面向 6G 通信的可重构计算架构映射研究报告

清华大学-中国移动联合研究院

6G 研究中心

2021.12

批注 [c1]: 本名称与评审时对应。

建议基本思路是

1 介绍背景

2 介绍基于初步分析物理层，得出的初步架构和工具链

3 基于初步架构分析哪些模块适合 CGRA

4 分析部分关键模块的 CGRA 映射思路

5 得出结论：CGRA 可用于通信，但并不是每个算法都适合。目前的分析基于的架构是对通信算法总体评估后的初步设计，对适合 CGRA 的模块，后续将更深入优化架构与调度机制

请清华同学看下是否合理

目录

1 6G 展望及通信计算需求概述

当前，全球新一轮科技革命和产业革命正在加速演进，人工智能（AI）、VR/AR、三维媒体与物联网等新一代信息通信技术正在迸发，随之而来的就是海量的传输数据。如果说现在的 5G 是移动物联网时代，那么 6G 就是万物深度互联的智联时代，在那个真正万物互联的时代，无论是高山、大海还是沙漠、沼泽等一切偏僻的地方，都不会出现“失联”和“没信号”，甚至在飞机上人们可以无需担心地进行联网。

未来 6G 作为“新基建”的排头兵，随着智慧交通、泛在电力物联网、智慧城市等数字化应用蓬勃生长，一场伴随 6G 万物智联而来的“数据海啸”正滚滚袭来，而海量数据引发的电力消耗也将成倍增长。6G 硬件设备高能耗的僵局亟待破解。2018 年随着 5G 建设加快，推动了移动通信基站设备产量，2018 年移动通信基站设备产量达 43225.2 万射频模块，同比增长 58.7%。数据显示：2018 年中国增加移动电话基站 29 万个，总数达 648 万个，同比增长 4.68%；2019 年受物联网业务高速增长、基站需求增大影响，移动电话基站总数达 732 万个，同比增长 12.96%。基站数量显著上升。

1.1 国家、行业或中国移动在技术发展、业务应用发展方面存在的需求

1) 6G 研究已全面展开

经过三十年的努力，我国在移动通信行业实现了“1G 空白、2G 跟随、3G 突破、4G 同步、5G 引领”的跨越式发展。面向未来 6G 前沿技术的研究与实现，势必又是一场国际化的竞争与合作。2020 年 2 月，ITU-R WP5D 工作组第 34 次会议决定启动面向 2030 年及未来新一代移动通信（6G）的研究工作。3GPP 预计将于 2023 年启动对 6G 的研究，6G 国际标准化预计将于 2025 年左右启动。欧美日韩等相继启动 6G 技术的前期研究，我国也同步启动 6G 研究，2019 年 6 月工信部成立 IMT-2030 推进组，全面布局 6G 需求、技术、频谱研究，推动国际合作发展。2019 年 11 月，科技部成立国家 6G 技术研发推进工作组和总体专家组。中国移动于 2019 年、2020 年先后发布了 6G 系列白皮书：《2030+愿景与需求白皮书》、《2030+网络架构展望白皮书》、《2030+技术趋势白皮书》，分别系统地介绍了 6G 愿景与需求，展望了 6G 网络特征和架构，研判了 6G 潜在关键技术趋势，获得业界高度关注。

白皮书中指出，为满足 6G“数字孪生、智慧泛在”的愿景下多元化、个性化的业务需求与场景，未来网络将采用更加灵活的可重构架构设计，满足更高计算能力和可扩展需求，以

实现未来网络按需服务和最大化资源利用率的目标。同时，这对网络底层硬件提出了更高的要求：第一，6G 硬件需要具备“时间”与“空间”维度的高灵活性，支持软件控制下的快速、局部可重构，实现多业务共享相同硬件；第二，6G 硬件在可重构同时要具备极高的算力，从而满足业务对高性能网络需求；第三，6G 硬件还需具备良好的功耗水平，通过提高计算能效比满足节能降耗的绿色发展理念。

一般而言，满足上述需求要从半导体制造工艺与芯片架构两个层面入手。在国内半导体行业受制造工艺限制、国际半导体制造工艺发展放缓的大背景下，深入开展底层数字处理芯片架构层面的研究十分必要。

图 1.1 6G 网络可重构技术趋势

2) 行业愿景与运营商需求

当前，全球 5G 商用已全面启动，并将与众多垂直行业深度融合，提升经济社会各行业各领域的数字化、信息化和智能化水平，构建“万物互联”的新时代。然而当前全球新一轮科技革命和产业革命正在加速演进，通信感知计算一体化、人工智能（AI）、VR/AR、全息通信、数字孪生网络等科技前沿对未来 6G 网络的速率、时延、算力等提出了更高的需求，从而对网络设备底层硬件的灵活性、算力、功耗等也提出了更高的要求，但目前运营商网络难以支持。

运营商现网的基站多采用软硬件高度集成一体化的架构设计，基于标准化统一要求，采用定制化芯片开发软硬件各功能模块，紧凑的设计有利于在保证可靠性的前提下，最大化发挥其整体性能。但是，定制化芯片仅针对相对固定功能进行开发，不具有软硬件可重构性，

当业务与场景变化时，定制化芯片必须重新进行设计、制造、替换后才能支撑新业务需求，且定制化芯片开发周期和研发成本极高，产业环境相对封闭，使得该方案在当前以及未来面临着诸多挑战。具体表现如下：首先，随着 5G+行业应用的快速推进，行业客户定制化、多样化、快速部署的需求与高度集成化的基站设计之间的矛盾开始显现。其次，经过多年的建设和升级，现网存在多模多频的多层网络需要同步维护和联合协调优化，维护和优化的复杂度和成本持续提升，而各自封闭的基站系统在一定程度上掣肘了网络性能的进一步发挥。另外，随着移动通信频率上移、带宽和收发通道数量的大幅增加，基站功耗成倍增加，电费不仅成为运营商的运营之痛，甚至已成为行业的公共话题。

这一系列的挑战在现有方案的演进路线上很难突破。制造工艺角度，随着半导体制造工艺逐步接近极限，单纯通过增加芯片数量和制程演进已无法满足芯片计算速度提升带来的功耗的急剧上升和灵活度要求，且我国半导体制造行业还一直面临国外的技术封锁。芯片架构角度，为了实现基站信号处理的高速、高效和低功耗，目前设备的核心数字处理芯片多采用深度优化的定制化芯片（ASIC），其不具有灵活性；而以 CPU 芯片为代表的通用处理器，虽然能够满足 6G 高灵活性的要求，但其高算力下的时延与功耗水平并不理想。此外，以 FPGA 为代表的可编程硬件虽然兼具灵活性和高算力，但其无法实现快速的、部分的可重构能力，且功耗水平相对 ASIC 较高。而可重构计算架构芯片 CGRA 既具有灵活的可编程性，又可在提升性能的同时降低功耗，提高计算的能效比，为未来移动网络提供了另一个硬件平台的选择路径，值得加大投入，深入研究。

可重构芯片具备软硬件可编程的特性，硬件架构和功能随软件变化而动态变化，打通了“应用定义软件、软件定义芯片”进而实现“应用定义芯片”这一人们长期追求的通道。以此为基础，以期实现 6G 基站功能的可重构，使 6G 基站具有按需定义和部署软件功能的能力。在实现网络开放、灵活和快捷的部署与升级的同时，兼顾 6G 基站的成本、性能和功率效率，助力运营商在未来的商业竞争中保持竞争力。

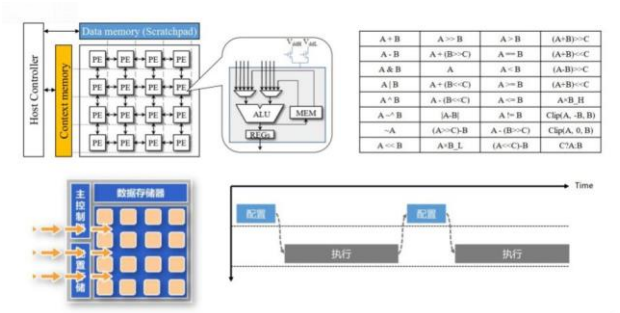

图 1.2 可重构计算芯片基本机构与配置机制

1.2 存在的科学技术问题

基于以上国家和行业的重要需求，本项目提出面向 6G 可重构计算架构设计问题：

1) 灵活性设计

针对未来网络支持端到端按需服务和网络升级维护的灵活性需求，以及通信感知计算一体化高算力需求，网络设备底层数字处理芯片需具有充分的灵活性和高计算密度，可重构计算架构具有粗颗粒度的软硬件可重构能力，但需要针对通信计算的关键算法，开展针对性的计算架构设计、验证与优化。这部分工作在国内外均处于单点研究阶段，系统级设计与验证亟待突破。

2) 高能效计算设计

随着半导体制程工艺逐步接近极限，传统数字处理芯片面临性能墙和功耗墙。且国内供应链受制程工艺限制，可重构计算芯片从芯片架构的重新设计入手，在国产制程工艺条件下，以期实现与传统处理芯片可比拟的能效比。

1.3 现有相关研究基础与局限

1) 相关研究基础

在 2000 年前后，国际上可重构计算的研究出现了一次热潮，大量可重构计算器件被设计和应用，如 PACT 公司的 XPP-III、IPFlex 公司的 DAPDNA-2、IMEC 的 ADRES 处理器等。在国内，西北工大、中科大和清华大学等也同步开展可重构计算架构的研究。这些研究充分证明了可重构架构在信号处理领域的优越性。2015 年，《国际半导体技术路线图》（ITRS 报告）将粗粒度可重构架构视为未来最有希望的可编程计算架构。2017 年，美国国防部高

级研究计划局（DARPA）发起了“电子复兴计划”，大力布局“软件定义硬件”研究计划，研究可重构架构。目前，动态可重构架构已经成为处理器体系结构领域最受关注的研究热点。近年 ISCA、MICRO、HPCA 等体系结构顶级会议上相关研究工作层出不穷。例如，威斯康辛大学麦迪逊分校和高通公司联合提出的 LSSD 架构能同时支持近似计算、卷积、流处理、深度学习等多种领域，并且取得接近每个领域专用加速器的性能和能效；斯坦福大学提出的 Plasticine 架构支持并行编程模型 Parallel Patterns，提升了可重构架构的可编程性；以色列理工学院提出的可重构架构支持计算阵列在线程内交互，避免了不必要的读写存储器操作；康奈尔大学提出的 ParallelXL 架构支持动态任务生成和调度，并实现了动态负载均衡。这些工作体现了可重构架构在跨领域应用、多任务并行和多模态处理上的巨大潜力。

未来 6G 网络同样也是智慧内生的通信、感知、计算一体化的网络，随着基于神经网络的人工智能算法的崛起，神经网络计算芯片的研究出现了井喷。其中，中科院计算所开发的 DianNao 系列是属于指令驱动的专用处理器（Application-Specific Instruction set Processor，ASIP），其计算单元配置非常灵活，但是每一步操作都需要精确的指令来控制存储器访问和具体的计算单元的操作类型，导致架构与片外存储交互频繁，使得性能和能效往往受限于总线带宽。随后，空间计算架构逐渐成为神经网络加速器的主流。这是由于空间计算架构提供了大量计算资源和丰富的互连，以及相对简洁的控制机制，非常适合神经网络的计算模式，如麻省理工的 Eyeriss，比利时鲁汶大学的 Envision。在空间计算架构之后，神经网络算法的高速演进又推动了神经网络计算可重构化的趋势。为了适应日新月异的网络拓扑结构、网络层类型和网络压缩技术，ISCA、HPCA 等顶级会议上有大量的神经网络计算工作均采用可重构空间计算架构。其中，FlexFlow 和 SCNN 采用可变的数据流，来适应不同网络的负载和稀疏度，以减少访存，提高能效；Bit fusion 采用 bit brick 技术来实现计算单元位宽可变能力；Ganax 有效结合 SIMD 和 MIMD 两种工作模式，取得了比 Eyeriss 更高的性能和能效；SCALEDEEP 和 HyPar 则同时支持神经网络推理和训练。

可重构处理器要求重配置速度高于 FPGA，功耗要远低于 FPGA，而效率又要接近 ASIC。可重构处理器不仅要求片上的运算和存储可以按需重构，外部存储子系统也要求能够被配置为不同的数据访问模式；面向高级语言的动态软硬件编译器在 runtime 时的优化同时包括了软件和硬件，编译器需要能够把高级语言转换为机器语言以及面向应用的硬件配置。可重构计算可以满足未来 6G 的通信算法以及人工智能计算的应用需求。

