华南理工大学

South China University of Technology

本科毕业设计（论文）

基于多级深度学习 IR 框架的可编程通用加速器设计

学　　院	计算机科学与工程
专　　业	计算机科学与技术
学生姓名	陈炫勋
学生学号	201730613014
指导教师	赖晓铮
提交日期	2021 年 5 月 20 日

摘　要

可编程深度学习加速器堆栈，也被称为通用深度学习加速器堆栈，是目前提升深度学习模型的算力的一个重要手段。它既解决了深度学习模型的性能受到硬件资源制约的现状，又比专用的深度学习加速器堆栈更灵活，比起专用的加速器堆栈，可编程加速器堆栈能够适应不同类型的硬件资源。

目前，业界比较成熟的可编程深度学习加速器堆栈是加州伯克利大学的通过 Chisel 实现的 VTA。它已经经受了一些考验，也取得了一些成绩，但是它很重要的一个不足之处在于，它的开发语言 Chisel，是 Scala 的第三方库，Scala 的学习门槛较高，如果有一种更加大众化，学习门槛更低的语言，想必更有利于可编程深度学习加速器堆栈的推广。

这次毕业设计的主要工作是，在参考加州伯克利大学的通过 Chisel 实现的可编程深度学习加速器堆栈 VTA 的基础上，使用我们团队自研的硬件构造语言 PyHCL，开发出我们自己的深度学习加速器堆栈 PyVTA，并且充分发挥出 PyHCL 门槛低的特性，尽量让代码更简单易懂，并且不会对模型的正确率和加速效果造成太大影响。本论文会介绍相关的硬件语言，包括和这次开发工作密切相关的两种硬件构造语言 Chisel 和 PyHCL，以及在硬件构造语言（HCL）之前出现，并且深刻地影响了 HCL 的硬件描述语言（HDL）和高层次综合（HLS）。此外，本文还会说明我们设计这个深度学习加速器堆栈的思路，包括它的硬件架构，流水线设计等等，并且展示我们在实际开发过程中遇到的问题和解决思路。最后，本文展示了在 PyVTA 上运行深度学习模型的效果，来说明我们的工作取得了一定成效，也还有很多值得深入探究的地方。

关键词：可编程深度学习加速器堆栈；硬件构造语言；硬件架构

Abstract

Programable deep Learning acceleration stacks,also called general deep learning acceleration stacks, is currently an important means to improve the computing power of deep learning models. It not only solves the current situation that the performance of the deep learning model is restricted by hardware resources, but also is more flexible than the specified deep learning acceleration stack. Compared with the specified acceleration stack, the programable acceleration stack can adapt to different types of hardware resources.

At present, more mature programable deep learning acceleration stack in industry is the VTA implemented by Chisel at the University of California, Berkeley. It has withstood some tests and achieved some results, but one of its very important shortcomings is that its programming language Chisel is a third-party library of Scala. Scala has a higher learning threshold. If there is a more popular languages with lower learning thresholds, it must be more conducive to the promotion of programable deep learning acceleration stacks.

The main work of this graduation project is to develop our own deep learning acceleration stack PyVTA based on the programable deep learning acceleration stack VTA implemented by Chisel at the University of California, Berkeley, and using the hardware construction language PyHCL developed by our team, and give full play to the low threshold of PyHCL, try to make the code simpler and easier to understand, and will not have much impact on the accuracy and acceleration of the model. This paper will introduce related hardware languages, including two hardware construction languages Chisel and Chisel, which are closely related to our development, and hardware description language (HDL) and high-level Synthesis (HLS), appearing before the hardware construction language (HCL), and having profoundly affected the HCL. In addition, this article will also explain our idea of designing this deep learning acceleration stack, including its hardware architecture, etc., and show the problems and solutions we encountered in the actual development process. Finally, this article shows the effect of running a deep learning model on PyVTA to show that our

work has achieved certain results, and there are still many places worthy of in-depth exploration.

Key words: programable deep learning acceleration stack; Hardware construction language; Hardware architecture

目　录

第一章 绪论

1.1 课题背景意义及研究现状

目前，深度学习领域方兴未艾，比起传统的机器学习，深度学习能有如此迅速的发展，很大一部分原因是神经网络算法有了可以发展起来的基础，而且也比传统的机器学习算法有不可替代的优势。它的出现首先得益于大数据,因为神经网络需要大量的数据，一个简单的分类器也需要大量的对神经网络模型进行训练，才能确保它的正确性；其次就是现在硬件的工艺越来越精细，在硬件方面给神经网络的运算提供的支持越来越多，硬件性能作为神经网络的瓶颈，也已经有了许多提升它的手段，像以前的一些神经网络模型，理论上是可行的，但是实际场景下，经常会出现因为硬件而导致耗时不可接受，而现在也有更好、运算速度更快的硬件资源来承担计算所需的负载，这些神经网络模型的耗时也逐渐变得可以接受。

而神经网络比起传统机器学习算法的主要优势有，首先是它在某些问题上，往往能取得比机器学习更好的效果，例如图像识别领域，绝大多数算法都是通过神经网络来实现的，而且对图像进行分类的结果，各种神经网络模型的表现也优于传统的机器学习算法；还有就是神经网络模型有较好的自学习、自组织和自适应性，例如对图像进行分类，有些神经网络模型不需要人工打标签，而是通过分类器自动对数据进行划分。由于神经网络模型比起传统的机器学习有一些很重要的优势，在目前以及可预见的将来，神经网络在人工智能领域都会有着举足轻重的地位。

然而，这不意味着现在的神经网络没有遇到任何问题。其中一个不可绕过的阻碍就是，之前提到的硬件方面的限制。正如之前所说，尽管现在的硬件设计水平越来越高，

硬件的工艺也越来越好，但是这不意味着硬件处理负载的能力就不再是神经网络计算的瓶颈了。很多时候，等待模型计算出结果依旧是一个很煎熬的过程，其根本原因还是硬件的算力不够。因此，在现有的硬件性能没法在短时间内快速提升的前提下，现在工业上往往会采用深度学习加速器堆栈，来提高硬件利用率，从而最大化整个硬件系统对神经网络算法的支持，提升算力，减少因为硬件的计算耗时。

但是，现在广泛使用的深度学习加速器堆栈，很多都存在这样一个问题：这些深度学习加速器堆栈，往往都只是适用于某种硬件结构，而对于别的硬件结构，它们变得不再适用，因此需要重新编写深度学习加速器堆栈的代码。这显然是一个极其耗神耗力的过程，这也限制了它们在工业界的大规模使用。FPGA，也就是现场可编程逻辑门阵列，在一定程度上缓解了这个问题。FPGA 是可编程电路板，也就是说，在 FPGA 上可以通过编程，实现不同的硬件结构，因此可以让 FPGA 去适应可编程加速器堆栈，而不是让可编程加速器堆栈去适应某个硬件结构。而且 FPGA 的优势也很明显，相对其它硬件来说，它很廉价，就算真的批量编程然后发现出错了，成本也是可以接受的。然而 FPGA 的编程是具有一定的门槛的，FPGA 的学习似乎并不是那么友好，而对 FPGA 进行编程也需要一定的时间成本，所以我们还是希望能够找到一个，能够根据不同的硬件结构，去自动适配这些硬件结构的可编程加速器堆栈。

VTA 深度学习加速器堆栈是 University of California, Berkeley（UCB，加州大学伯克利分校）的研究团队提出来的一个可编程的深度学习加速器堆栈。正如上文提到的，目前的深度学习加速器，从设计方法上来说主要分为两大类，分别为固定功能加速器和可编程加速器（领域专用加速器），在 VTA 之前的深度学习加速器堆栈，大多数都是固定功能加速器，也就是它们在硬件空间上静态布局，适用于某些特定的硬件结构。这

样的设计可能会在一些硬件结构上有良好的性能，却使得硬件资源的重用有了很大的限制。而 VTA 这样的可编程深度学习加速器堆栈，使得硬件资源的重用变为可能，这也是 VTA 设计的初衷。

加州大学伯克利分校通过硬件构造语言 Chisel 实现了 VTA，Chisel 也是加州大学伯克利分校的研究团队自研开发的硬件构造语言，它是用 Scala 实现的。作为一门新兴的硬件构造语言，Chisel 已经接受了业界的考验，加州大学伯克利分校的 RISC-V 指令集架构的项目开发过程中，就是通过 Chisel 实现了微处理器架构，从而在这个微处理器架构上验证 RISC-V。而相比起传统的硬件描述语言，如 Verilog、VHDL，Chisel 对程序员更加友好，同时也具有 Scala 的面向对象和函数式编程的优点，因此用 Chisel 实现 VTA 也是一种很好的选择。然而，Scala 作为函数式编程语言，对开发人员来说还是具有一定的门槛，而且目前 Scala 还不算最主流的开发语言，函数式编程对很多人来说也是一个比较新颖的概念，因此如果一个开发人员想要研究使用 VTA，还需要对 Scala 有一定的了解，然后再去学习用 Scala 实现的硬件构造语言 Chisel，在此基础上再去研究 VTA 的实现。这个过程可能并不是那么让人愉悦，而我们的目标就是，尽量接近“傻瓜式”，也就是尽量降低的开发，研究和使用的门槛，因此我们在参考 Chisel 实现的 VTA 的基础上，采用了另一种硬件构造语言 PyHCL，来实现这个深度学习加速器堆栈，并且起名为 PyVTA。而我们期望的目标是，一方面通过一种门槛更低的语言，使得代码的可读性更强，另一方面是，在开发的过程中，逐步发现 PyHCL 在功能上的一些不足，并且逐步完善 PyHCL，为这门硬件构造语言创造一个良好的生态。

PyHCL，顾名思义，是用 Python 实现的硬件构造语言，它能达到和 Chisel 一样的效果，即通过编译生成 VHDL 代码。相比起 Chisel，PyHCL 一个不容忽视的优势

就是，对开发人员的门槛更低，因此在可预见的未来，PyHCL 会在同类型的硬件构造语言当中，占有不可忽视的地位。

1.2 论文主要工作和结构

本文一共分为五章。

第一章是绪论，介绍了当前在深度学习领域，硬件性能的限制导致神经网络算法的算力无法完全发挥出来，以及相关从业人员为了解决这个问题做出的努力，其中一个已经有了不小成效的就是深度学习加速器堆栈。然后，指出了当前很多深度学习加速器堆栈的局限性，即具有很强的硬件专用性，很难适用于不同的硬件结构。为此，又有一部分人做出了不可忽视的贡献，其中加州伯克利大学以及另外一些研究团队，从通用化的思路出发，实现了深度学习加速器堆栈的通用化，这当中包括它们的成果，通用化的深度学习加速器堆栈 VTA。这是一个功能比较强大，而且已经经受了一定的业界考验的深度学习加速器堆栈，当然它也有一些缺点，其中一个就是，它是用硬件构造语言 Chisel 实现的， 而 Chisel 是 Scala 的第三方库，Scala 门槛较高，因此我们利用自研的硬件构造语言 PyHCL，来实现一个自己的深度学习加速器堆栈 PyVTA。

第二章介绍了在硬件构造语言之前出现的硬件描述语言和高层次综合，它们都是用于描述硬件的语言。

第三章介绍了这次涉及到的两种硬件构造语言，Chisel 和 PyHCL，并简单展示了它们的使用，还提到了 PyHCL 目前存在的一些问题和我自己的一些思考。

第四章是深度学习加速器堆栈 PyVTA 的一些设计思路，包括它在整个编译堆栈中所处的层次，它最底层的硬件架构，以及它的 JIT。

第五章展示了一些测试的结果，并且比较详细地分析了一个比较具有代表性的测试样例，以及最后在 PyVTA 上运行深度学习算法的结果。

第二章 硬件描述语言和高层次综合

2.1 硬件描述语言介绍

想对硬件构造语言（HCL）有一定的了解，首先要对硬件描述语言（HDL）的基本特性有一定的概念性掌握。接下来几节内容，会逐步按顺序介绍硬件描述语言 HDL，高层次综合 HLS，硬件构造语言 HCL。按照这个顺序来介绍它们是有理由的，因为它们每一个的出现都是为了解决前者存在的一些问题。

