清 华 大 学

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　教授

2024 年 6 月 6 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和快速傅里叶变换计算的效率.

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　使用工具和已有成果

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

通常情况下, PL 和 PS 端交互依靠 AXI 进行交互; AXI 总线支持多种数据交互方式, 对于信息量少的控制信号, 可以通过 GPIO 手段, 驱动较为简单; 对于大量的数据传输, 则可以使用 DMA 系统. 通过在 DMA 回环中加入一个负责数据处理或计算的 IP 核, 即可完成数据的输入, 处理和输出, 电路结构十分简单, 也可以减少生成硬件描述和比特流所耗费的时间.

第 3 章　系统架构

3.1　逻辑端系统架构

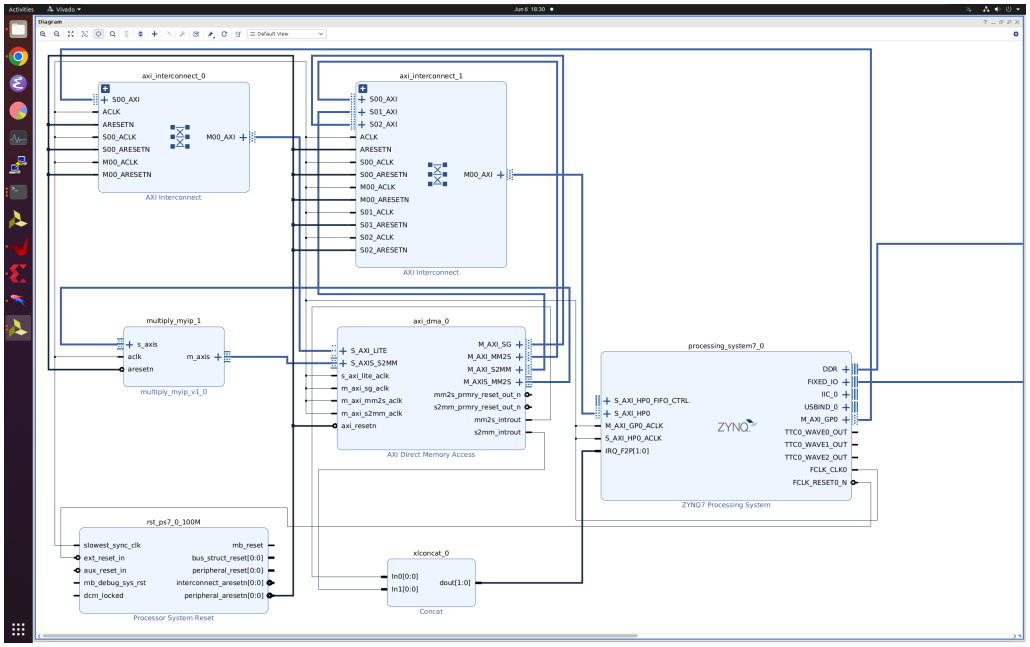

图 3.1　逻辑端连线结构图

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; 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.

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. 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; 如果使用非阻塞实现, 那么回调函数也需要进行一次系统调用, 其开销也较大, 因此在单次调用使用阻塞方式, 但用户可以自行建立线程或协程, 令其进行对 FPGA 的系统调用.

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.

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.

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	640
sdk DMA 程序	132
Linux 基准程序	约 2000
Linux SDK 程序	约 20000

可以看出, 在裸机上运行程序效率较高, 这可能是因为操作系统需要进行任务调度等原因, 导致程序效率略有降低; 在 SDK 上运行, 由于可以高效的直接操作 IO 端口, 因此效率显著提高; 而在系统级别进行 IO, 效率则显著较低. 但是经过观察可以发现, 程序的效率瓶颈主要体现在 IO 中, 系统中断, IO 读写等事件占据了用时的 90% 以上, 这说明在系统级别操作 FPGA, 会受到特权级和操作系统调度等影响, 效率会存在一定损失, 因此希望进行计算量更大的运算, 如 FFT 等.

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, 指出了在不同数据性质和计算需求下, 使用哪种方法交互效率较高. 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

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.

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：_____