**二值复数神经网络（BCNN）协处理器**



目 录

二值复数神经网络（BCNN）协处理器 I

1 绪论 1

1.1 研究背景与意义 1

1.2 研究现状 6

1.3 本文主要研究内容 17

1.4 本文的章节安排 19

2 CNN与BCNN算法概述 21

2.1 引言 21

2.2 卷积神经网络及其硬件加速原理 21

2.3 BCNN算法概述 28

2.4 本章小结 35

3 面向BCNN算法的协处理器设计 36

3.1 引言 36

3.2 协处理器设计架构 37

3.3 指令集（ISA）设计 41

3.4 SATU设计 42

3.5 内存设计及数据存储方式 54

3.6 本章小结 59

4 BCNN专用加速器的系统设计 60

4.1 引言 60

4.2 开源处理器Rocket概述 61

4.3 系统硬件架构设计 62

4.4 系统软件架构设计 66

4.5 本章小结 69

5 仿真与性能评估 70

5.1 引言 70

5.2 功能仿真 71

5.3 性能评估 76

5.4 本章小结 79

6 总结与展望 81

6.1 总结 81

6.2 展望 82

致 谢 84

参考文献 85

附录1 攻读硕士学位期间取得的学术成果 89

附录2 部分程序代码 90







# 面向BCNN算法的协处理器设计

## 协处理器设计架构

根据上述的AI芯片的设计流程，在确定了神经网络架构和算法后，就需要对加速器的硬件架构进行设计。芯片架构的设计就是将神经网络算法映射到硬件设计中，主要包括计算引擎设计和存储系统设计两个部分，往下又可划分为多个任务，例如计算单元的设计与优化、计算单元阵列以及网络拓扑的设计与优化、高效合理的存储系统设计、充分利用数据并行性和可重用性以及确定数据流（权重固定或输出固定等）等等。

经过第二章的铺垫后，我们已经对BCNN算法的架构和理论基础有了一定的了解。所以本节就针对BCNN算法和架构和特点，并借鉴了众多了现存的深度学习加速器，提出了一种新型的面向BCNN的专用加速器的系统架构。

### 设计架构

本次设计的BCNN加速器的协处理器采用了软硬件协同的设计思想，主要包括计算引擎、存储系统、数据通路以及各控制模块的设计。下图为本次设计的加速器的协处理器的整体架构框图，该协处理器的架构参考了Thierry等人设计的Versatile Tensor Accelerator（VTA）[60]，采用了访问执行解耦合（Decoupled Access/Execute）架构，这就意味着“内存访问”和“执行”指令可以在硬件的不同区域同时执行，该机制可以极大地提高指令执行效率。我们把硬件分解为三个控制模块，分别是负责处理“excute”指令的Execute Controller模块、负责处理“load”指令的Load Controller模块和负责处理“store"指令的Store Controller模块。



图 面向BCNN的深度学习加速器的协处理器的架构框图

除了这三个控制模块，本次设计的架构也有与VTA架构的不同之处，例如VTA架构中使用一个取指令模块（Instruction Fetch Module），直接与片外DRAM交互并取指令，这样的设计虽然简洁方便，但直接截断了与CPU集成的可能，无法将协处理器在边缘智能设备中实际应用。为此我们引用并设计了保留站（Reservation Station）模块，保留站技术是计算机科学中常用技术，主要用于解决连续进入指令流水线的指令之间的数据和资源的相关性问题。我们引入的保留站模块可以同时缓存多条指令，并监测不同类型的指令是否存在资源冲突，一旦检测到某一条指令与其他控制器中的指令间没有依赖关系，就将其送往对应的指令队列中，解决了相关性问题；除此之外，本次设计的保留站还含有RoCC接口，便于后续协处理器与主处理器的集成，极大提高了硬件的可实用性。

与VTA架构另一个不同之处在于，VTA架构中忽略了片上存储SRAM的使用，直接在DRAM进行数据的访存，极大的降低了硬件的执行效率且增加了功耗，而我们为了提高计算的执行效率和数据的复用性，在架构中添加了片上存储SRAM，其中包括简单的单端口SRAM组成的Databuffer、和双端口SRAM构成且拥有累加功能的Accumulator。且设计针对的BCNN算法，将每个像素的权重参数和输入数据压缩至2比特（实部和虚部各1比特），极大缩小了对存储空间的需求，在实现高吞吐量的同时降低了成本和功耗，也为我们将激活和权重数据迁移至片上SRAM存储提供了理论依据。

上文中我们对比了本次设计与VTA架构的两点不同，而在指令保留站完成对指令的预处理并将各指令按类别（如数据Load、数据Store、计算执行）分配到相应的队列后，各个模块会对队列中指令依次进行译码和执行，这与VTA架构是相同的。下面我们分别简要地介绍三个控制模块（Execute Controller模块、Load Controller模块、Store Controller模块）的结构和功能。

1. Execute Controller模块

该部分主要负责执行“execute”类型的ISA指令，例如矩阵乘法。该模块主要包含脉动阵列阈值单元（Systolic Array Threshold Unit，SATU）和循环展开单元（Loop Unroller Unit，LUU），其中循环展开单元用于处理长指令（CISC）并将其展开为多条精简指令（RISC），便于后续的计算执行；脉动阵列阈值单元作为本次设计的核心，参考自TPU中的脉动阵列和FINN中的矩阵向量阈值单元（MVTU），可同时完成DIM\*DIM个元素的点积运算（其中DIM指脉动阵列中每行或者每列计算单元的个数），且其中各个计算单元采用了简单的popcount(xnor)计算代替了复杂的乘加计算，极大地提高了数据的并行计算能力和硬件友好性，进而提高了计算效率；其中阈值单元使用DIM个比较器即可完成复杂的BN和二值化的过程，极大地降低了硬件开销。

虽然Execute Controller模块的功能比较复杂，但是其状态机十分简洁，只有“等待指令”、“preload数据”和“计算”三个状态。状态机的默认状态为“等待指令”状态，在收到数据预取指令后进入“preload数据”状态并预取数据，等待数据预取完成后又返回“等待指令”状态；再收到数据计算指令后，进入“计算”状态，通过SATU单元完成卷积计算和二值化过程。状态机设计之所以简单，是因为我们将复杂的逻辑功能直接映射成硬件电路。

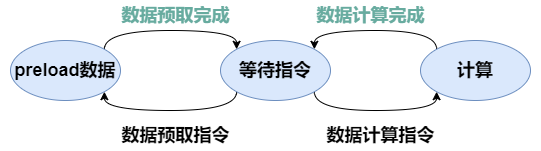


图 ExecuteController模块状态机

1. LoadController模块

该部分负责执行从DRAM向协处理器的Databuffer或者accumulator传输数据的指令。该模块主要包含一个DMA和数据传输的控制寄存器（如读取数据的行数、列数、步长、源地址、目的地址等）。特别指出，DMA可以对虚拟地址进行操作，从指令保留站中获取地址（虚拟地址）后会将其送入高速缓存（Translation Lookaside Buffer，TLB），经过映射得到DRAM的物理地址。如果在TLB未命中后，该虚拟地址将直接转回主处理器中的页表查找模块（Page Table Walker，PTW）来获取物理地址。

LoadController模块的功能也是由状态机控制，状态机的默认状态是“等待指令”状态，在该状态下可以完成对控制寄存器的配置；当存储器准备就绪后，如果DMA未有请求任务则进入“等待DMA请求”状态，如果DMA有请求任务则直接进入“load数据”状态，并通过DMA将DRAM中源地址对应的数据送到本地存储（Databuffer或者Accumulator）中目的地址对应的位置。直至最后一行数据传输完成，则返回“等待指令”状态。

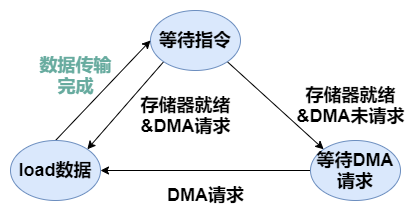


图 LoadController模块状态机

1. StoreController模块

该部分负责执行从协处理器的Databuffer或者Accumulator向DRAM传输数据的指令。结构和状态机都与LoadController模块类似，这里不再赘述。

## 指令集（ISA）设计

指令集指的是计算机中央处理器机器码所使用的指令的集合以及其背后的寄存器体系、总线设计等逻辑框架，作为软件设计和硬件设计的基础和纽带，指令集架构在整个设计过程中的地位可谓是举足轻重。常见的指令集架构大体可分为复杂指令集（CISC）和精简指令集（RISC）[71]，本次协处理器设计中涉及的指令集是依据RISC-V指令集架构中的扩展指令集设计的，且利用了单指令多数据（SIMD）的形式，主要包括三个类型的指令：数据移动指令、配置指令、计算指令。

### 数据移动指令

数据移动指令主要分为“mvin”和“mvout”两个，其中mvin指令主要负责将DRAM中的数据按照要求传输到协处理器中的片上存储中（Databuffer或者Accumulator），mvout指令则相反，主要负责将片上存储的数据传输到DRAM中。经过保留站处理后，mvin指令将送往Load Controller模块，mvout指令将送到Store Controller模块。

为了更好的说明数据移动指令，以下图mvin指令为例进行说明，该指令从DRAM中起始地址x出发，连续读取DIM个数据，作为一行数据送往SRAM中地址为Z的那一行；之后跳过步长mm\_stride直接到地址x+mm\_stride，并再次读取DIM个数据，送往SRAM中地址Z+1的那一行，以此类推，读取完设置的行数后，则说明该指令任务完成。

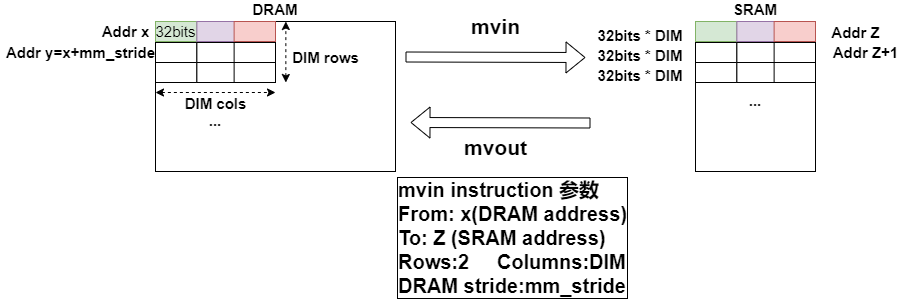


图 mvin指令的示意图

### 计算指令

计算指令，顾名思义，即完成一次卷积计算，包括对数据的移动和计算，这一过程功能较为复杂，如果使用一条指令来完成该功能，那指令长度势必非常长，且在译码过程中也会耗时很久，所以我们将该计算指令分为preload、compute.preloaded、compute.accumulated三个子指令，这三个子指令都将送往Execute Controller模块中依次执行。

以权重固定的脉动阵列为例，preload指令会提前加载特定数量的权重数据。compute.preloaded指令会将存储于SRAM中的输入激活值和偏置数据依次送往脉动阵列中PE的乘加计算输入端口，并与之前固定的权重数据完成popcount(xnor)计算，每列的计算结果经过不同的移位寄存器缓冲后进入阈值单元（也可以选择不通过阈值单元，例如BCNN算法中的倒数第二层输入是二值复数，而输出是全精度数），在完成阈值比较-拼接后，同步输出到Databuffer或者Accumulator中。computed.accumulated指令格式与compute.preloaded类似，不同的是在computed.accumulated前，不需要重新preload权重矩阵，而是与上一次矩阵运算preload的权重矩阵进行计算。

### 配置指令

配置指令是对上述两类有数据操作的指令进行参数配置，可分为config\_ex、config\_mvin、config\_mvout三种。其中config\_ex是对计算指令进行参数配置，包括是否需要激活函数、输入和权重矩阵是否转置、输入矩阵输入时步长等等，该指令将送往Execute Controller模块中。而config\_mvin和config\_mvout主要是配置数据在片外存储DRAM和片上存储SRAM传输时的步长等，将分别送往Load Controller模块和Store Controller模块进行译码与执行。

## SATU设计

当下的深度学习加速器一般都是面向DNN实现的，而DNN中最主要的且最关键的组件就是卷积层，这一点在BCNN中也不例外。下表展示了某款加速器芯片在实际应用中的计算量统计结果，由表中可见，在芯片执行算法的过程中，卷积计算占了总计算量的98.1%，所以深度学习加速器的设计步骤中，计算引擎的设计十分关键。

表 某加速器芯片的计算量实际测试结果[50]

|  |  |  |
| --- | --- | --- |
| 计算类型 | 计算量（MOPS） | 占比（%） |
| 卷积 | 34,275 | 98.1 |
| 反卷积 | 576 | 1.6 |
| ReLU | 123 | 0.2 |
| 池化 | 13 | 0.1 |

从基本需求出发，卷积层的本质就是要完成对数据的加权求和，这一过程就需要MAC（乘加器）来完成。AI芯片中常以TOPS来衡量性能，即该芯片可实现的峰值吞吐量，因为芯片电路中大多数的操作是MAC，即一次乘法伴随一次加法，所以TOPS也常用下面公式 来计算。所以为了充分利用性能，芯片设计中就需要最大限度地保证在运行时间每一个PE处于忙碌状态，这就是实现高吞吐量的关键。基于此想法，芯片研究人员开始重新使用20世纪80年代比较流行的脉动式阵列，将MAC处理单元分布到复杂且精细的脉动数据流中，例如2015年谷歌的TPU中就采用了这一方法，目前这一方法也广泛应用于深度学习加速器领域。

(3-1)

本节设计的脉动阵列阈值单元中也是采用了脉动阵列的方法，利用数据计算的并行性和连续性，提高了计算效率并降低了功耗。该单元针对BCNN算法进行设计，不同于平常的PE中采用了全精度浮点数的乘加计算，脉动阵列阈值单元中的PE采用了二值复数的popcount(xnor)计算，能效性和硬件友好性得到了大幅提升。而在脉动阵列附近增加的阈值单元，可直接对卷积结果进行BN和二值化过程，通过添加一系列简单的[comparator](https://cn.bing.com/dict/search?q=comparator&FORM=BDVSP6&cc=cn)，而避免了复杂的bn和sign计算，降低了芯片的功耗和成本。