硬件描述语言描述了硬件，或者更直观地理解，它描述了一个电路，还有这些电路当中电路元件的连接方式，还有它们的输入端口和输出端口等等。理解这一点很重要，这样才不会把硬件描述语言和高级程序语言（C，cpp，Java 等）混淆。尽管在形式上看来，硬件描述语言和高级程序语言很类似，很多硬件描述语言看起来很像 C 语言，但是这两者本质的差别在于，高级程序语言是有“顺序”的，它会按顺序一行一行执行，而硬件描述语言只是告诉了我们，这个电路是这样的，然后我们可能会在输入端口输入高电平或低电平，然后在输出端口得到相应的结果。因此，尽量不要尝试按照调试高级程序语言的方式，例如设置断点，一行一行执行的调试手段，来调试硬件构造语言，还有硬件描述语言也是一样，因为它们并不是顺序执行的，如果把它们当成高级程序语言，通过注释掉一些代码来进行调试，希望找到 bug 所在的话，那么这段代码就没能完全描述这个电路图，这样的调试就很大程度上失去了意义。

还有一个很典型的例子可以说明硬件描述语言和高级程序语言的不同 ,就是 for 循环结构在硬件描述语言中的使用情况。在高级程序语言中，for 循环是被广泛使用的，几乎每一种高级程序语言都支持循环结构 ,而硬件构造语言虽然也允许开发者写 for 循

环结构，但是其实这种做法是不建议的，还是之前的原因，高级程序语言是有顺序的，对于 for 循环结构，是会按顺序执行每一次迭代，直到满足跳出循环结构的条件；而硬件描述语言，如前所述，它是描述了一个电路图，因此对于 for 循环结构，例如 for(int i = 0; i < 10; i++) 这种，硬件描述语言会认为这个循环结构描述了 10 个电路结构，因此就会把相同的电路复制 10 次，这就造成了极大的资源浪费。在实际的使用中，往往使用计数器来代替这样的循环结构，实现对电路资源尽可能的复用。虽然 for 循环结构使得代码看起来更简洁，可能也让开发者的阅读门槛更低，但是硬件描述语言比高级程序语言对性能的要求更为苛刻，因为高级程序语言有更多的优化手段，例如 C 和 cpp 都有 GNU 编译器来对代码进行优化，开发人员可以不那么关注自己写的代码是否会对硬件很不友好，从而很大程度地影响性能，而硬件描述语言直接操作硬件，中间没有太多的优化层，很多优化需要靠开发人员自己手动实现。正因为如此，硬件描述语言不会过度追求代码的简洁，有时候它可能很冗余很繁琐，以此达到对硬件更直接的描述和更高效的利用。

2.2 高层次综合介绍

在硬件构造语言出现之前，高层次综合（HLS）也有一定的使用群体。高层次综合的目标是，把高级编程语言编写的代码，转化为硬件描述语言的代码。而这个转化的流程，主要分为以下几个步骤，首先是分配，需要确定用到哪些硬件资源，例如加法器，乘法器，寄存器，3-8 译码器等等；然后是部署（或称为调度），把算术负载部署到每个时钟周期内，也就是决定每个时钟周期应该执行什么算术操作；最后是绑定，把算术负载绑定到硬件上。目前比较成熟，应用比较广泛的有 Xilinx 官方的 Vivado HLS，AutoESL 公司的 AutoPilot 工具等等。

HLS 的优势在于，它降低了硬件开发的门槛，使得对硬件不那么了解的软件工程师也可以参与到开发当中，因为 HLS 是从更高的层次进行开发的。这里的层次，是指 FPGA 设计当中思考问题的级别，也就是从什么角度去描述想要的功能，可以粗略分为系统级、算法级、RTL 级、门和开关级，在不同的层次等级描述问题，会有不一样的描述方式，例如对 a 和 b 进行异或运算，在门和开关级就是 a, b 是一个异或门的输入，在 RTL 级就是 a xor b，在算法级可以是 a + b，可见越高层次级别，对某个功能的描述就越简单抽象。HLS 的高层次描述，就是像上面那个例子那样，从更高的层级角度描述问题，显然这样的描述对硬件本身的关注度降低，因此降低了开发门槛。而且，通过高层次综合写出来的实现，代码量更少，也更容易维护。

然而，HLS 并没有能够完全取代硬件描述语言，事实上 HLS 的发展并不顺畅，它有很多不容忽视的缺陷，其中最大的一个问题就是，它所做的优化是有限的，因此 HLS 生成的硬件描述语言代码，往往不是最优秀的，很可能会造成硬件的冗余，这种硬件冗余出现的原因往往是因为硬件描述语言代码的编写方式对硬件不友好，就像上文提到的，在硬件描述语言当中使用 for 循环，造成了大量的硬件资源上的重复，而开发人员又没有太多的手段去控制高层次综合如何把代码从高级编程语言转化为硬件描述语言，正如 GNU 编译器的优化也是有限的，把 C/cpp 编译得到的汇编代码也不一定是对处理器最友好的。对 GNU 的优化能力以及优化受到的限制，开发人员已经有了比较深刻的理解，所以往往能在编写高级程序语言的时候就先写出硬件友好的代码，实现 GNU 一些因为各种限制而不能实现的优化工作，但是对于高层次综合，开发人员有时候并没有太多手段来编写硬件友好的代码，从而涵盖无法自动优化的那一部分。这也是被许多开发人员广泛诟病的一点。高层次综合还有一个缺陷就是，它不支持高级编程语言的一

些特性，例如不支持系统调用，递归函数等等，这使得它对高级编程语言的支持有一定限制。

高层次综合虽然有这些缺陷，但是它提供了一个很好的思路，就是在高层次级别去对硬件进行描述，使得开发人员的学习门槛降低，代码的可读性也更强。虽然高层次综合现在还存在这些问题，但是我们可以在这个思路的基础上继续改进，以解放生产力。而要做到这一点，目前有两个大方向，一个是对高层次综合进行进一步的改造，改造的思路就是让高层次综合减少对高级编程语言的限制。之前提到的高层次综合的缺陷，其中一个就是很多高级编程语言的特性无法使用，因为高级编程语言 C/cpp 这些，不是直接面向 FPGA 的，因此对硬件的操作没有那么直接，很多需要靠编译器进行推断，而为了不让编译器做出的这些推断偏离我们的预期，很多高级编程语言的特性就不在高层次综合中使用了，因为这些特性往往会导致编译器的发生不可预见的推断。也就是说，为了功能的正确性，牺牲了开发人员的一部分便利。而现在想要开发人员更加方便，也就是说希望开发人员能够随意使用高级编程语言的所有特性，就好像写一般的高级编程语言代码一样，我们就需要功能更加强大更加激进的编译器，来实现之前可能会导致功能错误或者无法实现的优化。这是一种思路，但是现有的条件下并不容易实现，因为对编译器的改造还不是一个主流的方向。

另外一种做法就是重新造一门在高层次上抽象的语言，规范好语言特性，调整优化到硬件描述语言的路径。走这条思路的尝试也不少，目前应用比较广泛的有 Chisel 和 Spatial，而这次提到的的深度学习加速器堆栈 VTA 正是用硬件构造语言 Chisel 实现的。

2.3 本章小结

本章介绍了在硬件构造语言之前出现的硬件相关的语言，分别是硬件描述语言和高层次综合，它们的出现都解决了前者的一些问题，然而自身又具有一定局限性，因此硬件构造语言应运而生。

第三章 硬件构造语言 Chisel 和 PyHCL

本节会介绍和这次工作相关的两种硬件描述语言 Chisel 和 PyHCL。在这次的开发工作中，我们要参照 Chisel 实现的 VTA，来实现我们自己的 PyVTA，因此我们首先需要对这两种硬件构造语言都有一定的了解。

3.1 PyHCL 简介

正如 Chisel 可以被视为 Scala 的一个第三方库，PyHCL 也是一个 Python 的第三方库，因此它所有的用法，除了要符合 PyHCL 的语法之外，还需要符合 Python 的语法规则。搞清楚这一点，我们才能够在代码出现语法问题的情况下，查清楚究竟是因为对 Python 的不熟悉而导致的语法错误，还是对这个第三方库不熟悉而产生语法错误。而 PyHCL 通过编译得到 Verilog HDL，这当中也是有一条完整的链路的。PyHCL 代码会被 PyHCL 的编译工具编译成 FIRRTL，这是一种 RTL 语言，现在业界已经有很成熟的流程和手段，把 FIRRTL 编译成 Verilog HDL 代码了。也就是说，我们只需要把关注的重心放在 PyHCL 编译成 FIRRTL 这个过程中。PyHCL 在编译这件事上做的工作和 Chisel 是很类似的，Chisel 也是先把 Chisel 实现的代码编译成 FIRRTL，后面 FIRRTL 编译成 Verilog HDL 的工作，Chisel 就不再过多插手了。

3.2 数据类型和使用方法

本小节对这两种语言常用的数据类型和它们的一些使用方法做简单的介绍。

3.2.1 数据类型

Chisel 中，使用最为广泛的有以下数据类型：布尔类型 Bool，有符号整型 SInt，

无符号整型 UInt，向量类型 Vec[T] 和包裹类型 Bundle。与之对应的，PyHCL 也有这样的数据类型，分别表示为 Bool，S，U，Vec 和 Bundle。

3.2.2 字面量

字面量，顾名思义就是数据的值。例如一个 Bool 类型的对象，它会有一些特性，例如它的值，它的数据宽度等等。这里的字面量就是指数据对象的值。假设现在要构造一个 Bool 类型的对象，它的值为 1，可以这样定义：

```
true.B
```

其中 true 表示它的字面量，也就是值为 1，B 表示它的数据类型为 Bool 类型。类似地，构造一个值为 16 的 UInt 类型的对象，可以这样定义：

```
0.U
```

而在 PyHCL 中，要构造上面两个对象，可以这样定义：

```
Bool(true)
U(16)
```

显然，只有布尔类型，有符号类型，无符号类型这几种类型才会有字面量和数据宽度这些特性。

3.2.3 数据宽度

在硬件层面，所有的数据都可以视为若干个比特位的组合，例如 16 用比特位表示就是 10000。而一个数据对象的数据宽度，默认情况下是能表示它的字面量的最小宽度。就比如说，字面量为 16 的无符号整数类型的数据对象，它的数据宽度是 5，而字面量为 16 的无符号整数类型的数据对象，它的数据宽度是 6。然而，有些时候我们希望自

定义数据宽度，例如，要构造一个无符号整型的数据对象，字面量为 16，数据宽度为 8，在 Chisel 中可以这样写：

```
16.U(8.W)
```

相应的，在 PyHCL 中可以这么写：

```
U(8).w(16)
```

3.2.4 Bundle 类型

Bundle 可以封装不同类型的数据，作用类似于高级程序语言当中的结构体或者类。而用户使用 Bundle 的时候，通常是自定义一个类来继承基类 Bundle。

3.3 对 PyHCL 的一些未完善功能的思考和解决

PyHCL 是我们团队近期自研的硬件构造语言，迭代周期比较短，也没有经过很完善的测试，因此在功能上难免有些不足。所以我们在实际的开发过程中，往往会遇到这样的场景，即某个 Chisel 实现了的功能，PyHCL 有没有实现了等价的功能，如果已经实现，那就要确保这个功能的正确性再拿来使用；否则，我们可能要和 PyHCL 的开发人员进行沟通，让他们加上这些功能，又或者是我们自己根据这次开发工作的需要，来自己实现这部分功能。下面会列出 PyHCL 的一些比较典型的问题，并且展示解决办法，还有想出来这些办法的思路，再通过比较 PyHCL 实现的 PyVTA 和 Chisel 实现的 VTA，展示 PyHCL 如何用到这些功能。当然解决办法可能不止一种，可能还有些更好的解决办法自己也没想到，所以列出来的只是个人的思考，不一定是最优方案。

