tsinghua-name-bachelor.pdf

综 合 论 文 训 练

题目：面向 FPGA 的软硬件结合异步状态机接口设计与实现

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：蒋嘉铭

指导教师：向　勇　副研究员

2024 年 6 月 12 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名：_____ 导师签名：_____ 日　期：_____

中文摘要

随着科学技术的不断发展，产生了各种不同架构类型的计算器件，如CPU,GPU,DPU,FPGA 等等。这些计算器件的架构和擅长的计算领域各不相同, 其接口的调用协议也有很大差距, 因此产生了异构计算这一研究课题, 其中一个重要课题是 CPU 和 FPGA 之间的异构计算。二者在时序, 数据协议等方面有较大差别, 通过实现系统调用级别的调用接口, 可以将 FPGA 作为 CPU 的外设, 提高计算效率, 降低编程难度。

本文主要讨论 CPU 通过系统调用方式对 FPGA 进行调用, 从而实现异构计算的方法。通过一套在 AXI-DMA 的数据交互接口和协议, 使得 CPU 和 FPGA 之间能进行准确高效的数据交互, 从而命令 FPGA 完成指定计算任务。另外, 这一通用数据协议可以减轻软硬件开发人员的负担, 只需按照通用接口编写程序即可, 提高了通用性和可移植性。

关键词：FPGA；异构计算；DMA；Petalinux

ABSTRACT

With the development of science and technology, many computing devices with different architectures are developed, such as CPU, GPU, DPU, FPGA, etc. These devices' architecture and major occupation vary, therefore heterogeneous computing becomes a new academic topic, among which the combination of CPU and FPGA is very important. Their features of timing and data protocol differ. By implementing a call interface, FPGA could be considered as an external device of CPU, therefore accelerates computation and simplifies programming.

This essay mainly discuss how CPU call FPGA via system calls to achieve the goal of heterogeneous computing. By using a data exchange interface and protocol over AXI-DMA , CPU and FPGA can exchange data accurately and precisely, therefore CPU could instruct FPGA to perform certain computation task. Moreover, this generalized protocol contributes to simplify the development for both software and hardware developers, as it is only required to write programs corresponding to the given interface, therefore improves the usability and portability of the system.

Keywords: FPGA; heterogeneous computing; DMA; Petalinux

目　　录

主要符号表

FPGA	现场可编程门阵列 (Field-Programmable Gate Array)
CPU	中央处理器 (Central Processing Unit)
GPU	图形处理器 (Graphics Processing Unit)
Petalinux	AMD 公司推出的专门用于硬件的 Linux 发行版
DMA	直接内存访问 (Direct Memory Access)
AXI	高性能扩展总线接口 (Advanced eXtensible Interface)
SRAM	静态随机存取存储器 (Static Random-Access Memory)
GPIO	通用输入/输出端口 (General-purpose input/output)

第 1 章　引言

1.1　研究背景

上个世纪以来, 集成电路技术飞速发展, 为了适应不同性质的计算任务, 产生了各种不同的计算器件。CPU 是功能强大的通用处理器, FPGA 擅长解决重复度高, 数据量大的简单任务, GPU 擅长图形处理和人工智能计算, DPU 擅长数据处理, 还有各类高度分化的嵌入式芯片擅长解决领域内专门的计算问题。但是各种计算设备之间的数据协议和通信协议等多有不同, 如何能够综合多种设备的优点, 取长补短协同作用, 成为了一个重要课题, 异构计算应运而生[1]。

现有的异构计算研究成果主要集中在 CPU 和 GPU, 例如 OpenGL 库, 对于 CPU (以下通称软件) 和 FPGA (以下通称硬件) 的异构计算研究则较少, 缺少类似的通用数据接口。FPGA 虽然效率较高, 但是难以处理动态复杂的计算任务, 而 CPU 编程则相当灵活, 二者协同能够极大提高计算效率, 同时可以完成只在 FPGA 难以编程或运行的计算任务, 例如包含大量不确定循环的程序, 需要大量浮点数运算的程序等。但是, FPGA 对时序要求非常严格, 接口的数据协议复杂, 难以调试, 硬件编程门槛高等问题限制了 CPU 和 FPGA 异构计算框架平台的发展, 这也使得其发展水平落后于 CPU 和 GPU 的异构计算[2]。

1.2　研究现状

目前对于 FPGA 的利用主要集中在硬件加速和嵌入式系统上。在硬件加速应用中, 往往是单独使用 FPGA 芯片处理大量数据或进行 CNN 计算加速, 在这些应用中, FPGA 往往是只有数据处理模块和计算模块, 通过优化流水线, 状态机等结构, 力图实现效率最大化; 而嵌入式系统则更加重视灵活性, 相比于传统的嵌入式芯片, FPGA 可以重新编程, 从而可以适应不同环境下的不同应用需求, 提高了整个嵌入式系统的通用性和灵活性。另外, FPGA 还可以用于芯片开发的 IC 验证, 借助二者相似的结构, 生成出高度相似的芯片结构, 方便开发者验证生成的芯片的架构[3]。

CPU 和 FPGA 的异构计算也是当今研究的前沿领域。在这一异构系统中, CPU 主要负责控制, 交互, 异常处理等部分, 而 FPGA 则主要负责接收 CPU 的数据和信号, 完成计算任务。目前较为主流的框架是 OpenCL, 这一框架得到了主要硬件厂商的支持, 从而可以访问一些通常情况下难以访问的接口。这一架构提供了一个与设

备架构类型无关的通用语言模型, 从而实现各类软硬件接口的调用, 并且可以完成跨平台, 跨设备的分布式计算要求。但是各种计算器件的编程方式不尽相同, 通用语言并不能充分发挥高性能计算器件的计算能力, 且其语法比较复杂, 不能满足一些特化的需求, 因此在工业界受到的关注十分有限。

另外为了完成大量数据的异构计算, 也产生了多种软硬件交互手段。传统上向硬件写入固定数据依靠的是各种数据传输线路或者网络手段, 这些技术无法满足低延迟, 高带宽, 大数据量的软硬件交互需求。AXI 总线技术于 2003 年由 arm 公司提出, 是一种高带宽低延迟稳定性强的数据传输方案, 并且允许不对齐传输和突发传输, 灵活度很高。AXI 总线支持各种内存外设, 包括 GPIO, DMA, SRAM, BRAM 等, 允许在不同的设备上使用通用接口进行交互。信道之间相互隔离, 可以自由开关, 有助于降低整个系统的功耗[4]。

AMD 公司推出的 sdk 和 vitis 软件, 提供了在硬件上编写裸机程序的高级语言方案。这些软件附带了专门的库和接口, 例如各类驱动, 从而能够在软件上编写和硬件交互的程序, 利用 JTAG 烧写到开发板上进行实时的测试和调试。但是每次执行不同的任务, 都要重新烧写电路板, 开发周期长, 效率低, 因此只能进行一些简单的计算任务, 难以进行复杂的调度。

为了能够在 FPGA 上执行各种复杂任务, 需要功能强大, 扩展性强的操作系统。AMD 公司推出的 Petalinux 操作系统, 采用 Linux 内核, 提供了一整套在各类架构 CPU 上运行的工具链, 可以编译各种内核模块和应用程序。Petalinux 支持多种 FPGA 开发板, 装载 Vivado 生成的硬件描述, 自动生成硬件对应的设备树, 使得软硬件协同效率大幅提高。作为一个完善成熟的操作系统, 其存在特权级机制, 实现了用户态和内核态的隔离; 可以对各类任务进行自动调度, 无需开发者自行进行同步调度等工作; 支持各种文件系统, 利用 SD 卡等外设, 可以提前在文件系统中加载需要的文件, 无需利用数据线或网络进行传播, 提高了开发板的易用性和可靠性[5]。

可以看出, 在 FPGA 的实际应用中, 现有的架构已经可以充分发挥 FPGA 的计算性能和可扩展性, 但是缺少和软件端实时, 高效的互动手段。进行异构计算的开发者需要对软件和硬件都有充分的掌握, 才能写出合理的程序。并且在现有框架下, 软硬件开发者的协同耦合度有限, 开发效率受到影响。

1.3 研究内容及贡献

本研究从软硬件协同设计接口出发, 利用同时具有 arm CPU 核心和 FPGA 的 zedboard 进行开发, 成功实现了在 zedboard 上进行异步收发计算的接口, 实现了系统级的 CPU 对 FPGA 的实时控制, 并且能够异步向 FPGA 发送命令来执行计算任务, 计算结果正确。

本研究主要分为两部分: 第一部分是对于 Petalinux 系统的配置, 包括添加自定义系统调用, 添加内核驱动模块, 自定义用户态程序用于测试等。第二部分是对于 Vivado 硬件设计的配置, 包括 AXI-DMA 共享内存的设计, 对 IP 核的设计, 以及利用 vitis 编写裸机程序的设计等。

