中国科学技术大学

本科毕业论文

动态可重构的 FPGA 矩阵乘法加速系统设计

作者姓名:	张学涵
学　　号:	PB21000079
专　　业:	计算机科学与技术
导　　师:	宫磊 教授
完成时间:	2025 年 5 月 19 日

摘　要

本工作在 Xilinx KV260 多处理器系统芯片的 FPGA 上，成功实现了 3 个可重构分区的布局，并在上面分别部署了稀疏矩阵解压、稠密矩阵乘法、稀疏矩阵压缩的可重构模块。可重构分区通过 AXIMM 协议对 KV260 的内存直接访问，同时通过 AXIS 协议进行互联。可重构模块使用高层次综合语言编写，能充分利用 FPGA 的硬件资源，实现并行、流水线地处理。稠密矩阵乘法使用分块与脉动阵列优化，每个块大小 12×12，对于 128×128 矩阵乘法相较 CPU 朴素算法实现了 25× 的加速比。

在 KV260 的 CPU 上，编写程序调用 FPGA 上的计算模块，实现了异构加速计算。支持 COO, CSR 和 CSC 三种稀疏格式的稀疏矩阵解压/压缩/乘法操作。所有贡献开源于 github.com/fjtcin/dfx-3rp。

关键词：可重构计算；FPGA 加速；稀疏矩阵

ABSTRACT

This work successfully implements a layout of three reconfigurable partitions on the FPGA of a Xilinx KV260 multi-processor system-on-chip (MPSoC). Reconfigurable modules for sparse matrix decompression, dense matrix multiplication, and sparse matrix compression are deployed on these partitions respectively. The reconfigurable partitions directly access the KV260's memory via the AXI Memory Mapped protocol and are interconnected using the AXI Stream protocol. The reconfigurable modules are written in the high-level synthesis language, enabling full utilization of FPGA hardware resources for parallel and pipelined processing. The dense matrix multiplication is optimized using tiling and a systolic array, with each block sized at 12×12. For a 128×128 matrix multiplication, this achieves a 25× speedup compared to a naive CPU algorithm.

Applications have been developed to invoke the computational modules on the FPGA, realizing heterogeneous accelerated computing. The system supports sparse matrix decompression, compression, and multiplication operations for COO, CSR, and CSC sparse formats. All contributions are open-source at github.com/fjtcin/dfx-3rp.

Key Words: Reconfigurable Computing; FPGA Acceleration; Sparse Matrix

目　录

第一章　绪论

第一节　研究背景与意义

我们正处在一个信息爆炸的时代，以大数据、人工智能（AI）、物联网（IoT）等为代表的新兴技术蓬勃发展，驱动着社会各领域的深刻变革[1]。在这些技术的背后，海量数据的处理和分析是核心环节，而矩阵运算，特别是矩阵乘法，作为一种基础且计算密集型的操作，广泛应用于科学计算、图像处理、信号处理、机器学习、推荐系统、图计算等众多领域[2]。例如，在深度学习中，神经网络的训练和推理过程涉及大量的卷积和全连接层计算，这些本质上都可以归结为大规模的矩阵或张量运算[3]。在图计算中，邻接矩阵的乘法可用于发现节点间的路径关系[4]。随着模型复杂度和数据规模的持续增长，对计算能力的需求也呈现出指数级增长的态势，传统的计算模式面临着严峻的挑战。

中央处理器（CPU）作为通用的计算核心，虽然具有强大的逻辑控制能力和灵活性，但在处理高度并行化的数据密集型任务（如大规模矩阵乘法）时，其固有的串行执行特性和有限的并行处理单元（核心数）往往导致性能瓶颈[5]。图形处理器（GPU）凭借其众多的计算核心（流处理器）和高内存带宽，在并行计算领域取得了巨大成功，特别是在稠密矩阵运算和深度学习领域展现出卓越的加速效果[6]。然而，GPU 的体系结构相对固定，对于某些特定类型的计算（例如，具有高度不规则访存模式的稀疏矩阵运算）或者需要细粒度流水线定制的任务，其效率可能并非最优[7]。此外，GPU 通常功耗较高，且其编程模型（如 CUDA 或 OpenCL）虽然强大，但与底层硬件的映射关系不如 FPGA 直接。

现场可编程门阵列（FPGA）作为一种可编程硬件，提供了一种介于通用处理器和专用集成电路（ASIC）之间的解决方案[8]。FPGA 内部包含大量的可配置逻辑块（CLB）、存储单元（BRAM）、DSP 单元等，用户可以通过硬件描述语言（HDL）或高层次综合（HLS）工具对其进行编程，实现高度定制化的硬件加速器[9]。相比 CPU，FPGA 能够实现真正在硬件层面的并行和流水线处理，大幅提升计算密集型任务的性能功耗比。相比 GPU，FPGA 的架构灵活性允许针对特定算法进行深度优化，例如为稀疏矩阵的不同存储格式（如 CSR, CSC, COO 等）设计专门的处理单元和访存逻辑[10]。相比 ASIC，FPGA 虽然在绝对性能和功耗上可能稍逊一筹，但其可重复编程的特性大大降低了设计成本和风险，缩短了开

发周期，尤其适合算法快速迭代或需要适应多种应用场景的领域。

然而，传统的基于 FPGA 的加速器设计通常是“静态”的，即一旦配置了 FPGA，其硬件逻辑就固定下来，直到下一次完全重新配置。这种模式对于功能固定的应用是有效的，但在许多现代应用场景中，计算任务的需求可能是动态变化的。例如，一个复杂的数据处理流程可能包含多个阶段，每个阶段需要不同的加速核心；或者，系统需要处理不同格式、不同稀疏度的稀疏矩阵，为每种情况设计一个最优的静态加速器并在需要时切换，会导致整个 FPGA 的重配置，这个过程通常耗时较长（从毫秒级到秒级），中断服务，对于需要快速响应或持续服务的系统是不可接受的[11]。

为了克服静态 FPGA 设计的局限性，动态部分重构（Dynamic Partial Reconfiguration, DPR）技术应运而生[12-13]。DPR 允许在 FPGA 运行时，仅对其内部指定的一部分区域（称为可重构分区，Reconfigurable Partition, RP）进行重新编程，加载新的硬件逻辑（称为可重构模块，Reconfigurable Module, RM），而 FPGA 的其他部分（静态区域和其他 RP）可以保持正常工作，不被中断[14-15]。这项技术极大地增强了 FPGA 的灵活性和资源利用率。通过 DPR，可以在有限的 FPGA 资源上分时复用不同的加速功能，或者根据输入数据的特性（如稀疏矩阵的格式或密度）动态加载最优的处理模块，从而实现“按需定制”的硬件加速。

将 DPR 技术应用于矩阵乘法加速，特别是涉及到稀疏矩阵的复杂场景，具有重要的研究价值和应用前景。稀疏矩阵在现实世界中无处不在（如社交网络关系、物理模拟、自然语言处理中的词袋模型等），其存储和计算方式与稠密矩阵截然不同，且存在多种存储格式，不同格式适用于不同的计算场景和优化策略[16]。一个能够动态切换稀疏矩阵处理逻辑（如不同格式的解压缩、压缩、乘法核心）的 FPGA 加速系统，将能更高效地适应多样化的应用需求，提高硬件资源的利用率和系统的整体性能。

本设计正是基于上述背景，旨在探索和实现一个基于 FPGA 动态部分重构技术的矩阵乘法加速系统。系统以 Xilinx Kria KV260 Vision AI Starter Kit 为硬件平台，该平台搭载了 Zynq UltraScale+ MPSoC，集成了强大的 ARM 处理核心和 FPGA 可编程逻辑，构成了典型的异构计算系统[17]。利用 MPSoC 的 FPGA 部分实现硬件加速，利用 ARM 核心运行 Linux 操作系统进行任务调度、资源管理和与 FPGA 的交互。通过在 FPGA 上划分多个可重构分区，并利用高层次综合（HLS）语言（如 C/C++）设计可重构的矩阵运算模块（包括稀疏矩阵解压、稠密矩阵乘法、稀疏矩阵压缩等），实现了这些模块在运行时的动态加载和切换。

系统通过标准的 AXI 接口协议实现 FPGA 内部模块间以及 FPGA 与处理器系统（PS）端内存的数据交互。结合 Xilinx Runtime (XRT) 库，在运行于 CPU 上的应用程序可以方便地控制 FPGA 上的可重构模块，实现异构协同计算。

第二节　国内外研究现状

一、静态 FPGA 加速

许多研究集中于优化特定类型的矩阵乘法。对于稠密矩阵乘法，研究者们探索了各种并行计算架构（如脉动阵列）、分块策略、片上存储优化以及利用 HLS 简化设计流程的方法，以期达到接近理论峰值的性能[18-19]。对于稀疏矩阵乘法（SpMV, SpMM），挑战主要在于处理不规则的内存访问和计算模式。研究工作通常针对特定的稀疏格式设计专门的硬件结构，利用流水线、数据预取、负载均衡等技术来缓解访存瓶颈和提高计算单元利用率[20]。