3.3.1 Bundle 和平铺

无论是哪种语言，都需要表示 I/O 端口，我们以 Chisel 实现的 VTA 的一段代码为例，如图 3-1，来简要说明一下 I/O 端口在 Chisel 中是如何表示的，然后再看一下如何在 PyHCL 中表示相同的端口。

```
val io = IO(new Bundle {
  val launch = Input(Bool())
  val ins_baddr = Input(UInt(mp.addrBits.W))
  val ins_count = Input(UInt(vp.regBits.W))
  val vme_rd = new VMEReadMaster
  val inst = new Bundle {
    val ld = Decoupled(UInt(INST_BITS.W))
    val co = Decoupled(UInt(INST_BITS.W))
    val st = Decoupled(UInt(INST_BITS.W))
  }
})
```

图 3-1 Chisel 表示一簇 I/O 端口

从图 3-1 这段代码中可以看出，Chisel 的功能还是比较强大的，这一点在表示 I/O 端口时也体现得非常明显，从代码中可以看出，如果要表示多个 I/O 端口，并不需要对每个 I/O 端口都声明一次，而是只需要用一个变量来表示这一簇 I/O 端口，而这一簇 I/O 端口，在声明的时候只需要用 Bundle 类型嵌套起来，而且 Bundle 嵌套的成员也可以是任何合理的类型，例如在示例代码中，luanch 是最简单的 Bool 类型的输入，ins_baddr 和 ins_count 是 UInt 类型的指定了宽度的输入，而 vmd_rd 是继承了 Bundle 的类，这个类的成员是一系列的 Input 和 Output，也就是说 vmd_rd 可以视为一个 Bundle 的实例，

最后 inst 也是一个 Bundle 的实例，成员为一系列 Input 和 Output。代码里出现的 Decoupled 相当于解耦操作，返回一个 Input 或 Output，Decoupled, Fliped, ValidIO 会在后面提及。

通过上面的分析，可以看到 Chisel 实现 I/O 端口还是很简单的，开发人员只需要傻瓜式的把 Input 和 Output 塞到一个 Bundle 里就可以了。但是当事情到了 PyHCL 的时候，会发现问题可能没有那么简单。因为 PyHCL 还是一门新兴的硬件构造语言，所以功能比起 Chisel 还没有那么完善，其中一个很重要的地方就是，它虽然也可以实现表示 I/O 端口，但是如果它要实现用一个类来表示一簇 I/O 端口的话，那么这一簇成员是无法再有嵌套结构的，也就是说，这些 I/O 端口成员必须是简单数据类型，或者说，一个键值对，而不是像 Chisel 那样，最外层只需要一个 Bundle，而且 Bundle 里面又可以嵌套 Bundle。但是很多时候，又无法避免一些 Bundle 类型的数据结构，因此对于这种结构，其中一种解决办法就是把 Bundle 结构平铺开，例如 a 嵌套 b，b 又嵌套 c 和 d，那么平铺开之后，原来的成员 c 就变成了 a_b_c，原来的成员 d 就变成了 a_b_d，这两个成员都从嵌套的内层放到了最爱层，而命名规则是通过下划线，从外到内把原先的变量名连接起来。而要做到这一点，我们就需要实现一个展开的功能，具体的实现可以是这样的，参考图 3-2。

```
def mapper_helper(bundle, dic=None, prefix=""):
  tdic = {} if dic is None else dic

  for k in bundle.__dict__:
    v = bundle.__dict__[k]
    if isinstance(v, Pub):
      if prefix == "":
        tdic[k] = v
      else:
        tdic[prefix+"_"+k] = v
    elif isinstance(v, Bundle_Helper):
      if prefix == "":
        mapper_helper(v, tdic, k)
      else:
        mapper_helper(v, tdic, prefix+"_"+k)
    elif isinstance(v, List):
      for i in range(len(v)):
        if isinstance(v[i], Pub):
          if prefix == "":
            tdic[k+"_"+str(i)] = v[i]
          else:
            tdic[prefix+"_"+k+"_"+str(i)] = v[i]
        elif isinstance(v[i], Bundle_Helper):
          if prefix == "":
            mapper_helper(v[i], tdic, k+"_"+str(i))
          else:
            mapper_helper(v[i], tdic, prefix+"_"+k+"_"+str(i))

  return tdic


def mapper(bundle):
  dct = mapper_helper(bundle)
  io = IO(**dct)

  return io
```

图 3-2 PyHCL 平铺 Bundle 成员

先解释一下这个函数的参数。bundle 很好理解，就是这次需要展开的 Bundle，而 dic 是用来记录展开结果的字典，prefix 是命名的前缀。其中后两个参数都是考虑到递归调用的情况才引入的，在最外层的递归，这两个参数默认都为空，在内层的迭代里，dic 需要记录之前外层迭代的展开结果，而 prefix 记录外层展开得到的命名前缀。具体怎么使用这两个参数后面再做进一步解释。

然后浏览一下代码的大致思路。mapper_helper 函数遍历 bundle 的所有 kv 结构，k 当然会是一个字符串，对于每一个 kv 对，都会去根据 v 的类型，来决定对 v 如何操作。从代码可以看出，这里考虑了 v 的类型的几种可能性，可能是 PyHCL 的基本数据类型 Pub，或者是 Bundle 类型，还有可能是向量类型 List。对于这些类型也一一做了相应的处理。

如果值 v 是 PyHCL 的基本数据类型，那就没有太多好说的，直接把 v 加到记录展开结果的字典 dict 里，这里要注意的是，要考虑前缀是否为空的情况，前缀不为空，说明这不是最外层的递归，就好像之前说的，a 嵌套 b，b 嵌套 c，而这次递归已经展开到 a_b，接下来展开 c，那么命名就需要加上之前的前缀和下划线，才能得到命名 a_b_c，因此需要判断前缀是否为空。

如果值 v 是 Bundle 类型，那么就需要再对这个 v 进行展开，所以需要递归调用。同样地，这里也需要判断前缀是否为空。

如果 v 是 List 向量类型，那就要遍历向量的每个元素，对每个元素都执行类型的判断，当然还是不要忘记考虑上前缀的情况。

这是展开函数的实现，利用这个函数，就可以对 Bundle 类型进行展开了，如图 3-3。

```
class Fetch_IO(Bundle_Helper):
    def __init__(self):
      self.launch = Input(Bool)
      self.ins_baddr = Input(U.w(mp.addrBits))
      self.ins_count = Input(U.w(vp.regBits))
      self.vme_rd = VMEReadMaster()
      self.inst = Inst()

  # Fetch module
class Fetch(Module):
  # Construct IO
  io = mapper(Fetch_IO())
  # ...
```

图 3-3 通过平铺实现一簇 I/O 端口

上面这个思路，是在 PyHCL 没那么完善的前提下，自己实现的一个 Bundle 展开的功能。它能解决一部分问题，但显然这样暴力枚举 Bundle 的值 v 的类型，并不是一个很好的策略，在这里只考虑了值 v 的一些类型，例如 PyHCL 的基本数据类型，Bundle 类型和向量 List 类型，而对于其它类型它就没有解决办法了，如果要对其它类型做处理，又要进行开发，加上这种类型的处理情况。因此这种做法，对于很小的一个模块可以这么做，但是对于 I/O 端口和 Bundle 嵌套这些随处可见的情况，显然不是长久之策。而且，

它还有一个很严重的问题，就是对于向量的展开。假设现在 a 嵌套 b，b 嵌套 c，而 c 的类型是向量，有 10 个元素，那么按照这种做法，展开的结果会是 a_b_c_0, a_b_c_1, ... , a_b_c_9，这种表示法在某些情况下会大大增加代码的复杂度，例如现在需要遍历这个向量的所有成员，如果按照 Chisel 的做法，没有实现展开，那么就只需要一个循环，每次迭代索引 idx 自增一次，a.b.c[idx]就可以访问到每个元素了；而按照现在这种展开的做法，想要访问向量的每个元素，要么手动写 10 次访问的代码，要么想办法进行字符串的拼接，再进行循环。无论哪种方法，都是很不方便的。事实上，后来也遇到了这样的问题，虽然向量的元素只有两三个，但是已经明显能感受到，这种每个元素的重复，对代码质量的影响。

也正是因为上述原因，后来还是联系了一下 PyHCL 的开发人员，和他们进行沟通之后，他们添加了这个功能，所以最后可以这样实现。

```
class Fetch(Module):
  io = IO(
    launch = Input(Bool)
    ins_baddr = Input(U.w(mp.addrBits))
    ins_count = Input(U.w(vp.regBits))
    vme_rd = VMEReadMaster()
    inst = Inst()
  )
  # ...
```

图 3-4 PyHCL 改进后的 Bundle 实现一簇端口

以上就是对 Bundle 类型的 I/O 端口的处理。

3.3.2 Decoupled, Flipped 和 Valid

下面看一下硬件描述语言当中的一些常用方法，以及它们的实现。

首先是 Decoupled 方法，它做的事情很简单，就是为传进去的参数封装一层，这一层有 valid 和 ready 输出。要实现这个方法，也有不止一种做法，在做这次的开发工作之前，PyHCL 尚未实现 Decoupled 功能，我自己去看了一下，实现也并不算特别困难，所以就给 PyHCL 提了 commit，增加了自己对于这个方法的实现。这次采取的做法是，直接在外面套一层 Bundle，Bundle 有成员 valid 和 ready 作为输入输出，参数就赋给 bits，这也是 Chisel 大致的实现思路，PyHCL 就参考了 Chisel 的实现，代码可以按照图 3-5 方式来写。

```
def Decoupled(typ):
  from .bundle import Bundle
  from .cdatatype import U
  return Bundle(
    valid = U.w(1), #Output
    ready = U.w(1).flip(), #Input
    bits = typ
  )
```

图 3-5 PyHCL 实现 Decoupled

然后是 Filpped 方法，这个方法实现了对端口输入输出方向的翻转。要注意的是，我们希望的是能够对一个 Bundle 类型的所有成员进行输入输出方向的翻转，因此这又要涉及到 Bundle 的所有成员的遍历和嵌套。PyHCL 也是对这个功能的支持还不完善，对于如何实现这个功能，我也做了一些考虑，和别人做了一些讨论，最后还是决定按照

最直接的思路，和之前描述的平铺一样，对参数进行类型判断，每个类型都有相应的处理。具体实现参考图 3-6 代码。

```
def base_flipped(obj):# bj U.w(1)
  return Output(obj.typ) if isinstance(obj.value, Input) else Input(obj.typ)


def flipped(bundle):
  dic = bundle.__dict__
  for keys in dic:
    if isinstance(dic[keys], Pub):
      dic[keys] = base_flipped(dic[keys])
    elif isinstance(dic[keys], Bundle_Helper):
      flipped(dic[keys])
    elif isinstance(dic[keys], Vec):
      for v in dic[keys]:
        flipped(v)

  return bundle
```

图 3-6 PyHCL 实现 filpped

正如之前所说的，枚举每种类型有两个问题，一个是可能没法枚举完全，有时候我们很难得知，会有什么类型来调用这个函数；另外一个问题是，如果类型是向量的话，需要留意会不会出现不方便通过下标来访问的情况，从而导致代码繁琐。对于第一个问题，是枚举这种做法很难避免的，但是在这里，因为目前在整个项目中，参数 bundle 的类型也就这几种，因此枚举出 bundle 为基本类型，Bundle 类型或者向量类型的情况也足够了；而对于第二个问题，这里只是对向量的每个元素进行翻转操作，不会重新命名，所以也不存在命名使得代码更繁琐的情况。所以，这样实现翻转操作 Flip，是可行的。

