, SCHULTE M. Approximate computing: From circuits to applications [scanning the issue][J/OL]. Proceedings of the IEEE, 2020, 108(12): 2103-2107. DOI: 10.1109/JPROC.2020.3033361.
- [51] HAN J, ORSHANSKY M. Approximate computing: An emerging paradigm for energy-efficient design[C/OL]//2013 18th IEEE European Test Symposium (ETS). 2013: 1-6. DOI: 10.1109/ETS.2013.6569370.
- [52] XU W, SAPATNEKAR S S, HU J. A simple yet efficient accuracy-configurable adder design[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2018, 26(6): 1112-1125. DOI: 10.1109/TVLSI.2018.2803081.
- [53] LI Z, ZHENG S, ZHANG J, et al. Adaptable approximate multiplier design based on input distribution and polarity[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2022, 30(12): 1813-1826. DOI: 10.1109/TVLSI.2022.3197229.
- [54] SAADAT H, JAVAID H, PARAMESWARAN S. Approximate integer and floating-point dividers with near-zero error bias[C]//2019 56th ACM/IEEE Design Automation Conference (DAC). 2019: 1-6.
- [55] SCARABOTTOLO I, ANSALONI G, CONSTANTINIDES G A, et al. Approximate logic synthesis: A survey[J/OL]. Proceedings of the IEEE, 2020, 108(12): 2195-2213. DOI: 10.1109/JPROC.2020.3014430.
- [56] SHI K, BOLAND D, STOTT E, et al. Datapath synthesis for overclocking: Online arithmetic for latency-accuracy trade-offs[C/OL]//DAC '14: Proceedings of the 51st Annual Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 2014: 1–6. <https://doi.org/10.1145/2593069.2593118>.
- [57] JIANG H, SANTIAGO F J H, MO H, et al. Approximate arithmetic circuits: A survey, characterization, and recent applications[J/OL]. Proceedings of the IEEE, 2020, 108(12): 2108-2135. DOI: 10.1109/JPROC.2020.3006451.
- [58] LI F, LU Y, WU Z, et al. Ascache: An approximate ssd cache for error-tolerant applications[C]//2019 56th ACM/IEEE Design Automation Conference (DAC). 2019: 1-6.

- [59] RAHA A, SUTAR S, JAYAKUMAR H, et al. Quality configurable approximate dram[J/OL]. IEEE Transactions on Computers, 2017, 66(7): 1172-1187. DOI: 10.1109/TC.2016.2640296.
- [60] FRUSTACI F, BLAAUW D, SYLVESTER D, et al. Approximate srams with dynamic energy-quality management[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016, 24(6): 2128-2141. DOI: 10.1109/T-VLSI.2015.2503733.
- [61] RAHA A, RAGHUNATHAN V. Towards full-system energy-accuracy tradeoffs: A case study of an approximate smart camera system[C/OL]//DAC '17: Proceedings of the 54th Annual Design Automation Conference 2017. New York, NY, USA: Association for Computing Machinery, 2017. <https://doi.org/10.1145/3061639.3062333>.
- [62] BAUGH C, WOOLEY B. A two's complement parallel array multiplication algorithm[J/OL]. IEEE Transactions on Computers, 1973, C-22(12): 1045-1047. DOI: 10.1109/T-C.1973.223648.
- [63] HATAMIAN M, CASH G. A 70-mhz 8-bit/spl times/8-bit parallel pipelined multiplier in 2.5-/spl mu/m cmos[J/OL]. IEEE Journal of Solid-State Circuits, 1986, 21(4): 505-513. DOI: 10.1109/JSSC.1986.1052564.
- [64] SJALANDER M, LARSSON-EDEFORS P. High-speed and low-power multipliers using the baugh-wooley algorithm and hpm reduction tree[C/OL]//2008 15th IEEE International Conference on Electronics, Circuits and Systems. 2008: 33-36. DOI: 10.1109/ICECS.2008.4674784.
- [65] BRAYTON R, MISHCHENKO A. Abc: An academic industrial-strength verification tool[C]//TOUILI T, COOK B, JACKSON P. Computer Aided Verification. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010: 24-40.
- [66] MITCHELL J N. Computer multiplication and division using binary logarithms [J/OL]. IRE Transactions on Electronic Computers, 1962, EC-11(4): 512-517. DOI: 10.1109/TEC.1962.5219391.
- [67] ANSARI M S, MRAZEK V, COCKBURN B F, et al. Improving the accuracy and hardware efficiency of neural networks using approximate multipliers[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2020, 28 (2): 317-328. DOI: 10.1109/TVLSI.2019.2940943.