### SATU结构设计

脉动阵列阈值单元（SATU）是为二值复数神经网络加速器芯片提供强大算力的核心单元，通过该单元可快速地将二值复数输入特征图输入到脉动阵列中并进行卷积计算得到二值复数输出特征图或者全精度输出特征图，结构框图如下图所示，主要包括专用于二值复数点积计算的脉动阵列、权重输入FIFO、输入激活值FIFO、阈值FFO、输出激活值FIFO、片上存储和Threshold模块。SATU结构设计参考自MVTU，细节可见于2.3.5节。

OCM作为SATU的数据来源，从DRAM中以矩阵向量的形式缓存输入、权重、阈值数据到片上存储空间，在译码到预取或计算指令后通过相应的FIFO将权重或待计算数据送往脉动阵列中的指定位置，在脉动阵列完成矩阵乘法（如果脉动阵列为输出固定模式，则可能还需要在脉动阵列内部的Compare Splicer模块完成阈值计算）后保存中间数据或者最终输出激活值；得到最终输出激活值后，可通过“mvout”指令将结果传输回DRAM（如果脉动阵列为权重固定模式，则可能还需要送往threshold模块完成阈值比较-拼接计算）。

通过以上对数据通路的描述中，我们不难发现本次设计的SATU中存在两套完成 阈值比较-拼接计算的通路，这是因为为了适应不同的模型和微架构需要，本次设计的协处理器中SATU的脉动阵列满足权重固定（weight stationary）和输出固定（output stationary）两种模式，这两种模式可以通过配置指令完成选择；所以为了适配两种数据流模式，我们提出了两套完成阈值比较-拼接计算的方案。阈值比较计算我们在2.3.5节作出过解释，而之所以还需要进行拼接操作，是因为我们的设计中采用了类NHWC的新型数据存储方式（详见3.5.2），即多个通道的对应像素点的数据存储在一个存储单元中，所以为了保证每层的输出特征图可以直接作为下一层的输入特征图，我们就要通过位拼接操作将阈值比较后多通道的二值复数的输出结果归约起来，恢复为类NHWC的存储方式。

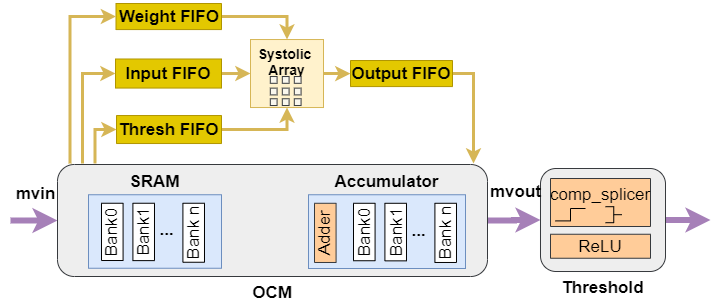


图 SATU的结构框图（OCM为片上存储，Threshold阈值单元）

以上我们介绍了SATU的主要结构和数据通路，下面我们针对其中的脉动阵列的设计细节展开讨论，其中包括此次设计的新型脉动阵列的结构描述以及两种数据流模式下的脉动阵列的运行机制。

1. 新型脉动阵列的结构

脉动阵列作为协处理器计算核心——SATU的核心计算模块，为协处理器提供了强大的并行“乘加”计算能力，主要用于处理深度神经网络中大量的矩阵乘加计算，结构图如下图 所示。该结构的设计参考了谷歌TPU的脉动阵列模块，但是与其相比又存在不同之处。

* 一方面在于TPU中采用的脉动阵列单元尺寸为256×256，这对边缘智能设备来说太庞大了。因为TPU主要是用于云端上神经网络的训练和推断任务，而本次设计的边缘智能设备主要用于完成二值复数神经网络的推理任务，所以对算力的要求远小于TPU，且出于低成本、低功耗的考虑，我们只需要将阵列的尺寸设置为16×16；
* 另一方面为了让输出固定模式下的脉动阵列完成卷积计算后，可以直接进行阈值比较计算并将得到的多个单比特二值结果进行“拼接”，再将结果输出到存储单元，我们在脉动阵列中又额外的增加了Compare Splicer模块，如下图 所示，通过config\_ex指令可配置是否进行阈值比较-拼接计算；
* 再者由于本次设计是面向BCNN算法，所以PE中的核心计算范式需要从整数或者全精度的乘加运算修改为二值复数的popcount(xnor)计算，具体见3.4.2节。

除了上述区别外，新型脉动阵列的结构与传统的结构无本质区别，每个PE通过流水线寄存器和相邻的PE连接，每个周期完成一次popcount(xnor)运算，计算结果从一个PE通过寄存器传输到另一级PE，并自动完成乘加计算，相比于CPU中的标量计算，功耗可以降低10到15倍。在脉动阵列的输入和输出端口还包含了一系列的移位寄存器，用于实现数据的同步。

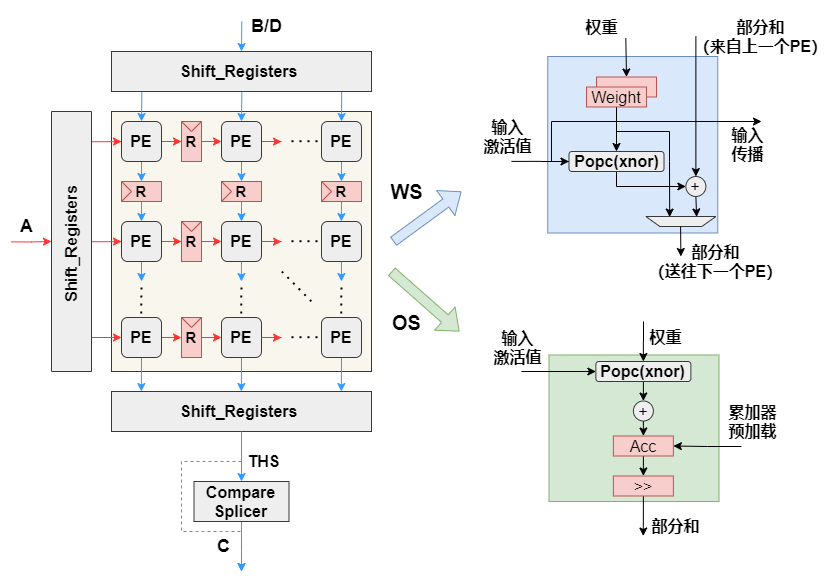


图 面向BCNN的新型脉动阵列结构示意图

1. 不同数据流模式下的运行机制

新型脉动阵列结构支持权重固定和输出固定两种数据流模式，下面以矩阵乘加计算（如下公式3-2）为例，从ISA角度简要介绍一下两种模式下新型脉动阵列的运行机制，如图 所示。公式3-2中A为输入特征图矩阵，B为权重矩阵，C为输出特征图矩阵，因为BCNN算法中没有偏置层，所以在公式3-2中没有偏置矩阵，并且我们假设在此之前我们已经将相应的输入激活值、权重矩阵数据已经通过mvin指令存储进Databuffer中。

(3-2)

首先介绍输出固定模式下的ISA运行机制：

1. 首先将矩阵A和矩阵B通过compute.preloaded指令按行依次加载进入脉动阵列，不断向前传播，在PE中完成popcount(xnor)计算并将结果与之前的部分和累加;
2. 将每个PE中计算得到的新的部分和固定保留在各自的PE中；
3. 重复前两个步骤直至完成矩阵A和矩阵B的乘加计算，之后将输出特征图C读出到Databuffer中，过程中根据功能控制信号THS（Threshold，由config\_ex指令配置）判断是否需要进行阈值比较-拼接计算。

继续介绍权重固定模式下的ISA运行机制：

1. 首先通过preload指令将矩阵B预加载进入相应PE的缓存器中并固定；
2. 紧接着通过compute.preloaded指令将矩阵A按行送入脉动阵列中并不断向前传播，此过程中每个周期每个PE完成一次popcount(xnor)计算并与上一级PE传播来的部分和进行累加，然后将得到的部分和结果向下一级传播；
3. 部分后不断传输并累加经过最后一级PE后进入Accumulator，等待后续的累加操作；
4. 重复步骤二、三直至完成矩阵乘法运算后，将Accumulator中的输出特征图C通过数据移动指令返回DRAM中，此过程中可通过地址信号的比特29（可参考3.5.1中的寻址策略）判断是否需要进行阈值比较-拼接计算。

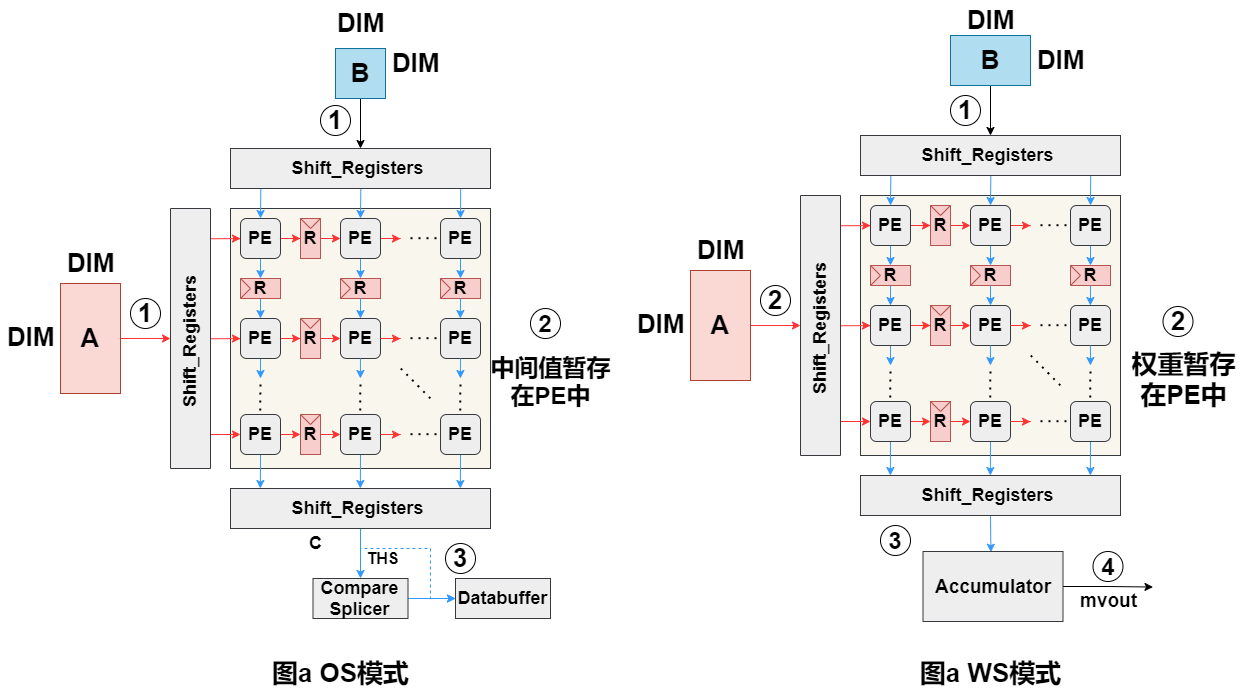


图 不同数据流模式下的运行机制

（图a为输出固定模式下，图b为权重固定模式下）

### 复数卷积运算单元设计

上一小节中我们从SATU出发，先说明了SATU的整体结构和数据通路，然后进一步阐述了针对BCNN设计的新型脉动阵列的结构以及其不同模式下的运行机制，这一小节我们将深入到脉动阵列内部的PE中，从电路层面阐述不同模式下PE的结构和运行机制，并介绍多通道并行的popcount(xnor)计算模块。

1. 输出固定模式的PE

首先介绍输出固定模式 下的PE，其电路结构如图 所示。每个PE中包含了一个32位（其中16位为实部，另外16位为虚部）的popcount(xnor)计算模块、两个32位（其中16位为实部，另外16位为虚部）的进位保留加法器（CSA）、两个寄存器、两个外部控制逻辑块以及三个二选一选择器。之所以使用32位的加法器就是为了避免计算过程的精度损失。除此之外还有一些其他的组件，例如双缓冲（double buffer）和外围逻辑电路（peripheral logic，PL)。双缓冲主要用于缓冲输入数据便于后续的计算，且通过双缓冲可以在当前计算周期未完成的情况下将先前的计算结果从脉动阵列输出，实现了non-stop计算，极大的提高了计算效率。外围逻辑电路可用于对输出结果进行移位操作，以达到缩减位宽的作用。为了减少关键路径的延迟，每个PE都拥有两个加法器、寄存器和两套独立的数据通路，每个通路负责一个寄存器。因此每个寄存器有一个独立的加法器和一个外围逻辑电路。针对两条数据通路，我们在电路的输入端和输出端各添加两个二选一mux和一个二选一mux，电路还有一个1比特控制信号PROP（Propagation）作为mux的选择信号，通过此三者可选择哪条通路用于计算，哪条通路用于传播。

根据3.4.3节中对输出固定模式下脉动阵列的ISA运行机制的描述，以计算公式3-2的矩阵运算为例，在计算开始的第一步，需先将全零的偏置矩阵D从in\_d端口预加载到PE的缓冲器中，此时PROG信号配置为低电平，矩阵D中元素通过传播通路进入寄存器C2，又从out\_c端输出进入下一个PE，经过不断传播，矩阵D按需分布在脉动阵列中。紧接着输入特征图矩阵A和权重矩阵B分别从in\_a和in\_b端口进入PE，此时PROG信号跳变为高电平，A和B的对应元素进入BC\_POPC(XNOR)进行复数popcount(xnor)计算的同时，又分别从out\_a和out\_b端口输出并通过pipeline寄存器进入下一级PE。BC\_POPC(XNOR)模块完成计算后将结果送入缓存buffer中，随后缓存buffer又将数据送往两个加法器中，分别与寄存器C1与C2暂存的累加值进行累加，但只有左侧通路新的累加值能返回寄存器C2。在完成式3-2的矩阵乘加运算得到输出矩阵C（此时矩阵C的各元素存储在各个PE的C2寄存器中）后，如果不需要在结果C上进行累加计算，则PROP信号恢复为低电平，脉动阵列即按行将每行PE中C2寄存器的累加值通过out\_c端口输出到片上存储Databuffer中，至此，一次矩阵运算才算完成，如果需要在结果C上进行新的矩阵运算并累加，则PROP信号维持高电平，并继续从in\_a和in\_b端口传入待计算数据，直至所有计算完成。

