一种数据流驱动的混合粒度可重构阵列

结构设计与仿真实现

摘 要

近年来，以数据为中心的计算需求变得越来越大，对计算平台提出了更高的要求。可重构计算平台充分利用了空间计算的特性和软件可编程性，可以满足数据密集并且灵活多变的计算场景。但针对可重构系统的研究还存在一些如下问题，(1)粗粒度阵列薄弱的控制能力使得其不能完美支持复杂应用；(2)基于动态重构的配置流驱动机制使系统频繁切换配置，降低了任务执行效率。

基于以上问题，本文研究一种数据流驱动的混合粒度阵列，通过对阵列进行静态重构，将执行任务的操作码提前配置到阵列中并进行互连关系的固定，再结合数据流驱动原理，使数据驱动静态阵列运行，实现操作码和操作数的融合，从而完成计算任务，静态重构和数据流的结合减少了配置切换次数和任务的执行时间；另外，通过对算法的核心代码进行分析，拆分代码中的数据流图(DFG)和控制流图(CFG)，使用粗粒度可重构阵列实现代码中的数据流图，细粒度可重构阵列实现代码中的控制流图，提高阵列的控制能力和执行的自主化，两种技术的结合实现了对计算密集型和控制密集型算法的支持。此外，结构支持多发射循环迭代和空间循环展开等优化技术，使得阵列具有较高的计算效率。

最后，对混合粒度的阵列进行行为建模，设计仿真器。通过软件语言描述模块功能并进行编程实现，搭建系统验证和测试平台，对算法进行模拟，收集数据并分析。从结果看，混合粒度可重构阵列很好的支持了复杂算法，同时实现了理想的执行效率。

关键词：数据流，混合粒度可重构阵列，仿真器

**DESIGN AND SIMULATION IMPLEMENTATION**

**OF A HYBRID-GRAINED RECONFIGURABLE ARCHITETURE BASED ON DATA STREAM**

**ABSTRACT**

~~The advent of the era of artificial intelligence has made data-centric computing demands more and more large, and traditional computing platforms have been unable to meet such needs.~~

KEY WORDS: data-flow,hybrid-grained reconfigurable array,simulator

目 录

**第一章 绪 论** 1

1.1 课题研究背景 1

1.2 研究现状 2

1.3 主要研究思路 10

1.4 论文组织结构 11

**第二章 可重构技术及数据流驱动技术** 12

2.1 可重构技术 12

2.1.1 可重构计算执行机制 12

2.1.2 可重构计算其他关键技术 15

2.2 数据流驱动技术 19

2.2.1 数据流计算机基本原理 20

2.2.2 数据流驱动执行机制 20

2.3 混合粒度阵列的优势 22

2.4 本章小结 23

**第三章 混合粒度可重构阵列结构设计** 24

3.1 系统整体设计 24

3.1.1 设计思想 24

3.1.2 模块整体介绍 25

3.2 粗粒度处理单元结构设计 27

3.2.1 输入输出buffer 29

3.2.2 可配置ALU单元 30

3.2.3 本地寄存器 31

3.2.4 本地配置器 32

3.2.5 粗粒度处理单元数据通路设计 34

3.3 细粒度处理单元结构设计 36

3.3.1 单元内部构成 37

3.3.2 细粒度控制算子 38

3.3.3 细粒度处理单元内部数据通路设计 39

3.4 阵列反馈机制 40

3.4.1 数据流和循环间依赖 40

3.4.2 多层次反馈机制 42

3.5 混合粒度阵列的优化技术 43

3.5.1 多发射循环迭代技术 43

3.5.2 空间展开循环迭代技术 44

3.6 应用映射实例 45

3.6.1 矩阵乘映射到两层循环方案 45

3.6.2 矩阵乘映射到一层循环方案 47

3.7 本章小结 49

**第四章 混合粒度可重构阵列的仿真实现** 51

4.1 仿真原理概述 51

4.2 仿真器整体框架 52

4.3 仿真器实现 54

4.3.1 模块实现 54

4.3.2 系统级仿真平台实现 60

4.4 仿真流程 60

4.4.1 算法映射 61

4.4.2 阵列运行 62

4.5 本章小结 63

**第五章 实验与分析** 65

5.1 实验目的 65

5.2 常见算法映射 65

5.2.1 矩阵乘运算 65

5.2.2 sobel边缘检测算法 66

5.2.3 图像中值滤波 68

5.2.4 GCD最大公约数 69

5.3 仿真结果及分析 70

5.4 本章小结 73

**第六章 总结和展望** 74

6.1 论文总结 74

6.2 后续工作展望 75

**参 考 文 献** 76

**致 谢** 80

**攻读硕士学位期间已发表或录用的论文** 81

图 录

[图1‑1 CPLD中与或门阵列 3](#_Toc533616838)

[图1‑2 FPGA中查找表结构 4](#_Toc533616839)

[图1‑4 Grap结构框图 5](#_Toc533616840)

[图1‑5 Grap架构组织图 5](#_Toc533616841)

[图1‑6 Morphsys结构框图 6](#_Toc533616842)

[图1‑7 RC内部细节图 7](#_Toc533616843)

[图1‑8 XPP架构图 8](#_Toc533616844)

[图1‑9 WAVECACHE结构图 9](#_Toc533616845)

[图1‑10 WAVECACHE框架图 9](#_Toc533616846)

[图2‑1三种计算平台在灵活性和计算效率上的对比[16] 12](#_Toc533616847)

[图2‑2可重构计算系统的组成[18] 13](#_Toc533616848)

[图2‑3三种计算系统在某一应用上的时空利用图[18] 14](#_Toc533616849)

[图2‑4互连网络可配置原理图 17](#_Toc533616850)

[图2‑5互连网络全连接 17](#_Toc533616851)

[图2‑6互连网络局部连接 18](#_Toc533616852)

[图2‑7蝶式变换的数据流图 21](#_Toc533616853)

[图2‑8动态数据流情况下的输入弧 22](#_Toc533616854)

[图3‑1可重构系统的设计流程图[19] 25](#_Toc533616855)

[图3‑2混合粒度阵列架构整体图 26](#_Toc533616856)

[图3‑3可重构阵列系统图 27](#_Toc533616857)

[图3‑4粗粒度处理单元内部结构简图 28](#_Toc533616858)

[图3‑5输入输出端口数据线位宽编码 29](#_Toc533616859)

[图3‑6 PE内部输入buffer组织图 30](#_Toc533616860)

[图3‑7循环代码变量作用域示意图 32](#_Toc533616861)

[图3‑8 PE内部配置寄存器编码格式 33](#_Toc533616862)

[图3‑9 CPE内部数据通路图 34](#_Toc533616863)

[图3‑10变量作用域和生存周期示例 35](#_Toc533616864)

[图3‑11分支跳转代码及数据控制流图实例 36](#_Toc533616865)

[图3‑12 细粒度处理单元内部结构简图 38](#_Toc533616866)

[图3‑13细粒度处理单元内部数据通路 40](#_Toc533616867)

[图3‑14 静态和动态数据流执行方式 41](#_Toc533616868)

[图3‑15处理单元之间的反馈图 42](#_Toc533616869)

[图3‑16处理单元内部的反馈示意图 43](#_Toc533616870)

[图3‑17不同配置方案空间消耗和时间消耗 44](#_Toc533616871)

[图3‑18两种优化技术相结合示例 45](#_Toc533616872)

[图3‑19矩阵乘代码 46](#_Toc533616873)

[图3‑20矩阵乘两层循环映射方案 47](#_Toc533616874)

[图3‑21脉动矩阵方式实现矩阵乘 48](#_Toc533616875)

[图4‑1程序运行在硬件平台上和仿真器平台上 52](#_Toc533616876)

[图4‑2仿真器实现的模块框架 54](#_Toc533616877)

[图4‑3阵列仿真器系统设计流程 55](#_Toc533616878)

[图4‑4 PE时钟划分图 59](#_Toc533616879)

[图4‑5系统验证平台 60](#_Toc533616880)

[图4‑6算法映射及验证流程 61](#_Toc533616881)

[图4‑7 HGRA仿真过程示意图 62](#_Toc533616882)

[图4‑8仿真器执行代码跳转示意图 63](#_Toc533616883)

[图5‑1高阶矩阵乘任务划分 66](#_Toc533616884)

[图5‑2 Sobel卷积因子 66](#_Toc533616885)

[图5‑3 Sobel算法的数据流图 67](#_Toc533616886)

[图5‑4 Sobel算法的控制流图 68](#_Toc533616887)

[图5‑5 9元素排序取中值示意图 69](#_Toc533616888)

[图5‑6 最大公约数算法实现代码 70](#_Toc533616889)

[图5‑7最大公约数算法的CDFG图 70](#_Toc533616890)

[图5‑8不同规模矩阵在GPU和HGRA平台上性能数据 71](#_Toc533616891)

[图5‑9 Sobel边缘检测算法在GPU和HGRA上的仿真数据 72](#_Toc533616892)

表 录

[表1‑1可重构系统、通用计算系统、专用计算系统特性比较表 2](#_Toc533603515)

[表2‑1 粗粒度阵列(CGRA)和细粒度阵列(FPGA)特性对比 15](#_Toc533603516)

[表3‑1 ALU算子汇总表 31](#_Toc533603517)

[表3‑2 PE内部配置字功能详解 33](#_Toc533603518)

[表3‑3细粒度处理单元支持的算子 38](#_Toc533603519)

[表4‑1 PE类内部实现方法 56](#_Toc533603520)

[表4‑2 control类内部实现方法 57](#_Toc533603521)

[表4‑3中间层调度函数表 58](#_Toc533603522)

[表4‑4函数层次调用关系 59](#_Toc533603523)

[表5‑1图像中值滤波不同平台运行时钟周期数 72](#_Toc533603524)

# 第一章 绪 论

## 课题研究背景

集成电路行业诞生以来的六十多年里，CPU、GPU、FPGA等计算平台在各个领域发挥了及其重要的作用，一步一步的引领者人类进入到了信息化的社会，但是最近几年这些计算平台难以提供各种场景所需要的计算能力。从外部环境看，随着人工智能时代的到来，为了适应各种复杂的场景，很多机器学习算法、神经网络算法、深度学习算法开始出现，这些算法的出现要求计算平台能够提供较大的算力，要求计算平台能够提供较大的灵活性，能够根据计算需求快速变换计算架构。

从内部环境看，各个计算平台都是在特定的历史条件下产生的，具有一定的局限性，CPU的出现是为了实现复杂的工业控制，所以适合于控制密集型的算法和应用，其提供的算力是有限的[1] ；GPU的出现是为了解决CPU在图形处理方面能力不足这一问题，其主要目的是协助CPU加速图形图像处理[2] ；FPGA的出现是为了实现硬件可编程，是可重构计算平台的一种，其提供了灵活的可配置性，提供了丰富的计算资源，在过去几十年里被芯片设计人员广泛选择用来作为设计和验证平台[3] 。这些计算平台都是基于冯·诺伊曼思想而设计的结构[4] ，这些结构的特点是适用范围广，但是计算能力有限。一种计算平台的计算能力和平台的架构设计有着莫大的关系，所以，一种新型的可重构计算架构的研究才显得尤为重要。

在这些内外环境的条件下，如何在计算能力和灵活性之间做一个权衡是一个比较难解决的问题，可重构处理器提供了一种更好的选择[5] ，可重构处理器充分利用硬件的空间资源，实现了在特定资源限制下计算能力的极大提升；同时可重构处理器根据应用的不同，自由灵活的配置电路，以较小的时间代价实现了不同算法功能的实现。

传统的通用处理器架构的实现是通过一套固定的硬件资源在时间维度上复用实现对一个任务的处理，其通过将代码编译成时间上连续的指令序列，再通过指令执行期间去调度数据，从而完成数据的流动，实现相应功能，其优势体现在较高的灵活性；专用处理器架构则是通过特定的硬件结构如乘法器、加法器等，将软件算法直接映射成硬件电路，通过利用空间维度上的复用实现计算任务，原始数据顺序地流过相应的处理单元，数据被处理，从而完成计算，这种结构充分发掘了算法中的并行性，具有较高的计算效率，但是灵活性低，研发周期长。而可重构计算结合了通用计算的灵活性和专用计算的高效性。通过将应用编译成配置信息，配置信息可以固化硬件电路，配置信息的改变实现了可重构系统的灵活性和高效性。

表1‑1可重构系统、通用计算系统、专用计算系统特性比较表

|  |  |  |  |
| --- | --- | --- | --- |
|  | 通用处理器 | ASIC处理器 | 可重构处理器 |
| 速度 | 慢 | 快 | 快 |
| 开发难度 | 易，周期短 | 难，周期长 | 易，两者之间 |
| 资源消耗 | 少 | 多 | 中等 |
| 并行性 | 指令级并行 | 空间并行 | 空间并行 |
| 可升级性 | 有 | 没有 | 有 |

## 研究现状

在20世纪60年代可重构的概念就被Gerald Estrin[7] 提出，但是那时候对通用处理器的研究正如火如荼，很少有研究学者会把目光转移到可重构计算上来；另一方面，局限于那个时候制造工艺不是很先进，所以可重构计算并没有进入到很多研究学者的研究范围。但是随着计算需求的不断变化和制造工艺的改进，可重构计算出现了以下几种典型的研究成果。

1. FPGA

随着硬件电路中可编程开关的实现，使得某些电路的功能可以被改变，这是可重构系统发展起来的先决条件。通过在芯片内部集成一些存储单元，在其中记录一些信息，这些信息被用于改变电路的某些结构。在这些信息被固化前，芯片电路的功能是不确定的，通过将存储器内容固化，这些芯片就具有了特定的功能，这就是早期的可重构的雏形。而这些可编程的存储器主要包括：可编程只读存储器(PROM-Programmable Read-Only Memory)、电可擦除只读存储器(E2PROM-Eletrically Erasable Programmable Read-Only Memory)、紫外线可擦除只读存储器(EPROM-Erasable Programmable Rean-Only Memory)。再后来随着技术的进步，出现了可编程逻辑器件(PLD-Programmable Logic Device)，通过可编程的器件可以实现大部分的数字逻辑。在之后出现了复杂可编程逻辑器件(CPLD)和现场可编程门阵列(FPGA),这种可编程逻辑器件可以实现较大规模的电路。如图1‑1所示，是CPLD中的与或门阵列，图中是由两部分组成的，与阵列和或阵列，图中·和×处的横竖两条线是连接的，·表示连接是固定的，不能编程，而×表示连接是可以改变的，通过可编程手段被改变，所以通过改变可编程的连接点，实现需要的逻辑，图中实现的逻辑如下：

 (1-1)

 (1-2)



图1‑1 CPLD中与或门阵列

Fig. ‑ AND gate and OR gate array in CPLD

如图1‑2所示，FPGA内部可重构单元采用的是查找表（Look Up Table）结构,查找表结构其本质上是一个可擦除的RAM，图中左侧是独立的16比特RAM,其中存储不同的数据，输入A,B,C,D组成4位的选择信号，在四输入确定的情况下数据选择器会选择某一个具体的RAM单元，其中存储的数值就是在输入为ABCD情况下的输出F，4位的输入其实就相当于RAM存储单元的地址，通过A,B,C,D来对RAM单元进行寻址，而被寻址的内容是可被更改的，这是实现可重构的重要原因。

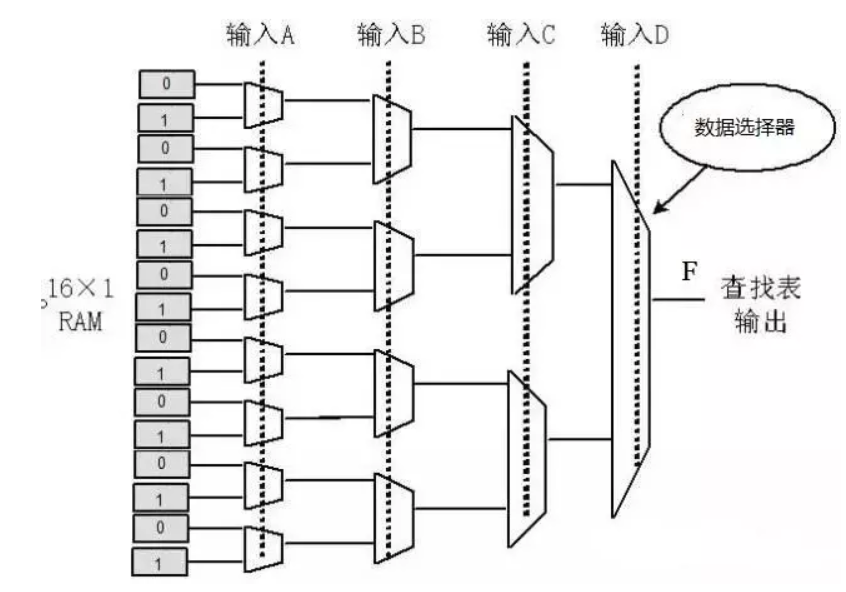


图1‑2 FPGA中查找表结构

Fig. ‑ LUT structure in FPGA

1. Grap[8]

Grap是美国加州大学伯克利分校研究出的一种新型计算架构，其是由一个MIPS处理器和一个细粒度可重构阵列FPGA组成的异构系统。Grap的核心研究目的就是将细粒度的FPGA嵌入到通用处理器单元中，研究这种结构的软硬件环境，探讨这种结构在加速计算方面的可能性，据文献[8] 所述，该混合结构可以使得应用的运算速度被加速到2到24倍，计算效率极大提升。

如图1‑3所示，Grap架构设计者们设计了一套完整的系统，在这个结构上实现了结构化程序运行、基于该结构和环境的库、任务切换、虚拟存储器和多用户等功能。图中标准处理器执行的指令是基于MIPS的扩展指令，增强了标准处理器中的控制能力，使其负责程序的控制和对可重构阵列配置信息的加载。可重构阵列和主核共享数据缓存，同时还能和MIPS处理器和存储器直接进行数据交互，保证了数据存取的高效性。可重构阵列是由称作block块的实体组成，如图1‑4 Grap架构组织图所示，可重构阵列中每一行的第一个block是一个控制块，其他剩余的是逻辑块，阵列的列数固定位24列，而行数是不固定的，可以根据具体的具体的应用设置成不同值以提高系统的兼容性，行数最大值可以设置成32。