- [68] BOOTH A D. A SIGNED BINARY MULTIPLICATION TECHNIQUE[J/OL]. The Quarterly Journal of Mechanics and Applied Mathematics, 1951, 4(2): 236-240. <https://doi.org/10.1093/qjmam/4.2.236>.
- [69] MACSORLEY O L. High-speed arithmetic in binary computers[J/OL]. Proceedings of the IRE, 1961, 49(1): 67-91. DOI: 10.1109/JRPROC.1961.287779.
- [70] RUBINFIELD L. A proof of the modified booth's algorithm for multiplication [J/OL]. IEEE Transactions on Computers, 1975, C-24(10): 1014-1015. DOI: 10.1109/T-C.1975.224114.
- [71] WALLACE C S. A suggestion for a fast multiplier[J/OL]. IEEE Transactions on Electronic Computers, 1964, EC-13(1): 14-17. DOI: 10.1109/PGEC.1964.263830.
- [72] DADDA L. Some schemes for parallel multipliers[J]. Alta Frequenza, 1965, 34 (5): 349-356.
- [73] Sign extension in booth multipliers[EB/OL]. <http://i.stanford.edu/pub/cstr/reports/csl/tr/94/617/CSL-TR-94-617.appendix.pdf>.
- [74] [美] 拉贝艾. 数字集成电路[M]. 周润德, 译. 电子工业出版社, 2010.
- [75] ROBINSON M, SWARTZLANDER E. A reduction scheme to optimize the wallace multiplier[C/OL]//Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No.98CB36273). 1998: 122-127. DOI: 10.1109/ICCD.1998.727032.
- [76] ASIF S, KONG Y. Design of an algorithmic wallace multiplier using high speed counters[C/OL]//2015 Tenth International Conference on Computer Engineering & Systems (ICCES). 2015: 133-138. DOI: 10.1109/ICCES.2015.7393033.
- [77] PARHAMI B. Computer arithmetic: Algorithms and hardware designs[M/OL]. 2nd ed. Oxford University Press, 2010. <https://ashkanyeganeh.com/wp-content/uploads/2020/03/computer-arithmetic-algorithms-2nd-edition-Behrooz-Parhami.pdf>.
- [78] VITOROULIS K. Parallel prefix adders[Z]. Concordia University, 2006.

- [79] SKLANSKY J. An evaluation of several two-summand binary adders[J/OL]. IRE Transactions on Electronic Computers, 1960, EC-9(2): 213-226. DOI: 10.1109/TEC.1960.5219821.
- [80] KOGGE P M, STONE H S. A parallel algorithm for the efficient solution of a general class of recurrence equations[J/OL]. IEEE Transactions on Computers, 1973, C-22(8): 786-793. DOI: 10.1109/TC.1973.5009159.
- [81] LADNER R E, FISCHER M J. Parallel prefix computation[J/OL]. J. ACM, 1980, 27(4): 831–838. <https://doi.org/10.1145/322217.322232>.
- [82] BRENT, KUNG. A regular layout for parallel adders[J/OL]. IEEE Transactions on Computers, 1982, C-31(3): 260-264. DOI: 10.1109/TC.1982.1675982.
- [83] HAN T, CARLSON D A. Fast area-efficient vlsi adders[C/OL]//1987 IEEE 8th Symposium on Computer Arithmetic (ARITH). 1987: 49-56. DOI: 10.1109/ARITH.1987.6158699.
- [84] KNOWLES S. A family of adders[C/OL]//Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336). 1999: 30-34. DOI: 10.1109/ARITH.1999.762825.
- [85] HARRIS D, SUTHERLAND I. Logical effort of carry propagate adders[C/OL]//The Thirty-Seventh Asilomar Conference on Signals, Systems & Computers, 2003: Vol. 1. 2003: 873-878 Vol.1. DOI: 10.1109/ACSSC.2003.1292037.
- [86] DIMITRAKOPoulos G, NIKOLOS D. High-speed parallel-prefix vlsi ling adders[J/OL]. IEEE Transactions on Computers, 2005, 54(2): 225-231. DOI: 10.1109/TC.2005.26.
- [87] BEAUMONT-SMITH A, LIM C C. Parallel prefix adder design[C/OL]//Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001. 2001: 218-225. DOI: 10.1109/ARITH.2001.930122.
- [88] ROY R, RAIMAN J, KANT N, et al. Prefixrl: Optimization of parallel prefix circuits using deep reinforcement learning[C/OL]//2021 58th ACM/IEEE Design Automation Conference (DAC). 2021: 853-858. DOI: 10.1109/DAC18074.2021.9586094.