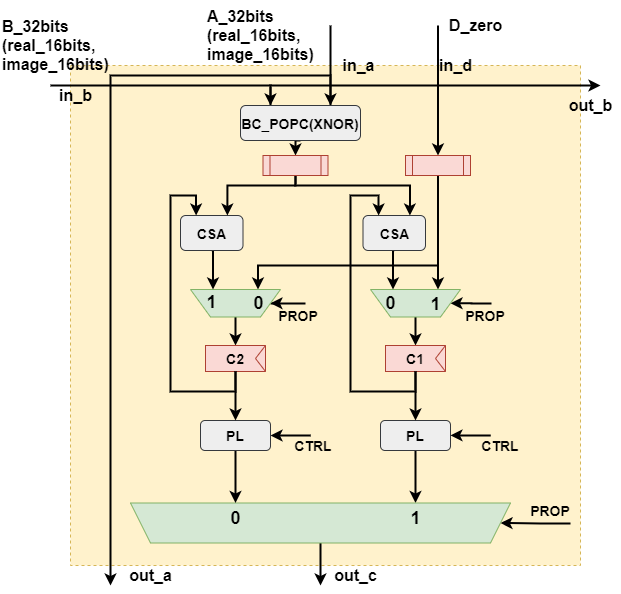


图 输出固定模式下PE的电路结构

1. 权重固定模式的PE

介绍完输出固定模式下PE的结构和运行机制后，我们展开对权重固定模式的PE的描述。与输出固定模式相同，权重固定模式也有两条数据通路，通过PROP信号控制两条通路的使用顺序。不同的是权重固定模式下的PE少了一个加法器，多了一个二选一mux，通过该mux选择寄存器C1或者C2保存的权重值与in\_a输入的激活值进入BC\_POPC(XNOR)模块进行二值popcount(xnor)计算。

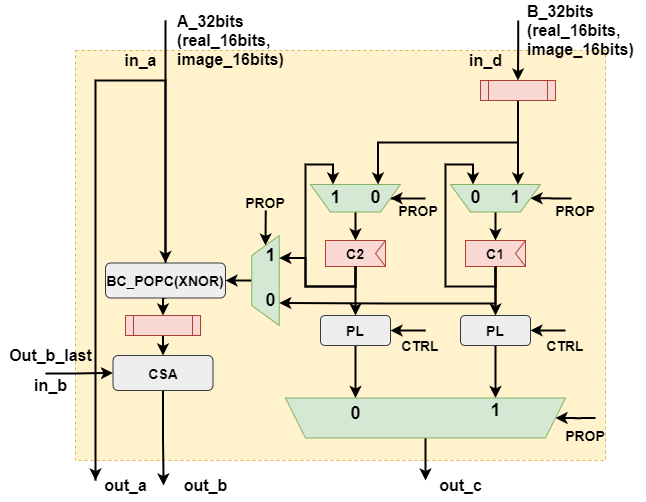


图 权重固定模式下PE的电路结构

权重固定模式下PE进行矩阵乘加运算的工作机制与输出固定模式下的存在一定的区别， 默认PROG信号为低电平，首先需将权重矩阵B（而非偏置矩阵）从in\_d端口预载入到寄存器C2中存储；完成预载任务后，PROG信号跳变为高电平，开始传输输入特征图矩阵A按列传入脉动阵列第一列的的PE中，不断向前传播，并与C2寄存器中保留的权重值进行popcount(xnor)计算，之后与in\_b端口输入的上一级的部分和进行累加，完成上述过程后，PE的计算结果会从输出端口out\_b输入给下一级PE或者Accumulator中，此时（即使上一个矩阵计算还没有完全完成）可以直接开始下一个矩阵计算的预载，PROG信号保持不变，新的权重矩阵B通过in\_b端口传输进入寄存器C1中，待预载完成（此预加载过程也可在前一个矩阵计算未完成的同时进行加载），PROP信号跳变为低，后续计算过程与前面描述的并无差异。当然如果不需要预载新的权重矩阵，也可以直接从in\_a端口传入新的输入矩阵A此时PROP信号保持不变。由上述的过程描述，我们可以发现，整个计算过程中的预载入、计算、累加计算等步骤都伴随着PROP信号的跳变。

1. BC\_POPC(XNOR)运算模块

上述中我们介绍了不同数据流模式下PE的结构和运行机制，然后不论是哪一种模式，都离不开BC\_POPC(XNOR)模块提供的二值复数的popcount(xnor)运算支持。在2.2.1节我们介绍了二值复数以及其点积计算的理论知识，而二值复数的数据特点与计算的独立性就决定了我们可以硬件设计中使用大量的并行性设计，进而提高计算效率、节约存储资源和减少访存次数。基于这两点出发，我们提出了新型的存储方式（见3.5.2节）与新型的计算电路结构。

二值数据的popcount(xnor)运算早已在BNN相关的算法和硬件设计中普及，所以我们将“复数运算”和popcount(xnor)运算的概念相结合得到了二值复数的点积运算中，但这样的并行性在多通道的卷积运算中还是不够的。所以在此基础上，我们创新性的提出多通道“同步”存储和“同时”计算的二值复数点积运算。

以两个通道的同一像素点为例，假设输入激活值为（1，-1）、（-1，1），权重值为（1，1）、（-1，1）,若采用普通设计：

1. 首先完成第一个通道的计算，即
2. 继续计算第二个通道，即
3. 求和得到结果，即

而如果采用新型计算结构，

1. 首先使用“0”替代“-1”，即像素点（1，-1）使用（1，0）表示，则我们直接得到输入激活值（10，01）与权重值（10，11），其中10、01等表示将二值复数实部和虚部分别并行存储的输入值，可看作2’b10、2’b01等。
2. 根据公式 3-3、3-4分别计算得到结果（2，-2）。

(3-3)

(3-4)

令式中popc(xnor)计算结果为*x*，之所以还需要计算，是因为我们使用“0”替代了“-1”，所以计算结果中“-1”的个数应为“通道数-*x*”（此处通道数为2），则多通道合并的popc(xnor)计算结果为（*x*-(通道数-*x*)=2\**x*-通道数）。

由上述论证，我们将通道数推广，设为*chs*，经过并行存储后的输入特征值A为，权重B为，其中、、、分别表示实部和虚部，其位宽等于通道数*chs*。则A\*B计算结果如下式3-5、3-6所示：

实部： (3-5)

虚部： (3-6)

本次设计中我们将通道数推广到16，则BC\_POPC(XNOR)模块的电路结构如图 所示，输入特征值A与权重值B进入模块后将其实部和虚部分别存储在相应的寄存器中，紧接着并行完成四个16位的popcount(xnor)计算（包括乘2与减通道数的运算，不过这两个运算很简单，只需要通过移位和一个减法器即可实现)，再根据复数运算的特性将相应的popcount(xnor)计算结果相加或者取反相加即可得到最终结果。

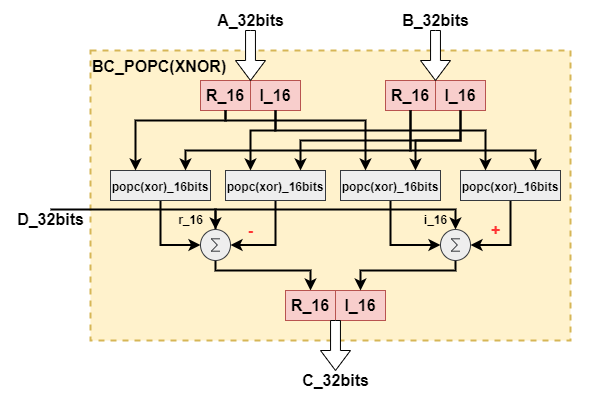


图 32bits二值复数popcount(xnor)计算单元

使用上述的结构的复数卷积运算单元，不仅极大的降低了硬件资源开销，还提高了计算的并行度，减少了访存次数，进而提升了计算效率。例如16个通道的卷积运算，原需要16次卷积运算并求和，而现在只需要一次卷积运算即可完成，不考虑访问内存的时间，计算效率就已经提升了16倍，如果考虑上访问内存的时间，计算效率会提升更多。

### Compare Splicer模块设计

在本文的2.3.5节我们解释了，BN和二值化过程是BNN以及BCNN算法中必不可少的，而这两个过程即使是在推理任务中也需要大量复杂的浮点运算，硬件实现时更是需要耗费大量的资源，所以我们参考了MVTU结构，提出了使用阈值比较（Threshold compare）计算替代BN和二值化过程，并且在训练和推理任务中都得到不错的效果。因为本次设计专注于在边缘侧设备的推理任务，所以我们在训练中得到每层的阈值，并保存在存储系统中，当需要进行阈值比较计算，则将阈值读取并送往相应的Comparator即可。

当脉动阵列所有通道的计算结果并行进入上述的Comparator后，此时我们将得到16个二值复数，如果不加处理直接送入片上存储Databuffer或Accumulator，结果将以NHWC存储方式进行存储，并不符合我们提出的类NHWC存储类型（见3.5.2节），这就不便于下一层卷积计算，所以必须要将16个通道的实部和虚部分别拼接得到一个32比特（其中16位为实部，另16位为虚部）的输出特征值。

经过上述分析，我们设计了Compare Splicer模块，其电路结构如图 所示，电路实现也十分简单，只需要一系列的比较器和位拼接即可完成，用极少的硬件开销即可实现复杂的BN和二值化过程。

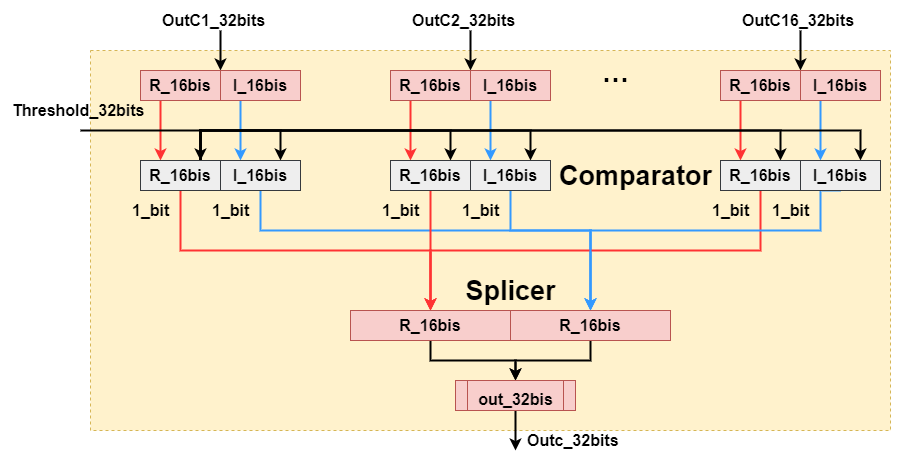


图 Compare Splicer模块结构示意图

## 内存设计及数据存储方式

除计算引擎的设计外，设计一个高效且合理的存储系统也是边缘智能设备乃至整个深度学习加速器领域的关键点。一个加速器的存储架构至少要满足以下两个条件：

1. 内存足够大，能够存储神经网络中尽可能多（乃至所有）的初始输入、权重、中间激活值等，减少对片外存储的访问；
2. 带宽足够宽，因为深度学习相关的都是数据密集型的应用，所以输出结构必须要有足够大的带宽向脉动阵列传输权重和激活值或者接收输出激活值，并跟上脉动阵列的执行速度；

显然易见，存储系统的高效性是提高加速器系统吞吐量和计算效率的关键因素，而在实际应用中我们也要考虑到设计的合理性，例如我们不能为了高效性而在存储系统中全部采用昂贵的片上SRAM存储，这样势必增大了芯片的面积和成本，这就不满足了边缘智能设备的设计中低成本的要求，例如表 中所示的以某款加速器协处理器的面积分布，仅320KB的片上存储就约占了芯片总面积的67%；同理我们也不能为降低成本而全盘使用片外DRAM，因为实验证明了神经网络计算过程中的绝大多数功耗都是来自DRAM访问，而不是乘加计算本身，且DRAM的访问延迟过高，也会限制计算引擎的执行效率，这对低时延、低功耗的边缘智能设备来说也是致命的。因此我们必须要采用分布式多级存储系统，即通过使用一块或者两块DRAM和少量的分布式本地SRAM来实现较高的MAC利用率，从而在实现高吞吐量的同时降低成本和功耗。

|  |  |  |
| --- | --- | --- |
| 组件尺寸 | 面积（um2） | 占系统百分比（%) |
| 脉动阵列（16×16） | 116K | 11.3 |
| Databuffer（256KB) | 544K | 52.9 |
| Accumulator（64KB) | 146K | 14.2 |
| CPU | 171K | 16.6 |
| 总计 | 1029K | 100 |

表 某款加速器系统的版图面积分布表[68]

本节中设计的存储系统不仅采用了分布式多级的方式，且针对二值复数神经网络的数据还提出了新型的存储方式，提高了内存空间的利用率和计算的并行性，满足了二值复数神经网络计算中需要快速提取和传输高并发数据的需求。

### 内存设计

想要协处理器中的SATU能够达到理想的算力，必要条件就是能够保证输入数据准确且及时地传输到计算单元中。本次设计的分布式多级存储系统包含了片外存储、cache以及片上存储（如图 中图a所示），本小节仅详细说明了片上存储部分。片上上存储主要包括单端口Databuffer、双端口Accumulator和逻辑控制电路三个部分。

作为片上存储系统的主要存储单元，Databuffer和Accumulator用于存储脉动阵列的输入和输出，输入数据一般存储在Databuffer中，而中间值（partial sums）和最终结果（final results）一般存储在accumulator中。二者都采用了“row-addressed”的结构，即每个地址对应的memory的一行，而每行的位宽为DIM个元素宽度之和，其中DIM为脉动阵列每行或者每列PE的个数，利用此结构每次内存访问都是一次性读取或者写入DIM个元素，减少了内存访问次数。Databuffer和Accumulator的用途不同，其每个元素的宽度也可以根据算法应用的需求而定，因为本次BCNN算法应用中，因为每个layer的通道数都是16的倍数，且针对的是二值复数，所以我们将二者的单个元素位宽都配置为32位。下面说明一下Databuffer和Accumulator的电路实现和寻址策略。