二、动态重构技术应用

DPR 技术本身已经相对成熟，并在通信、自适应滤波、软件定义无线电等领域有所应用[21]。在计算加速领域，DPR 被用于根据需要加载不同的加密算法、图像处理算子或数据压缩算法[22-23]。将 DPR 用于矩阵运算的研究相对较少。可行的初步探索包括根据矩阵的尺寸或特性动态调整 FPGA 上矩阵乘法器的并行度或数据位宽，或利用 DPR 在不同类型的数值计算核心（如浮点加法器、乘法器）之间切换。

三、异构计算与运行时管理

随着 SoC FPGA（如 Xilinx Zynq 系列）的普及，基于 FPGA 的异构计算系统成为主流。研究者们关注如何高效地在处理器和 FPGA 之间划分任务、传输数据以及管理 FPGA 上的硬件资源[24]。Xilinx PYNQ 框架和 XRT 运行时库等工具的出现，极大地简化了在高级操作系统（如 Linux）层面调用 FPGA 加速器的开发流程。

四、现有研究的局限性

尽管已有大量关于 FPGA 矩阵加速和 DPR 技术的研究，但将两者深度结合，特别是针对包含多种稀疏/稠密矩阵操作、需要动态适应不同稀疏格式的复杂计

算流，进行系统性设计和实现的研究尚不多见。现有工作要么侧重于静态优化单一类型的矩阵运算，要么 DPR 的应用场景相对简单，未能充分发挥 DPR 在处理复杂、多变矩阵计算任务流中的潜力。此外，如何在 HLS 设计流程中高效地集成 DPR，以及如何通过 XRT 等运行时环境有效管理多个可重构分区的动态加载和任务调度，也是需要进一步探索的问题。

第三节　本设计的主要工作与创新点

本毕业设计针对现有研究的不足，着眼于设计并实现一个动态可重构的 FPGA 矩阵乘法加速系统，旨在提供一个既高效又灵活的解决方案，以适应现代应用中复杂多变的矩阵计算需求。具体工作和创新点如下：

基于 DPR 的灵活矩阵运算硬件架构设计 在 Xilinx KV260 平台的 FPGA 上，设计并实现了包含三个独立可重构分区（RP）的硬件架构。这种多 RP 架构允许多个不同的矩阵运算模块（RM）共存或被快速替换，为实现复杂的计算流水线或并行处理不同任务提供了硬件基础。

面向多类型矩阵运算的可重构模块开发 使用 Vitis HLS 工具，以 C/C++ 语言开发了一系列针对不同矩阵运算任务的 RM，包括：

- 稀疏矩阵解压（将特定稀疏格式转换为稠密格式）
- 稠密矩阵乘法
- 稀疏矩阵压缩（将稠密格式转换为特定稀疏格式）
- （隐含支持）稀疏矩阵格式转换（可通过解压 + 压缩 RM 组合实现）
- （隐含支持）稀疏矩阵乘法（可通过解压 + 稠密乘法，或解压 + 稠密乘法 + 压缩等 RM 组合/序列实现）

这些模块被设计为可动态加载到 RP 中，使得系统能够根据具体任务需求，配置相应的硬件加速逻辑，并切换对不同稀疏矩阵格式的处理。

标准化接口与异构系统集成 设计的 RM 均采用标准的 AXI 接口协议。AXIMM 接口用于高效访问 KV260 的共享 DDR 内存，实现大规模数据的吞吐；AXIS 接口用于连接不同的 RM，支持在 FPGA 内部构建数据流驱动的计算流水线。系统运行在 KV260 的 ARM 处理器上的 Ubuntu Linux 操作系统，通过 Xilinx Runtime (XRT) 库，实现了上层软件对 FPGA 硬件资源（包括 DPR 操作和 RM 任务执行）的统一管理和调用，构建了一个完整的异构加速计算平台。

本设计的创新之处在于：系统性地将动态部分重构技术应用于涵盖多种（稀疏、稠密、转换）矩阵运算的 FPGA 加速场景，并构建了一个包含多 RP、HLS 设计的 RM、标准 AXI 接口、以及基于 Linux+XRT 的软硬件协同控制的完整异构系统。这为应对未来更复杂、更动态的计算挑战提供了一种有潜力的技术途径。

第四节　论文结构安排

本论文共分为七章，结构安排如下：**第一章：绪论**主要介绍研究背景、意义、国内外研究现状、本设计的主要工作与创新点以及论文的结构安排。**第二章：相关技术概述**详细介绍本设计所涉及的关键技术，包括矩阵运算基础、FPGA 技术原理、高层次综合（HLS）、动态部分重构（DPR）技术、AXI 总线协议、异构计算系统以及 Xilinx KV260 平台和 XRT 运行时环境。**第三章：系统总体设计**阐述动态可重构矩阵乘法加速系统的整体架构，包括硬件系统设计（FPGA 分区规划、静态与动态区域划分、接口设计）和软件系统设计（操作系统层面、XRT 应用层面、任务调度逻辑）。**第四章：可重构模块硬件实现**详细介绍使用 Vitis HLS 设计和实现各个可重构矩阵运算模块（稀疏解压、稠密乘法、稀疏压缩等）的过程，包括算法分析、HLS 优化（并行、流水线）、接口实现以及综合与实现结果。**第五章：软件系统实现与集成**描述在 KV260 的 ARM 处理器上配置 Linux 环境、编写主机端应用程序以控制 DPR 流程（加载/卸载 RM）、管理数据传输以及调用 FPGA 执行加速任务的具体实现方法。**第六章：实验结果与分析**对实现的系统进行测试和评估。包括功能验证、稠密矩阵乘法在 FPGA 上的加速性能测试（与纯 CPU 实现对比），以及系统灵活性和资源利用率的分析。**第七章：总结与展望**总结本论文完成的主要工作和取得的研究成果，分析存在的不足，并对未来可以进一步研究的方向进行展望。

第二章　相关技术概述

本章旨在详细介绍本工作所涉及的核心技术。这些技术涵盖了从基础算法理论到硬件平台、设计方法学、接口协议以及软件运行时环境等多个层面。深入理解这些技术对于后续章节中系统设计、实现与评估的阐述至关重要。本章将依次介绍矩阵运算基础、FPGA 技术原理、高层次综合（HLS）、动态部分重构（DPR）技术、AXI 总线协议、异构计算系统、Xilinx Kria KV260 平台特性以及 Xilinx Runtime（XRT）环境。

第一节　矩阵运算基础

根据矩阵中非零元素的分布，矩阵可分为稠密矩阵和稀疏矩阵。稠密矩阵的元素大多为非零值，而稀疏矩阵则包含大量的零元素。针对稀疏矩阵的特性，发展出了多种压缩存储格式，如坐标列表（COO）、压缩稀疏行（CSR）、压缩稀疏列（CSC）等，以节省存储空间并优化计算效率。本项目涉及的核心计算任务包括稀疏矩阵与稠密矩阵之间的相互转换（解压与压缩）、稀疏矩阵格式之间的转换，以及稠密矩阵乘法和稀疏矩阵乘法。这些运算，特别是涉及大规模矩阵时，对计算性能提出了极高要求，从而驱动了对硬件加速方案的探索。

第二节　FPGA 技术原理

现场可编程门阵列（FPGA）是一种半定制电路，其硬件结构可以在制造完成后由用户根据需求进行配置。FPGA 内部主要由可配置逻辑块（CLB）、输入输出块（IOB）、可编程布线资源以及嵌入式存储器（如 BRAM）、数字信号处理单元（DSP Slice）等构成。用户通过加载特定的配置文件（比特流）来定义这些单元的功能及其互连方式，从而实现所需的数字逻辑电路。与通用处理器相比，FPGA 能够实现高度的并行计算和深流水线操作，充分利用硬件资源，从而在特定计算密集型任务上展现出显著的性能优势和能效比。其可重构性也为算法的迭代升级和功能调整提供了灵活性。

第三节　高层次综合

高层次综合（High-Level Synthesis, HLS）是一种设计方法，它允许开发者使用 C/C++ 等高层次语言来描述硬件行为，然后通过 HLS 工具将其自动转换为低层次的硬件描述语言（HDL），如 Verilog 或 VHDL。HLS 的出现极大地提高了 FPGA 的设计效率，缩短了开发周期，并使得不熟悉底层 HDL 的软件工程师也能够参与到 FPGA 开发中。在本项目中，可重构模块采用 Vitis HLS 进行编写，通过利用 HLS 提供的并行化、流水线化等优化指令（pragmas），能够有效地将算法映射到 FPGA 的硬件资源上，实现高效的硬件加速器设计。

第四节　动态部分重构技术

动态部分重构（Dynamic Partial Reconfiguration, DPR），在 Xilinx 平台中也称为 Dynamic Function eXchange (DFX)，是一项先进的 FPGA 技术。它允许在 FPGA 正常运行期间，对其内部的特定区域（即可重构分区, Reconfigurable Partition, RP）进行动态地、部分地重新编程，而 FPGA 的其他部分保持原有功能并继续工作。这项技术为系统带来了极大的灵活性，使得 FPGA 能够根据应用需求实时切换不同的硬件加速模块（可重构模块, Reconfigurable Module, RM），或者在线更新硬件功能，而无需中断整个系统的运行。在本项目中，通过在 FPGA 上成功布局 3 个可重构分区，并部署不同的矩阵运算模块，可在不同稀疏矩阵格式处理模块间进行切换，以适应多样化的加速场景。