- [89] MA Y, ROY S, MIAO J, et al. Cross-layer optimization for high speed adders: A pareto driven machine learning approach[J/OL]. Trans. Comp.-Aided Des. Integ. Cir. Sys., 2019, 38(12): 2298–2311. <https://doi.org/10.1109/TCAD.2018.2878129>.
- [90] GENG H, MA Y, XU Q, et al. High-speed adder design space exploration via graph neural processes[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, 41(8): 2657-2670. DOI: 10.1109/TCAD.2021.3114262.
- [91] VAHDAT S, KAMAL M, AFZALI-KUSHA A, et al. TOSAM: An energy-efficient truncation- and rounding-based scalable approximate multiplier[J/OL]. IEEE Trans. VLSI Syst., 2019, 27(5): 1161-1173. DOI: 10.1109/TVLSI.2018.2890712.
- [92] MOGAMI T. Deep neural network training without multiplications[A]. 2020. arXiv: 2012.03458.
- [93] KIM M S, DEL BARRIO A A, HERMIDA R, et al. Low-power implementation of mitchell's approximate logarithmic multiplication for convolutional neural networks[C/OL]//2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC). 2018: 617-622. DOI: 10.1109/ASPDAC.2018.8297391.
- [94] KIM M S, BARRIO A A D, OLIVEIRA L T, et al. Efficient mitchell' s approximate log multipliers for convolutional neural networks[J/OL]. IEEE Transactions on Computers, 2019, 68(5): 660-675. DOI: 10.1109/TC.2018.2880742.
- [95] WU Y, CHEN C, XIAO W, et al. A survey on approximate multiplier designs for energy efficiency: From algorithms to circuits[A]. 2023. arXiv: 2301.12181.
- [96] KULKARNI P, GUPTA P, ERCEGOVAC M. Trading accuracy for power with an underdesigned multiplier architecture[C/OL]//2011 24th Internatioal Conference on VLSI Design. 2011: 346-351. DOI: 10.1109/VLSID.2011.51.
- [97] LIU C, HAN J, LOMBARDI F. A low-power, high-performance approximate multiplier with configurable partial error recovery[C/OL]//2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2014: 1-4. DOI: 10.7873/DATE.2014.108.

- [98] QIQIEH I, SHAFIK R, TARAWNEH G, et al. Energy-efficient approximate multiplier design using bit significance-driven logic compression[C/OL]//Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. 2017: 7-12. DOI: 10.23919/DATE.2017.7926950.
- [99] HASHEMI S, BAHAR R I, REDA S. Drum: A dynamic range unbiased multiplier for approximate applications[C/OL]//2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2015: 418-425. DOI: 10.1109/ICCAD.2015.7372600.
- [100] CHEN C, YANG S, QIAN W, et al. Optimally approximated and unbiased floating-point multiplier with runtime configurability[C]//2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD). 2020: 1-9.
- [101] MILLER J F, HARDING S L. Cartesian genetic programming[C/OL]//GECCO '08: Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation. New York, NY, USA: Association for Computing Machinery, 2008: 2701–2726. <https://doi.org/10.1145/1388969.1389075>.
- [102] MILLER J F. Cartesian genetic programming[M]. Springer-Verlag, 2011.
- [103] MRAZEK V, SARWAR S S, SEKANINA L, et al. Design of power-efficient approximate multipliers for approximate artificial neural networks[C/OL]//2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2016: 1-7. DOI: 10.1145/2966986.2967021.
- [104] MILLER J, THOMSON P, FOGARTY T. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study[J]. Genetic Algorithms and Evolution Strategies in Engineering and Computer Science, 1997: 105-131.
- [105] MILLER J F. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach[C]//GECCO'99: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999: 1135–1142.
- [106] MILLER J F, THOMSON P. Cartesian genetic programming[C]//POLI R, BANZHAF W, LANGDON W B, et al. Genetic Programming. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000: 121-132.