本研究的成果可以用于进行 CPU 和 FPGA 之间异步协同开发, 在项目的基础上修改乘法 IP 核, 即可自定义硬件部分的计算内容; 在软件端可以自由更改系统调用, 来完成对 FPGA 数据的更进一步的操作和交互等。

1.4 文章结构

第二章介绍本项目涉及到的各种硬件, 软件, 系统的相关信息及已有研究成果, 主要包括 Zedboard 开发板, Vivado, Vitis 等开发工具, Petalinux 操作系统等。

第三章具体介绍系统的整体架构, 分为 PL(硬件逻辑), PS(软件系统) 和 linux 内核三部分, 具体介绍彼此之间交互方式和原理。

第四章介绍系统各个部分的实现细节, 包括系统调用, 内核模块, 硬件 IP 核及其布线, 软件测试程序等部分。

第五章展示实验结果, 体现通用数据接口的正确性和有效性, 展示其在海明窗乘法计算的效率。

第六章简单总结了本文工作, 并指出了未来可能的若干研究方向。

第 2 章　工具介绍

2.1　Zedboard 开发板

Zedboard 是一款由 avnet 发行的 Zynq-7000 型开发板, 支持 usb-uart 串口, HDMI 输出, JTAG/SD/闪存三种启动方式, 作为一款低成本开发板各项功能完善, 便于使用。另外 Xilinx 公司提供了对 zedboard 各项支持, 例如对 Zedboard 开发板的开发套装, 预装的 Zynq7 IP 核, Petalinux 支持的 bsp 等等, 非常适合进行软硬件协同的开发[6]。

Zedboard 可以分为两部分: PS(系统部分) 和 PL (逻辑部分)。二者独立供电, 因而可以独立完成计算任务。开发板的硬件部分可以视为典型的 FPGA 模块, 而软件部分依靠板载一枚双核 ARM Cortex-A9 芯片, 二者之间通过 AXI 技术实现了高效的数据交互。通常情况下由软件通过 AXI 通道向硬件发送数据和信号, 逻辑端进行高性能运算, 从而实现异构计算的目的。另外, Zedboard 也支持 microblaze 软核, 作为直接建立在 FPGA 上的处理器, 具有很强的灵活性; 另外使用的 RISC 指令集也较为简单, 方便进行流水设计, 提高计算效率。

在本次实验中, 通过 JTAG 方式下载程序到开发板, 并且通过串口与其进行系统级的会话。这样做的好处是流程较为简单, 可以在线进行调试, 无需反复插拔 SD 卡。当系统镜像确定之后, 也可以使用 SD 卡启动, 这样可以省去 JTAG 烧写的时间, 并且更容易在不同开发板之间移植这一系统设计。

2.2　Vivado/Vitis 开发套件

Vivado 是功能强大的硬件设计软件, 可以进行硬件程序编写, 硬件布线, IP 核编写, 仿真, 时序分析等。Vivado 提供了多种多样的功能完善的 IP 核, 硬件开发者无需研究这些常用设备的配置, 时序等问题, 只需按照需要配置相应的参数, 在自动布线的辅助下连接线路即可。在 Vivado 设计中, Zynq7000+ 被包装成一个处理器系统, 提供若干 AXI 接口, 可以直接和硬件对应的接口连线, 设计十分方便。

Vitis 是在 Vivado2019。2 之后版本引进的新型软件开发平台, 代替原有的 sdk 平台。Vitis 更接近于现代的集成开发环境, 支持对软件程序的各种操作。在 Vitis 中, 可以用硬件描述文件 (.xsa) 建立平台, 利用 Xilinx 提供的库进行交互, 编写裸机程序。同时 Vitis 还提供了 gdb 调试程序, 串口工具等, 可以实现高效的开发验证。

在开发周期中, 往往是先使用 vitis 编写轻量级的程序验证硬件的功能符合预期, 之后转移到 petalinux 编写驱动程序或者系统调用。

2.3 Petalinux 操作系统

为了实现多种多样的嵌入式设计, 完成各种不同的计算任务, 实现任务的灵活调度, 特权级控制, 中断异常处理等功能, 一个功能完整的操作系统是必不可少的。Petalinux 操作系统专门为硬件系统设计, 是在传统嵌入式系统工具 yocto 上改造产生的。Petalinux 支持使用 Vivado 生成的 xsa 文件配置硬件描述, 可以自动将硬件描述转化为设备树上的节点, 并且提供常见设备的驱动程序。由于是基于 linux 内核, 开发人员可以很方便地添加系统调用或者内核驱动, 来适应不同的设备需要。另外 Petalinux 提供了常见开发板的板级支持包 (bsp), 起到了隔离软硬件的作用, 将复杂的硬件设备以驱动的形式暴露给软件, 这样开发者就可以在 bsp 的基础上进行开发, 减少了系统配置等工作的工作量。Petalinux 的全部组件都是开源代码, 用户可以根据实际需要进行更改, 开源社区对其的支持也较多。

2.4 Zedboard 上已有的软硬件交互手段

在 Zynq7000 芯片的设计中, 已经配置了对于 AXI4 协议的支持。AXI4 分为三大类: 可以支持猝发的 AXI 总线, 可以和 CPU 传输单个数据的 AXI-Lite, 以及大规模传字节流的 AXI-stream。在 AXI 总线上, 存在着不同类型的硬件设备, 采用不同的硬件协议, 以满足不同的需求场景。例如, 当进行视频输出时, 需要大量无需地址映射的字节流, 可以采用 AXI-stream 协议直接读写 VGA 端口; 当需要传输中断信号或使能信号时, 可以使用 AXI-Lite 来传递简单的信号, 这也正是 AXI-DMA 上 S_AXI_LITE 起到的作用: 利用这一个字节进行使能和信号传递, 效率高, 时延较小。

另外, 不同类型的总线设备, 其时序, 时钟, 使能方式, 字节长短, 地址描述等信息可能都不尽相同, 因此需要一个统一的总线适配器, 即 AXI-Interconnect 连接器。不同的总线设备, 可以任选主从身份与 AXI interconnect 连接。这些内置的 IP 设备会自动根据主从地位调整交互逻辑, 以达到协同作用的目的。同时这一连接器还支持优先级排序, 仲裁, 并发, 乱序执行等功能, 效果非常强大。

另外, PL 端的 BRAM 也可以供 PS 使用。PS 可以通过 AXI 总线访问 BRAM, 只不过这种做法和 GPIO 一样, 传输的数据量比较小, 因此在本项目中, 主要使用

AXI-DMA 进行试验, 但是同时可以利用 GPIO 和 BRAM 进行极少量控制信息的收发。

第 3 章　系统架构

3.1　逻辑端系统架构

图 3.1　逻辑端连线结构图

PL 端主要由 Zynq 处理器的硬件抽象, AXI-DMA 数据控制系统和计算器件三部分组成。硬件视角中的 Zynq 处理器是一个具有多种接口的 IP 核, 可以暴露各种控制信号, 数据 IO 端口, 时钟设置, 中断设置等接口给硬件端其余的 IP 核, 其内部行为对 FPGA 完全透明。为了实现多种数据接口, 需要开启 Zynq 上 GPIO, DMA 相关接口, 并配置相应的中断设置; 为了和上位机交互, 需要开启 uart 串口。

处理器重置系统 (Processor System Reset) 是为了辅助 Zynq 处理器出现的 ip 核。它可以配置协调时钟周期, 对处理器将要接收到的信号进行过滤, 避免不稳定信号或者脉冲波动等影响处理器的正常运行。

AXI 数据总线是数据传输的核心, 包括了 AXI-DMA 内存直接访问组件和两个 AXI Interconnect 的多路控制器。

AXI 总线是一种高性能高带宽的总线系统, 支持猝发传输和非对称字节传输。该总线的一个显著特点是, 各个通道直接完全隔离, 每个通道都有独立的握手信号和控制信号等。AXI 共有五个通道: 读地址通道, 写地址通道, 读数据通道, 写数据

通道, 写响应通道。通常的操作流程是, 访问读地址通道读取地址, 然后通过读数据通道读取对应的数据; 写入数据同理。AXI 总线支持多个主从设备乱序操作, 这得益于其内部复杂的控制系统。

DMA 允许双向高速内存访问, 相较于 GPIO, BRAM 等方式, 访问速度更快, 稳定度更好, 扩展性更强。DMA 同时支持 S2MM 和 MM2S 两个方向, 在这里 S 代表 Stream, 即数据流; MM 代表 Memory Map, 因为 S2MM 和 MM2S 分别代表从数据流写入内存和从内存写入数据流。和其他的总线上的设备一样, DMA 也提供主从两种接口, 在接口上, 主从和读写两两组合提供四种接口, 可以完成双向的收发。