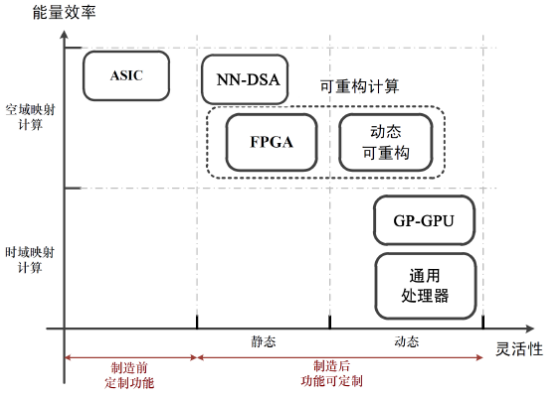

图 1.3 可重构计算芯片特性

2) 现有技术局限性：

由于所需支持的 6G 典型算法具有粒度混合的运算精度、复杂嵌套的数据依赖、串并耦合的计算流等特点，以往的可重构芯片的编程维度不足、计算流调度能力有限。我们需要提升可重构芯片的编程维度、计算流调度能力，使其可以精确、充分地调度和分配硬件资源，避免资源闲置或浪费，提高计算能效。

同时，6G 典型算法具有规则的数据局部性和非规则的数据稀疏性，传统可重构芯片的硬件结构难以兼顾这两个特点，我们需要在设计时结合数据局部性和稀疏性的特点设计硬件架构，充分利用硬件层面的优化空间，提升计算和访存效率。

传统可重构架构中，算法到硬件的跨层联合优化能力还有待提升，我们要提升整个工具链的灵活性和联动性，充分利用芯片的潜在算力和低功耗设计，提升计算能效。

2 CGRA 架构设计

研究团队基于对通信处理要求的分析，对 CGRA 的架构进行了初步设计。初步的方案设想主要从两个维度来实现可重构计算阵列：计算单元 PE（Process Element）、PE 单元组成的阵列 PEA（Process Element Array），两个维度合起来构成了可重构处理器 RPU（Reconfigurable Process Unit）。

2.1 可重构计算单元 PE 设计

1、单元架构

PE 单元是可重构处理器中负责单个算子计算的元素，它的结构包括了控制模块 Ctrl、数据接口 Interface、本地寄存器 LR、执行单元 FU。设想架构如图所示：

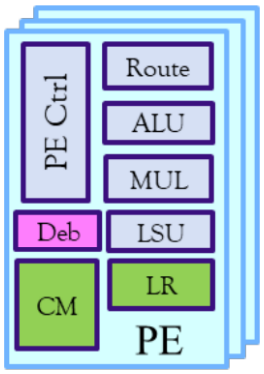

图 2.1 PE 单元结构图

2、控制模块 Ctrl

控制模块主要由指令寄存器、指令译码器构成，从外部送入配置信息，由控制模块负责 PE 操作类型的控制。指令（配置信息）主要包括操作码 opcode——经过译码后使能算术、逻辑、循环、条件等某种算子对应的计算电路，驱动 PE 完成计算，还包括数据的来源、输出——译码后驱动 MUX 相关电路选择数据输入输出方向。

3、数据接口 Interface

数据接口是 PE 与其它 PE 互联、与内存进行数据通信的桥梁。根据初步设想的 mesh 网格互联方案，这里 PE 需要配置四个方向的互联接口——上下左右，各包含一条 32bit 的输出通道、一条 32bit 的输入通道，后续实际设计过程中可能加入创新的方法——对不同位宽的数据进行标记，在 32bit 位宽上并行传输多个窄位宽的数据。此外，数据接口还包括 Load/Store 单元 LSU，负责与内存的数据通信，主要完成计算地址、向内存发送地址、数据、取数的操作。数据接口通过 MUX 与执行单元相连，在控制模块的控制下完成数据的输入、计算、输出。

4、本地寄存器 LR

PE 在计算过程中，可能会有一些中间结果暂存在 PE 内部，在后面的计算周期中重新

被 PE 使用，所以为了避免中间结果到内存的频繁搬运，本方案计划在 PE 内部放入存储单元——本地寄存器 Local Reg，用于存放 PE 的中间计算结果。

5、执行单元 EU

执行单元负责实际的计算操作，预期方案的操作类型包括加法、减法、乘法、乘累加、逻辑运算、循环控制、条件控制、路由等多种类型，不同的运算类型有着不同的延时，由专门的寄存器记录延时，保证数据输入输出时序的正确性。EU 由控制模块译码指令后直接驱动执行。

2.2 可重构计算阵列 PEA 设计

1、阵列架构

目前设想的阵列方案如下：PEA 是可重构处理器中负责统一映射、协同计算的完整模块，是可重构计算系统的主体，多个 PEA 可以互联扩展成更大的 PEA，用于完成模式图更大、更复杂的计算。PEA 通过外部接口如协处理器接口、内存接口与主处理器、内存进行通信，协同完成功能，主处理器主要将不适合 CPU 而适合可重构处理器计算的内容交付给 PEA，在 PEA 上计算完成后送回结果到内存，供主处理器使用和处理。PEA 总体架构如下：

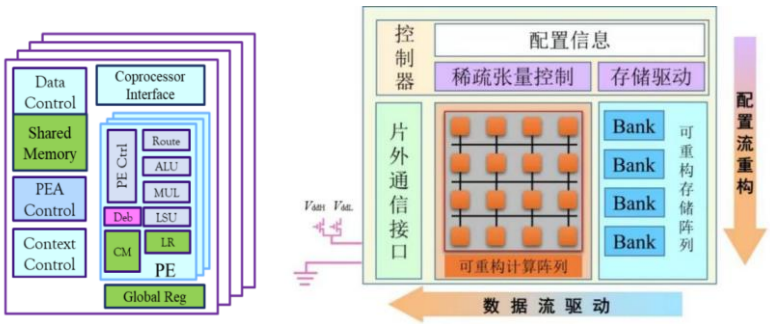

图 2.2 PEA 总体架构

2、PEA 内部互联

PEA 内部有多种可能的互联方式，一种最为常见的是 n*n 的阵列上，每个 PE 与上下左右四个 PE 直接连通，相距较远的 PE 之间通信需要经过其它 PE 的路由。

3、PEA 的外部接口

PEA 阵列完成计算需要受到主处理器的控制，而获取和存储数据则要拥有和内存通信的权限，因此，计划设计协处理器接口负责与主处理器的通信，设计配置总线控制器、数据总线控制器来协助与内存的通信，此外，为了提高 PEA 内部对一些频繁使用的数据获取的高效性，加入 PEA 的共享数据存储器，与 PEA 连通，相比内存具有更快的通信速度。设想架构如下：

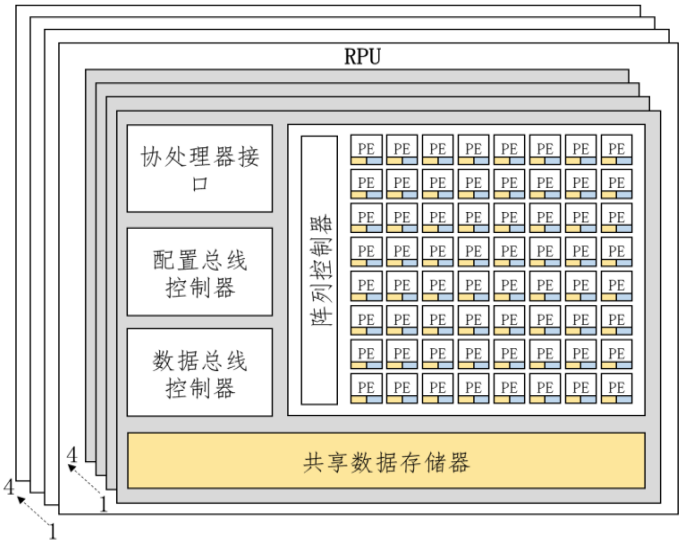

图 2.3 RPU 总体结构

2.3 可重构计算系统设计方案

完成可重构处理器主体部分的设计后，还需要考虑包含其外围电路的设计，以及包含可重构处理器的 SoC 设计。

可重构处理器 SoC 设计方面，其系统的总体架构预计如下图所示，主要包括 RISCV CPU 作为主处理器，SRAM 作为数据和指令内存存储 SoC 运行所需要的指令和数据信息，PEA 构成的 RPU 可重构处理器作为协处理器，完成一些特定的任务。CPU 与 RPU 之间有协处理器接口，SRAM 与 CPU、RPU 之间有数据、指令总线及相关控制器，CPU 通过协处理器接口将指令配置信息发送给 RPU，在 PEA 阵列上完成计算，PRU 通过与 SRAM 的总线互联，获取计算数据和存回计算结果。

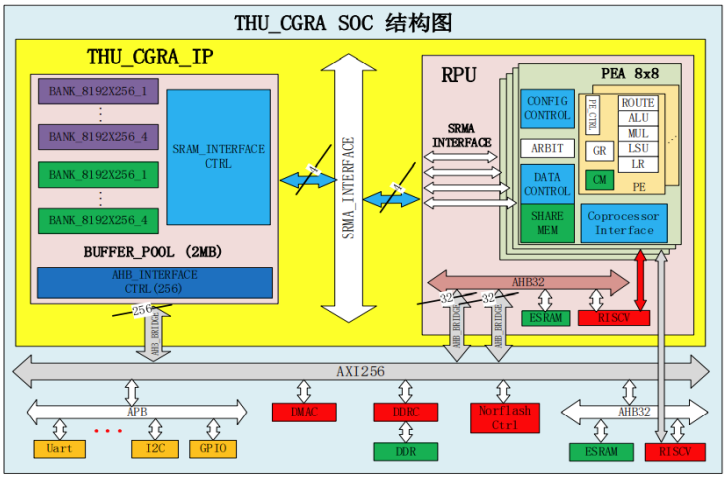

图 2.4 SOC 总体结构

3 工具链设计

3.1 编译器设计

通过资料调研，目前项目中的面向 CGRA 编译器的整体思路如下：

1.将用户的应用程序通过可重构编译器的 LLVM 前端通用的词法、语法和语义分析后，提取中间表达式 IR (IR. Intermediate Representation)。

2.对中间表达式 IR 进行优化后提取合适的数据流图。

3.任务划分和存储分配。

4.算子调度和映射工作，最终生成该应用程序需要在可重构处理器硬件上进行加速计算的二进制配置信息~~(Context Bitcam)~~。

批注 [c2]: 这里笔误还是专有名词

目前已掌握利用 LLVM 前端生成 IR 并对 IR 进行优化的方法。

3.2 模拟器搭建

目前已初步搭建好基于 C++语言的面向该 CGRA 架构的仿真测试平台，该仿真平台整体可分为两个子模块：1.PE 模块。主要包含各种算术运算和数据的传输。2.指令配置模块。

主要为各 PE 装载指令，指导 PE 的行为。

通过已经映射好的算法，可将每个 PE 的指令输入指令配置模块中，为 PE 装载指令进行各种运算从而完成整个应用算法。并且为各操作加入时序信息，结合调试功能，最终可达到时序仿真的效果，从而更好地对算法从架构本身、PE 功能、映射调度等各个方面进行评估。

4 物理层典型算法在 CGRA 实现的具体分析

4.1 总体流程分析

移动通信网络主要包括核心网（CN）和接入网（RAN）两大部分。其中，接入网中，通信基站的物理层处理因其严格的时序要求与极低的延时要求，成为移动通信网络中的关键性能瓶颈之一。如前所述，由于 6G 移动通信网络更加严苛的要求，传统采用 FPGA 与 ASIC 实现物理层的硬件方案面临重大挑战。同时，CGRA 因其良好的性能、灵活的配置能力以及高效的算力，可以更好满足 6G 通信算法的需求。

本章将针对物理层的典型算法，探索各个模块是否适合在 CGRA 中实现，并研究其映射方法。需要说明的是，本章所验证的各模块均基于已有的仿真或实现代码，就算法本身而言，仅是具体功能模块的一种实现方式。本章研究的目的并非对算法本身进行优化，而是以各个模块的实现算法为基础，验证物理层处理在 CGRA 架构中实现的可行性，并遴选出更加适合在 CGRA 中实现的物理层模块。由于 CGRA 具有可重构的通用性，而本报告所研究的物理层模块算法均具有一定代表性，因此相关研究可作为蓝本，为下一步的 CGRA 架构优化提供参考。