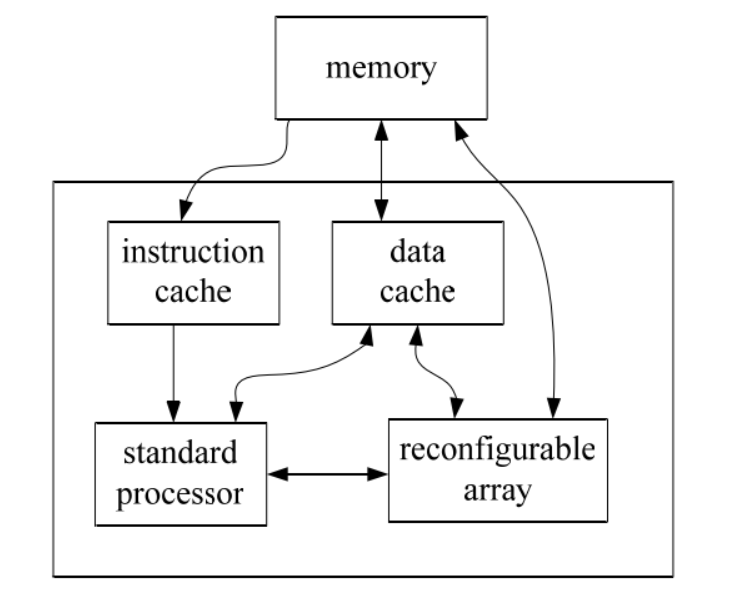


图1‑3 Grap结构框图

Fig. ‑ structure chart of Grap

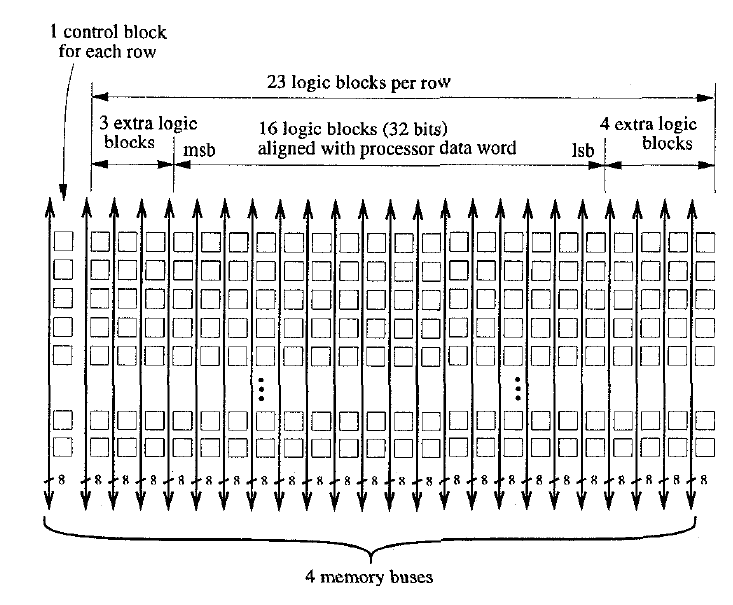


图1‑4 Grap架构组织图

Fig. ‑ architectural organization of Grap

1. Morphsys[9]

Morphsys是美国加州大学的可重构计算研究项目。其系统结构图如图1‑5所示，是由一个RSIC处理器、一个可重构计算阵列、指令数据缓冲器、DMA控制器和高带宽的数据接口组成。其中可重构阵列是由若干可重构单元(RC-Reconfigurable Cell)组成，这些可重构单元通过配置存储器实现功能的重构和互连的重构，该系统中实现的可重构阵列是粗粒度的可重构阵列，其数据通路的位宽是16位。

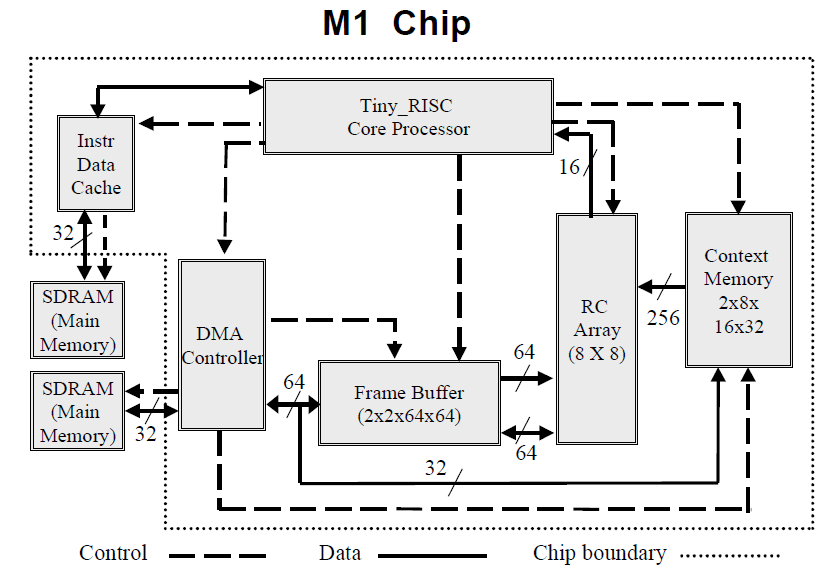


图1‑5 Morphsys结构框图

Fig. ‑ structure diagram of Morphsys

该架构系统主要面向视频压缩、图形和图像处理、数据加密和DSP转换等应用领域，和上面介绍的Grap系统一样，其主要的控制任务是通过RISC处理器来执行。可重构阵列是8×8的RC网络构成，支持多配置文件。如图1‑6所示，每一个可重构单元是由配置寄存器、两个多选一的MUX、算术运算器ALU、乘法器MULT、一个移位器和4个寄存器组成的寄存器堆组成。8×8的矩阵被划分成四个部分，通过二维网络实现连接。Morphsys结构是配置流控制的一种计算机架构，通过运行期间动态切换可重构阵列之间的互连关系从而实现动态的配置切换。

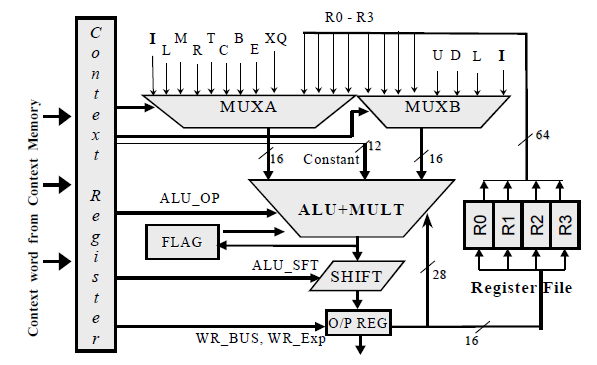


图1‑6 RC内部细节图

Fig. ‑ internal graph of RC

1. XPP[10]

XPP(eXtreme Processing Platform)是PACT公司提出来的可重构计算平台，其可重构阵列是由粗粒度单元组成，动态可配置的，即其可配置是在程序运行期间实现的，对于可重构阵列而言，在时间维度上，阵列接受的是时间上连续的配置信息流而不是指令，所以这种结构从本质上来讲还是控制流驱动的计算系统。

如图1‑7所示，XPP架构主要是由以下几个主要部件构成：

1. 可重构处理阵列簇(PAC-Processing Array Cluster)，其是由若干个可重构处理阵列单元(PAE-Processing Array Element)和配置管理器(CM-Configuration Manager)组合而成；
2. 配置管理器和层次化的配置管理器树;
3. 内部通信网络;
4. 若干I/O模块;

XPP结构支持数据流工作方式，但是整个结构是配置流驱动的，多个阵列簇支持多任务自动切换。其工作流程是：SCM-Supervising Configuration Manager负责和外部存储器交换配置信息，在获得配置信息之后将相应的配置信息分配给处于不同簇的配置管理器CM，配置管理器通过配置总线向可重构处理单元写入配置信息，处理单元根据获得的配置信息执行相应的操作，运算结果分布式的存储在各个可重构处理单元内部或者FIFO中，准备下一次的使用。由于是动态配置的，配置信息在系统内部传递的速度反映出该系统的吞吐率，所以在XPP系统中可以看到被特殊设计的配置高速总线。

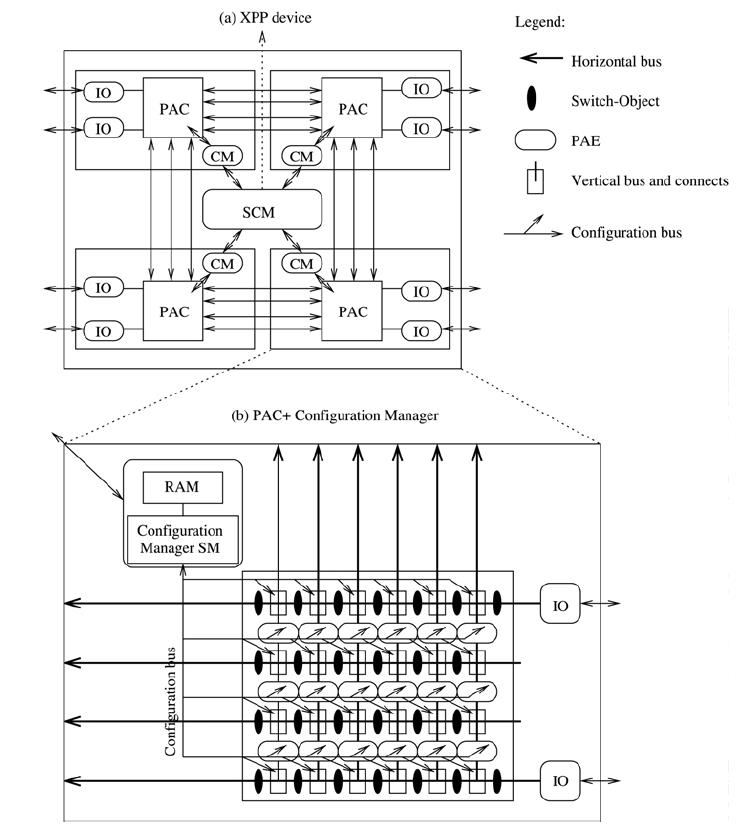


图1‑7 XPP架构图

Fig. ‑ architectural organization of XPP

1. WAVECHCHE**错误!未找到引用源。**

Wavecache是华盛顿大学的Steven Swanson及其团队设计开发的一款处理器，是基于Wavescalar指令集研究开发的，处理器的设计思想是alu-in-cache，即在处理单元内部设计较多的输入缓冲器和输出缓冲器，处理单元内部内嵌一个ALU运算单元，从整个PE外部来看，这是一种computation-in-memory结构。

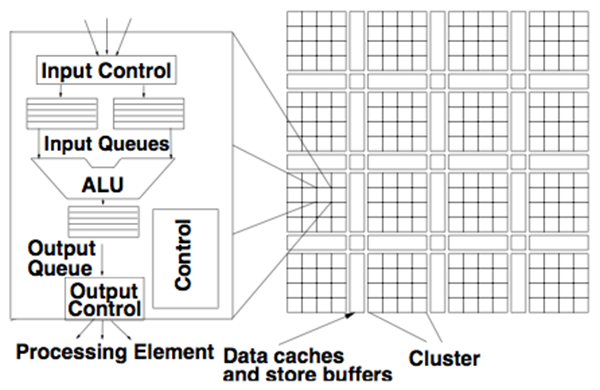


图1‑8 WAVECACHE结构图

Fig. ‑ structure diagram of WAVECACHE

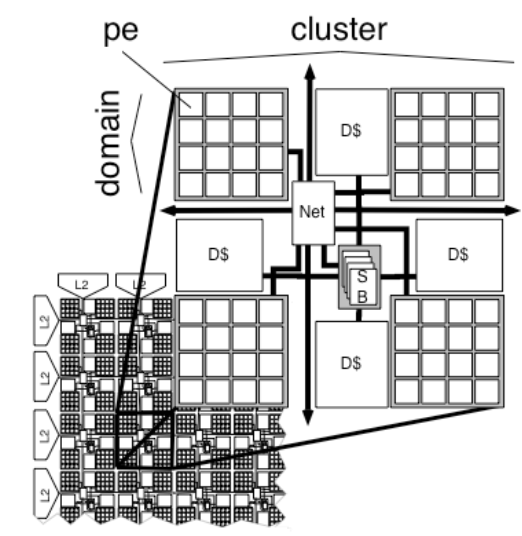


图1‑9 WAVECACHE框架图

Fig. ‑ diagram of WAVECACHE

如图1‑8，图1‑9所示，该处理器是由将近1K个处理单元组成，这些处理单元被划分到16个簇中，每一个簇拥有16个处理单元。在一个簇内部各个处理单元之间通过L1-cache连接，外部再通L2-cache和外部存储器连接。处理单元内部分布着大量的缓存器，输入缓冲器要实现对输入数据进行协调处理，再送入到ALU进行运算，最后送到输出缓冲器队列，通过输出逻辑控制输出。

## 主要研究思路

从文献[8] [9] [10] [11] 中可以看出，众多研究学者提出的可重构系统架构，在特定领域发挥着重要的作用。但是分析下来可以发现这些结构存在着以下两个方面的劣势，如下：

首先，可重构阵列被广泛用于计算密集型的场景，用于实现图像图形处理这一类算法的加速，这一类应用具有计算密集、数据规整的特点。所以，可重构处理器的设计往往注重计算能力而忽略了控制能力。从以上介绍的几种可重构系统可知，大部分的可重构阵列需要一个通用处理器作为主控，通过CPU对任务进行控制，调度可重构阵列进行计算，这种结构必然需要可重构阵列和主控之间频繁的数据交互，二者之间的连接网络将成为设计中的性能瓶颈。所以本文针对这种痛点，研究设计将粗粒度单元和细粒度单元进行混合，组成的阵列结构以解决控制能力薄弱的问题，减小主控和阵列之间的交互代价，其中细粒度处理单元实现对任务的控制，粗粒度处理单元实现对任务的计算，这样做的好处是主核处理器在触发加速任务时，只需要将加速任务的上下文信息一次性提交给可重构阵列，可重构阵列通过细粒度单元和控制单元实现对任务的划分和调度，从而给主核处理器减压，提高计算效率。

其次，关于可重构阵列中处理单元的执行方式，以上提到的几种结构都基于控制驱动，比如Grap、Morphsys的指令流驱动，XPP的配置流驱动。控制流驱动实现的处理器其执行方式依赖于指令/配置的具体内容，指令中携带需要执行的操作码和操作数，在串行执行的处理机系统中，操作数可能早就准备就绪，但是由于指令没有进入到执行单元，操作码未被激活，造成操作数据闲置，另一方面，程序运行过程中需要产生大量的配置，对于可重构计算的系统来说，任意时刻的配置动作会导致整个电路执行时间上升，程序运行的时间大部分是消耗在配置切换上，所以针对这些缺点，本文选择数据流驱动的执行方式，将配置信息中的操作码和操作数分开，设计后的配置中只包含操作码和互连信息，不包含操作数，操作数信息通过互连网路自动实现流转，结合数据流执行机制可以大大减少数据闲置时间，同时这种方式可减少配置量，减少配置时间，加快计算。

## 论文组织结构

本论文提出了一种基于数据流原理的混合粒度可重构阵列处理器的架构，并面向具体的应用探讨了整个阵列结构的实现细节，同时基于该阵列结构，设计并实现了该硬件结构的软件仿真器，用于评估本论文所提出的阵列结构的功能和性能。本文设计的阵列结构是一种全新的计算架构，将数据流计算技术和空间计算技术相结合，实现了一款处理器芯片既能提供较大的计算能力，又能提供一定的灵活性，其软件可配置的特性使得其在不同的应用场景之间进行切换，减少了芯片设计和制造过程中的时间成本。

本论文的章节安排如下：

第一章介绍了本论文研究课题所面向的研究背景，确定了做混合粒度可重构计算阵列的必要性和重要性。

第二章介绍了混合粒度可重构阵列结构设计过程中涉及到的主要技术，包括数据流技术和可重构技术，为本研究提供了强有力的理论支撑，体现了研究的创新性。

第三章详细介绍了混合粒度可重构计算阵列的设计细节，包括粗粒度处理单元和细粒度处理单元的结构设计，执行机制设计，数据通路设计等，介绍了该阵列结构支持的两种应用映射优化技术以及保证阵列正确运行的反馈机制；最后，以矩阵乘应用为例，介绍了应用分析和映射的基本方法。

第四章介绍了基于该阵列架构的仿真器的实现细节。介绍了仿真器的原理和该仿真器的基本框架以及每一个的模块的具体实现，最后介绍了系统级仿真平台的搭建和验证测试流程。

第五章通过几种算法的映射，分别在通用GPU平台和混合粒度阵列平台上进行仿真，获得仿真数据，并对比分析仿真数据得出结果。

第六章对论文进行了总结，归纳本文的主要成果并分析其中的不足，并对该研究领域进行了展望。

# 第二章 可重构技术及数据流驱动技术

## 可重构技术

可重构计算系统是指能够通过软件字段来改变电路功能，并通过外围的辅助软硬件来运行一定的计算任务的系统[16] ，其最主要的优点就是灵活的可配置性。它摒弃了ASIC电路设计不灵活的缺点，同时又抛弃了通用计算系统计算效率低的缺点，结合了两者的长处，既能实现计算效率高还能灵活可配置，正成为研究的热点。如图2‑1，是可重构计算在时间和空间利用率上和其他两种计算平台的比较。



图2‑1三种计算平台在灵活性和计算效率上的对比[17]

Fig. ‑ Comparison of three different computing platforms

### 可重构计算执行机制

一般来讲，可重构计算系统包含可重构计算阵列、任务控制器和存储器，如图2‑2。可重构计算阵列是可重构系统的核心，它负责数据处理和数据的传递，是可配置性的实现主体；任务控制器负责对系统进行控制，负责对运行在计算阵列上的任务进行调度，实现数据的输入和输出，它时刻监控着计算阵列和整个系统的运行情况，是整个系统能成功运转的核心；存储器部分负责阵列的数据存取，由于可重构计算阵列和主处理器构成一个异构计算平台，所以可重构计算阵列需要有自己的本地存储器，这一部分主要实现阵列数据的存取以及和主处理器进行数据交换。