- [107] MRAZEK V, HRBACEK R, VASICEK Z, et al. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods[C/OL]//Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. 2017: 258-261. DOI: 10.23919/DATE.2017.7926993.
- [108] VASICEK Z, MRAZEK V, SEKANINA L. Automated circuit approximation method driven by data distribution[C/OL]//2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2019: 96-101. DOI: 10.23919/DAT E.2019.8714977.
- [109] ČEŠKA M, MATYAŠ J, MRAZEK V, et al. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished [C/OL]//2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2017: 416-423. DOI: 10.1109/ICCAD.2017.8203807.
- [110] MENG C, QIAN W, MISHCHENKO A. Alsrac: Approximate logic synthesis by resubstitution with approximate care set[C/OL]//2020 57th ACM/IEEE Design Automation Conference (DAC). 2020: 1-6. DOI: 10.1109/DAC18072.2020.9218627.
- [111] LEE S Y, RIENER H, MISHCHENKO A, et al. Simulation-guided boolean resubstitution[A]. 2020. arXiv: 2007.02579.
- [112] AHMED M A O. Trained neural networks ensembles weight connections analysis[C]//HASSANIEN A E, TOLBA M F, ELHOSENY M, et al. The International Conference on Advanced Machine Learning Technologies and Applications (AMLTA2018). Cham: Springer International Publishing, 2018: 242-251.
- [113] LECUN Y, BOTTOU L, BENGIO Y, et al. Gradient-based learning applied to document recognition[J/OL]. Proceedings of the IEEE, 1998, 86(11): 2278-2324. DOI: 10.1109/5.726791.
- [114] KRIZHEVSKY A, SUTSKEVER I, HINTON G E. Imagenet classification with deep convolutional neural networks[C/OL]//PEREIRA F, BURGES C, BOTTOU L, et al. Advances in Neural Information Processing Systems: Vol. 25. Curran Associates, Inc., 2012. https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

- [115] SIMONYAN K, ZISSERMAN A. Very deep convolutional networks for large-scale image recognition[A]. 2015. arXiv: 1409.1556.
- [116] KRIZHEVSKY A. Learning multiple layers of features from tiny images[D/OL]. University of Toronto, 2009. <https://www.cs.utoronto.ca/~kriz/learning-features-2009-TR.pdf>.
- [117] JACOB B, KLIGYS S, CHEN B, et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference[C/OL]//2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2018: 2704-2713. DOI: 10.1109/CVPR.2018.00286.
- [118] VENKATACHALAM S, KO S B. Design of power and area efficient approximate multipliers[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2017, 25(5): 1782-1786. DOI: 10.1109/TVLSI.2016.2643639.
- [119] ZENDEGANI R, KAMAL M, BAHADORI M, et al. Roba multiplier: A rounding-based approximate multiplier for high-speed yet energy-efficient digital signal processing[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2017, 25(2): 393-401. DOI: 10.1109/TVLSI.2016.2587696.
- [120] ZERVAKIS G, TSOUUMANIS K, XYDIS S, et al. Design-efficient approximate multiplication circuits through partial product perforation[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2016, 24(10): 3105-3117. DOI: 10.1109/TVLSI.2016.2535398.
- [121] MRAZEK V, VASICEK Z, SEKANINA L, et al. Scalable construction of approximate multipliers with formally guaranteed worst case error[J/OL]. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2018, 26(11): 2572-2576. DOI: 10.1109/TVLSI.2018.2856362.
- [122] The-openroad-project/asap7[EB/OL]. <https://github.com/The-OpenROAD-Project/asap7>.
- [123] DesignWare library: Datapath and building block IP[EB/OL]. Synopsys. <https://www.synopsys.com/dw/buildingblock.php>.
- [124] ZHOU S, WU Y, NI Z, et al. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients[A]. 2018. arXiv: 1606.06160.

