****

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](javascript:showViewModel(1,'1346832541109460992'))**

|  |  |
| --- | --- |
| **学院**  **专业**  **学生姓名**  **学生学号**  **指导教师**  **提交日期** | **计算机科学与工程** |
| **计算机科学与技术** |
| **陈炫勋** |
| **201730613014** |
| **赖晓铮** |
| **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在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 VTA，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 VTA, 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

# 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

本文一共分为四章。

## 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。

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。

第五章展示了一些在开发过程中遇到的问题，以及一些解决的思路，和一些测试的结果，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 VHDL，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 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。

# 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，在这里用一张图来表示这种堆栈结构，并且对每一层结构做一些说明。

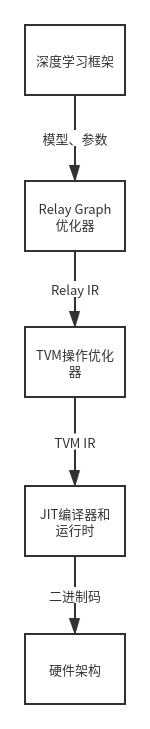


图 4-1 深度学习加速器堆栈 PyVTA 堆栈结构概览

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的硬件架构可以用一张图简略描述。

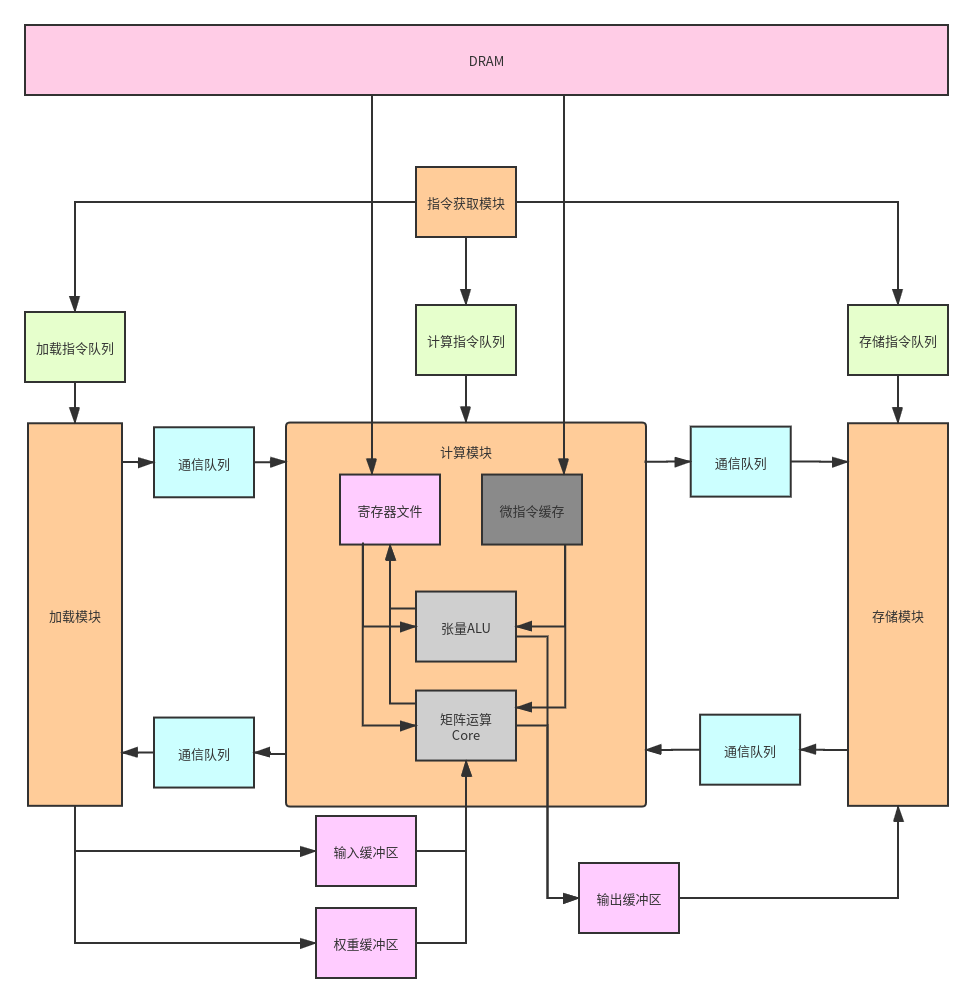


图 4-2 深度学习加速器堆栈 PyVTA 硬件架构概览

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。

4.2.1 对硬件架构的模块的介绍

从这张架构图可以看出，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，它们的格式可以用一张图表示。

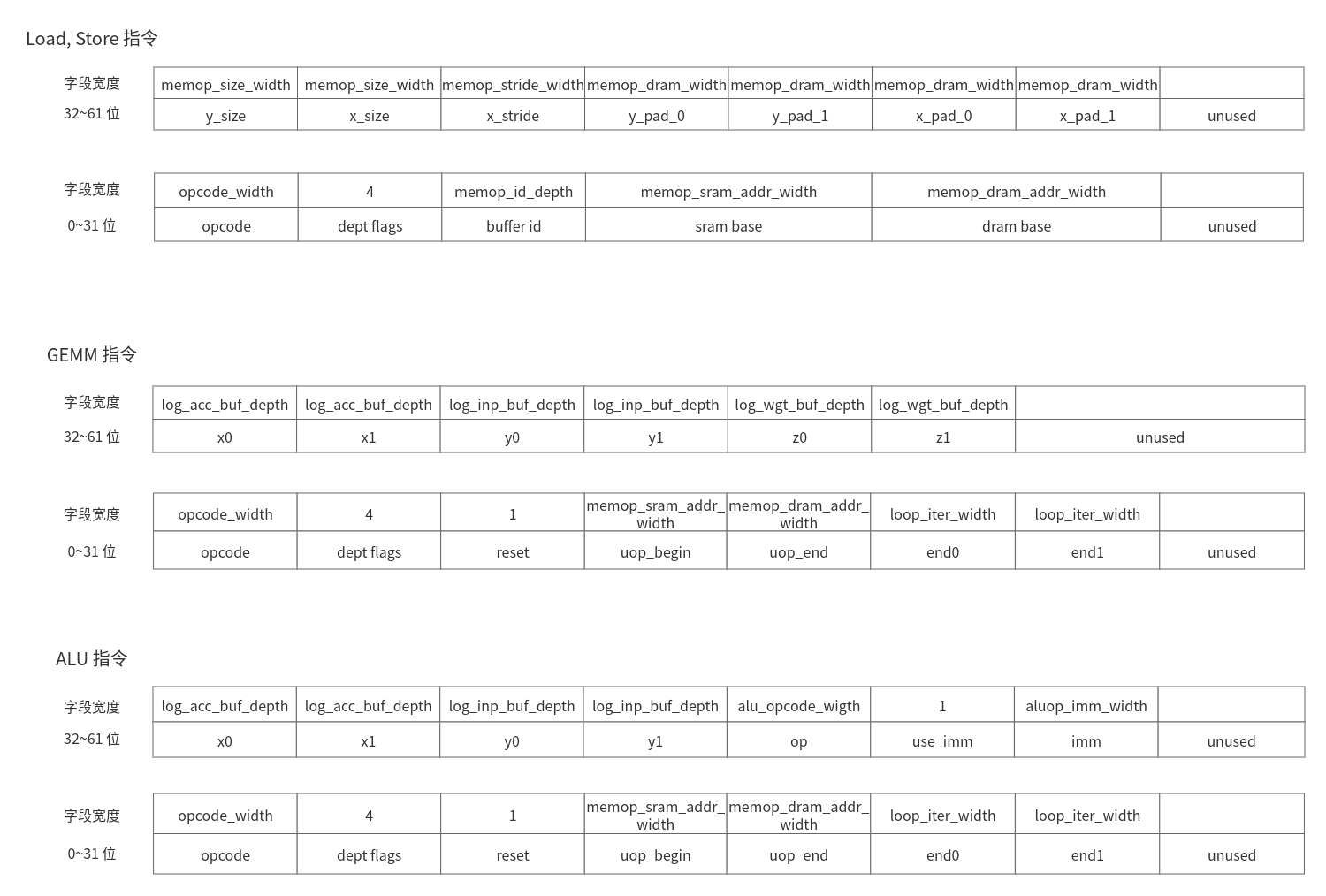


图 4-3 指令格式

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，可以用一张图表示。

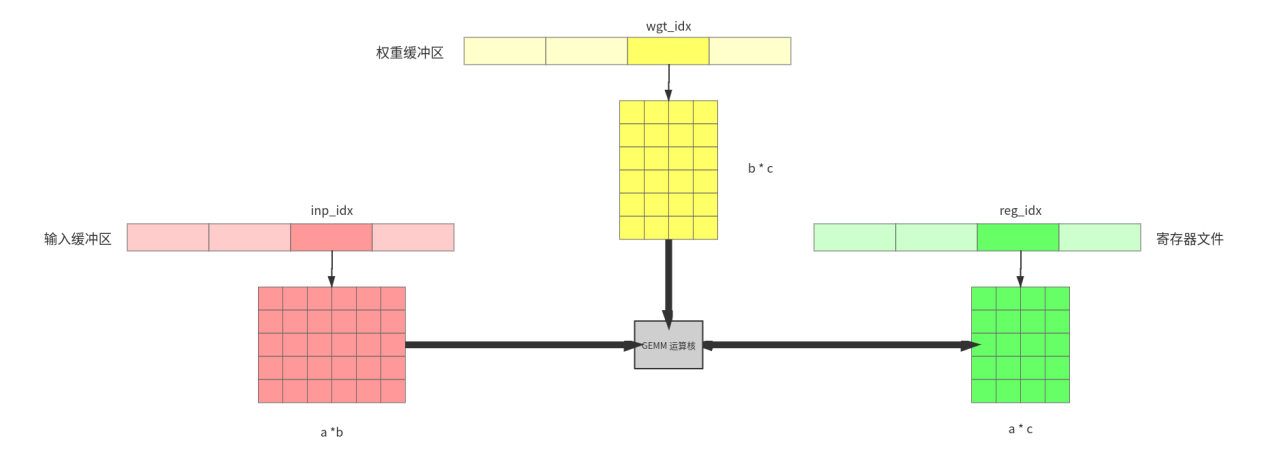


图 4-4 矩阵乘积运算在硬件层面的实现

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，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。