图2‑2可重构计算系统的组成[18]

Fig. ‑ component of reconfigurable computing system

可重构处理器系统是充分利用了时间特性和空间特性的一种计算系统，如果时间特性占据主要地位时，就是通用计算平台；如果空间特性占据主要地位时，就是专用计算平台。下面通过举一个例子来说明可重构计算在时空利用程度上与通用计算系统和专用计算系统的不同。假设我们要完成一个多项式的操作，如下：

 (2-1)

第一种实现方式是通过专用计算系统实现的，其通过将算法中所用出现的算子全部在空间上展开，多项式中有三个乘法操作，两个加法操作，所以通过设计三个乘法器两个加法器，让数据进入到相应的处理单元中进行处理，得出计算结果，几个处理单元同时进行，在几个时钟周期就可以计算出需要的结果，这种计算架构根据算法本身进行定制，具有计算效率高的特点。

第二种实现方式是通过通用计算系统实现的，通过将多项式的内容编译成一条条时间维度上的指令，指令中包含需要执行的操作码和操作数存储的寄存器编号，按照时间顺序进入到计算单元中执行，并将计算结果存储到相应的寄存器中，准备好给下一条指令使用。通过这种时间维度上的推移和空间维度上的复用，完成了一个计算任务。如图2‑3所示，这种计算结构只需要包含一些寄存器和算术计算单元，其具体的执行流程如下：所给定的表达式被编译器编译成5条指令，这些指令依次会出现在指令寄存器中，程序计数器会在每个时钟周期从指令寄存器中取出一条指令送到译码单元，译码之后算数逻辑单元会从指令指定的寄存器中取出操作数并执行操作码规定的动作，当该程序对应的所有的指令被执行完毕，计算任务执行完成。

这种计算模式的优势是所需要的硬件资源是固定的，适用范围较广，但是由于指令是在时间维度上展开的，其结构面临着计算效率低的困难。具体表现在：首先，这种体系结构依赖于指令集，指令集限制了具体硬件电路的适用范围，也就是说硬件电路支持的操作种类完全依赖于指令集中规定的指令数，由于指令个数的限制，在遇到复杂的代码的时需要将复杂的功能拆分成几条简单的指令来实现，降低了计算效率；其次，通用处理器的设计初衷就是为了实现通用计算，必然要实现指令集中规定的指令操作，面临着资源浪费的可能，假设设计了一个64位的通用平台处理器，但是只用于处理16位的程序，那么处理器中很多的系统资源都是浪费的。所以，这种时间域上的计算平台满足了通用性却也牺牲了计算效率。



图2‑3三种计算系统在某一应用上的时空利用图[18]

Fig. ‑ The realization of one application on three different platforms

第三种方式是通过可重构计算系统来实现，可重构计算系统结合了以上两种计算系统的优点，充分利用了时间域特性和空间域特性，针对某一特定的任务，可重构系统可以充分利用其空间资源将任务进行空间展开，体现出对空间特性的利用，而针对一系列的任务，可重构系统充分利用时间特性，实现任务级的流水，这两者的结合使得可重构计算既可以具备灵活性又可以具备较高的计算效率。具体到上面的例子，针对可重构计算系统它的执行流程如下：首先，整个硬件电路是由一系列的可配置计算单元构成的，多项式被分解成3个乘法和2个加法，所以会被生成两个加法配置包和3个乘法配置包，将配置信息固化到可配置的5个处理单元上，处理单元组成的阵列电路就形成了一块专用电路，注入原始数据并激活就可以执行相应的操作，最终得出相应的计算结果。可重构计算允许改变电路计算功能来适应不同的算法和应用，是一种动态的体系结构，这种动态变化保证了灵活性，在阵列被固化之后形成的专用硬件电路保证了计算效率。所以，在计算密集型需求场景中，可重构计算是一种较为合适的解决方案。

### 可重构计算其他关键技术

以上介绍了可重构计算的执行原理，分析了可重构相比较于其他计算平台在执行机制上的不同，并总结了其优势所在。本小节介绍可重构计算系统中其他关键技术。

**粒度**：可重构计算最早的产物就是现场可编程门阵列（FPGA）,FPGA的配置单元的最小位宽是1bit，所以FPGA是粒度为1的可重构计算平台，也就是细粒度可重构阵列。通常来讲，我们把配置单元的最小位宽小于等于4bit的可重构阵列称为细粒度可重构阵列，大于4bit的称为粗粒度的可重构阵列。

一般来讲，在进行可重构系统设计时粒度的选择需要考虑很多方面的因素，粗粒度阵列和细粒度阵列在阵列结构、组成单元、配置方式、执行机制等方面有着很大的区别。细粒度阵列是由查找表、逻辑门等构成，执行机制是进行比特级的操作；而在粗粒度阵列中，处理的是字级的数据，内部也是需要包含完整的功能单元，需要有集成众多算子的算术逻辑单元和特殊设计的互连网络。另一方面，选择不同的粒度会导致整个可重构系统具有不同的配置信息量，具有不同的配置时间和配置功耗，最主要的影响是两种不同的粒度的计算效率是不同的，所以在进行可重构系统设计时要充分考虑到这些因素。表2‑1所示的是粗粒度可重构阵列(CGRA)和细粒度可重构阵列(FPGA)的特性对比。

**静态和动态重构**：可重构系统中可重构的实现来自于两方面，首先是计算单元的可重构，是指通过改变某一个计算单元要实现的功能，另一方面来自于计算单元之间的连接关系可重构，根据具体的任务来改变处理单元之间的连接形式，从而达到互连关系的可重构，所以重构在时间维度上有两种表现形式，重构的系统在某一段时间内其结构不发生变化，每一条配置指定的功能和连接关系在任务时间窗口内不改变，这是静态的可重构系统。早期的FPGA是这种系统的典型，优点是配置切换的次数少，功耗低，但是缺点是必须等到一个任务已经执行完成并保存现场之后才能开启配置的切换。

表2‑1 粗粒度阵列(CGRA)和细粒度阵列(FPGA)特性对比[19]

|  |  |  |
| --- | --- | --- |
| 可重构处理器 | FPGA | CGRA |
| 处理单元 | LUT(bit) | ALU(字级) |
| 互连结构 | 复杂 | 简单 |
| 功耗 | 高 | 低 |
| 配置信息 | 大 | 小 |
| 配置时间 | 长 | 短 |
| 灵活性 | 很高 | 较高 |

另一种是重构系统在某一段时间内会根据配置信息的具体内容来实现计算单元和互连的动态切换，每一条配置都会激活一次配置切换，程序的内容被编译成配置序列，每一条配置在任务的时间窗口内其执行的功能和连接关系都随时可变，这是动态的可重构系统，其优点是配置的切换是在程序的执行过程中进行的，具有更大的灵活性，缺点是需要的配置信息比较多，整个可重构系统的功耗比较大。

**互连网络：**可重构系统的实现不仅依赖于每一个计算单元的功能可以根据需要变化，同时依赖于可以改变的互连网络。如图2‑4所示，互连网络可配置的原理是通过多选一的选通电路实现，通过选择信号将两个处理单元连接在一起实现数据通路，其中的选择信号就是配置信息中的某一位或几位。

可重构系统的互连网络是复杂的，通常来说有两种方案可供选择。第一种是全连接的方式，即处于阵列中的每一个计算单元都可以和出自己以外的任何一个计算单元连接，如图2‑5所示，假设阵列中有9个处理单元，处理单元A要实现全连接的功能，针对单元A的某一个输入端口就需要设计一个8选1的MUX电路，这种情况下，随着阵列硬件资源的增多，这种选择电路的规模是成倍增长的，很显然对于规模较大的可重构系统的实现，这种方案不是最好的选择。第二种方案是处于可重构阵列中的某一个计算单元只可以和自己周边的上下左右相邻的计算单元连接，如图2‑6，单元A的输入端口只能和单元B,C,D,E连接，对于单元A的某一个端口通过固定的4选1的MUX电路就可以实现选通，实现电路比较简单固定，易于实现。

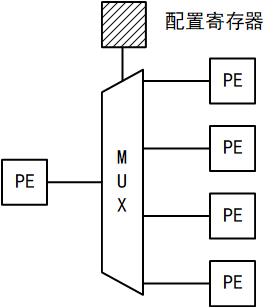


图2‑4互连网络可配置原理图

Fig. ‑ Schematic of connect network

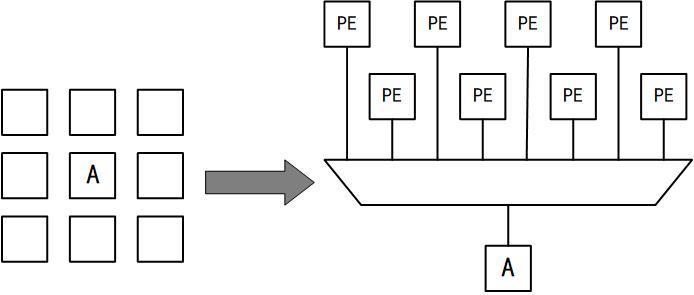


图2‑5互连网络全连接

Fig. ‑ Full connection of net

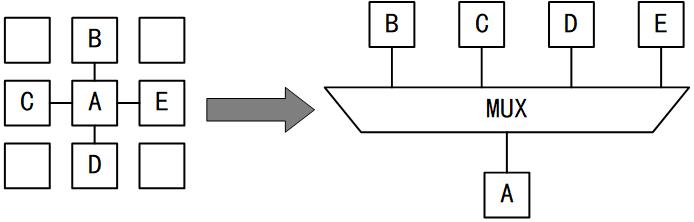


图2‑6互连网络局部连接

Fig. ‑ Local connection of net

比较这两种方案，各有自己的优缺点，第一种可以提供较大的灵活性，布局布线算法可以设计得较为简单，但是实现的代价是需要消耗较多的硬件资源，布线网络也变得很复杂。对于规模较小的可重构系统可以选取这种方案。第二种方案的优点就是互连方式较为固定，对于可重构系统的布线器实现较为简单，同时硬件电路实现复杂度较低，但是灵活性不够，需要强大的布局算法的支撑。

**存储系统：** 现代通用处理器平台的存储系统是分层次的，从速度快到慢分类看，分别是基于SRAM技术的cache、基于DRAM技术的内存以及存储器，这种层次的存储系统是在处理器速度比较快而存储器速度比较慢的情况下提出来的，使得处理器系统和存储器系统能更协调的工作。在可重构阵列中，存储系统越来越成为其计算性能的瓶颈。存储系统设计中有两个主要问题，首先，可重构处理器和通用处理器组合的系统，需要协同工作完成某一个具体的任务，从外部看这是一个多核系统，需要考虑存储系统的一致性的问题。现在通用的做法是，需要DMA在一个任务开始的时候将阵列运算所需要的数据全部搬运到可重构阵列的存储器上，在整个阵列运行期间阵列的运算数据都是和自己的存储系统进行交互，只有在任务结束之后才会将结果数据再搬运到主处理器的存储系统上。

第二个问题是可重构计算属于空间计算范畴，其较大的并发性对存储器的压力是很大的，所以可重构处理器的存储系统要采用分层次、分布式的存储系统。如图3‑2所示，根据作用范围的不同，需要有全局存储器、本地存储器和分布式的输入输出buffer，全局存储器在阵列运行的早期存储搬运过来的原始数据。本地存储器存储阵列运行期间产生临时数据，分布式的输入输出buffer是每一个处理单元内部独有的，其功能是用于缓存进入到单元的数据，并对数据进行匹配输出。

**协同编译器技术**：[20] 基于冯诺依曼结构的通用处理器从诞生以来经过了这么多年的研究，其在编译技术上已经相当成熟了。编译器是高级编程语言出现后的产物，能完全实现高级语言在不同平台上执行代码。可重构系统的编译技术目前还处于起步阶段，现阶段的可重构系统其任务划分、配置生成等都依赖于手工来实现。在由通用处理器和可重构处理器组成的异构系统上，编译技术需要被重新设计，因为两者使用不同的编译技术实现代码的执行，通用处理器上执行的代码被编译成一条条指令按序执行，可重构系统的代码被编译成一段配置信息通过可重构阵列中的计算单元和互连网络来实现，现阶段协同编译技术的实现没有被完美的支持。

针对异构计算平台来说，协同编译技术的实现显得尤为关键，这种编译技术需要实现从源程序中自动划分出主处理器执行的代码和可重构阵列处理器执行的代码，再分别调用两种计算平台的编译器实现代码的编译，同时还要有对整个代码任务进行调度的能力，根据硬件资源和当前任务执行情况动态的进行任务的切换。

## 数据流驱动技术

可重构系统可以选择动态可重构或者静态可重构，动态可重构系统在任务运行期间频繁切换配置来执行任务，具有配置信息量大、配置时间长和配置功耗大等缺点，但是灵活性较高；静态可重构系统在任务运行的时间窗口内配置不需要切换，具有配置量小、配置时间短和计算效率高等优点。而本文研究的可重构系统是面向特定领域的，需要较高的计算效率，所以选择对静态可重构系统进行研究。

静态可重构系统在配置信息被固化到处理单元中后，其具体的硬件电路基本固定，在任务运行期间每一个处理单元会执行相同的操作，在这种结构下，由于操作码和操作数分离，数据需要主动驱动阵列运行，让每一批次的数据流经阵列单元并被处理，实现操作数和操作码的融合，从而完成计算，这是数据流驱动的执行机制。静态可重构系统需要数据流驱动阵列执行，保证其正确性，保证最大效率。下面介绍数据流驱动的主要思想和基本原理。

### 数据流计算机基本原理

电子计算机诞生以来，计算机的体系结构都是基于冯·诺伊曼结构及其变种来实现的，这种结构是基于控制流实现的，程序被编译成一条条的指令存储到指令寄存器中去，计算机通过执行指令来实现代码中指定的功能。基于控制流实现的计算机是一种通用计算平台，其主要的特点就是通用性强，可以适用于各种应用场景，但是控制流计算机在并行计算场景上有着天生的困难[12] [13] 。

数据流驱动计算机能较好的支持并行计算，它突破了以指令为基础的数据流动，它以数据来驱动相应操作的执行，当相应的操作所需要的数据都准备就绪之后就可以激活该操作进行相应的动作，在数据流里面成为点火(fire)行为，其操作结束之后产生的数据可以作为下一步的操作需要的源操作数。由于硬件结构中没有程序计数器，点火行为可以在空间硬件资源上同时的进行，所以这种隐含的并行性使得数据流计算机很适合于并行计算[14] 。

### 数据流驱动执行机制

数据流处理机中的指令主要是由数据令牌和操作码构成，数据令牌是由两部分组成，分为数据和数据的目的地址，每一条数据令牌都包含这两条信息。操作码被分配到相应的硬件资源上并具有一个地址，当操作码点火操作需要的数据令牌到达该操作码指定的硬件资源地址时，该操作码指定的操作被启动，

如图2‑7所示，表示的是快速傅里叶变换中的蝶式变换，通过这张图可以知道数据流计算中，数据流图包含两种不同的节点，分别称为输入弧（link）和动作（actor）。由图可知，两个动作之间必有一输入弧，两输入弧之间必有一动作，如图所示，其中圆圈表示动作，小圆圈表示输入弧，每一动作表示进行一步运算，其结果由输入弧传送到下一动作。



图2‑7蝶式变换的数据流图

Fig. ‑ data flow graph of butterfly transformation

数据流计算机可以分为静态的数据流计算机和动态的数据流计算机。对于静态数据流处理机，在两个动作之间的输入弧上只能存在一个数据令牌，也就是说该条输入弧上的数据令牌是没有标号(tag)的，如图2‑7，任意时刻每一个动作节点只会接受一个来自于输入弧上的数据来激活自己的操作。由于数据令牌没有标记，所以如果在任意时刻任一条输入弧上同时出现了两条以上的数据令牌的话，节点对于传输到自己的数据令牌就没有能力区分它们中哪些是属于同一批的操作数，节点会任意选择其中某一个数据令牌中的数据进行操作，从而会出现错误。由于这种限制静态数据流处理机只能用于处理一般的循环，循环迭代之间不能进行流水，在计算性能上有较大的损失。

而动态数据流处理机让每一个数据令牌带上记号(tag)，这样可以在两个节点之间的输入弧上出现多个数据令牌，由于记号的存在，每一个动作节点可以匹配相同批次的数据进行点火动作。所以动态数据流处理机不需要像静态数据流处理机那样通过控制输入弧上的数据令牌个数来保证程序的正确执行。如图2‑8所示，这样每一个节点可以不间断的进行点火操作以达到最高的执行效率。所以，动态数据流处理机更适合于做循环代码的展开操作，可以充分提高程序中的并行性。



图2‑8动态数据流情况下的输入弧

Fig. ‑ input arc of dynamic data flow

## 混合粒度阵列的优势

针对可重构阵列的研究目前还存在着一下几点问题：

1. 可重构阵列的研究大部分都集中在对粗粒度阵列的研究，粗粒度阵列在计算上的能力是巨大的，但是控制能力薄弱是其主要的缺点，所以大部分的粗粒度阵列将对任务的控制交给主核去做，这样会导致主核和可重构阵列进行频繁的数据交互，交互通路会成为性能瓶颈。

本文研究将计算任务整体交付给可重构阵列进行执行，提出混合粒度可重构阵列结构，粗粒度阵列实现计算功能，细粒度阵列实现控制功能，目的是为了减少主核参与阵列执行的时间，主核只在任务开始和任务结束时参与运算数据的存和取，其他时间阵列可以自主的进行任务的调度。

1. 大部分粗粒度可重构阵列为了实现更灵活的支持更多应用，大部分采用的是动态可重构系统，电路在运行期间可随时改变功能和连接方式，这将导致配置信息量增多，任务运行的大部分时间花费在配置切换上，影响计算效率。

本文研究设计静态可重构阵列，可重构阵列被配置成静态的电路，通过数据流驱动阵列执行任务，极大的减少了配置量，任务运行期间没有配置的切换，较少配置时间，同时数据流技术减少了数据等待，最大化数据的利用效率。