第五节　AXI 总线协议

高级可扩展接口（Advanced eXtensible Interface, AXI）是 ARM 公司提出的一种高性能、高带宽、低延迟的片上总线协议，已成为业界标准，广泛应用于 SoC（System-on-Chip）设计中。AXI 协议定义了主设备（Master）和从设备（Slave）之间的通信规范，支持多种通信模式。本项目中主要使用了两种 AXI 接口：AXI Memory Mapped (AXIMM) 协议和 AXI Stream (AXIS) 协议。AXIMM 协议用于可重构分区对 KV260 的内存进行直接访问（DMA），实现高效的数据读写；AXIS 协议则用于可重构模块之间的互联，支持高速、连续的数据流传输，非常适合流水线式的数据处理架构。

第六节　异构计算系统

异构计算系统是指在一个系统中集成多种不同类型计算单元（如 CPU、GPU、FPGA、DSP 等）的计算平台。这种架构旨在结合不同处理单元的优势，例如 CPU 擅长处理复杂的控制流和非结构化任务，而 FPGA 则擅长执行大规模并行计算和定制化的数据处理。通过合理的任务划分和协同工作，异构计算系统能够实现比单一类型处理器更高的性能和能效。本项目基于 Xilinx KV260 MPSoC 构建了一个异构计算系统，其中 ARM Cortex-A53 CPU 负责运行操作系统（Ubuntu Linux）和上层应用程序，并调度 FPGA 上的硬件加速模块执行计算密集型的矩阵运算任务。

第七节　Xilinx Kria KV260 平台特性

Xilinx Kria KV260 Vision AI Starter Kit 是基于 Kria K26 系统级模块（SOM）的开发平台。K26 SOM 是一款多处理器系统芯片（MPSoC），集成了四核 ARM Cortex-A53 处理系统（Processing System, PS）和拥有丰富可编程逻辑（Programmable Logic, PL）资源的 FPGA。KV260 平台专为边缘 AI 和视觉应用设计，但其强大的异构处理能力和灵活的接口使其也适用于其他加速计算任务。它提供了 DDR4 内存、多种外设接口以及对 Xilinx 开发工具链的良好支持。其 MPSoC 架构天然支持 PS 与 PL 之间的高效协同，是实现本项目中 CPU 控制、FPGA 加速的理想硬件基础。FPGA 部分支持动态部分重构，为实现灵活可变的加速器提供了硬件保障。

第八节　Xilinx 运行时环境

Xilinx Runtime (XRT) 是一个开源的、标准化的软件平台，旨在简化主机 CPU 与 Xilinx FPGA 加速器之间的交互。XRT 包括用户空间库、API 以及内核驱动程序，它为应用程序提供了一个统一的接口来管理和控制 FPGA 上的加速内核，无论这些内核是部署在 Alveo 数据中心加速卡还是像 KV260 这样的嵌入式 SoC 平台上。XRT 负责处理诸如加载比特流（包括部分重构的比特流）、分配和迁移内存缓冲区、调度内核执行以及收集性能数据等任务。在本项目中，部署在 KV260 的 CPU 上的 Ubuntu Linux 操作系统中，应用程序通过 XRT 提供的 API 来调用和管理 FPGA 上的动态可重构矩阵运算模块，实现了高效的异构加速计算流程。

第三章　系统总体设计

在深入探讨具体的硬件实现和软件编程细节之前，本章将阐述动态可重构矩阵乘法加速系统的整体架构。系统设计遵循软硬件协同设计（Hardware/Software Co-design）的理念，旨在充分利用 Xilinx Kria KV260 平台的异构计算能力，结合动态部分重构技术，实现一个既高性能又具备高度灵活性的矩阵运算加速解决方案。本章将分别从硬件系统架构和软件系统架构两个层面进行阐述，为后续章节的详细实现奠定基础。

第一节　硬件系统架构

一、设计总览

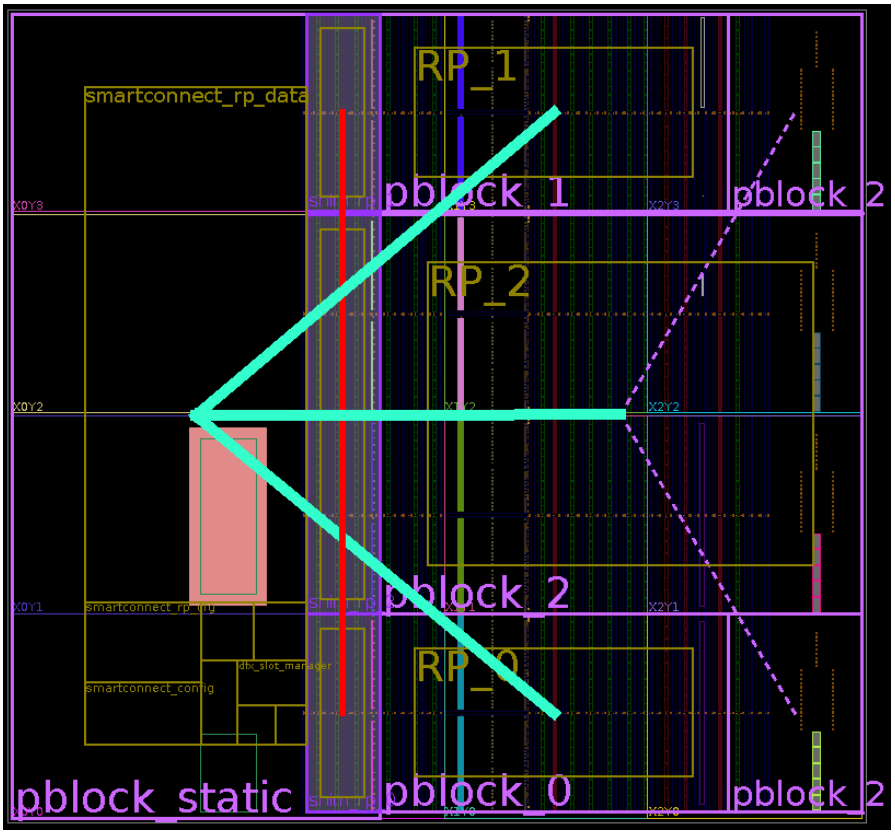

图 3.1　FPGA 布局。RP_0 装载稀疏矩阵解压 RM，RP_1 装载稀疏矩阵压缩 RM，RP_2 装载稠密矩阵乘法 RM。

如图 3.1 所示，在 FPGA 的整体规划上，我们将其划分为左侧的静态区域（Static Region）和右侧的动态区域（Dynamic Region）。表 3.1 展示了分配给静态

表 3.1　FPGA 资源分配

资源类型	静态区域	RP_0	RP_1	RP_2
LUT	23040	19352	19352	54064
FF	46080	39360	39360	109440
BRAM	0	36	36	72
DSP	288	216	216	528

区域和 3 个 RP 的资源情况。静态区域承载了系统运行所必需的基础逻辑，例如与 PS 的接口控制器、时钟管理、中断管理等，以及连接 RP 的 FIFO 缓冲。本设计成功实现了三个独立的可重构分区（RP），这三个分区为动态加载不同的计算模块提供了物理基础。每个可重构分区均设计为能够容纳一个可重构模块（RM）。具体部署的 RM 包括稀疏矩阵解压模块、采用脉动阵列结合分块策略的稠密矩阵乘法模块以及稀疏矩阵压缩模块。这些模块均采用 Vitis 高层次综合（HLS）语言编写，通过精心设计的 pragma 指令，实现了高度的并行计算和流水线操作，从而最大化地利用 FPGA 的硬件资源，提升运算效率。

二、模块连接

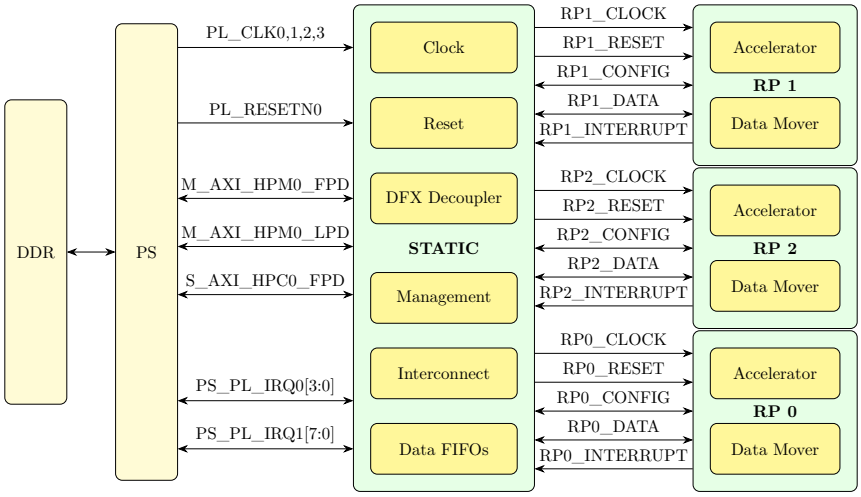

图 3.2　静态区域与动态区域间的连接。

接口设计是确保数据高效流转和模块协同工作的关键。如图 3.2 和图 3.3 所示，在可重构分区与 KV260 片上内存的交互方面，采用了 AXI4 Memory Mapped (AXIMM) 协议。这使得每个动态加载的 RM 都能够直接、高效地对系统主存进行读写操作，为大规模矩阵数据的传输提供了高带宽通道。静态区域与动态区域之间，以及动态区域与 PS 之间的控制与状态信号（CONFIG）交互，则通过