最后来看一下 Valid 方法。Valid 方法也是对参数做了一层封装，并且引入一个输出端口 valid，实现起来的思路和之前的 Decoupled 和 Flipped 类似，都是判断参数的类型，

而且都要考虑 Bundle 嵌套的情况。具体如何实现可以参考图 3-7 代码。

这样，就实现了硬件构造语言中的 Decoupled，Flipped 以及 Valid 这些常用的方法。

```
def Valid(typ):
  from .bundle import Bundle
  from .cdatatype import U
  from .cio import Output
  from ..core._repr import CType
  from .vector import Vec

  coupled = Bundle(
    valid=U.w(1),
    bits=Output(typ)
  )

  if isinstance(typ, CType) or isinstance(typ, type):
    coupled.bits = Output(typ)
  elif isinstance(typ, Vec):
    coupled.bits = Output(typ)
  elif isinstance(typ, Bundle):
    coupled.bits = Bundle()
    dic = typ.__dict__
    for keys in dic:
      if isinstance(dic[keys], CType) or isinstance(dic[keys], type):
        coupled.bits.__dict__[keys] = Output(dic[keys])
  return coupled
```

图 3-7 PyHCL 实现 Valid

3.3.3 状态机

在硬件设计中，经常要根据某些状态来决定硬件下一步的动作，这些状态包括输入输出端口的高低电平，寄存器的值等等。为了描述这种根据某些状态来进行下一步动作的情况，我们引入状态机来表示这种情况。以 VTA 的代码为例，看一下 Chisel 是如何描述一个状态机的，见图 3-8。

```
switch(wstate) {
  is(sWriteAddress) {
   when(io.host.aw.valid) {
    wstate := sWriteData
   }
  }
  is(sWriteData) {
   when(io.host.w.valid) {
    wstate := sWriteResponse
   }
  }
  is(sWriteResponse) {
   when(io.host.b.ready) {
    wstate := sWriteAddress
   }
  }
 }
```

图 3-8 Chisel 表示状态机

Chisel 可以理解为 Scala 的库，因此 Chisel 描述状态机也要符合 Scala 的语法。switch 后面的括号里是要判断的状态，然后每种情况的分支用 is 来判断。与之类似的，PyHCL 实现状态机也是大同小异，通过 withwhen 来实现判断，见图 3-9 。

```
with when(wstate == sWriteAddress):
    with when(io.host.aw.valid):
      wstate <<= sWriteData
  with elsewhen(wstate == sWriteData):
    with when(io.host.w.valid):
      wstate <<= sWriteResponse
  with elsewhen(wstate == sWriteResponse):
    with when(io.host.b.ready):
      wstate <<= sWriteAddress
```

图 3-9 PyHCL 表示状态机

3.4 本章小结

本章介绍了这次工作相关的两种硬件构造语言，PyHCL 和 Chisel，包括它们的编译

流程和产物，它们常用的一些数据类型，然后花了一些篇幅来说明现在 PyHCL 的一些不完善的地方，以及我个人对这些问题的思考和解决思路。

第四章 深度学习加速器堆栈基础

我们实现的深度学习加速器堆栈 PyVTA，最终的目标是集成到 Apache TVM。Apache TVM 是一个开源的深度学习编译栈，支持了 CPU、GPU 和一些特定的加速器。下面介绍一下 PyVTA 的堆栈结构，它在 Apache TVM 中的层次，以在架构层面了解 PyVTA，然后再去探究它的一些实现细节。

4.1 堆栈结构概览

既然说 PyVTA 是一个深度学习加速器堆栈，那它肯定有堆栈的层次结构，在这里用图 4-1 来表示这种堆栈结构，并且对每一层结构做一些说明。

图 4-1 深度学习加速器堆栈 PyVTA 堆栈结构概览

深度学习框架。这是最顶上的一层，这一层就是深度学习从业人员非常熟悉的 TensorFlow, PyTorch 等等这些深度学习框架，程序员通过这些框架来表达深度学习模

型，在这个堆栈结构当中，可以把它理解为模型的抽象表达。

Relay Graph 优化器。这一层是 TVM 的高层次表示，也就是说，这一层接收深度学习框架的模型和参数，并且把它表示成 Relay。Relay 做了这样一件事，把各种深度学习框架表达的模型和参数，或者说计算图，概括成 Relay 这种编程语言，这样就起到了适配各种深度学习框架，把它们用同一种编程语言表示的作用。而 Relay 本身也具有一定的可扩展性，这一点页体现在在 TVM 编译堆栈的层次结构中，Relay 考虑到了当前常用的一些深度学习加速器堆栈的设计，针对深度学习加速器堆栈做了一些适配工作，例如，Relay 有针对深度学习加速器堆栈的一组优化，会量化输入以匹配深度学习加速器堆栈的低精度数据类型；还会转换数据的排列方式，尽可能地对数据进行重用；此外，Relay 还会转换输入和权重的排列方式，以此利用深度学习加速器堆栈的张量函数。

TVM 操作优化器。这一层接收上层的 Relay 表示，并且把工作负载分给加速器的各硬件。PyVTA 的最终目的就是加速运算，因此我们需要把运算负载分配到 PyVTA 加速器上，而想要最大化利用 PyVTA 的硬件，最大化并行的程度，就需要合理地调度。TVM 使得这种调度过程实现了自动化。调度的重要性不言而喻，首先是，它把计算平铺开，使得数据的重用性最大化；其次，它的线程是并行的，PyVTA 的运行时可以把若干个任务放到任务流水线上进行作业，这和流水线处理器的思路是类似的；还有，它把运算拆分成多个子计算，这些子计算会被映射到高级硬件内联函数，例如 GEMM 矩阵运算或者批量 DMA 负载。

JIT 编译器和运行时。这一层实现编译，并且执行编译产物。从这一层开始往下，是属于 PyVTA 的层次结构，把 PyVTA 集成到 Apache TVM 也就是把这层往下的集

成到 Apache TVM 编译器堆栈中。后面会对 JIT 编译器和运行时做进一步的介绍。

硬件架构。这个硬件架构当中值得注意的是，它体现了 PyVTA 的可参数化这一特性。具体来说，这个架构可参数化的地方有，GEMM 内核的大小，SRAM 芯片的形状，输入数据以及其它各种数据的宽度等等。参数化的意义在于，同样的设计，可以在不同的硬件资源上实现。这就呼应了在绪论当中提到的，深度学习加速器堆栈的通用化，这也正是可编程的深度学习加速器堆栈，比起其它深度学习加速器堆栈的优势所在，之前的很多深度学习加速器堆栈都往往只适用于比较固定的硬件资源，即使 FPGA 是可编程的，依旧没有完全解决这个问题，而 VTA 和 PyVTA 真正意义上地实现了对硬件的通用化，面对各种各样的硬件资源，我们不再需要重新设计一套加速器堆栈来适配硬件，不需要重新编程，这就是 VTA 和 PyVTA 相比起其它深度学习加速器堆栈的一个决定性的优势。后面会对硬件架构做进一步的介绍。

4.2 硬件架构

PyVTA 的硬件架构可以用图 4-2 简略描述。下面对这个架构图当中的各个模块，以及它们之间的交互，做简要介绍。

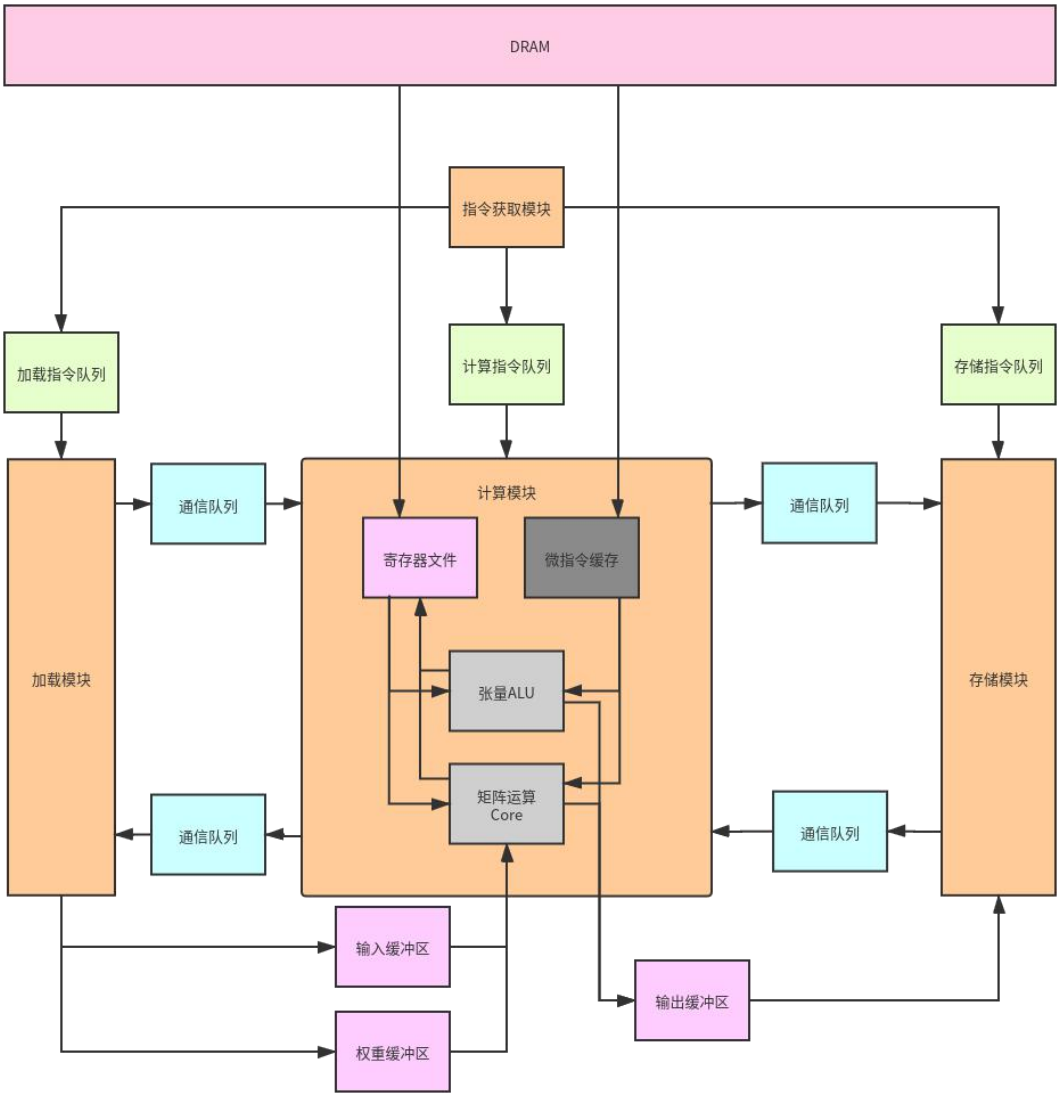

图 4-2 深度学习加速器堆栈 PyVTA 硬件架构概览

4.2.1 对硬件架构的模块的介绍

从这张架构图 4-2 可以看出，PyVTA 的硬件架构可以大致分为四个模块：指令获取模块（instruction module），加载模块（load module），计算模块（compute module）和存储模块（store module）。这其实和现代处理器的思路是类似的，只不过 PyVTA 的这些结构比现代处理器简单得多，毕竟深度学习加速器的功能和复杂度和现代处理器不可同日而语。为了对 PyVTA 如何在硬件层面上实现功能有一个初步的认识，我们还是

来看一下各个模块是如何协同工作的。

首先要明确的一点是，在整个 TVM 编译堆栈层次结构中，所有的高级编程语言的代码，最后都会通过编译汇编等等一系列过程，转化成二进制码。这些指令的二进制码会在后面做进一步介绍。编译的工作是由 PyVT 硬件架构的上一层，也就是 JIT 编译和运行时这一层实现的。编译出来的二进制码会被存储在 DRAM。下面一一介绍各个模块实现的大致功能。