鉴于以上的几点问题，本文通过将以上介绍的两种技术结合在一起，设计了一种全新的可重构计算架构，相比较于其他的可重构计算架构这种计算机架构具有以下几点优势：

首先，本文研究的混合粒度阵列主要是由粗粒度处理单元CPE(Coarse Processing Element）、细粒度处理单元FPE（Fine Processing Element）构成，如图3‑2所示。阵列的基本工作方式是针对不同的算法和应用，提取出算法中的控制数据流图CDFG(Control-Data Flow Graph）,结合阵列上处理单元算子资源分布，生成配置信息。这些配置信息分布式的存储在每一个处理单元的配置寄存器中，决定每一个处理单元的功能、处理单元之间的数据交互方式以及处理单元内部的数据通路的形式。通过改变每一个计算单元内部配置寄存器的内容，重新配置处理单元及其之间的互连方式，从而实现了可重构的能力。其主要的特点就是程序中的控制流图（CFG）和数据流图（DFG）的配置信息分别是由阵列上的细粒度处理单元和粗粒度处理单元来实现的，整个阵列区域可以实现特定程序的完整执行而不需要主核处理器的参与，实现计算效率的极大提升。

其次，可重构阵列结构采用数据流驱动的执行方式，数据流驱动的执行方式相比较于控制流驱动的执行方式在理论上具有高度的并行性，由于数据流计算机所关注的点是程序运行过程中的数据，所以只要是数据准备好了，多个计算点可以同时开始计算，不局限于时间上的依赖性，只要空间资源足够多，数据流计算可以达到理论并行度。数据流驱动的计算机其实本质上是一个小型的分布式计算的系统。

最后，充分利用数据流技术和可重构技术相结合延伸出的新特点[21] ，通过支持循环级流水和代码空间展开，充分挖掘该结构的优势，大大提高了该结构的计算吞吐率。循环级流水和空间展开技术是相辅相成的，通过将循环块的空间展开，一层的循环体结构能够全部分配到空间的硬件资源上同时并行的计算，没有数据依赖关系，循环体被空间展开之后，循环层之间的流水就变得更加简单和明显。

## 本章小结

本章介绍了介绍了可重构技术的起源、关键技术，总结各种可重构系统所采用的技术特点，体现各种不同结构的优缺点。同时介绍了数据流计算机的相关概念以及其执行机制，分析了其和控制驱动型的计算机相比较的优势所在，同时最后，分析了本文设计的结构在理论上的优势并介绍本结构的主要特点。

# 第三章 混合粒度可重构阵列结构设计

## 系统整体设计

### 设计思想

一般而言，可重构阵列系统的设计可以按照如图3‑1可重构系统的设计流程图所示的流程图来进行，从图中可以看出主要分为四个部分，应用分析、软硬件设计、软硬件接口设计以及系统验证和测试，本文主要进行硬件结构设计和对结构的仿真实现。

应用需求分析是进行系统设计的导向因素，可重构系统设计方法，设计结果都是要考虑到设计需求，通过从应用出发，提取出应用中的核心算法，再进行系统建模和架构设计。现代的集成电路设计方法中，进行软硬件划分是至关重要的。在可重构系统中进行软硬件划分关系着整个系统的性能。根据上一步中的算法，确定算法中需要软件实现的部分和需要硬件实现的部分，要充分的分析和调研，因为这一步决定着硬件电路的实现复杂度和最终芯片的性能。软硬件接口是软件和硬件之间交互的一个通道，硬件的结构比较固定，资源有限，需要软件协助进行任务的控制，软件的行为能被硬件理解并被执行时软硬件接口设计的重点。最后，系统的验证和测试是保证整个可重构系统能正确工作的手段，是系统设计中不可或缺的步骤。

图3‑1所示的电路设计流程在通用计算领域已经是成熟，尤其是面向CPU的高级编译技术，实现了整个设计过程的自动化，但是可重构计算的设计自动化研究起步比较晚，针对可重构计算的编译技术仍处于起步阶段。所以本论文的着重点是在进行阵列的架构设计，通过手动进行软硬件任务的划分，分析算法，手动的进行配置文件生成，将配置文件导入到硬件仿真器中进行模拟测试，查看结果并优化设计。



图3‑1可重构系统的设计流程图[19]

Fig. ‑ Design flow chart of reconfigurable system

### 模块整体介绍

整个可重构阵列结构的设计是分模块的，如图3‑2所示，阵列的中间部分是粗粒度处理单元阵列CPE，左右两侧分布的是细粒度处理单元阵列FPE，阵列上、下分别是Load和Store单元，再加上任务控制器、配置控制器、全局存储器和本地存储器，构成一个完整的阵列结构。阵列中形成了两条流通路，分别是配置流通路和数据流通路，配置流是在一个任务开始前被写入到阵列中各个处理单元，进行阵列的配置和重构。数据流是流动在阵列可重构单元之间的原始数据，携带计算信息。



图3‑2混合粒度阵列架构整体图

Fig. ‑ The overview of HGRA

整个阵列的执行过程是这样的：当主核处理器执行到加速代码段时，主动生成跳转指令，将其发送给可重构阵列中的任务控制器，任务控制器解码指令后，发送控制信号给配置控制器，配置器把该任务的所有配置从全局的存储器中取出，按照处理单元ID分别存储到处理单元的配置寄存器中，各个计算单元的配置工作完成；同时，任务控制器不断检测配置信息完成情况，一旦检测到有配置信息配置完成，即发送执行信号给整个阵列，启动信号注入成功，相应配置包被激活，各个处理单元的功能和互连关系被确定,每个计算单元独立计算，基于数据流驱动的原则，FPE实现对循环的控制，CPE实现循环体的计算，LE/SE实现对数据的load和store操作，当循环结束的时候，阵列发送结束信号给任务控制器，任务控制器进行状态的切换和跳转，接着激活下一套配置包，直到所有的配置包信息都被执行完毕之后，任务控制器反馈任务结束信号给主核，并将任务运行结果搬运到主核处理器能读取到的存储空间中去，主核处理器接收到整个任务执行完成信号读取数据并进行后续计算，阵列进入空闲状态，整个任务执行完成。

从以上的整体结构框图中可以看出，本论文研究的可重构阵列主要是由粗粒度阵列和细粒度阵列组成，粗粒度阵列处于整个阵列的中间，数量较多，主要是执行以字级操作为主的粗粒度计算；而阵列的外部是细粒度的阵列片，主要是由细粒度的处理单元组成，数量较少，主要是对粗粒度的执行进行必要的控制操作，比如分支跳转操作等。粗粒度阵列和细粒度阵列相结合，具有较大的优势，以往的情况中，在处理稍微有些控制代码的算法时，需要主核处理器进行控制，将任务划分成各个执行块，阵列执行时要频繁和主核进行数据交互，但是混合粒度的阵列可以解决这个问题，在任务开始是，主核只要将任务执行需要的一些启动信号、参数和配置信息的存储地址传递给混合粒度的阵列后，就可以执行自己的代码而不需要再参与到阵列计算中去，不仅减少了两者相互交互的时间，同时不用频繁占用总线资源，为主核执行其他任务创造了条件。

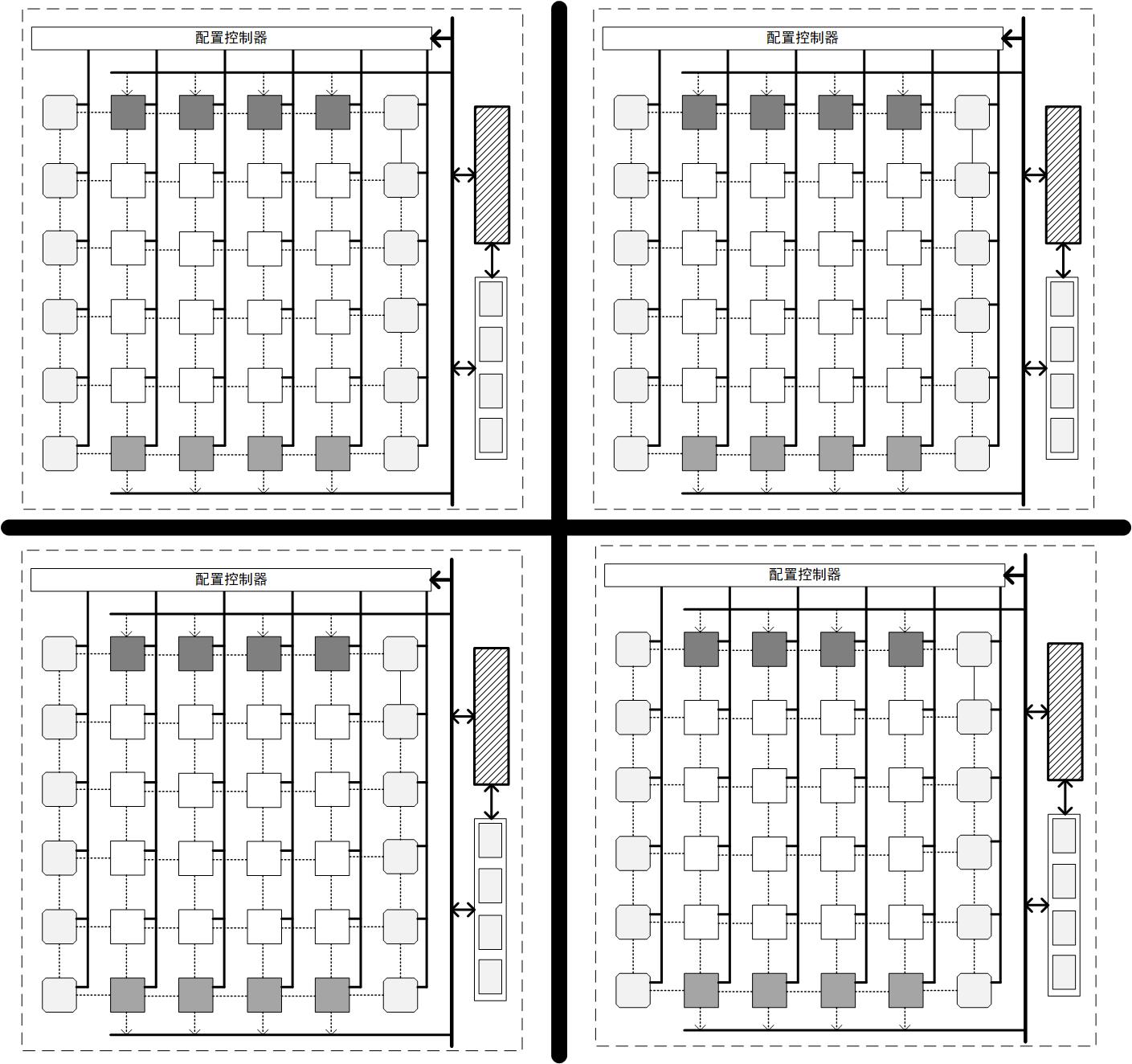


图3‑3可重构阵列系统图

Fig. ‑ System chart of reconfigurable array

图3‑2所示的是一块阵列的简图，图所示是本论文研究的整个系统示意图，其是由四个簇构成，每个簇内有8×8的粗粒度处理单元组成粗粒度阵列，7×2的细粒度处理单元组成的细粒度阵列，各个簇之间通过共享存储器进行数据交互。

## 粗粒度处理单元结构设计

粗粒度处理单元(CPE)是计算的实际载体，对CPE的结构设计关系着整个结构的性能。本文中讨论的粗粒度处理单元的数据通路是以32位为基的，主要是进行字级操作。

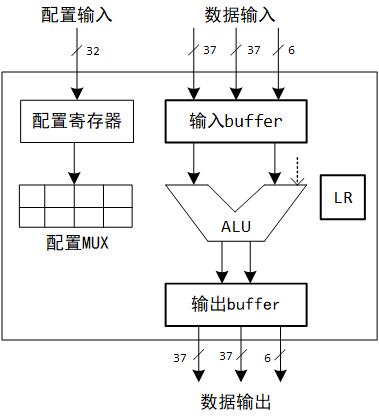


图3‑4粗粒度处理单元内部结构简图

Fig. ‑ Inside view of coarse-grained processing element

如图29所示，粗粒度处理单元的端口构成是两个37bit的输入端口，一个6bit的输入端口，两个37bit的输出端口，一个6bit的输出端口和一个32bit的配置输入端口，其中37bit和6bit的端口位宽安排如图 30，其中v表示有效标志位(valid)，占据1bit，表示某一个数据再阵列流动中是否有效；t表示数据标志位(tag)，占据4bit，表示输入到阵列处理单元中的某一个数据的批次，本论文研究的可重构阵列支持动态的数据流驱动，在两个计算节点之间可以有多个批次的数据存在，为了区分不同批次的数据需要为每一个进入处理单元的数据打上一个标签，用来区分不同的数据。D表示粗粒度真实数据,占据32bit,是进入到阵列中参与计算的原始数据；b表是细粒度真实数据，占据1bit，参与阵列计算的数据，也可以作为阵列的激活计算信号。

粗粒度处理单元的内部是由输入输出buffer、可配置功能的ALU、本地寄存器LR(local reg)、配置寄存器和可配置MUX构成。



图3‑5输入输出端口数据线位宽编码

Fig. ‑ Coding scheme of input and output data

### 输入输出buffer

可重构计算是属于空间计算的概念，在空间计算中数据分布式的被传输到具体的计算单元上，这正是空间计算在计算并行性上的优势，在进一步提高可重构系统的计算效率时，数据的输入和输出就成了制约系统计算效率提高的瓶颈[23] 。此外，本文研究的可重构阵列系统使用动态的数据流驱动技术，在每一个计算单元之间可以存储多组数据以保证计算单元只要处于空闲状态就可以开始时新的计算，提高并行性。

输入buffer是一个存储的实体，处于处理单元的入口处。本论文中研究的输入buffer的数据结构是一个表结构，该表结构接受三个输入三个输出。其内部是按照行进行组织的，如图3‑6所示，图中每一行是一个存储体，每一行都接受有三个输入，分别为tag+valid1+data1、tag+valid2+data2和tag+valid3+data3，这三个输入时独立的，在某一时刻只要数据携带的有效标志位有效并且在表中有空位就可以将该数据存入表中，在数据被存入到表中相应的空位后，出发表的匹配逻辑，匹配逻辑是这样的：首先检查表的输入配置情况，根据输入情况分别取检查相应存储单元上的有效标志位是否有效，有效是表明该数据输入成功，否则表示该位置上还没有数据输入。当一行上输入的数据都有效后，将存储单元lineOK信号为置位1，并将该行的行号压入队列，该队列维护数据行准备就绪的顺序，在表出数时需要用到。

输入buffer是CPE中的关键部件，它能够实现对输入数据的缓存并且对其进行匹配输出。当输入的两个数据带有相同的tag标记时，被认为是相同批次的数据，可以发射进入ALU运算单元，再结合ALU中的配置码实现对该数据对的操作,完成配置中规定的操作。输入buffer可以提供存储空间，使其成为未匹配数据的中转站，基于此，该结构支持循环迭代多发射功能，多层循环间的数据被分布式的存储在各个不同的CPE的输入buffer中，tag匹配机制保证了不同循环层数之间的数据不会冲突，有力支撑了循环动态展开

输入buffer中的表结构中每一行的大小是73bit，行的深度是可配置的，本论文中设置成4，即输入buffer的存储实体部分的大小是4×73bits.Buffer出数的时候一次最大的出数位数是73bit。

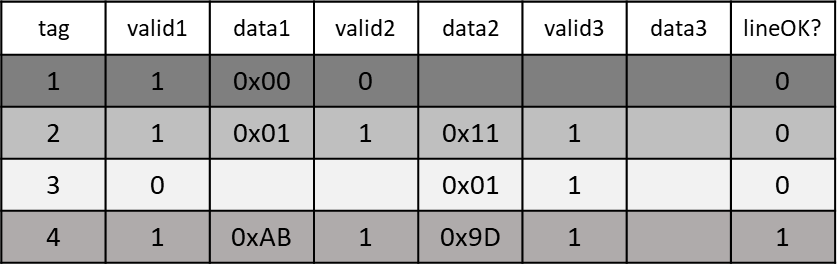


图3‑6 PE内部输入buffer组织图

Fig. ‑ The orgnization of processing element buffer

本论文为处理单元PE设置了输出buffer，其主要目的是实现在处理单元内部进行数据的存储，处理单元内部丰富的存储资源保证了动态数据流在阵列之间流动，使得进行循环级并行提供了条件。输出buffer是标准的FIFO(first in first out)结构,如所示，valid+tag+data的组合构成了FIFO的一个存储单元，FIFO的深度也是可以配置的，本论文配置成深度为4的输出buffer，由于CPE结构具有两个粗粒度的输出端口，输出buffer的个数为2。两个输出buffer的存储大小为2×37×4bits。

### 可配置ALU单元

可配置ALU单元[22] 是实现可配置计算的主要部件，通过将实现不同功能的配置码配置到ALU单元中，激活计算时ALU接受数据根据配置码指定的功能完成计算，将结果输出到相应的输出buffer上。如图3‑4所示，ALU接受3个输入，其中两个是常规输入，均为32比特。另一个是隐藏输入，主要是为一些特殊计算准备的，通过特殊的配置字可以调用ALU的第三个输入端口，一般情况下该输入端口是对用户透明的，不可见的。

表3‑1 ALU算子汇总表

|  |  |  |  |
| --- | --- | --- | --- |
| 算子 | 实例 | 算子 | 实例 |
| opcode | alu\_out(alu\_out\_b) | opcode | alu\_out(alu\_out\_b) |
| enable | // | left shift | in1<<in2 |
| signed add | in1+in2 | right shift | in1>>in2 |
| signed mul | in1\*in2 | and | in1&in2 |
| signed sub | in1-in2. | or | in1 | in2 |
| unsigned add | in1+in2 | xor | (!in1&in2)||(in1&!in2) |
| unsigned mul | in1\*in2 | +1 | in1+1 |
| unsigned sub | in1-in2. | not | !in1 |
| compare | in1 < in2 ? 1 : 0 | come | in1 == in2 ? 1 : 0 |
| MAC | in1\*in2+local\_reg | MUX | in3 == 1:in1?in2 |

ALU是由很多该阵列结构支持的算子组合而成，如表3‑1所示是混合粒度可重构阵列支持的算子列表，通过4比特的opcode来配置选择相应的算子并在ALU中激活，在通过其他互连的配置字实现阵列的可配置计算。

