# **课题任务，目的和意义**

首先看一下我们这次的课题究竟做了什么事情。我们在参考 Chisel 实现的可编程深度学习加速器堆栈 VTA 的基础上，采用了自研的硬件构造语言 PyHCL，来实现这个深度学习加速器堆栈，并且起名为 PyVTA。

在说明我们的工作之前，首先介绍一下深度学习加速器堆栈。它用于加速深度学习模型，解决当前深度学习模型运行速度太慢的问题。然而当前深度学习加速器都是专用的，也就是适用于某些特定的硬件资源。为了适配业界各种各样的硬件资源，一些研究人员也做出了各种努力，其中取得了一定成效的有加州伯克利大学 UCB 研发出来的可编程深度学习加速器堆栈 VTA，这是用它们自己的硬件构造语言 Chisel 实现的。而 Chisel 可以视为 Scala 的第三方库，具有一定门槛。为了降低开发人员的门槛，我们决定用我们自研的硬件构造语言 PyHCL 来实现加速器，在这里暂且称呼它为 PyVTA。当然，除了降低可读性的门槛之外，我们这次的工作也是为了给我们自研的 PyHCL 创造更好的语言生态，毕竟有人用的东西才能体现它存在的意义，并且我们会在使用它的过程中，发现并完善它存在的一些问题。

# **原始材料/参考文献**

这次毕业设计参考的主要是这篇论文，包括整个深度学习加速器堆栈的设计也是参考了这篇论文提到的设计。

# **堆栈结构**

首先，要介绍整个深度学习编译堆栈 TVM 的层次结构。

最上面一层是深度学习框架，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，在这一层我们用代码实现各种深度学习算法。

Relay Graph Optimizer，它接收深度学习框架表示的算法模型和参数，并且把它转化成 Relay。Relay 可以理解为一种编程语言，它表示的是计算图。

TVM Operator Optimizer。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，由此可见，PyVTA 被集成到了整个 TVM 编译堆栈中。

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。

# **硬件架构**

这是PyVTA的硬件架构。

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，用橙色标记：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。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缓冲区，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），也可能是向量的 ALU 运算（Tensor ALU），如果是向量的 ALU 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。

# **指令编码**

所有的代码都会被编译汇编成若干条指令，这些指令都是二进制码，它们的格式可以用这个图表示。

在这里主要关注 opcode 字段，指令通过 opcide 字段来区分指令类型。

# **GEMM 运算**

接下来我们关注一下主要的工作负载，矩阵乘积运算和向量加法运算。

首先是 GEMM 运算，从这张图可以看出，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矩阵，在这个例子中，输入矩阵为 5\*6，权重矩阵为6\*4，并进行矩阵乘积运算，然后把运算的结果写到寄存器文件，在这里得到的矩阵为6\*4。而对缓冲区的寻址，是通过输入索引 inp\_idx，权重索引 wgt\_idx 进行的，写回寄存器文件的位置是根据 reg\_idx 得到的。

# **ALU 运算**

ALU 运算，如图，本质上是向量之间的相加运算，运算的两个操作数分别为在寄存器文件中，下标为 src\_idx, dst\_idx 的寄存器值，在这里把它们称呼为源操作数和目的操作数。从寄存器文件中取出来这两个操作数做相加运算之后，把结果写回索引为 dst\_idx 的寄存器。

# **流水线设计**

引入流水线的目的是为了增大整个系统的吞吐量，也就是单位时间内处理任务的数量。而能引入流水线的前提是，系统的一些模块有闲置的时候。例如在这个系统中，如果是非流水线执行任务，会像这样。

假设任务t 是一个矩阵乘法运算，这几个模块会这样执行这个任务。加载模块从DRAM取出要运算的矩阵写入缓冲区，计算模块从缓冲区取出矩阵进行运算并写到缓冲区，存储模块再从缓冲区取出结果，写回DRAM。

这几个模块都由闲置的时候，因此我们考虑把这些闲置的时间利用起来，可以设计这样的流水线。

这是最理想的状态了，每个模块处理它那部分任务的时间都是一样的，因此所有闲置的时间都得到了利用。而实际上，情况很有可能是这样的。

矩阵运算和向量ALU运算往往是耗时的瓶颈，这样就导致了有些时间段的闲置。当然这个似乎也没什么很好的办法，计算本来就是性能的瓶颈，只能想办法尽量提高算力。

# **测试：矩阵相乘**

我们通过一些简单的测试，来验证一下它的功能。这里展示一下矩阵相乘的测试结果，通过深度学习加速器堆栈得到的结果是符合预期的。而矩阵相乘在硬件层面的执行流程，在之前介绍矩阵运算的时候已经提及。

# **结果：分类速度和正确率**

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，加速器是否会对模型的正确率有太大影响？另一个考察角度就是深度学习加速器堆栈是否真的缩短了模型运行的事件。

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，结果可以用下面两张图展示。

从两张图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。

# **总结**

基于 VTA，用团队自研的 PyHCL 实现了 PyVTA，并且大致保留了原有的功能

通过了一些例程的测试，包括简单的矩阵和向量运算，也能运行一些深度学习模型

# **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。

问题：

通用化在哪里？？？ 怎么展示 PYHCL的工作，写在论文里了还要不要展示？？？ 流水线，寄存器状态读取有没有？？？ 结果不是自己做的，要不要提前说明？？？ GEMM 三层循环要不要讲？？？