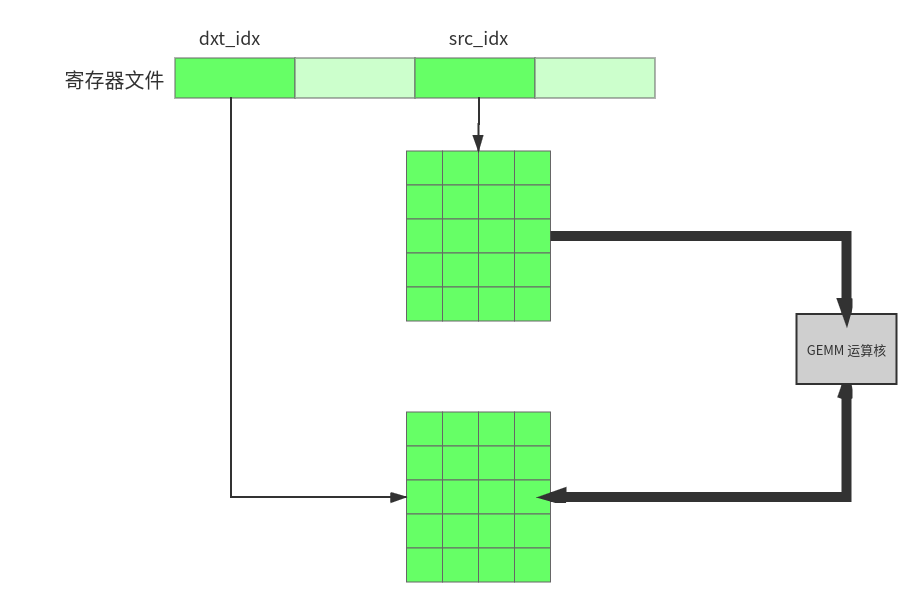


图 4-5 向量 ALU 运算在硬件层面的实现

## 4.3 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。

## 4.4 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，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 代码工作和成果演示

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的开发工作和成果。

5.1 开发工作

在开发自研的深度学习加速器堆栈PyVTA的过程中，也遇到了一些在这次开发中很典型的问题，其中很大一部分是语言特性的问题，也就是说，硬件描述语言Chisel能实现的功能，我们团队自研的硬件描述语言PyHCL能否实现等价的功能。解决了这些问题，开发工作其实就没有太大难度了。因为PyVTA代码的每个文件，结构都是大同小异的，都是描述了硬件架构的某个模块或功能，所以涉及到的语言表示的问题都是类似的，例如状态机如何表示，I/O顿口如何声明等等。因此在开发这个文件时遇到的问题，在其它文件那里也很可能遇到。因此，下面会列出来这些问题，并且展示解决办法，还有想出来这些办法的思路，并且和Chisel实现的VTA进行比较，展示PyHCL如何实现了Chisel已经实现的功能。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。

### 5.1.1 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，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。

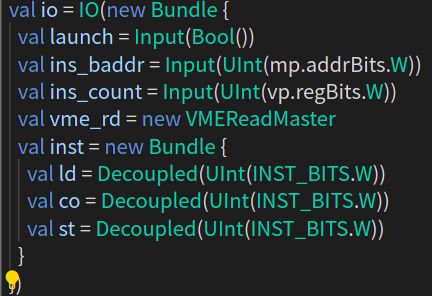


图 5-1 Chisel 表示一簇 I/O端口

从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会在后面的4.2.2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，具体的实现代码可以是这样的。

