**目 录**

[（一）算法应用与设计需求分析 2](#_Toc100068435)

[1、蒙特卡洛算法在智能决策中的应用 2](#_Toc100068436)

[2、可重构计算研究背景 5](#_Toc100068437)

[3、可重构计算工作方式 9](#_Toc100068438)

[4、机载智能计算单元设计需求分析 17](#_Toc100068439)

[（二）可重构计算架构设计 22](#_Toc100068440)

[1、 CGRA整体架构 22](#_Toc100068441)

[2、 CGRA基本结构组成 23](#_Toc100068442)

[3、 CGRA访存通路 31](#_Toc100068443)

[4、可重构计算单元设计 36](#_Toc100068444)

[（三）110\*512\*512\*512\*10 四层网络配置与验证 48](#_Toc100068445)

[1、Layer\_0\_1\*110\*110\*512配置与验证 48](#_Toc100068446)

[2、Layer\_1\_1\*512\*512\*512配置与验证 58](#_Toc100068447)

[3、Layer\_2\_1\*512\*512\*512配置与验证 68](#_Toc100068448)

[4、Layer3\_1\*512\*512\*10配置与验证 78](#_Toc100068449)

[5、CGRA性能要求评估 90](#_Toc100068450)

[（四）未来工作展望 90](#_Toc100068451)

## （一）算法应用与设计需求分析

### 1、蒙特卡洛算法在智能决策中的应用

蒙特卡洛方法是一种以概率统计理论为指导的数值计算方法，又被称为计算机随机模拟方法。对于某个随机变量，使用抽样的结果或者大量的重复试验在进行统计时代替这个变量是其基本思想。

蒙特卡洛树搜索作为一种搜索方法，在决策空间中抽取随机样本，并根据结果构建搜索树，最后在给定域中寻找最优决策。它结合了树搜索的精度和随机抽样的一般性。蒙特卡洛树搜索对人工智能领域：游戏、顺序决策树、规划问题等方法产生了深远的影响。

蒙特卡洛树搜索的策略分为两个阶段：第一个是树内策略，为当模拟采样得到的状态存在于当前的蒙特卡洛树搜索时使用的策略。树内策略可以使用上限置信区间算法 UCT，随着模拟的进行，策略可以得到持续改善；第二个是默认策略，如果当前状态不在蒙特卡洛树搜索内，使用默认策略来完成整个状态序列的采样，并把当前状态纳入到搜索树中，默认策略可以使随机策略或基于目标价值函数的策略。

蒙特卡洛树搜索大概可以被分成四步：选择(Selection)，拓展(Expansion)，模拟(Simulation)，反向传播(Backpropagation)。

MCTS算法详细过程如下。

（1）节点挑选

对于挑选节点，选用Upper Confidence Bounds（UCB）的方法进行筛选，公式如下：

 (1)

其中为节点估计的值（比如评估的分值，或者胜率等等），*N*为该子节点的父节点被访问的次数，为该子节点被访问的次数。由公式可看出，若父节点被访问次数增多，或者该子节点被访问次数较少，或者子节点本身分值较高，则该子节点被访问的概率就会增大。*C*为可调参数*，C*越大就越偏向于广度搜索，*C*越小就越偏向于深度搜索。

从根节点开始，根据UCB算法算出每个子节点的分值，并进行排序，在我方行进阶段选择分值最大的子节点，在敌方行进阶段选择分值最小的子节点，直到底部没有子节点时停止。

如果一个节点下面还有未探索的子节点，那么先拓展下面的子节点就可以了，如果没有子节点，那么就用UCB算法来得到下一个子节点，然后便利下直到有未探索的节点可以探索。

（2）节点拓展

在节点选择阶段结束时，在选中的子节点中以均匀分布的规则创建若干子节点的过程为节点拓展

选择均匀分布的方式拓展子节点，如遇到已拓展的节点，则重新随机生成其他子节点，更新信息并返回。

（3）节点模拟

模拟是一个移动的序列，从当前节点开始，到终端节点结束。为了让子节点得到一个初始的评分。让游戏随机进行，直到得到一个游戏结局，这个结局将作为该子节点的初始评分。一般使用胜利/失败来作为评分，只有1或者0。该过程可用网络直接计算reword。

（4）反向传播

反向传播是从叶节点（模拟开始）到根节点的遍历。模拟结果被传送到根节点，并更新反向传播路径上每个节点的统计信息。反向传播保证每个节点的统计信息能够反映该节点所有后代的模拟结果。

反向传播模拟结果的动机是为了更新反向传播路径上所有节点v（包括模拟开始的节点）的总模拟收益Q(v)和总访问次数N(v)。

Q(v)：总模拟收益是节点v的一个属性，最简单形式的就是所有考虑的节点的模拟结果之和。

N(v)：总访问次数是节点v的另一个属性，代表这个节点有多少次出现在反向传播路径上。

每个访问过的节点都需要更新这两个量。如随机找一个节点，这个节点的统计信息反映了它多大可能是最佳下一步（总模拟收益），以及它被访问的频率（总访问次数）。收益高的节点是接下来探索的优秀候选节点，同时那些访问次数低的节点也是需要接下来继续探索的节点（因为它没有被探索完全）。把前面计算的这个分值反馈到“相关的”的节点上了，这个相关的意思是从根节点开始一直到这个拓展节点经过的所有节点，他们的分值和访问次数都需要更新。

（5）终止条件

为了避免程序无限的搜索下次，需定义搜索终止条件，如搜索时间、搜索深度、迭代次数等。

根据空战优势函数，空战态势与进入角、视线角、相对距离、过载、相对高度等有关。若当前时刻为*t*决策时刻，则*t*决策时刻对应的空战态势为*O*(*t*)；相应的，根据空战态势*O*(*t*)，可以得到此空战态势下的空战优势*S*(*t*)；机动动作的状态空间为*E*={*E1，E2，…，E9*}，通过智能决策算法在第*t*决策时刻选择的机动动作*A*(*t*)。

假设已知*t-*1决策时刻的空战态势*X*(*t-1*)以及机动动作*A*(*t-*1)，现在需要对*t*决策时刻的机动动作进行决策。传统方法主要以试探法为主，在*t*决策时刻分别选取不同的机动动作，估计相应的空战态势，得到空战优势*S*(*t*)，选择使得空战优势*S*(*t*)最大的机动动作*Ei*作为该时刻的机动动作。

由于求解未来*ｋ*步所有决策过程的空战优势期望，将需要罗列9k种可能动作，随着探索步数*k*的增加，使用穷举方法不现实。空战机动动作选择是通过试探未来的机动动作来确定下一步决策的空战优势概率。蒙特卡洛树搜索（Monte Carlo Tree Search, MCTS）是一种平衡历史收益和未来开发的树搜索策略。其基本原理是：首先随机选择机动动作策略，然后通过预期收益来更新原来选择策略的价值，设定随机策略选择的概率，与先前计算出的策略价值成正比，通过大量反复模拟，让最好的策略出现。因此，使用蒙特卡洛树搜索的方式来进行求解。

对于空战场景，敌我双方都试图占据优势态势，击败对方，更不存在合作的可能，因此空战场景是一场零和博弈；敌我双方的机动动作和状态信息的都可以被双方的机载探测设备获取，各机在获取自身状态的同时，执行机动动作时必然要考虑对方的机动动作和状态信息，实际上，双方在决策机动动作时都同时考虑了一个空战态势，因此双方的对战场景是信息对称的；飞机在执行机动动作和火控解算时，当飞控指令或火控指令下达时，飞机的飞控系统或火控系统会严格执行相应的指令，动作的执行是确定性的，不存在随机性；飞机的机动作是根据执行严格按指令下达的顺序执行，不会存在当前动作未执行完，而执行下一条指令；在空战智能决策中建立了机动动作库，各个动作之间彼此是相互的独立，机动动作编号转换为机动指令，因此各个机动动作是离散的。通过以上分析可知，通过对空战智能决策场景的建模，将空战的决策离散为9种基本机动动作，而空战场景是一场敌我双方信息对称的零和博弈，因此，蒙特卡洛算法完全适用于空战场景。

空战智能决策算法中的价值网络（Value network）结构是深度神经网络（Deep Neural Networks，DNN），包括4个全连接层，输入层和两个隐藏层的激活函数为ReLu激活函数，输出层的激活函数为Softmax,其网络结构如下图所示。

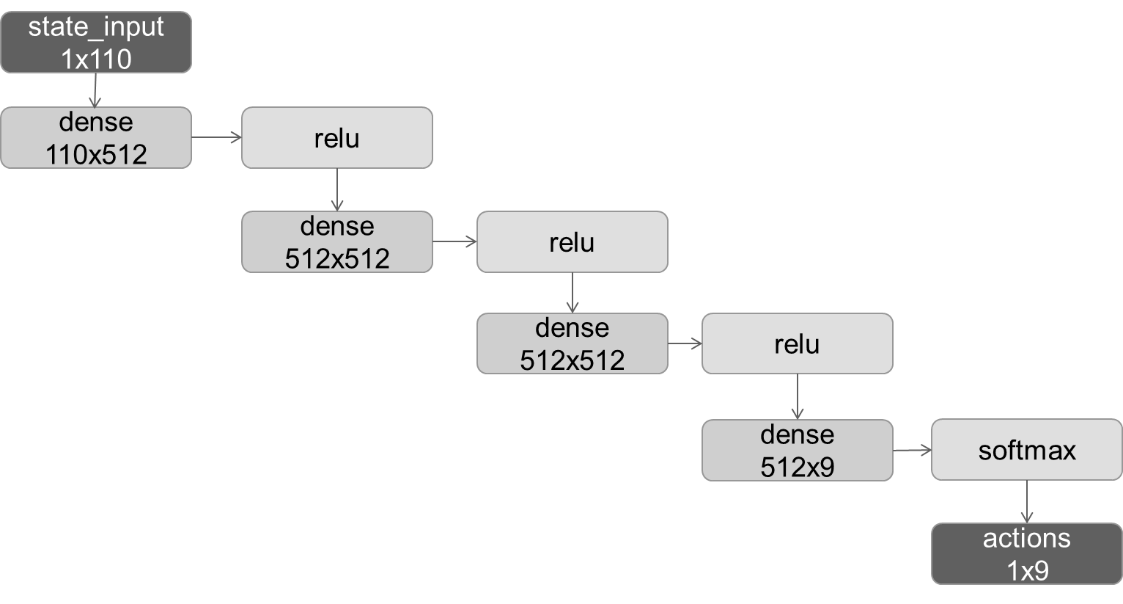


图1.1 空战智能决策算法中的网络结构

### 2、可重构计算研究背景

可重构计算的思想最早提出于 20 世纪 60 年代，在 1963 年，美国加州大学洛杉矶分校的Gerald Estrin 教授在一篇标志性的论文中首先提出了可重构计算的概念。他将计算系统定义为一个主处理器加上一组可重构的硬件构成，其中主处理器负责为可重构硬件分配计算任务，根据不同任务的计算特点，通过重构的方式对可重构硬件的行为进行配置，从而实现对某一特定任务的加速；当在可重构硬件上执行的任务完成之后，可重构硬件的行为可以被重新配置，以实现对其他计算任务的加速。但是受限于当时的集成电路发展水平，可重构计算这个先进的概念显得过于超前，远远超过了当时的电子技术水平。