### 本地寄存器

LR(local reg)是CPE中的另一个重要部件，既可以作为常量寄存器，还可以作为通用寄存器。LR输入输出端口的可配置功能，使得该寄存器可以灵活使用，方便快捷的构建CPE内部的数据通路。当ALU的输出连接到LR的输入，LR的输出连接到ALU时，就可以实现累加等操作。

本地寄存器[24] 的设计灵感来自于通用处理器中的一些专用寄存器，在处理阵列中设置一个本地寄存器，可以实现很多灵活的功能。例如在一些软件语言中，有局部变量的概念，局部变量是在程序的代码块中定义的，在程序代码块运行结束之后该变量对应的存储空间被释放。而本地寄存器可以实现局部变量的某些功能，在任务开始之前通过配置文件将一些运行代码所必须要的参数提前导入到PE中的本地寄存器中，这种操作类似于实现了局部变量的赋值操作，通过这种提前赋值的操作，可以大大减少每一个处理单元的激活时间，加快计算。

另一方面，前面介绍到ALU被设计成三个输入端口，第三个输入端口就是为了实现那些需要三个操作数的操作，比如乘累加操作，其中的第三个输入来源就是本地寄存器。ALU的某个输入来源于本地寄存器，ALU的运算结果存储到本地寄存器，通过这种设置实现了乘累加的操作。

本地寄存器的作用有很多，比如，循环程序中每一层的循环都有自己的作用域，如图3‑7是三层循环作用域的层次结构，最外层循环是作用域1，最内层循环是作用域3.现假设在作用域1中有一个变量a需要在程序的运行过程中被作用域3中的代码使用，很明显变量a要在作用域3作用期间一直要有效并且要保持值得稳定。通用处理器在处理这类情况时，变量a可以存储在存储器中的任何一个地方只要编译器能告诉通用处理器其地址是多少就可以，但是在以数据流为驱动的可重构计算平台上，需要通过这样的一个节点将变量a存储在处理单元的本地寄存器上，作用域3中的代码需要变量a的时候，通过产生激活信号激励该处理单元输出变量a的值，知道该变量的值被下一个值覆盖。以上就是本地寄存器的另一个作用。



图3‑7循环代码变量作用域示意图

Fig. ‑ Scope of variable in a code

### 本地配置器

处理单元内部配置器的功能主要有两个方面，首先将ALU的操作码配置到ALU内部实现PE功能的可配置，其次将PE内部的各个部件通过互连配置字进行配置，实现数据通路的可配置。配置器[25] 主要是由配置寄存器和配置MUX组成，将配置寄存器进行分段，每一段的存储值作为配置MUX的选择信号，进行通路的选择，配置MUX全部激活完成之后，处理单元内部的数据通路被形成，所要执行的操作也被确定，整个配置过程结束。

配置寄存器的深度可以达到4层，每一个配置寄存器需要存储20比特的配置信息，整个配置寄存器堆可以同时存储4x20bit的配置信息，也就是说可以同时将4套配置包信息分布式的存储到各个处理单元中去，节约配置存取的时间。任务控制器激活某一配置包，配置信息会体现在CPE内部的配置MUX上，通过MUX的选通功能，CPE内部的数据通路被建立起来，当所需数据准备好之后，便可以激活相应功能的计算.



图3‑8 PE内部配置寄存器编码格式

Fig. ‑ Coding scheme of config register

如图3‑8所示是20bit配置寄存器的编码格式，方框上面的数字表示该配置字需要的比特数，其中每一个配置字所表示功能的详细介绍参见表3‑2

表3‑2 PE内部配置字功能详解

|  |  |  |  |
| --- | --- | --- | --- |
| 配置字 | 含义 | 配置字 | 含义 |
| opcode | Alu实现功能配置字 | end\_node | 处理单元是否是流图结束节点配置字 |
| lr\_from | 本地寄存器来源配置 | pe\_mode | 处理单元执行模式配置 |
| alu\_in1\_from | Alu输入端口1来源 | tag\_mode | 处理单元是否是动态数据流模式 |
| alu\_in2\_from | Alu输入端口2来源 | combine | 处理单元是否是组合逻辑配置字 |
| ob1\_from | 输出buffer1来源配置 | reset | 置位信号配置字 |
| ob2\_from | 输出buffer2来源配置 |  |  |

### 粗粒度处理单元数据通路设计

混合粒度可重构阵列的可重构特性体现在三个方面，处理单元外部的互连关系可以配置；处理单元内部的数据通路[26] 可以配置；处理单元实现的功能可以配置，以上三种不同层面的可配置特性保证了该阵列配置的灵活性。

如图3‑9所示，从整体上看，CPE内部可以分成三个级别，第一级别是输入buffer，第二级别是ALU运算单元，第三级别是输出buffer，所以内部的数据是按照这种级别在流动，首先，从输入端口Din1/Din2/Bin进入到输入buffer中，其次输入buffer匹配完成之后，将匹配后的数据输入，数据进入到ALU运算单元中进行运算，最后运算的数据从ALU输出，存储到输出buffer，输出buffer在配置信息的作用下将数据输出到相应的另一个CPE中去，数据完成从一个处理单元到另一个处理单元的传输。

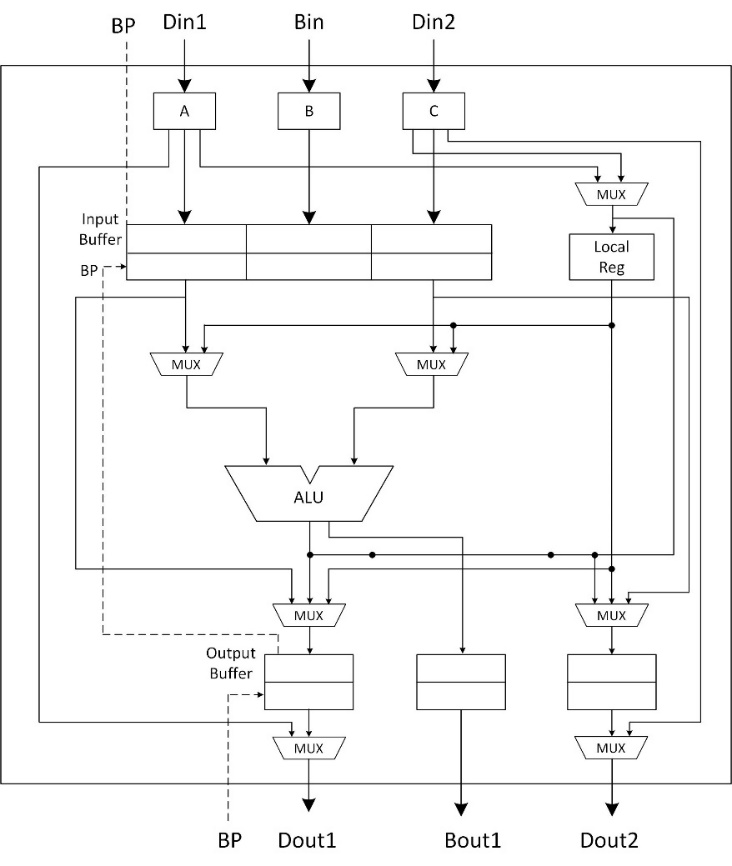


图3‑9 CPE内部数据通路图

Fig. ‑ Datapath of coarse-grained processing element

图中A,B,C是MUX选择器，根据配置的不同将输入的数据选择不同的输出端口进行输出，以达到对数据通路的配置目的。图中其他的MUX是本地配置器的组成部分，通过读取配置寄存器相应位段的内容，根据位段中存储的内容进行MUX的选择操作，最终实现CPE内部数据通路的构造。

图34是粗粒度处理单元内部数据通路的整体展示，而在实际的运行过程中只有某些数据通路是有效的，是和处理单元的计算模式密不可分的，下面介绍一下处理单元在不同计算模式下内部的数据通路。

（1） 计算模式

计算模式是可重构阵列执行任务中出现频率最多的一种模式，是指数据从输入端口进入到数输入buffer中去，再从输入buffer输出到alu参与计算，最后alu将计算结果存到输出buffer中去，完成计算。

（2） 中转模式

中转模式是为了解决在数据流驱动情况下，原始代码中一些变量作用域和作用时间不同而设置的一种计算模式，如图3‑10所示，左图中变量j要在内存循环有效期间一直有效，其生存周期要比变量j的生存周期长；右图中变量temp在循环期间一直被改变，其生命周期要比变量sum短，针对这种变量之间生存周期不一致的问题，需要一个节点延长变量的生存周期，保证数据流在阵列之间不会发生错误。中转模式正式为了解决这个问题而设置的计算模式，通过粗粒度处理单元来实现。

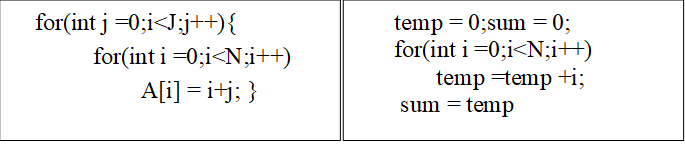


图3‑10变量作用域和生存周期示例

Fig. ‑ Examples of variable nature

（3） 旁路模式

旁路模式是为了阵列在布局布线时提供的一种解决方案，阵列的布局和布线关系到互连网路的复杂程度，也关系到阵列计算效率，增加旁路模式使得处理单元内部的模块处于闲置的状态，数据从输入端口进入直接传输到输出端口，从布线角度看来，数据流过处理单元和数据流过一条有电阻的导线没有区别

## 细粒度处理单元结构设计

针对计算密集或数据密集类算法，其代码的主要内容是实现计算，没有太多的循环控制和分支跳转，粗粒度阵列对这种类型的算法有很好的支持，但是对于控制密集型的算法粗粒度阵列结构不能完美的支持。正是由于这种限制我们提出了基于细粒度阵列实现代码控制流图（CFG），粗粒度阵列实现代码数据流图(DFG)的思想，提出让细粒度节点作为代码中循环控制和分支跳转等功能的实现载体，让粗粒度节点专注于计算。[27]



图3‑11分支跳转代码及数据控制流图实例

Fig. ‑ Branches and data-control flow graph

如图3‑11 (a)所示,是一段循环结构的C语言代码，图3‑11(b)是这段代码展开一层循环的CDFG图，从(b)图可知，循环是从a节点开始的，也中止于a节点，a节点的功能就是每次有一个结束信号输入的同时就产生一个输出激活信号，由于这些信号属于控制信号，参与运算的位宽不超过两位，适合于使用细粒度的处理单元进行处理。通过细粒度节点a可以实现循环步进和循环边界的控制.图中b节点是一个选择通路的节点，对应于代码中就是if-else结构，也是一种细粒度功能的节点。

### 单元内部构成

细粒度处理单元是进行任务控制的主要成员，文中研究的细粒度的处理单元是由4个细粒度的输入端口和一个细粒度的输出端口组成，如图3‑12所示，细粒度节点的输入输出的位宽是6bit,其位数编码安排参见图3‑5所示，6比特是由1比特的信号有效位、4比特的数据标记为和1比特的布尔运算位组成，输入数据经过入口控制逻辑后，进入查找表，查找表是由RAM组成，内部存储着四输入的真值表结构，通过配置字提前写入进去，当四个输入拥有不同的二进制值得时候，通过查找和选择出相应的真值表中的值输出到输出端口，从而实现查找表规定的逻辑功能。细粒度处理单元中的查找表是一块存储器，可以通过改写其存储内容实现特定的细粒度节点功能。从而实现对代码中循环控制和分支跳转的支持。

入口控制逻辑的功能是实现输入数据的缓存和匹配输出，其结构是一个表结构的存储实体，具体参见图3‑6，四个端口的输入数据通过输入端口选择输入buffer中的空位置存储数据，并激活该存储行的监察电路，当该行的四个输入全部存储了最新的数据，该行为置为就绪行，就绪行的数据可以输出到4输入的查找表结构中去，得到输出，一次细粒度的处理操作完成。和粗粒度处理单元中输入buffer不同的是，细粒度处理单元中的输入buffer是要根据数据进入单元的顺序输出4个端口的结果到查找表中进行查找操作，而粗粒度处理单元在执行这一步时并不需要维持这种顺序，粗粒度处理单元的输入buffer是一种优先的表结构，先就绪的数据行可以提前进入到ALU中进行计算。这是两者不同的地方。



图3‑12 细粒度处理单元内部结构简图

Fig. ‑ inside view of fine-grained processing element

### 细粒度控制算子

细粒度处理单元的功能是可以配置的，可以通过配置其内部的查找表改变其实现的逻辑功能，其作用是实现任务代码中的控制流图部分，细粒度处理单元还可以进行计算任务中的一些逻辑操作。原理上，可以通过4输入的查找表配置出需要的逻辑功能，本文基于任务出发，分析得出如表3‑3所示的细粒度节点常见的逻辑功能，表3‑3同时显示了不同逻辑功能在阵列执行过程中的作用。

表3‑3细粒度处理单元支持的算子

|  |  |  |  |
| --- | --- | --- | --- |
| 细粒度节  点算子 | #输入/#输出 | 实现功能 | 对应C code  结构 |
| loop begin | 2输入/1输出 | 输出循环迭代开始信号 | 循环开始控制语句(for等) |
| loop end | 2输入/2输出 | 输出循环迭代结束信号 | 循环结束控制语句-(for等) |
| Join | n输入/1输出 | 数据流路径汇总 | 代码块结束符‘}‘等 |
| Switch | 1输入/2输出 | 激活分支路径 | if--else结构 |
| Break | 2输入/1输出 | 激活break路径 | break结构 |
| Joinbp | n输入/1输出 | bp信号的路径汇总 | 循环间依赖 |

Loopbegin算子和loopend算子是实现对循环头开始和结束的控制，如图3‑11中的节点a所示，节点a是以上两个算子的组合，对循环迭代进行控制；join算子对应的节点实现数据流图中路径的汇总，在每一条分支的数据流图路径都产生结束信号，经由join节点产生一个路径结束信号表示该数据流图的结束；switch算子对应的节点用于实现代码中的if-else结构；break算子对应节点用于实现构建带有break的代码的数据控制流图，实现代码中的break功能。Joinbp算子对应的节点用于处理这样的场景，一个输出端口扇出到多个输入端口是的bp信号一致性问题，joinbp汇总bp信号，反馈给上一个CPE的输出端口，保证了反馈机制的正确性。

### 细粒度处理单元内部数据通路设计

细粒度处理单元是通过一个四输入的查找表来实现的，从原理上看原始数据输入到查找表的四个输入端口上进行查找操作输出结果，细粒度计算过程结束。但本文研究的处理单元是和粗粒度处理单元协同工作，整个阵列工作在静态数据流和动态数据流下，进入到处理单元的数据携带有数据标号，所以细粒度处理单元数据进入单元内部是需要进行匹配操作，再输出到查找表中进行查找表操作。

如图3‑13所示，Bin1,Bin2,Bin3,Bin4是细粒度的输入，带有数据标记的输入数据存储到细粒度处理单元的输入buffer中，经过匹配操作，带有相同数据标记的输入数据从输入buffer输出，进入到查找表，选择相应的逻辑，输出到细粒度处理单元的输出端口。

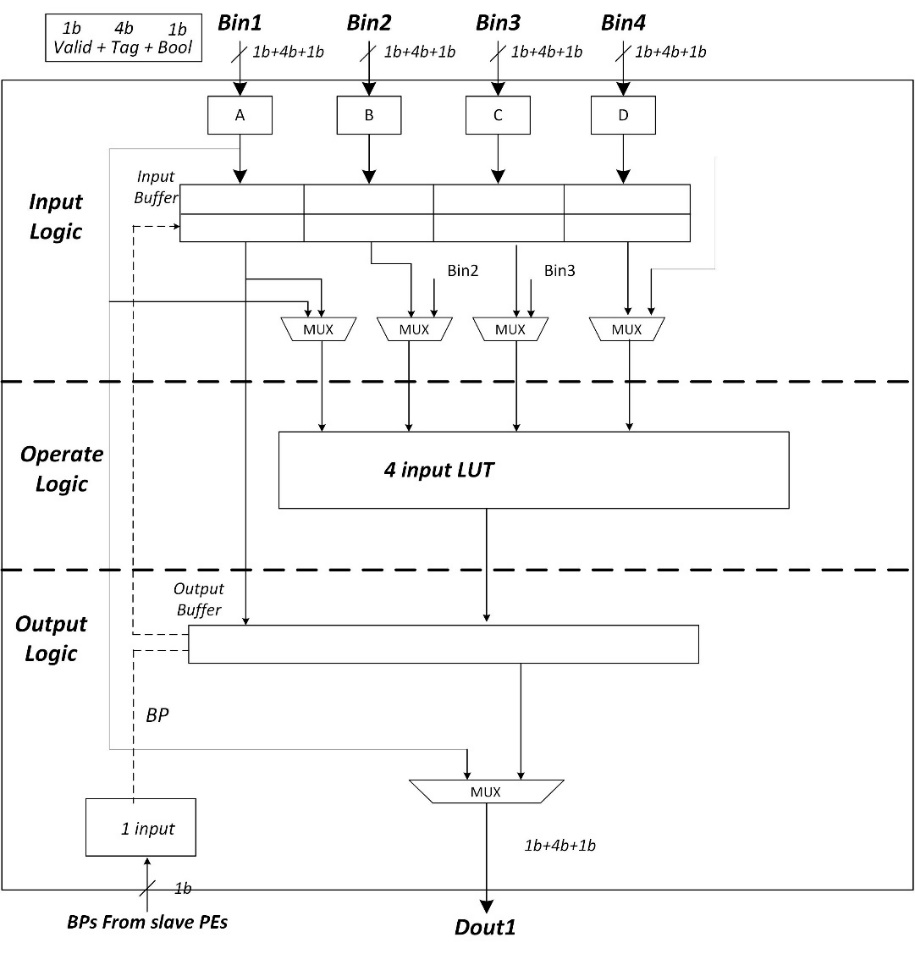


图3‑13细粒度处理单元内部数据通路

Fig. ‑ Datapath of fine-grained processing element

## 阵列反馈机制

### 数据流和循环间依赖

在控制流驱动的计算机中，程序代码被编译成指令序列，CPU通过调取指令来执行相应的功能，由于通用计算平台的硬件寄存器资源比较有限，指令中的数据需要存储到寄存器中去，在寄存器中存储的数据还没有被使用的情况下新的数值不能够写入到寄存器中，由于结构冲突产生依赖。