Vivado 预装的 AXI Direct Memory (通称 AXIDMA) 即为适配 AXI 总线的 DMA 设备, 其上提供了若干和 AXI 总线控制的相关寄存器, 比较重要的有 S_AXI_Lite, M_AXI_MM2S,M_AXI_S2MM, S_AXI_MM2S, S_AXI_SS2M 等端口。根据上面介绍的 DMA 性质, 不难分析出这些端口的数据在总线和 DMA 之间, DMA 内部的数据流向。另外, 其上还有两个方向各自的中断寄存器, 通过 concat 模块合并之后连接到处理器上即可支持中断操作。

由于 AXI 总线和 DMA 设备具体实现细节较多, 因此 Vivado 提供了 AXI Interconnect (以下通称连接器)。这个连接器可以连接多个主从设备, 使他们可以同时访问总线以及其上的设备。连接器可以支持不同种类的存储设备, 也能支持并发读写。另外, 连接器还提供仲裁功能, 可以通过设置优先级来控制不同设备访问总线的优先级和次序。对于不同的通道访问请求, 可以并发完成。

在本实验中使用了两个 AXI 连接器, 根据图上编号分为 0 号和 1 号。数据流从处理器出发, 经过 1 号连接器后到达 AXIDMA, 通过 M_AXI_MM2S 接口进入到读通道, 写入 DMA 数据流; 之后数据通过 S_AXI_S2MM 进入写通道, 写入到乘法计算等 IP 核中进行计算。在 IP 核计算完成后, 数据会进入到 0 号连接器中, 最后数据流会流向处理器。

3.2 SDK 代码架构

在 Vitis 上开发的代码, 经过工具链编译之后, 可以直接在硬核上运行。Vitis 提供了一系列基础的 C 标准库的实现, 同时也支持 Xilinx 自主研发的库, 可以访问一些硬件接口。例如, 可以利用 xil_printf() 函数将简单的字符串输出到串口中, 也可以进行 AXI, GPIO, DMA, SRAM 等一系列和硬件交互的简单操作。一般来说, 当硬件程序编写完成, 经过仿真, 时序和上板验证正确无误后, 即可使用 Vitis 操控其上

的硬件设备端口, 进行简单的验证。

在 SDK 中, 各种设备的地址和配置已经被映射到软件程序的内存空间, 用户可以通过和 Vivado 设计中, 各个硬件部分的名称相对应的宏定义来获知设备在软件空间的地址, 从而进行各种读写操作。

当 Vitis 验证无误后, 就可以在 Petalinux 上开发, 改用系统级的操作方式与其进行互动。由于在 Vitis 上没有系统和内核, 无法支持复杂的操作, 只能进行较为简单的裸机程序开发, 难以通过上位机对其进行实时可靠的操作, 因此我们希望将程序移植到操作系统上操作。这样做会牺牲一定的效率, 但是其扩展性和可用性得到了显著的提高。

3.3 系统端代码架构

PS 端操作系统使用 Petalinux 2023.1, 在 Zedboard 上 arm 架构的硬核上运行, 使用 zc702 的板级开发包作为项目基础, 在此基础上, 在内核引入了系统调用, 添加 DMA 驱动的内核模块, 以实现对 DMA 的读写。

PS 端主要由内核补丁, 内核模块和用户态测试程序三部分组成。通过修改内核代码, 可以自定义系统调用, 并且可以在内核中加入调试信息, 来检测系统是否正常运行; 内核模块可以在线修改内核, 在本项目中主要用于使用 axidma 的驱动; 而用户态测试程序则使用这些系统调用, 来实现在用户态的系统调用。

Petalinux 的修改大多采取 recipe 形式, project-spec/meta-user/ 目录下包含了 recipes-{apps,bsp,kernel,modules} 四个用户自定义代码目录, 结构均为一个.bb 配置文件和 files/ 目录, 配置文件会捕获项目中所需的文件, 而 files 中则包含 Makefile 和源代码 (内核则是放置补丁文件) 在使用 petalinux-create 创建项目时, 会自动将对应的目录注册到项目顶层的编译目标中, 之后可以修改 Makefile 和文件来自定义任意需要的项目。

Petalinux 启动后会自动以 root 权限进入 Linux 内核, 此内核没有库, 因此无法动态编写程序, 即使编写了程序也无法在内核中编译。因此用户态程序必须提前在生成系统映像之前编译到根文件系统 (rootfs) 中, 如果使用 SD 卡启动, 也可以将其用 arm 工具链编译得到的架构正确的程序写入到 ext4 分区的文件系统中, 进入内核后同样可以使用。

进入到内核之后, 预先添加的用户态程序会出现在 /usr/bin 中, 而内核模块一般在 /lib/modules。在设备树上的更改会出现在 /sys/firmware/devicetree, 其目录结构和

设备树的结构相同。对于 GPIO 等驱动比较简单的设备, 可以直接通过 echo, cat 等方式读取写入有关驱动文件, 即可和开发板上的硬件进行简单的交互, 例如点亮开发板上 LED 灯等简单应用。

3.4 内核代码架构

Linux 内核主要分为内核代码, 驱动代码, 不同架构的特化代码以及配套的辅助工具, 在本项目中, 只对以上几个部分进行修改。由于内核是 32 位 arm 架构, 因此必须要在 arch/arm 修改代码才能发挥作用。kernel/ 目录下的代码是平台无关的通用内核代码。driver/ 目录下包含了各类驱动,

本项目在 driver/axidma 中引入了 axidma 的驱动程序, 这个驱动程序会在初始化时自动检测 axidma-chrdev 字符设备, 并根据其配置值获取通道的数量, 方向, 最大传输容量等信息。用户在用户态调用 axidma_init 就可以从内核获得这些信息, 并在用户态掌握信道信息。当需要访问 axidma 时, 用户可以将要访问的信道 (信道信息自动包含方向和 DMA 设备), 缓冲区, 数据字节量包装, 之后使用 ioctl 这一系统调用, 即可与 DMA 进行数据交互。

调用可以采用单向和双向两种方式, 同时支持阻塞和异步回调; 对于数据量较小的情况, 使用阻塞方式更加简洁直观; 对于数据量较大, 且预计硬件端执行时间较长时, 可以采用异步方式进行传输, 从而提高计算效率。值得注意的是, 由于 FPGA 不能在运行过程中重新编程, 因此其上的 DMA 数量无法随着需求增加。如果确定存在并行执行多任务的需求, 那么在硬件设计时, 就必须提前设置足够多的 AXIDMA 和对应的计算器件, 否则只能在软件端形式上存在并行, 事实上在 FPGA 的 AXI FIFO 中, 仍然处于串行状态, 无法达到并发的效果。

第 4 章　系统的具体实现

4.1　数据接口的实现

由于 FPGA 中没有数据类型之分, 而且 FPGA 难以进行复杂的数据流控制, 因此数据流应当尽可能简单, 且在数据流的设计方案中, 应当由 FPGA 开发者起主导作用, 软件端配合硬件端的需求准备数据。在数据接口中, 所有的数据都应当以比特流发送到 AXI 总线, 利用连接器实现数据的流动。双方开发者应当对各种基本数据类型的编码方式达成共识, 例如使用大端序还是小端序, 浮点数统一采用 IEEE 标准等, 整数应当说明清楚采用的位长和是否是有符号数字。

以定长向量乘法为例, 硬件规定的协议是, 固定接收 2048 个单精度浮点数, 分别对应 a[0],b[0],a[1],b[1]..., 返回值是 1024 个单精度浮点数, 分别位于 c[0],c[2]...c[2046]。在确定了硬件需求之后, 软件端的用户态应用即可利用 C 语言等高级语言将数据按照硬件希望的方式进行排列, 放置在一块连续的缓冲区中, 之后通过 DMA 接口转化成字节流发出; 另外还需要准备另一块大小事先约定好的缓冲区, 来接受硬件通过 DMA 发还的计算结果。

这一数据交互接口形式非常简单, 因此产生了很强的扩展性, 允许软硬件开发者在达成一致的情况下, 增加非数据类数据, 如校验和, 控制信息等。经过上板实际测试, 发现 AXI-DMA 数据接口非常稳定, 数据没有出现错误现象, 因此没有在协议中强制设置校验和。如果使用校验和, 那么程序必须接受完所有输出之后才能开始计算, 对效率影响较大。但是对于非常需要数据稳定性的程序, 可以自行设计校验和。另外, 对于可以流式处理数据的硬件程序 (如向量乘法), 如果等待接收到所有输入后再进行计算, 效率更低; 可以分段进行传输, 使用两个线程分别维护和硬件的输入和输出, 在接受到计算结果后发送下一段程序, 即可实现异步流水计算。

4.2　硬件的具体实现

硬件的大部分内容使用 Vivado 2022.2 中提供的已有的 IP 核, 包括 AXI-DMA, AXI Interconnect, Zynq 7000 Processor 等。硬件的配置基本上保持默认值, 做简单微调即可。例如关闭 AXIDMA 模块的 scatter gather 模式, 这一模式可能会导致大规模数据的传输出现异常; 为了能够在软件端正常驱动, 必须引入中断控制, 因此将两个 intr 接口连接之后接入 Zynq 的 irq 模块, 从而能够在操作系统中捕捉到中断。另