进入20世纪90年代之后，随着微电子技术和计算机技术的发展，可重构计算的研究条件才逐渐成熟；另一方面，随着应用领域的不断拓展，对现代计算系统的性能、灵活性和功耗等方面同时提出了更加苛刻的要求，可重构计算这一研究领域因此获得重视，成为学术界和产业界的研究热点。在 1999 年，美国加州大学伯克利分校的可重构技术研究中心的两位研究人员 Wawrzynek 和 DeHon 为可重构计算提出了一种广义的定义，他们将可重构计算定义为具有区别于其它组织结构的以下两类突出特点的计算体系结构：

1) 具备制造后芯片的定制能力；

2) 具备很大程度的计算任务到芯片的功能映射的能力。

凡是满足以上两个特征的计算方式都可以称之为可重构计算。第一个特征使得可重构计算区别于传统的 ASIC 实现方式，即芯片在制造完成之后，其中的功能单元仍然具备重构的能力。第二个特征使得可重构计算区别于传统的 GPP 实现方式，通过集成规则的可编程阵列结构提高空间并行度，从而更高效地完成计算任务。

可重构计算系统在生产制造后是可编程的，其编程过程通过配置信息的加载和切换来完成。配置信息表达了可重构阵列内部的处理单元（PE）的运算操作，也通过定义处理单元之间的互联结构表达了内在的数据依赖关系。每套配置信息代表可重构阵列的一次运算过程，一个计算任务可以通过多套配置信息的组合来表示。相比于只支持时域映射的 GPP 结构和只支持空间映射的 ASIC 结构，可重构计算的基本特征在于同时支持时域和空域的映射：每套配置信息包含了空间拓扑关系，节省了实时分析和动态寄存数据依赖关系的硬件开销；多套配置信息的组合又可以通过时域映射表达完整的计算任务，提供了算法映射的灵活性。 正如图1.2所示，可重构计算是一种对通用处理器和专用集成电路设计的折衷，完美融合了两者优势的计算模式。

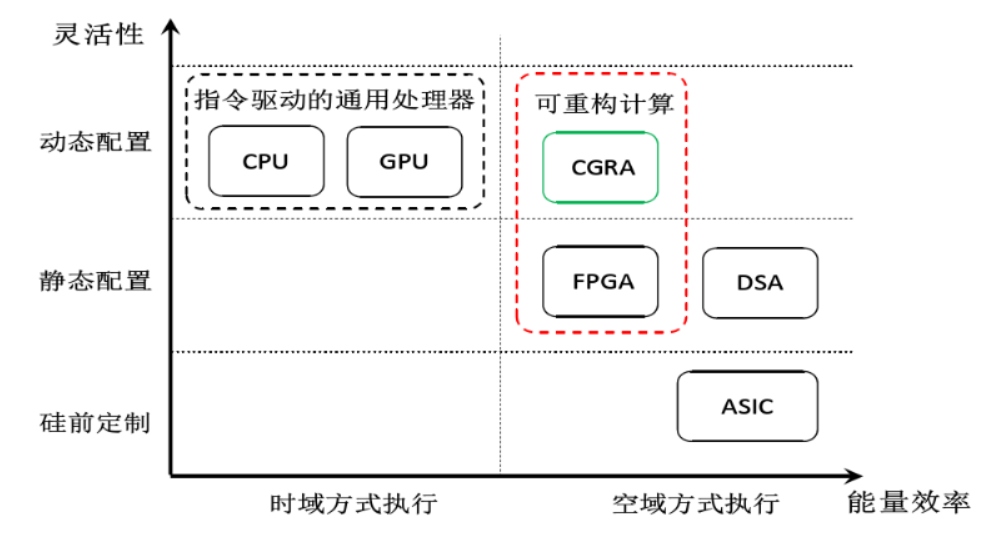


图1.2 可重构计算架构的灵活性和能效的折中与定位

在集成电路领域，可编程性是指电路在生产制造以后仍然可以通过指令或配置信息（本质上均是数据）实现不同功能的能力，因此，按照某种功能实现的 ASIC芯片通常不具有可编程能力，而通用处理器 GPP 则具有可编程能力。GPP 的硬件电路支持多种指令（通常几十条或几百条），用户可基于这些指令通过编写程序实现所需的应用功能，电路的这种性质称为软件可编程性。具有软件可编程性的电路通常是通过对硬件资源进行时分复用来实现复杂的功能，这种方式也称为时间计算（temporal computing）。与此相反，如果不对硬件资源进行时分复用，而是使用不同的硬件资源分别实现不同的功能，组合起来实现所需要的功能，则称为空间计算（spatial computing），如 ASIC 芯片、FPGA 等。与 ASIC 不同，在 FPGA芯片（如反熔丝型 FPGA、基于 SRAM 的 FPGA）中，用户可以定义或改变电路的硬件结构，以实现所需要的功能，这种性质称为硬件可编程性，具有此性质的还有可编程存储器（如 Flash）等。

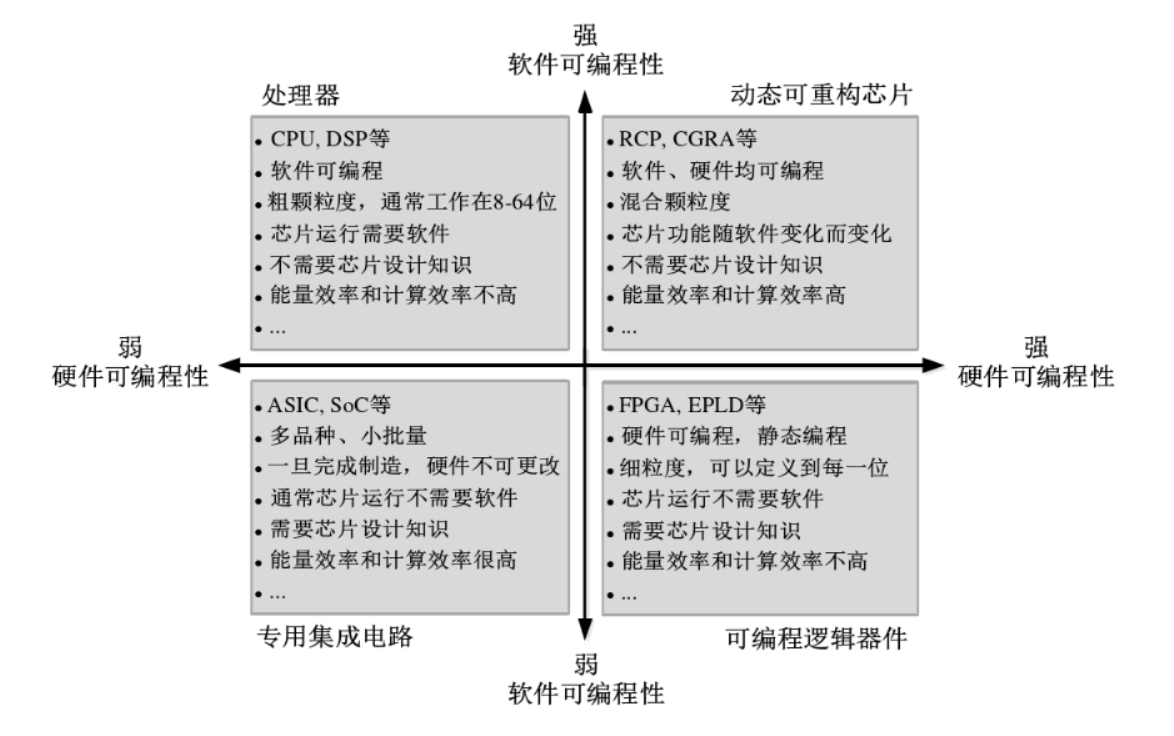
从体系结构设计空间的角度看，动态可重构计算芯片处于 GPP 与 ASIC 之间，既像 GPP 一样具有软件可编程的特性，支持时间计算，又像 ASIC 和 FPGA 一样支持空间计算。所不同的是，ASIC 并不具有硬件可编程特性，FPGA 具有硬件可编程特性，但并不支持时间计算，只有动态可重构计算芯片既具有硬件可编程特性，又具有软件可编程特性。对不同结构芯片（或称处理器）的软硬件可编程性比较如图1.3所示。

图1.3 不同架构软硬件可编程性的比较

根据配置信息对可重构硬件配置时机的不同，可重构计算一般被分为静态可重构（static reconfigurable）和动态可重构（dynamic reconfigurable）。静态可重构以 FPGA（field-programmable gate array，现场可编程门阵列）为代表，其特点是在系统初始化阶段，一次性将可重构硬件配置为系统所需的功能，在整个任务执行期间功能不再改变，直到完成该任务的全部计算之后，可重构硬件的功能才可以被再次配置。动态可重构以 CGRA（coarse-grained reconfigurable array，粗粒度可重构阵列）为代表，其特点是在任务执行的过程中可以随时重新配置可重构硬件的功能。根据处理单元的数据位宽不同，可重构计算又可以分为细粒度（fine grained）和粗粒度（coarse grained）两种，一般将数据位宽不超过 4 的称为细粒度，大于 4 的称为粗粒度。例如，FPGA 就是典型的细粒度静态可重构计算结构，Morphosys、DAPDNA-2、DREAM、XPP-III、ADRES、BilRC、DRRA、REMUS和 DySER等都是粗粒度动态可重构计算结构。

动态可重构技术是可重构计算当前的研究热点。不同于静态可重构需要停止可重构硬件的执行过程才能完成配置，动态可重构所需的配置时间要短得多，一般为几个到几十个时钟周期，并且可以在运行的状态下实时进行配置。当一个复杂的计算任务无法整个配置在有限的可重构硬件上时，可以将需要加速的任务划分为多个子任务，在执行的过程中，让不同的子任务分时使用可重构硬件，根据当前执行的子任务实时改变可重构硬件的配置信息。动态可重构技术可以根据计算任务的实际需求，分时复用可重构硬件资源，大大提升了资源的利用效率。

动态可重构计算的组织结构与通用处理器类似，包含可重构控制器（reconfigurable controller，RCC）和可重构数据通路（reconfigurable datapath，RCD）两个主要模块。其中，前者功能相当于通用处理器的控制器，后者定位相当于 ALU（arithmetic logic unit，算术逻辑单元）。一个显著的区别在于，可重构数据通路通常采用多个基本运算单元（processing element，PE）组成的阵列式结构。每个PE 的功能都类似于 ALU，而且阵列式结构具备灵活的互联网络，可以实现空间映射，因而数据通路计算能力更强。因为数据通路的功能变得复杂，传统指令集已经很难描述其功能，所以配置信息（configuration）被用来控制可重构数据通路的行为。可重构控制器的主要作用是组织配置信息，将它们以合适的顺序发给数据通路。

静态可重构计算的组织结构与 ASIC 类似，其可重构数据通路的功能在运行中不变，因而它的控制器只需要在系统初始化时加载比特流配置信息，在运行过程中不需要工作，等效为配置信息加载模块。区别于 ASIC，静态可重构计算的数据通路是灵活可编程的，因而其面积、资源和功耗都远远大于 ASIC 的数据通路。当然，静态可重构计算可以被视为动态可重构计算的特例，其相当于配置信息深度只有1的动态可重构计算，因此可重构控制器几乎是不必要的。

因为动态可重构计算结构需要在运行中不断切换配置信息，所以其配置会带来额外的功耗、降低结构的能量效率。相比之下，静态可重构计算的配置就完不影响计算的性能和功耗，具有更好的计算效率。但是，静态可重构计算有明显的局限性，其编程受限于硬件资源的数目。如果应用的规模太大，则只能通过中断运行和初始化配置实现，其代价较大，所以动态可重构计算具有更强的通用性。