首先，在物理层算法的选择方面，每一代移动通信技术在物理层方面均具有较大变化，但是，根据其承载数据的基本类型而言，物理层的处理可以分为几个基本的物理信道：

1. 下行数据信道：完成基站到终端的用户数据传输，在 NR 中对应 PDSCH。

2. 上行数据信道：完成终端到基站的用户数据传输，在 NR 中对应 PUSCH。

3. 随机接入信道：完成终端随机接入流程，在 NR 中对应 PRACH。

4. 广播信道：完成网络广播功能，在 NR 中对应 PBCH。

5. 下行控制信道：完成基站到终端的控制数据传输，在 NR 中对应 PDCCH。

6. 上行控制信道：完成终端到基站的控制数据传输，在 NR 中对应 PUCCH。

在上述各信道中，尤以承载用户数据的 PDSCH 与 PUSCH 的数据处理要求最高。尤其是 PDSCH，由于基站需向多个终端发送数据，其数据处理要求最为严苛，是系统关键瓶颈。因此，本章针对下行数据信道，探讨物理层处理在 CGRA 中的映射。具体算法方面，4G 与 5G 物理层协议中，OFDM 与 MIMO 技术因优秀的性能成为核心的物理层技术。在 6G 时代，相关技术因其较高的成熟度，仍然将扮演重要的角色。因此，本章仍针对该技术路线的物理层协议进行分析，对相关技术的验证对未来移动通信网络的研发也有启发意义。

在讨论算法前，需要明确，并非所有物理层流程均适合在 CGRA 实现。由于 CGRA 主要采用多个 PE 并行处理进行加速，除了基础算法、算子等方便，还需要在以下方面对各模块考察：

1. 数据规则程度：CGRA 由于结构特征，多针对特定位宽的数据进行处理。因此，处理数据的规则程度直接影响到 CGRA 处理效率。例如，物理层中的部分比特流处理需要针对每一比特进行计算，而 CGRA 每次均可处理多比特数据。因此，如要采用 CGRA 加速，则需要用多比特位宽表示单比特数据，或需要对原始比特流进行分块封装再采用 CGRA 加速，这都会带来系统复杂度增加。

2. 并行程度：如前所述，CGRA 采用多计算单元并行化的设计。因此，算法的并行程度直接影响到 CGRA 加速的效率。对于并行度较好的代码，尤其是具备流水线特征与流式处理特征的算法，CGRA 可以更好起到加速效果。

3. 计算密集度：CGRA 将大量面积用于 PE 等计算单元的设计，相对于传统的通用 CPU，具备更好的数据计算能力。相应的，对于计算密集型应用，CGRA 中的 PE 可以充分被利用，发挥其优势。对于控制型应用在 CGRA 中的执行，则无法充分利用计算单元的潜力。

结合以上分析，一个典型的 PDSCH 流程如下图：

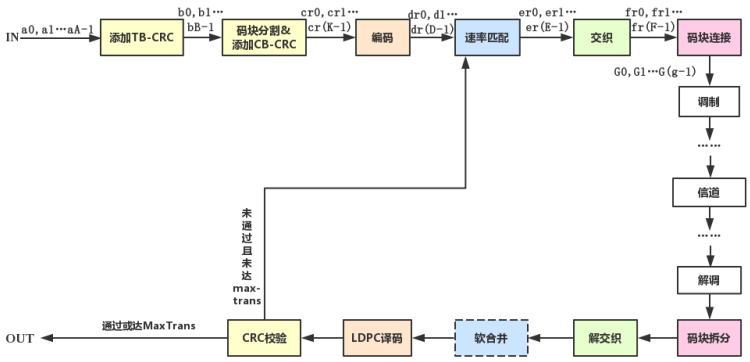

图 4.1 数据信道典型流程

如图所示，数据信道的输入信息，为经高层协议栈处理后生成的传输块 TB。为集中分析物理层算法的映射，这里假设单用户单 TB 的情况。其中，TB 为长度 A 的二进制比特流。

（1）CRC 处理：根据 TB 长度，CRC 编码模块需判定采用 CRC24A 或者 CRC16，对原始 TB 添加 CRC 部分，即输出长度为 B=A+24 或 16 的比特流。该模块需要对 TB 中的每一比特进行处理，具有一定计算密集度。在后续的码块分割完成后，多个码块还需要增加第二层 CRC，因此，其也具备一定并行度。但是，由于 CRC 属于比特级处理，且仅涉及移位与异或操作，其可以在传统 CPU 中进行串行处理。若需要在 CGRA 中处理，需对原代码进行一定优化。后续章节会探讨 CRC 在 CGRA 的实现。

（2）码块分割：该部分主要完成对长度为 B 的比特信息的切割。这里，以 NR 协议为例，根据协议，首先确定 BG 和 Kcb 取值。判定标准如下：

如果 $A \le 292$，或 $A \le 3824$ 且 $R \le 0.67$，或 $R \le 0.25$；选择 BG2。否则选择 BG1。

计算 Kcb 中，对于 BG1，Kcb=8848；对于 BG2，Kcb=3840。

通过比较 B 与 Kcb 取值，可确定码块分割与否并对分割后的码块添加二层 CRC24B。

此外，该部分还需要根据 TB 特性，确定 LDPC 编码使用的 lift factor Zc 的取值，以及最终的码块长度 K，并将分割后的码块填入 filler bit。

从以上描述可知，该部分主要涉及大量的条件判断以及一部分移位操作，实际数据计算需求并不多。此外，该部分针对各种长度的比特流信息进行操作，数据规整性很差。因此，

使用 CGRA 映射相关模块，无法充分发挥芯片能力。因此，该部分不建议通过 PEA 运行。

（3）编码：该模块输入为码块长度 K 比特流，而输出为 D 比特数据流（K+校验比特）。具体校验比特长度又和码率配置、BG 选择等信息有关。对于 LDPC 与 Polar 编码而言，虽然其仍然是比特级操作，但是由于具有较为规则的流程，以及涉及矩阵操作，可以采用 CGRA 实现加速效果。

（4）速率匹配：对于数据信道，需要通过速率匹配获得相应目标码率。同时，物理层多通过混合重传方式，提高物理层信息传输可靠性。以 LDPC 为例，输出 D 比特流经过速率匹配，与配置码率对应。同时，编码后的 D 比特将暂存于环形 buffer 结构中，并配合 HARQ 流程，进行多次传输，通过增量冗余提高译码性能。该部分主要涉及对存储结构的操作和条件分支，并不涉及大量计算。因此，不适合在 CGRA 中运行。

（5）交织：本模块主要通过输入比特与读出比特顺序的差别，降低比特之间相关性，提高传输可靠性。由于该模块并不涉及实际计算，也不适合在 CGRA 中运行。

（6）码块连接：本模块将所有处理完成后的并行比特流变为串行比特流，不涉及计算，因此不适合在 CGRA 中实现。

（7）调制：本部分完成比特流到符号的映射。目前，使用最多的调制方式为 QAM 调制，包括从 BPSK 到 256QAM。从逻辑上分析，本部分主要包括两步骤，即根据调制阶数对比特流进行串并转换，再根据对应调制星座图的映射关系，完成比特流到符号的映射。对于比特流的串并转换，主要为条件分支，不涉及计算。对于比特流到符号的映射，可通过预先生成映射表，直接查表完成，也不涉及密集运算。因此，该部分不适合在 CGRA 中实现。

（8）iFFT：此步骤主要用于生成 OFDM 符号，iFFT 作为一种高效算法广泛应用于频域到时域的变换中。其中，iFFT 输入为固定位宽的频域 IQ 对，一般取 16bit，输出也对应 16bit 的时域 IQ 对。iFFT 点数与具体带宽配置、子载波间隔等有关，且必须为 2 的整数次幂。由于 iFFT 基于蝶形结构运算，具有较高的并行性，结构规整，因此很适合在 CGRA 中实现。作为 OFDM 系统的重要组成部分，在 6G 移动通信系统中也将占重要地位，是验证分析的重点。

经过 iFFT 处理后，获得时域符号。此后，时域符号还将经过预编码、端口映射、中射频处理、数模转换、放大滤波等流程，通过天线将信号发送至无线信道，再经由接收天线，

重新得到模拟电信号。后续信号经过滤波、放大、模数转换，采样获得数字信号，再经过中射频处理，通过信道估计，重新获得对应时域信号。此后，信号进入基带接收流程：

（9）FFT：此步骤主要用于将时域 OFDM 信号变换为频域数据。FFT 为 iFFT 互逆过程，两者算法结构高度类似，只是在具体旋转因子的计算等方面需要改动。因此，该模块与 iFFT 可以合并为同一模块进行验证。

（10）解调：本部分是调制过程的逆过程。其输入为频域 IQ 数据对，根据不同调制方法，依据最大似然准则，获得每个比特的似然比信息。在实际实现中，多采用一定比特的定点数表示对应似然比。由于该部分根据不同调制方式，可以预先计算出对应似然比公式，只需要少量运算即可，可以根据实际情况确定是否在 CGRA 中运行。

（11）码块拆分：该部分用于将之前合并的码块进行重新拆分，分别进行后续处理。由于此部分为码块合并的逆过程，主要涉及不同比特的分流，为条件分支，不适合在 CGRA 中处理。

（12）解交织：该部分为交织的逆过程，主要对数据进行顺序调整，并不涉及密集计算，不适合在 CGRA 中实现。

（13）软合并：该部分主要是配合 HARQ 流程，对多次接收的 RV 版本进行增量冗余合并，提高译码的性能。此部分主要涉及用于缓存接收数据的存储结构，以及对应比特似然比的叠加。由于此部分不涉及复杂计算，可以视实际情况在 CGRA 进行映射。

（14）译码：此部分用于将多个比特的似然比信息进行译码，从而确定每比特信息，完成判决。目前，较为常见的为 LDPC 与 Polar 两种信道编码，其译码方式虽然未进行标准化，但由于数据处理要求很高，是系统关键瓶颈之一。由于其译码算法都涉及多次迭代，且对应数据较为规则，因此很适合在 CGRA 中进行加速。以 LDPC 常见的译码方式 min-sum 近似译码算法来看，多个比特的似然比在存入校验矩阵后，需要迭代进行多次信息节点与校验节点信息传递，对应需要在矩阵中计算符号、最值、加和等操作，通过 CGRA 处理可以发挥其并行化优势。

（15）CRC 校验：此部分主要通过 CRC 校验，确定对应比特流是否正确接收，从而保证发送比特正确性。由于此部分涉及计算量较低，可以根据实际情况确定是否在 CGRA 处理。

综上，根据每部分算法的特征，编译码算法、傅里叶变换由于规则的数据格式以及规整的算法流程，建议作为重点模块在 CGRA 中进行映射；而 CRC、解调等部分，对计算要求相对较低，可以根据系统设计确定是否在 CGRA 中实现；码块分割、速率匹配等部分，由于主要涉及条件分支等控制流程，不建议通过 CGRA 实现，可以在 SoC 中的 RISC-V 或者通过系统总线连接通用 CPU 实现。

以下几章，将分别针对实现代码，讨论关键模块的映射思路。

4.2 CRC 编码

LDPC 编码过程中存在两次 CRC 编码，第一次是在编码第一步对需要发送的信息码 data_in 进行 CRC 编码。如果信息码的长度大于 3824bit，则采用 CRC24A 编码；如果信息码的长度小于或等于 3824bit，则采用 CRC16 编码。第二次编码是在信息码进行码块分割后，对分割后的各段码块进行 CRC 编码，采用的编码方式为 CRC24B 编码。以上提到的三种 CRC 编码的生成多项式如下表所示：

编码方式	生成多项式	对应的二进制除数
CRC24A	$X^{24}+X^{23}+X^{18}+X^{17}+X^{14}+X^{11}+X^{10}+X^{7}+X^{6}+X^{5}+X^{4}+X^{3}+X+1$	1_1000_0110_0100_1100_1111_1011
CRC24B	$X^{24}+X^{23}+X^{6}+X^{5}+X+1$	1_1000_0000_0000_0000_0110_0011
CRC16	$X^{16}+X^{12}+X^{5}+1$	1_0001_0000_0010_0001

表 4.2.1 三种 CRC 编码的生成多项式及其对应的十六进制除数

接下来，我们首先讲述如何将 Matlab 形式的三种 CRC 编码方式转换为 C++代码形式，之后讲述如何将 C++代码形式的 CRC 编码方式映射到我们的 CGRA 阵列上。