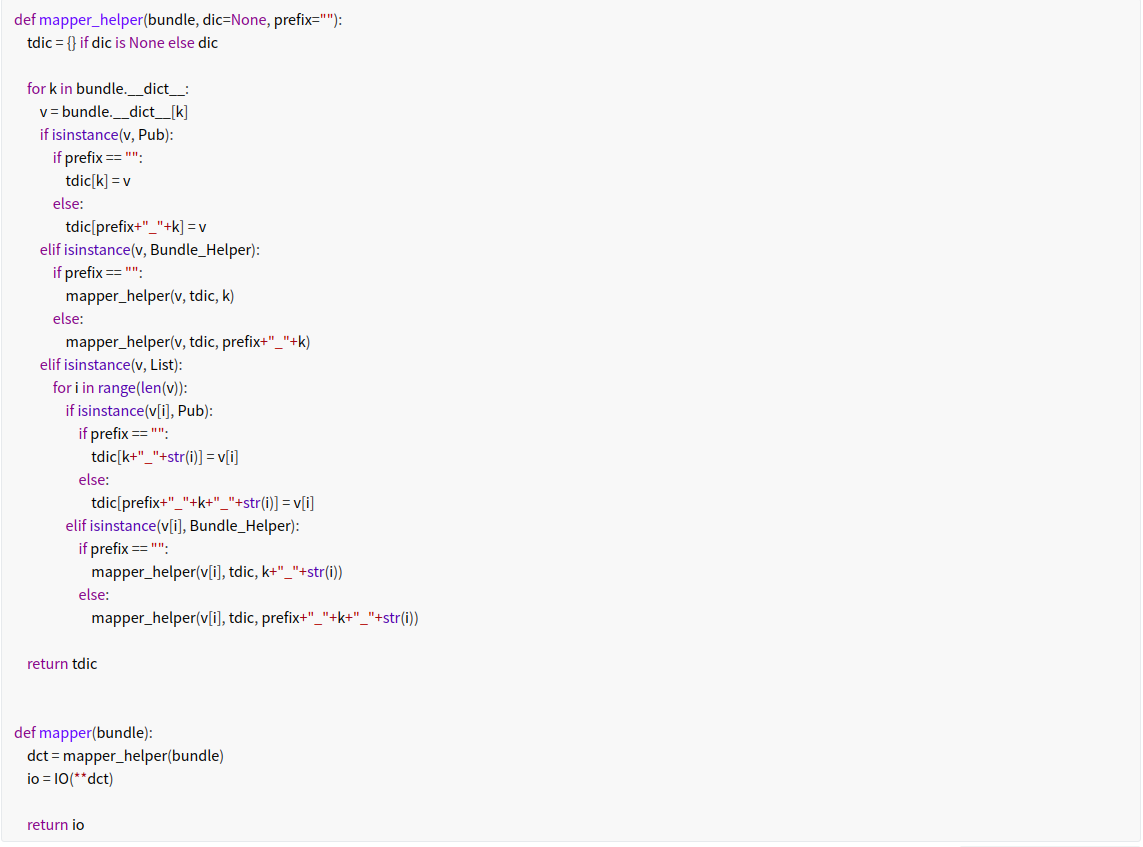


图 5-2 PyHCL 平铺 Bundle 成员

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。

这就是展开函数的实现，利用这个函数，就可以对Bundle类型进行展开了。

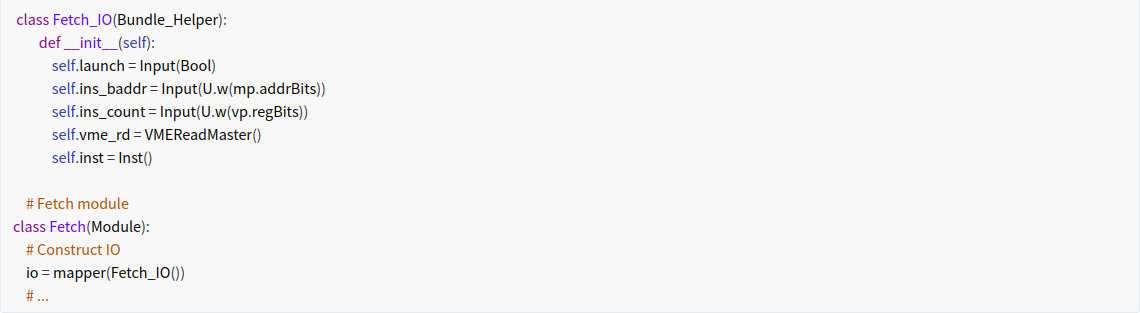


图 5-3 通过平铺实现一簇 I/O 端口

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

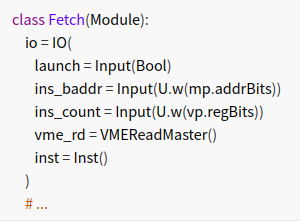


图 5-4 PyHCL 改进后的 Bundle 实现一簇端口

以上就是对Bundle类型的I/O端口的处理。

### 5.1.2 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踢了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，代码可以按照如下方式来写。

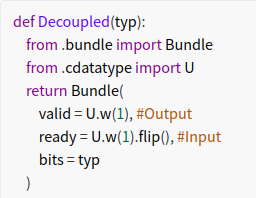


图 5-5 PyHCL 实现 Decoupled

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4.2.1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。具体实现参考一下代码。