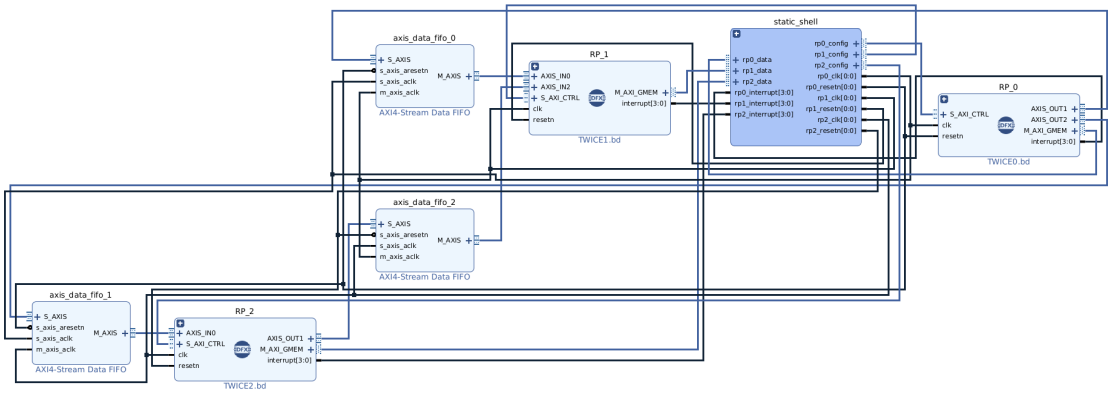

图 3.3　静态区域与 RP 间的连接，以及不同 RP 间的互联。

Network 0					
/RP_2/rm_comm_box_0					
/RP_2/rm_comm_box_0/m_axi_gmem (49 address bits : 16E)					
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_HIGH	0x0000_0008_0000_0000	32G	0x0000_000F_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_LOW	0x0000_0000_0000_0000	2G	0x0000_0000_7FFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_LPS_OCM	0x0000_0000_FF00_0000	16M	0x0000_0000_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_QSPI	0x0000_0000_C000_0000	512M	0x0000_0000_DFFF_FFFF
/RP_0/rm_comm_box_0					
/RP_0/rm_comm_box_0/m_axi_gmem (49 address bits : 16E)					
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_QSPI	0x0000_0000_C000_0000	512M	0x0000_0000_DFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_HIGH	0x0000_0008_0000_0000	32G	0x0000_000F_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_LPS_OCM	0x0000_0000_FF00_0000	16M	0x0000_0000_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_LOW	0x0000_0000_0000_0000	2G	0x0000_0000_7FFF_FFFF
/RP_1/rm_comm_box_0					
/RP_1/rm_comm_box_0/m_axi_gmem (49 address bits : 16E)					
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_HIGH	0x0000_0008_0000_0000	32G	0x0000_000F_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_DDR_LOW	0x0000_0000_0000_0000	2G	0x0000_0000_7FFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_LPS_OCM	0x0000_0000_FF00_0000	16M	0x0000_0000_FFFF_FFFF
/static_shell/zynq_ultra_ps_e_0/SAXIGP0	S_AXI_HPC0_FPD	HPC0_QSPI	0x0000_0000_C000_0000	512M	0x0000_0000_DFFF_FFFF
Network 1					
/static_shell/zynq_ultra_ps_e_0					
/static_shell/zynq_ultra_ps_e_0/Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G] ,0x0080000000 [512M])					
/RP_0/AccelConfig_0/s_axi_ctrl	s_axi_ctrl	reg0	0x00_8000_0000	16M	0x00_80FF_FFFF
/RP_0/rm_comm_box_0/s_axi_control	s_axi_control	reg0	0x00_8100_0000	16M	0x00_81FF_FFFF
/RP_1/AccelConfig_0/s_axi_ctrl	s_axi_ctrl	reg0	0x00_8200_0000	16M	0x00_82FF_FFFF
/RP_1/rm_comm_box_0/s_axi_control	s_axi_control	reg0	0x00_8300_0000	16M	0x00_83FF_FFFF
/RP_2/AccelConfig_0/s_axi_ctrl	s_axi_ctrl	reg0	0x00_8400_0000	16M	0x00_84FF_FFFF
/RP_2/rm_comm_box_0/s_axi_control	s_axi_control	reg0	0x00_8500_0000	16M	0x00_85FF_FFFF
/static_shell/dfx_slot_manager/siha_manager_0/s_s_axi		reg0	0x00_A010_0000	64K	0x00_A010_FFFF

图 3.4　AXI 地址空间分配。

AXI4 Lite 接口实现。图 3.4 展示了为 AXIMM 与 AXI Lite 接口分配的地址空间。同时，为了实现可重构模块之间的数据流传递和协同处理，例如将解压后的数据直接送入乘法模块，我们设计了基于 AXI4 Stream (AXIS) 协议的互联接口。这种流式接口非常适合于流水线式的处理流程，能够有效减少数据在模块间传递的延迟。我们在 RP 间添加了数据缓冲队列，以实现更好的流水化。

三、可重构分区

图 3.5 以稠密矩阵乘法的 RP 为例，展示了 RP 的内部结构。除了 HLS 编写的加速内核外，最重要的便是 `rm_comm_box` 数据移动器 IP 核。它具备用于从 DDR/BRAM 等存储器读写数据的 AXIMM 接口（DMA），一个为加速器提供输入的 AXIS 输出端口，以及一个用于从加速器读取数据的 AXIS 输入端口。该 IP 核包含两个引擎：mm2s 引擎通过 AXIMM 从内存地址读取数据，并将数据通过 AXIS 流输出端口提供出去；s2mm 则是相反的步骤。

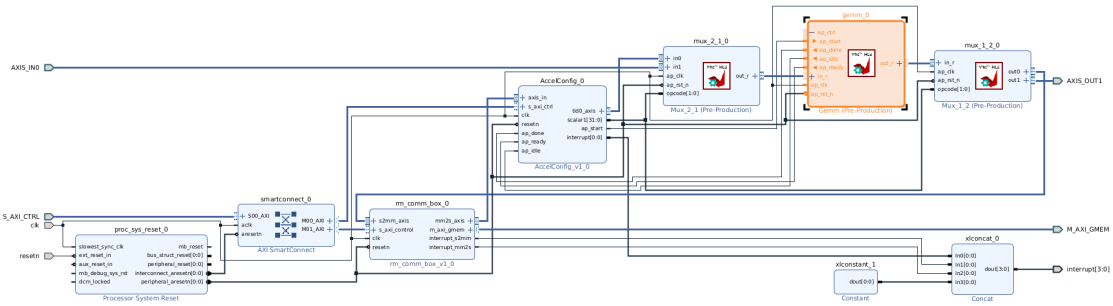

图 3.5　加载了稠密矩阵乘法 RM（橙色高亮模块）的 RP_2 内部设计。

由于 RP 的输入/输出来源不确定，可能是 DDR AXIMM 也可能是其他 RP 的 AXIS 端口，我们编写了两个数据选择器。数据选择器的控制信号通过 `rm_comm_box` 提供的虚拟 AXIS 通道传输（由主机应用程序负责），这有别于其他 AXI Lite 传输的控制信号。

第二节　软件系统架构

软件系统架构是建立在 KV260 的 ARM 处理器之上，负责管理硬件资源、调度计算任务以及提供用户交互接口。整个软件栈可以划分为操作系统层面、Xilinx 运行时（XRT）应用层面以及顶层的任务调度逻辑。

在操作系统层面，KV260 的 CPU 上部署了 Ubuntu Linux 操作系统。Linux 系统为上层应用提供了稳定的运行环境和丰富的系统服务，包括内存管理、进程调度以及设备驱动程序等。FPGA 作为一种可编程设备，其驱动和管理由 Linux 内核模块以及 Xilinx 提供的运行时库共同完成。

XRT 是连接用户空间应用程序与 FPGA 加速硬件之间的关键桥梁，提供了一套标准的 API，应用程序可以通过这些 API 来管理 FPGA 设备、分配和迁移数据缓冲区、加载 FPGA 比特流（包括用于动态重构的部分比特流）以及控制加速内核的执行。在本系统中，我们编写的 C/C++ 应用程序利用 XRT 提供的接口，实现了对 FPGA 上动态加载的矩阵运算模块的调用和管理，从而实现异构加速计算。