1. 电路实现

先说明电路实现，Databuffer由多个 banks组成，bank结构十分简单，仅由一块单端口SRAM和输入输出队列构成。相比之下，Accumulator的结构稍显复杂：除了两块双端口SRAM和输入输出队列，还包含了一系列的加法器去支持数据累加操作，如下图 图b所示。

而之所以Databuffer使用单端口sram，而Accumulator使用双端口sram，是因为Databuffer在算法运行过程的load、prelaod和excute阶段的每个周期只需要完成写数据或者读数据任务，而不存在同时进行读写两个任务的机会。例如load阶段我们将数据从DRAM加载进入Databuffer，此阶段的每个周期内只需要完成写数据任务；perload阶段我们将数据从Databuffer预加载进入脉动阵列，此阶段的每个周期只需要完成读数据任务；与preload阶段相同，excute阶段将矩阵乘加计算所需的输入数据依次从Databuffer按行读取进入脉动阵列，也不存在读数据任务，所以为了节约资源，Databuffer只需要使用单端口SRAM即可实现。Accumulator的工作机制与Databuffer则有些许不同，例如在权重固定模式下的脉动阵列，每完成一次矩阵乘加计算的结果将进入Accumulator并与目标地址保存的原值进行累加，这就需要在每个周期同时完成一次读数据任务、累加计算任务、写数据任务，虽然这三个任务可以通过寄存器缓存并使用状态机依次完成，但是在边缘智能设备这类实时性要求较高的应用场景中，这种方式显然是达不到算力要求的，所以我们采用了流水线形式将 三个任务并行执行，极大的缩短了算法的运行时间，因此accumulator也必须要使用双端SRAM实现。

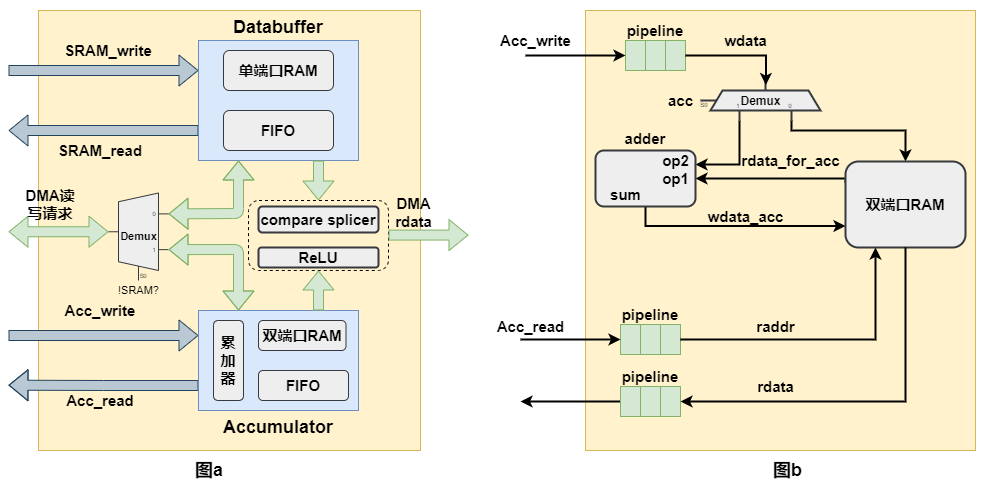


图 片上存储示意图（图a是片上存储整体结构框图，图b是Accumulator结构框图）

1. 寻址策略

在算法运行过程中，我们需要根据数据需要将数据传输到不同存储单元，例如将输入激活值送往Databuffer，中间值送往Accumulator，且对于Accumulator来说，还需要增加部分功能信号对其功能进行控制，为了数据传输不会产生紊乱和错误，就需要指定一个准确有效的寻址策略，还应该基于此要求，我们设计了以下寻址策略，如图 所示。

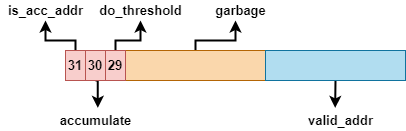


图 寻址策略示意图

内存的地址宽度为32比特，其中高三位是默认的功能位，且具有特殊意义，其中：

* 第31位（地址最高位）用于区别当前地址属于哪个存储单元，为1时表示寻址Accumulator，为0时表示寻址Databuffer；
* 第30位在寻址Databuffer或者读Accumulator时忽略。而当我们向Accumulator写数据时，第30比特位作用很关键；如果需要覆盖该地址之前的数据则可将该比特位置0，如果想要与该地址之前的原值累加，则需要置1。
* 第29位在寻址Databuffer或者写Accumulator时忽略。而当我们从Accumulator读数据时，如果要将读数据通过Compare Splicer模块进行处理时则需将该位置1，如若不处理则置0。

地址信号的低位是寻址的有效地址位，其宽度由Databuffer和Accumulator中存储行数较多的一方决定，假设Databuffer的bank数为4，每个bank的行数4096，则有效地址位的宽度。除功能位和有效地址位外，其余的地址信号都是无效位，默认为0，不需要处理。但是存在一种特殊情况，在处理执行指令时，如果指令中32位地址信号全为1时（包括无效位），则不需要去内存中读取数据，内部会自动向脉动阵列中传输全零数据。正常情况下，当地址信号进入三大控制模块（Execute Controller模块、Load Controller模块和Store Controller模块）后功能位首先被译码，并将有效地址映射到对应的Databuffer或者Accumulator的特定位置完成读写任务。

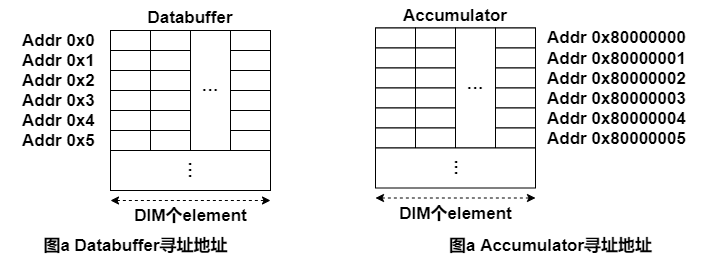


图 Databuffer与Accumulator寻址地址示意图

### 二值复数数据新型存储方式

BCNN算法中数据类型主要是二值复数，该数据类型与普通的整数和浮点数有些许区别，如果不采用合适的存储方式，必将造成存储空间利用率低，数据访问次数过多等问题，进而导致资源和能耗的浪费。基于此，我们针对二值复数提出了一种类似NHWC形式的新型的存储方式，如下图 所示。

无论逻辑表达上是几维的数据，在计算机中存储时都是按照一维形式来存储的。例如普通的NHWC格式的物理存储方式，即先取C方向数据；然后W方向；再H方向；最后N方向。此硬件平台采用的数据通道格式与存储方式，虽然类似于NHWC，但是还是有一定区别：在先取Channel方向的数据后，不是分别存储在单个的存储单元，还是进行合并后存储在同一个存储单元。之所以选择这种存储方式，原因如下：

1. 二值神经网络的每个通道中每个像素点的数据位宽仅为2比特（实部和虚部各1比特），若使用32比特的存储空间去存储2比特的数据，显然是十分浪费且不合理的；
2. 将单bit的数据合并存储后，可以减少memory的访问次数，极大的增加访问效率和计算效率，还降低了多次数访问带来的功耗浪费；
3. 另外一个优势，在于二值复数计算时不同通道计算结果的独立性，与SATU中脉动阵列配合后，一次数据读取可以完成多个通道的计算，且通过popcount计算，避免了重复累加的过程，可完美地实现多通道的并行处理。

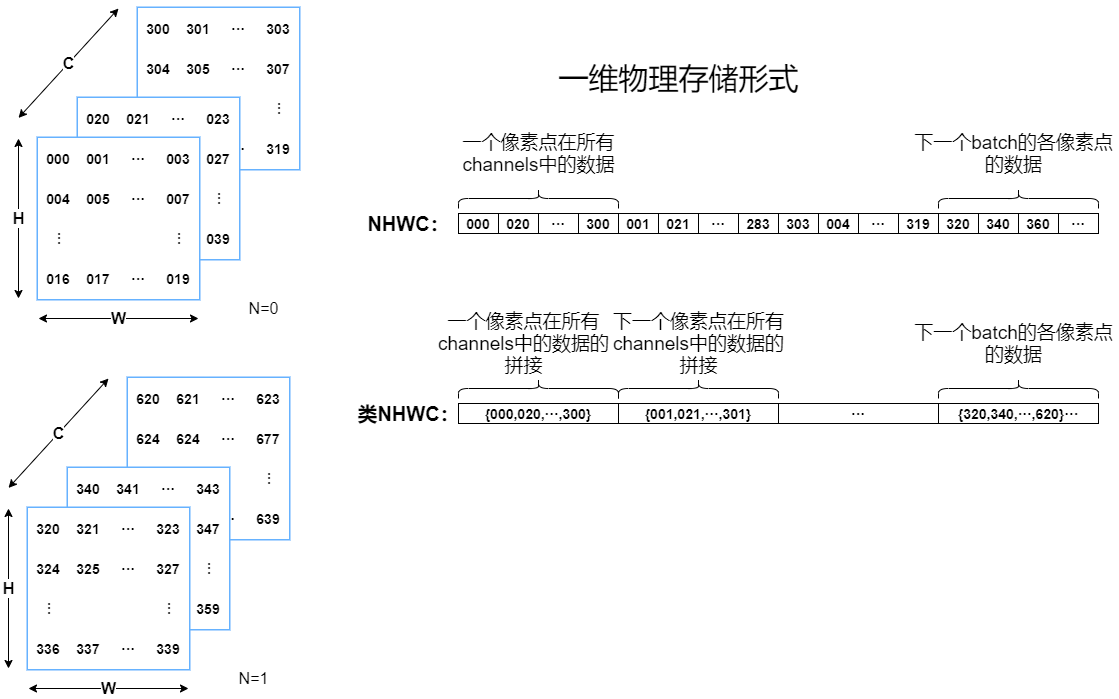


图 类NHWC与NHWC存储方式对比示意图

## 本章小结

本章中我们由浅入深、由整体到局部地介绍了面向BCNN的加速器的协处理器的设计。我们从“软硬件协同”的设计思想和“高性能、低功耗、低成本”的设计目标出发，首先介绍了协处理器的硬件整体架构和指令集架构的设计，接下来我们创新地提出了SATU（脉动阵列阈值计算单元）作为协处理器的计算核心，又分别详细地阐述了SATU中复数卷积运算单元和Compare Splicer模块的设计，最后一节我们介绍了协处理器片上存储的设计，其中包括我们提出的适用于二值复数的新型数据存储方式和“row-addressed”的内存设计。

# BCNN专用加速器的系统设计

## 引言

在近几年的自研人工智能芯片热潮中，众多研究团队、独角兽公司试图在边缘侧智能设备方向上有所突破，其中就包含了众多AI算法专用IP核的开发，但是想要这些IP核落实到实际应用，就不可避免的要嵌入完整的SoC系统中，例如苹果手机中最大的应用处理器芯片就是一块含有AI处理核的SoC。在第三章中，我们面向BCNN算法，从硬件电路角度设计了一款专用于BCNN算法加速推理任务的协处理器并提出了相应的指令集架构，与其他AI专用IP核相同，要想该协处理器能够应用在边缘设备，就必须要将其集成到SoC中。

而在AI芯片的开发过程中，特别是在边缘侧，降低功耗、减小芯片面积几乎是目前所有AI芯片都亟待解决的课题。相比手机、个人电脑等对处理器性能有极高要求的设备来说，边缘智能设备的关注点主要在处理器的控制能力而非其微弱的算力，所以在满足项目要求的前提下，大部分公司都开始尽量选择开源的处理器和工具链，而近些年兴起的RISC-V也逐渐备受青睐。

RISC-V最早起源于2010年美国加州大学伯克利分校Krste Asanovie教授主持的一个关于开源计算机系统的研究项目。RISC-V是个自由开放的指令集，其标准化工作主要由RISC-V基金会主持，该组织的会员数目前已超过100个并在不断增加。无须向基金会支付授权费用，任何想要使用RISC-V指令集设计实现处理器的公司以及个人都可以不受限制地使用其设计、制造和销售RISC-V芯片和软件。在RISC-V问世之初，移动端设备主要都是ARM处理器把持，PC的市场也主要由Intel公司的x86处理器占据，正因上述的自由开放，近些年RISC-V不断攻城略地，生态环境不断发展，给这两大巨头带了危机感，迫于形势，两大巨头也先后投入了对RISC-V的研究。



图 RISC-V基金会及其成员[71]

目前网络上利用RISC-V设计的开源项目数量众多，其中包括BOOM、Rocket、PicoRV32、RI5CY等等优秀的产品。本次面向BCNN算法的加速器的设计中，我们就使用了Rocket开源处理器，在不影响加速器性能的前提下，大大降低了加速器的设计成本。所以本章中我们首先对Rocket进行介绍，并完成Rocket与第三章中设计的协处理器的集成工作。

## 开源处理器Rocket概述

本小节我们将展开对开源处理器Rocket的介绍，因为本次加速器的设计工作重点在于面向BCNN算法的协处理器的设计，所以本节中我们只对Rocket做一个整体的概述，而不关注其设计细节。

Rocket由RISC-V的“摇篮”——美国加州伯克利大学设计，是一款32/64位（可配）、顺序执行、五级流水线的RISC-V处理器。Rocket处理器还包含了分支预测缓存（Branch Prediction Buffer，BTB）、分支历史表（Branch History Table，BTB）、返回地址栈（Return Address Stack，RAS）等模块单元，所以可以有效完成分支预测的功能。除此之外，Rocket还支持分页虚拟内存，所以可以很好地移植Linux操作系统。