4.2.1 CRC 编码的 C++代码实现

crc_calc_212.m 文件中的 Matlab 代码模拟线性反馈移位寄存器（LFSR）实现 CRC 编码。以 CRC16 编码为例，其代码和硬件原理图如图 5.2.1.1 和图 5.2.1.2 所示：

```
%%  gCRC16(D) = [D16 + D12 + D5 + 1]
  s=zeros(1,16);
  for i=1:len_data
      T2=mod(s(16)+data_in(i),2);
      s(16)=s(15);
      s(15)=s(14);
      s(14)=s(13);
      s(13)=mod(s(12)+T2,2);
      s(12)=s(11);
      s(11)=s(10);
      s(10)=s(9);
      s(9)=s(8);
      s(8)=s(7);
      s(7)=s(6);
      s(6)=mod(s(5)+T2,2);
      s(5)=s(4);
      s(4)=s(3);
      s(3)=s(2);
      s(2)=s(1);
      s(1)=T2;
   end
```

图 4.2.1.1 Matlab 实现 CRC16 编码

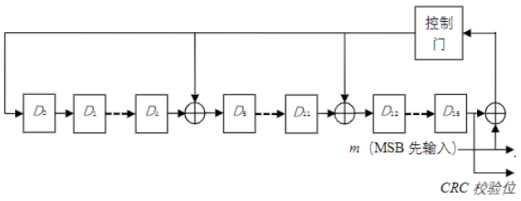

图 4.2.1.2 Matlab 实现 CRC16 编码的硬件原理

显然 C++代码也可通过模拟线性反馈移位寄存器实现 CRC 编码。但这个方法在 CGRA 映射时存在两个明显缺点：1）CGRA 单元每次运算只处理 1bit 数，效率太低，并且浪费了 CGRA 单元 32bit 的数据带宽；2）资源开销太大。实现 CRC16 编码需要 16 个寄存器和 3 个异或单元；如果实现 CRC24A 编码则需要 24 个寄存器和 14 个异或单元。这种资源开销显然不是 CGRA 映射所能容忍的。3）硬件配置方法缺乏通用性。显然要实现 CRC24A，CRC24B，CRC16 三种编码方式，就需要对 CRGA 阵列进行三种不同的配置。因此我们需要寻找一种更高效、更通用的算法来实现 CRC 编码，以实现对 CGRA 阵列的高效利用。下面，我们将介绍如何使用 C++语言模拟竖式计算来实现 CRC 编码。

设信息码为 M，生成多项式对应的除数为 P，P 的 bit 数为 n+1，CRC 编码相当于将信息 M 左移 n 位后再除以 P，得到 n bit 余数 crc（不足 n bit 就在高位补 0）。值得注意的是，在 CRC 编码的竖式计算中，所有的加法和减法都是不进位或不借位的，相当于是异或操作。求得的余数 crc 即为信息码 M 进行 CRC 编码后得到的校验码。即校验码 crc = (M<<n) % P。以 CRC16 为例，设信息码 M=0x84，长度为 1byte=8bit，生成多项式 X16+X12+X5+1 对应的除数为 0x11021。则求 M 的 CRC16 校验码的竖式计算过程如图 4.2.1.3 所示：

```
                                1000 1100
                ┌──────────────────────────
1 0001 0000 0010 0001 │ 1000 0100 0000 0000 0000 0000
                      1000 1000 0001 0000 1000 0000
                      ──────────────────────────
                      1100 0001 0000 1000 0000
                      1000 1000 0001 0000 1000
                      ──────────────────────────
                       100 1001 0001 1000 1000
                       100 0100 0000 1000 0100
                      ──────────────────────────
                      1101 0001 0000 1100
```

图 4.2.1.3 求 0x84 的 CRC16 校验码的竖式计算过程

显然这种竖式计算方法对于任意长度的信息码都具有通用性，但考虑到算法在 CGRA 阵列上映射时，CGRA 计算单元的数据线宽只有 32bit，因此我们选择将信息码按字节划分，每次处理 1byte 的数据。我们首先编写了求单字节的 CRC16 编码的 C++代码，封装为 C++函数 CRC16_1byte，如图 4.2.1.4 所示。但是，该函数存在一个缺陷：如果将其改为 CRC24 以上的 CRC 编码方式，则需要将输入的 8bit 数先左移 24 位以上，被除数总位数超过了 32bit。

此时在 CGRA 阵列上映射，由于计算单元的数据线宽只有 32bit，就会发生溢出。因此，我们对算法进行了适当优化，使得在数据线宽为 32bit，每次处理 1byte 信息码的条件下，CGRA 从最高支持 CRC24 编码，变为了最高支持 CRC32 编码。优化后的函数封装为 CRC16_1bte2 如图 4.2.1.5 所示。

```cpp
unsigned int CRC16_1byte(char data_in)
{
    unsigned int crc = data_in << 16;
    unsigned int ccitt16 = 0x11021;
    unsigned  cmp_value = 0x10000;

    ccitt16 <<= 7;
    cmp_value <<= 7;

    while (cmp_value >= 0x10000)
    {
        if ((crc & cmp_value) != 0)
        {
            crc ^= ccitt16;
        }
        ccitt16 >>= 1;
        cmp_value >>= 1;
    }
    return (crc & 0xFFFF);
}
```

图 4.2.1.4 C++函数 CRC16_1byte

```cpp
unsigned int CRC16_1byte2(char data_in) {
    unsigned int ccitt16 = 0x11021;
    unsigned int crc = (data_in << 8);

    for (int j = 0; j < 8; j++)
    {
        if (crc & 0x8000)
        {
            crc <<= 1;
            crc ^= ccitt16;
        }
        else
        {
            crc <<= 1;
        }
    }
    return crc;
}
```

图 4.2.1.5 C++函数 CRC16_1byte2

在实现了单字节的 CRC16 编码的基础上，多字节的 CRC16 编码很容易便可实现。假设

我们现在需要求信息码 M=0x8423 的 CRC16 编码。我们首先将 M 分为 0x84 和 0x23 两个字节，当我们得到 0x840000 除以 0x11021 的余数（这里余数是 0xD10C）后，需要求 0x84230000 除以 0x11021 的余数，只要将原来的余数 0xD10C 左移 8 位后与 0x230000 相加，再除以 0x11021，就得到了 0x84230000 除以 0x11021 的余数，即 0x84000000 除以 0x11021 的余数加上 0x230000 除以 0x11021 的余数。根据这个方法，我们就可以写出处理多字节的 CRC16 算法。此算法被封装为了 C++函数 CRC16，如图 4.2.1.6 所示：

```
unsigned int CRC16(char data_in[], int size) {
    unsigned int ccitt16 = 0x11021;
    unsigned int crc = 0;
    for (int i = 0; i < size; i++)
    {
        crc ^= (data_in[i] << 8);
        for (int j = 0; j < 8; j++)
        {
            if (crc & 0x8000)
            {
                crc <<= 1;
                crc ^= ccitt16;
            }
            else
            {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

图 4.2.1.6　C++函数 CRC16

显然，上述实现多字节的 CRC16 编码的算法对于不同的 CRC 编码方式具有通用性。因此，我们对 C++函数 CRC16 略微改动，便可实现多字节的 CRC24A 编码和 CRC24B 编码。实现多字节的 CRC24A 编码和 CRC24B 编码的算法分别被封装为了 C++函数 CRC24A 和 CRC24B，如图 5.2.1.7 和图 5.2.1.8 所示：

```cpp
unsigned int CRC24A(char data_in[], int size) {
    unsigned int ccitt24A = 0x1864CFB;
    unsigned int crc=0;
    for (int i = 0; i < size; i++)
    {
        crc ^= (data_in[i] << 16);
        for (int j = 0; j < 8; j++)
        {
            if (crc & 0x800000)
            {
                crc <<= 1;
                crc ^= ccitt24A;
            }
            else
            {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

图 4.2.1.7　C++函数 CRC24A

```cpp
unsigned int CRC24B(char data_in[], int size) {
    unsigned int ccitt24B = 0x1800063;
    unsigned int crc = 0;
    for (int i = 0; i < size; i++)
    {
        crc ^= (data_in[i] << 16);
        for (int j = 0; j < 8; j++)
        {
            if (crc & 0x800000)
            {
                crc <<= 1;
                crc ^= ccitt24B;
            }
            else
            {
                crc <<= 1;
            }
        }
    }
    return crc;
}
```

图 4.2.1.8　C++函数 CRC24B

4.2.2 CRC 编码算法的 CGRA 映射

以 C++函数 CRC16 为例，其在 4*4 的 CGRA 阵列上的映射方式如图 4.2.2.1 所示

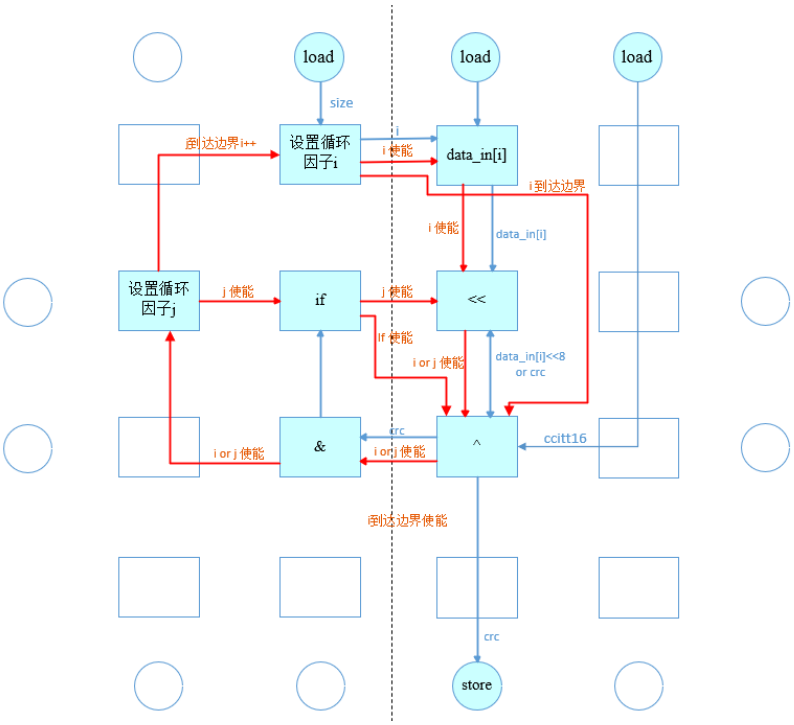

图 4.2.2.1　C++函数 CRC16 在 4*4 的 CGRA 阵列上的映射

上图中蓝色箭头代表数据流，红色箭头代表控制流。下面阐述 CGRA 阵列的工作原理。首先，单元（1, 2）从端口读入信息码字节数 size，将其作为循环变量 i 的边界。单元（2, 1）将 8（已在配置时寄存在 PE 里）设置为作为循环变量 j 的边界。之后，单元（1, 2）开始 i 的循环操作，将循环变量 i 传给单元（1, 3），i 使能信号驱动（1, 3）读入第 I byte 的信息码 data_in[i]。之后，将数据 data_in[i]传给单元（2, 3），i 使能信号驱动（2, 3）单元执行移位操作，将 data_in[i]左移 8 位（数据 8 已在配置时寄存在 PE 里）。之后，将左移 8 位的 data_in[i]传给单元（3, 3），同时（3, 3）单元中寄存着上一次 i 循环后得到的 crc（初始时为 0），i 使能信号驱动（3, 3）执行 data_in[i]与 crc 的异或操作得到新的 crc，并寄存在（3, 3）中。之后，将 crc 和 0x8000 传递给单元（3, 2），i 使能信号驱动（3, 2）执行&操作。之后，（3, 2）将 i 使能信号传递给单元（2, 1），驱动（2，1）开始 j 的循环操作。

j 的循环开始后，单元（2, 2）从单元（3, 2）读取&操作结果，j 使能驱动（2, 2）执行 if 操作。之后，将寄存在单元（3, 3）中的 crc 传给单元（2, 3），j 使能信号驱动（2, 3）单元执行移位操作，将 crc 左移 1 位（数据 1 已在配置时寄存在 PE 里）。之后，将左移 1 位的 crc 和从端口读入的被除数 ccitt16 传给单元（3, 3），j 使能信号和由（2, 2）出发，途径（2, 3）传来的 if 使能信号共同驱动（3, 3）执行异或操作得到新的 crc，并寄存在（3, 3）中。之后，将 crc 传递给单元（3, 2），j 使能信号驱动（3, 2）执行 crc 和 0x8000（已在配置时寄存在 PE 里）的&操作。之后，（3, 2）将 i 使能信号传递给单元（2, 1），驱动（2, 1）开始 j 的下一轮循环。如果 j 到达了循环边界 8，则终止 j 循环，并驱动单元（1, 2）执行 i 的下一轮循环。如果 i 到达了循环边界 size，则终止 i 循环，则使能信号经过（1, 3）和（2, 3）传递给（3, 3），驱动单元（3, 3）将寄存的 crc 输出进行 store。