### 3、可重构计算工作方式

#### 3.1 可重构计算硬件架构

可重构处理器的通用系统框架包括两个主要模块即可重构数据通路和可重构控制器，如图17所示。它们分别拥有自己的片上存储器，用于暂存数据和配置。输入和输出端口用于从外部读写数据，可重构控制器从外部读取总体的配置信息（configuration），然后可重构数据通路从可重构控制器获得自身的配置信息即配置字（context）并在控制信号的控制下完成计算过程。可重构数据通路获得配置字的顺序由可重构控制器根据数据通路的状态来确定，所以可重构数据通路主要实现任务的数据流，而可重构控制器实现任务的控制流。该框架和通用处理器的冯诺伊曼体系结构的主要区别是数据通路支持空间映射和指令由配置信息取代。

可重构处理器的主要架构由主控制器和可重构处理单元阵列（PEA）组成。主控制器控制着整个结构的执行，并通过数据存储器与 PEA 交换数据。PEA 由几个到几十个的处理单元（PE）构成，而且每个 PE 可以执行粗粒度位宽的定点数。每个 PE 里面包含一个算术逻辑单元（ALU），一些寄存器和一些多路复用选择器（MUX）。根据配置字的不同，每个 PE 里面的 ALU 的可以执行各种的常见操作（如：加法、减法、移位、与、或、非等）。PE 的互连形式也是各种各样的，如 crossbar, mesh, mesh-plus以及 morphosys等典型的互连形式。

总的来说，虽然不同的可重构处理器有自己独特的具体框架，但可重构控制器与可重构数据通路紧耦合或松耦合的组织形式具有很强的普适性，仍是可重构处理器的基本架构。

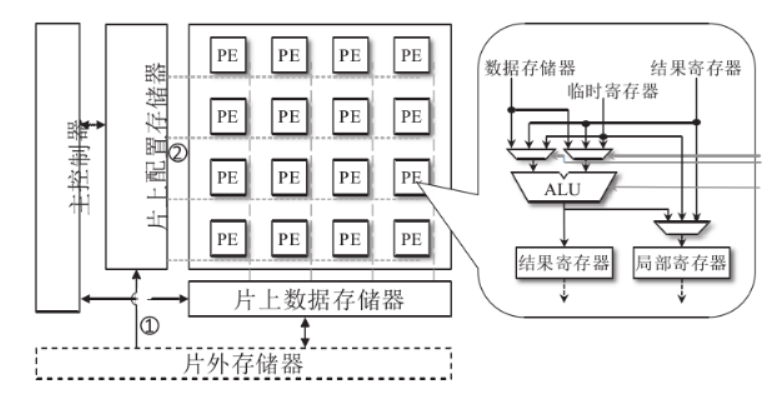


图1.4 典型的可重构处理器结构

(1)可重构数据通路

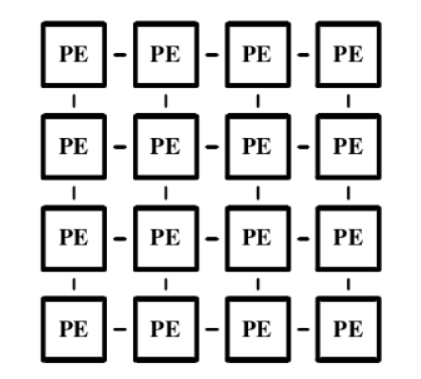
如图1.4所示，可重构数据通路一般由处理单元（Processing Unit, PE）和互联两部分组成。与超标量处理器的数据通路类似，可重构数据通路也有多个功能单元即 PE，而且数目更多。PE不是独立的处理器，而是类似于单核处理器执行单元的硬件结构，用于完成各种计算操作。空间并行的 PE 利用并行化执行方式（例如 ILP）来提升计算能力。互联（也常称为路由）的功能是在不同 PE 之间以及 PE 与外部模块之间进行快速通信，例如将一个 PE 的计算结果直接传递到另一个 PE 的输入。互联的结构通常是开关网络或多路选择器网络。

图1.5 可重构数据通路的典型结构

作为可重构处理器的主要计算核心，PE阵列是按照一定的互联方式组织在一起的多个可重构处理单元，它在配置信息的控制下完成计算过程。处理单元阵列的规模并不固定，一般根据可重构处理器面向的主要应用和完成的计算任务来决定。其中处理单元的功能在不同的可重构处理器中的实现方式可能并不相同，但是一般来说每个处理单元都可以被动态配置为执行基本的算数运算和逻辑运算。阵列内互联的结构通常是开关网络或多路选择器网络，它的功能是在不同处理单元之间以及处理单元与阵列外部模块之间进行快速通信，例如将一个处理单元的计算结果直接传递到另一个处理单元的输入。

PE 是可重构数据通路的基本计算资源，根据其结构设计分成同构和异构两种，根据计算的数据位宽又可以进一步细分为细、中、粗三种粒度，这里不再详述。PE 阵列的计算方式也分成时域计算和空间计算两种。对于前者，每次配置必须在一个工作周期完成，所有 PE 是同步执行的，互相之间不能存在依赖关系，例如ADRES；对于后者，不同 PE 的计算间可以有数据依赖关系，可以是串行执行的，所以每次配置需要多个周期完成，例如 DYSER。

互联是可重构数据通路的重要特征，使其区别于超标量、VLIW 等结构，具备了空间映射的能力。数据的流动可以用互联快速实现，所以基本的“生产者-消费者”关系的实现方式从超标量、VLIW 处理器的寄存器读写变成了硬件连线。这样，可重构数据通路可以完整实现大小合适的数据流图（data flow graph，DFG）而不依赖于寄存器堆，提高计算效率。互联的具体组织形式不是固定的，越灵活的互联需要越大的硬件代价，但是不灵活的互联会影响映射和布局布线的效果。互联的种类大致可以分为分段（segmented）式和点对点（point-to-point，p2p）式。分段式互联在大部分 FPGA 中被采用；而点对点式互联又分成一维和二维两种，前者有 Garp、RAPID和 PipeRench等结构采用，后者有 Morphosys和ADRES等结构采用。另外，可重构计算还有总线和片上网络等较松散的互联形式。图1.5中的互联是基本的二维 MESH 结构。

(2)可重构控制器

可重构控制器的功能是可编程的有限状态机（finite-state machine，FSM），即根据可重构数据通路的计算结果，选择后续的配置字。可重构控制器从数据通路的输出通道获得系统的当前状态，如寄存器的数据、PE 的计算结果等，然后进行判断产生下一个配置字的地址，相当于通用处理器的程序计数器（program counter）。如果程序不存在控制分歧，那么配置字只需要按照预定的顺序进行加载，可重构控制器也不需要获取数据通路的计算结果。对于 FPGA，可重构控制器甚至不需要动态切换配置而只需要进行一次初始化配置。

可重构控制器对可重构数据通路的控制方式一般分成两种：集中式控制和分布式控制。集中式的可重构控制器为整个可重构数据通路生成一个配置字序列和一个配置字地址。分布式的可重构控制器为每个 PE生成独立的配置字序列，每个 PE 都需要一个配置字地址。在分布式控制结构中，每个 PE 内部都有一个配置调度器（scheduler），而所有调度器的集合就是分布式控制的可重构控制器。

在集中式控制的可重构结构中，PE 单元的执行是被动的，它不能控制自己的配置流，所以这种可重构结构也被称为被动控制 PE 阵列（Non-Autonomous PEA），整个可重构数据通路对应一个控制流也就是一个线程（thread），适合使用 ILP 相关技术。典型的例子包括 MATRIX、ADRES、TRIPs、DYSER 和 REMUS 等。相反，在分布式控制的可重构结构中，PE 单元的执行是主动的，可以控制自己的配置流，所以这种可重构结构也被称为主动控制 PE 阵列（Autonomous PEA），每个 PE 对应一个控制流或线程，适合使用 TLP（thread-level parallelism，线程级并行）相关技术。典型例子包括 RAW、As AP、PicoChip、TIA等。分布式控制结构在原理上已经非常接近多核处理器，它们的主要区别在于 PE 的功能和互联的方式。多核结构的每个核心单元都是独立的处理器，灵活性更强，但是它们之间的互联一般通过较慢的总线或是共享存储器实现，性能较低。

可重构控制器常常由一个通用处理器（松耦合）或是处理器的控制单元（紧耦合）来实现。为了提高速度，也有结构设计了专用的控制器模块如地址发生器和可编程查找表。

#### 3.2 可重构计算执行模式

可重构计算作为一种兼具高能效和高灵活性的动态可编程的计算范式，其主要的计算思想就是：（1）配置信息加载配置一次执行多次，尽量长时间不切换更改，保证类似 ASIC 方式的数据流驱动，提高能效；（2）允许配置信息本地存储，支持按需动态加载和切换，实现灵活的软硬件可编程性；（3）计算任务尽量通过空间并行的执行方式实现，提高计算速度。也就是说，可重构计算范式是一种同时具备时“时域自由度和空域自由度”的实现方式，这也是其具有高能效、高灵活和高适用性的根本原因所在。

空域自由度：任务运算的数据依赖关系体现在可重构计算处理中处理单元 PE的互连关系上。每个具体的操作被空间映射到某个具体的 PE 单元上，而运算操作间的数据依赖关系可以显示的明确不同 PE 之间的“生产者-消费者”的关系，数据通过 PE 间互连网络进行空间维度的传递，实现较高的计算并行度，同时有效降低数据通信的功耗和时间开销。

时域自由度：可重构处理器可以实现配置信息的快速动态切换，根据任务需要，灵活重构。同时在时域上，也可以通过“配置一次、执行多次”的机制，实现配置信的高效利用，降低冗余的配置读取和译码等的开销。这也对可重构计算空域自由度的一种正向反馈机制。

(1) 计算范式

可重构计算范式同时在空域和时域上具有较高的自由度，而经典冯诺依曼架构仅有时域的高自由度。这是可重构计算区别与其他计算形式的根本原因。

可重构计算范式的高能效性主要表现在两点：第一，空域上可重构计算依靠处理单元 (PE) 间的互联显式实现依赖关系。应用中的基本操作及它们之间的依赖关系可以用控制数据流图（CDFG, Control-DataFlow Graph）来表示。每个应用中都会存在多种数据依赖关系和控制依赖关系，对应的应用程序中需要确保依赖关系正确执行。目前主流编译器都会提取应用的 CDFG 用于优化。然而由于冯诺依曼计算范式只有时域一个维度，编译器在生成其程序时需要将二维的 CDFG 改写成一维的指令流，利用寄存器名称隐式表达指令间的依赖关系。之后通用处理器在执行程序时要依赖多端口寄存器堆或流水线旁路逻辑重发现这些依赖关系。相比而言，可重构计算范式加入了空间维度，这使得 CDFG 中的依赖关系可以通过配置数据通路上处理单元的互联网络显式表达，避免了重发现依赖关系的能量和性能额外开销。第二，时域上可重构计算功能单次配置，多次执行。冯诺依曼计算范式下指令的加载和执行是耦合的：每条指令执行前都需要经过取指、译码等步骤。即便编译器发现一段指令需要重复执行，它也无法在程序中省掉这段指令在第一次执行后的取指和译码等步骤，只能生成带有控制的指令流，期望通用处理器架构上的分支预测器和指令高速缓存可以减小重复加载指令的开销。相比而言，可重构计算范式的配置信息功能更为自由。对于需要重复执行的功能，配置信息中可以指定数据通路多次执行该功能；对于连续变化的功能，配置信息可以利用配置管理器使数据通路快速连续重构以满足功能要求。