不同于传统的处理器设计，Rocket处理器采用一门新的语言Chisel（Constructing Hardware in an Scala Embedded Language）编写，这也是加州伯克利基于Scala语言设计的一种开源的硬件描述语言。Chisel充分利用了Scala的优势，将面向对象（object orientation）、类型参数化（parameterized types）、函数式编程（functional programming）、类型推断（type inference）等软件语言概念引入了硬件描述语言，进而提升了硬件描述的抽象级别，并给与了硬件设计人员更强大的硬件开发能力。使用Chisel编写的硬件电路，可以通过firrtl工具编译得到对应的Verilog设计，还可以得到对应的C++模拟器。Rocket使用Chisel编写，就可以很容易得到对应的软件模拟器。同时，因为Chisel是面向对象的，所以Rocket的很多类可以被其他开源处理器、开源SoC直接使用。

Rocket处理器已流片11次之多，其准确性不言而喻，而与同为标量处理器的ARM Cortex-A5在同时采用台积电40nm工艺的情况下，其性能也毫不逊色，如下表 所示。从对比中我们不难发现，Rocket处理器面积仅为ARM Cortex-A5的一半，功耗甚至不到其一半，而其性能却更优。

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ARM Cortex-A5** | **RISC-V Rocket** | **Ratio** |
| 寄存器宽度 | 32 | 64 | 2 |
| 主频 | >1Ghz | >1GHz | 1 |
| Dhrystone | 1.57DMIPS/MHz | 1.72DMIPS/Hz | 1.1 |
| 面积（不包含Cache） | 0.27mm2 | 0.14mm2 | 0.5 |
| 面积（包含16KBCache） | 0.53mm2 | 0.39mm2 | 0.7 |
| 动态功耗 | <0.08 mW/MHz | 0.034 mW/MHz | >0.4 |

表 Rocket与 ARM Cortex-A5在台积电40nm工艺下的性能对比

## 系统硬件架构设计

在完成BCNN协处理器的基础上，我们通过与Rocket开源处理器集成，得到了面向BCNN算法的加速器，其结构如图 所示。其中s加速器协处理器通过RoCC（Rocket Custom Coprocessor Interface)接口与Rocket处理器完成指令和数据通信，当Rocke处理器取指令并经过初步译码后得出该指令属于RoCC指令，即可通过RoCC结构将指令送往协处理器并译码执行。

如果该指令为load或者store指令，需要与片外存储DRAM交互，且指令中的DRAM物理地址对应的虚拟地址在协处理器映射未命中时，协处理器会见该虚拟地址返回到Rocket并取回对应的物理地址。在取得准确的物理地址后，load controller或者store controller中的DMA将会通过TileLink总线（TileLink是由伯克利大学孕育的芯片公司SiFive提出的一套全新的芯片级总线互连标准，可以为SoC的通用处理器、协处理器、DMA等设备提供一个具有高吞吐率、高速和低延迟传输特性且可扩展的片上互联方式）向L2 Cache读取或写入数据。如果该指令为配置或者计算指令，则只需要在协处理器内部即可完成处理。

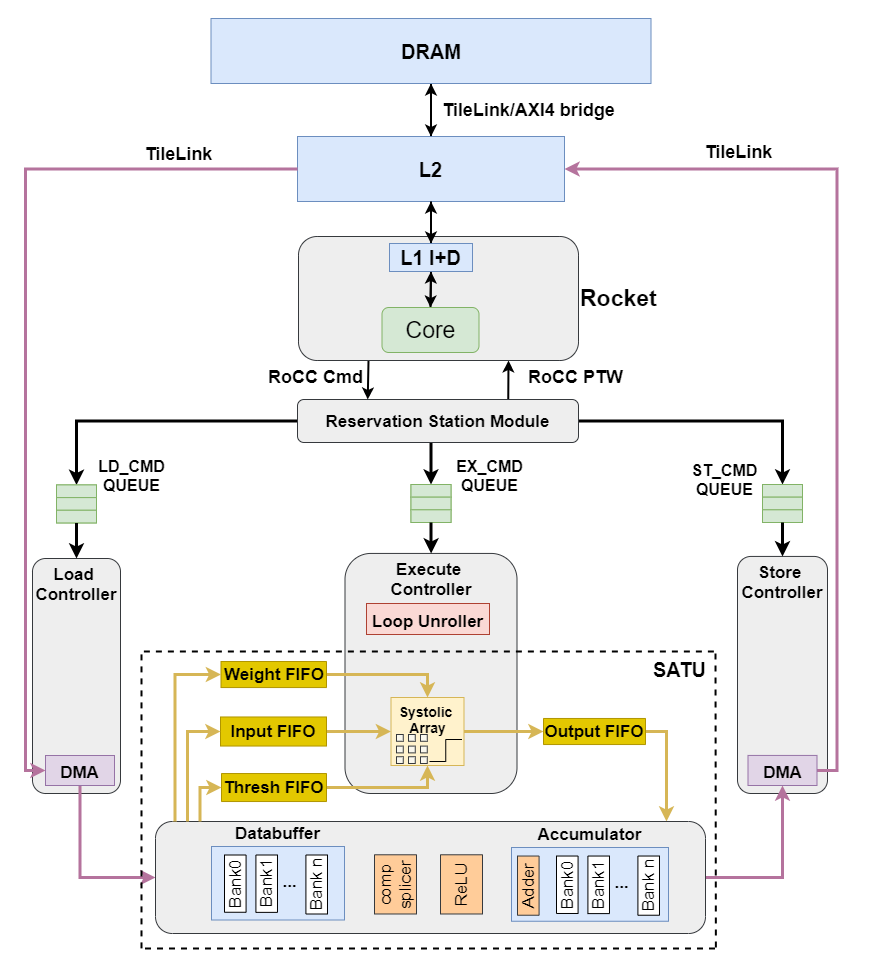


图 面向BCNN的加速器整体框图

（其中包括Rocket处理器、BCNN专用协处理器、DRAM以及Cache）

### 多级分布式存储系统

在第三章我们提到了BCNN加速器的多级分布式存储系统，当时只是说明了其分布式的特征，而本段我们将对“多级”的特性进行详细的描述。片外存储DRAM、L2 Cache、L1 Cache、协处理器中的片上存储（Databuffer和Accumulator）以及设计中的缓冲buffer、队列等共同构成了整个加速器的多级分布式存储系统。片外存储DRAM作为整个加速器的“仓库”，存储了与算法有关的所有初始数据和指令，由于其成本低廉，存储容量也最大，但是DRAM的读写速度远远小于CPU和协处理器的运行速度，这就使得系统要花费大量的时间在访问内存上，从而导致系统整体性能的降低。所以为了解决系统中数据读写速度不匹配的问题，我们在电路系统中引入了高速缓冲器，即L1 Cache与L2 Cache。

Cache一般由访问速度更快的SRAM构成，为了让论文结构完成，本段先对Cache的工作原理做一个简单介绍。Cache可分为标记存储器和数据存储器两个基本组成部分，其中标记存储器负责存储Cache的控制位的块地址标签，控制位可控制Cache的读写操作，地址标签保存Cache中各块的地址，该地址指与主内存映射的块地址，并且与Cache数据存储器中的数据一一对应。当主处理器或者协处理器读取数据时，可先将物理地址通过数据总线传输到Cache中，并于Cache中的块地址标签进行比较。如果相符合，则表示命中，只需要将Cache中对应的数据通过数据总线传输给需求方。如果不相符合，则表示未命中，此时需要将物理地址送到主存储器中，将数据读取并缓存在Cache中，再由Cache传输给需求方。可见提高Cache命中率就可以提高系统性能。而提高命中率一般可通过适当增加Cache容量、改善映射方式和块替换方式等方法来实现。由于篇幅限制，此处就不再赘述了。

L2 Cache位于与L1 Cache、协处理器与主内存之间，是整个存储系统的“中转站”和“交通枢纽”。L2 Cache通过TileLink和AXI4总线的桥接电路与DRAM控制器（图 中未标明）连接并访存DRAM中的存储数据；通过TileLink总线与协处理器中的DMA连接，完成协处理器与主存储器的信息交互。L1 Cache直接例化在Rocket处理器中，位于CPU和L2 Cache之间，分为数据缓存和指令缓存两个部分，因为内置在主处理器中，所以L1 Cache的容量和结构对CPU的性能影响最大，但是受CPU管芯面积限制，L1 Cache一般会做的稍微小一点。在此特别，DMA的数据通道宽度、TileLink总线宽度（SystemBusKey）、Cache数据通道宽度（cacheblockbytes）等这些参数都是紧耦合的，可通过在Rocket的参数中进行配置。

协处理器中的片上存储在此不做介绍，可参考3.5节。

### RoCC接口

上文中我们介绍了加速器的多级分布式存储系统，本小节我们将详细介绍Rocket处理器和BCNN协处理器之间的通信接口——RoCC接口。在第三章介绍协处理器的设计时，为了便于后期的集成工作，我们在协处理器中预留了RoCC接口。RoCC接口旨在扩展Rocket处理器并方便处理器和协处理器之间解耦通信，如图 所示，RoCC接口可以分解为多个子接口，特别是连接了处理器核和协处理器的cmd（Command）子接口，通过该接口主处理器可以向协处理器发送指令，协处理器则可以通过resp（Response）接口向主处理器进行反馈或者返回数据。为了允许协处理器访问内存，RoCC 接口还提供了 mem\_req和 mem\_resp到数据Cache的子接口。

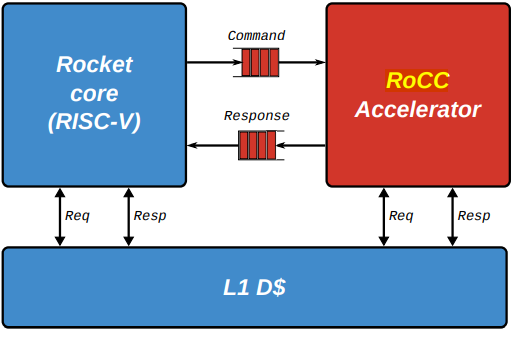


图 Rocket核与协处理器通过cmd与resp接口连接

除了这四个通道之外，RoCC 接口还提供了很多扩展子接口，可用于实现高级功能，例如可以将协处理器与 FPU 连接、与CPU共享PTW、与片外存储系统直接交互等等。 RoCC 接口还提供了一些状态和中断信号，用于主、协处理器的信号同步或标志信号错误。

## 系统软件架构设计

在本节之前，我们设计了面向BCNN算法专用加速器的协处理器并与RISC-V处理器集成得到了一个完成的加速器系统，为BCNN中的卷积神经计算性能的提升提供了硬件基础。但是对于一个神经网络的算法，从神经网络模型的实现到实际硬件电路的运行（包括计算流和数据流），这一过程需要一个多层次的软件结构进行管理。

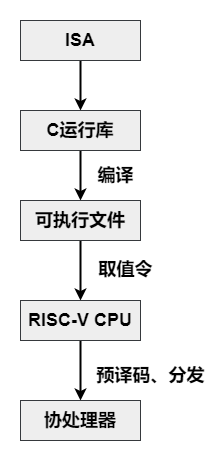


图 系统软件架构流程图

所以为了让面向BCNN的专用加速器能够启动并发挥强大的性能，我们提出了一套系统软件架构，如下图 所示。首先我们结合了3.3节的指令集架构，从底层出发，通过C APIs进行编程，得到了加速器的C运行库；在C运行库中，我们可以使用C语言调用库中的函数来完成需要的算法；在完成算法的代码后，紧接着我们通过GCC工具链将C运行库编译为机器可执行的文件；得到可执行文件并将其加载进入硬件系统的存储器后，剩下的就是专用加速器执行的过程，RISC-V处理器从存储器中取指令并译码执行，如果某条指令在简单的预译码后被判定为属于协处理器的扩展指令，则该指令会通过RoCC接口送往协处理器并继续译码执行。

上文中我们简单地阐述系统软件架构的流程，虽然每个流程我们都一笔带过，但实际上每个流程都涉及了众多的原理，在此处就不一一详解，下面对其中的GCC工具链、C运行库以及编译过程进行简单介绍。

### GCC工具链简介

工欲善其事，必先利其器。本次软件架构设计中离不开各类工具的辅助，例如编译过程中使用的RISC-V GCC（GUN Compiler Collection）工具链。GCC工具链是多个程序的集合，包括GCC、Binutils、GDB和C运行库等。

其中GCC（GNU C Compiler）是编译工具，本文中我们就是通过RISC-V专用的GCC编译器将C语言编写的程序转成加速器可执行的二进制代码，RISC-V GCC不仅支持本地编译，即在某个平台上编译该平台可运行的程序，还可以进行交叉编译，即跨平台编译其他平台运行的程序。

Binutils是一组二进制程序处理工具的集合，这些工具的功能都十分强大，在RISC-V开发和调试起到了至关重要的作用，其中包括addr2line（用于将程序地址转成其所对应的程序源文件及所对应的代码行）、objcopy（用于转换文件的格式，例如将.bin翻译为.elf，或将.elf翻译为.bin等）、ld（用于链接）、as（用于汇编）、ar（用于创建静态库）、readelf（用于显示ELF文件的信息，因为常见的编辑器无法直接查看ELF文件）、objdump（主要用于反汇编）等等工具。

GDB（GUN Project Debugger）主要用于调试C、C++、Ada等编程语言编写的程序，通过该工具我们不仅可以查看处理器和程序的运行状态（包括处理器中通用寄存器的值、存储器的值、变量值等），还可以修改这些状态。

C运行库又称为C运行时库（C Run Time Library, CRT），GCC编译工具的运行离不开C运行库的支持。为了解释C运行库的作用，我们需要回顾C语言标准，C语言标准主要包括两个部分：一部分描述C的语法，另外一部分描述C标准库。其中C标准库由多个标准头文件组成，每个头文件中都定义了相应的函数、类型声明、变量和宏定义等，例如常见的printf函数，其原型就定义在stdio头文件中。特别指出C标准库仅针对C语言本身，并没有提供实现，且与即将要运行C语言的平台无关，所以要在一个平台上运行C语言，我们就必要实现面向该平台的C运行库。

### 编译过程简介

C语言编译是一个非常复杂的过程，其中涉及但不仅限于编译器知识、硬件知识、工具链知识，此处简而言之，编译过程是将高层的C/C++语言编写的程序转为处理器能够执行的二进制代码的过程，其中包括预处理（Preprocessing）、编译（Compilation）、汇编（Assembly）、链接（Linking）四个步骤[71]。

预处理主要是对C语言程序代码做一些文本处理，例如删除所有的#define、注释，展开宏定义，处理条件预编译指令（例如#if、#ifdef、#else等）、保留#pragma编译器指令以及添加行号和文件标识等操作。