- [125] JIAO L, LUO C, CAO W, et al. Accelerating low bit-width convolutional neural networks with embedded fpga[C/OL]//2017 27th International Conference on Field Programmable Logic and Applications (FPL). 2017: 1-4. DOI: 10.23919/FPL.2017.8056820.
- [126] WANG Y, WANG Y, LI H, et al. Systolic cube: A spatial 3d cnn accelerator architecture for low power video analysis[C/OL]//DAC '19: Proceedings of the 56th Annual Design Automation Conference 2019. New York, NY, USA: Association for Computing Machinery, 2019. <https://doi.org/10.1145/3316781.3317919>.
- [127] LANGHAMMER M, PASCA B. Floating-point DSP block architecture for FPGAs[C]//FPGA '15: Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: Association for Computing Machinery, 2015: 117–125.
- [128] BOUTROS A, ELDAFRAWY M, YAZDANSHENAS S, et al. Math doesn't have to be hard: Logic block architectures to enhance low-precision multiply-accumulate on fpgas[C/OL]//FPGA '19: Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: Association for Computing Machinery, 2019: 94–103. <https://doi.org/10.1145/3289602.3293912>.
- [129] ANDRONIC M, CONSTANTINIDES G A. Polylut: Learning piecewise polynomials for ultra-low latency fpga lut-based inference[C/OL]//2023 International Conference on Field Programmable Technology (ICFPT). 2023: 60-68. DOI: 10.1109/ICFPT59805.2023.00012.
- [130] SHI K, ZHOU H, WANG L. Vib: A versatile interconnection block for fpga routing architecture[C/OL]//2023 International Conference on Field Programmable Technology (ICFPT). 2023: 79-87. DOI: 10.1109/ICFPT59805.2023.00014.
- [131] ULLAH S, MURTHY S S, KUMAR A. SMAproxlib: Library of fpga-based approximate multipliers[C/OL]//DAC '18: Proceedings of the 55th Annual Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3195970.3196115.
- [132] ULLAH S, REHMAN S, PRABAKARAN B S, et al. Area-optimized low-latency approximate multipliers for fpga-based hardware accelerators[C/OL]//

- DAC '18: Proceedings of the 55th Annual Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 2018. DOI: 10.1145/3195970.3195996.
- [133] YAO S, ZHANG L. Hardware-efficient fpga-based approximate multipliers for error-tolerant computing[C/OL]//2022 International Conference on Field-Programmable Technology (ICFPT). 2022: 1-8. DOI: 10.1109/ICFPT56656.2022.9974399.
- [134] ULLAH S, REHMAN S, SHAFIQUE M, et al. High-performance accurate and approximate multipliers for fpga-based hardware accelerators[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2022, 41(2): 211-224. DOI: 10.1109/TCAD.2021.3056337.
- [135] Multiplier v12.0 LogiCORE IP product guide (PG108)[EB/OL]. Xilinx, 2015. <https://docs.xilinx.com/v/u/en-US/pg108-mult-gen>.
- [136] BETZ V, ROSE J, MARQUARDT A. Architecture and cad for deep-submicron fpgas[M]. Kluwer Academic Publishers, 1999.
- [137] SNOEK J, LAROCHELLE H, ADAMS R P. Practical bayesian optimization of machine learning algorithms[C]//Advances in Neural Information Processing Systems: Vol. 25. Curran Associates, Inc., 2012.
- [138] BERGSTRA J, BARDENET R, BENGIO Y, et al. Algorithms for hyper-parameter optimization[C]//Advances in Neural Information Processing Systems: Vol. 24. Curran Associates, Inc., 2011.
- [139] PRABAKARAN B S, MRAZEK V, VASICEK Z, et al. ApproxFPGAs: Embracing asic-based approximate arithmetic components for fpga-based systems [C/OL]//2020 57th ACM/IEEE Design Automation Conference (DAC). 2020: 1-6. DOI: 10.1109/DAC18072.2020.9218533.
- [140] SOEKEN M, HAASWIJK W, TESTA E, et al. Practical exact synthesis[C/OL]// 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2018: 309-314. DOI: 10.23919/DATE.2018.8342027.
- [141] ANDERSON J H, WANG Q, RAVISHANKAR C. Raising fpga logic density through synthesis-inspired architecture[J/OL]. IEEE Transactions on Very Large