4.3 LDPC 编码

在 LDPC 编码部分，其运算密集型部分为循环内的赋值与逻辑计算。

4.3.1 LDPC 编码的 C++算法实现

LDPC(低密度奇偶检验)码是由稀疏校验矩阵定义的线性分组码，具有能够逼近香农极限的优良特性，其描述简单，具有较大的灵活性和较低的差错误码特性，可实现并行操作，译码复杂度低，吞吐量大，极具高速译码的潜力，在码长较长的情况下，仍然可以有效译码。这些特点都使 LDPC 编码算法适合于在 CGRA 上实现。

LDPC 编码的主要过程和有相关性的模块如图 4.3.1.1 所示：

图 4.3.1.1 LDPC 编码过程流程图

其中编码的过程处于码块分割、添加 CB-CRC 码字之后，生成 encoder_bits，之后再进行后续操作。genLU 和 encoder_jia 函数共同完成 LDPC 的 LU 编码的过程。为了提高仿真速度，将生成 LU 矩阵和矩阵求解的过程分别置于两个函数中实现。每次完成 LU 分解（即 genLU）后即储存相应矩阵，以便之后遇到相同输入参量时可以直接调用。

LU 码需要用 Tanner 图表示，如图 4.3.1.2 所示。将 Tanner 图中的校验节点与比特节点的连接看成一个交织器的结构。

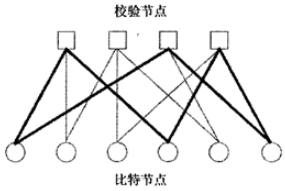

图 4.3.1.2 LU 编码 Tanner 图

考虑每个码由 p bits 组成，编码时将校验矩阵分为 A、B 两部分，其中 A 为(n-k)*(n-k)的满秩矩阵，码字则对应(t , s)形式，其中 t 为校验符号；s 为信息符号。由此可得

$$(A,\ B)\begin{pmatrix} t^T \\ s^T \end{pmatrix} = 0$$

即

$$t^T = A^{-1}Bs^{-1}$$

考虑到矩阵求逆的复杂性，为简化计算，将其中一个需要处理的码字空间矩阵 A 分解为 L、U 两个部分，其中 A=LU。其中 L 为下三角矩阵，U 为上三角矩阵，通过引入中间变量可以将方程化为以下两组：

$$Lv = Bs^T$$

$$Ut^T = v$$

利用 L 和 U 的三角结构，采用前后迭代的算法能快速算出方程的解，可避免矩阵求逆等效率较低的运算。以上介绍为 genLU 函数的主要功能。在得到校验矩阵 L 和 U 后，利用 encoder_jia 函数进行编码操作。

encoder_jia 函数主体内容主要分为以下几个方面，首先是赋值，用输入 Press_L1、Press_U1、Press_H1 对 L1、U1、H1 三个矩阵赋值，这三个矩阵在编码的过程中主要起到地址存储的作用，方便后续从矩阵中取得对应元素。具体内容如图 5.3.1.3 所示：

```
for (j = 0; j < max_L1; j++)
{
  for (i = 0; i < row_H; i++)
  {
    L1[i][j] = *(press_L1++);
  }
}

for (j = 0; j < max_U1; j++)
{
  for (i = 0; i < row_H; i++)
  {
    U1[i][j] = *(press_U1++);
  }
}

for (j = 0; j < max_H1; j++)
{
  for (i = 0; i < row_H; i++)
  {
    H1[i][j] = *(press_H1++);
  }
}
```

图 4.3.1.3 encoder_jia 赋值部分代码图

之后开始编码部分。其代码如图 4.3.1.4 所示，先将对应数组 b，x，c 赋值为 0，之后依次进行计算。由于计算过程比较类似，这里只介绍数组 b 的计算，其余内容较为类似，相同点再此不多赘述。b 矩阵元素计算过程为：从输入 info 矩阵中取出元素和 b[i]做自累加操作，其地址 h 从 H1 矩阵中依次提取。在内层循环结束之后，对 b 做二进制转换，之后存储，生成响应的二进制码流。最后将 b[0]的值赋给 x[0]，方便 x 的后续操作。

```
for (i = 0; i < row_H; i++)
{
  b[i] = 0;
  for (j = 0; j < Length_H1[i]; j++)
  {
    h = H1[i][j];
    b[i] = b[i] + (int)info[h - 1];
  }
  b[i] = b[i] % 2;
}
x[0] = b[0];
```

图 4.3.1.4 encoder_jia 编码部分代码图

对 x、c 矩阵的操作如图 4.3.1.5 和图 4.3.1.6 所示：

```
for (i = 1; i < row_H; i++)
{
  x[i] = 0;
  for (j = 0; j < Length_L1[i] - 1; j++)
  {
    l = L1[i][j];
    x[i] = (x[i] + x[l - 1]) % 2;
  }
  x[i] = (b[i] + x[i]) % 2;
}

c[row_H - 1] = x[row_H - 1];
```

图 4.3.1.5 encoder_jia 编码部分代码图

```
for (i = row_H - 2; i > -1; i--)
{
  c[i] = 0;
  for (j = Length_U1[i] - 1; j > 0; j--)
  {
    u = U1[i][j];
    c[i] = (c[i] + c[u - 1]) % 2;
  }
  c[i] = (x[i] + c[i]) % 2;
}
```

图 4.3.1.6 encoder_jia 编码部分代码图

其主体内容与 b 矩阵的赋值类似，第 5 个循环即对 c 操作的循环中是从 x 矩阵中取出元素和 x[i]做自累加操作，其地址 l 从 L1 矩阵中依次提取。在内层循环结束之后，先将 b[i]与 x[i]相加之后做二进制转换，存储到 x[i]，生成响应的二进制码流。最后将 x[row_H-1]的值赋给 c[row_H-1]，方便 c 矩阵的后续操作。

第 6 个循环中，从 c 矩阵中取出元素和 c[i]做自累加操作，其地址 u 从 U1 矩阵中依次提取。在内层循环结束之后，先将 c[i]与 x[i]相加之后做二进制转换，存储到 c[i]，生成响应的二进制码流。

到此，编码的过程基本结束，编码码流以矩阵 c 的形式存储。

最后一部分是将编码与原码整合。其代码如图 4.3.1.7 所示：

```
for (i = 0; i < row_H; i++)
{
    p = press_P[i];
    finalx[k + p - 1] = c[i];
}
for (i = 0; i < k; i++)
    finalx[i] = info[i];
```

图 4.3.1.7 encoder_jia 原码编码整合部分代码图

第 7 个循环中，首先从 press_P 矩阵中按序取地址 p，之后计算 finalx 对应地址，将 c 矩阵中的值赋值到 finalx 矩阵。

第 8 个循环中对 finalx 数组中第 1 到第 k 个元素进行操作，将 info 中对应的原码元素复制到 finalx 中。

至此，编码过程结束。

4.3.2 LDPC 编码算法的 CGRA 映射

在 LDPC 编码部分，其运算密集型部分为 encoder_jia.cpp 这个 C++文件中的循环内的赋值与逻辑计算。下面按照循环顺序依次对代码进行 CGRA 映射。

第 1 次循环的映射如图 4.3.2.1 所示：

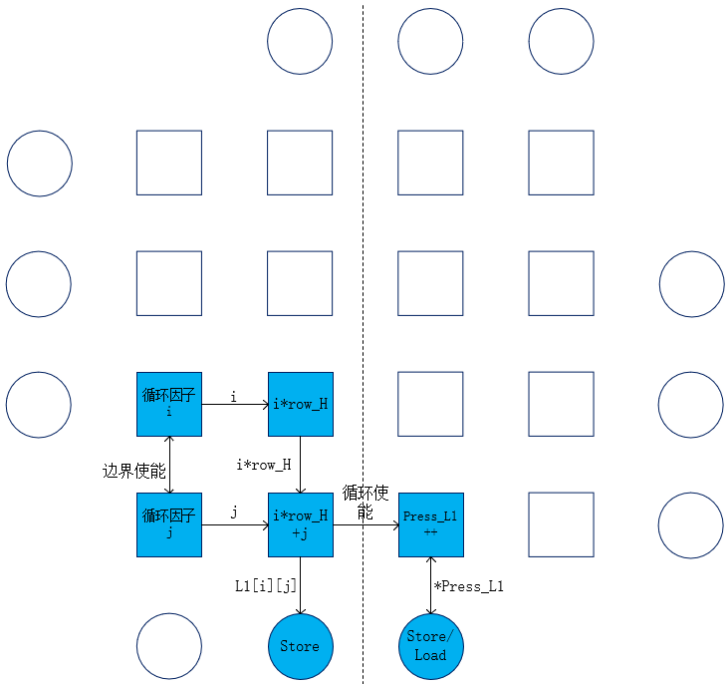

图 4.3.2.1 encoder_jia 编码部分第 1 次循环代码图

在映射图中其中圆形代表 load 单元和 store 单元，方形代表 PE 模块。其中第 1 个循环主要做的工作是访问指针 press_L1 指向的内存空间，读取其中的值，并将读出的值放入 L1 数组中相应位置进行存储。

在 CGRA 中数据流与控制流的走向具体介绍如下所示：由于代码结构为两层循环，内层循环的操作为数组赋值，因此先设置外层循环因子 j 和内层循环因子 i，j 给 i 一个循环开始的使能信号，循环因子将 i 传递给右边 pe 进行乘法运算得到 i*row_H 并自增，接着该 pe 将 i*row_H 传递给下方 pe，该 pe 负责加法运算，将结果给 LSU 作为 store 的地址，其同时给右边 pe 使能信号，让其将地址 Press_L1 传给 LSU 完成 load 数值和 store 数值的操作，此后地址 Press_L1 进行自增。当循环因子 i 到达循环边界，将使能发送给循环因子 j 使其自增并判断边界条件，此后重复以上操作直至整套循环结束。