外, 需要在 Zynq 处理器上打开 uart1 串口, 来实现和上位机的互动。

硬件的计算器件需要根据自行需要编写, 直接用 verilog/VHDL 等硬件语言编写或者用 HLS 技术转化得到均可, 然后将其封装成 IP 核放置在 DMA 回路上即可工作。可以编写多个不同的 IP 核来进行选片, 使能, 数据处理清洗等功能, 然后用 AXI interconnect 连接即可。

在硬件程序编写完成后, 需要导出硬件描述等数据供 Petalinux 使用。因此在硬件生成硬件描述, 综合, 生成比特流等任务完成后, 还需要生成硬件描述, 即 xsa 文件 (在较老的版本会生成 hdf 文件, 二者性质相同), 且在生成硬件描述时也需要包含比特流的信息。

4.3 SDK 代码的具体实现

在使用 Petalinux 验证之前, 有必要先使用 Petalinux 的裸机接口进行操作。利用 Vitis 编写的程序, 结构比较简单, 有高度封装的 Xilinx 官方库可供使用。

```
int mulHannmingWindow(u64 * bufin, u64 * bufout)
{
    int Status;
    //send batch data to the multiply_window
    //Beaware the data width
    Status = XAxiDma_SimpleTransfer(
    &AxiDma_mul, (UINTPTR)bufout,
                MAX_PKT_LEN*8, XAXIDMA_DEVICE_TO_DMA);

    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }

    //transfer data from here to multiply_window module
    Status = XAxiDma_SimpleTransfer(
    &AxiDma_mul, (UINTPTR)bufin,
                MAX_PKT_LEN*8, XAXIDMA_DMA_TO_DEVICE);
```

```
    if (Status != XST_SUCCESS) {
        return XST_FAILURE;
    }
    while (
    (XAxiDma_Busy(&AxiDma_mul,XAXIDMA_DEVICE_TO_DMA)) ||
    (XAxiDma_Busy(&AxiDma_mul,XAXIDMA_DMA_TO_DEVICE))) {
            /* Wait */
    }
}
```

上述代码是一个向量乘法的例程。输入数据已经写入到 bufin 中, bufout 是空间足够大的缓冲区。SDK 调用 Xilinx 库中的 XAxiDma_SimpleTransfer 函数, 这一函数可以将一块连续内存对应的字节流发送到 DMA; 在乘法 IP 核完成计算之后, 会自动将数据写回, 这时再次调用 SimpleTransfer 函数即可接收到输出数据。需要注意的是, 该函数是非阻塞函数, 因此需要阻塞等待两条信道信息全部收发完毕后才能继续执行后续指令。值得注意的是, 在 SDK 环境下, Xilinx 库提供了对多种硬件设备的较为简单的地址映射方式, 因此访问硬件设备较为简单, 无需编写复杂的驱动程序。

将上述代码编译通过后, 可以先在 Vitis 中编写代码进行测试。测试通过后, 再进入到 Petalinux 中进行系统级的实现。注意下面中并没有进行中断处理, 因此需要暂时关闭 Zynq 处理器上的 irq 中断接口。也可以使用 XAxiDma_IntrGetIrq 方法来获取中断信息, 进行中断处理程序。在 vitis 开发环境中, 可以直接使用 XAxiDma_SimpleTransfer 方法进行 DMA 传输。之后用 XAxiDma_Busy 方法判断信道是否传输完毕, 当信道都空闲时代表传输完毕, 可以从 DMA 到信道一端的缓冲区中取得计算结果。

4.4 Petalinux 的具体实现

4.4.1 Petalinux 的相关配置

Petalinux 的安装较为简单, 只需要在 AMD 下载页面下载对应的安装包即可安装。同时为了创建项目, 需要一并下载正确的 bsp 文件。由于 Zedboard 使用的是 Zynq7000 处理器, 所以本研究使用了 Petalinux 2023.1 版本, zcu702 bsp 进行开发。

在使用 Petalinux 之前, 首先需要引入工具链对应的环境变量, 可以通过

source $PETALINUX_PATH/settings.sh 实现。首先需要建立一个项目, 使用的命令是 petalinux-create -t project -n {项目名称} -s {bsp 文件}。

然后执行 petalinux-build 进行编译。编译过程需要大量的第三方库, 可以预先在下载中心下载需要的各类依赖的缓存 (sstate-cache), 编译速度会得到显著提高。编译完成之后, 会自动得到可以在开发板上启动的镜像文件, 默认存放在 linux/images 目录下。如果需要 SD 卡启动, 需要将 BOOT.bin, image.ub, boot.scr 复制到 SD 卡的 fat32 分区, 将 rootfs.tar.gz 解压到 ext4 分区。然后将 Zedboard 上的跳帽移动到正确位置 (M06-M02 依次为 3V3,3V3,GND,GND,3V3), 即可运行空白镜像, 进入 Petalinux 内核。

除了使用 SD 卡启动, 使用 JTAG 启动是一种更简单方便的方法。使用 JTAG 启动需要 Xilinx 下载器, 其驱动一般在按照 Petalinux 套件时自动安装。如果没有找到驱动, 可以运行 Vivado 目录下的 data/xicom/cable_drivers/lin64/install_script/install_driver/install_drivers 来安装应用, 或者从互联网下载。

通过 JTAG 启动的命令是 petalinux-boot –jtag {–u-boot/–kernel}, 分别表示以 u-boot 和内核两种方式启动。使用 JTAG 启动时, 上位机通过 JTAG 数据线将镜像文件写入开发板的 QSPI 闪存, 然后处理器根据 p7_init.tcl 等文件指导处理器启动内核。使用 JTAG 启动时, 所有的跳帽都要移动到 GND 模式。通过 JTAG 启动可以在无需反复插拔 SD 卡的情况下, 直接启动映像, 操作比较方便; 可以通过 JTAG 进行调试操作, 使用 SD 卡只能通过串口进行数据交互, 出现异常情况难以调试。

为了节省在开发板上启动的时间, 如果只需要研究软件和系统级别的程序的有效性, 也可以使用 qemu 启动, 命令为 petalinux-boot –qemu {–u-boot/–kernel}。通过 qemu 可以规避 zedboard 上板时耗时数分钟的启动流程, 更快速度进入内核进行调试。但是对于和硬件交互的部分, 由于技术上无法对 FPGA 进行虚拟化, 因此只能上板进行实地测试。

bsp 开发包中, 包含了对于板上的各种设备的基本的配置情况, 如常见的驱动, 设备树等等, 但是无法将硬件工程中的各种硬件包含到项目工程中。因此需要执行 petalinux-config –get-hw-description={xsa 文件所在文件夹} 来导入硬件配置。在导入硬件配置之后, 设备树也会进行对应的更改; 用户可以可以在 project-spec/meta-user/recipes-bsp/device-tree/device-tree/system-user.dtsi 中修改设备描述来自定义硬件行为。

另外, 在新版本的 Petalinux 中, 进入内核的用户登录, 会默认使用 petalinux 这

一普通用户, 没有 root 权限, 进行程序测试比较困难。可以利用 petalinux-config -c rootfs, 选择 serial-autologin-root 即可自动以 root 身份登录内核。

4.4.2 对 Petalinux 内核的修改

为了增加系统调用, 监测内核运行流程, 我们需要获取内核代码。首先从 Petalinux 发布信息中获知 linux-xlnx 对应的 git 版本号, 然后使用 petalinux-devtool modify linux-xlnx 来获取 linux 内核源码, 代码会出现在 components/yocto/workspace/sources/linux-xlnx/linux-xlnx 中。代码修改的方式和 Linux 内核类似, 通过 patch 提交代码的更改内容, 编译器将会利用补丁进行增量构造编译, 这样做补丁的规模很小, 并且容易理解改动的部分。我们可以使用 git diff > kernel.patch 的方式修改内核, 然后在 recipes-kernel/linux/linux-xlnx_%.bbappend 中加入 SRC_URI:append="file://kernel.patch", 使编译器能够找到内核补丁, 并按照内核补丁进行修改。

修改 Linux 内核的主要目的是为了进行系统调用。很多驱动程序需要在内核态运行, 监测内核函数的运行时间和运行状态, 因此必须对内核进行修改。对内核修改的最主要方式, 就是增加系统调用, 给用户态程序提供访问内核态的方法。

Petalinux 支持各种架构的处理器, 但是 Zedboard 开发板上自带的处理器是 32 位 arm 架构, 因此在修改架构相关的代码时, 应当在 arch/arm/ 路径中修改。增加一个系统调用, 需要在内核中注册对应的函数, 声明这一系统调用, 为其分配系统调用编号, 并提供该程序的实现。