指令获取模块会从 DRAM 获取指令和数据，然后根据指令的类型，把这些指令分发给加载模块，计算模块和存储模块，这种分发通过压入每个模块的指令队列来实现。

加载模块从指令队列获取加载指令，而加载的意思就是把数据写到寄存器文件，这里的数据指的是输入和权重，因此加载模块把输入和权重先放到输入缓冲区（input buffer）和权重缓冲区（weight buffer），等待计算模块来获取。

计算模块也是先从指令队列获取指令，然后这个计算指令可能是矩阵运算（GEMM），也可能是张量的 ALU 运算（Tensor ALU），如果是张量的逻辑运算，那么就从寄存器文件中获取数据进行逻辑运算，而如果是矩阵乘法运算，就从输入缓冲区和权重缓冲区取出权重和缓冲进行矩阵点乘运算。矩阵乘法运算在神经网络上是最常见的运算，因为这些网络的计算过程都是，通过对前一层的输入和这一层的权重进行乘法运算，得到的输出作为后一层的输入。因此设计一个专门的矩阵乘法运算核（GEMM Core）是有必要的。总之，张量逻辑运算或矩阵乘积运算完后，把运算结果写入寄存器文件或者输出模块，因为有些数据的流向可能是寄存器文件，而另一些数据可能需要写回 DRAM，因此需要写入 DRAM 的数据，会被先写到输出缓冲区（output buffer），等待存储模块来获取。计算模块还有一个微指令寄存器（micro-op cache），存储微指令

（micro-op）,而微指令是对 DRAM 当中获取的指令进行进一步拆分得到的,例如 load 指令可能会被拆分成若干个操作，这么做是为了更加充分地利用流水线，这和现代处理器对指令的拆分是类似的，当然这个微指令的拆分与否，不太影响整个硬件架构。实际上如果不做这样的指令拆分，直接把 DRAM 的指令拿出来就进行计算，也是完全可行的，不过为了使得深度学习加速器堆栈的性能更好，我们的设计还是保留了指令拆分的步骤。

存储模块从指令队列取出的是存储指令，它的功能是把数据存储到 DRAM，而这些数据往往是计算模块计算出的结果，放在输出缓冲区，因此存储模块只需要把输出缓冲区的数据取出来，存到 DRAM 就可以了。

此外，加载模块和计算模块之间，存储模块和计算模块之间，都有通信队列，实现模块间的通信。

以上，就是对这个硬件架构图里的各个模块的简略介绍。

4.2.2 指令编码

指令编码指的是指令通过编译汇编等过程，最后得到的二进制码，它们的格式可以用图 4-3 表示。

Load, Store 指令

字段宽度	memop_size_width	memop_size_width	memop_stride_width	memop_dram_width	memop_dram_width	memop_dram_width	memop_dram_width	
32~61 位	y_size	x_size	x_stride	y_pad_0	y_pad_1	x_pad_0	x_pad_1	unused

字段宽度	opcode_width	4	memop_id_depth	memop_sram_addr_width	memop_dram_addr_width	
0~31 位	opcode	dept flags	buffer id	sram base	dram base	unused

GEMM 指令

字段宽度	log_acc_buf_depth	log_acc_buf_depth	log_inp_buf_depth	log_inp_buf_depth	log_wgt_buf_depth	log_wgt_buf_depth	
32~61 位	x0	x1	y0	y1	z0	z1	unused

字段宽度	opcode_width	4	1	memop_sram_addr_width	memop_dram_addr_width	loop_iter_width	loop_iter_width	
0~31 位	opcode	dept flags	reset	uop_begin	uop_end	end0	end1	unused

ALU 指令

字段宽度	log_acc_buf_depth	log_acc_buf_depth	log_inp_buf_depth	log_inp_buf_depth	alu_opcode_wigth	1	aluop_imm_width	
32~61 位	x0	x1	y0	y1	op	use_imm	imm	unused

字段宽度	opcode_width	4	1	memop_sram_addr_width	memop_dram_addr_width	loop_iter_width	loop_iter_width	
0~31 位	opcode	dept flags	reset	uop_begin	uop_end	end0	end1	unused

图 4-3 指令格式

可以看到，指令通过 opcode 字段分为三类，load 或 store 被归为一类，都是对 DRAM 的读写，GEMM 指令是矩阵运算，ALU 是张量运算。上文提到的指令获取模块也正是通过这个字段把不同类型的指令分发给相应的模块的。下面对每种指令的一些关键字段进行简单地介绍。

load 和 store 指令，如前文所述，在 SRAM 和 DRAM 之间进行步长为 2D 的 DMA 读写，因此 sram_base 字段和 dram_base 字段是对 SRAM 和 DRAM 的寻址，x_stride 指明了步长，x 和 y 指示了数据的长度，而 pad 相关的字段是为了对齐而进行的填充。

GEMM 指令，是进行矩阵乘法运算的指令，而做乘法的两个矩阵，一个来自于输入缓冲区，一个来自于权重缓冲区，因此 y0, y1 字段是输入缓冲区的索引，z0, z1 字段是权重缓冲区的索引，x0, x1 字段是累加寄存器的索引。例如输入缓冲区有多个输入数据，根据这个索引就可以在缓冲区中找到这次要做运算的输入数据。而如上文提到的，

从 DRAM 取出来的任务级指令，可能会被拆分成若干条微码指令，因此需要从微码指令缓存中取出微码指令，而 uop_bgn, uop_end 字段就指明了这条微码指令在微码缓存中的位置。GEMM 指令如何从输入缓冲区和权重缓冲区中取出数据，如何从微码缓冲区取出微码指令，如何存储数据到输出缓冲区，这些内容放到 3.2.3 做进一步介绍。

ALU 指令，是进行向量运算的指令，例如向量相加，通常这会发生在神经网络中的激活，规范化，池化等任务。

以上就是对任务级指令的一些基本介绍。

4.2.3 GEMM 运算

上文说道，计算模块中的 GEMM 矩阵乘法计算，会从输入缓冲区和权重缓冲区取出数据，并进行矩阵乘积运算，然后把运算的结果写入输出缓冲区。而这个过程的具体实现，可以用图 4-4 表示。

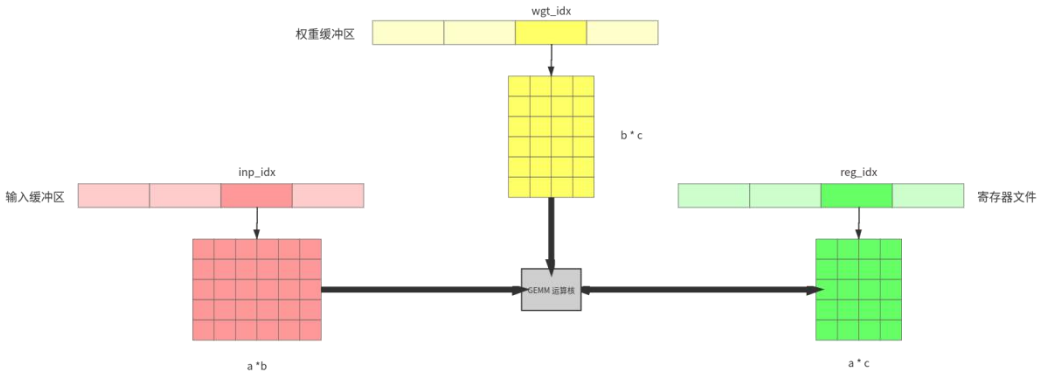

图 4-4 矩阵乘积运算在硬件层面的实现

从这张图可以看出，GEMM 核的操作数包括输入、权重、偏置和累加和。这些操作数都来源于寄存器文件，在架构图中，寄存器文件包括画在计算模块的寄存器文件，和各种数据的缓冲区，包括输入缓冲区，权重缓冲区等等。在这些操作数中，输入是通

过 inp_idx 索引，在输入缓冲区查找到的；权重是通过 wgt_idx 索引，在权重缓冲区得到的。而 inp_idx 和 wgt_idx 这两个索引，都是通过查找微码指令中的字段，然后对它们进行仿射计算得到的。而输入和权重进行矩阵乘法运算之后，需要和目标寄存器的原来的值相加，得到的结果再写回目标寄存器，这个目标寄存器在寄存器文件中的索引为 reg_idx。

这一节的内容简要介绍了 GEMM 运算如何在 PyVTA 的硬件架构上执行的。

4.2.4 ALU 运算

ALU 运算，图 4-5，本质上是向量之间的相加运算，运算的两个操作数分别为寄存器文件中，下标为 src_idx, dst_idx 的寄存器值。从寄存器文件中取出来这两个值做相加运算之后，把结果写回索引为 dst_idx 的寄存器。

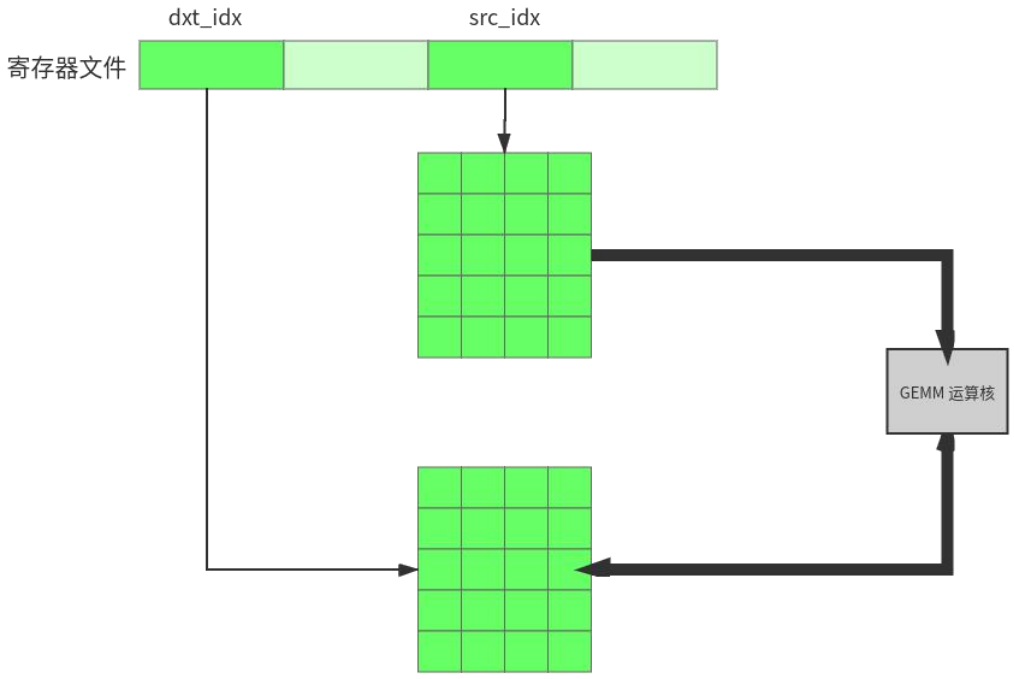

图 4-5 向量 ALU 运算在硬件层面的实现

4.3 流水线设计

从 PyVTA 的硬件架构图，也就是图 4-2 中，可以看出，这样的硬件架构是有实现流水线的潜质的。而 VTA 和 PyVTA 都确实实现了三级的 加载-计算-存储 任务级流水线。在说明如何设计这个流水线之前，首先假设一下，在没有引入流水线的情况下，PyVTA 是如何实现功能的。

4.3.1 没有流水线的情况

从图 4-2 可以看出，一个最小的任务，是根据输入和权重，计算出输出，这也是绝大多数神经网络的工作负载。因此，这个任务执行的流程是，加载模块从 DRAM 取出输入和权重，并且加载到输入和权重缓冲区，然后计算模块从输入和权重缓冲区取出输入和权重，存到寄存器文件，并且从微指令缓存取出计算指令，根据计算指令的种类，进行 GEMM 计算或 ALU 计算，最后计算出的结果，写到输出缓冲区，由存储模块取出来，写回 DRAM。