第四章　可重构模块硬件实现

本章详细阐述构成动态可重构矩阵乘法加速系统核心计算能力的可重构模块（RM）的设计与实现过程。这些模块是专门为加载到第三章定义的三个可重构分区（RP）中而设计的。我们将采用 Xilinx Vitis 高层次综合（HLS）工具，利用 C/C++ 语言进行算法描述和硬件优化。对于每个关键的 RM（稀疏矩阵解压、稠密矩阵乘法、稀疏矩阵压缩），本章将覆盖其核心算法分析、HLS 实现策略、接口设计、关键优化技术以及最终的综合与实现结果（基于目标平台 Xilinx Kria KV260 的 Zynq UltraScale+ MPSoC PL 部分的资源特性）。

第一节　Vitis HLS 设计方法论概述

Vitis HLS 作为一种高层次综合工具，允许设计者使用 C、C++ 或 OpenCL C 等高级语言描述硬件行为，然后将其自动转换为寄存器传输级（RTL）代码。这种方法显著提高了设计效率，使得复杂的算法能够更快地在 FPGA 上实现。在本系统中，所有可重构模块均采用 C/C++ 进行开发。核心设计策略是首先实现算法的功能正确性，然后通过迭代地应用 HLS 优化指令（pragmas）来提升性能和资源利用率。关键的优化指令包括用于实现指令级并行的 `PIPELINE` 指令，用于循环展开以实现数据级并行的 `UNROLL` 指令，用于优化存储器访问的 `ARRAY_PARTITION` 指令，以及用于定义模块接口的 `INTERFACE` 指令。设计目标是为每个 RM 生成高效的、能够充分利用 FPGA 并行性和流水线性质的硬件实现。

第二节　稀疏矩阵解压/压缩模块

稀疏矩阵解压模块负责将以特定稀疏格式存储的矩阵转换为稠密矩阵格式。其核心算法依赖于稀疏格式的定义。以 CSR（Compressed Sparse Row）格式为例，输入数据通常包括值数组（values）、列索引数组（column indices）和行指针数组（row pointers）。稀疏矩阵压缩模块的功能与解压模块相反，它将稠密矩阵转换为指定的稀疏格式。

稀疏矩阵解压模块的算法逻辑是遍历行指针数组，确定每行非零元素的数量和位置，然后根据列索引数组和值数组将这些非零元素填充到稠密矩阵的相

应位置，其余位置则填充零。在 HLS 实现中，通常会涉及嵌套循环：外层循环遍历行，内层循环遍历该行内的非零元素。为了高效访问输入稀疏数据（通常存储在主存中），模块通过 AXIMM 主接口读取。输出的稠密矩阵同样通过 AXIMM 主接口写回主存，或者通过 AXIS 接口流式传输给下一个处理模块。输入输出的接口选择由数据选择器实现，其选择信号由主机程序指定，并在传输矩阵数据前先行传至 FPGA 上。

稀疏矩阵压缩模块模块首先遍历输入的稠密矩阵，识别所有非零元素，并记录它们的值和索引。随后，根据目标稀疏格式（如 CSR）的要求，将这些信息组织成相应的数组结构。例如，生成 CSR 格式需要统计每行的非零元素个数以构建行指针数组，同时记录非零元素的值和列索引。

第三节　稠密矩阵乘法模块

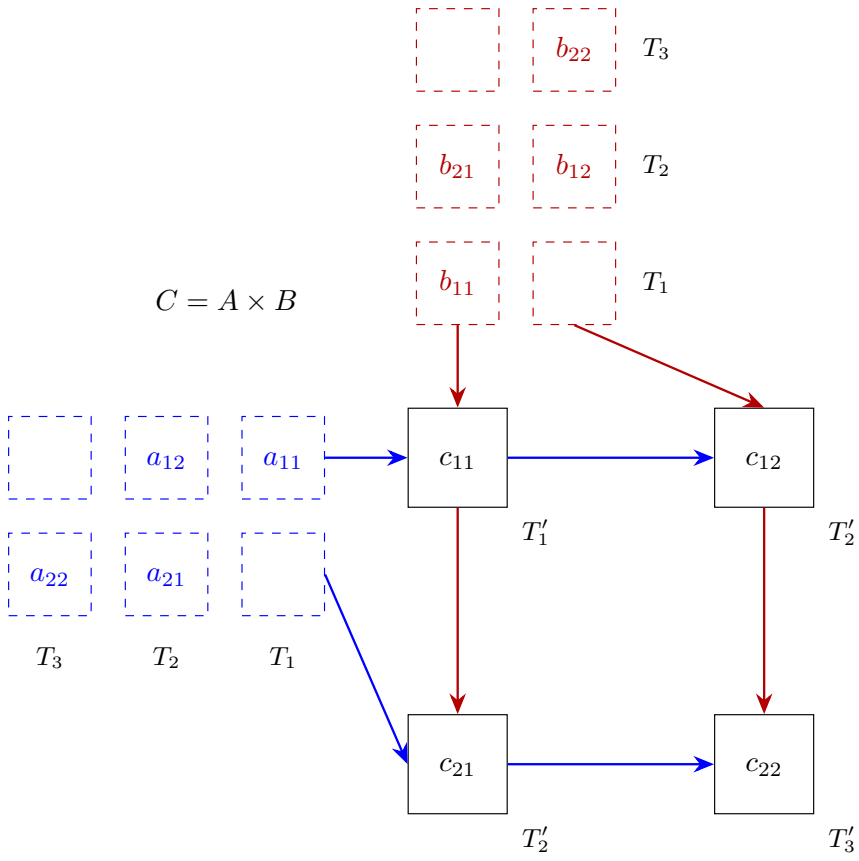

图 4.1　脉动阵列算法作用于 2×2 矩阵乘法。

稠密矩阵乘法模块是本加速系统的核心计算单元之一，其设计采用了脉动阵列（Systolic Array）结构，并结合分块（Tiling）策略以处理大规模矩阵。

如图 4.1 所示，在矩阵乘法 $C = A \times B$ 的脉动阵列算法中，每个 PE 计算矩阵

C 的对应元素，其从左侧和上方接收矩阵 A 和矩阵 B 的元素，并在下一时钟周期传递给右侧和下方的 PE。PE 将每个时钟周期输入的元素相乘并累加，最终便得到了矩阵 C 的元素值。输入矩阵 A 和 B 分别对行和列进行 `ARRAY_PARTITION`（如此便能对不同行/列的元素并行访问），每行/列相较于上一行/列延迟一个周期送入 PE。

脉动阵列因其规整的数据流和高度的并行性，非常适合在 FPGA 上实现矩阵乘法。算法将输入矩阵 A 和 B 分割成小块（tiles），然后将这些小块数据送入脉动阵列进行计算。每个处理单元（Processing Element, PE）在脉动阵列中执行乘加运算。分块策略有助于管理片上存储资源（如 BRAM），使得模块可以处理远大于片上存储容量的矩阵。HLS 实现中，需要精心设计数据加载、PE 阵列的计算逻辑以及结果写回的控制流程。

针对脉动阵列的 PEs，其内部的乘加操作循环体应用 `PIPELINE` 指令是标准做法，由 Vitis 自动处理而无需手动设置。为了实现 PE 之间的数据并行流动，相关的循环通常会被完全 `UNROLL`。用于缓存输入矩阵分块的片上存储器（BRAMs）会使用 `ARRAY_PARTITION` 指令进行分区，以提供足够的并行读写带宽。接口方面，输入矩阵数据（或其分块）通常通过 AXIS 从接口输入，这便于从前级模块（如解压模块）或 PS 端高效接收数据。计算结果（稠密矩阵或其分块）通过 AXIS 主接口输出，可以流向后级模块（如压缩模块）或 PS 端。对于直接从主存加载分块或写回分块的场景，也会配置 AXIMM 主接口。输入输出的接口选择同样由数据选择器实现。

第四节　模块间的协同与动态特性

以上设计的可重构模块均具备标准的 AXI 接口（AXIMM 用于内存访问，AXIS 用于流式数据传输），这使得它们不仅可以独立工作，也可以在 FPGA 的三个可重构分区中灵活组合，通过 AXIS 接口形成高效的数据处理流水线。例如，稀疏矩阵乘法任务可以通过动态加载稀疏解压 RM、稠密乘法 RM 和稀疏压缩 RM，并使它们通过 AXIS 接口依次串联工作。矩阵数据的输入输出也进行了流式处理，在输入/输出矩阵的元素前，会输入/输出矩阵的元数据（行数、列数、稀疏矩阵的非零元素个数）。每个 RM 都作为独立的编译单元生成部分比特流，为第三章所述的动态重构系统提供了基础。

第五章　软件系统实现与集成

在前两章分别完成系统总体架构设计和可重构硬件模块实现后，本章将聚焦于运行在 Xilinx Kria KV260 平台处理器系统（PS）上的软件系统实现与集成。这包括配置嵌入式 Linux 操作环境以及编写主机端 C/C++ 应用程序。用户通过 Xilinx 提供的 `xmutil` 命令管理 FPGA 上的可重构模块（RM），精确地调用和协调 FPGA 上的硬件加速任务。应用程序借助 XRT 在 PS 和 PL 间传输数据。