注册一个系统调用需要修改 tools/arch/arm/syscalls.tbl, 在文件的最后加入新的系统调用的编号, 系统调用后台函数的名字和系统的函数名, 注册之后系统调用就会被内核识别, 加入到系统调用的列表中。然后在 unistd.h 中加入函数的定义, 通过 __SYSCALL() 宏将系统调用和后台实现进行链接。后台函数可以在 kernel/ 目录下新建文件或在原有的文件后添加调用的具体实现。按照 Linux 内核编写规范, 应当使用 SYSCALL_DEFINEX 宏来定义, 内核的宏会自动生成函数的定义。另外需要在 syscalls.h 中加入函数的定义, asmlinkage 代表从栈而非寄存器上接收参数, 是内核函数的共有特点。long 类型是为了避免 64 位机器上使用 int 导致高 32 位会被注入非法指令的问题。

除了直接修改代码, 也可以通过创建内核模块的方式修改内核。通过 petalinux-create -t modules -n {模块名字} –enable 即可创建一个自定义内核模块。编译完成之后会模块会被放置在 /lib/modules/linux/extra/模块名字.ko, 利用 insmod/rmmod 即可

修改动态修改内核。通过内核模块修改函数, 无需经过繁琐的内核编译环节, 也无需改变内核原有代码, 比较适合测试内核模块的功能。在本项目中, 使用了 xilinx-axidma 这一 axidma 驱动, 通过将 axidma 设备映射到用户空间的内存, 来实现收发双向信道。

为了实现系统调用, 单纯的内核模块并不能满足系统调用的需要, 因此需要将内核模块直接作为内核的一部分, 在内核的编译期就生成驱动, 从而可以在系统调用中引用驱动相关操作, 解放开发者, 也便于在内核监测程序的效率。由于本驱动类似于字符驱动设备, 因此将源码放置在了 driver/axidma 目录下。为了编译这个项目, 需要在 driver/, 的目录中加入 obj-y+=axidma/ 来递归调用 axidma/ 目录下的 Makefile。在 Makefile 中列出所有需要的.o 目标, 设置为 obj-${CONFIG_AXIDMA}+=.o 即可。在 init_module 中加入 printk 语句, 可以发现内核启动时输出对应语句, 使用 modprobe -a 命令也能找到对应的 xilinx-axidma.ko。

4.4.3 设备树的修改

在使用 xilinx-axidma 这一驱动时, 需要对设备树进行修改。设备树是 Linux 中通用的维护硬件设备的描述性文件, 在 bsp 中就提供了开发板上各种外设的基本描述。为了引入用户自定义的设备, 需要在 project-spec/meta-user/recipes-bsp/device-tree/files/system-user.dtsi 中添加对应的代码:

```
&amba_pl {
        axidma_chrdev: axidma_chrdev@0 {
                compatible = "xlnx,axidma-chrdev";
                dmas = <&axi_dma_0 0 &axi_dma_0 1>;
                dma-names = "tx_channel", "rx_channel";
        };
};


&axi_dma_0 {
        clock-names =
        "s_axi_lite_aclk", "m_axi_sg_aclk",
        "m_axi_mm2s_aclk", "m_axi_s2mm_aclk";
        clocks = <&clkc 15>, <&clkc 15>,
```

```
            <&clkc 15>, <&clkc 15>;
            dma-channel@40400000 {
                    xlnx,device-id = <0x0>;
                    dma-channels = <1>;
            };
            dma-channel@40400030 {
                    xlnx,device-id = <0x1>;
                    dma-channels = <1>;
            };
};
```

上述代码的含义是, 在 dts 中已有的 amba_pl 节点下添加 axidma_chrdev 这一字符驱动设备, 并且根据 AXI-DMA 的数量进行配置。axi_dma_0 对应的是设备树中 dma 设备的编号, 后面的 0,1 表示为这个 dma 设备开放两条通道, 分别对应 tx (transmit) 和 rx (receive)。

在此基础上, 还要对 dma 设备进行修改。在设备树的编译逻辑中, 在 system-user。dtsi 中进行的修改会覆盖自动生成的设备树的对应域, 因此只需要在文件中修改 bsp 提供的默认设备树中不能满足驱动要求的部分。首先需要引入对时钟的描述, &clkc 15 对应的是设备树的引用。对于两条 dma 通道, 需要为他们分配不同的通道编号, 这样驱动程序才能区分不同的信道, 而用户也可以根据设备树中的描述来判断 dma 通道号对应的 AXI DMA 片, 来保证与正确的 IP 核进行数据交互。

4.4.4 用户态程序设计

在通常的应用场景下, 应用都是运行在用户态下, 只有系统调用和异常时会进入内核态。因此在用户态程序试图使用在用户态尝试进行函数调用前, 应当根据将要发送的数据性质, 预先分配好将要向 FPGA 进行数据交互的双向缓冲区。之后需要将数据转化成字节流, 这一部分代码的逻辑和 SDK 代码相似。之后通过 call_fpga 这一系统调用进入内核态, 调用驱动代码。调用内核的操作默认是阻塞的, 但是可以通过多线程技术访问不同的 DMA 片来实现异步操作。

由于 FPGA 硬件电路设计的性质, 可用于计算的门的数量是有限的, 无法产生任意多个 DMA 模块和计算器件, 且硬件计算速度往往比软件快很多, 大部分时间用于和 DMA 的 IO, 其中包括读写文件, 内存复制, 中断处理, 陷入内核等多项开销。

因此在设计程序时, 程序员需要自行考虑调用 FPGA 的优化是否值得这些开销。

下面是用户态代码:

```
axidma_dev_t axidma_dev = axidma_init();
input_channel = tx_chans->data[0];
output_channel = rx_chans->data[0];
struct timespec clk;

clock_gettime(CLOCK_REALTIME,&clk);
time_t sts=clk.tv_sec;
int stns=clk.tv_nsec;

axidma_oneway_transfer(dev,input_channel,
input_buf,input_size,true);


clock_gettime(CLOCK_REALTIME,&clk);
time_t mis=clk.tv_sec;
int mins=clk.tv_nsec;

axidma_oneway_transfer(dev,output_channel,
output_buf,output_size,true);

clock_gettime(CLOCK_REALTIME,&clk);
time_t eds=clk.tv_sec;
int edns=clk.tv_nsec;
printf("Time Elapsed: %d,%d,%d Interval %d,%d\n",
stns,mins,edns,mins-stns,edns-mins);
```

其中 axidma_oneway_transfer 即为用户态进行 DMA 交互的核心代码, 其参数含义分别对应使用的 DMA 驱动设备, 对应的通道 (由设备后台维护通道的方向), 用于传输数据的缓冲区和大小, 以及是否采用阻塞模式, 真值对应采用阻塞模式。

如果采用异步模式, 需要为对应的通道设置回调函数。另外, 缓冲区需要提前使用 axidma_malloc 分配, 或者使用 axidma_register_buffer 在驱动设备中注册, 才能正常使用程序功能。但是一块内存空间只要经过上述操作之后, 将一直保持可以与 DMA 交互的功能, 也可以直接在原地进行计算和拷贝等操作, 因而大大减少了内存拷贝的开销。

```
int axidma_oneway_transfer(axidma_dev_t dev,
int channel, void *buf,size_t len, bool wait)
{
    int rc;
    struct axidma_transaction trans;
    unsigned long axidma_cmd;
    dma_channel_t *dma_chan;

    dma_chan = find_channel(dev, channel);
    trans.wait = wait;
    trans.channel_id = channel;
    trans.buf = buf;
    trans.buf_len = len;
    axidma_cmd = dir_to_ioctl(dma_chan->dir);

    rc = ioctl(dev->fd, axidma_cmd, &trans);
    return rc;
}
```

这个函数的实现依赖于 linux 内核中驱动编写常用的函数 ioctl。这一系统调用接受一个文件指针, 一个代表操作类型的整数, 还有一个参数列表。系统会根据操作类型和参数列表调用后台的 _IOC() 来执行对应的文件 IO 操作, 从而实现通过设备树映射的 DMA 设备的读写操作。

4.4.5 使用方法总结

作为本章的总结, 在此梳理用户使用整套 CPU-FPGA 异构计算的流程。

在硬件开发过程中, 需要设计 AXI-DMA 相关的数据环路, 并设计计算 IP 核心, 使 AXI 总线数据流进入计算组件后以同样的形式回流到 DMA 中, 并将 AXI-DMA

的中断端口连接到 Zynq 处理器。最后生成硬件描述文件.xsa 和比特流文件.bit 即可。

在系统配置上, 首先建立 Petalinux 项目, 引入对内核的修改, 并将硬件信息导入到系统中。之后, 需要根据硬件文件对设备树进行修改。在 system.dtsi 中加入 AXI-DMA 字符驱动设备, 之后配置各个 DMA 设备的编号。每个设备应当具有对应的 tx/rx-channel, 对应传输和接收。另外各个信道应当具有不同的编号, 各个 DMA 设备也应当具有不同的编号。这样在编译程序之后, 驱动程序初始化之后就可以自动将这些设备转化为若干条通道。