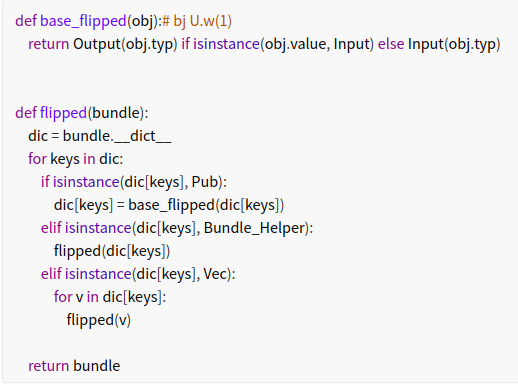


图 5-6 PyHCL 实现 filpped

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以下代码。

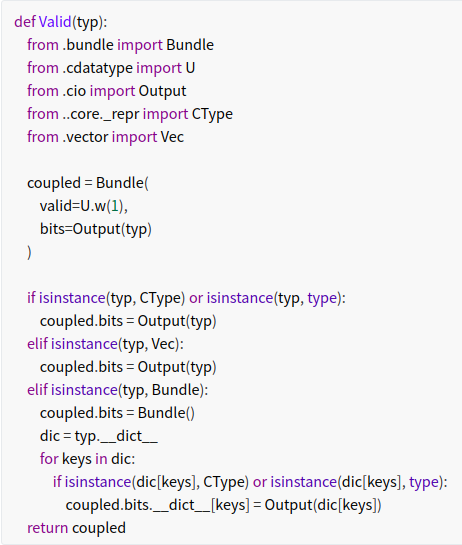


图 5-7 PyHCL 实现 Valid

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。

### 5.1.3 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VCR.scala为例，看一下Chisel是如何描述一个状态机的。

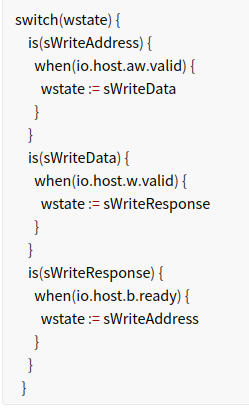


图 5-8 Chisel 表示状态机

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。

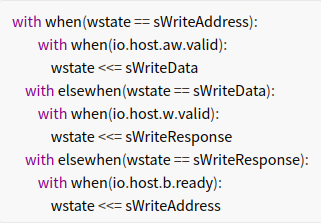


图 5-9 PyHCL 表示状态机

通过withwhen来对情况的分支进行判断。

以上几节，简单列出来了一些代码开发工作中遇到的问题，以及想出解决办法的思路，还有最后采用的解决办法。

## 5.2 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。

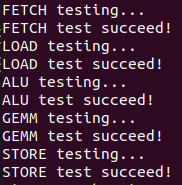


图 5-10 测试结果

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。

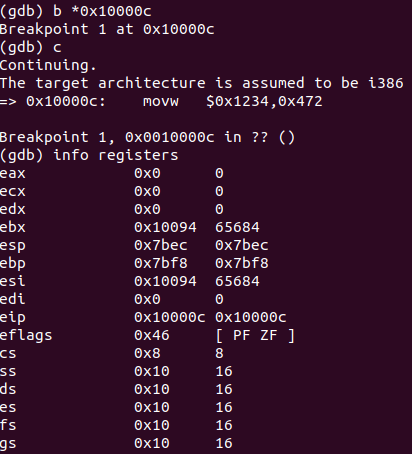


图 5-11 GDB 调试结果

上图是在虚地址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，结果可以用下面几张表展示。



图 5-12 几种分类模型在深度学习加速器堆栈上的正确率（单位 ：%）



图 5-13 几种分类模型在深度学习加速器堆栈上的耗时（单位 ：毫秒）

从这两张表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

本章首先介绍了在代码实现的过程中遇到的一些具体问题，以及解决这些问题的一些想法和最后的取舍，然后简单说明了一下测试的一些手段，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。

第六章 结论

6.1 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。

## 6.2 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。