这就是一个最小的任务的执行过程，先后经过了加载模块，计算模块和存储模块。如果没有流水线，那么整个 PyVTA 对这些任务的执行就是，执行完这个计算任务，把输出写回 DRAM 之后，再去执行下一个任务。用图来表示可以是这样的，如图 4-6

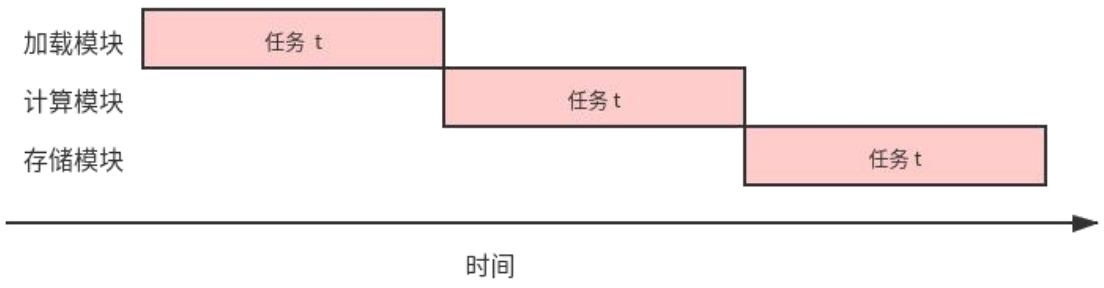

图 4-6 非流水线

4.3.2 等划分的流水线

这样执行任务，显然没有最大程度地利用硬件资源，因为从图 4-6 中可以看出，在加载模块工作的时候，计算模块和存储模块都处于被阻塞的状态，计算模块要等到加载模块完成这次属于它的任务，并且把输入和权重写到输入和权重缓冲区，计算模块才能开始工作；同理，存储模块要等到计算模块把输出写到输出缓冲区，才能开始工作。这会使得深度学习加速器堆栈的性能大打折扣，为此我们引入流水线来最大限度地降低这种资源浪费，用图 4-7 来表示这个流水线的设计思路。

图 4-7 等划分的流水线

从图 4-7 中可以看出，和非流水线执行这些任务不同， 在加载模块执行任务 t，即从 DRAM 取输入和权重加载到输入和权重缓冲区时，计算模块可能在执行上一个任务 t - 1 的一部分，等到输入和权重被加载到缓冲区时，计算模块可能也执行完了上一个任务 t - 1 中属于它的那部分工作，把结果写入输出缓冲区，于是这时的计算模块从缓冲区取出输入和权重，进行任务 t 计算；同理，计算模块执行任务 t 的时候，存储模块也不会空等，而是可能在执行任务 t - 1 的取出结果并写回 DRAM 这一步骤。

那么，如何衡量这个流水线对性能的提升呢？基于现有的功能，我们很难去真正观察到执行一个任务时每个模块的耗时，不过基于流水线的理论知识，我们可以大致估计

流水线对性能的提高。流水线的目的就是提高吞吐量，在这里，吞吐量可以被量化单位之间内，执行任务的个数。在最理想的情况下，我们假设每个模块执行任务的延时都是相等的，那么在任务源源不断地被执行的理想情况下，这个流水线的性能就是非流水线的三倍，因为非流水线的情况下，需要三个单位时间才能执行一个任务，而引入流水线后，三个单位时间可以执行三个任务。

4.3.3 不等划分的流水线

当然，这是非常理想化的情况，这里至少做了两个理想化的假设，一个是流水线上会有源源不断的任务送进来被执行，另一个是所有模块执行一个任务的属于它的那部分的耗时相等。第一个假设相对来说是比较符合现实情况的，因为深度学习算法往往是计算密集型的，对硬件资源的占用非常大，因此可以视为有源源不断的任务进入流水线；但是第二个假设和实际情况差别不小，实际上，在这几个模块中，计算模块的负担经常会远远比加载模块和存储模块要重，尤其是 GEMM 核执行的矩阵乘法运算，这也是符合经验的，计算往往比取址取值这些操作要耗时更多。而在加入了这个更加现实的考虑因素以后，流水线执行源源不断的任务，更有可能会是图 4-8 的这种情况。

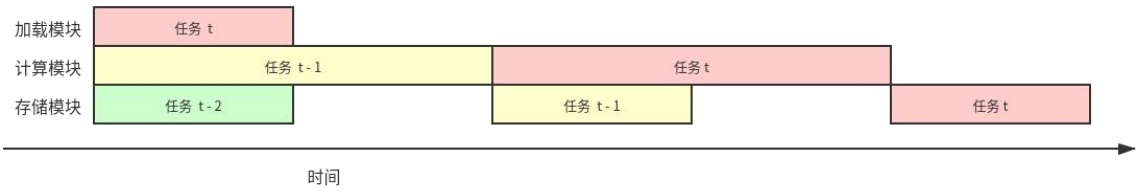

图 4-8 不等划分的流水线

从图 4-8 中可以看出，制约流水线性能的主要是计算模块的耗时。如果是之前每个

模块耗时相等的理想化的流水线，三个单位时间内可以执行三个任务，而现在，需要五个单位时间才能执行三个任务。

既然流水线的划分是不一致的，也就是说每个模块执行任务的耗时可能是不相等的，那么模块之间就需要通信，例如加载模块需要告知计算模块，任务 t 的加载部分已经完成，输入和权重已经被加载到输入和权重缓冲区的哪个位置，计算模块在收到消息之后才能得知，应该从缓冲区的这个位置获取输入和权重进行计算，而计算模块也需要告知存储模块，应该从输出缓冲区的哪个位置获取计算结果。因此，在每个模块之间需要一个进行通信的消息队列，这在图 4-2 中也已经体现出来了。

通过之前的分析，我们已经发现，流水线的划分程度越细致，也就是流水线的深度越深，那么性能就越高。事实上情况是否真的如此？对流水线进行更细致的划分又是否合适？对于第一个问题，在流水线的深度不会太深的情况下，性能和深度的确是呈正相关的，但是也不能忽略状态存储和读取的耗时。在刚才的分析中，我们其实还做了一个理想化的假设，就是每个时钟周期的状态存储和读取的时间消耗可以忽略不计。无论是什么硬件资源执行什么任务，它都会经过若干个时钟周期，每个时钟周期都会有时钟信号，控制硬件资源加载状态寄存器，因为想要顺序执行，就要保存上个时钟周期的状态。而流水线划分越细致，这种状态的读取和存储就越频繁，因此在流水线深度足够的情况下，读取和存储状态寄存器的耗时不再可以忽略不计，在设计流水线的时候也需要考虑这一点。

而对于第二个问题，其实也有过这样的想法，既然流水线的瓶颈是计算模块，那可不可以把计算模块再划分为 GEMM 运算和 ALU 运算，从而使流水线的深度更深？这在理论上是可行的，因为 GEMM 负责矩阵运算，ALU 负责向量加法运算，这两者有

各自的计算核，不会占用相同的硬件资源；而实践上也可能是一种可行的方案，但是这样会增加硬件的复杂度，在 GEMM 运算核和 ALU 运算核之间要有通信队列，所以最后还是没有采用这种方案，而是采用了默认的三阶段的加载-计算-存储流水线。

4.4 JIT 运行时系统

JIT 编译和运行的功能，在 TVM 编译堆栈中，处于底层的硬件架构的上一层。这一层负责把代码编译成二进制码，并且运行它们。其中，运行的时候可以在 CPU 主机和加速器上协同完成深度学习的工作负载。这是可以理解的，因为之前已经说到了，PyVTA 具有运算的功能，也就是完成深度学习工作负载的功能，而 CPU 当然也是可以做到这些的，所以为了最大化地利用资源，当然是要把这两者都利用起来。而为了这两者能够正确实现这些功能，JIT 运行时的设计也遵循了一些规范，下面对这些规范做简单介绍。

首先是把工作负载调度到 CPU 和 PyVTA 上的调度策略。这很容易理解，因为有些类型的工作负载显然对 CPU 更友好，也就是说由 CPU 来实现，速度会更加快或资源会得到更充分利用，例如 CNN 中的低一层卷积层，这一层的算术强度比较低，把这些工作负载交给 CPU 来执行，表现非常良好。当然，调度策略还需要考虑到 PyVTA 自身的功能限制，因为现代处理器的功能毕竟还是比 PyVTA 强大不少，有些操作符 PyVTA 能够有对应的实现，这时候这种工作负载也只能分发到 CPU 上。

其次是，要考虑到硬件的限制。其中非常显著的一点就是，VTA 片上存储能力是有限的， 微指令缓存需要存储微内核，还有微指令，这样的负载是比较重的，因此 JIT 需要动态地把微内核和微指令加载到微操作缓存中，也就是需要即时地了解到微操作缓存当前的存储情况，存储空间大小等信息，在合适的时刻把微内核或微指令加载到微操作

缓存中。一般来说，JIT 会在编译的时候才生成微内核，并且加载到微操作缓存，而且也不会一次性把整个微内核都加载到微操作缓存上，因为整个微内核的大小是相当可观的，JIT 会倾向于实现在必要的时候加载一部分微内核。

总之，JIT 的设计，在最大程度上迁就了深度学习加速器堆栈的硬件架构的限制。

4.5 本章小结

本章主要介绍深度学习加速器堆栈 PyVTA 的内容，包括 VTA 在整个编译堆栈当中的层次，VTA 的硬件架构，流水线的设计，还有 JIT。而在硬件架构这一部分，其实也是我们实践工作的重点，对于硬件架构这一部分，我们首先介绍了这个硬件架构中的各个模块，以及它们之间如何交互通信。然后简单列出来了指令的二进制格式，它们一些重要字段的含义。最后，在硬件架构这一部分，比较重要的计算是矩阵运算 GEMM 和向量运算 ALU，因此我们也对这两种运算的实现做了分析。

第五章 代码测试和成果演示

回顾一下之前的内容，首先我们介绍了硬件描述语言（HCL），它是目前硬件开发中的主流语言工具，当然它也存在一些问题，其中比较关键的一点就是，它写起来代码比较繁琐，而且开发门槛相对比较高，需要对硬件有一定的基础，这显然不利于软件从业者参与到开发中去。

顺着这个思路，我们介绍了高层次综合（HLS），它是用高级编程语言来写代码，然后通过分配，部署和绑定等步骤，把高级编程语言转化为硬件描述语言。高层次综合的优势在于降低了开发门槛，因为开发人员是用高级编程语言来写代码，这使得对硬件并不是那么了解的软件从业人员也可以参与到硬件开发中去。而高层次综合也还有不完善的地方，首先它对硬件的利用效率比不上硬件描述语言。这是合乎情理的，因为高层次综合毕竟不是直接描述硬件，因此转化得到的硬件描述语言，可能并不是最优的，即转化得到的硬件描述语言描述的硬件可能有各种重复和冗余，而高层次综合能做的优化工作又比较有限，很多这种可能产生冗余的情况，在转化过程中又没法自动优化；其次，高层次综合不完全意味着开发人员可以按照高级编程语言的方式去进行开发，很多高级编程语言的特性是不被高层次综合支持的，例如系统调用，递归调用等等，这使得对它的使用并不像高级编程语言那样随心所欲。当然，虽然有着这些问题，高层次综合依旧在硬件开发中占据一席之地。

为了解决这些问题，有两种思路，一个是继续改进高层次综合，另一个就是重新造出一套类似的东西，这方面的工作也有一些开发团队进行了值得赞许的尝试，例如斯坦福大学的 Spatial，加州伯克利大学的 Chisel，它们都是硬件构造语言，也就是通过高级编程语言构造出硬件描述语言。而深度学习加速器堆栈 VTA 就是用硬件构造语言 Chisel