（2）编程方法

可重构计算范式在时域和空域的自由度需要以编程方法的形式向程序员提供。可重构计算范式下编程方法需要开发应用中的并行性以充分利用时域和空域的计算资源。本小节根据并行性的开发方式将编程方法做如下分类：

硬件描述语言：HDL 让程序员自由表达时域和空域上的功能，使得程序员可以充分利用可重构计算范式。但 HDL 抽象层次过低，降低了程序员的开发效率。

命令式编程：为了提升抽象层次，可重构计算范式借助通用处理器的命令式编程语言提高了开发效率。由于原始的命令式编程语言（如 C 语言）难以表达并行性，可重构计算范式需要借助基于 HLS 的编译器自动开发并行性以充分利用计算资源。但自动并行化一直是编译技术的核心难题，尤其在非规则应用领域还没有系统的自动并行编译流程。

并行编程方法：近年来随着可重构计算研究对编程模型问题越来越重视，多种起源于通用处理器的并行编程方法被移植到可重构计算范式下。之前列举的基于 SIMT 的编程方法和基于流计算特征的编程方法都属于这一类。不同的编程方法下同一个应用在可重构处理器上的性能会截然不同。目前学术界主流观点认为在编程方法层面提升性能的最有效途径是根据应用领域定制编程方法。

（3）数据访存机制

可重构处理器是一个典型的空间并行的计算架构，其计算效率能不能充分发挥，很大程度上取决于其计算数据能不能及时从片外搬运到片上计算阵列的 PE 单元来。从硬件的空间组织形式来看，可重构处理器和 GPU 有很大的相似性，大量并行的计算阵列块，往往越到片上阵列的计算单元，其需求访存带宽越大。可重构处理器一般需要采用多级分层的片外和片上存储系统。片上主存储器一般采用大容量的动态随机存储器 DDR 或者一些高带宽存储器（HBM，HighBandwidth Memory）。片上则采用 Cache、片上共享存储 SM 和分散于阵列的寄存器堆文件组成三级存储体系结构，来满足片上对数据的访问需求。缓存 Cache 是一种硬件控制且对程序员透明的存储结构，它能提高片上数据局部性访问，提高数据片上的利用率，降低片外访问频率。片上共享存储 SM，是一种类似便签存储器的快速访存结构，可由软件控制和能够支持用户可编程，且访存延时确定，能提高片上可重构阵列的访问效率和计算性能。通常片上共享存储 SM 采用 Multi-Bank的结构实现数据的并行访问，以及乒乓结构实现数据预取等，进一步提高片上数据的带宽。另外，通过分散的 PE 内部的局部寄存器文件（LR，Local Registers）甚至阵列共享的全局寄存器文件（GR，Global Registers），进一步利用片上数据产生和消耗的局部性，来提高中间计算结果在片上 PE 单元间的传递和读取速度，降低对存储器 Memory 的访问，来加速计算和降低访存开销。

（4）互连机制

PE 的互连形式是可重构处理器的重要参数，它对 DFG 图调度的难易程度、应用的执行性能等都有很大的影响。一般来讲，互连形式越丰富，调度工作就越容易，同时也需要更多的互连硬件资源。因此，PE 互连形式对于循环映射的方法设计有很重要的影响。可重构处理器阵列通常采用点对点的方式，如图1.6所示，其中典型的互连（Router）结构包括：一维交叉（Crossbar）、二维 Mesh、二维 Mesh-plus/二维 Mesh-Torus、二维 Mesh-Morphosys 等。所以，可重构计算通过 PE 之间的互连，使得 PE 的输出可以直接的传输到相连的其它 PE 的输入，而不需要经过寄存器堆或者通过片上共享存储器来中转这种直连的“生产-消费”的ASIC 互连方式，不仅可以让产生的中间结果可以直接被处理，速度快且功耗低，也可以让应用在空间上进行映射然后并行执行，取得类似 AISC 执行的性能和功耗。受芯片面积和功耗的限制，mesh 结构是最为常用的一种二维互连结构。其中，每一个 PE 只能与周围最为邻近的4个 PE 相互连通传递数据。二维互连形式的 PE 之间的连通的灵活性会比一维互连形式低一点。它的执行方式一般是在时间上对整个计算阵列往下扩展的。因此，二维互连形式的可重构计算阵列上的循环 DFG映射一般复杂度较高，是一个 N-P 完全问题。

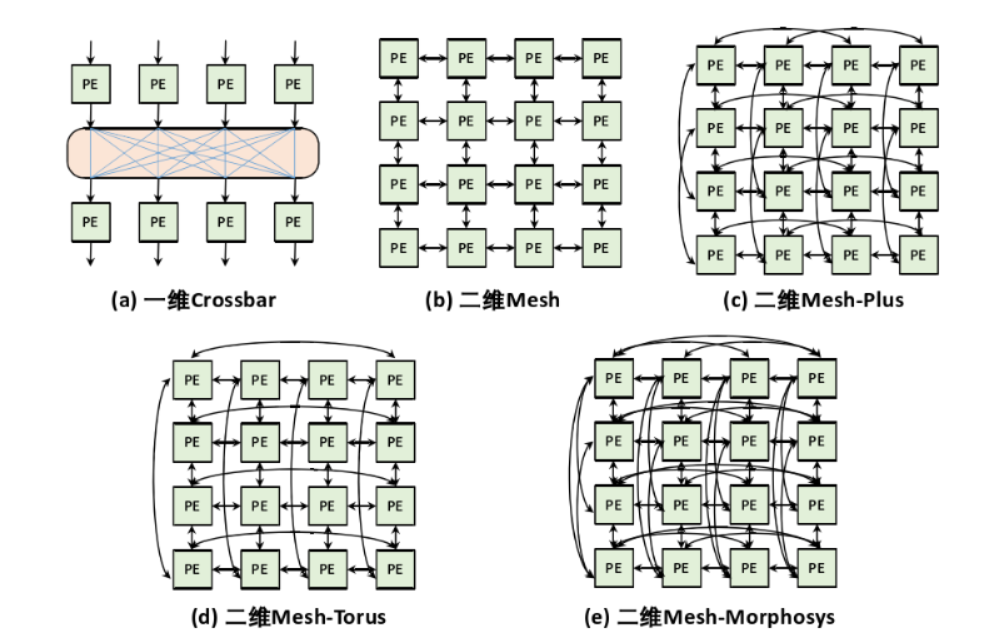


图1.6 可重构处理器计算阵列常用的片上互连方式

显然，PE 的互连资源直接影响了应用任务进行空间映射的难度和最终的执行性能。一般来说，互连资源越丰富，应用在可重构处理器阵列上的映射难度越低，执行性能往往也会越好，而相应硬件实现资源的代价也越大。比如功耗和面积都会急剧增大，以及芯片最后物理布局布线和生产制造都会变得更困难。所以，可重构计算的硬件设计者往往需要根据实际的应用需求和项目需求，尤其是当可重构阵列规模变大后，进行合理的评估后，对可重构互连方式进行适当折中和优化。

（5）配置机制

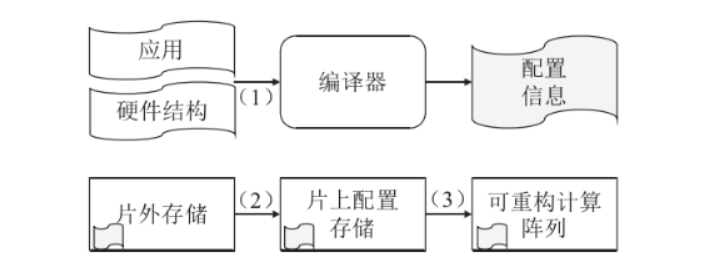
配置机制是可重构处理器最为特别的地方，合理的配置方式能够让可重构处理器快速的改变阵列的功能。因此，分析清楚与配置代价相关的硬件结构，对映射工作有重要的意义。如图1.7所示，（1）编译器首先根据应用和硬件结构描述生成配置信息并事先存储在片外存储器中，（2）然后，配置信息被从片外存储器中取出来存储在片上配置存储器中，（3）配置存储器里面的配置信息被分发到可重构计算阵列中的每一个 PE 及互连控制上来完成阵列功能及互连形式的配置。步骤（1）（2）（3）都跟配置代价息息相关，而且他们都跟硬件的实现形式有很大关系。

图1.7 可重构处理器配置过程

在步骤（1）中，配置信息的大小除了跟应用程序本身和编译技术相关，还跟硬件结构的具体实现形式有很大关系。丰富的阵列连通资源可以使调度更加容易。但是，更加丰富的互连通道也需要更多的连线、多路复用选择器等互连硬件资源。这些互连硬件资源就需要更多的配置信息来控制因此，更为灵活的和规模更大的可重构计算阵列会引入更大的配置信息。配置信息的大小直接影响着配置信息访存和分发的延时。

在步骤（2）中，片外存储器一般容量比较大，而访问延迟也比较大。配置信息一般是提前预取到片上存储器中的。因此，片上存储器的大小就对（2）部分的重构代价具有很重要的影响。如果片上配置存储足够大，可以把一个应用的配置信息完全预取进来，那么运行时这部分就只有很少的重构代价。如果片上配置存储器不是足够大，运行时还有可能需要从片外取配置信息，这会产生相当大的重构代价。在实际的实现中，片上配置存储器的大小总是有限的。配置信息被取到片上配置存储器之后，怎么样把这些配置信息分发到可重构计算阵列上面去对配置代价也有重要的影响。

总的来说，更多的配置信息有可能会带来映射的灵活性，但是也会增加配置信息的传输代价。大容量的片上配置存储可以预取更多的配置信息，减少运行时的配置延时，但同时也会增加芯片面积功耗。更大的配置带宽可以减少配置延时代价，但同时也会带来更多功耗开销。因此，在可重构阵列的实现的时候，这些因素是相互折中的。

### 4、机载智能计算单元设计需求分析

设计需求首先分析蒙特卡洛树搜索算法的具体运算需求。以AlphaGo算法为例，AlphaGo算法的计算主要包括蒙特卡洛树搜索和神经网络模型计算，而DeepMind团队在用AlphaGo算法和人类旗手以及其他棋类算法比赛时，使用的主要的处理器为CPU、GPU以及Google自主研发的TPU处理器。因此可以根据算法运行时使用的处理器规模衡量算法复杂度。

截止目前，DeepMind团队先后在Nature上发表了两篇文章，分别是最初版的AlphaGo和最新版的AlphaGo Zero。但是DeepMind团队对AlphaGo的更新是逐步进行的，根据AlphaGo的战绩，可以分为四个版本。

（1）AlphaGo Fan，AlphaGo的第一个公开版本，因为在2015年10月以5:0的比分战胜欧洲围棋冠军樊麾而得名。该版本AlphaGo使用48个CPU和8个GPU进行计算，分40个搜索线程；同时，DeepMind团队尝试了分布式结构，使用1202个CPU和176个GPU进行计算，同样分为40个搜索线程。经测试，分布式结构的棋力仅有微小提升。

（2）AlphaGo Lee，非公开版本，2016年3月以4:1的比分打败世界围棋冠军李世石。根据相关信息，该版本使用分布式结构，基于48个TPU进行计算而不是GPU。