在数据流驱动的计算平台中，循环间依赖依然存在。在静态数据流驱动的计算模式中，其执行循环的机制见图3‑14所示，图中显示了5层循环在时间维度上的执行示意图，在静态数据流驱动情况下，每一层的循环执行完成之后，才会激活下一层的循环代码继续执行，在整个阵列中只存在一个批次的数据，将代码中循环间依赖在时间上铺展开来，从而消除数据依赖的影响，但是这种执行机制带来了执行效率低下的缺点。



图3‑14 静态和动态数据流执行方式

Fig. ‑ Static and dynamic data flow drive

本文使用动态数据流驱动技术，只要空间硬件资源足够，循环块可以快速发射到阵列上去进行执行，如图3‑14所示，假设某一循环代码有5层以上的循环代码需要执行，在使用动态数据流驱动技术后，第三层的循环块有可能在时间上超前循环块2被执行，如果两层循环之间不存在数据依赖关系，这种执行机制不存在问题，但是如果两层循环之间存在数据依赖关系，超前执行的循环块会导致整个阵列执行的代码发生错误，产生错误的结果。

另一方面，数据流驱动的优势在于只要处理单元产生数据，在时钟的驱动下会立即传递到下一个需要该数据的处理单元，在两个处理单元的输入弧上就会堆积多个有效数据，但是如果两个处理单元之间的输入弧上的存储空间不足，会导致有些有效数据还没有被下一个处理单元使用而是被数据重写覆盖掉，这也会导致阵列计算的出错。

为了解决以上问题，在阵列的处理单元之间和处理单元内部引入了反馈机制，从而保证数据流能有次序、有节奏的出现在各个处理单元之间的输入弧存储空间上。

### 多层次反馈机制

多层次的反馈机制在于保证数据流能有节奏的出现在各个处理单元的存储机构中，使得数据流能均匀的分布在整个阵列上，减少数据流驱动技术带来的数据存取冲突，加强阵列处理程序的健壮性。

首先是处理单元之间的反馈机制，如图3‑15所示，在两个处理单元PE1和PE2之间的某一对输出输入端口上分布着存储单元，但是存储单元的空间不是无限大的，所以需要通过反馈线来牵制存取数据，反馈机制中基本的原则是每一条数据输出线都有与之对应的反馈线（bp信号线），用来表示是否能将数据继续存储到下一个处理单元中。图中左图是一个输出端口直接输入到下一个PE的输入端口，是一对一的连接方式，PE2在运行期间会先检查自身的输入buffer是否能放置PE1传输过来的数据，在PE2检查期间会将这根bp信号线置成相应的值，PE1在出数的时候会先检测该信号线在决定要不要出数。

图中右侧是一个输出端口输入到多个PE的输入端口上，是一对多的连接方式。这种情况下的反馈要复杂一点，其实现的原理也是通过反馈信号线实现的，在运行期间，PE2和PE3均需要提前检测自身的输入buffer是否能放置PE1传输过来的数据，检测之后产生的信号需要通过一个细粒度处理单元(图中节点a)处理之后，传递给PE1，PE1在检测信号的值之后再决定是否将数据输出。

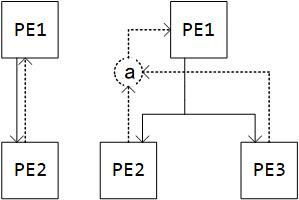


图3‑15处理单元之间的反馈图

Fig. ‑ Feedback between processing element

阵列中的所有处理单元都设置有这种反馈机制，从而保证了数据的均匀分布，充分利用阵列的硬件资源，保证了程序在阵列上的正确执行。

其次是处理单元内部的反馈机制，处理单元内部进行任务处理主要分两各部分，第一部分是上一个PE的输出buffer到当前PE的输入buffer，第二部分是当前PE的输出buffer经过ALU到达当前PE的输出buffer，如图3‑16所示，从输入buffer中取出数据进行ALU运算再将结果存储到输出buffer中，存在一个问题，如果输出buffer被数据占据没有存储空间，那么经由ALU运算产生的数据就会丢失，所以在ALU进行运算之前需检查输出buffer的状态，有存储空间即反馈一个信号给输入buffer控制逻辑，告知其可以出数并进行ALU运算。图中B.1,B.2,B.3分别表示三个输出buffer的反馈信号，A.1,A.2,A.3表示的是输入buffer传递给上一个PE的反馈信号，也就是处理单元之间的反馈信号。

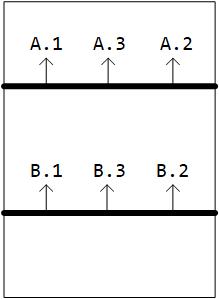


图3‑16处理单元内部的反馈示意图

Fig. ‑ Feedback in processing element

## 混合粒度阵列的优化技术

### 多发射循环迭代技术

在传统处理器上循环程序被编译成顺序执行的指令，新的循环迭代必须要等到上一次循环迭代结束之后才开始执行。如图3‑11(b)所示，一般的执行流程是store节点结束之后，a节点才开始激活下一次的循环迭代。这种执行方式显然不适合HGRA结构，本文提出了一种多发射循环迭代的技术，实现了循环体在阵列上流水。如图3‑17B所示，矩形表示的是每一次的循环迭代，多发射技术实现的是在循环迭代1还没有结束的时候循环开始控制节点就发射循环控制变量，激活循环块2，再激活循环块3，以此类推，按照这种方式实现了循环间的流水，最优情况下细粒度节点可以每一个时钟节拍发射一个循环控制变量，直至循环边界，大大提高了计算效率。[28]



图3‑17不同配置方案空间消耗和时间消耗

Fig. ‑ Time and space lapses in different configuration scheme

### 空间展开循环迭代技术

上面介绍的多发射循环迭代技术是发掘程序中的时间可重叠性，由于HGRA结构中在某一固定时间窗口内CPE执行的功能固定，这为将更多的运算需求分配到其他闲置的计算单元上提供了可能。空间展开循环迭代技术充分挖掘了程序中的空间可并行性，如图3‑17C所示，循环迭代1,2,3在空间上展开，属于不同循环块的计算单元可自主执行计算任务，不相互干扰，性能获得较大提升。如图3‑18



图3‑18两种优化技术相结合示例

Fig. ‑ The combination of two optimization techniques

是两个4x4矩阵相乘的例子，使用空间展开技术，一个时钟节拍可以完成第i行与第j列所有元素的相乘，即阶段1完成的操作，通过使用4个CPE同时执行将代码中最内层的循环展开到空间阵列上，整个代码从3层循环减低到2层循环，降低了循环的维度。[29]

通常针对某一种算法，可以同时使用两种优化技术，如图3‑18所示，四个乘法的CPE是空间展开循环迭代技术的应用，该技术的应用可以降低循环层数，减少配置复杂度。同时多发射循环迭代技术的应用，使得阶段1到阶段3以流水线的方式执行计算,充分挖掘了时间可重用性.两种技术的应用为高效率计算创造了条件。

## 应用映射实例

### 矩阵乘映射到两层循环方案

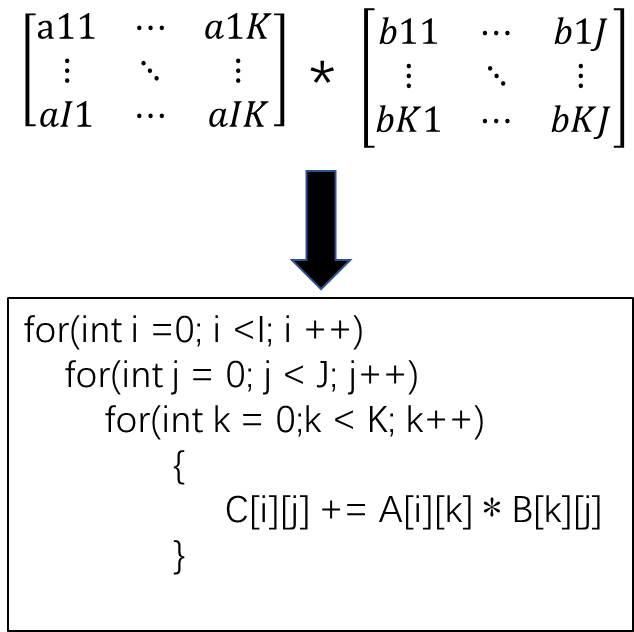


图3‑19矩阵乘代码

Fig. ‑ Code of matrix multiplication

矩阵运算在图像处理和科学运算中有着广泛的应用，在图形和图像处理领域中，很多算法的本质也都是进行矩阵的运算，这类算法适合于可重构计算，因为可重构计算中大量的空间计算资源可以加速处理过程，提升计算效率。

如图3‑19所示，是一段矩阵相乘的代码，其是由三层循环构成，最外层是选择A矩阵的行，第二层循环是选择B矩阵的列，第三层是选择行列找中的某一个元素，其最主要的运算是将某一行的矩阵元素和某一列的矩阵元素分别相乘最后求和得出新矩阵的一个元素。

如图3‑20所示是将三层循环展开到两层，将最内层循环执行的内容展开到空间上，内层的K次循环在空间上并行执行，充分利用可重构阵列中的空间源。图中体现的空间展开技术，在代码层面上可以用上另外一种可重构阵列的优化技术，循环多发射技术，即以图3‑20为处理块对第二层的循环进行动态的数据流展开，实现循环间的流水，也就是图中算乘法的CPE在计算完乘法操作输出运算结果之后，可以立即开始进行下一次循环迭代的乘法操作，在处理单元上实现了流水，充分发挥阵列的并行性。



图3‑20矩阵乘两层循环映射方案

Fig. ‑ Mapping scheme one of matrix multiplication

### 矩阵乘映射到一层循环方案

对于矩阵乘应用，最大并行性的实现依赖于可重构阵列中硬件资源的丰富程度，在硬件资源丰富的前提下可以实现将三层循环展开到只有一层循环。其具体做法是，以图3‑20所示的处理块为基础，将处理块进行空间展开，B矩阵的每一列和每一行被展开到空间上，形成多个处理块，对矩阵A的行进行遍历实现循环流水，这些处理块在数据准备完成的情况下可以并行的实现计算，极大的提高可重构阵列的计算效率。

另外一种映射到一层循环的方案是脉动矩阵的方式[30] ，如图3‑21所示，计算阵列是固定的，将两个矩阵的数据流动进入计算阵列，进入到处理单元的数据再处理单元计算完成之后从处理单元的输出端口输出，再接着进入到下一个计算单元继续进行计算，这种脉动的方式降低了可重构计算中数据存取瓶颈带来的性能损失，数据流动中一份数据被多次使用，不需要重复从存储器中加载，减少了存取数据的时间，是阵列专注于计算，提升了计算效率。

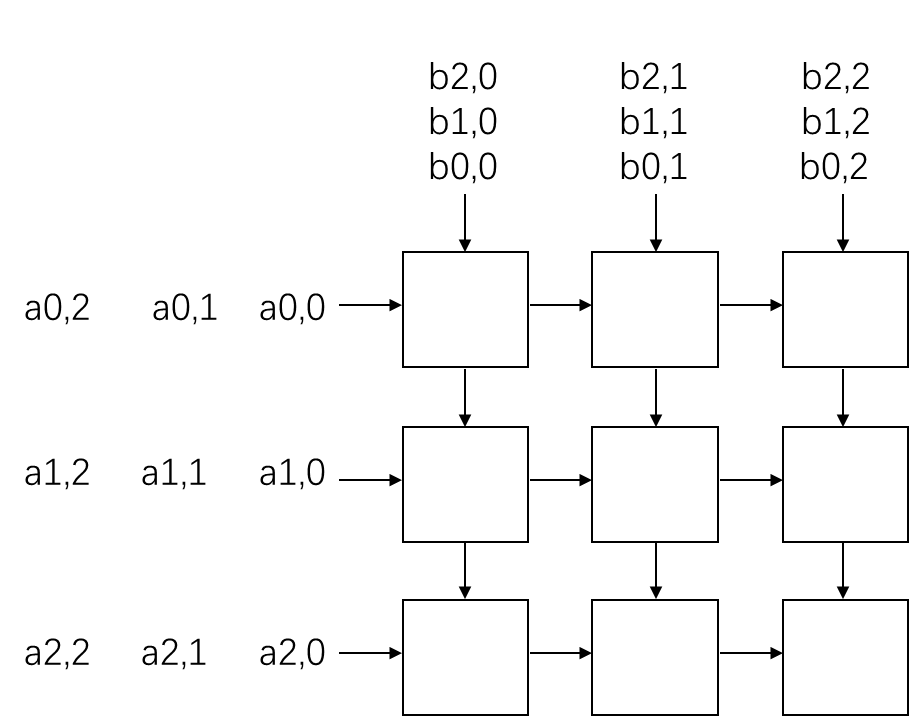


图3‑21脉动矩阵方式实现矩阵乘

Fig. ‑ Implement matrix multiplication by systolic array

## 本章小结

本章主要介绍了整个阵列上各个模块的设计和实现，分别介绍了粗粒度处理单元的设计和内部的数据通路、细粒度处理单元的设计和数据通路，然后介绍了阵列中处理单元之间的反馈机制和处理单元内部的反馈机制，介绍了混合粒度可重构阵列的在性能的优化技术，最后，通过矩阵乘的应用实例介绍了使用优化技术时的映射方法。

# 第四章 混合粒度可重构阵列的仿真实现

## 仿真原理概述

随着电路设计的规模越来越大，电路架构变化的频度越来越快，传统在电路设计结束后进行整个电路验证的方法已经不能适应这种快速变化的场景，使得电路架构设计人员变得被动，不能快速调整设计方案，整个电路的研发周期因此变得很长，容错成本极高。所以在整个电路架构设计阶段有一款能快速验证架构设计人员所设计的电路架构是否满足设计要求的工具变得极为重要。一般的电路设计中，这个阶段通常是缺失的，依靠架构设计人员丰富的专业技能来保证正确性往往是不够的。

在架构设计初期，需要确定整个电路的功能和性能，在这一阶段设计的电路是一种较高层次的行为级电路，所以比较适合对整个电路进行软件上的建模，通过软件来描述所要设计电路的具体的功能模块，通过软件的方式实现每一个模块中的功能，这就是软件仿真器需要实现的功能。[31]

设仿真器是一套软件工具，在芯片架构设计中扮演着重要的作用。通常来讲，我们要运行一段程序在计算机系统上需要有以下几个必要条件，

操作系统----对程序任务进行划分和调度

编译器----将代码编译生成硬件电路能识别的指令

硬件电路----执行具体的指令

在以上三种条件中，缺少任何一项程序都不能被完整的运行，以上三个条件的关系如图所示，层次关系是越往下越重要。现在我们面对的情况是最底层的硬件电路是我们需要设计的，那么其他任何基于硬件电路上的工具都没有办法使用。而软件仿真器的实现原理是通过将需要设计的电路的软件模型构建出来，运行在通用的处理器平台上，如图，通过已经存在的一套环境来构造一套软件工具，软件仿真器试运行在确定的操作系统之上，但是这是对所要设计电路的一种抽象，运行在通用平台的软件仿真器从外部来看就是需要设计的硬件电路的模拟。[32]

如图4‑1所示，在硬件平台上运行应用或者算法，是借助于操作系统进行任务的调度，指令是直接运行在最底层的硬件电路上，而仿真器是运行在操作系统

图4‑1程序运行在硬件平台上和仿真器平台上

Fig. ‑ Programs run on hardware platform and simulation platform

上的一种大型系统软件，其作用是借助于已经存在的硬件资源为应用和算法运提供一个HGRA计算平台，仿真器是一个中间件，需要将HGRA中的一些指令操作映射到最底层的硬件电路电路上，对于上层应用或者算法而言，仿真器以下的部分是透明的，实现了具体电路的软件化。

## 仿真器整体框架

本论文所设计的混合粒度可重构阵列仿真器是基于X86平台，Windows10系统下，基于visual studio开发。如图4‑2所示，仿真器需要描述出阵列上的处理单元、执行控制器、任务调度控制器、主核接口和存储器系统等模块部分。仿真器本身提供的是一个框架，会根据读进来的原始配置文件和原始输入数据重构模拟的电路结构。整个仿真系统主要分成两个主要大的部分，物理模块和交互模块，下面分别介绍：

1. 物理实现部分

组件系统：组件系统主要是由静态的模块组成，包括细粒度的处理单元、处理度的处理单元、Load/Store单元、互连网络单元等。通过软件代码描述硬件底层的各个组件，将其作为一种使用的实体存在。这些模块是独立的，静态的，是一个仿真系统必不可少的部分。

数据流控制系统：这部分是在读入配置信息后，模拟硬件行为，将配置信息分别存储到每一个可重构单元的配置寄存器中去，再通过配置寄存器配置处理单元内部的配置MUX，使其形成配置信息指定的电路，形成配置电路后仿真器从原始输入读取数据，数据流开始进入到阵列，控制系统的职责就是要在此时对数据流的流动进行协调和控制，控制的规则就是架构设计时预定的规则，包括处理单元外部数据流动规则和处理单元内部数据流动的规则。有了这些规则的保证和控制系统对这些规则的执行，使得仿真系统能能真实的模拟出这些规则是否是正确的。

存储模块系统：任何一种计算平台的存储系统都是复杂的，在前面介绍过可重构阵列系统的存储系统是分层次的。由于可重构阵列是和主核处理器组成一个多核系统，在进行仿真器设计的时候需要实现多层次的存储系统，包括为仿真器提供原始数据的外部存储器、一定容量和速度的内存以及cache缓存。这些组件作外为挂模块连接在仿真器上，在整个仿真器运行期间提供访存支持。

1. 仿真功能实现部分

任务控制系统：在可重构阵列和通用CPU组成的异构计算平台中，任务的划分和调度至关重要，而任务控制系统在这里面扮演者重要的作用，控制系统需要在主核协调下，对需要加速任务进行划分，同时协调任务段需要被执行的硬件资源地址，对整个计算阵列进行监控，在某一硬件执行资源执行完毕之后要对阵列资源进行回收和初始化，同时将新的任务段安排到该执行块上去。所以，任务控制系统是进行任务级调度的枢纽，关系着任务执行的效率。一般的任务控制流程是先开始对阵列的硬件资源进行初始化，接着开始注入数据开启阵列运行任务，控制器不断监控硬件资源状态，进行调度，最后任务结束后，进行任务的整体交付。