进行开发的。而我们这次的工作，就是在参考 VTA 的基础上，用团队自研的硬件描述语言 PyHCL，开发出我们自己的深度学习加速器堆栈 PyVTA。

在介绍了开发工具的发展历程，以及它们之间的内在联系之后，我们通过画图展示了深度学习编译堆栈 Apache TVM 层次结构 ,以及深度学习加速器堆栈 PyVTA 在 Apache TVM 当中的位置，并且简要描述了每一层的功能。然后还介绍了 PyVTA 的硬件架构，在这个架构下的不同模块是如何协同工作的，以及 PyVTA 的 JIT 运行时系统。

以上就是理论基础的部分，接下来大致展示一下对代码的测试和做出来的成果。

5.1 测试

既然是开发工作，那肯定需要一些测试。测试的思路就是，通过一些简单的测试样例，来测试各个模块的功能是否正常，以及各模块之间是否能正常协作。这次我们的测试主要包括了对一些重要功能的测试，例取指模块模块是否能够正常取址，加载模块是否能够把正确的指令加载到寄存器文件的相应位置，ALU 运算和 GEMM 运算是否能得到预期的结果，存储模块是否能够正常写入，等等，并且通过打印必要的信息来观察测试结果。如果测试通过，将会打印测试通过的信息。

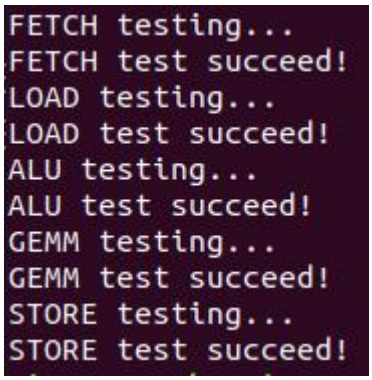

图 5-1 测试结果

上面是一些测试样例通过之后打印出来的信息，在这里我们对其中一个测试样例，GEMM test 的例子进行分析，因为计算模块显然是整个硬件架构当中最核心的模块，而 GEMM 矩阵运算相对于 ALU 向量运算来说难度会更大一些，所以我们选取最有代表性，也最容易出问题的 GEMM test 测试样例进行具体的分析。

5.1.1 GEMM test 的测试流程概括

GEMM test 测试样例稍微有点儿长，不过整个例子执行的流程还是比较清晰的，因此在对这个测试样例进行具体的分析前，还是先用文字和图片大致描述一下它的每个步骤。先看文字描述，首先是它根据配置文件，确定运行这个测试样例的宿主设备，有可能是虚拟机，也有可能是 FPGA 设备。然后它会做一些计算声明，包括主存上的一些变量，和片上缓存的一些变量，以及它们的对应关系等等。然后就会对上面所描述的计算声明进行编译，得到一个可执行文件并保存下来。等到用户在输入具体的矩阵进行预算的时候，只需要加载这个可执行文件，把矩阵的值当做输入参数就可以进行计算了，计算的结果和预期的结果进行比较，相等则测试通过，否则测试不通过。

我们也可以用流程图表示整个过程，如图 5-2。

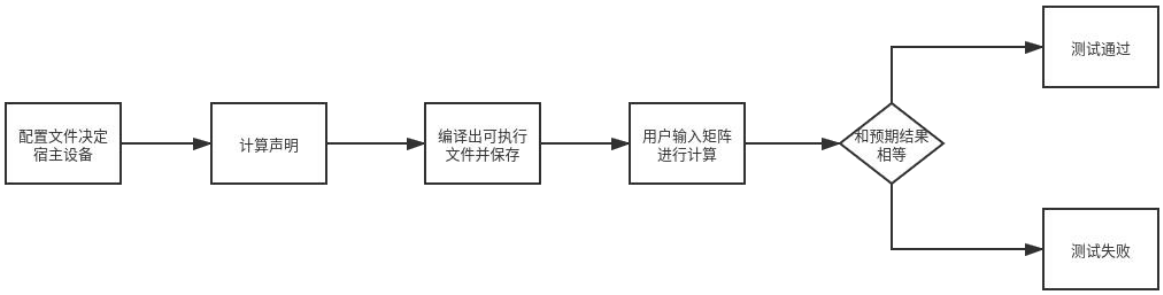

图 5-2 测试流程

5.1.2 根据配置文件确定宿主设备

解析配置文件，当配置文件的宿主设备这一项为 FPGA 时，执行图 5-3 代码。

```
from __future__ import absolute_import, print_function

import os
import tvm
from tvm import te
import pyvta
import numpy as np

env = pyvta.get_env()

from tvm import rpc
from tvm.contrib import utils
from pyvta.testing import simulator

host = os.environ.get("PYVTA_RPC_HOST", "192.168.2.99")
port = int(os.environ.get("PYVTA_RPC_PORT", "9091"))

if env.TARGET == "pynq" or env.TARGET == "de10nano":

    assert tvm.runtime.enabled("rpc")
    remote = rpc.connect(host, port)

    pyvta.reconfig_runtime(remote)

    pyvta.program_fpga(remote, bitstream=None)

elif env.TARGET in ("sim", "tsim", "intelfocl"):
    remote = rpc.LocalSession()

    if env.TARGET in ["intelfocl"]:
        pyvta.program_fpga(remote, bitstream="pyvta.bitstream")
```

图 5-3 确定宿主设备

5.1.3 计算声明

这是这个测试样例的关键部分，它描述了计算的一些元素，包括矩阵乘积的两个操作数矩阵，和结果矩阵。注意，是描述，而非真正去执行计算。执行计算的过程发生在用户调用它们的时候。它描述的是这样一个过程，见图 5-4，其中 A，B 是主存上的矩阵变量，它会被加载模块加载到缓冲区，并用 A_buf, B_buf 这两个缓冲区变量保存起来。运算模块会对这两个缓冲区变量进行矩阵运算，得到的结果用缓冲区变量 C_buf 保存。存储模块把缓冲区变量 C_buf 的值读取出来，写回到 DRAM 的矩阵变量 C。这就是整个计算声明的大致流程。

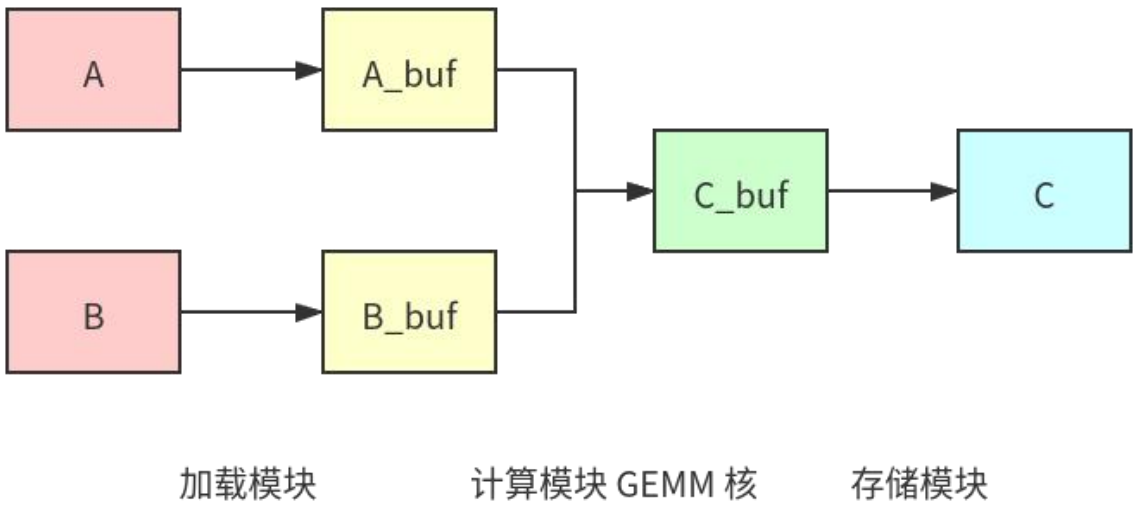

图 5-4 矩阵乘法过程和相关的变量

这么说可能有点难以理解，我们接下来进行进一步分析，首先看主存上的两个操作数矩阵，如图 5-5。

```
m = 16
n = 16
o = 1
A = te.placeholder((o, n, env.BATCH, env.BLOCK_IN), name="A", dtype=env.inp_dtype)
B = te.placeholder((m, n, env.BLOCK_OUT, env.BLOCK_IN), name="B", dtype=env.wgt_dtype)
```

图 5-5 主存上的操作数矩阵

这里的 A 和 B ,可以理解为主存上的矩阵变量 A 和 B ,其中 A 的形状为 (o * BATCH) * (n * BLOCK_IN)，B 的形状为 (n * BLOCK_IN) * (M * BLOCK_OUT) 。

接下来，声明 PyVTA 片上缓冲区的两个矩阵变量 A_buf 和 B_buf，如图 5-6。

```
A_buf = te.compute((o, n, env.BATCH, env.BLOCK_IN), lambda *i: A(*i), "A_buf")
B_buf = te.compute((m, n, env.BLOCK_OUT, env.BLOCK_IN), lambda *i: B(*i), "B_buf")
```

图 5-6 片上缓冲区的操作数矩阵

这里需要注意，虽然主存上的矩阵变量和缓冲区的矩阵变量，它们存储的矩阵都是一样的，但是这两个变量不能混淆，因为计算模块只能对缓冲区变量进行运算，所以需要用缓冲区变量把矩阵保存下来。

然后，就是声明缓冲区输出变量 C_buf 和主存输出变量 C，如图 5-7。

```
C_buf = te.compute(
    (o, m, env.BATCH, env.BLOCK_OUT),
    lambda bo, co, bi, ci: te.sum(
        A_buf[bo, ko, bi, ki].astype(env.acc_dtype) * B_buf[co, ko, ci, ki].astype(env.acc_dtype),
        axis=[ko, ki],
    ),
    name="C_buf",
)

C = te.compute(
    (o, m, env.BATCH, env.BLOCK_OUT), lambda *i: C_buf(*i).astype(env.inp_dtype), name="C"
)
```

图 5-7 缓冲区输出变量和主存输出变量

再次强调，这里不是说变量 C_buf 的值等于 A_buf * B_buf，事实上这些变量都没有初始化，这个语句只是描述了 C_buf 在运算的时候是用来保存 A_buf * B_buf 的值的这个事实。

还有一点值得注意的是，在 GEMM 运算核对矩阵进行乘法运算的时候，可能需要对矩阵的形状进行一个强制转换，因为 GEMM 运算核是要求矩阵具有一定形状的，这个形状由配置文件决定。这样可以减少 GEMM 运算核硬件设计的复杂度，相比较而言，把矩阵形状强制改变一下，并不是一件很困难的事情。当然，在运算结束后，还需要把结果强制转换回原来应该有的形状。也就是说，对于用户而言，如果不深入探究 GEMM 运算核的实现，那么这个强制改变矩阵形状的过程，用户是完全看不到的，这一切是发生在黑箱里的。

5.1.4 编译可执行文件

图 5-8 把这个功能模块编译为 gemm.o 文件。

```
my_gemm = pyvta.build(s, [A, B, C], "ext_dev", env.target_host, name="my_gemm")

temp = utils.tempdir()
my_gemm.save(temp.relpath("gemm.o"))

remote.upload(temp.relpath("gemm.o"))

f = remote.load_module("gemm.o")
```