（3）AlphaGo Master，2017年1月和人类顶尖高手对战取得了60:0的战绩。该版本初步将策略网络和价值网络合并为一个网络，仅使用一台设备，4个TPU进行计算。根据Google最初公开的数据，TPU的计算性能是GPU的15倍以上。

（4）AlphaGo Zero，2017年11月，以100:0打败AlphaGo Lee, 89:11打败AlphaGo Master。该版本更进一步简化了输入特征信息，同样使用一台设备，4个TPU进行计算。但是在训练过程中，使用了60个GPU进行计算，及19个CPU作为参数服务器。

从上面4个版本的算法对比可以看出，如果使用CPU和GPU进行算法计算，顺利运行算法需要50个左右CPU处理器和10个左右的GPU处理器。而在算法训练中，一般需要几十个GPU处理器。算法描述中没有具体给出所用的CPU和GPU处理器型号，但是根据对相关论文的调研和研究，在利用GPU进行神经网络训练时，通常使用Nvidia的GTX系列GPU，该系列GPU专为大型服务器而设，其功耗非常大，单个GPU的运行时功耗通常在100W以上。

通过AlphaGo算法运行环境配置，可知蒙特卡洛树搜索算法的运算需求，主要涵盖支持MCTS搜索过程的运算需求，以及神经网络处理部分的运算需求。

#### 4.1 MCTS搜索过程运算需求

MCTS搜索过程是较为典型的决策树运算过程。决策树算法的特点是根据不同的条件进行判断和分支选择，并且有非常多的条件规则和分支选择规则。在机载应用中，决策树算法的树生成过程和树剪枝过程，即决策树的学习过程，通常在地面服务器中进行，只将优化后的树形结构移植到机载环境中执行。但在某些情况下，同样需要在机载计算系统中实时进行决策树算法的动态生成和调整。

决策树结构可以和if-then规则对应，对于if-then形式的规则，体现在处理器中的操作包括：

1）从存储空间（外部存储器或者内存）中获取规则条件；

2）根据规则条件，从规则数据库中搜索相应规则；

3）依据规则条件，执行逻辑判断，按照规则结果进行分支选择；

因此，决策树算法运行时的计算需求涉及两个方面，一方面是处理器执行数据读写和执行分支跳转类指令的能力；一方面是计算机中的数据库管理能力，反映在硬件的存储器性能和操作系统的数据库系统或文件系统支持。

其中，处理器执行能力，依赖于处理器架构，并受编译器能力影响。处理器的处理能力一方面在很大程度上依赖流水线处理技术，将指令处理流水化，以充分利用运算器和存储器的资源，加快处理速度。而在执行跳转类指令时，由于跳转类指令会破坏流水线，导致流水中断。另一方面，跳转类指令往往需要跨地址进行数据寻址和指令寻址，会破坏存储器中指令和数据的时间局部性和空间局部性，导致取指和取数据延迟增加。因此，处理器的指令处理瓶颈，往往在跳转类指令。处理器执行跳转类指令的能力，是影响决策类算法计算速度的重要指标。除处理器的执行能力外，编译器对决策类算法的计算效率也有影响。决策类算法通常采用高级语言实现，编译器将其编译为机器语言，编译过程中对高级程序语言的优化方式、优化水平，影响生成的机器语言的执行效率，最终反映在算法的指令规模和处理器执行各条指令的效率。

CPU处理器架构核心在于设置了运算器、控制器和存储器，利用专门的控制器控制指令的执行，同时设置了大量的数据寄存器和状态寄存器进行指令的控制和运行。因此，CPU架构特性加上指令集的设置，使其能够较好的支持控制类指令的执行。

另一方面，决策类算法通常需要构建决策树，或者生成if-then规则来存储整个决策算法，而决策树自身的结构复杂度，或者由决策树生成的规则的规模，都非常高。在决策类算法执行过程中，根据每一步的输入条件，需要从众多的分支或规则中搜索满足条件的规则。因此，计算系统对决策树的存储和管理能力，以及对大量规则的管理能力，直接影响每一步的决策效率。具体的，若用树形结构存储决策树，则内存空间大小能否满足决策树的存储规模需求，是影响决策树搜索速度的最大限制。若内存空间不足，则需将决策树的树形结构存储至外部存储器，而外部存储器的访存速度相比于内存，平均会下降一至两个数量级。因此，从内存空间的指标，决策树算法对计算系统的存储空间，尤其是内部存储空间的能力需求较大。若用规则形式存储决策树，则需用文件系统或数据库系统存储和管理众多的规则，规则的存储和管理方式，以及规则查询方法，直接影响规则搜索的效率。因此，计算系统是否具备高效的数据库管理能力，是决策规则能否快速查找的关键。

综上所述，决策树类算法对计算系统的能力需求，主要包括两个方面：计算能力，尤其是处理大量复杂的分支跳转类控制指令的能力；存储能力，尤其是对大量离散化数据的高效管理能力。

#### 4.2 神经网络运算需求

神经网络算法的两大特性即计算密集性和存储密集性。计算密集性体现在神经网络算法的三个方面：（1）计算规模十分庞大，如卷积神经网络算法中较为简单的AlexNet算法，其乘加运算的次数就有几十亿之多；（2）计算指令占整个算法指令数的比例非常高，在神经网络算法中，基于神经网络的简单且标准的网络模型和结构，算法的绝大部分操作都是卷积运算，而卷积运算仅包括乘法和加法运算。（3）深度神经网络算法在执行前向计算时，计算指令分布密集，由深度神经网络的结构，决定了网络的较低层都是标准卷积运算，中间夹杂的诸如dropout等概率选择，也可以转换为乘法运算；同时网络的较高层可能会有一层或多层全连接运算，其运算核心也是乘法和加法。在整个网络进行前向过程中，几乎没有控制类指令。因此，神经网络算法的典型计算密集性特性，使得该类算法十分适合利用具备大规模并行结构的处理器实现。

神经网络算法的存储密集性体现在算法模型的规模。一方面，通过对多种不同应用和不同架构的深度神经网络算法分析可知，深度神经网络算法模型的规模普遍较大，较简单的网络模型，其参数数量都在百万以上，而在计算机中存储时，若采用浮点数存储，则需百兆以上的存储空间。而深度神经网络算法复杂度不断增加，其模型规模必将不断增大，这对计算系统的存储资源提出了很大的需求。另一方面，通过对深度神经网络算法执行过程分析可知，深度神经网络算法执行中，需要频繁访问存储空间以获取卷积核、特征图等矩阵形式的数据，同时需要将每一步运算中产生的新特征图以及更新的卷积核参数写回至存储空间中。因此在网络运行过程中，处理器与存储器之间将会进行频繁且巨量的数据传输，这对处理器内部数据总线的带宽，提出了非常高的需求。

综上所述，神经网络算法对计算系统的能力需求，主要包括两个方面：计算能力，尤其是处理大规模并行化的乘加运算的能力；存储能力，尤其是高速存储空间大小以及处理器与存储器之间的数据总线带宽。

基于上述分析，机载智能计算单元，需要设计高性能CPU主控单元来支持MCTS搜索过程的处理，并设置面向神经网络运算加速的专用协处理器来支持神经网络的快速处理。综合考虑已有技术成熟度，选用高性能CPU主控，多核CGRA联协处理神经网络的方式，进行机载智能计算单元设计。

## （二）可重构计算架构设计

### 1、 CGRA整体架构

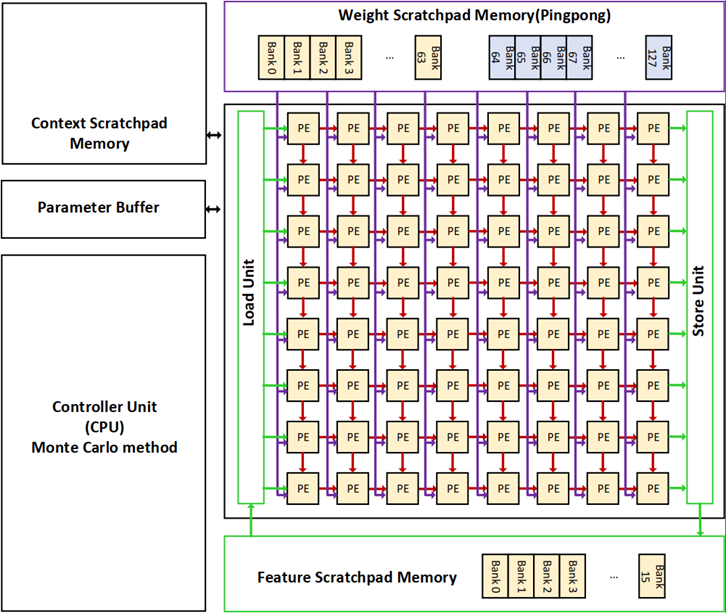


图2.1 CGRA的 SOC系统示意图

上图是整体的可重构计算硬件架构 CGRA（Coarse-Grained Reconfig-urable Architectures）的 SOC（System on Chip）系统示意图，它与主流的可重构体系结构的设计技术是一样的：采用通用的CPU处理器作为主要控制单元, 并包含一个或多个粗粒度可重构单元阵列（PEA，Processing Element Array）等组成的基本硬件架构，多个 PEA 之间往往采用总线方式或者片上网络 NOC 进行集成和通信。

这里的可重构处理单元阵列 PEA 主要作为可重构处理器单元模块（Processing Element, PE）的数据计算通路，主要负责可重构处理器 CGRA 的任务处理和加速工作，主控处理器可与其进行交互，如控制信息发送和运行状态信息的收集等，并控制 PEA 阵列自主发起数据和配置信息（Context，一般包含控制 PEA 运行的具体信息）的搬运工作（一般由专门的数据控制器和配置控制器负责），以及运行状态。

由于可重构处理器的具有“配置一次，执行多次”的时域和空域运行的特点，数据与配置信息的更新差别巨大，为了提高可重构阵列的数据复用和降低访存开销，也会采用数据和配置分离的缓存 Cache 结构。通用处理器的执行程序、可重构处理器单元 RPU需要的数据和配置信息都是事先存储在外部的主存储单元中，然后再随程序的运行不断搬运进和搬运出可重构处理单元的。

可重构处理器在硬件架构层的工作流程如下：

主控制器(Controller Unit)控制可重构处理器的配置控制器及缓存（Context Scrachpad Memory，CSPM）从主存储器加载配置信息到可重构处理单元阵列（Process Element Array）,并控制PEA控制器开始执行配置信息。PEA根据配置信息，通过数据控制器级缓存（Data Scrachpad Memory, DSPM）从主存储器读写数据进行计算，计算到特定状态时候，通过主控制器接口将状态反馈主控制器。系统中的主控制器采用通用处理器，这样可重构处理器可以方便地与现有的操作系统和外设中断等功能集成。

### 2、 CGRA基本结构组成

CGRA基本结构主要分成了四个主要功能：CMU（控制管理单元，Controller Management Unit）、PEA（执行单元阵列，Processing Element Array）、DMA（Direct Memory Access）、DSPM（片上数据缓存，Data Scratch-Pad-Memory）。

#### 2.1 CMU（控制管理单元，Controller Management Unit）

因为可重构计算数据通路的功能变得更复杂了，传统指令集已经很难描述其功能，所以我们采用专用的配置信息（configuration）来控制可重构数据通路的行为。而CMU主要作用就是组织、加载和切换配置信息，将它们在正确的时刻且以合适的顺序发给可重构数据通路。CMU的功能相当于通用处理器的控制器，所需的控制信息存储根据需要加载并存储在本地的配置存储单元。