在完成预处理后就可以对文件进行编译，即进行一系列的词法分析、语义分析、语法分析以及优化处理后生成相应的汇编代码。紧接着就是汇编过程，该过程会对编译得到汇编代码进行处理并生成处理器能够识别的指令，这些指令通常保存在.o文件（目标文件）中。相对于编译过程来说，汇编过程较为简单，因为每一条汇编语句都一一对应一条指令，所以只需要通过调用Binutils中的汇编工具as，对照处理器指令和汇编指令的对照表进行逐句翻译即可。

注意到虽然汇编后的指令已经能被处理器识别，但是在链接之前还是不能直接运行的，要想成为能够被加载的可执行文件，还需要通过调用Binutils中的链接器ld来链接程序运行依赖的其他库文件、引导程序和所有需要的目标文件，最终生成一个ELF（Executable and Linkable Format）格式的可执行文件，至此一个编译过程才算完成。

### 面向BCNN协处理器的C运行库

基于上述理论，我们使用C语言编写了面向BCNN协处理器的C运行库，编译获得可执行文件并在加速器系统中顺利执行。

基于C标准库，我们首先定义了协处理器RoCC接口的扩展指令格式，如下代码4-1所示，该定义采用了内联汇编的方式，不仅提高了程序的速度，还降低了内存需求，其中asm用于内联汇编表达式的声明，volatile为可选参数，用于向GCC编译器声明该内联汇编表达式不可优化。

#define ROCC\_INSTRUCTION\_0\_R\_R（x, rs1, rs2, func7)                                   \

  {                                                                                  \

    asm volatile(                                                                     \

        ".insn r " STR(CAT(CUSTOM\_, x)) ", " STR(0x3) ", " STR(func7) ", x0, %0, %1" \

        :                                                                            \

        : "r"（rs1), "r"（rs2));                                                       \

  }

代码4-1 协处理器扩展指令底层内联汇编代码

基于上述定义，我们又展开了对协处理器指令集架构中的mvin、mvout、preload、compute.preloaded、compute.accumulated等指令的定义，其中也包括XCUSTION\_ACC、ADDR\_LEN、k\_MVIN等宏定义声明，详细代码见附录2。在前面这些工作的基础上，我们可以精准地控制协处理器的功能，并完成我们需要的数据传输和数据计算任务；另外我们还可以通过声明新的变量和参数并调用这类底层指令去实现更为高级的函数接口，方便用户直接调用。

## 本章小结

本章我们首先引入了对RISC-V生态链和RISC-V开源核Rocket的介绍；基于Rocket处理器和第三章设计的协处理器，我们通过集成得到了完整的面向BCNN的专用加速器的硬件系统；同时我们又设计了软件系统，其中包括RISC-V GCC工具链、面向协处理器ISA的C运行库以及一整套开发流程，至此我们完成了面向BCNN的专用加速器软硬件全系统的设计工作。

# 仿真与性能评估

## 引言

在第三章和第四章的工作中，我们完成了面向BCNN专用加速器的硬件系统和软件系统的设计，在本章中我们将继续基于“软硬件协同”的思想对整个加速器系统进行仿真验证[72]，并利用Vivado工具对加速器的硬件电路进行性能评估。

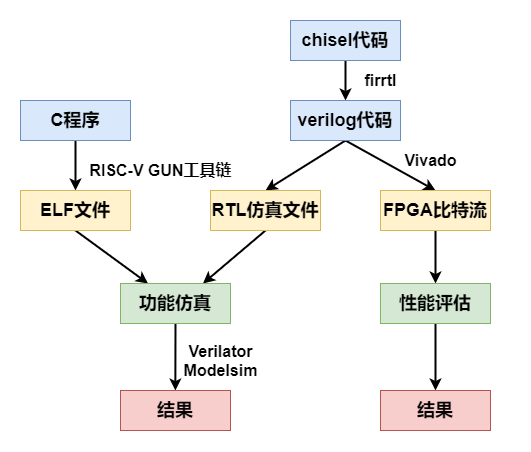


图 5-1 仿真与性能评估流程图

图5-1中展示了本次功能仿真与性能评估的流程，其中功能仿真的软件部分我们在4.4节已经做了介绍，这里不再赘述；因为我们的硬件电路设计没有使用传统的Verilog硬件描述语言而是使用了Chisel语言（该语言的优势可参考4.2节的介绍），所以在进行“软硬件协同”仿真前，我们需要通过Firrtl（Flexible Internal Representation for RTL)工具将chisel代码转译为Verilog代码，便于后续的仿真与性能评估。得到Verilog语言描述的硬件电路后，我们就可以搭建Testbench测试文件、例化待测试模块、写入ELF文件并通过仿真工具Verilator实现功能仿真；因为设计中含有一个功能准确完善的RSC-V处理器，所以我们可以通过主处理器比较仿真结果（完成数据传输或计算任务并写回DRAM的值）和预期结果是否一致来判定设计的正确性；也可以通过Modelsim查看仿真的波形文件，追踪内部信号，查看信号值与信号间的逻辑关系，进而判定设计的正确性或进行debug与设计优化。

除了设计的功能仿真，我们还可以使用Vivado套件对Verilog代码进行综合、布局布线、统计资源消耗以及评估设计性能等，并输出比特流文件，将比特流文件烧录进入FPGA中。

本小节中我们简单介绍了功能仿真与性能评估的流程，下面两节我们将详细描述本次设计的面向BCNN的专用加速器的功能仿真和性能评估结果。

## 功能仿真

本次设计的协处理器中主要用于加速BCNN算法中二值复数的卷积计算，且脉动阵列的维度为16，所以功能仿真中我们主要测试了不同尺寸矩阵的数据移动、矩阵卷积计算、卷积结果累加、阈值比较-拼接运算等功能。

1. 矩阵尺寸为16×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为16×16，即卷积核大小为4×4、卷积核个数为16；A为输入激活矩阵，尺寸为16×16；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行了阈值比较-拼接运算）；其输入A、B及输出C如下图所示：

测试步骤：

1. 随机化获取矩阵A、B数据，存储在DRAM中；
2. 将矩阵A、B数据”mvin”至Databuffer中；
3. 将权重矩阵B的数据”preload”至脉动阵列中；
4. 将输入矩阵A的数据依次送入脉动阵列并计算，计算结果会送往Accumulator中；
5. 将Accumulator中的最终计算结果”mvout”至DRAM中的二维数组C（可选择是否进行阈值比较-拼接运算）；
6. 将数组C中数据与预期数据进行比对，得出结论。

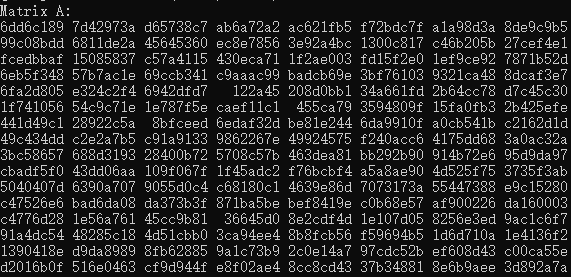


图 输入矩阵A部分数据



图 输入矩阵B部分数据

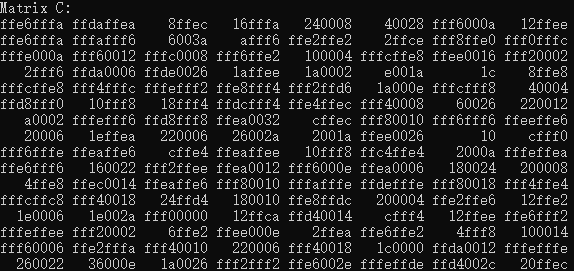


图 输出矩阵C（未进行阈值比较-拼接运算）部分数据



图 输出矩阵C（进行了阈值比较-拼接运算）

由图 、图 与图 （输出矩阵中元素高16位为实部、低16位为虚部，二者为有符号数，若最高位为1，则表示结果为负数）中数据所示，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为4×4情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。仿真波形如下图 所示，其中图 表示矩阵A、B中的数据按行进入Databuffer，并被按行读取进入脉动阵列中；图 表示权重矩阵B的数据缓存进入PE的寄存器，输入矩阵A依次通过各PE并与权重矩阵B完成popcount(xnor)和累加计算；图 表示矩阵A、B的计算结果依次进入Accumulator并被按行读取返回DRAM或进入Compare Splicer模块；图 表示上述的计算结果经过阈值比较-拼接计算后返回DRAM。

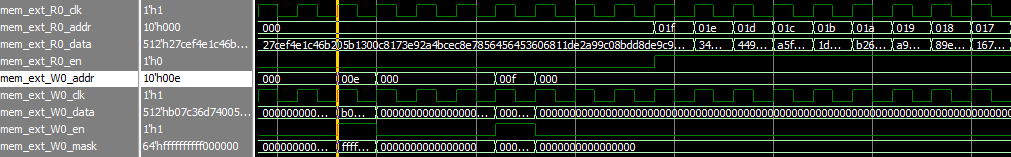


图 数据“mvin”至Databuffer并读取进入脉动阵列

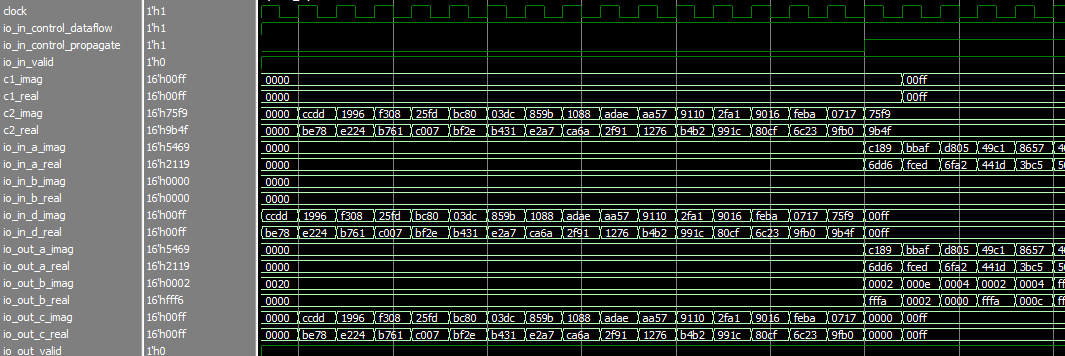


图 矩阵A、B数据进入脉动阵列并进行计算

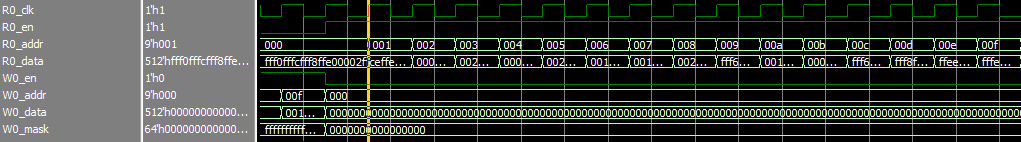


图 矩阵C数据从Accumulator中“mvout”

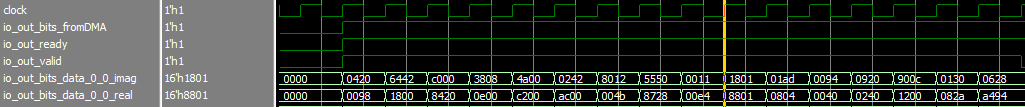


图 经过Compare Splicer模块的输出

1. 矩阵尺寸为9×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为9×16，即卷积核大小为3×3、卷积核个数为16；A为输入激活矩阵，尺寸为16×9；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行阈值比较-拼接运算）；其输入A、B及输出C如下图所示，波形文件不做展示：

测试步骤与上述测试步骤相同，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为3×3情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。

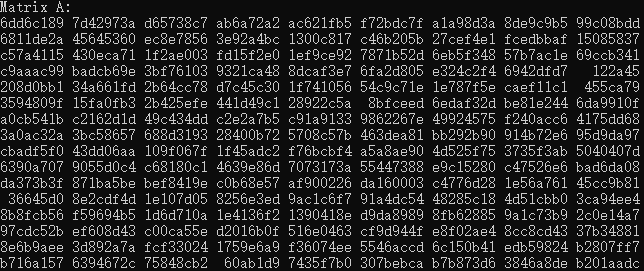


图 输入矩阵A部分数据

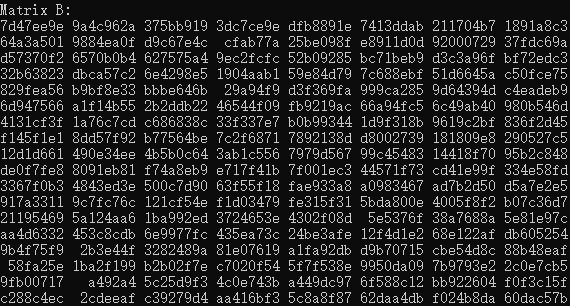


图 输入矩阵B部分数据

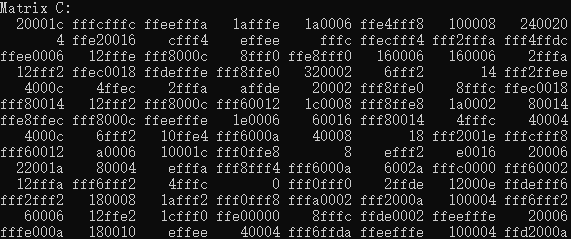


图 输出矩阵C（未进行阈值比较-拼接运算）



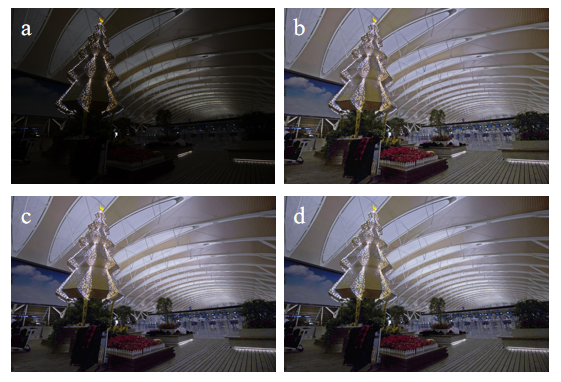
图 输出矩阵C（进行了阈值比较-拼接运算）

1. 矩阵尺寸为1×16的矩阵运算

以为例，假设B为卷积核矩阵，尺寸为1×16，即卷积核大小为1×1、卷积核个数为16；A为输入激活矩阵，尺寸为16×1；C为输出激活矩阵，尺寸为16×16（未进行阈值比较-拼接运算）或者1×16（进行阈值比较-拼接运算）；其输入A、B及输出C数据、波形文件在此不做展示，测试步骤与上述测试步骤相同，进行结果比对后可知计算结果正确，表明协处理器可以准确地完成卷积核大小为1×1情况下的二值复数的卷积计算，且可以同时完成16个卷积核的16个通道的计算。