图 5-8 编译成可执行文件

5.1.5 计算并验证结果

现在输入两个矩阵 A_orig 和 B_orig，形状分别为 (o * BATCH) * (n * BLOCK_IN)，(n * BLOCK_IN) * (M * BLOCK_OUT) ，矩阵的每个元素是从 [-128, 128) 的随机数。然后分别通过 PyVTA 的矩阵运算，和 Python 自带的矩阵运算，得到两个结果 C_nd 和 C_ref 进行比较，验证 PyVTA 得到的矩阵运算结果是否正确。

```
ctx = remote.ext_dev(0)

A_orig = np.random.randint(-128, 128, size=(o * env.BATCH, n * env.BLOCK_IN)).astype(A.dtype)
B_orig = np.random.randint(-128, 128, size=(m * env.BLOCK_OUT, n * env.BLOCK_IN)).astype(B.dtype)

A_packed = A_orig.reshape(o, env.BATCH, n, env.BLOCK_IN).transpose((0, 2, 1, 3))
B_packed = B_orig.reshape(m, env.BLOCK_OUT, n, env.BLOCK_IN).transpose((0, 2, 1, 3))

A_nd = tvm.nd.array(A_packed, ctx)
B_nd = tvm.nd.array(B_packed, ctx)
C_nd = tvm.nd.array(np.zeros((o, m, env.BATCH, env.BLOCK_OUT)).astype(C.dtype), ctx)

f(A_nd, B_nd, C_nd)

C_ref = (A_orig.astype(env.acc_dtype) + B_orig.astype(env.acc_dtype)).astype(C.dtype)
C_ref = C_ref.reshape(o, env.BATCH, m, env.BLOCK_OUT).transpose((0, 2, 1, 3))
np.testing.assert_equal(C_ref, C_nd.asnumpy())
```

图 5-9 矩阵运算并验证结果

5.2 调试工具 GDB

而在测试的过程中，如果发现问题，在排查的过程中，可能要用到 GDB 调试工具，通过设置断点，单步执行，查看寄存器状态等手段来进行调试。在很难根据代码来推测底层硬件元件发生了什么事情的时候，通过 GDB 调试工具查看寄存器状态，有时候可能是一种行之有效的手段，所以在这里也展示一下如何查看寄存器的状态，以及能看出来什么信息，如图 5-10。

```
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 1, 0x0010000c in ?? ()
(gdb) info registers
eax            0x0      0
ecx            0x0      0
edx            0x0      0
ebx            0x10094  65684
esp            0x7bec   0x7bec
ebp            0x7bf8   0x7bf8
esi            0x10094  65684
edi            0x0      0
eip            0x10000c 0x10000c
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
```

图 5-10 GDB 调试结果

上图 5-10 是在虚地址 0x10000c 处设置断点，在程序执行到这里而暂停的时候，查看当前所有寄存器的值。

事实上，在整个深度学习加速器堆栈的开发过程中，细化到查看寄存器状态的调试的场合并不多，不过既然在某些场合下有用，还是值得学习的。

5.3 结果展示

要展示我们这次做出来的深度学习加速器堆栈 PyVTA 的成果，要从两个方面来考察，一个是在深度学习加速器堆栈上模型的正确性，举个例子，在深度学习加速器堆栈上，用一些分类模型实现图片分类的任务，最后会得到一个准确率，那么我们就要考察，

应用深度学习加速器堆栈，是否会对准确率造成不可忽视的影响，至少不能让深度学习加速器堆栈大幅拉低正确率；另一个考察角度就是深度学习加速器堆栈是否真的实现了模型运算时间的加速。

为了从这两个角度来考察，我们从 ImageNet 中获取数据集。ImageNet 是一个图像数据库，目的是为了给深度学习以及相关领域的从业人员提供训练集。我们对这个数据集，用常见的几种分类模型，在没有深度学习加速器堆栈，或应用在不同的深度学习加速器堆栈上的情况下，统计它们的分类正确率，以及耗时，结果可以用下面图 5-11，图 5-12 展示。

模型	无加速	VTA	PyVTA
VGG-16	27.69	28.43	28.75
PReLU-net	23.02	23.59	23.86
plain-34	28.06	28.87	29.12
ResNet-34	24.38	24.84	25.34
ResNet-152	21.44	21.76	21.78

图 5-11 几种分类模型在深度学习加速器堆栈上的正确率（单位 ：%）

模型	无加速	VTA	PyVTA
VGG-16	422	213	246
PReLU-net	621	435	488
plain-34	436	264	289
ResNet-34	674	496	523
ResNet-152	823	549	584

图 5-12 几种分类模型在深度学习加速器堆栈上的耗时（单位 ：毫秒）

从图 5-11 和图 5-12 中可以看出以下信息，首先是在列出的这几种分类模型中，引入深度学习加速器堆栈都会对分类的正确率有一点儿负面影响，不过这种影响在可接受的范围内。其次，我们自己开发的深度学习加速器堆栈 PyVTA，比它的“前辈” VTA，对分类正确率的负面影响稍大，不过也是很小的差别。还有，两种深度学习加速器都对这几种模型的算力有一定程度的加强，它们的运算速度的确得到了提高，其中 VTA 的加速效果还是比我们自己开发的 PyVTA 的效果要好一些。

综合从表中获取到的这些信息，我们可以认为，无论是 VTA 还是 PyVTA，都对一些深度学习模型有一定的加速效果，且不会对模型本身的正确性有太大的影响，而且 PyVTA 在加速效果方面暂时还比不上它的前辈，因此还有改进的空间。

5.4 本章小结

本章首简单说明了一下测试的一些手段，然后具体说明了一个具有代表性的测试的例子，最后展示了一下在深度学习加速器堆栈上运行深度学习模型的效果，作为这次实践工作的成果。

结论

1 总结

本论文的研究基础是，基于加州伯克利大学实现的深度学习加速器堆栈 VTA，利用团队自研的硬件构造语言 PyHCL，实现一个深度学习加速器堆栈 PyVTA。本文先介绍了在硬件构造语言之前出现的两种用于描述硬件的语言，分别是硬件描述语言(HDL)和高层次综合（ HLS ），然后指出了它们解决了的一些问题，以及自身的一些局限性。然后，针对它们的局限性，我们引入了硬件构造语言，其中有加州伯克利研发的已经比较成熟的 Chisel，和我们自研的 PyHCL。接下来，我们使用 PyHCL，参考 Chisel 实现的 VTA，实现 PyVTA。接下来介绍了 PyVTA 的设计，其中比较重要的是硬件架构，因此我们花了一些篇幅来介绍硬件架构当中的各个模块，以及它们之间如何协同作业。然后展示了我们在实现过程中遇到的一些问题，包括 PyHCL 尚未完善的地方，以及我们的一些解决办法，并且演示了一下如何有效地对各个模块的功能进行测试。最后，我们在深度学习加速器堆栈上运行了一些分类模型，并且得到了不错的效果。

2 未来工作展望

之前已经演示过深度学习模型在深度学习加速器堆栈上的运行表现，从这个结果我们就可以看出来，我们自己开发的 PyVTA，在加速效果上还比不上它的前辈 VTA，这可能是语言特性的原因，也可能是对深度学习加速器堆栈设计得还有不足的原因。这些都是值得做进一步探究的。

还有一个值得研究的方面就是，PyHCL 目前仍然是一个不太完善的工具，很多功能都是没有实现的，例如在开发工作中提到的，Bundle 对输入输出端口的支持，还有

Decoupled 这些功能，都是在这次开发过程中，发现缺少了，才添加上去的，而这些功能还没有经过很严谨的测试，只是说在这次开发过程中，没有表现出比较大的问题。因此，从可持续的角度来说，PyHCL 还是需要相关人员去共同维护的。

参考文献

[1] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm et al., “A hardware–software blueprint for flexible deep learning specialization,” IEEE Micro, vol. 39, no. 5, pp. 8–16, Sep. 2019.

[2] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," DAC Design Automation Conference 2012, San Francisco, CA, USA, 2012, pp. 1212-1221, doi: 10.1145/2228360.2228584.

[3] J. Keeney and V. Cahill, "Chisel: a policy-driven, context-aware, dynamic adaptation framework," Proceedings POLICY 2003. IEEE 4th International Workshop on Policies for Distributed Systems and Networks, Lake Como, Italy, 2003, pp. 3-14, doi: 10.1109/POLICY.2003.1206953.

[4] M. Gordon, "The semantic challenge of Verilog HDL," Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science, San Deigo, CA, USA, 1995, pp. 136-145, doi: 10.1109/LICS.1995.523251.

[5] R. Nikhil, "Bluespec System Verilog: efficient, correct RTL from high level specifications," Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04., San Diego, CA, USA, 2004, pp. 69-70, doi: 10.1109/MEMCOD.2004.1459818.

[6] V. Sieh, O. Tschache and F. Balbach, "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions," Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing, Seattle, WA, USA, 1997, pp. 32-36, doi: 10.1109/FTCS.1997.614074.

[7] Izraelevitz, A., Koenig, J., Li, P., Lin, R., Wang, A., Magyar, A., ... & Bachrach, J. (2017, November). Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In Proceedings of the 36th International Conference on Computer-Aided Design (pp. 209-216). IEEE Press.

[8] Hennessy, J. L., & Patterson, D. A. (2019). A new golden age for computer architecture.

Communications of the ACM, 62(2), 48-60.

[9] M. Shaw and P. Clements, "The golden age of software architecture," in IEEE Software, vol. 23, no. 2, pp. 31-39, March-April 2006, doi: 10.1109/MS.2006.58.

[10] J. L. Henning, "SPEC CPU2000: measuring CPU performance in the New Millennium," in Computer, vol. 33, no. 7, pp. 28-35, July 2000, doi: 10.1109/2.869367.

致谢

首先感谢我的本科导师赖晓铮老师,在毕设期间给了我充分的指导,从选题的确定，到项目工程的开发，到论文的撰写和修改，导师都帮我解答了很多困惑，有在选题大方向上给我的引导，也有对项目代码和论文上一些细节上和我的讨论。还有很重要的一点是，导师这次和我经过讨论后确定下来的题目，是一个以前我没怎么接触过的领域，身边的本科计算机同学，以后如果还走计算机这条路的话，大部分都会选择算法或开发这两条路，而这次毕业设计的方向是 AI 加速，通过这次毕业设计，我得以初步接触到这个领域，从长远的角度看，这是一个拓宽眼界的难得的机会。非常感谢导师在这次毕业设计中给我的指导。

然后,非常感谢大学四年来身边的这些人,从刚进大学认识的舍友郭彬涛,李林昊，李安，还有其他一些因为职业规划相似而认识的同学，在职业规划这方面，首先要非常感谢郭彬涛，对我现在的职业选择有着不可忽视的正面影响，计算机各方面的内容，从专业基础知识，包括操作系统，计算机网络，数据库，C/cpp 等等，到 leetcode，很多都是我在没有目标的时候，看到有人这么做，我也就跟着做了，虽然也会走弯路，不过至少不会闲下来消磨时间。还有就是在几次团队作业中认识的范滔和钟镇炽，当时我处在要投暑期实习简历的时间节点，能明显感觉到自己的知识非常不成体系，然后也不知道怎么去建立一个属于自己的知识体系,应该先学什么后学什么,怎样去学。这样的疑惑，我觉得至少大半以上的技术人员都会经历过。而他们让我意识到，应该看这些书，这些书应该这样看，国内外有这些课和配套实验，它们应该这样用。总之，我现在对自己的职业规划还是有一定的方向的，这其中也经历过很多迷茫的时候，也试过在大三才意识到自己可能不适合往算法方向发展，从而忍痛转方向。对于这些，我觉得，一个人所经

历的一切，才造就了现在的他，无论是吃过的亏还是正确的选择，都是不可抹去的。而当身边有一些人在认真做自己规划好的事情的时候，这个人自己也会觉得，没有那么多可迷茫的，大家都这么做的，我一样也可以做到。所以，非常感谢身边遇到的这些人。

最后，感谢家人，他们一直都是我最坚强的后盾，往近了说，转正失利和秋招处处受挫，当时自己也处于比较焦虑的状态，而家里人从小到大给我的想法都是，事情总是有办法的，哪怕结果不那么如意，也不意味着一切都结束了，除了生死，没有什么事情是过不去的。也正是因为从小到大都被这样的想法教育着长大，我才能够去面对人生中大大小小的困难。所以非常感谢家里人二十多年来对我的塑造。

愿自己成为一个合格的技术人员。愿自己少一些急功近利，多一些耐心和沉淀。