第一节　嵌入式 Linux 环境配置

本系统采用 Xilinx 官方提供的 Ubuntu Linux，其针对 Zynq UltraScale+ MPSoC 进行了优化，并内建了对 FPGA 可编程逻辑（PL）以及动态功能交换（DFX）的支持。启动 KV260 并加载 Linux 系统后，需进行基础的系统配置，例如网络连接、用户管理，并安装必要的开发工具链，为主机应用程序编译提供支持。

第二节　主机应用程序设计与架构

主机应用程序采用 C/C++ 语言编写，旨在提供一个用户友好的接口来调用 FPGA 实现的各种矩阵运算任务。

以稀疏矩阵乘法为例，在运行主机程序前，用户需通过 `xmutil` 命令装载相应 RM。主机应用程序先从本地文件读取矩阵数据，随后启动 3 个 RM（稀疏矩阵解压、稠密矩阵乘法、稀疏矩阵压缩）。借助 XRT，主机程序先分别指定这 3 个 RM 的输入输出选择信号：稀疏矩阵解压的输入为 AXIMM 接口的 DDR 内存，输出为 AXIS 接口连接稠密矩阵乘法的输入；稠密矩阵乘法的输出为 AXIS 接口连接稀疏矩阵压缩的输入；稀疏矩阵压缩的输出为 AXIMM 接口的 DDR 内存。随后，主机程序向稀疏矩阵解压 RM 传输两个稀疏矩阵的数据流，该数据流最终从稀疏矩阵压缩的 RM 流出，这便是稀疏矩阵乘法的结果，由主机程序接收后保存至本地文件。主机程序最后关闭 3 个 RM。用户可通过 `xmutil` 卸载相应 RM。

第六章　实验结果与分析

本章旨在对前述章节设计并实现的动态可重构 FPGA 矩阵乘法加速系统进行全面的测试与评估。我们将通过一系列实验来验证系统的功能正确性、测量动态重构的关键性能指标、评估核心矩阵运算任务的加速效果（与 CPU 基准对比），并量化硬件资源的利用率。本章的目标是提供实验证据，证明所提出设计的有效性和实用性。

第一节　实验环境与配置

实验硬件平台为 Xilinx Kria KV260 Vision AI Starter Kit，其核心是 Zynq UltraScale+ MPSoC。使用的 Vitis HLS 与 Vivado 工具链版本均为 2022.1。可编程逻辑（PL）部分根据第四章所述的可重构模块设计，在综合实现后，各模块的工作时钟频率设定为 250 MHz。处理器系统（PS）端的 ARM Cortex-A53 四核处理器运行频率为 1.33 GHz。软件环境方面，KV260 上部署了 Xilinx 特制的 Ubuntu 22.04 LTS 操作系统，主机应用程序以及 CPU 基准程序（单线程的朴素矩阵乘法）均使用 GCC 11.4.0 进行编译，并开启 `-O1` 优化。

第二节　系统功能验证

功能验证是确保系统按预期工作的首要步骤。我们针对第五章中描述的所有加速计算任务进行了测试：稀疏矩阵解压为稠密矩阵、稠密矩阵压缩为稀疏矩阵、稀疏矩阵格式转换（例如，从假设的 COO 格式 RM 到 CSR 格式 RM 的转换流程）、稠密矩阵乘法（结果分别为稠密和稀疏），以及稀疏矩阵乘法（结果分别为稠密和稀疏）。实验结果表明，所有 FPGA 加速任务的输出均与 CPU 参考结果一致（在单精度浮点数允许的误差范围内），从而验证了整个系统（包括 HLS 模块设计、AXI 接口、XRT 调用、DPR 流程及主机应用程序逻辑）的功能正确性。

第三节　GEMM 加速性能分析

我们对核心矩阵运算任务在 FPGA 上的加速性能进行了评估，并与纯 CPU 执行时间进行了对比。性能指标主要关注端到端执行时间，即包括数据传输与 FPGA 内核执行的总时间。

对于稠密矩阵乘法（GEMM），我们测试了 128×128 单精度浮点稠密矩阵乘法，FPGA 上脉动阵列的每个块大小为 12×12。CPU 执行时间约为 32 毫秒，而 FPGA 上的稠密矩阵乘法 RM 总共耗时 1.3 毫秒，实现了约 25× 的加速比。

第四节　硬件资源利用率

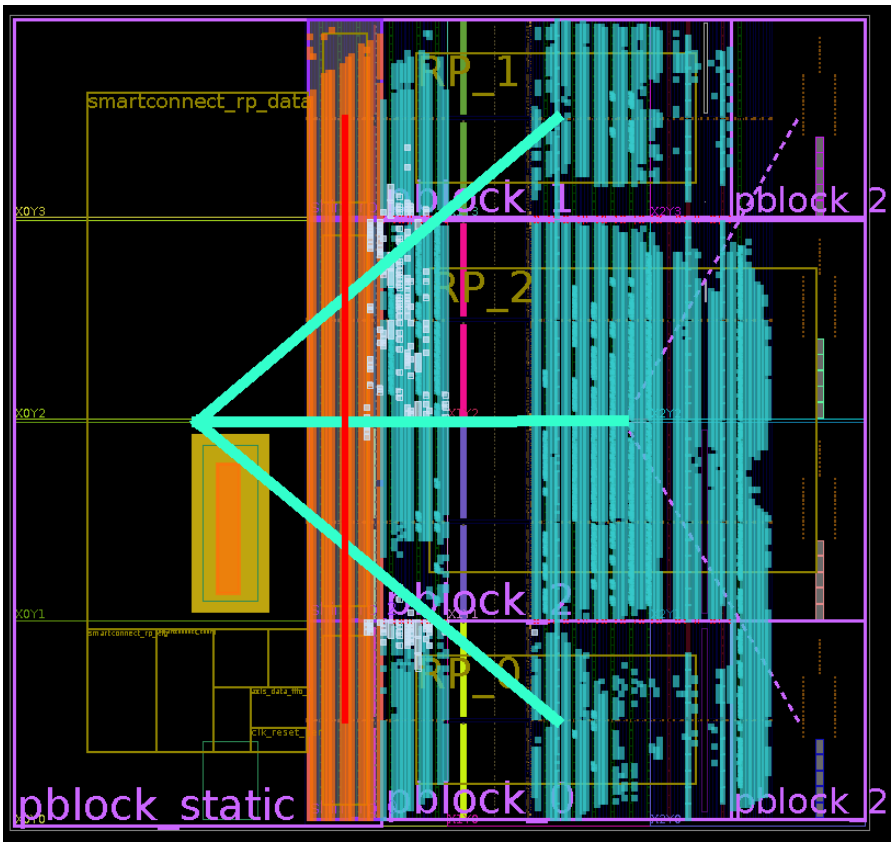

图 6.1　FPGA 实现。RP_0 上实现了 COO 矩阵解压 RM，RP_1 上实现了 COO 矩阵压缩 RM，RP_2 上实现了稠密矩阵乘法 RM。

我们将 COO 矩阵解压、COO 矩阵压缩以及稠密矩阵乘法作为一个整体，其在 FPGA 上的实现结果如图 6.1 所示，其能耗占用与资源利用分别由图 6.2 和图 6.3 所示。可以看出，为支持 128×128 单精度浮点矩阵，我们几乎消耗了板上所有的 BRAM。更大的矩阵规模将无法在这块 KV260 开发版上实现。可以说，板上的全部资源都已被高效利用。如果增加分块脉动阵列的块大小，RP 将无法在 250 MHz 的频率下运行，这是因为更大的脉动阵列会占用更高比例的 LUT 和

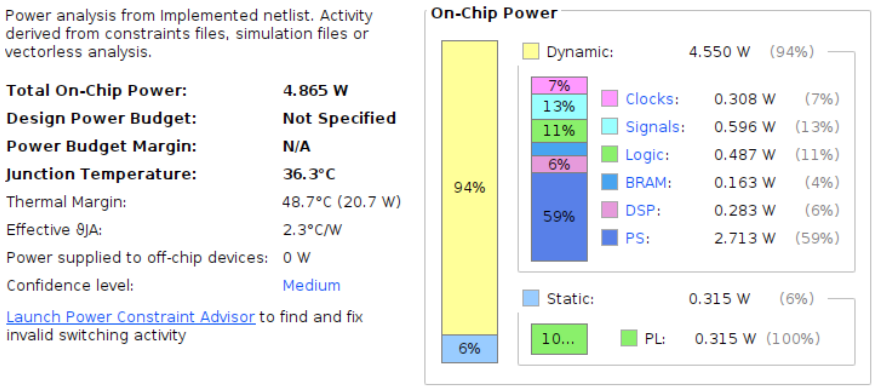

图 6.2　能耗占用。

Resource	Utilization	Available	Utilization %
LUT	50635	115808	43.72
LUTRAM	4207	56960	7.39
FF	69069	234240	29.49
BRAM	131	144	90.97
DSP	313	1248	25.08
BUFG	7	352	1.99