- Scale Integration (VLSI) Systems, 2012, 20(3): 537-550. DOI: 10.1109/TVLSI.2010.2102781.
- [142] RIENER H, MISHCHENKO A, SOEKEN M. Exact dag-aware rewriting [C/OL]//2020 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2020: 732-737. DOI: 10.23919/DATE48585.2020.9116379.
- [143] CONG J, DING Y. On area/depth trade-off in lut-based fpga technology mapping[C/OL]//DAC '93: Proceedings of the 30th International Design Automation Conference. New York, NY, USA: Association for Computing Machinery, 1993: 213–218. <https://doi.org/10.1145/157485.164675>.
- [144] MISHCHENKO A, BRAYTON R K. Verification after synthesis[C]// International Workshop on Logic Synthesis (IWLS). 2006: 263-267.
- [145] KUEHLMANN A, PARUTHI V, KROHM F, et al. Robust boolean reasoning for equivalence checking and functional property verification[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2002, 21 (12): 1377-1394. DOI: 10.1109/TCAD.2002.804386.
- [146] LIU G, ZHANG Z. A parallelized iterative improvement approach to area optimization for lut-based technology mapping[C/OL]//FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. New York, NY, USA: Association for Computing Machinery, 2017: 147–156. <https://doi.org/10.1145/3020078.3021735>.
- [147] WOLF C. Yosys open synthesis suite[EB/OL]. <https://github.com/YosysHQ/yosys>.
- [148] HáLEčEK I, FIŠER P, SCHMIDT J. Towards and/xor balanced synthesis: Logic circuits rewriting with xor[J/OL]. Microelectronics Reliability, 2018, 81: 274-286. <https://www.sciencedirect.com/science/article/pii/S0026271417305899>. DOI: <https://doi.org/10.1016/j.microrel.2017.12.031>.
- [149] HáLEčEK I, FIŠER P, SCHMIDT J. Are xors in logic synthesis really necessary? [C/OL]//2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS). 2017: 134-139. DOI: 10.1109/DDECES.2017.7934583.

- [150] HÁLEČEK I, FIŠER P, SCHMIDT J. On xaig rewriting[C]//International Workshop on Logic Synthesis (IWLS). 2017.
- [151] AMARÚ L, GAILLARDON P E, DE MICHELI G. Majority-inverter graph: A new paradigm for logic optimization[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2016, 35(5): 806-819. DOI: 10.1109/TCAD.2015.2488484.
- [152] HAASWIJK W, SOEKEN M, AMARÙ L, et al. A novel basis for logic rewriting[C/OL]//2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC). 2017: 151-156. DOI: 10.1109/ASPDAC.2017.7858312.
- [153] RAI S, KUMAR A. Logic synthesis with xor-majority graphs[M/OL]. Cham: Springer Nature Switzerland, 2024: 91-118. https://doi.org/10.1007/978-3-031-37924-6_5.
- [154] CONG J, WU C, DING Y. Cut ranking and pruning: enabling a general and efficient fpga mapping solution[C/OL]//FPGA '99: Proceedings of the 1999 ACM/SIGDA Seventh International Symposium on Field Programmable Gate Arrays. New York, NY, USA: Association for Computing Machinery, 1999: 29–35. <https://doi.org/10.1145/296399.296425>.
- [155] CONG J, LI Z, BAGRODIA R. Acyclic multi-way partitioning of boolean networks[C/OL]//31st Design Automation Conference. 1994: 670-675. DOI: 10.1145/196244.196609.
- [156] NETO W L, AUSTIN M, TEMPLE S, et al. LSOracle: a logic synthesis framework driven by artificial intelligence: Invited paper[C/OL]//2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). 2019: 1-6. DOI: 10.1109/ICCAD45719.2019.8942145.
- [157] SCHLAG S, HEUER T, GOTTESBÜREN L, et al. High-quality hypergraph partitioning[J/OL]. ACM J. Exp. Algorithmics, 2023, 27. <https://doi.org/10.1145/3529090>.
- [158] WANG X, YANG M, LI Z, et al. Parallelized technology mapping to general plbs by adaptive circuit partitioning[C/OL]//2021 International Conference on Field-Programmable Technology (ICFPT). 2021: 1-5. DOI: 10.1109/ICFPT52863.2021.9609877.