CMU负责一个完整应用程序代码的执行。包含了CU与MU两个核心组件:

1. CU（Controller Unit）

CU用于流程控制，主要执行程序中的控制流代码。CU的典型使用流程包含指令流与数据流，通常包含ITCM （Instruction Tightly Coupled Memory,）与DTCM（Data Tightly Coupled Memory）：

ITCM，映射在系统地址空间，实际存储内容由编译器决定；

DTCM，映射在系统地址空间，实际存储内容由编译器与程序员共同管理。堆栈区域由程序员负责，其他部分由编译器管理。

CGRA采用了高性能CPU作为CU的核心单元，CPU包含了运算器、控制器与存储器的架构，结合大量的数据寄存器与状态寄存器支持，加上指令集的设置，使其能够较好的支持如MCTS搜索等控制类指令的执行。

1. MU（Management Unit）

用于管理、调度加速器，主要用于管理数据流。 MU的典型使用流程用到的CSPM（Context Scatch-pad-memory）与PB（Parameter Buffer）。

CSPM是一段CGRA Context缓存区，在典型执行流程中不映射到系统地址空间，由编译器调度管理。因此编译器需要管理一份“缓存区地址空间Context”到“系统地址空间Context”的映射关系，以保证CGRA可以正确工作；

备注：考虑到系统软件的简单，优先考虑通过硬件管理“映射关系”

PB是一段实时的参数空间，CU将TASK执行需要的参数通过PB/MU传给PEA阵列用于执行。

配置控制器通常是一个可编程的有限状态机，按照主控制器的控制，以及数据通路的计算结果，选择后续的配置信息。为了减少配置信息加载对性能的影响，可重构处理器通常都会有片上的配置缓存。尤其对于支持快速动态重构的动态可重构处理器，配置缓存必不可少。

#### 2.2 PEA（执行单元阵列，Processing Element Array）

PEA（执行单元阵列，Processing Element Array），主要负责加速数据流代码的执行。可重构数据通路一般由PE单元和互联网络两部分组成。PE单元是可重构计算的基本单元。与众核通用处理器中的各个核相比，PE单元不是独立的处理器结构，用于完成计算操作。按照PE单元的数据位宽，可重构数据通路可以分为细粒度(1-bit)和粗粒度(大于1-bit) 两种。

动态可重构处理器与通用处理器最显著的区别在于，可重构数据通路通常采用大量空间并行的计算处理单元（PE，Processing Element），进而组成一个结构规整的阵列式结构PEA (Processing Element Array)。因此，互联网络是可重构处理器区别于通用处理器的重要特征。按照PE单元操作间的“生产者-消费者”关系，PE运算数据在空间利用互联网络快速流动。互联网络具体组织形式不固定，一般而言越灵活的互联实现代价越大。PE单元和互联网络组成了PEA，是可重构计算范式中空域-时域自由度的主要载体。它的功能更接近于通用处理器中执行单元及相关支持硬件。PEA主要包含PE、RC、LSU等核心组成单元，具体功能如下：

1. PE（处理单元，Processing Element）

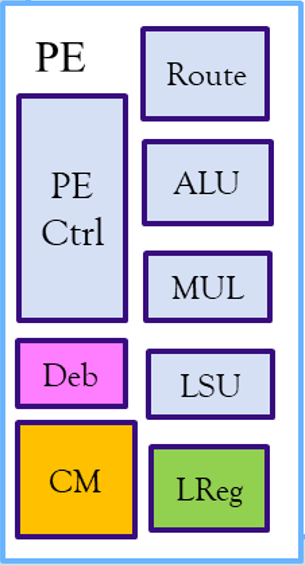


图2.2 PE处理单元结构

PE是具备数据处理能力的功能单元的统称，通过加法器、MUX等硬件结构支持一系列的基础算子，通过配置可以实现不同的计算功能，如图2.2所示。其中的Context Memory(CM)存储该PE的配置信息，在PE工作时候，PE Ctrl从CM中取出配置信息进行取址译码操作，根据配置信息结合硬件单元将PE的数据通路与计算单元配置。

PE是可重构处理器应用任务最基本的计算单元，其所支持的计算位宽，也就是我们所说的计算粒度，一般都是按 Word/Byte 等位宽大小的数据进行处理的。由于计算粒度比较粗（相对于 Bit 级的细粒度可重构处理器而言），其硬件实现的面积和功耗开销均较低。另外，可重构处理单元支持运算操作的种类，也是影响可重构处理器计算能力一个关键因素，支持的功能越多越复杂，硬件开销就会越大。由于可重构处理器主要针对于一些数据密集型和计算密集型应用任务的加速处理，所以大多数情况每个 PE 单元都是支持一些常用的算术逻辑运算（Add、Mul 等），阵列左右两侧（每行第一个和最后一个）的PE还支持访存（Load/Store）运算。

每个 PE 包含一个局部寄存器堆 LR（图中的LReg），同样采用数据和配置信息分离存储的方案设计，降低 LR 的面积，并提高 LR 的利用率。

其中 LR 可以存放用于运算的数据和用于迭代的数据。本地数据寄存器，用于存放运算的输入和输出数据，可以供PE的运算单元访问，包括 ALU 运算的操作数 input1、input2 和 input3，以及 LSU 运算的 Addr 和 Store-data 的输入；本地迭代寄存器，用于存储配置信息中的迭代和 Idle信息，只能被配置中的迭代字段读取（具体参加后面的解释）。

LR都能够被 PE 运算写入，通过 PE 单元的 data\_out 通道将输出结果写入 LR。

另外需要注意，PEA 必须写入配置后，配置非空时，PEA 才会自动启动时钟，此时才能对 PEA 进行数据的读写和计算处理，包括 GR 和 Scratchpad Memory 的访问等。

另外需要特别注意，PEA 必须写入配置后，配置非空时，PEA 才会自动启动时钟，此时才能对 PEA 进行数据的读写和计算处理，包括 GR 和 Scratchpad Memory 的访问等。

（2）GR（全局寄存器链/堆，Global Register file）

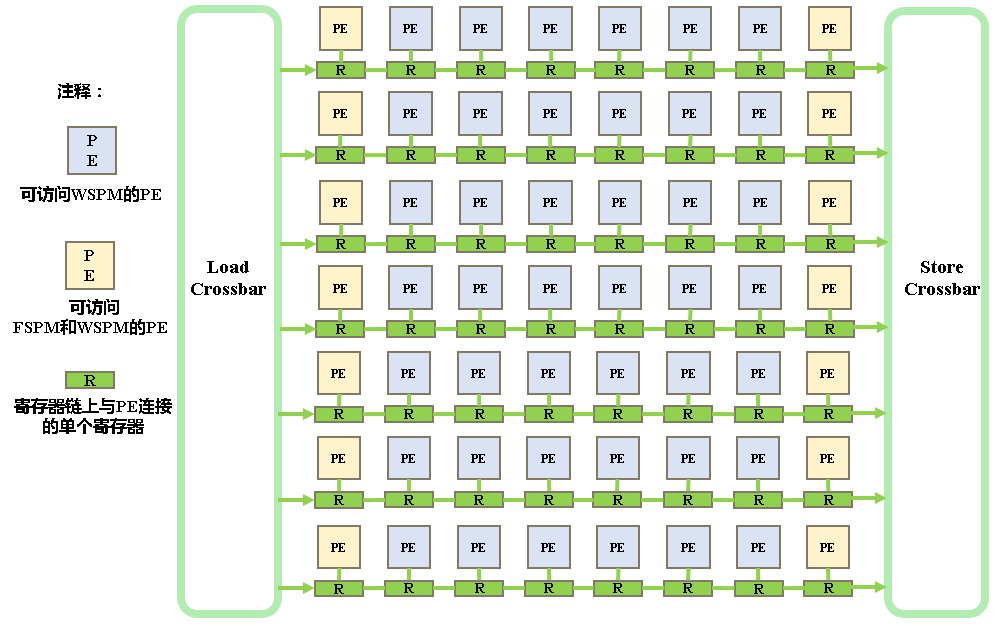


图2.3 全局寄存器链结构

CGRA配置了8行寄存器链，每行寄存器链8个reg，分别与对应PE互联。寄存器链的设计考虑到对以下问题的支持优化：

面向矩阵滑窗相关问题：feature来自矩阵滑窗，每次重复读取重叠区域feature效率低。考虑用寄存器链模拟滑窗，降低访存操作。

辅助实现矩阵reshape相关操作：例如向量乘以矩阵在8\*8阵列上会获得64个结果，需要从空间上的8\*8转换成1\*64，存入同一个bank。在阵列上做routing，降低性能。数据存入对应寄存器链，阵列空闲可以提供更多的算力为下一个算法。

对于矩阵访存相关算子，利用寄存器链设计帮助访存操作，节省阵列算力，增大任务级并行性。

（3）LSU (Load/Store Unit)

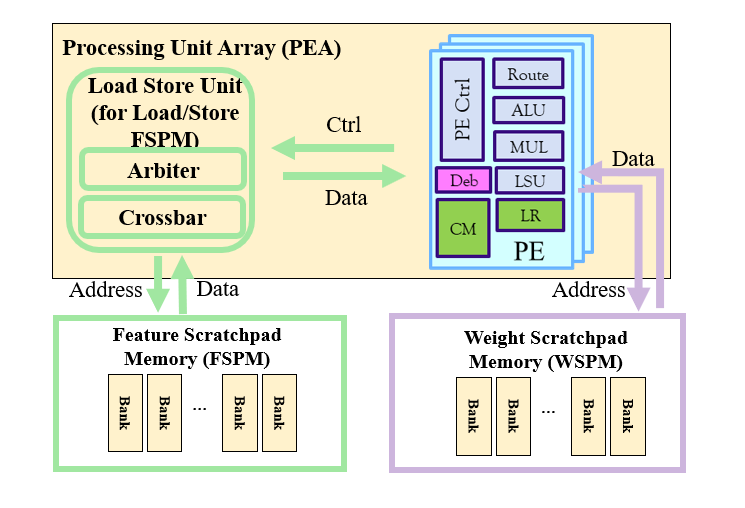


图2.4 LSU单元

LSU充当PEA与DSPM之间的传输中介，可根据配置模式选择寻址方式计算内部缓存空间地址，从缓存中读取数据，并同时返回到计算阵列中。

Load/Store单元需要具备的特性如下：

Load/Store单元最多同时从计算单元接收M个访存请求；

* + 1. 支持多种寻址模式：

立即数寻址Load/Store[Offset]，Offset来着Load/Store局部寄存器；

寄存器寻址Load/Store[Base]，Base来自计算阵列；

基址相对寻址Load/Store[Base+Offset]，Base来自计算阵列，Offset来自Load/Store局部寄存器；

* + 1. 支持访存冲突检测与仲裁，假设LD/ST模块发起M个请求同时访问N个SPM-Bank。LD/ST需要将M个请求加入到N个访存队列中，将冲突的请求顺序完成。等待M个请求均完成以后，同时将结果返回到计算阵列中。LD/ST需要反馈信息给PEA，PEA暂停整个阵列的执行。
    2. 支持数据格式转换。当DSPM中的数据是以字节（Byte）或者半字（half word）为粒度的数组时，比如PE支持的Float Point 16-bit与Integer 8-bit的两种数据格式。LSU可通过切换通道位宽来支持两种数据格式之间的转换