在软件开发过程中, 需要设计能够在 32 位 ARM 处理器上运行的 C 或 C++ 程序。用户态程序应当通过 petalinux-create 建立用户态程序, 在用户态程序中, 首先要初始化驱动设备, 根据设备树的编号确定用于交互的 DMA 通道, 利用 axidma_oneway_transfer 进行和 DMA 的单向通信, 并根据实际需要使用同步或异步方式。在得到结果后, 可以用同样方式从 DMA 接收计算结果。

第 5 章　实验结果及分析

5.1　正确性测试

5.1.1　硬件的正确性测试

硬件的测试采用传统的硬件测试方式, 使用仿真, 观察时序, 编写自动化测试脚本等手段进行测试。测试硬件的计算功能完毕后, 还需要考虑其接受数据流的行为。测试无误后将其封装成 IP 核即可。另外本项目中使用的 FFT IP 核是由 Vivado 内置的, 因此无需考虑其正确性问题。

5.1.2　SDK 代码的正确性测试

在 Vitis 上编写裸机测试代码完成之后, 可以使用 Vitis 自带的 IDE 功能进行调试和测试。在 Vitis 中切换成 debug 模式, 使用 gdb 单步执行调试; 可以通过直接观察内存数值进行计算, 也可以使用 xil_printf() 语句输出运行结果到串口, 在上位机分析运行结果。

5.1.3　Petalinux 代码正确性测试

由于和 FPGA 交互使用的是字节流, 因此在用户态添加了 Encoder 和 Decoder 两个应用辅助进行用户态字节流的编解码。首先利用编码器将所需的人类可读数据转化成字节流, 然后在执行 dma 回环程序, 接收到字节流, 写入到输出文件中。解码器将输出文件解码, 利用 printf 输出成可读数据。由于单精度浮点数可能会产生计算误差, 因此不能直接比较二进制字节流, 而应当将数据转化为浮点数计算相对误差。经验证, 对于浮点数乘法, 其计算结果准确无误。运行结果如图:

对比向量的前五项, 可以发现结果正确。

a：0.000289, 0.000289, 0.000289, 0.000290, 0.000290

b：0.0289, 0.0289, 0.0289, 0.0290, 0.0290

$a \times b$：0.00000835, 0.00000835, 0.00000835, 0.00000841, 0.00000841

5.2　效率测试

对于效率的计算, 使用的是 1024 个单精度浮点数的向量乘法作为测试模板。在 sdk 测试时, 基准程序为直接用 for 循环测试, 测试程序为上述使用 AXI DMA 的代

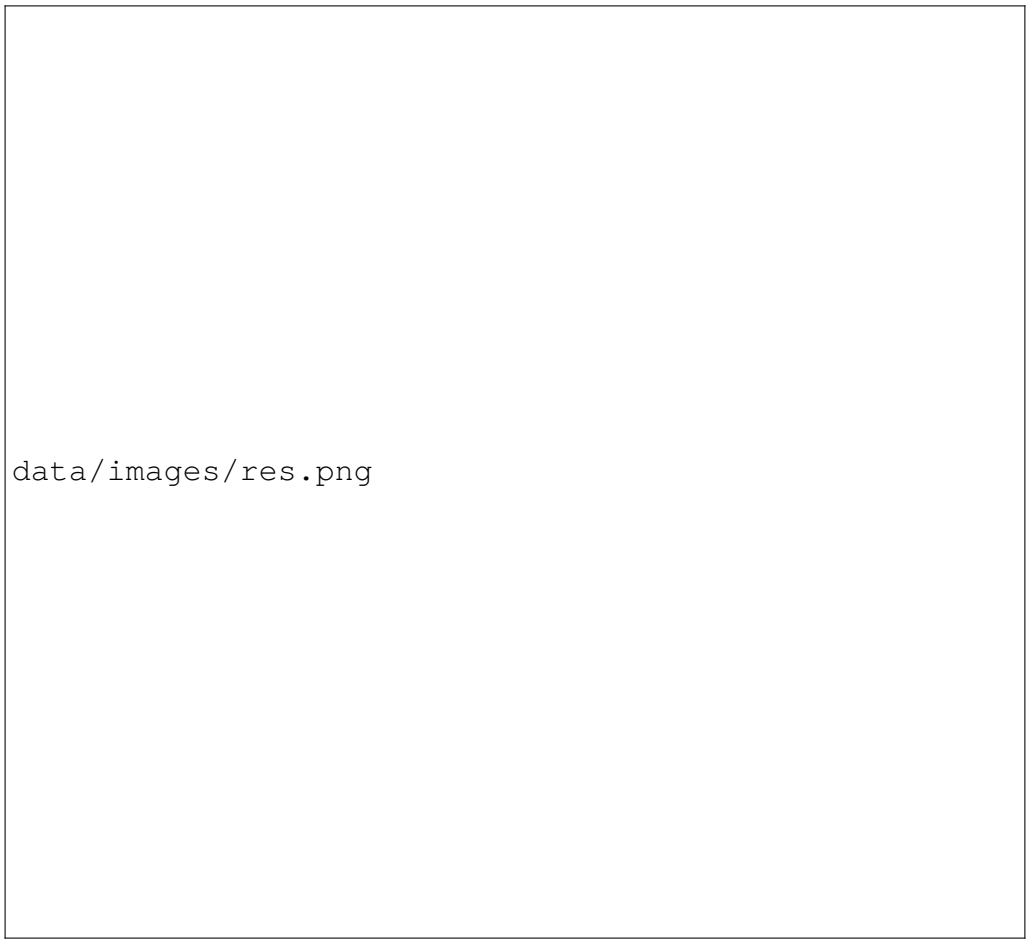

图 5.1　Petalinux 运行结果

码段, 记录时间使用的方法是 xtime_l.h 库中的 XTIME_GetTime 函数, 这一函数会返回程序使用的周期数, 结合处理器的二分频即可得出纳秒精度实际时间。

在 Linux 测试时, 基准程序为在用户态进行循环操作, 测试程序为使用 AXI-DMA 读写数据, 使用 getnstimeofday() 方法得到纳秒精度时间。由于时间差距较大, 因此下列数据仅保留两位有效数字, 用于观察数量级关系。

表 5.1　1024 个单精度浮点数向量乘法时间

测试程序	时间 (ns)
SDK 基准程序	约 710000
SDK DMA 程序	约 47000
Linux 基准程序	约 27000
Linux DMA 程序 (同步)	约 200000

可以看出, 在 SDK 直接编程进行 CPU 运算性能并不好, 这可能是因为 CPU 的主频有限, 且 SDK 生成的代码效率较低, 优化较少, 程序运行需要约 200000 个周期; 在 SDK 上直接运行 DMA 效率却远高于在 Linux 上调用 DMA, 这是因为在操作系统上运行程序会受到操作系统的调度, 可能因为系统调用, 中断等因素产生了一些开销。下面展开分析用户态 DMA 程序的具体时间开销, 尝试寻找优化空间。

表 5.2　Petalinux DMA 运行时间

运行部分	时间 (ns)
向 DMA 发送异步传输数据	20000
向 DMA 发送同步传输数据	50000
硬件端数据流动及计算	100000
从 DMA 接收输出	50000

在用户态的用时测试并不容易, 一些用时需要通过间接计算得到。通过切换向 DMA 传输数据的同步和异步模式, 可以得到函数调用的时间, 数据传输的时间, 硬件计算的时间等信息。

通过分析可以得到, 虽然整体耗时较长, 但是绝大多数时间都可以并行化。可以使用一个协程进行数据发送, 之后即可进行异步处理, 由 FPGA 进行计算即可, 无需占用 CPU 额外资源, 等待协程信号即可。另外, 由于对于同等数据规模, 传输时间是固定的, 因此对于同等数据进行更大量的计算, 如 FFT, 矩阵乘法, 线性规划等, 应当能获得更好的性能结果。

第 6 章　结论

本研究利用 Zedboard 开发板, 分别实现了 SDK 和 Petalinux 与 FPGA 上的数据交互, 验证了程序平台的正确性, 并分析了其性能, 指出了不同交互方式的优缺点和效率方式。总的来说, 在裸机上运行要快于在操作系统上运行, 但是操作系统能提供完善的调度系统和中断机制, 更有利于进行复杂任务。本文提出的基于 AXI-DMA 的字节流传输方案, 具有较高的带宽, 较强的稳定性和较为简单的实现, 可以方便软硬件开发者对数据流展开开发, 从而降低异构计算编程的门槛和代码复杂度。

本文还有一些不足之处, 例如数据传输效率一般等问题, 可能可以通过使用其他形式的 DMA 驱动进行优化, 如 DMA-proxy 等。

针对软硬件异构计算, 在本研究的基础上仍然有许多发展方向, 如基于网络通信的远程异构计算, 多处理器和多 FPGA 架构, 基于中断处理例程自动调度的操作系统等。

插图索引

表格索引

参考文献

[1] Khokhar A A, Prasanna V K, Shaaban M E, et al. Heterogeneous computing: Challenges and opportunities[J]. Computer, 1993, 26(6): 18-27.