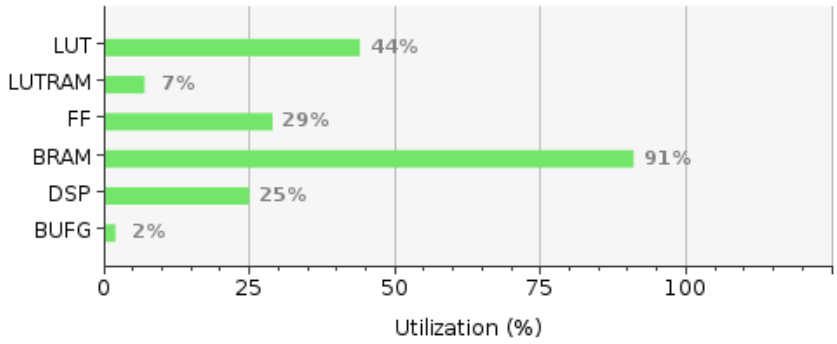

图 6.3　资源利用。

DSP 等资源，以至于 Vivado 的布线无法满足时序要求。

第五节　讨论与分析总结

实验结果综合表明，所设计的动态可重构 FPGA 矩阵乘法加速系统成功实现了预期的功能，并在 GEMM 运算任务上展现了相对于 ARM CPU 的显著加速效果。稠密矩阵乘法的脉动阵列设计和稀疏处理流水线的构建，有效地利用了 FPGA 的并行计算能力。动态部分重构机制虽然引入了一定的时间开销（约 5 毫秒），但其赋予系统的灵活性和适应性，使其能够高效应对多种计算需求和数据格式，这对于需要处理多样化任务的场景具有重要价值。

第七章　总结

第一节　本论文主要工作总结

本论文围绕在 Xilinx Kria KV260 多处理器系统芯片（MPSoC）上构建一个高性能、高灵活性的动态可重构矩阵乘法加速系统展开研究与实现。面对传统固定功能加速器在处理多样化矩阵运算任务及适应不同数据格式方面的局限性，本文提出并成功实现了一个基于动态部分重构（DFX）技术的异构计算解决方案。

首先，在系统总体设计层面（第三章），我们规划了基于 KV260 的软硬件协同架构。硬件上，在 FPGA 的可编程逻辑（PL）部分划分了静态区域和三个独立的可重构分区（RP）。静态区域负责基础平台支持，而 RP 则用于动态加载不同的计算模块。软件上，在处理器系统（PS）端的 ARM 处理器上部署了 Ubuntu Linux 操作系统，并利用 Xilinx Runtime (XRT) 作为主机应用程序与 FPGA 硬件交互的桥梁。

其次，在可重构模块硬件实现层面（第四章），我们采用 Vitis 高层次综合（HLS）语言，设计并优化了一系列核心矩阵运算的可重构模块（RM）。这些模块包括稀疏矩阵解压模块、采用脉动阵列结合分块策略的稠密矩阵乘法模块以及稀疏矩阵压缩模块。通过精心设计的并行与流水线优化，这些 RM 能够高效利用 FPGA 资源。模块间通过 AXI4 Stream (AXIS) 协议实现高速数据互联，并通过 AXI4 Memory Mapped (AXIMM) 协议直接访问 KV260 的系统内存。

再次，在软件系统实现与集成层面（第五章），我们阐述了在 KV260 的 ARM 处理器上配置 Linux 环境，介绍了主机端 C++ 应用程序的设计，实现异构计算。

最后，通过一系列实验（第六章），我们对所构建的系统进行了全面的功能验证、稠密矩阵运算任务的加速性能评估，并分析了系统的灵活性和硬件资源利用率。

第二节　存在的不足与挑战

尽管本系统取得了一定的成果，但在研究和实现过程中也清晰地揭示了一些固有的不足之处以及面临的技术挑战。一个核心的考量在于动态部分重构（DPR）

本身引入的开销。具体而言，加载部分比特流所需的时间，在实验中表现为 5 毫秒左右，对于那些执行时间极短或者需要极其频繁切换可重构模块（RM）的计算任务而言，这一延迟可能显著影响甚至抵消 FPGA 的加速效益。与此同时，数据传输瓶颈也是限制整体性能提升的关键因素。尽管采用了 AXIMM 协议实现直接内存访问，PS 与 PL 之间的数据交互带宽和延迟，以及在某些复杂流水线中 RM 之间可能需要通过 DDR 进行的中间数据交换，依然对加速比的上限构成了制约，特别是对于那些计算密度不高、数据吞吐量大的模块。

此外，Kria KV260 作为一款入门级的 MPSoC 平台，其 FPGA 内部的逻辑资源、DSP 单元以及 BRAM 总量是相对有限的。这一硬件平台的资源约束，自然地限制了单个可重构模块所能达到的最大复杂度，例如稠密矩阵乘法中脉动阵列的规模，同时也可能制约了可同时部署的 RP 数量或每个 RP 的大小，从而影响了系统整体的并行处理能力。设计和调试 DFX 系统本身也带来了额外的复杂性。相较于传统的静态 FPGA 设计流程，DFX 要求更精细的分区规划、严格的接口隔离规则，以及更为繁琐的部分比特流生成、管理与验证过程，这些都对开发效率和调试难度提出了更高的要求。最后，从应用范围来看，当前实现的 RM 库虽然覆盖了基础的核心矩阵运算，但其在支持更多高级矩阵算法或更广泛、更特殊的稀疏数据格式方面，其完备性仍有提升空间，这可能限制了系统在某些特定专业领域的直接适用性。

第三节　未来工作展望

基于当前工作的成果和已识别的不足，未来的研究可以向多个富有前景的方向拓展和深化。首要的努力方向应聚焦于进一步降低 DPR 开销并提升系统的动态响应能力。这可能涉及对更先进重构技术的探索，例如研究配置数据压缩方法、利用未来 Xilinx 工具链可能支持的更细粒度重构机制，或者探索硬件加速的配置管理单元设计，乃至结合预测性加载和智能化调度策略，以期在用户层面隐藏或显著减少 DPR 切换带来的延迟。与此同时，持续优化数据通路和内存访问效率也至关重要。这包括深入研究片上高速缓存（如 BRAM/URAM）的更高效管理和分配策略，探索更先进的 DMA 传输模式以减少 CPU 干预和传输延迟，并强化 RM 之间通过 AXIS 等方式的直接流式数据传输，最大限度地避免数据经由外部 DDR 的低效中转。

在提升系统功能性和易用性方面，可以致力于扩展可重构模块（RM）的功

能覆盖范围和设计灵活性。这包括开发一个更加丰富和多样化的 RM 库，以支持如矩阵分解、迭代求解器、特征值计算等更高级的矩阵运算，并兼容更多标准或用户定义的稀疏数据格式。同时，研究高度参数化的 RM 设计方法，使得 RM 能够在一定范围内通过寄存器配置等轻量级方式调整其内部并行度、数据位宽或特定算法路径，从而在完全 DPR 和静态配置之间找到更优的平衡点，提升模块的复用性和适应性。软件层面，开发更高层次的抽象接口或领域特定语言（DSL）将极大简化用户对复杂 DFX 系统的编程和任务部署，未来的系统甚至可以集成基于机器学习的智能调度器，使其能够根据实时工作负载特性和系统状态自动优化 RM 的选择、加载时序及 RP 资源分配。

最后，将本研究中积累的设计理念和实践经验应用于新兴的 FPGA 架构和异构计算平台，无疑是一个充满机遇的研究方向。例如，Xilinx Versal ACAP 等新一代器件集成了更为强大的 AI 引擎、智能引擎以及更成熟和灵活的 DFX 支持能力，在这些新平台上重构和扩展本系统，有望实现性能和能效的飞跃。此外，对系统在不同工作负载和 RM 配置下的功耗进行细致的建模、分析与优化，特别是针对边缘计算等功耗敏感应用场景，开发面向低功耗的 DFX 策略和 RM 设计方法，也将是未来工作中一个具有实际应用价值的研究点。

参 考 文 献

[1] LECUN Y, BENGIO Y, HINTON G. Deep learning[J]. Nature, 2015, 521(7553): 436-444.

[2] GOLUB G H, VAN LOAN C F. Matrix computations[M]. JHU Press, 2013.

[3] CHETLUR S, WOOLLEY C, VANDERMERSCH P, et al. cudnn: Efficient primitives for deep learning[A]. 2014.

[4] KEPNER J, GILBERT J. Graph algorithms in the language of linear algebra[M]. SIAM, 2011.

[5] HENNESSY J L, PATTERSON D A. Computer architecture: a quantitative approach[M]. Elsevier, 2011.

[6] OWENS J D, HOUSTON M, LUEBKE D, et al. Gpu computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.

[7] BELL N, GARLAND M. Implementing sparse matrix-vector multiplication on throughput-oriented processors[C]//Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 2009: 1-11.

[8] XILINX. Ultrascale architecture and product data sheet: Overview (DS890)[Z]. 2025.

[9] NANE R, SIMA V M, PILATO C, et al. A survey and evaluation of FPGA high-level synthesis tools[J]. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2015, 35(10): 1591-1604.

[10] ZHUO L, PRASANNA V K. Sparse matrix-vector multiplication on FPGAs[C]//Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). 2005: 63-74.