第 2 次循环的映射如图 4.3.2.2 所示：

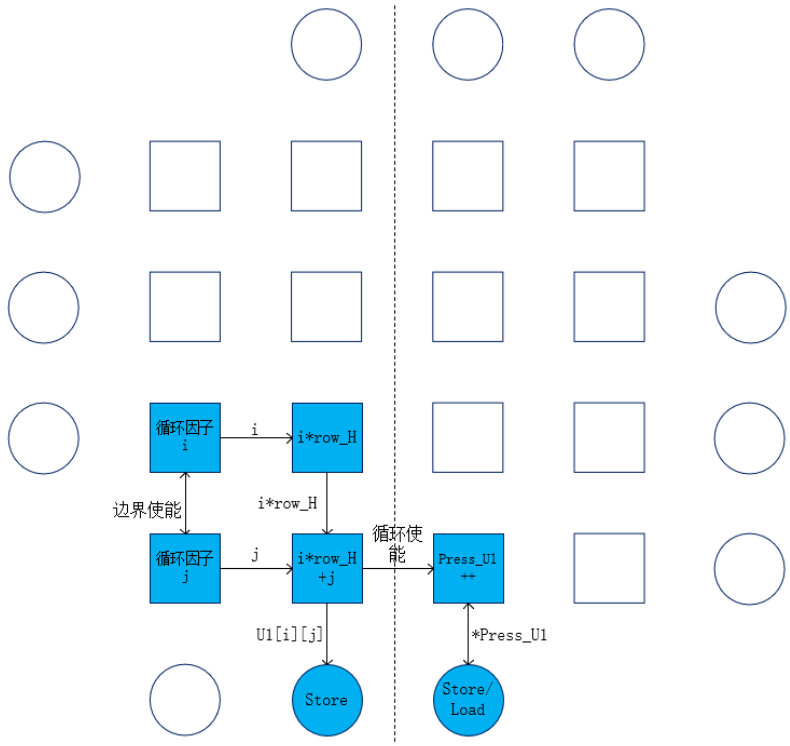

图 4.3.2.2 encoder_jia 编码部分第 2 次循环代码图

第 2 个循环主要做的工作是访问指针 press_U1 指向的内存空间，读取其中的值，并将读出的值放入 U1 数组中相应位置进行存储。

CGRA 具体映射与第 1 次循环基本一致，不再赘述。

第 3 次循环的映射如图 4.3.2.3 所示：

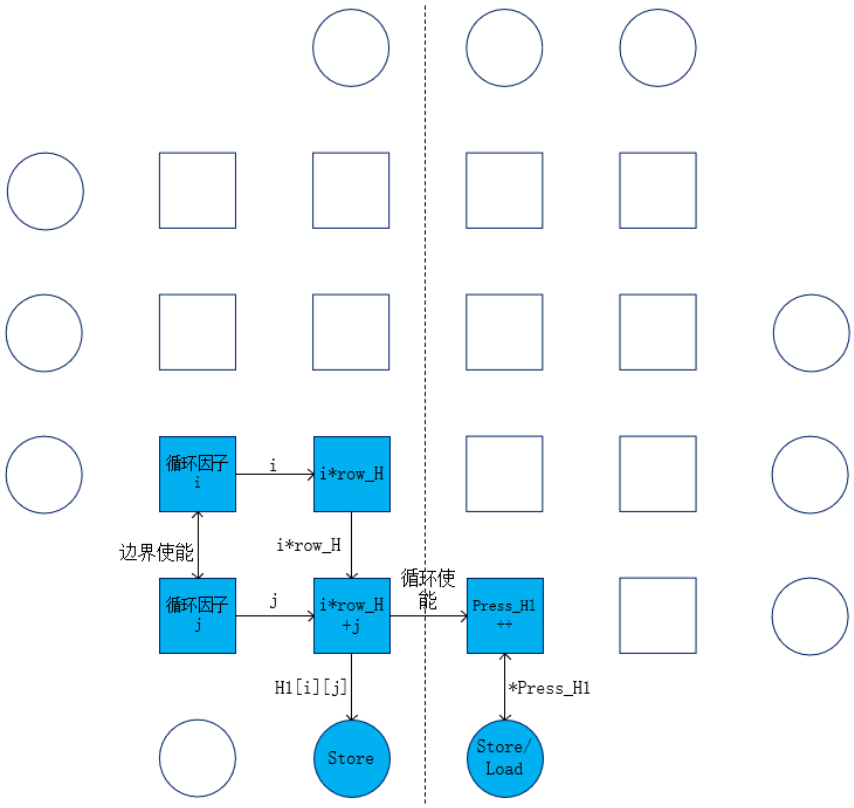

图 4.3.2.3 encoder_jia 编码部分第 3 次循环代码图

第 3 个循环主要做的工作是访问指针 press_H1 指向的内存空间，读取其中的值，并将读出的值放入 H1 数组中相应位置进行存储。

CGRA 具体映射与第 1 次循环基本一致，不再赘述。

第 4 次循环的映射如图 4.3.2.4 所示：

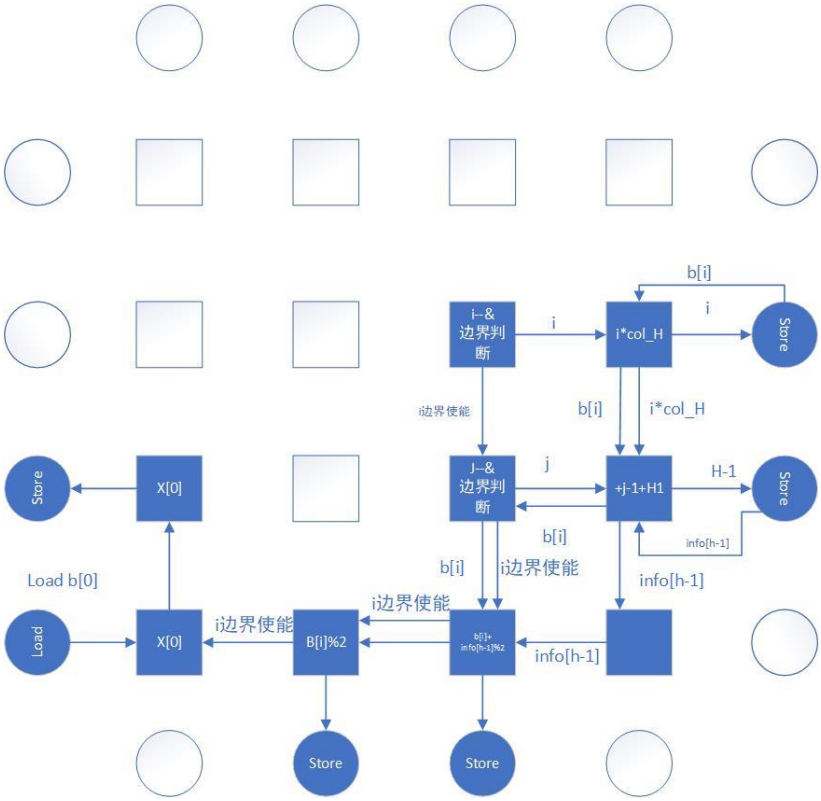

图 4.3.2.4 encoder_jia 编码部分第 4 次循环代码图

在映射图中其中部分圆形代表 load 单元，部分圆形代表 store 单元，方形代表 PE 模块。其中第 4 个循环主要做的运算是生成二进制码流 b，并以数组的方式存储。首先需要对循环边界 j 判断，如果满足条件，则对地址进行计算，之后在 store 单元中输出地址。之后从阵列中读出 info[h-1]的数据，之后计算得到的数据与对应的 b[i]进行计算，计算的使能为循环边界的 i 使能，对其进行模 2 计算，输出得到的 b[i]数据，生成二进制码流。在判断 i 循环是否结束之后，输出将 x[0]重新赋值为 b[0]。

在 CGRA 中数据流与控制流的走向具体介绍如下所示：首先对 i 进行自增操作，之后判断是否到达边界，将边界使能传到 j 自减及边界判断的 PE 和 i*col_H 的地址计算 PE，i*col_H 的地址计算 PE 计算得到地址输出到 Store 单元，之后读出 b[i]数据，经过多个 PE 传递至 b[i] +info[h - 1]的计算 PE。j 自减及边界判断 PE 将传递至 j-1+H1 计算 PE，之后输出地址，从 Store 单元中读出 info[h-1]的数据，传递至 b[i] +info[h - 1]的计算 PE，并完成计算，

批注 [c3]: 此处为何要求自减，不太明白

通过 Store 单元存储。之后将计算所得数据传递至模 2 操作 PE，通过 Store 单元输出计算结果，并存储 b 的二进制码流。左侧 Load 模块读入 b[0]具体数值，并将其赋值到 x[0]上，之后 i 边界使能经过传递，传递至 x[0]的赋值 PE，并由此判断是否进行传递和存储，如果满足 i 循环结束条件则传递至上方 PE，同时通过 Store 进行存储。

第 5 次循环的映射如图 4.3.2.5 所示：

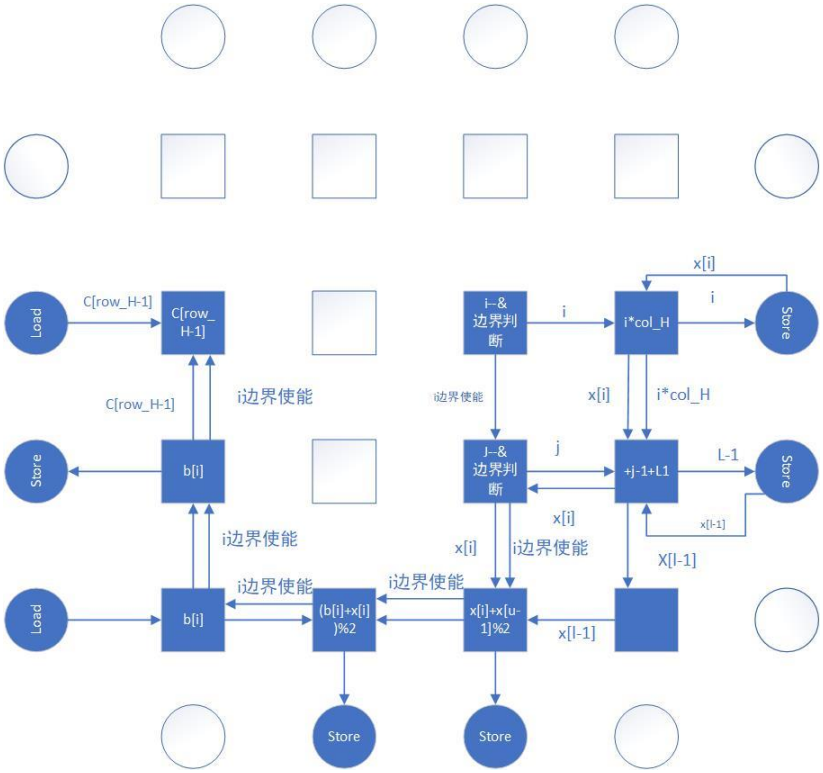

图 4.3.2.5 encoder_jia 编码部分第 5 次循环代码图

第 5 次循环，主要做的是生成二进制码流 x，并以数组的方式存储。首先需要对循环边界 j 判断，如果满足条件，则对地址进行计算，之后在 store 单元中输出地址。之后从阵列中读出 x[l-1]的数据，之后计算得到的数据与对应的 x[i]进行计算，计算的使能为循环边界的 i 使能，对其进行模 2 计算，输出得到的 x[i]数据，生成二进制码流。在判断 i 循环是否结束之后，输出将 c[row_H-1]重新赋值为 x[row_H-1]。第 5 次循环与第 4 次循环最大的差别就是 b[i]的读入，这个需要从 load 单元中读出 b[i]，之后对 x[i]进行地址计算和逻辑计算，

与 b[i]相加。

CGRA 具体映射与第 4 次循环基本一致，不再赘述。

第 6 次循环的映射如图 4.3.2.6 所示：

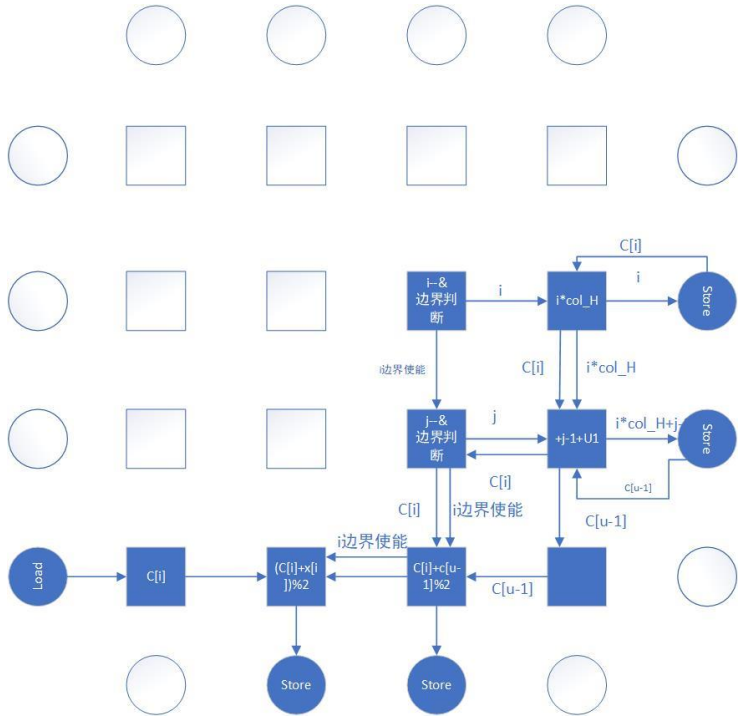

图 4.3.2.6 encoder_jia 编码部分第 6 次循环代码图

第 6 次循环，主要做的是生成二进制码流 c，并以数组的方式存储。首先需要对循环边界 j 判断，如果满足条件，则对地址进行计算，之后在 store 单元中输出地址。之后从阵列中读出 c[u-1]的数据，之后计算得到的数据与对应的 c[i]进行计算，计算的使能为循环边界的 i 使能，对其进行模 2 计算，输出得到的 c[i]数据，生成二进制码流。