[2] Munshi A. The opencl specification[C]//2009 IEEE Hot Chips 21 Symposium (HCS). IEEE, 2009: 1-314.

[3] Zhang X, Ramachandran A, Zhuge C, et al. Machine learning on fpgas to face the iot revolution [C]//2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). IEEE, 2017: 894-901.

[4] Math S S, Manjula R, Manvi S, et al. Data transactions on system-on-chip bus using axi4 protocol[C]//2011 International Conference on Recent Advancements in Electrical, Electronics and Control Engineering. IEEE, 2011: 423-427.

[5] Yeniçeri R, Hüner Y. Hw/sw codesign and implementation of an imu navigation filter on zynq soc with linux[C]//2020 7th International Conference on Electrical and Electronics Engineering (ICEEE). IEEE, 2020: 351-354.

[6] Crockett L H, Elliot R A, Enderwitz M A, et al. The zynq book: embedded processing with the arm cortex-a9 on the xilinx zynq-7000 all programmable soc[M]. Strathclyde Academic Media, 2014.

附录 A　外文资料的书面翻译

单芯片异构计算: 未来会包含自定义逻辑, FPGA 和 GPGPU 吗?

目录

A.1　摘要

摘要

为了提高未来微处理器指数级的性能增长, 提高其能耗效率已经成为了优先级最高的事项。通过和其他非传统的处理器的结合, 如自定义逻辑, FPGA, GPGPU 等, 单芯片的异构计算具有实现更高能耗效率的潜力。尽管这些非传统处理器在提高效率上效果很好, 他们的优点可能随着未来带宽的减少而消失。为了明白在不同的科技约束下不同方式的相对优势, 工作是基于之前的为了支持微处理器的异构微核心的模型建立上。与之前的在简单和复杂处理器之间用性能, 能耗, 空间等作交换所不同的是, 我们的模型必须考虑传统处理器和各种类型的微处理器之间不那么明显的关系。更进一步说, 我们的模型支持推断未来根据国际半导体技术路线图所预测的设计。我们的模型的推测能力建立在当今从不同类型的调优过的微核心 (GPU,FPGA,ASIC) 等测量的性能和数据得出的各个微核心独特的指标。我们的研究成果强化了当今对于微核心的潜力和能力的理解, 并提供了对他们相对优点

的新的视角。

尽管随着处理器技术的不断发展, 晶体管密度一直以指数方式增长, 但由于阈值电压的下降速度有限, 在不同晶体管之间切换的能耗并未减少。因此, 当前性能的主要决定性因素经常是能效, 而非空间不足或频率不足。除了对于功率的担心之外, 片外的带宽同样被认为会对将来设计的增长起主要影响作用。其中, 引脚数量达到了历史性的每年 10% 增长, 和每 18 个月增长一倍的晶体管密度形成了鲜明对比。在当今带宽和功率限制的微核心中, 架构师必须探索更多新的和非传统的方式来提高扩展性。

如今, 非传统的架构的应用提供了提高能量效率的一条可靠的路径。一类解决方案致力于使用高度效率化的自定义的用于计算重要任务的核心, 另一类方案包括多用途的图像处理单元 (GPGPU), 与此同时可编程的 SIMD 引擎提供了加速过的计算性能。近期, 在 FPGA 上运行的应用也达到了相对于传统应用的更高的性能效率。

尽管功率效率在当今理论效率峰值上起到了关键的作用, 有限的投影带宽量同样会使得简单增加能量效率失去作用。对于未来的多核心异构计算开发者来说, 未来的多核处理器的合并的技术方案的选择仍然不明朗, 即使存在一个解决方案也会非常困难。他们必须重视下列问题: 非传统方案的价值值得得到的东西吗? 不管如何带宽会限制这些方案吗? 什么时候 (更昂贵的) 自定义逻辑会比灵活 (但是低效) 的方案更好? 在这个广阔且较少探索的领域, 存在很多的选择和自由度, 因此调查哪个方案值得深入研究是非常必要的。

建立在 Hill 和 Marty 的工作之上, 这篇文章将他们的模型扩展到了在他们的论文中称做 Ucore 的非传统计算核心上, 并研究了扩展性, 功率和带宽对于他们的影响。在我们的模型所使用的应用中, 基于 GPU, FPGA, 自定义逻辑的 Ucore 并用来优化这个应用中可以并行的部分, 与此同时传统的处理器执行其中的串行部分。和他们的工作的精神类似, 我们的模型的目标并非确定架构的具体参数, 而是区分未来的研究和讨论的重要趋势。

为了更高效地使用我们地模型, 许多 Ucore 特定地参数必须提前确定。不像依赖对于简单和复杂核心的性能功率关系的认知的传统模型, 传统核心和 Ucore 在本文的关系并不明显。为了获取我们的 Ucore 的参数, 我们测量了在目前最先进的 CPU,GPU 和 FPGA 上运行调优过的负载的性能和功率。我们同样获得了在综合过的 ASIC 核心上的估计数据。通过使用我们的模型, 我们研究了在国际半导体技术路线图约束的空间, 功率和带宽与不同的假想的异构多核心的轨迹的匹配程度。我

们的结论支持以下观点:

- 尽管存在带宽的限制, Ucore 整体上提供了来自于更高的功效的显著的更好的性能。然而, 在所有情况下, 充分获取 Ucore 的性能需要至少 90% 的并行度。
- 片外的带宽对于不同 Ucore 的相对优势影响最大。尤其是自定义逻辑, 除非应用的计算密度非常大, 否则很快就会达到带宽上限。这也使得其他更灵活的 Ucore, 如 FPGA 和 GPU, 可以更好地在这些样例中保持效率。
- 尽管在不需要关心带宽的时候, 在并行度中等或高的的时候 (90% ∼ 99%), FPGA 和 GPGPU 这样的器件仍然可以和自定义逻辑相竞争。
- 所有的微核心, 尤其是基于自定义逻辑的微核心, 以降低功耗为目标比提高性能更有效。

大纲: 第二部分为读者提供了背景, 第三部分介绍我们微核心的模型。第四部分介绍了获取性能和功率的方法。第五部分提供了基准数据。第六部分提供了在国际半导体技术路线图的标定模型下的投影和关键数据。我们在第七部分提供了结论和未来的研究方向。

A.2 背景

我们对于异构计算的研究从 Hill 和 Marty 对于芯片上多核处理器的分析性建模的基础上眼神, 加入了基于非传统计算范式的 Ucore, 如自定义逻辑, FPGA, GPU 等。图 1 描绘了在我们的研究中的芯片的模型。(a) 中的对称多核处理器类似于如今的商用多核处理器的结构, 包括了典型的配备了私有和公用低级多级缓存的微处理器。

(b) 中的多核处理器包括了一个用于顺序执行的快速处理器, 同时附近的最小化规模的基准核心 (我们称他们为等效基准核心, 在 [11] 中被命名为 BCE) 执行代码的并行部分。

最后, 图 (c) 描述了一个假想的异构多核处理器, 包括一个传统的并行处理器和海量的 Ucore。尽管在这张图片中显示 Ucore 是一个独立的结构, 他们同样也可以被视为是紧密排列连接的处理器, 举例来说是专门的功能单元。因为我们的研究聚焦在当今的测量技术, 我们并没有展示 “动态” 多核处理器模型的结果, 他们理论上可以对一个应用的串行和并行部分都利用所有的资源。由于功耗和带宽不确定, 这超过了我们的模型研究的范围。

A.2.1 回顾阿姆达尔定律

在展示我们的模型对于单芯片异构计算的扩展, 我们首先回顾一下阿姆达尔定律和 Hill&Marty 提出的多核处理器的分析模型。阿姆达尔定律计算的加速比是依据程序中可以被优化的时间 f 和这段时间的加速比 S。总加速比为 $1/f/S+1-f$。

在 Hill&Marty 的对称和非对称多核芯片模型的阿姆达尔定律的扩展性在下方重新打印了一次, 在其中包含了两个新的参数 n 和 r 来分别表达总共可用的计算资源和其中专门用于串行部分的资源, 以 BCE 核心单元为单位进行统计。(所有的加速比都是相对于单个 BCE 核心上的性能)

$$\text{Speedup}_{\text{symmetric}}(f,n,r)=\frac{1}{\frac{1-f}{perf_{seq}(r)}+\frac{f}{(n/r)\times perf_{seq}(r)}}$$

$$\text{Speedup}_{\text{asymmetric}}(f,n,r)=\frac{1}{\frac{1-f}{perf_{seq}(r)}+\frac{f}{perf_{seq}(r)+n-r}}$$