1. LLIE-cBNN算法验证

LLIE-cBNN算法是武汉似然技术有限公司提出的一种实时、轻量型二值微光图像增强算法，可以在各类不同的灯光条件包括不均匀和弱光情况对图像进行提亮处理，我们首先将算法推广到BCNN，训练得到最优权重参数，并将参数加载至协处理器的片上存储；之后利用C运行库实现了LLIE-cBNN算法，并联合仿真得到图像增强后的结果，如下图 所示：



图LLIE-cBNN图像对比（图a为原始图片，图b为全精度DNN算法增强后结果，

图c为BNN算法增强后结果，图d为BCNN算法增强后结果）

综上所测，我们判定本次设计的面向BCNN的专用加速器可以准确有效地完成BCNN算法的加速任务。

## 性能评估

本次性能评估使用了创龙TLZ7xH-EVM开发板（如图 所示）和Vivado 设计套件。其中TLZ7xH-EVM开发板基于创龙SOM-TLZ7xH核心板（如图 所示）设计，该核心板是一款基于Xilinx Zynq-7000 系列XC7Z045/XC7Z100高性能处理器设计的异构多核 SoC，处理器集成 PS 端双核ARM Cortex-A9 + PL 端 Kintex-7架构28nm可编程逻辑资源。Vivado 设计套件由Xilinx公司推出，主要用于FPGA平台的开发和调试，其中包括RTL代码综合、资源评估、布局布线、生成比特流等功能。



图 TLZ7xH-EVM开发板



图 SOM-TLZ7xH开发板核心板

在整个加速器系统的Chisel代码转为Verilog代码后，我们利用Vivado套件对电路进行综合与实现，系统时钟设置为100Mhz，其中系统整体资源占用如表 所示，协处理器的资源占用如表 所示，脉动阵列模块的资源占用如表 所示。观察表 ，我们发现由于采用了popcount(xnor)计算，面向BCNN的脉动阵列模块只占用了少量了LUT和Register而不需要DSP等资源，具有极高的硬件友好性。

表 系统整体逻辑资源占用表

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 173079 | 218600 | 79.18 |
| Register | 118416 | 437200 | 27.09 |
| F7 Mux | 1964 | 109300 | 1.80 |
| F8 Mux | 270 | 54650 | 0.49 |
| BRAM | 369 | 545 | 67.71 |
| DSP | 164 | 900 | 18.22 |
| BUFGCTRL | 3 | 32 | 9.38 |

表 协处理器逻辑资源占用表

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 126288 | 218600 | 57.77 |
| Register | 96841 | 437200 | 22.15 |
| F7 Mux | 481 | 109300 | 0.44 |
| F8 Mux | 71 | 54650 | 0.13 |
| BRAM | 192 | 545 | 25.23 |
| DSP | 149 | 900 | 15.56 |
| BUFGCTRL | 0 | 32 | 0 |

表 脉动阵列模块逻辑资源占用表

|  |  |  |  |
| --- | --- | --- | --- |
| 资源类型 | 占用资源 | 全部资源 | 资源占用百分比（%） |
| LUT | 74885 | 218600 | 34.26 |
| Register | 42341 | 437200 | 9.68 |
| F7 Mux | 0 | 109300 | 0 |
| F8 Mux | 0 | 54650 | 0 |
| BRAM | 0 | 545 | 0 |
| DSP | 0 | 900 | 0 |
| BUFGCTRL | 0 | 32 | 0 |

除了资源占用外，我们还对整个电路系统的功耗和性能进行了评估，如下图 所示，在100Mhz频率下，电路的整体功耗为2.076W，其中动态功耗和静态功耗分别为1.903W和0.172W。由于本次设计的加速器面向的是二值复数神经网络的加速任务，采用了大量的并行结构，计算方式也与传统的整型或浮点型乘加计算方式有别，所以其性能的评估方式不同，假设电路工作在100Mhz频率，加速器中的脉动阵列含有16×16（共256）个计算单元，每个计算单元可在一个周期内完成32个通道的popcount(xnor)计算、复数运算（一次加法与一次减法计算）以及累加计算，所以本次加速器的性能为，即约3.3TOPS。

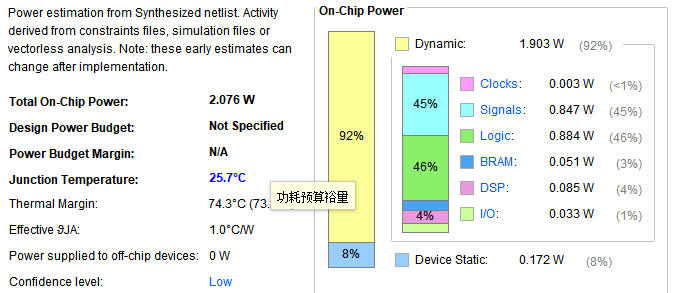


图 FPGA功耗分析报告

基于上述分析，我们针对性能、功耗以及能效三个方面，将本次设计的专用加速器与1.1.2小节介绍的几款边缘侧神经网络加速器芯片进行对比，对比结果如表 所示。由表中数据分析可得本次设计的BCNN专用加速器在能效比并不逊色于几款商用加速器，且我们还应该知道这几款商用加速器芯片都是采用ASIC制造的，我们的BCNN专用加速器仅使用FPGA实现。

表 BCNN加速器与几款边缘侧商用加速器芯片对比

|  |  |  |  |
| --- | --- | --- | --- |
| 加速器 | 性能（TOPS） | 功耗（W） | 能效比 |
| Xavier | 30 | 30 | 1 |
| EyeQ4 | 2 | 6 | 0.33 |
| EyeQ5 | 12 | 5 | 2.4 |
| Envision | 0.075 | 0.2 | 0.375 |
| BCNN 加速器 | 3.3 | 2.076 | 1.6 |

综上所述，我们评估BCNN专用加速器的资源占用、性能、功耗、能效并与其他边缘端神经网络加速芯片进行了对比，可见我们本次设计的加速器能够满足边缘智能设备低成本、低功耗、高能效的要求。

## 本章小结

本章中我们搭建了RISC-V软件编译环境和仿真平台，并在编译环境中添加了协处理器的扩展指令，进而对BCNN专用加速器系统进行了功能仿真和电路优化。完成电路优化并确定电路功能准确后，我们又使用Vivado设计套件对加速器的资源占用、性能、功耗以及能效进行评估，并在创龙TLZ7xH-EVM开发板上综合实现了加速器电路。最后我们将面向BCNN的专用加速器与其他几款商用神将网络加速器芯片进行对比，结果表明本次设计具有良好性能，且具有低成本、低功耗、高能效等优势，适用于边缘侧智能设备。

# 总结与展望

## 总结

随着神经网络不断向边缘智能设备端发展，为了解决深度卷积神经网络部署到边缘端设备时计算资源、存储资源不足，能效低等问题，基于“软硬件协同”的思想，我们在本文中设计并实现了一款面向BCNN的专用加速器，该加速器主要由RISC-V处理器Rocket核和BCNN专用加速协处理器组成，二者的所有代码均由Chisel语言实现。我们在加速器的协处理器中央内置了一块SATU，该单元采用了脉动阵列结构，且大量应用了并行化设计，不仅可以高效地完成二值复数卷积计算，且具有低功耗、高能效等优势，非常适用于边缘智能设备。本文主要工作内容如下：

1. 阐述二值神经网络和复数神经网络的研究背景和研究现状，进而引出对二值复数神经网络的研究，并结合对边缘智能设备的研究提出了本次课题的研究方向与内容。
2. 阐述了卷积神经网络的基本原理和硬件加速的实现方法，其次我们重点说明了二值复数神经网络的组成、二值化原理和二值复数计算原理，并介绍了MVTU，为后面的专用协处理器的设计铺垫。
3. 设计了面向BCNN的专用加速器的协处理器，首先我们提出了本次协处理器设计架构并设计了协处理器指令集，之后结合了脉动阵列和MVTU设计了面向二值复数神经网络卷积计算的SATU（脉动阵列阈值计算单元）；又针对二值复数的数据特点，提出并设计了“row-addressed”型片上存储系统和新型的类NHWC存储方式，不仅提高了存储资源利用率还极大地提高了计算效率。
4. 在完成协处理器设计后，我们将协处理器与开源处理器Rocket核集成，进而实现了面向BCNN的专用加速器的系统设计。除此之外，我们还介绍了RISC-V开源生态链、开源处理器Rocket、加速器系统的多级分布式存储系统和RoCC接口；在指令集架构的基础上，我们又完成了加速器软件系统的设计并通过C语言从指令集层面实现LLIE-cBNN算法的二值复数卷积运算部分。
5. 搭建了软硬件协同验证平台，并对专用加速器进行功能分析、仿真与优化；在此之后，我们又基于Vivado工具对专用加速器电路进行性能、功耗与能效评估，并利用创龙TLZ7xH-EVM开发板实现了面向BCNN的专用加速器系统电路。

## 展望

本文提出、设计并在FPGA中实现了面向BCNN的专用加速器，主要用于完成边缘侧BCNN算法的推理任务，不仅具有超高的计算性能，还同时兼备低功耗、高能效等优势，达到了边缘智能设备的设计需求，但仍然存在优化和改进的空间：

1. 在前文中，我们提出BCNN算法的第一层和最后一层需要全精度计算，但是我们专用加速器的协处理器仅支持二值复数计算，后续可以在协处理器的SATU中扩展，搭配多个全精度整数与浮点数脉动阵列计算单元，并在扩展指令集中添加数据流向选择功能，如图 所示，提高片上系统算力的完备性。

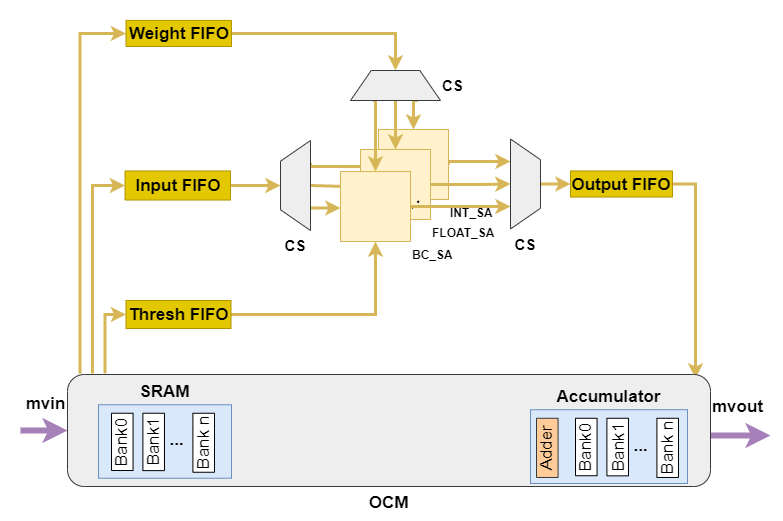


图 多核SATU单元

1. 在第五章的功能仿真验证中我们发现，当卷积计算中卷积核大小小于4×4，即矩阵运行时脉动阵列中有部分PE处于“闲置”状态，但数据信号和控制信号仍通过该PE，造成不必要的功耗与带宽浪费并降低了计算效率。为解决非标准维度的矩阵计算问题，后续添加动态配置PE状态功能，即在对“计算”指令译码过程中判断待计算矩阵的维度是否为标准维度，若不是，则根据指令中的行列信息，随时地动态地配置PE状态，关闭部分PE功能。
2. 本文设计的协处理器内部不具有im2col功能，在进行卷积计算的行滑动和列滑动操作时需要通过软件完成，后续可以添加im2col模块，搭配DMA模块，从硬件角度自动完成数据的加载任务，极大地提高计算效率。

# 致 谢

# 参考文献

[1] J. M. R. Lopez, K. Heider, J. Scheffran. Frontiers of urbanization: Identifying and explaining urbanization hot spots in the south of Mexico City using human and remote sensing %J Applied Geography, 2017, 79:

[2] 赵永强, 饶元, 董世鹏, 张君毅. 深度学习目标检测方法综述 %J 中国图象图形学报, 2020, 25(04): 629-654

[3] Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo, J. J. P. O. T. I. Zhang. Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing, 2019, 107: 1738-1762

[4] 侯宇青阳, 全吉成, 王宏伟. 深度学习发展综述 %J 舰船电子工程, 2017, 37(04): 5-9+111

[5] X. Wang, Y. Zhao, F. J. I. J. O. M. L. Pourpanah, Cybernetics. Recent advances in deep learning, 2020, 11: 747-750

[6] Y. Han, X. Wang, V. C. M. Leung, D. T. Niyato, X. Yan, X. J. I. C. S. Chen, et al. Convergence of Edge Computing and Deep Learning: A Comprehensive Survey, 2020, 22: 869-904

[7] H. Li, A. Kadav, I. Durdanovic, H. Samet, H. P. Graf. Pruning Filters for Efficient ConvNets. %J CoRR, 2016, abs/1608.08710:

[8] J.-H. Luo, J. Wu. AutoPruner: An end-to-end trainable filter pruning method for efficient deep model inference %J Pattern Recognition, 2020, 107(prepublish):

[9] Y. Yao, X. Wu, L. Zhang, S. Shan, W. Zuo. Joint Representation and Truncated Inference Learning for Correlation Filter based Tracking. 2018

[10] L. Jian-Hao, Z. Hao, Z. Hong-Yu, X. Chen-Wei, W. Jianxin, L. Weiyao. ThiNet: Pruning CNN Filters for a Thinner Net. %J IEEE transactions on pattern analysis and machine intelligence, 2019, 41(10):