交互系统：交互系统是面向仿真器使用人员的，一个好的交互系统可以方便程序调试人员进行代码调试。本文设计的仿真器的交互系统包括两个部分，分别是仿真环境搭建和调试接口的配置，首先在开始任务前配置仿真环境，包括处理单元内部输入输出buffer的深度，和存储器系统中的cache大小。Cache-line大小的设置，通过配置这些可选参数搭建好仿真环境；第二部分是调试接口的配置，程序员分析应用之后生成配置信息文件和参数文件，这些文件需要提供给仿真器，仿真器读取这些原始数据后进行阵列的配置并进行执行，最后将仿真结果反馈到指定的结果文件和日志文件中。



图4‑2仿真器实现的模块框架

Fig. ‑ The module framework of simulation

## 仿真器实现

### 模块实现

仿真器设计是层次化和模块化的，通过对混合粒度可重构阵列进行行为级的建模，使用软件语言实现了阵列系统。层次化体现在仿真器是分成不同级别的层级，最底层是各个模块的具体功能定义，实现对硬件结构的描述；中间层是各个模块的动态调度层，仿真时根据数据流进行模块调度实现仿真任务；最上层是时钟驱动层，驱动整个阵列在同步时钟下执行任务并进行时钟统计。

模块化设计体现在整个阵列中的各个功能部件都是分模块的，具备低耦合的特性，模块化的设计可以降低测试代价，增加可扩展性，有利于进行系统设计。模块的设计分为两个部分，分别是概念设计和物理设计，概念设计阶段进行模块功能定义和接口定义，是每一个模块对硬件功能的抽象，接口设计是为了可扩展性要求进行的，通过接口设计可以将模块封装起来，通过接口和其他的模块进行交互。物理设计是指模块功能的编程实现，通过采用不同的数据结构和函数等软件工程手段实现模块定义的硬件功能，体现对硬件最真实的模拟。[33]



图4‑3阵列仿真器系统设计流程

Fig. ‑ Design flow of simulaton system

如图4‑3所示，是阵列仿真器实现整个系统的实现过程。仿真器设计分别按照模块层、中间调度层和驱动层的层次吮吸进行设计，设计采用自底向上的设计方法。分别包含处理单元模块(CPE/FPE/LE/SE)、控制系统模块和存储系统模块等静态模块设计，中间调度层要实现配置信息的解析、配置信息固化和处理单元内部函数调度等功能和模块，驱动层要实现软件仿真器模拟硬件行为的时钟驱动以及对整个系统的时钟周期的建模。设计中每一个层次的每一个模块需要进行验证，最后将模块进行整合形成阵列系统的行为仿真器。

1. 模块层：
2. 处理单元

处理单元软件实现主要包含内部的数据结构和一些内部执行方法，通过一个类来实现，表4‑1是处理单元类的方法和解释。

表4‑1 PE类内部实现方法

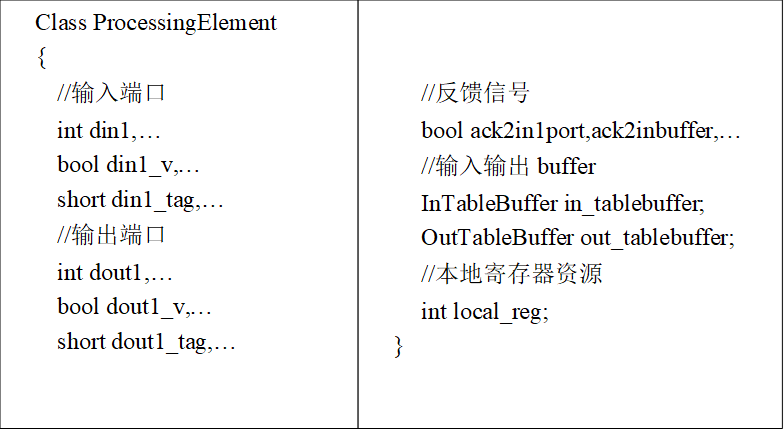
|  |  |
| --- | --- |
| 函数名 | 解释 |
| void inbuffer\_in() | 数据写入到输入buffer中 |
| void inbuffer\_out() | 输入buffer中数据进行tag匹配并输出 |
| void alu() | 执行ALU操作 |
| void outbuffer\_in() | Alu运算结果写入到输出buffer中 |
| void outbuffet\_out() | 从输出buffer中读出数据到输出端口 |
| void valid\_clear() | 输出端口数据读走之后进行有效位清零 |
| void clock\_cnt() | 处理单元时钟周期的统计 |

上表显示的是处理单元内部一些功能的实现函数，下面介绍一下处理单元类内部的数据结构和接口，如下所示，处理单元主要是有两个37bit的粗粒度输入端口、一个6bit的细粒度输入端口、两个37bit的粗粒度输出端口和一个6bit细粒度的输出端口组成，内部主要是对整型数据进行存取和运算，实现如下，内部主要包含输入输出端口的定义和输入输出buffer的定义。

1. 控制系统模块

控制系统主要进行任务的控制，实现阵列执行任务的自主化，其要实现的功能从对各种状态进行维护，要通过软件的手段设计一个状态机进行阵列中各个部件的控制，其主要流程如下：从主核接受任务信息，发送任务一的配置启动信息，配置控制器开始执行任务一的配置信息导入和固化工作，检测到任务一配置完成信息，启动任务一的执行，在检测阵列运行状态期间检查处理单元中配置寄

存器状态，如果配置寄存器有空的位置，则并行的通知配置控制器进行任务二的



配置，在收到阵列对任务一执行结束信号之后，启动任务二的激活信号，以此类推，知道整个任务执行结束，反馈结束信号给主核，整个任务执行结束。下表所示是控制模块内部的主要实现方法。

表4‑2 control类内部实现方法

|  |  |
| --- | --- |
| 函数名 | 解释 |
| void task\_receive() | 接受主核的任务和参数信息 |
| void config\_reg\_supervise() | 配置寄存器监控并激活配置导入 |
| void status\_supervise() | 阵列状态监控 |
| void task\_start\_end() | 任务激活信号注入 |
| void task\_feedback() | 任务结束反馈结果给主核 |
| void clock\_cnt() | 时钟周期的统计 |

1. 存储系统模块

存储系统模块是为了仿真器能更加真实的贴近硬件的行为，本设计的存储系统是分层次进行的，包含存储器实体和cache缓存。实现了存储器读写和对cache的支持，cache的特性可以配置，cache大小，cache-line大小，替换策略等均可通过软件字段更改，搭建合适的仿真环境。[34]

中间调度层主要是以配置信息为对象，实现配置信息中执行内容的调度和实现，实现配置信息从软件配置字到硬件配置字的解析，保证模块层中各个模块对硬件配置字的完美识别。中间层主要包含几个主要的函数，其调用各个模块属性方法，实现配置中规定的行为，将静态的模块调度起来实现了动态执行，如表4‑3所示是调度函数列表和解释。

表4‑3中间层调度函数表

|  |  |
| --- | --- |
| 函数名 | 解释 |
| void pe\_sim\_step1() | PE从上一个单元端口处取数存入到输入buffer，并从输入buffer输出 |
| void pe\_sim\_step2() | 读取输入buffer的数据执行alu计算 |
| void pe\_sim\_step3() | 读取alu结果，写入输出buffer并在输出 |

驱动层解决的整个阵列执行的驱动问题，在每一次时钟周期内，上层的驱动函数被调用，每一个模块在该时钟周期内执行相应的属性方法，通过这种形式实现对硬件电路的模拟，上层驱动主要是由函数来实现，如图所示，其表示的是不同层次的函数的调用关系，从图中可以看出，在上层驱动函数被执行时，会调用中间层调度函数，而中间层调度函数会调用底层模块的属性函数，最终完成行为的模拟。

表4‑4函数层次调用关系

|  |  |  |
| --- | --- | --- |
| 函数调用关系（调用>>被调用） | | |
| void PeSimProcessing() | void pe\_sim\_step1() | void inbuffer\_in() |
| void inbuffer\_out() |
| void pe\_sim\_step2() | void alu() |
| void pe\_sim\_step3() | void outbuffer\_in() |
| void outbuffet\_out() |

驱动层要解决的另一个问题是时钟周期的模拟问题，在本文中时钟周期的定义如下：PE内部数据经过处理单元分为两个部分，步骤1（S1）和步骤2（S2），S1表示数据从上一个PE的输出buffer到当前PE的输入buffer的过程，S2表示的数据进入ALU运算并将结果存储到当前PE的输出buffer的过程。仿真器中模拟的时钟周期是每一个时钟周期步骤2和步骤1都进行数据的读取操作，进行数据的锁存。但是数据是否被锁存进入到某一个处理单元的buffer中是要检查反馈信号的值来确定。反馈机制保证了数据锁存不会发生错误。从图4‑4可以看出，执行简单的ALU操作时，一个处理单元消耗两个时钟周期。对于复杂的alu操作，不同算子对应着不同的时间消耗，更加贴近硬件时钟周期的概念。



图4‑4 PE时钟划分图

Fig. ‑ Diagram of clcok division in processing element

### 系统级仿真平台实现

根据图4‑3所示，在阵列系统设计完成之后，将主核系统、存储系统整合在一起，形成一个完整的仿真测试平台，在这个系统中可重构阵列是用软件构建的仿真器平台，主核系统是基于RSIC-V开发的控制平台，存储系统采用的是马里兰大学开发的周期精准的DRAMsim仿真期器，构建的整体系统能够在主核指令的控制下自主的完成仿真任务。

如图4‑5所示，整个系统中我们模拟了在主核的控制下，DMA将阵列需要的原始数据和配置信息从主核的存储器中搬到阵列本地的存储器上，接着主核发送任务控制指令，阵列读取到指令之后开始从本地的存储器中取出配置信息配置到处理单元中，接着被配置的处理单元开始按照固化的电路读取数据，阵列开始执行任务，最终将运算结果存回到主核处理器对应的存储器中，整个任务执行结束。



图4‑5系统验证平台

Fig. ‑ System verification platform

## 仿真流程

仿真器进行算法的仿真主要分为两个步骤，首先程序员要根据应用和硬件资源分析代码，进行代码的改写，最终生成适合于阵列运行的配置信息文件和参数信息文件，这一步骤成为算法映射；最后将生成的配置信息和参数信息导入到仿真器中，注入启动信号，开始阵列的仿真运行。

### 算法映射

通过将算法的具体代码映射成配置信息和阵列中处理单元需要的参数信息，算法的具体内容被分布式的同步到阵列上的处理单元中，算法映射到阵列上执行的主要步骤如下：[35]

（1）分析算法和代码改写，在算法映射之前对算法解决的问题和内容进行了解，编写算法实现的具体代码，并优化代码，减少冗余，使代码能够更高效的在阵列上处理。

（2）从算法代码中提取数据控制流图（CDFG），程序代码主要是由数据流和控制流组成的，将代码分成数据流图和控制流图两部分，为将代码分别映射到粗粒度处理单元和细粒度处理单元上做好准备。

（3）将数据控制流图和阵列空间结构相结合，在阵列上构建数据流图和空间流图，编写配置信息文件和参数文件。如图所示，是矩阵相乘进行配置形成的配置信息文件截图，配置信息中每一行表示被配置的一个处理单元，配置信息文件中的行数就是当前算法需要的处理单元个数。



图4‑6算法映射及验证流程

Fig. ‑ Algorithm mapping and validation process

### 阵列运行

在算法映射阶段主要产生了配置信息文件和参数信息文件，这是阵列能正确执行算法给定内容的必需信息，生成的这两个信息文件主要存储在阵列的全局存储器上，主核控制存储地址。仿真器模拟阵列执行任务主要分为以下几个步骤：

1. “硬件”初始化，首先根据应用类型初始化仿真器参数，生成一个具体的仿真环境供任务执行。
2. 读取配置信息和参数信息，仿真器中的任务控制器在任务开始时按照主核给定的配置信息和参数信息存储的地址，读取相应的信息并通过配置路径存储到处理单元的配置寄存器中去；
3. 阵列自主运行阶段：整个可重构阵列根据配置信息指定的功能和连接关系执行相应的动作并将数据传输到指定的处理单元中去，在阵列运行结束时数据会被存储到本地存储器或全局存储器，阵列处于闲置状态；



图4‑7 HGRA仿真过程示意图

Fig. ‑ Schematic diagram of simulation process in HGRA

1. 任务控制器进行任务切换：任务控制器时刻监控配置信息在阵列上的执行情况，在一套配置信息执行结束阵列处于闲置状态后，切换并激活下一套的配置继续进行任务的执行。

如图4‑7所示是阵列仿真过程的主要流程图，在进过多次循环执行之后一个任务的多个配置包被执行，在配置包信息执行结束后，整个任务的执行也就结束了

如图4‑8所示，从软件代码上看，主要是ClockDrive的类进行上层的统筹，分别执行类的方法，调用其他对象的内部的具体属性方法。在配置信息固化到阵列中之后，启动的注入信号首先激活ClockDrive对象，对象被激活完成之后开始调用类Controller对象执行任务的调度和配置文件的写入，在Controller对象执行结束后，反馈到驱动层，驱动层开始调用PeSimProcess类进行阵列的执行，最后反馈结果，驱动层进行时钟基数，以此类推，知道任务结束。



图4‑8仿真器执行代码跳转示意图

Fig. ‑ Workflow diagram of simulator

## 本章小结

本节讨论了基于混合粒度可重构阵列的仿真器的实现，介绍了仿真器包含的模块以及每一个模块的实现，同时介绍了仿真器搭建的验证平台，最后介绍了从应用到仿真结果的仿真的具体流程。

# 第五章 实验与分析

## 实验目的

本章主要对不同的算法进行实验，以验证混合粒度可重构阵列可以支持控制密集的算法，同时为了验证HGRA结构可以实现较高的计算效率。下文介绍的算法1,2属于控制型的算法，算法2,3,4属于数据密集型的算法，从下文可以看出，HGRA结构发挥了细粒度节点的作用，能支持算法中的控制部分，同时对于数据密集型的算法，本结构也能获得较好的性能表现，达到了设计目标。

## 常见算法映射

### 矩阵乘运算

矩阵乘运算在图像处理和计算机视觉领域的算法中被广泛使用，对矩阵乘运算核心的计算性能优化可以使得很多算法获得性能的提升。

如图3‑19所示，矩阵乘核心代码是由三层循环构成，第一层循环实现对A矩阵行的遍历，第二层循环实现对B矩阵列的遍历，第三层循环实现对所取得的行数据和列数据元素进行遍历。将矩阵乘代码的第三层全部展开到空间，鉴于硬件空间资源的限制，将第二层循环部分展开到空间上，第一层循环和第二层的部分循环实现流水，通过利用空间计算的特性和循环流水的特性，极大提高了矩阵乘运算的计算效率。

在矩阵规模比较大的时候，完全的空间展开需要的硬件资源太多，可重构阵列不能满足这种需求，利用矩阵运算的固有特性，进行矩阵分块，将高阶的矩阵转换成低阶的矩阵进行运算，简化运算步骤。如图5‑1所示，将高阶矩阵的乘转化成多个阵列子任务进行执行，图中是两个16阶方阵的乘法，将每一个方阵分成四个8阶方阵，实现小方阵的矩阵乘。两个16阶方阵的乘法被划分成多个小矩阵的相乘和相加操作。而小矩阵的操作阵列可以完美的支撑，从而实现了高阶矩阵的相乘操作。



图5‑1高阶矩阵乘任务划分

Fig. ‑ task partitioning of matrix multiplition

### sobel边缘检测算法

Sobel算法也称为Sobel滤波，是一种在图像处理和计算机视觉领域上得到广泛应用的图像边缘检测算法，其是由两个3×3的矩阵组成差分算子，也称为卷积核，将卷积核在图像上进行平面卷积操作，可以得出横向和纵向的像素亮度值，最后将两个方向上的亮度值进行处理，从而实现图像边缘的检测。[36]



图5‑2 Sobel卷积因子

Fig.‑ Convolution factor of Sobel algorithem

如图5‑2所示，Sobel算法具体的步骤是：假设A是原始图像像素点组成的矩阵，通过图片矩阵分别和卷积因子做矩阵乘运算并进行图片滑动，完成算法执行过程。

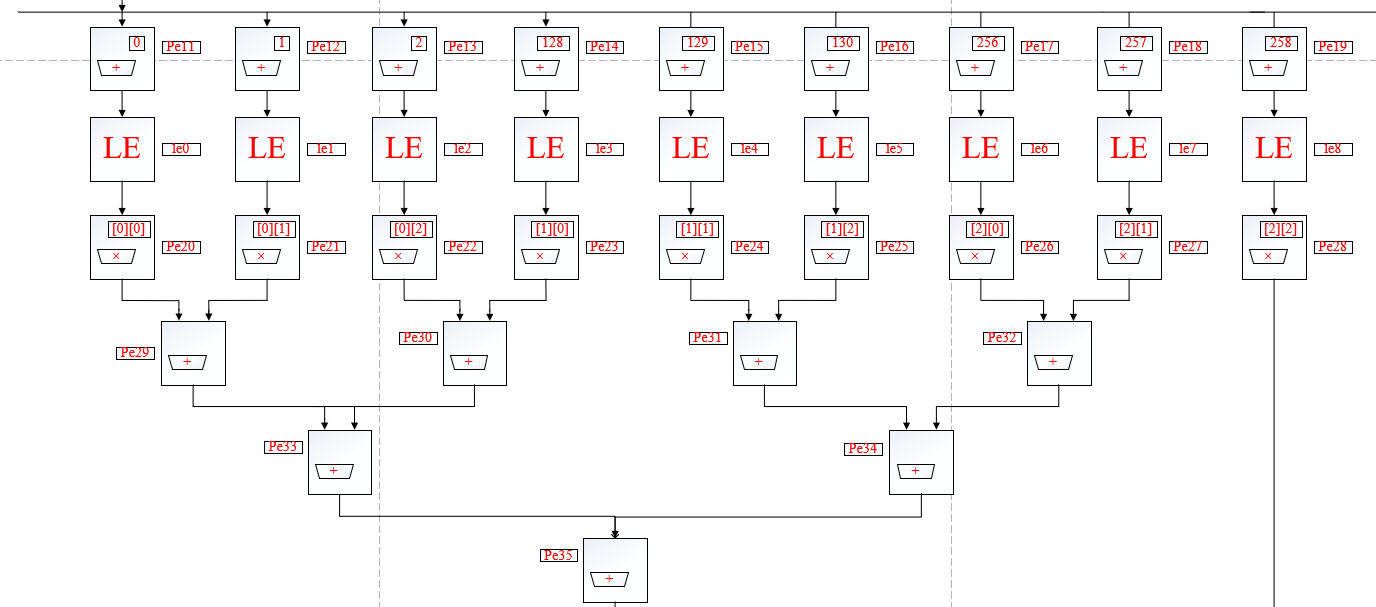


图5‑3 Sobel算法的数据流图

Fig. ‑ Data flow of Sobel algorithem

图5‑3图5‑4所示分别是从sobel算法中抽取的数据流图和控制流图，图5‑3中最上面三行的处理单元进行乘法操作，下面三行进行加法操作，每一行的所有操作结束之后可以进行下一批次的数据操作，实现流水。图5‑4中实现的是循环头的