- [159] HOSNY A, HASHEMI S, SHALAN M, et al. DRiLLS: Deep reinforcement learning for logic synthesis[C/OL]//2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC). 2020: 581-586. DOI: 10.1109/ASP-DA C47756.2020.9045559.
- [160] AMARUL, GAILLARDON P E, MICHELI G D. The epfl combinational benchmark suite[C]//International Workshop on Logic Synthesis (IWLS). 2015.
- [161] lsils/benchmarks: Epfl logic synthesis benchmarks[EB/OL]. <https://github.com/lsils/benchmarks/tree/master>.
- [162] İPEK E, MCKEE S A, CARUANA R, et al. Efficiently exploring architectural design spaces via predictive modeling[J/OL]. SIGOPS Oper. Syst. Rev., 2006, 40(5): 195–206. <https://doi.org/10.1145/1168917.1168882>.
- [163] GROSNIT A, MALHERBE C, TUTUNOV R, et al. BOiLS: Bayesian optimisation for logic synthesis[C/OL]//2022 Design, Automation & Test in Europe Conference & Exhibition (DATE). 2022: 1193-1196. DOI: 10.23919/DATES5 4114.2022.9774632.
- [164] BASAK CHOWDHURY A, TAN B, CAREY R, et al. Bulls-eye: Active few-shot learning guided logic synthesis[J/OL]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2023, 42(8): 2580-2590. DOI: 10.1109/TCAD.2022.3226668.
- [165] BROCKMAN G, CHEUNG V, PETTERSSON L, et al. Openai gym[A]. 2016. arXiv:1606.01540.
- [166] RAFFIN A, HILL A, GLEAVE A, et al. Stable-baselines3: Reliable reinforcement learning implementations[J/OL]. Journal of Machine Learning Research, 2021, 22(268): 1-8. <http://jmlr.org/papers/v22/20-1364.html>.
- [167] Cirkit: a logic synthesis and optimization framework based on various epfl logic sythesis libraries.[EB/OL]. <https://github.com/msoeken/cirkit>.
- [168] iMAP: Logic optimization and technology mapping tool[EB/OL]. <https://gitee.com/oscc-project/iMAP>.
- [169] TZIANTZIOULIS G, CHANG T J, BALKIND J, et al. Opdb: A scalable and modular design benchmark[J/OL]. IEEE Transactions on Computer-Aided

- Design of Integrated Circuits and Systems, 2022, 41(6): 1878-1887. DOI: 10.1109/TCAD.2021.3096794.
- [170] MURRAY K E, PETELIN O, ZHONG S, et al. Vtr 8: High performance cad and customizable fpga architecture modelling[J]. ACM Trans. Reconfigurable Technol. Syst., 2020.
- [171] ARORA A, BOUTROS A, RAUCH D, et al. Koios: A deep learning benchmark suite for fpga architecture and cad research[C]//International Conference on Field Programmable Logic and Applications (FPL). 2021.
- [172] Iwls 2005 benchmarks[C]//International Workshop on Logic Synthesis (IWLS). 2005.
- [173] Benchmark designs for the quartus university interface program (quip)[R]. Altera, 2005.

攻读学位期间研究成果

第一作者:

- [1] 2024 IEEE Transactions on Circuits and Systems II: Express Briefs. (在投)
- [2] 2023 IEEE International Conference on Field-Programmable Technology (ICFPT).
- [3] IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2022.
- [4] 2022 IEEE International Symposium on Circuits and Systems (ISCAS).

合著作者:

- [1] 第三作者, 2023 IEEE International Symposium of Electronics Design Automation (ISEDA).
- [2] 第四作者, 2022 IEEE International Symposium on Circuits and Systems (ISCAS).
- [3] 第三作者, 2021 IEEE International Conference on Field-Programmable Technology (ICFPT).