我们面向智能机载算法计算的特点，根据计算访存比设计了PEA的LSU接口数量：LSU没有采用将Memory bank与接口灵活性互联的crossbar设计，因为考虑到我们的算法访存需求比较高而相对不需要权重与PE之间有较多的灵活性设计。因此我们将144个Memory Bank分成高灵活性的Feature和低灵活性的Weight两部分，以减少大量互联带来的硬件开销，Weight Bank 每两个Bank绑定一个PE的Load，以此实现乒乓存储。16个Feature Bank可以与最左侧与最右侧的任意PE与LSU互联。

#### 2.3 DMA（Direct Memory Access）

DMA是CGRA实现内部缓存（DSPM）与外部Memory交换数据的模块。分成两个主要的功能RDMA与WDMA。

1. RDMA将外部Memory的数据搬到内部DSPM中。在两个地址空间中同时支持Gather/Scatter
2. WDMA将内部DSPM的数据搬到外部Memory中。在两个地址空间中同时支持Gather/Scatter
3. 支持AXI总线接口：
4. 支持Outstanding；
5. 地址（Address）与数据长度（Length）需要支持1字节（Byte）粒度；（目前仅RDMA有此需求，WDMA尚未明确该需求）；
6. 支持ISP Linebuffer的特殊接口；
7. DMA支持通途的压缩、解压缩功能；（待确认澄清）；

#### 2.4 DSPM（片上数据缓存，Data Scratch-Pad-Memory）

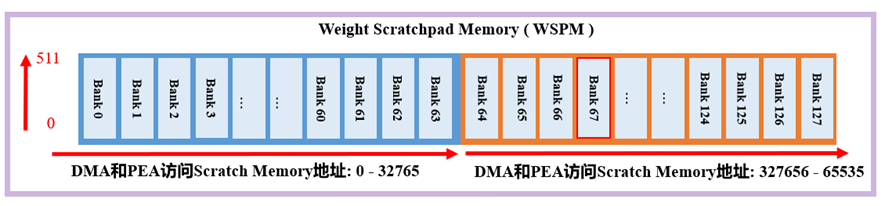


图2.5 Weight Scratchpad Memory (WSPM)

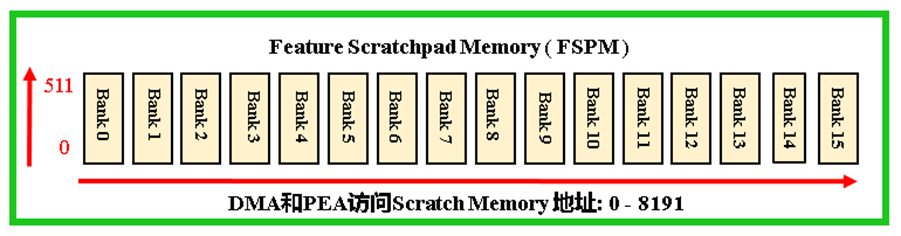


图2.6 Feature Scratchpad Memory (FSPM)

DSPM是一块CGRA独有的Buffer，典型的执行过程中不映射到系统地址空间（Debug功能可以映射到系统地址空间），包含以下特点：

1. Buffer地址空间通过编译器以软件的方式管理；
2. 支持多BANK划分；
3. 支持多Master（Load/Store、DMA、CMU）访问DSPM的视角一致；
4. DSPM支持可配置的地址交织方式：高位交织；[9:0]：高位[9:8]，低位[1:0]，[6:5]。

为了减少互联开销，将Bank划分为高灵活性(Feature Scratchpad Memory)和低灵活性(Weight Scratchpad Memory)的两个部分：

Weight Scratchpad Memory(WSPM)每两个bank绑定一个PE的load用来进行乒乓存储; Feature Sratchpad Memory(FSPM) 有16个Bank，可以与最左侧和最右侧的任意PE的LSU互联。该设计考虑到了DFG在阵列映射上普遍具有方向性的特点，以经典的脉动阵列计算矩阵乘法为例，矩阵输入流与结果的输出流都具有典型地单项流动的特点。因此不需要采用Crossbar的设计将阵列的每一个PE都与FSPM进行连接，FSPM在适度放弃灵活性的同时一定程度上节省了互联的设计开销。

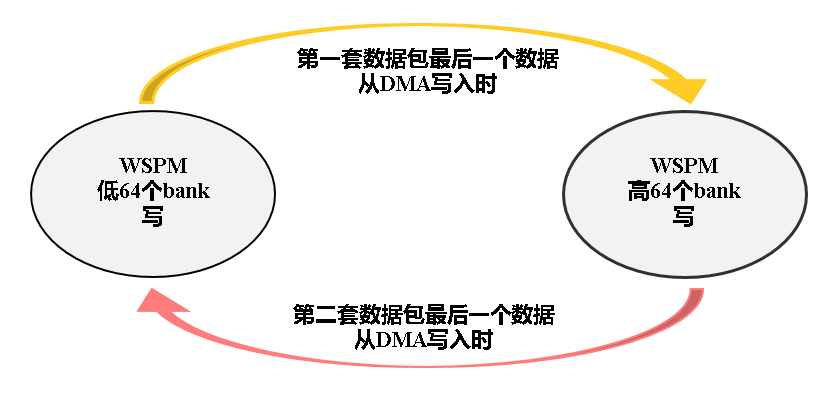


图2.7 WSPM与主存的数据关系

WSPM模块采用了特殊设计的乒乓结构，作为一种数据缓冲优化设计技术，乒乓结构可以看成是另一种形式的流水线技术。输入的数据流在通过“输入数据流选择单元”时，时间等分地将数据流分配到两个数据缓冲模块——即WSPM的上下两端内。该模块总共有128个 BANK，每个BANK的规格是 512×32 双端SRAM，通过DMA请求的数据。

WSPM的乒乓设计如下图所示：

在第一个缓冲周期，将输入的数据流缓存到低 64位BANK 中，长度为length1 (小于等于 512);

在第二个缓冲周期，通过“输入数据流选择单元”的切换，将输入的数据流缓存到高 64位BANK 中，长度为length2 (小于等于256),同时将低 64位BANK缓存的第一个周期的数据通过“输出数据流选择单元”的选择，送到相应的PE进行运算处理；

在第三个缓冲周期，通过“输入数据流选择单元”的再次切换，将输入的数据流缓存到低 64位BANK, 同时将高 64位BANK缓存的第二个周期的数据通过“输出数据流选择单元”的切换，送到相应的PE进行运算处理，如此循环往复。

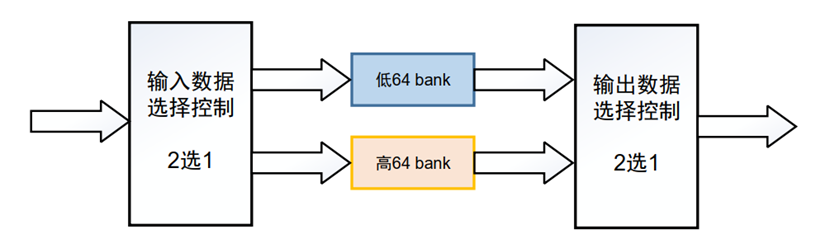


图2.8 WSPM中的乒乓结构

这样的结构，或者说技术，有着以下的特点：

（1）输入数据流和输出数据流都是连续不断的，没有任何停顿，因此特别适用于对数据流进行流水线式处理。因此，乒乓操作法常应用于流水线式算法，完成数据的无缝缓冲与处理；

（2）需要使用双倍的存储器资源；

（3）适用于数据来不及每次传输都进行处理，需要缓存的情况。

### 3、 CGRA访存通路

#### 3.1 访存通路介绍

本节通过一个典型的执行流程说明，CGRA的访存通路能力。CGRA的访存通路需要在三个方面提供竞争力：

1. 总线的带宽

DMA提供三种访存方式，Contiguous address space、Gather、Scatter，任意一种方式均需要将总线带宽充分利用。为了充分利用总线带宽，软硬件需要满足以下能力：

1. Burst能力，单次的连续访问需要以Burst方式进行。因此单次的数据长度BurstLength = BurstNumber \* bus\_width；
2. Outstanding能力，Outstanding能力分成两种情况：

单次连续地址的数据访问超过Burst支持的最大数据长度。将一次连续的访存，拆成多个Burst，为了充分利用带宽连续发送多个Outstanding请求出去。其中，Outstanding能力需要满足以下等式：

BusBandwidth \* Latency = MinOutstandingBurstNumber \* BurstLength

其中，BusBandwidth表示我们需要的总线带宽，典型情况为最大带宽；Latency表示Master访问Memory的数据通路延迟；MinOutstandingBurstNumber表示最小Outstanding Burst请求的数量；BurstLenght表示burst数据长度，典型为最大Burst数据长度；

备注：Outstanding情况下，需要足够大的数据Buffer保证Master不反压总线。

第二种情况：Gather、Scatter，多次连续地址的数据访问。多个Gather请求，需要以Outstanding的请求方式发送出去。此时，BusBandwidth \* Latency = OutstandingBurstNumber \* BurstLength中BurstLength受限于Gather/Scatter请求，因此BusBandwidth可能无法充分利用。

1. 高效ASIC数据格式转换

DMA的首要任务是充分利用总线带宽，其次，DMA要提供典型的数据变化能力，实现DSPM内部的数据格式转换。基于DSPM存储空间，DMA需要提供多种数据格式变换的能力，包括：

1. MatrixTranspose；基于DSPM-BANK的最大位宽，实现基于子矩阵转置的矩阵转置；
2. SPM Gather；
3. SPM Scatter；
4. LSU模块实现PEA与DSPM之间带宽匹配，处理PEA与DSPM在物理实现中“位宽”与“数量”的差异。
5. 典型情况下，DSPM总容量为512KB，分成32个BANK，单个BANK容量为16KB，规格为128bit宽\*1024深；如下是TSMC28HPCPLUS，ssg0p9vm40c的SRAM面积情况：
   * 1. 最大提供：2048\*144；
     2. 典型情况下：1024\*128在面积、容量、位宽上是较好的选择；
6. PEA阵列通过LSU模块访问DSPM，LSU提供的能力包括：
   * 1. 在“PEA-LD/ST请求”与“DSPM-BANK访存端口”之间匹配“数量”与“位宽”的差别。实现访存带宽的匹配，满足：

LD/ST \* 32bit = BANK\_Number \* BANK\_width；

* + 1. LSU具备处理Memory Bank冲突的能力。LSU允许多拍完成所有的LD/ST请求，并反压住PEA阵列，当所有请求的数据准备好以后，同时反馈给PEA阵列，PEA阵列不感知多拍的处理。（目的：简化软件编程，尤其是边界处理的困难，但是性能优化仍需要避免BANK冲突）；
    2. CouplingOperations：最大支持128个32bit的LD/ST；此时需要满足以下条件：
       - 128个LD/ST，分成32组CouplingLoad或者CouplingStore，每个CouplingOperations访问128bit宽，128bit地址对齐的DSPM空间。
       - LD与ST之间，无法组成CouplingOperations；
    3. LoneOperation：最大支持32个32bit的LD/ST；此时
       - 32个LD/ST，任意两个LD或者任意两个ST不访问连续的128bit地址空间（该连续地址空间128bit地址对齐）；
    4. MixOperation：同时存在CouplingOperation与LoneOperation，LSU支持对LD/ST自动分类：CouplingOperation或者LoneOperation；在不冲突的情况下，同时发起所有的CouplingOperation与LoneOperation；