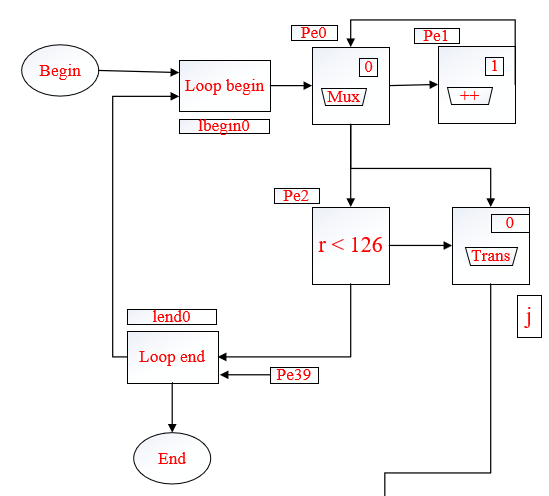


图5‑4 Sobel算法的控制流图

Fig. ‑ Control flow of Sobel algorithem

流图，实现循环步进控制和边界控制。将图中的节点一一映射到阵列的处理单元上就是实现了整个任务的映射，将每一个单元的功能信息和互连信息编写下来，即实现了配置文件的生成。

下面以sobel边缘检测算法为例，介绍算法映射配置的过程，通过分析发现代码包含4层循环，最外面两层是对图像像素进行遍历，最内两层是对卷积核进行遍历，将最内两层循环展开到空间上进行计算，最外两层上实现循环流水，实现计算的最大并行性.

### 图像中值滤波

图像中值滤波算法是进行图像处理的一种算法，其通过将每一个像素点的灰度值设置成领域窗口内所有像素点灰度值的中间值。通过让像素点的值接近真实值，从而消除掉孤立的噪声点。其处理过程是使用二维的滑动窗口对图像进行滑动操作，然后选取窗口内的所有像素点进行排序处理，选取中间的数据作为窗口内某点的像素值，通常二维窗口可以选取3×3或5×5区域。

本实验选取3×3的窗口，图像大小是320×240大小，一次性选取窗口内的9个像素点进行排序，算出中间值，替换窗口内的像素值，继续进行图像滑动，知道图像中的所有像素点都被处理过，任务结束。该算法映射中最主要的步骤是要在可重构阵列上实现9个元素的排序，如图5‑5所示是一种可能的实现方式。图中每一行的PE实现最大元素的选取，最开始9个元素输入到阵列中，每一个处理单元向下输出比较运算的较小值，向右输出比较运算的较大值，经过5行阵列的运算，中间值被选取出来，即图中标成灰色处理单元中的数值，将其输出就是9个元素中的中间值。



图5‑5 多元素排序取中值示意图

Fig.‑ Implementation of sort algorithem in HGRA

### GCD最大公约数

GCD算法(Greatest Common Divisor)是求两个正整数最大公约数的算法，也称为欧几里德算法，根据所述其原理是两个整数的最大公约数也是其中较小的数和两数的差的最大公约数，根据这个原理，可以多次作差直到其中一个数字变成零，即可以找出最大公约数。比如数字65和91，其最大公约数也是65和26（91-65）的最大公约数，也是26和39（65-26）的最大公约数，也是26和13（39-26）的最大公约数，也是13和（26-13）的最大公约数。由此可以看出，GCD算法是多次迭代的一种算法，其算法中需要动态决定循环的次数，并进行每一次的循环迭代前的判断，是一种控制密集型的算法。如图所示是其实现的一种可能代码。如下图所示是其控制数据流图，将流图中的节点分别映射到粗粒度阵列和细粒度阵列上即可以实现该算法的映射。

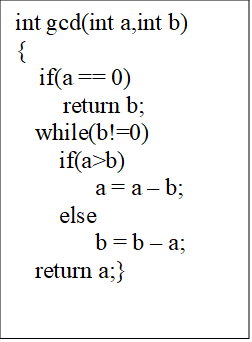


图5‑6 最大公约数算法实现代码

Fig.‑ Algorithm implementation code of GCD



图5‑7最大公约数算法的CDFG图

Fig.‑ CDFG diagram of GCD algorithem

## 仿真结果及分析

本小节通过将计算密集型的算法分别映射HGRA平台上和GPU平台上，获得不同算法在不同计算平台上的仿真性能数据并对性能数据进行分析，其中GPU平台是NVIDIA GeForce MX150，基于CUDA10.0版本进行测试，两种平台的软件工具均是Visual Studio2017。基于GPU的测试数据是原始代码运行的时钟周期数，通过调用CUDA工具包中CLOCK()函数实现，代码运行未调用任何CUDA的优化库。

首先对于矩阵乘运算，通过选取三组不同规模的矩阵进行矩阵乘运算的仿真，其性能数据如图，矩阵的规模分别是32×32,64×64,128×128，GPU平台上开启和矩阵规模同等大小的多线程进行仿真，即针对于32阶方阵，开启32线程进行程序运行，在HGRA平台上充分利用空间资源进行循环空间展开，并进行循环间流水，从图5‑8可以看出，在矩阵乘运算上，HGRA结构获得了较好的性能表现。

图5‑8不同规模矩阵在GPU和HGRA平台上性能数据

Fig. ‑ Performance of different scale matrix multiplication in GPU and HGRA platform

在Sobel边缘检测算法的仿真实验中，选取的图片的像素点规模是300×300，在HGRA上映射方法是卷积核的循环层数空间展开，将这部分的空间展开资源复制一份映射到其他空闲资源上，实现对第二层循环的2倍加速;GPU平台的仿真方案是开启和图像维度相同的线程数即开启298线程进行仿真，如图所示是Sobel检测算法的仿真性能数据，在HGRA上获得1.19倍于GPU的性能优势。

图5‑9 Sobel边缘检测算法在GPU和HGRA上的仿真数据

Fig. ‑ Performance figure of Sobel algorithem in GPU and HGRA platform

对于图像中值滤波的应用，本实验选取的图像大小分别为320×240和480×360,选取的滑动窗口区域大小为3×3，由于在每一个滑动窗口内要进行9个元素的排序，是一种比较规整的数据操作，没有较多的控制代码，适合于混合粒度可重构阵列的执行，通过按照图5‑5所示的将排序程序映射到阵列上，由于一次的排序操作在一个阵列簇上可以完成，本文研究的阵列有四个簇可以使用，在本算法上可以获得四倍的加速比，表所示是基于GPU平台和HGRA平台的仿真数据对比，由于算法中存在不能并行处理的排序过程，所以本阵列和GPU的仿真结果没有太大的差别。

表5‑1图像中值滤波不同平台运行时钟周期数

|  |  |  |  |
| --- | --- | --- | --- |
| 图像规模/平台 | HGRA | GPU | 提升比 |
| 320×240 | 5.56×105 | 5.71×105 | 1.03 |
| 480×360 | 7.22×105 | 6.83×105 | 0.94 |

针对于求最大公约数算法，通过分析可以知道，算法中中较多的分支跳转，同时循环之间存在依赖，所以在算法本质上很难进行并行运算，只能进行代码的串行执行。我们分析一下前文中提到的各种可重构计算的平台，由于Morphsys结构的可重构部分是粗粒度的阵列，其控制部分是通过RISC主核来进行的，所以在执行多分支结构的最大公约数算法的时候要频繁的将分支判断部分交给主核执行，在和主核进行通信的过程中，减缓了计算速度。而本文研究的可重构阵列将细粒度节点糅合到阵列结构中，可以将GCD算法中的分支判断部分交给阵列上的细粒度节点来进行，根据判断的结果激活相应的分支块，通过将该算法的控制数据流图映射到可重构阵列上并进行运行，我们发现混合粒度阵列很好的支持了控制密集型的算法。

## 本章小结

本章通过分析两种类型应用的代码，分别将数据密集型的算法和控制密集型的算法映射在可重构平台上，并通过和GPU平台进行对比，通过仿真运行获得性能数据，分析发现，相比较通用的GPU平台，在采取各种优化技术后HGRA结构不仅具有较高的计算效率，还能只是控制型的算法和应用，具有较广泛的应用前景。

# 第六章 总结和展望

## 论文总结

本文的主要工作：

本论文分析了通用计算平台和专用计算平台在处理计算任务时所面临的问题，通用计算平台灵活性好但是计算效率不高，专用计算平台有较高的计算效率但是电路的设计构建时间成本太大，面向这些问题本文介绍了一种混合粒度可重构阵列的结构设计，通过可重构技术解决了通用计算平台和专用计算平台的缺点，兼顾了电路重构的灵活性和计算效率的优点，同时可重构阵列支持数据流驱动技术，可以灵活的进行静态数据流执行方式和动态数据流执行方式的配置，根据不同的应用选取不同的映射方法，可以实现计算效率的极大提升。最后，针对可重构阵列结构使用软件语言构建出其行为模型，搭建出平台验证测试的整个系统。

取得的主要成果：

可重构系统的研究的主要研究热点是可重构系统是被控制流驱动的，配置信息携带大量的信息，处理单元获得配置信息之后开始构建其余外部处理单元的连接通路，这是一种动态的配置系统。本文最大的突破点就是实现任务内的静态配置，任务级的动态配置。在一个任务执行期间处理单元之间的连接关系已经确定，减少频繁切换配置造成的动态功耗。

采用了数据流驱动技术，在面向图像处理领域中，应用的数据是比较有规律的，使用数据流驱动技术能更充分的利用这种特性，提高计算效率，充分利用空间计算带来的优势。

基于动态数据流驱动技术设计粗粒度处理单元的内部结构，在CPE内部增添特殊结构的输入buffer，在存储单元中进行数据的匹配，保证了计算的正确性。

最后，通过对整个阵列结构进行抽象，构建其行为模型，通过软件语言实现，作为验证的平台。

## 后续工作展望

由于时间有限，可重构系统设计到的范围比较庞大，本文只是基于数据流技术和可重构技术实现了一种混合粒度的阵列，实现了和通用处理器和阵列组成异构系统的仿真实现，而可重构阵列这一研究课题还有较大的空间值得研究人员进行研究，在本文的研究中，发现了阵列的设计过程中有以下值得更深入研究的地方。

首先，针对可重构系统的编译器技术发展缓慢，在任务的分析过程中，任务划分、任务流图导出、任务映射和配置文件生成等都是手动设置的，自动化程度不高，所以编译器的研究要重点关注以下两个方面的问题，第一点编译器要实现能自动从算法或应用中抽取数据流图和控制流图的功能，以方便底层配置人员的配置过程；第二点要支持交叉环境下对任务的自动识别，识别出任务代码中哪些部分可以被阵列加速那一部分不能被阵列加速，提高系统执行任务的完整性和智能化。

其次，阵列处理所面向的计算密集型应用，数据的存取往往会成为性能提升的关键，在数据读取上实现数据预取，从而加快数据流通速度，减少阵列流水等待时间。

# 参 考 文 献

1. Estrin G. Organization of computer systems: the fixed plus variable structure computer[C]//Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference. ACM, 1960: 33-40.
2. Owens J D, Houston M, Luebke D, et al. GPU computing[J]. Proceedings of the IEEE, 2008, 96(5): 879-899.
3. Kuon I, Tessier R, Rose J. FPGA architecture: Survey and challenges[J]. Foundations and Trends® in Electronic Design Automation, 2008, 2(2): 135-253.
4. Backus J. Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs[M]. ACM, 2007.
5. Compton K, Hauck S. Reconfigurable computing: a survey of systems and software[J]. ACM Computing Surveys (csuR), 2002, 34(2): 171-210.
6. Hartenstein R. A decade of reconfigurable computing: a visionary retrospective[C]//Proceedings of the conference on Design, automation and test in Europe. IEEE Press, 2001: 642-649.
7. Estrin G, Bussell B, Turn R, et al. Parallel processing in a restructurable computer system[J]. IEEE Transactions on Electronic Computers, 1963 (6): 747-755.
8. Hauser J R, Wawrzynek J. Garp: A MIPS processor with a reconfigurable coprocessor[C]//Field-Programmable Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on. IEEE, 1997: 12-21.
9. Singh H, Lee M H, Lu G, et al. MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications[J]. IEEE transactions on computers, 2000 (5): 465-481.
10. Baumgarte V, Ehlers G, May F, et al. PACT XPP—A self-reconfigurable data processing architecture[J]. the Journal of Supercomputing, 2003, 26(2): 167-184.
11. Swanson S, Michelson K, Schwerin A, et al. WaveScalar[C]//Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2003: 291.
12. Lam M S, Wilson R P. Limits of control flow on parallelism[C]//ACM SIGARCH Computer Architecture News. ACM, 1992, 20(2): 46-57.
13. Beck M, Johnson R, Pingali K. From control flow to dataflow[J]. Journal of parallel and distributed computing, 1991, 12(2): 118-129.
14. Gurd J R, Kirkham C C, Watson I. The Manchester prototype dataflow computer[J]. Communications of the ACM, 1985, 28(1): 34-52.
15. 李国杰. 一种新的体系结构——数据流计算机[J]. 计算机研究与发展, 1981(11):3-10.
16. 段然, 樊晓桠, 高德远, 等. 可重构计算技术及其发展趋势[D]. , 2004.
17. 姚于斌, 毛志刚. 面向图像处理的可重构协处理器结构研究[J]. 信息技术, 2008, 32(4): 102-104.
18. 陶文卿, 毛志刚, 何卫锋. 面向媒体处理的可重构阵列的结构设计与研究[D]. 上海: 上海交通大学, 2010.
19. 袁开坚. 基于粗粒度可重构处理器的任务映射技术研究[D]. 战略支援部队信息工程大学, 2018.
20. Goldstein S C, Schmit H, Budiu M, et al. PipeRench: A reconfigurable architecture and compiler[J]. Computer, 2000, 33(4): 70-77.
21. Bouwens, Frank, Mladen Berekovic, Andreas Kanstein, and Georgi Gaydadjiev. "Architectural exploration of the ADRES coarse-grained reconfigurable array." In International Workshop on Applied Reconfigurable Computing, pp. 1-13. Springer, Berlin, Heidelberg, 2007.
22. Mei B, Vernalde S, Verkest D, et al. Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling[J]. IEE Proceedings-Computers and Digital Techniques, 2003, 150(5): 255.
23. Hartenstein R. A decade of reconfigurable computing: a visionary retrospective[C]//Proceedings of the conference on Design, automation and test in Europe. IEEE Press, 2001: 642-649.
24. Mirsky E, DeHon A. MATRIX: a reconfigurable computing architecture with configurable instruction distribution and deployable resources[C]//FCCM. 1996, 96: 17-19.
25. Ledzius R C, Flemmons J L, Maturo L R. Reconfigurable computing system and method and apparatus employing same: U.S. Patent 6,539,438[P]. 2003-3-25.
26. Wong D, Phillips C E, Cooke L H. Integrated processor and programmable data path chip for reconfigurable computing: U.S. Patent 6,282,627[P]. 2001-8-28.
27. Fine-and coarse-grain reconfigurable computing[M]. New York, NY, USA: Springer, 2007.
28. Schmit H H, Cadambi S, Moe M, et al. Pipeline reconfigurable fpgas[J]. Journal of VLSI signal processing systems for signal, image and video technology, 2000, 24(2-3): 129-146.
29. DeHon A, Wawrzynek J. Reconfigurable computing: what, why, and implications for design automation[C]//Proceedings of the 36th annual ACM/IEEE Design Automation Conference. ACM, 1999: 610-615.
30. McWhirter J G. Recursive least-squares minimization using a systolic array[C]//Real-Time Signal Processing VI. International Society for Optics and Photonics, 1983, 431: 105-114.
31. 李云峰. 现代计算机仿真技术的研究与发展[J]. 计算技术与自动化, 2002, 21(4):75-78.
32. 肖田元，张燕云，陈加栋. 系统仿真导论 : Introduction to system simulation[M].清华大学出版社，2010.
33. Rao D M, Wilsey P A. Dynamic component substitution in web-based simulation[C]//Proceedings of the 32nd conference on Winter simulation. Society for Computer Simulation International, 2000: 1840-1848.
34. Rosenfeld P, Cooper-Balis E, Jacob B. DRAMSim2: A cycle accurate memory system simulator[J]. IEEE Computer Architecture Letters, 2011, 10(1): 16-19.
35. Kim Y, Park I, Choi K, et al. Power-conscious configuration cache structure and code mapping for coarse-grained reconfigurable architecture[C]//Proceedings of the 2006 international symposium on Low power electronics and design. ACM, 2006: 310-315.
36. Gao W, Zhang X, Yang L, et al. An improved Sobel edge detection[C]//Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on. IEEE, 2010, 5: 67-71.
37. Wang J H, Lin L D. Improved median filter using minmax algorithm for image processing[J]. Electronics Letters, 1997, 33(16): 1362-1363.

# 致 谢

# 攻读硕士学位期间已发表或录用的论文

[1] 俞磊，景乃锋，王琴. “一种基于数据流驱动的混合粒度可重构阵列架构”. 微电子学与计算机（已录用）