CGRA 具体映射与第 4 次循环基本一致，不再赘述。

第 7、8 次循环的映射如图 4.3.2.7 所示：

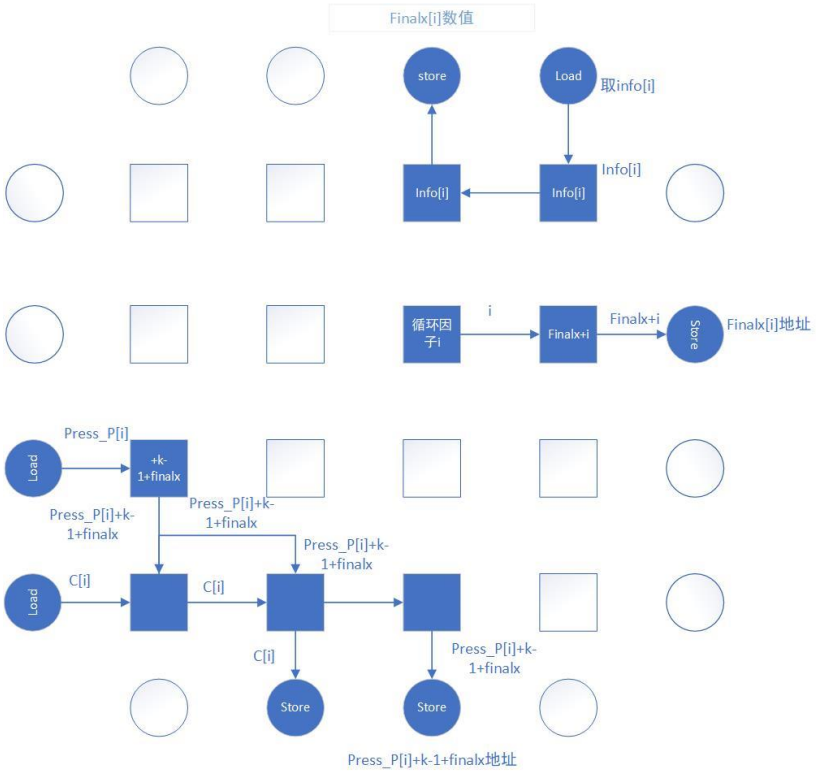

图 4.3.2.7 encoder_jia 编码部分第 7 次循环代码图

因为 7、8 次循环只要为按要求赋值，计算要求不高，主要是对 info 中的数据进行取值，之后赋值到 finalx 数组当中。

CGRA 映射中，循环因子 i 传递至 i+finalx 计算 PE，之后通过 Store 单元存储 finalx[i]的地址。针对第 7 次循环，左侧 load 单元读取 c[i]和 press_P[i]的值，其中 press_P[i]经过 press_P[i]+k-1+finalx 计算 PE，之后传递至最底层左数第三个 PE，通过 Store 单元保存 finalx 地址数据。c[i]通过最底层左数第二个 PE 通过 Store 单元保存 finalx 对应的数值。针对第 8 次循环，上侧 Load 功能 PE 读入 info[i]数值，经过 PE 传递，在赋值给 finalx[i]之后通过 Store 功能 PE 保存。

4.4 LDPC 译码

4.4.1 LDPC 译码概述

LDPC（低密度奇偶校验）译码，奇偶校验表示其纠错编码采用奇偶校验。低密度是指校验矩阵 H 是一种非常稀疏的矩阵，其中-1 的个数远小于 0 的个数。这对于译码算法而言是至关重要的，这表现在两个方面。其一是对于一个大的校验矩阵而言，太多的 1 会导致计算上的困难；其二是译码算法（此处特指置信传播类算法）中的很多假设实际上是不成立的，在 H 不满足低密度约束下对其性能会有很大影响。

在 LDPC 译码中信息在变量节点与校验节点中来回传递，其过程主要可分为如下四部分。其中$U_{mn}^{(l)}$与$V_{nm}^{(l)}$分别表示第 l 次迭代中校验节点 m 传递给变量节点 n 和变量节点 n 传递给校验节点 m 的信息。$V_n^{(l)}$为第 l 次迭代后变量节点 n 的后验信息。$V_n^{(0)}$为从信道获得的第 n 个变量节点的最似近然比。

(1)初始化：

$$V_{nm}^{(0)} = V_n^{(0)}$$

(2)校验节点更新：

$$\tanh\left(\frac{U_{mn}^{(l)}}{2}\right) = \prod_{N(m)\backslash n} \tanh\left(\frac{V_{nm}^{(l-1)}}{2}\right)$$

其中 N(m)\n 表示除变量节点 n 以外的所有与校验节点 m 相连的变量节点的集合。

一般为便于硬件计算，通常采用 min_sum 近似简化译码算法，上式被简化为：

$$U_{mn}^{(l)} = \left(\prod_{N(m)\backslash n} \operatorname{sign}\left(V_{nm}^{(l-1)}\right)\right) * \min_{N(m)\backslash n} V_{nm}^{(l-1)}$$

(3)变量节点更新：

$$V_{nm}^{(l)} = V_n^{(0)} + \sum_{M(n)\backslash m} U_{mn}^{(l)}$$

$$V_n^{(l)} = V_n^{(0)} + \sum_{M(n)} U_{mn}^{(l)}$$

其中 M(n)\m 表示除校验节点 m 以外的所有与变量节点 n 相连的校验节点的集合。

M(n) 表示所有与变量节点 n 相连的校验节点的集合。

(4)奇偶校验：

将变量节点的后验信息用于奇偶校验，若校验条件满足，则判决输出，否则重复迭代上

述步骤 2 与 3，直至满足校验或达到最大迭代次数。

4.4.2 LDPC 译码源码介绍及映射分析

代码中各变量意义如下：

变量名	意义	备注
H_matrix[i][j]	校验矩阵	-1 个数远小于 0
Col_base	校验矩阵行数	
Row_base	校验矩阵列数	
expand_z	矩阵扩张系数	
LLR_V2C[m][n]	变量节点 m 传递给校验节点 n 的信息	
LLR[m]	从信道获得的第 m 个变量节点的最似近然比	
de_flag	译码模式标志	1 表示线性算法译码，0 表示非线性算法译码
LLRC_mid[z]	校验节点更新的中间值	
LLR_C2V[n][m]	校验节点 n 传递给变量节点 m 的信息	
LLRV_mid[z]	变量节点更新的中间值	
Final_x[m]	变量节点的后验信息	

表 4.4.1 源码各变量名介绍

在映射分析中，CPU 负责外围主迭代循环与部分决策计算，主循环又包括下面四个循环运算操作，为运算的主要部分，由 CGRA 阵列完成。

（1） 初始化：

其 C++代码如下：

```
//initializing the matrices
for ( j=0;j<col_base;j++)
{
    for(i=0;i<row_base;i++)
    {
        if(H_matrix[i][j]!=-1)
        {
            for(z=0;z<expand_z;z++)
            {
                LLR_VtoC[i][j*expand_z+z]=LLR[j*expand_z+z];
            }
        }
    }
}
```

图 4.4.1 初始化源码

其中在迭代前，将变量节点传递给校验节点的信息矩阵初始化为从信道获得的变量节点的对数似然比（LLR）。该部分操作在 CGRA 阵列上的映射如下：

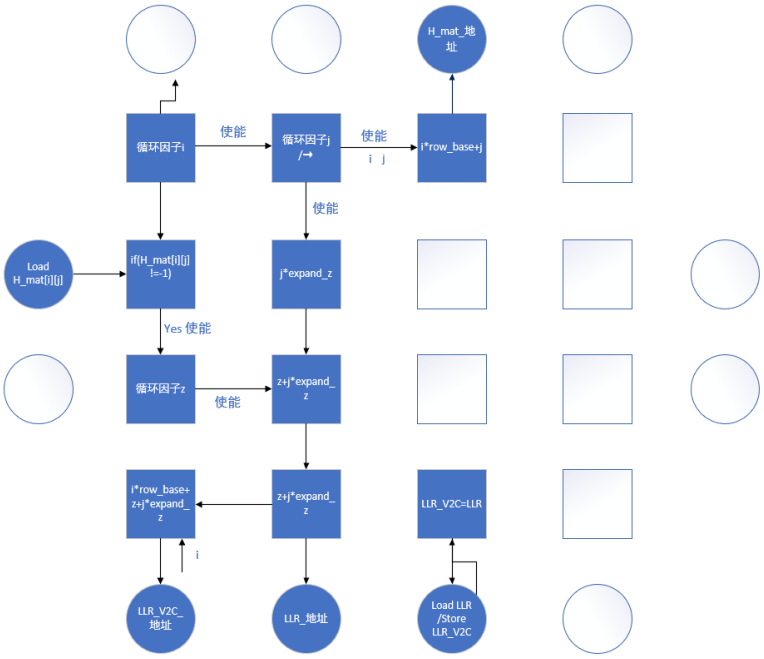

图 4.4.2 初始化部分映射

图中/表示同时进行多个操作，且↑↓←→表示数据路由。

(2) 校验节点更新

校验节点更新根据变量 de_flag 的不同有两种更新方式，一种是线性算法，适合在硬件上完成，另一种是采用 tanh 函数的非线性算法，也可以采用查找表的形式在硬件上较快地完成。由前所述，两种算法中的部分 if 决策操作可交给 CPU 完成，两种算法的源码与在阵列上的映射方案分别如下：

（1 线性算法：

采用 min-sum 近似简化译码算法的 C++源码如下：

```
//bit-to-check messages  //校验节点更新
if (de_flag == 1)//模式1 线性算法
    for ( i=0;i<row_base;i++)
    {
       for (j=0;j<col_base;j++)
       {
          if( H_matrix[i][j]!=-1 )
          {
             for(z=0;z<expand_z;z++)
                 min[z] = 1000;

             for (k=0;k<col_base;k++)
                 if (H_matrix[i][k]!=-1&&k!=j)
                 {
                    posi_k = H_matrix[i][k];
                    for(z=0;z<expand_z;z++)
                    {
                       temp = fabs(LLR_VtoC[i][k*expand_z+((z+posi_k)%expand_z)]);
                       if (temp < min[z])  //寻找最小的模
                       {
                          min[z]=temp;
                       }
                    }
                 }
```

图 4.4.3 线性算法取最小模部分源码

```
    for(z=0;z<expand_z;z++)
        LLRC_MID[z] = min[z];

    for (k=0;k<col_base;k++)
        if (H_matrix[i][k]!=-1&&k!=j)
        {
            posi_k = H_matrix[i][k];
            for(z=0;z<expand_z;z++)
            {
                if ((LLR_VtoC[i][k*expand_z+((z+posi_k)%expand_z)])>0)
                    LLRC_MID[z]=LLRC_MID[z];
                else
                    LLRC_MID[z]=LLRC_MID[z]*(-1);//Sign
            }
        }
    for(z=0;z<expand_z;z++)
        LLR_CtoV[i*expand_z+z][j]=LLRC_MID[z];//
}
```

图 4.4.4 线性算法取 Sign 函数部分源码

CGRA 阵列映射方案如下：

源码可分为两部分，但可在一个阵列上映射完成。

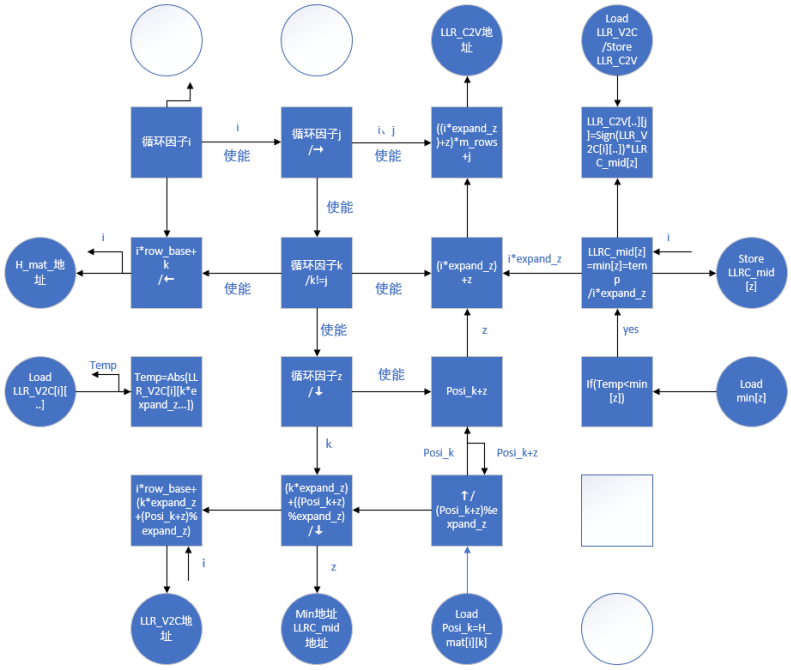

图 4.4.5 线性算法部分映射

图中 sign()对应代码中的符号函数，可用一个 PE 单独实现。

（2 非线性算法

直接采用 tanh 函数计算的算法 C++源码及映射方案分别如下：

```
else  //模式2 非线性算法
    for ( i=0;i<row_base;i++)
    {
        for (j=0;j<col_base;j++)
        {
            if( H_matrix[i][j]!=-1 )
            {
                for(z=0;z<expand_z;z++)
                    LLRC_MID[z] = 1;

                for (k=0;k<col_base;k++)
                    if (H_matrix[i][k]!=-1&&k!=j)
                    {
                        posi_k = H_matrix[i][k];
                        for(z=0;z<expand_z;z++)
                            LLRC_MID[z]=LLRC_MID[z]*tanh(LLR_VtoC[i][k*expand_z+((z+posi_k)%expand_z)]/2);
                    }
                for(z=0;z<expand_z;z++)
                    LLR_CtoV[i*expand_z+z][j]=2*atanh(LLRC_MID[z]);
            }
        }
    }
```