[11] W. Wen, C. Wu, Y. Wang, Y. Chen, H. Li. Learning Structured Sparsity in Deep Neural Networks. %J CoRR, 2016, abs/1608.03665:

[12] L. Jiang, H. Zhang, L. J. a. S. P. Yu. Robust Time-Frequency Reconstruction by Learning Structured Sparsity, 2020:

[13] T. Geng, C. Wu, C. Tan, B. Fang, A. Li, M. Herbordt. CQNN: a CGRA-based QNN Framework. in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), 22-24 Sept. 2020: 1-7

[14] J. Choi, P. I.-J. Chuang, Z. Wang, S. Venkataramani, V. Srinivasan, K. J. A. Gopalakrishnan. Bridging the Accuracy Gap for 2-bit Quantized Neural Networks (QNN), 2018, abs/1807.06964:

[15] 冯佳玮. 基于FPGA的二值卷积神经网络加速研究与实现: [硕士学位论文]. 哈尔滨理工大学, 2022

[16] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, Y. J. a. L. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1, 2016:

[17] A. Li, T. Geng, T. Wang, M. C. Herbordt, S. Song, K. J. J. P. O. T. I. C. F. H. P. C. Barker, Networking, Storage, et al. BSTC: a novel binarized-soft-tensor-core design for accelerating bit-based approximated neural nets, 2019:

[18] A. Li, S. J. I. T. O. P. Su, D. Systems. Accelerating Binarized Neural Networks via Bit-Tensor-Cores in Turing GPUs, 2021, 32: 1878-1891

[19] T. Geng, T. Wang, C. Wu, C. Yang, W. Wu, A. Li, et al. O3BNN: an out-of-order architecture for high-performance binarized neural network inference with fine-grained pruning, 2019:

[20] T. Geng, A. Li, T. Wang, C. Wu, Y. Li, R. Shi, et al. O3BNN-R: An Out-of-Order Architecture for High-Performance and Regularized BNN Inference, 2021, 32: 199-213

[21] T. Geng, T. Wang, C. Wu, C. Yang, S. Song, A. Li, et al. LP-BNN: Ultra-low-Latency BNN Inference with Layer Parallelism, 2019, 2160-052X: 9-16

[22] N. Narodytska. Formal Analysis of Deep Binarized Neural Networks. in: IJCAI:

[23] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, S. Sagiv, T. Walsh. Verifying Properties of Binarized Deep Neural Networks. in: AAAI:

[24] G. Chen, H. Meng, Y. Liang, K. J. I. T. O. P. Huang, D. Systems. GPU-Accelerated Real-Time Stereo Estimation With Binary Neural Network, 2020, 31: 2896-2907

[25] C.-H. J. I. A. Huang. An FPGA-Based Hardware/Software Design Using Binarized Neural Networks for Agricultural Applications: A Case Study, 2021, 9: 26523-26531

[26] N. Fasfous, M. R. Vemparala, A. Frickenstein, L. Frickenstein, W. J. I. I. P. Stechele, D. P. S. Workshops. BinaryCoP: Binary Neural Network-based COVID-19 Face-Mask Wear and Positioning Predictor on Edge Devices, 2021: 108-115

[27] Y. Ma, H. Xiong, Z. Hu, L. J. I. C. C. O. C. V. Ma, P. R. Workshops. Efficient Super Resolution Using Binarized Neural Network, 2019: 694-703

[28] Y. Li, T. Geng, A. Li, H. J. A. Yu. BCNN: Binary Complex Neural Network, 2021, abs/2104.10044:

[29] M. Rastegari, V. Ordonez, J. Redmon, A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. in: ECCV:

[30] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, Y. J. A. Zou. DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients, 2016, abs/1606.06160:

[31] W. Tang, G. Hua, L. Wang. How to Train a Compact Binary Neural Network with High Accuracy? in: AAAI:

[32] X. Lin, C. Zhao, W. J. A. Pan. Towards Accurate Binary Convolutional Neural Network, 2017, abs/1711.11294:

[33] S. Darabi, M. Belbahri, M. Courbariaux, V. J. A. Nia. BNN+: Improved Binary Network Training, 2018, abs/1812.11800:

[34] M. Ghasemzadeh, M. Samragh, F. J. I. T. a. I. S. O. F.-P. C. C. M. Koushanfar. ReBNet: Residual Binarized Neural Network, 2018: 57-64

[35] B. Zhuang, C. Shen, M. Tan, L. Liu, I. D. J. I. J. C. V. Reid. Structured Binary Neural Networks for Image Recognition, 2022, 130: 2081-2102

[36] J. Bethge, C. Bartz, H. Yang, Y. Chen, C. J. I. W. C. O. a. O. C. V. Meinel. MeliusNet: An Improved Network Architecture for Binary Neural Networks, 2021: 1438-1447

[37] S. Zhu, X. Dong, H. J. I. C. C. O. C. V. Su, P. Recognition. Binary Ensemble Neural Network: More Bits per Network or More Networks per Bit?, 2019: 4918-4927

[38] J. Bethge, H. Yang, M. Bornstein, C. J. I. C. I. C. O. C. V. W. Meinel. BinaryDenseNet: Developing an Architecture for Binary Neural Networks, 2019: 1951-1960

[39] C. Trabelsi, O. Bilaniuk, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, et al. Deep Complex Networks, 2018, abs/1705.09792:

[40] S. Wisdom, T. Powers, J. R. Hershey, J. Le Roux, L. E. J. A. Atlas. Full-Capacity Unitary Recurrent Neural Networks, 2016, abs/1611.00035:

[41] H.-S. Choi, J.-H. Kim, J. Huh, A. Kim, J.-W. Ha, K. J. A. Lee. Phase-aware Speech Enhancement with Deep Complex U-Net, 2019, abs/1903.03107:

[42] Y. Cao, Y. Wu, P. Zhang, W. Liang, M. J. R. S. Li. Pixel-Wise PolSAR Image Classification via a Novel Complex-Valued Deep Fully Convolutional Network, 2019, 11: 2653

[43] N. P. Jouppi, C. Young, N. Patil, D. A. Patterson, G. Agrawal, R. S. Bajwa, et al. In-datacenter performance analysis of a tensor processing unit, 2017: 1-12

[44] M. Courbariaux, Y. Bengio, J.-P. David. BinaryConnect: Training Deep Neural Networks with binary weights during propagations. in: NIPS:

[45] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, Y. J. A. Bengio. Binarized Neural Networks, 2016, abs/1602.02505:

[46] A. G. Anderson, C. P. J. A. Berg. The High-Dimensional Geometry of Binary Neural Networks, 2017, abs/1705.07199:

[47] M. Alizadeh, J. Fernández-Marqués, N. D. Lane, Y. Gal. An Empirical study of Binary Neural Networks' Optimisation. in: ICLR:

[48] F. Lahoud, R. Achanta, P. Márquez-Neila, S. J. A. Süsstrunk. Self-Binarizing Networks, 2019, abs/1902.00730:

[49] L. Hou, Q. Yao, J. T.-Y. J. A. Kwok. Loss-aware Binarization of Deep Networks, 2017, abs/1611.01600:

[50] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, et al. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference, 2017:

[51] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, N. J. A. Sebe. Binary Neural Networks: A Survey, 2020, abs/2004.03333:

[52] J. Gao, B. Deng, Y. Qin, H. Wang, X. J. I. G. Li, R. S. Letters. Enhanced Radar Imaging Using a Complex-Valued Convolutional Neural Network, 2019, 16: 35-39

[53] D. P. Reichert, T. J. C. Serre. Neuronal Synchrony in Complex-Valued Deep Networks, 2014, abs/1312.6115:

[54] G. M. Georgiou, C. Koutsougeras. Complex domain backpropagation.

[55] T. Kim, T. J. N. C. Adalı. Approximation by Fully Complex Multilayer Perceptrons, 2003, 15: 1641-1666

[56] L. C., B. J., Z. Z. A 256Gb/s/mm-shoreline AIB-Compatible 16nm FinFET CMOS Chiplet for 2.5D Integration with Stratix 10 FPGA on EMIB and Tiling on Silicon Interposer %J Proceedings of the Custom Integrated Circuits Conference, 2021, 2021-April:

[57] Y.-H. Chen, T.-J. Yang, J. S. Emer, V. J. I. J. O. E. Sze, S. T. I. Circuits, Systems. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices, 2019, 9: 292-308

[58] B. Moons, R. Uytterhoeven, W. Dehaene, M. J. I. I. S.-S. C. C. Verhelst. 14.5 Envision: A 0.26-to-10TOPS/W subword-parallel dynamic-voltage-accuracy-frequency-scalable Convolutional Neural Network processor in 28nm FDSOI, 2017: 246-247

[59] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network, 2016: 243-254

[60] T. Moreau, T. Chen, Z. Jiang, L. Ceze, C. Guestrin, A. J. A. Krishnamurthy. VTA: An Open Hardware-Software Stack for Deep Learning, 2018, abs/1807.04188:

[61] R. Venkatesan, Y. S. Shao, M. Wang, J. Clemons, S. Dai, M. R. Fojtik, et al. MAGNet: A Modular Accelerator Generator for Neural Networks, 2019: 1-8

[62] J. Cong, J. J. I. a. I. C. O. C.-a. D. Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation, 2018: 1-8

[63] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-M. W. Hwu, et al. DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs, 2018: 1-8

[64] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs, 2017: 1-6

[65] Y. Wang, J. Xu, Y. Han, H. Li, X. J. N. a. E. I. D. a. C. Li. DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family, 2016: 1-6

[66] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, et al. A Configurable Cloud-Scale DNN Processor for Real-Time AI, 2018: 1-14

[67] G. Zhou, J. Zhou, H. J. T. I. I. C. O. a.-C. Lin, Security,, Identification. Research on NVIDIA Deep Learning Accelerator, 2018: 192-195

[68] H. Genç, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, et al. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration, 2021: 769-774

[69] A. Gonzalez, J. Zhao, B. Korpan, H. Genç, C. Schmidt, J. C. Wright, et al. A 16mm2 106.1 GOPS/W Heterogeneous RISC-V Multi-Core Multi-Accelerator SoC in Low-Power 22nm FinFET, 2021: 259-262

[70] Y. Lecun, L. Bottou, Y. Bengio, P. J. P. I. Haffner. Gradient-based learning applied to document recognition, 1998, 86: 2278-2324

[71] D. A. Patterson, J. L. Hennessy. EBOOK : Computer Organization and Design; the Hardware / Software Interface, RISC V Edition.

[72] 周围. 基于FPGA的深度学习软硬件协同设计的仿真实现: [硕士学位论文]. 电子科技大学, 2022

# 附录1 攻读硕士学位期间取得的学术成果

# 附录2 部分程序代码

**宏定义声明：**

#define k\_CONFIG 0

#define k\_MVIN 2

#define k\_MVOUT 3

#define k\_COMPUTE\_PRELOADED 4

#define k\_COMPUTE\_ACCUMULATE 5

#define k\_PRELOAD 6

#define XCUSTOM\_ACC 3

#define ADDR\_LEN 32

**mvin指令声明：**

#define mvin(dram\_addr, sram\_addr, cols, rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, dram\_addr, \

((uint64\_t)(rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(cols) \

<< ADDR\_LEN) | (sram\_addr), k\_MVIN)

**mvout指令声明：**

#define mvout(dram\_addr, sram\_addr, cols, rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, dram\_addr, \

((uint64\_t)(rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(cols) \

<< ADDR\_LEN) | (sram\_addr), k\_MVOUT)

**preload指令声明：**

#define preload(BD, C, BD\_cols, BD\_rows, C\_cols, C\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | ((uint64\_t)(BD\_cols) \

<< ADDR\_LEN) | (uint64\_t)(BD), ((uint64\_t)(C\_rows) \

<< (ADDR\_LEN + 16)) | ((uint64\_t)(C\_cols) << ADDR\_LEN) \

| (uint64\_t)(C), k\_PRELOAD)

**compute.preloaded指令声明：**

#define compute\_preloaded(A, BD, A\_cols, A\_rows, BD\_cols, BD\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(A\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(A\_cols) << ADDR\_LEN) | \

(uint64\_t)(A), ((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(BD\_cols) << ADDR\_LEN) | (uint64\_t)(BD), \

k\_COMPUTE\_PRELOADED)

**compute.accumulated指令声明：**

#define compute\_accumulated(A, BD, A\_cols, A\_rows, BD\_cols, BD\_rows) \

ROCC\_INSTRUCTION\_0\_R\_R(XCUSTOM\_ACC, \

((uint64\_t)(A\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(A\_cols) << ADDR\_LEN) | \

(uint64\_t)(A), ((uint64\_t)(BD\_rows) << (ADDR\_LEN + 16)) | \

((uint64\_t)(BD\_cols) << ADDR\_LEN) | \

(uint64\_t)(BD), k\_COMPUTE\_ACCUMULATE)

**卷积计算代码：**

conv(int batch\_size, int in\_channels, int in\_dim,

        int out\_channels, int kernel\_dim,

        int out\_dim,

        int stride, int padding,

        elem\_t input[batch\_size][in\_dim][in\_dim][in\_channels],

        elem\_t weights[out\_channels][kernel\_dim][kernel\_dim][in\_channels],

        acc\_t bias[out\_channels],

        elem\_t output[batch\_size][out\_dim][out\_dim][out\_channels]) {

    for (int b = 0; b < batch\_size; b++) {

        for (int orow = 0; orow < out\_dim; orow++) {

            for (int ocol = 0; ocol < out\_dim; ocol++) {

                for (int och = 0; och < out\_channels; och++) {

                    acc\_t result = bias[och];

                    for (int krow = 0; krow < kernel\_dim; krow++) {

                        for (int kcol = 0; kcol < kernel\_dim; kcol++) {

                            for (int kch = 0; kch < in\_channels; kch++) {

                                int irow = orow \* stride + krow - padding;

                                int icol = ocol \* stride + kcol - padding;

                                elem\_t pixel = irow < 0 || irow >= in\_dim ||

                                    icol < 0 || icol >= in\_dim ? 0 : input[b][irow][icol][kch];

                                result += weights[och][krow][kcol][kch] \* pixel;

                            }

                        }

                    }

                    output[b][orow][ocol][och] = result;

                }

            }

        }

    }

}