对于所有的芯片模型, 我们都假设并行任务是均匀的, 无限可分的, 完美调度的。在 Hill&Marty 的初始报告中, 他们使用 Pallock 定律作为模型的输入, 可以发现微架构单独的性大致随着使用的晶体管的平方根成正比例。

A.2.2 异构计算中的 U-core

我们研究的焦点是之前没有研究过的非传统计算核心。我们的研究包括三个已经在能效和性能上表现可观的非冯诺依曼架构的计算方式: (1) 自定义逻辑 (2) GPGPU (3) FPGA。特别是自定义逻辑, 通过对特定的问题设计专门的集成电路 (ASIC), 能够实现能量效率最高的计算方式 (通常能达到 100 到 1000 倍效率和性能的提升)。然而, 自定义逻辑很难开发, 并且难以用来重新开发新的应用。有几个提议已经提出了自定义逻辑和通用处理器在同一个处理设备共存的方案。

灵活的可编程加速器, 例如 GPGPU, 已经在目标应用中表现出相对于传统微处理器显著的优势。GPGPU 从 SIMD 向量化中继承了容量, 并且可以利用多线程来隐藏较长的延迟。在灵活加速器中的领域中, 例如 FPGA 的可编程结构展示了在单芯片异构计算的潜力。不像传统的自定义逻辑, FPGA 通过可编程查阅表 (LUT) 实现了灵活性, 并且可以用来实现任意的逻辑电路。作为这个灵活性的交换, 在自定义逻辑和 FPGA 之间, 存在着 10 到 100 倍的空间和功率差距,

A.2.3 相关工作

相当数量的工作已经比较了例如自定义逻辑, GPU, FPGA 和多核心处理器的异构选项。然而, 大多数的比较并没有研究出空间和技术的区别, 也没有研究出和平台有关的限制。相当多的基于 ITRS 的研究之前已经被进行过, 主要聚焦于多处理器芯片的扩展性。逐渐增加的关于单芯片非对称多核心处理器的研究也和本工作密切相关。

除了多核心处理器, 还有很多其他类型的异构设计存在类似的区域专属的处理器, 目的包括数字信号处理, 网络数据负载处理和物理模拟等。值得注意的工作包括集成了含有多个向量协处理器的 PowerPC 流水线的 IBM Cell 处理器, 区域特化的计算模型, 以及正在产生的 CPU-GPU 集成系统。

通过应用特化指令初期 (ASIP), 同样可以实现自定义功能, Tensilica 的 Xtensa 工具可以生成自定义 ISA, 寄存器文件, 功能单元和其他数据路径组件来适应一个应用。在可以重新配置编程领域, PipeRench 和 RaPiD 是早期的可以为特化数据流流水线计算编写重编程结构程序的工具, 与此同时 Garp, Chameleon, PRISC 和 Chimaera 将重配置结构和传统处理器核心相结合。Tartan, CHIMPS 和其他高性能综合工具可以用来生成 FPGA 的高级代码。

很多关于阿姆达尔定律的扩展已经被提出。Gustafon 的模型重新研究了阿姆达尔定律中关于固定输入大小的假设。Moncrieff 等人的模型包括不同并行程度并且标注了不同异构系统的优点。Hill&Marty 从单芯片的多核处理器中推导出观点, 并指出了顺序处理处理器的性能的重要性。Eyerman 等人的多核心模型介绍了关键部分。Woo 和 Lee 考虑了能耗, 功率和其他优劣标准, 如单位瓦特的性能。Cho 和 Melhem 关注最小化能耗和节能属性。

A.3 扩展 Hill 和 Marty 的成果

在这个部分, 我们展示了对 Hill 和 Marty 成果的扩展, 目的是为了考虑基于 Ucore 的单芯片异构多核心设计的功耗和性能。和他们的研究的精神相似, 我们的模型也刻意保持简单, 并且舍弃了一些细节, 如内存层次和互联, 以及 Ucore 的通信方法。

A.3.1 功耗扩展模型

为了扩展我们的模型使之具备能耗控制效果, 我们需要建立花费模型, 并设置一个无论在运行一个负载的串行阶段和并行阶段都不能超过的能量预算。我们首先假设单个 BCE 核心在满状态运行时消耗 1 单位能量, 这包含流失能量和动态能量。我们假设未被使用的核心可以在不增加静态功率时完全关闭。为了建立功耗预算, 我们引入参数 P, P 的定义是相对于单个 BCE 核心的激活功耗。为了建立一个顺序微型处理器相对于单个 BCE 核心的功耗, 一个简单的能量定律被用于捕捉功耗能能量之间的超线性关系: $power_{seq}(perf) = perf^{\alpha}$, 其中 α 根据 [53] 中的结论为 1.75。假设 Pollack 定律成立 ($perf = \sqrt{r}$), 那么顺序处理器的能量耗散可以通过如下方式计算: $power_{seq} = perf^{\alpha} = (\sqrt{r})^{\alpha} = r^{\alpha/2}$。

由于串行核心需要更高的功率, 我们在后面的研究中使用的加速公式是一个修改过的非对称模型, 叫做 “非对称无负载” 模型, 这一模型考虑了功率不足的串行核心在并行部分断电的情况。

$$\text{Speedup}_{\text{asymmetric-offload}}(f, n, r) = \frac{1}{\frac{1-f}{perf_{seq}(r)} + \frac{f}{n-r}}$$

。

随着我们的新参数和假设就绪, 表一的前三行总结了 Hill&Marty 研究中原有的变量 n 和 r 是如何被空间预算 A (用 BCE 单元数衡量) 和功率预算 P 所约束的。对于一个 “被约束的” n 的含义很简单, 就是能够用于全局加速的最大数量的 BCE 资源。举例来说, 如果功率约束的 n 小于空间约束的 a, 那么只有功率约束了能够为整体加速贡献的资源。相似的, 表 1 中串行功率的约束也同样限制了顺序性能所用的资源。

A.3.2 带宽扩展模型

除了功率和空间吗, 有限的片外带宽也是另一个影响多核扩展性的因素。我们假设一个 BCE 核心至少消耗一个应用必要的带宽。对于一个特定的应用, 我们设其为 1 单位。这样做乐观地假设了一个应用或者内核所使用的空间能够装载进芯片上的内存, 这样就无需考虑内存层次等细节。然而我们在后面的第五章中会展示, 芯片上内存能容纳程序的输入的调优过的算法往往都会达到必要带宽 (根据第五章中我们测量的负载的结果) (如果应用的带宽能够用一个函数或者必要带宽的一个系数来衡量, 表 1 中的约束就可以被轻易修改)

对于每一个有自己的带宽要求的负载, 以单位 BCE 的必要带宽的单位描述的 B, 定义了可用的最大的板外带宽。对于非 BCE 处理器的核心, 的估计, 我们可以假设带宽和 BCE 性能成正比, 举例来说, 如果一个核心的性能是 BCE 速度的两倍, 那么他消耗的带宽就是 2。表 1 总结了 n 和 r 如何约束机器支持的最大的必要带宽。

A.3.3 Ucore 模型

功率和带宽已经就位, 现在我们将 Ucore 引入我们的模型。回忆我们之前的定义, Ucore 指的是类似于自定义逻辑, GPU 和 FPGA 的非传统计算模型。为了建模 Ucore, 我们需要从一个假设开始。加我们假设一个 BCE 大小的 Ucore 执行一段代码的并行部分的性能相对于单个 BCE core 是 μ 倍。另外, 一个正在运行的 Ucore 消耗的功率为 φ, 以 BCE 单位功率为基准。μ 和 φ 一同描述了 Ucore 的设计空间。举例来说, 如果一个 Ucore 的参数是 $\mu > 1, \varphi = 1$, 那么它可以被视为一个消耗同等功率水平的加速器。类似的, 如果 $\mu = 1, \varphi < 1$, 那么相当于性能相等但是能耗更低的节能器。对于基于 Ucore 的异构多核处理器, 新的加速公式如下:

$$\text{Speedup}_{\text{heterogeneous}} = \frac{1}{\frac{1-f}{perf_{seq}(r)} + \frac{f}{\mu(n-r)}}$$

我们假设传统的微处理器并不会对并行部分的加速。类似于 Hill&Marty, 我们假设并行部分的效率随着使用的资源数量的增加线性增长。

表 1 展示了变量 n 如何类似地被 Ucore 需求的功率和带宽限制。(注意较小的 φ 是如何降低功率预算 P 的影响的, 同时较高的 μ 会提高带宽消耗)。我们的模型就位之后, 下一部分描述了我们如何获得单芯片的上使用例如 ASIC, FPGA 和 GPU 等 Ucore 异构计算的参数。

致　　谢

感谢向勇老师, 吴竞邦老师, 龚思衡同学以及实验室所有为我提供过帮助的师生; 感谢父母给我生命, 朋友给我陪伴与支持; 还要感谢我的挚友冯子, 彼此支撑渡过人生中的沼泽, 而后策马奔赴刚铎。

声　　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____