[11] BLODGET B, JAMES-ROXBY P, KELLER E, et al. A self-reconfiguring platform[C]//Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2003: 565-574.

[12] XILINX. Vivado design suite user guide: Dynamic function exchange (UG909)[Z]. 2024.

[13] VIPIN K, FAHMY S A. FPGA dynamic and partial reconfiguration: A survey of

architectures, methods, and applications[J]. ACM Computing Surveys (CSUR), 2018, 51(4): 1-39.

[14] BECKHOFF C, KOCH D, TORRESEN J. Go ahead: A partial reconfiguration framework[C]//Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines. IEEE, 2012: 37-44.

[15] KOCH D, TORRESEN J, BECKHOFF C, et al. Partial reconfiguration on FPGAs in practice—tools and applications[C]//Proceedings of the International Conference on Architecture of Computing Systems (ARCS). IEEE, 2012: 1-12.

[16] SAAD Y. Iterative methods for sparse linear systems[M]. SIAM, 2003.

[17] XILINX. Kria KV260 vision AI starter kit user guide (UG1089)[Z]. 2024.

[18] DE FINE LICHT J, KWASNIEWSKI G, HOEFLER T. Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis[C]//Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). 2020: 244-254.

[19] PUŞCAŞU A, CIOBANU C B, BUIU O. Systolic array matrix multiplication accelerator[C]//Proceedings of the International Semiconductor Conference (CAS). IEEE, 2024: 207-210.

[20] DORRANCE R, REN F, MARKOVIĆ D. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs[C]//Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA). 2014: 161-170.

[21] BOBDA C, HARTENSTEIN R. Introduction to reconfigurable computing: architectures, algorithms, and applications: Vol. 1[M]. Springer, 2007.

[22] GONZALEZ I, LOPEZ-BUEDO S, GOMEZ F J, et al. Using partial reconfiguration in cryptographic applications: an implementation of the idea algorithm[C]//Proceedings of the International Conference on Field Programmable Logic and Applications (FPL). 2003: 194-203.

[23] RAM R S, PRABHAKER M L C, SURESH K, et al. Dynamic partial reconfiguration enchanced with security system for reduced area and low power consumption [J]. Microprocessors and Microsystems, 2020, 76: 103088.

[24] MITOLA J. Software radio architecture: a mathematical perspective[J]. IEEE Journal on selected areas in communications, 2002, 17(4): 514-538.

附录 A　运行实例

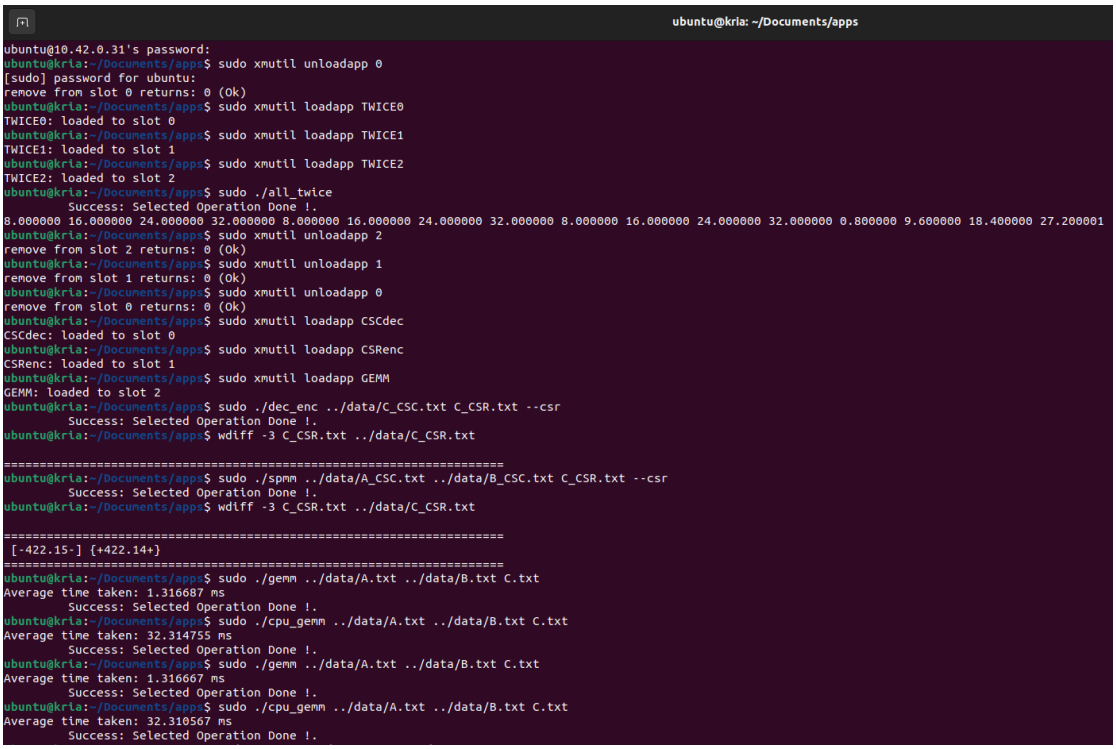

图 A.1　128×128 矩阵乘法的运行实例。

在图 A.1 中，我们先用 `xmutil` 命令装载了三个 `TWICE` RM，它们的功能是将输入流翻倍后输出。我们随后执行 `all_twice` 应用以测试 RM 间的互联，此应用定义 RP_0 的输入为 DDR，RP_0 输出为 RP_1 的输入，RP_1 输出为 RP_2 的输入，RP_2 的输出为 DDR，也就是将最开始的输入翻 8 倍后输出。

我们接下来卸载三个 `TWICE` RM，装载 `CSCdec`（CSC 矩阵解压 RM），`CSRenc`（CSR 矩阵压缩 RM），`GEMM`（稠密矩阵乘法 RM）。`dec_enc` 应用将 `CSCdec` 的输出连接 `CSRenc` 的输入，实现 CSC 格式转化为 CSR 格式。`spmm` 应用执行稀疏矩阵乘法，其输入为 2 个 CSC 矩阵，输出为 CSR 矩阵。`wdiff` 工具用于比对 FPGA 输出与真实值。`gemm` 和 `cpu_gemm` 应用分别测试 FPGA 和 CPU 上稠密矩阵乘法的运行耗时。

附录 B　稠密矩阵乘法加速

表 B.1　不同矩阵大小下的加速比（12×12 脉动阵列分块）

矩阵大小	CPU 耗时	FPGA 耗时	加速比
12×12	0.015	0.012	1.3×
16×16	0.034	0.019	1.8×
32×32	0.26	0.047	5.5×
36×36	0.37	0.052	7.1×
60×60	1.7	0.17	10×
64×64	2.0	0.24	8.3×
96×96	7.3	0.56	13×
120×120	15	1.0	15×
127×127	17	1.3	13×
128×128	32	1.3	25×

表 B.1 展示了对各种矩阵大小获得的加速比。CPU 耗时从 120×120 至 128×128 矩阵乘法翻了一倍。缓存未命中（Cache Miss）是影响其性能的关键因素，主要分为容量性未命中与冲突性未命中。当矩阵维度 N 为 2 的幂次方（如 128）时，对于行主序存储的矩阵 $\boldsymbol{B}$，在按列访问（计算 $\boldsymbol{B}_{kj}$ 时 k 递增）过程中，内存访问步长为 $N \times \text{sizeof(element)}$。若元素为 4 字节浮点数，$N = 128$ 时的步长为 512 字节。此类 2 的幂次方的步长极易与缓存的组索引机制（通常基于地址的低位比特）产生不良耦合，导致同一数据流（如 B 矩阵的一列）中的多个缓存行映射到数量极为有限的缓存组上。当这些并发访问的缓存行数量超过目标组的相联度时，将引发剧烈的缓存颠簸（Thrashing）。

与 $N = 128$ 的情形不同，当矩阵维度 $N = 127$ 时，其列访问步长并非 2 的幂次方。这种非 2 的幂次方的步长使得内存地址在映射到缓存组时呈现出更优的分布特性，从而有效规避了因地址对齐引发的系统性缓存组冲突。因此，尽管 $N = 127$ 的矩阵同样面临容量性未命中的挑战，但其性能表现显著优于 $N = 128$ 的情况，后者额外承受了严重的冲突性未命中负担。

附录 C　复现说明

表 C.1　编译环境

OS	Ubuntu 22.04.5 LTS
CPU	Intel Core i9-13900HX
Memory	62.6 GiB

HLS 代码由 Vitis 2022.1 综合成 RTL IP，随后用 Vivado 2022.1 执行综合、实现和比特流生成，运行环境如表 C.1 所示。在此环境下，Vivado 流程需运行 30 分钟并占用 55 GiB 内存。若希望减小内存占用，可修改流程的并行线程数量。同时，我们在 github.com/fjtcin/dfx-3rp-bin 提供了生成的二进制文件，可直接使用。

致　谢

我要特别感谢宫磊老师，在本项研究从选题、设计到最终完成的各个阶段，他都给予了耐心细致的指导和富有启发性的见解。宫老师的悉心帮助对本工作的顺利完成至关重要。

2025 年 5 月