图 4.4.6 非线性算法源码

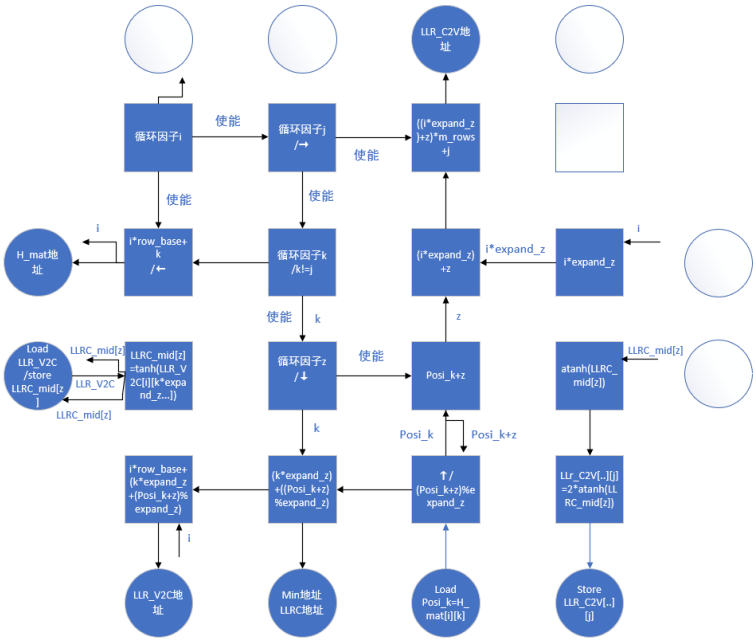

图 4.4.7 非线性算法部分映射

(3) 变量节点更新

变量节点更新主要分为两部分的计算，一是变量节点传递给校验节点的信息矩阵计算，二是变量节点的后验信息矩阵计算。同样也是部分 if 决策操作交给 CPU 完成，阵列完成主

要的计算部分，源码与映射分析分别如下：

（1 变量节点传递给校验节点信息矩阵 LLR_V2C 的计算源码如下：

```
//变量节点更新
for (j=0;j<col_base;j++)
{
    for ( i=0;i<row_base;i++)
    {
        if (H_matrix[i][j]!=-1)
        {
            for(z=0;z<expand_z;z++)
                LLRV_MID[z]=0;

            for (k=0;k<row_base;k++)
            {
                if (H_matrix[k][j]!=-1&&k!=i)
                {
                    posi_k = H_matrix[k][j];
                    for(z=0;z<expand_z;z++)
                        LLRV_MID[z]=LLRV_MID[z]+LLR_CtoV[k*expand_z+((z+expand_z-posi_k)%expand_z)][j];
                }
            }
            for(z=0;z<expand_z;z++)
                LLR_VtoC[i][j*expand_z+z]=LLRV_MID[z]+LLR[j*expand_z+z];
        }
    }
}
```

图 4.4.8 LLR_V2C 计算源码

CGRA 阵列映射如下：

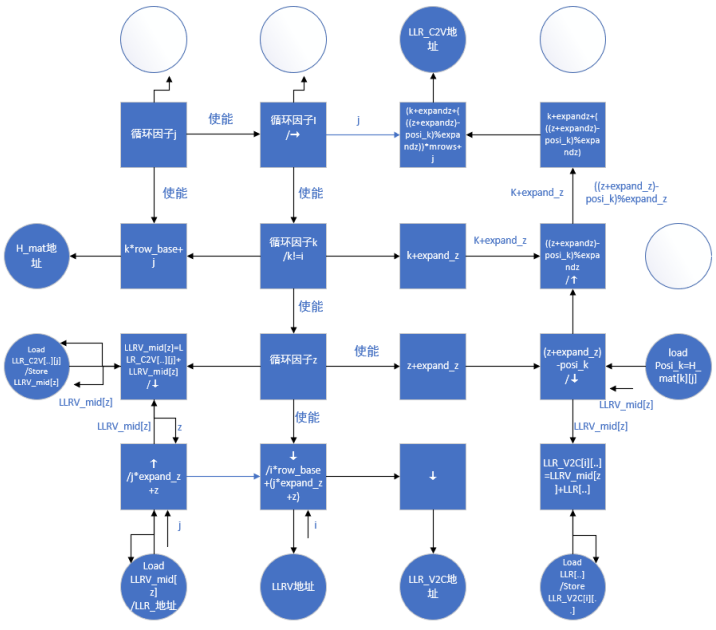

图 4.4.9 LLR_V2C 计算部分映射

（2 变量节点的后验信息计算

先计算得变量节点的中间信息，并对其数值进行二值化得到变量节点的后验信息。

该部分源码如下：

```
for (j=0;j<col_base;j++)
{
    for(z=0,z<expand_z;z++)
        LLRV_MID[z]=0;
    for ( i=0;i<row_base;i++)
    {
        if(H_matrix[i][j]!=-1)
        {
            posi_i = H_matrix[i][j];
            for(z=0;z<expand_z;z++)
                LLRV_MID[z] = LLRV_MID[z] + LLR_CtoV[i*expand_z+((z+expand_z-posi_i)%expand_z)][j];
        }
    }

    for(z=0;z<expand_z;z++)
    {
        finalx[j*expand_z+z]=LLRV_MID[z]+LLR[j*expand_z+z];
        if (finalx[j*expand_z+z]>0)
            finalx[j*expand_z+z]=0;
        else
            finalx[j*expand_z+z]=1;//二值运算
    }
}
```

图 4.4.10 后验信息计算源码

阵列映射如下：

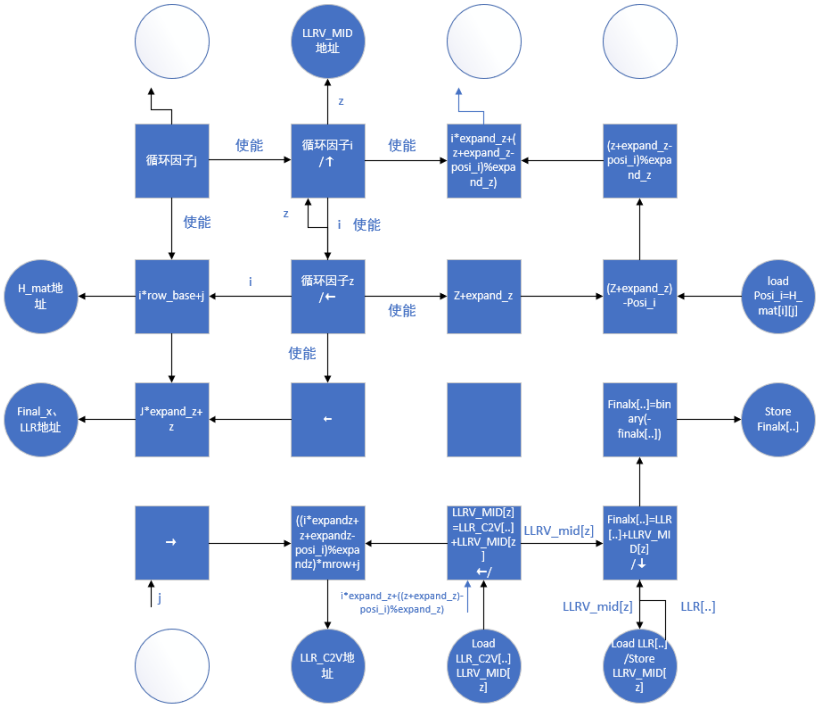

图 4.4.11 后验信息计算源码

图中 binary()表示二值函数，可用一个 PE 单独实现。

(4)奇偶校验

进行完所有运算后，最后对其奇偶校验，该部分源码如下：

```
//奇偶性校验
int parity=0, count_num=0;
for ( i=0;i<row_base;i++)
{
    for(z=0;z<expand_z;z++)
    {
        parity = 0;
        for (j=0;j<col_base;j++)
        {
            posi_j = H_matrix[i][j];
            parity = parity+finalx[j*expand_z+((z+posi_j)%expand_z)];
        }
        if (parity%2==1)
        {
            count_num=1;
            break;
        }
    }
}
if(count_num==0)
    return;
```

图 4.4.12 奇偶校验源码

在阵列上的运算映射如下：

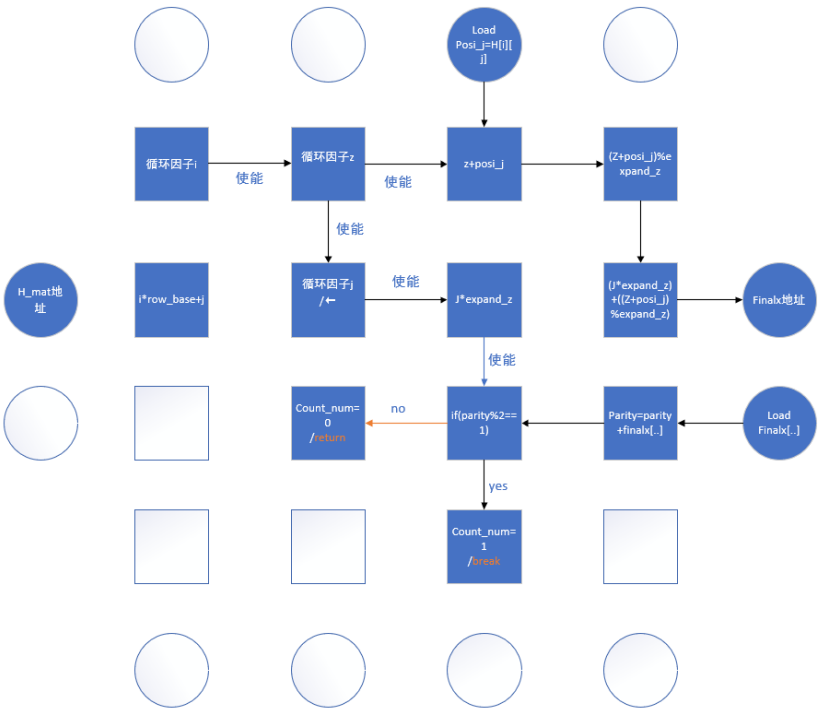

图 4.4.13 奇偶校验部分映射

至此全部内部循环计算已完成。如果满足表达式条件则可退出迭代(return)，否则发出(break)信号让 CPU 启动下一次迭代，直至满足表达式条件或者达到最大迭代次数。

5 总结与展望

随着人们日益增长的对高性能通信的需求，全球各国都在重视 6G 的研发，我国对于 6G 技术的研发迫在眉睫。我国的通信基站数量显著上升，然而传统的 FPGA 开发样机，用 ASIC 批量生产基站的方式已经不能满足未来 6G 技术的要求，为了能够支持 6G 技术的应用落地，我们提出了一种基于可重构架构技术的芯片设计思路，实现了在达到 FPGA 灵活性的同时兼具 ASIC 的高能效。

我们介绍了可重构架构的历史和发展现状，并且阐述了可重构架构的设计流程，分为算法设计、架构设计、工具链设计以及调度优化。我们分析了整套 PDSCH 通信算法的实现思路，着重介绍了在我们提出的架构上实现运算的 LDPC 编码、CRC 添加、LDPC 译码算法的算法原理和可重构架构的映射方式。

总结起来，CGRA 由于其出色的性能、功耗与灵活性，是 6G 通信技术，尤其是 6G 物理层处理的重要实现方式。研究团队通过对物理层典型算法在 CGRA 中映射实现的分析，初步验证了 CGRA 中运行物理层典型算法的可行性。研究团队还经过进一步的梳理与验证，初步遴选出了适合在 CGRA 中进行加速的物理层模块，并明确了由于自身算法特点而在 CGRA 中实现效率低下的物理层模块，为下一步更有针对性的深入研究与验证工作打下扎实基础。此外，根据物理层算法位宽灵活、访存频繁、循环嵌套多等特点，研究团队还对 CGRA 的架构进行了并行化、访存方式改进等优化设计，进一步提高了算法与硬件架构的契合度。

在明年，我们将针对上述分析的算法在模拟器上进一步仿真其时序特性，对架构各方面细节进行完善，具体包括 PE 模块算子、PE 模块互联通道、IO、存储模块等方面。此外我们还将设计完成面向该架构的指令格式以及编译器功能，实现更高效能高效率算法映射，为 6G 通信技术提供更加高效灵活的硬件基础。