#### 3.2 典型访存通路



图2.9 CGRA执行的典型流程

如图2.9所示，典型的CGRA执行流程分成如下步骤：

1. DMA从DDR（或者OCRAM）搬运数据到片上的DSPM中。需要DMA充分利用总线带宽。
2. DSPM-DMA通过ASIC的数据搬运在DSPM中完成数据格式变换。当ASIC功能无法支持时，通过阵列完成。
3. 阵列执行“循环加速任务”，通过LSU模块实现PEA与DSPM之间带宽匹配，处理PEA与DSPM在物理实现中“位宽”与“数量”的差异。

#### 3.3 CGRA同步机制

CU与MU是CGRA内部的两个主要的控制Master，CU与MU通过控制流同步共同完成一项工作；

1. 控制同步
2. CU与MU间同步

CU通过LOAD/STORE访问MU的寄存器，给MU发送执行命令；

CU访问MU的启动寄存器，支持阻塞式启动与非阻塞式启动两种：

阻塞式启动，当且仅当CU发送给MU的命令执行完成后，CU的STORE操作才会完成；

非阻塞时启动，当CU给MU 发送启动命令后，CU的STORE操作立即完成，CU可以继续执行后续操作；

1. MU内部同步

MU内部的控制流同步，通过指令队列完成。MU内部有四条指令队列：RDMA、WDMA、B2B、PEX，不同的队列之间的指令可以并行执行，同一个指令队列内的指令按照程序序顺序执行；

并行的指令队列之间通过Sync.id

PEX与B2B有两个Sync.id；

RDMA与WDMA仅有一个Sync.id；

1. 数据同步
2. CGRA内部无数据同步机制，包括但不限于访问SRAM的仲裁、Coherence与Consistency等；CU与MU独占并分别管理各自的SRAM；

CU管理的SRAM资源包括：ITCM、DTCM；

MU管理的SRAM资源包括：CSPM、DSPM；

1. 通过“控制流”管理“数据同步”

CU需要访问DSPM的数据时，需要保证CU发到MU中的命令（RDMA/WDMA/B2B/PEX）均已经完成，否则将造成DSPM访问冲突；

“CU通过DMA将PEA的执行结果从DSPM搬运到DTCM（此非典型通路）”或者“CU通过DMA将一些数据从DDR搬运到ITCM或DTCM（此过程是一种典型的计算过程）”，需要CU执行阻塞式DMA，当发送DMA以后CU就挂起任务，直到DMA执行完成，CU再继续执行计算；此时，DMA与CU是串行工作，DMA仅作为CU一个高速的搬运数据部件，并不能DMA与CU并发工作。

### 4、可重构计算单元设计

#### 4.1 PE实现原理

图3.13 PE 结构原理图

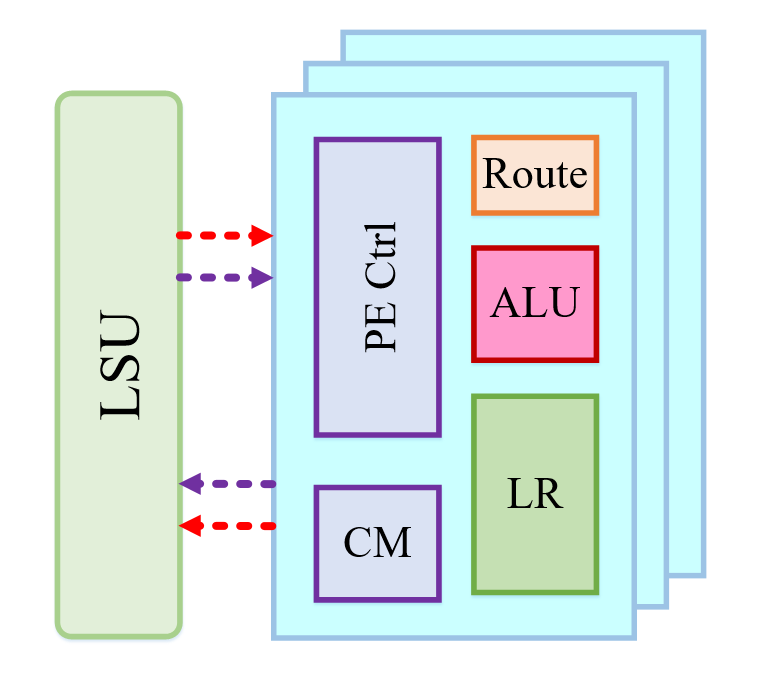


图2.10 PE结构图

PE是可重构处理器应用任务最基本的计算单元，也是整个可重构计算执行的核心部分。作为最底层的计算载体和功能实现部分，他还需要考虑粒度选择、寄存器设置、和功能选取。其支持运算操作的种类数量极大程度地影响着可重构处理器计算能力，即支持的功能越多越复杂，硬件开销就会越大。由于可重构处理器主要针对于一些数据密集型和计算密集型应用任务的加速处理，所以大多数情况每个 PE 单元都支持一些常用的算术逻辑运算（ADD、MUL、MAC等）。

从可重构计算单元向外部看， 输入需要接受来自阵列输入接口的数据、 其他计算单元的输出数据、 阵列内部缓存的输出数据等， 输出需要定义是否驱动输出互连线。这些输入和输出的选择单元实现了可重构路由功能。而阵列左右两侧（每行第一个和最后一个）的PE还支持访存（Load/Store）运算。

因此，我们根据其需要实现的功能对PE进行了设计。如图 3.13 所示，作为可重构阵列的处理单元，我们需要关注 PE内部的核心功能部分主要是：PE 控制器 Ctrl、配置存储 CM、局部寄存器堆 LR、PE 路由器 Router、PE 的主要执行单元ALU。

1. PE\_Ctrl模块

PE\_Ctrl单元作为PE执行的存储控制模块，可以根据已有配置信息发出使能信号，实现对单个PE单元的行为控制。

1. CM模块

CM（Conetxet Memory）单元存储了PE执行计算所需要的全部配置信息，包括用于存储配置信息中的迭代和 Idle信息，其只能被配置中的迭代字段读取。

本项目中的处理单元 PE 的配置信息全部采用 64-bit 的配置信息格式， 目前用到的主要包含三大类：

* + 1. 顶层型配置信息
    2. ALU 运算型配置信息（还包含 MUL/MAC 类运算）
    3. 访存运算型配置信

这些配置信息的主要特点有：

1. 适用于目前项目中大规模粗粒度阵列的设计要求
2. 高灵活性、高扩展性
3. 支持多种运算类型、阵列大小、储存大小、不同数据位宽等
4. 存在配置长度扩展位和许多配置编码保留位，用以实现未来配置信息扩展等
5. LR模块

寄存器在同步电路中是必不可少的。合理的寄存器位置有利于实现流水线等算法加速执行的手段。算法映射的过程中也需要寄存中间数据。

阵列同时计算的数据带宽一般情况下大于外部存储带宽， 需要拆分成多次输入， 所以输入完成之前， 数据需要寄存。因此基本单元的输入端使用两组到三组寄存器单元， 对应二目或者三目操作。输出的情况也是同样， 输出端也需要设置寄存器。输入输出寄存器是否需要放入阵列计算的流水线中， 则可以根据流水线的时序要求灵活掌握。如果时钟的要求不高，为了算法映射的便利， 流水线中接入输入或者输出寄存器之一即可。另外的寄存器提供旁路。

图3.11给出了计算单元中寄存器位置的示例。借鉴通用处理器的设计经验， 由于数据存在局域性原理（ 如果一个数据被访问， 那它或者它附近的数据很可能在较短的时间内被再次访问）， 在可重构计算单元中也可以设计堆寄存器， 用来存放最近几个周期可能使用的数据。三目运算单元的输入寄存器可以和堆寄存器复用。

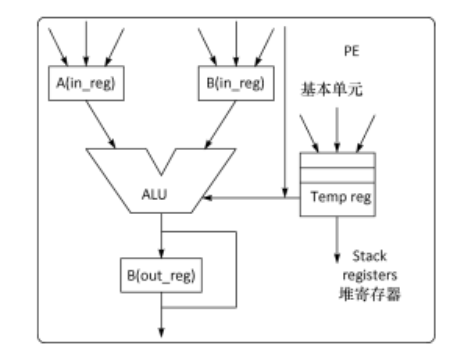


图2.11 PE中的寄存器示例图

每个 PE 包含一个局部寄存器堆 LR（图中的LReg），同样采用数据和配置信息分离存储的方案设计，降低 LR 的面积，并提高 LR 的利用率。

其中 LR 可以存放用于运算的数据和用于迭代的数据。本地数据寄存器，用于存放运算的输入和输出数据，可以供PE的运算单元访问，包括 ALU 运算的操作数 input1、input2 和 input3，以及 LSU 运算的 Addr 和 Store-data 的输入。LR都能够被 PE 运算写入，通过 PE 单元的 data\_out 通道将输出结果写入 LR。

PEA 必须写入配置后，配置非空时，PEA 才会自动启动时钟，此时才能对 PEA 进行数据的读写和计算处理，包括 GR 和 Scratchpad Memory 的访问等。

另外需要特别注意，PEA 必须写入配置后，配置非空时，PEA 才会自动启动时钟，此时才能对 PEA 进行数据的读写和计算处理，包括 GR 和 Scratchpad Memory 的访问等。

1. ALU模块

对于可重构的计算基本单元， 可配置功能集合的选取很重要， 与通用处理器指令集中的操作码（operating code ， Opcode） 定义类似。表2.1给出了ALU常见的功能选择， 有算术、 比较、 逻辑和移位等几个大类的操作。

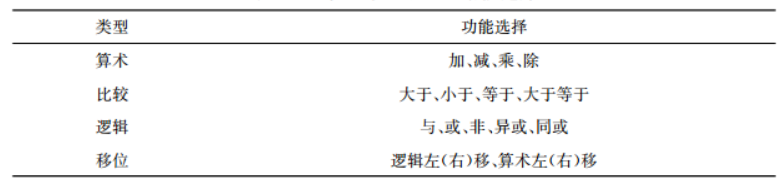
由于阵列单元存在大量重复的情况， 因此在所有单元中加入一个面积很大但是使用频率较低的硬件电路， 如乘法器是很不经济的行为， 这就涉及异构的概念。异构的情况可以分为两类, 一类是广义上，从功能出发，算法映射需要不同的结构， 如路由单元、 缓存结构、 控制器等， 它们和计算单元设计不同， 又不可或缺。

表2.1 ALU实现的计算或逻辑功能

另一类是从效率出发， 优化功耗和面积等。在理想情况下， 设计一个大而全的计算单元是灵活的， 但是有些功能的使用率很低， 闲置也会导致面积和功耗的额外浪费， 在这种情况下应该对单元进行适当裁剪以得到基本计算单元的异构形式。

算法需求和映射效率（ 硬件效率） 在很多情况下是统一的， 如饱和运算使用比较运算也可以实现， 只是占用更多的单元。一方面硬件的效率降低， 另一方面导致阵列计算资源消耗过多。算法需要拆分更多的ＤＦＧ图（ 阵列可映射的ＤＦＧ 图规模更小）， 降低了映射的效率。绝对值运算的选择也是如此。因此， 基本计算单元功能设计单项之间并不独立， 同样的算法存在多种基于指令集的表达方式。

1. Route模块

可重构计算单元输入和输出之间的连接关系， 以及阵列内部外部存储单元的接口设计， 都是可重构路由单元需要讨论的范畴。

对于阵列， 计算单元行或者列之间多采用总线、mesh连接 （ 邻域相连） 和crossbar结构。总线和mesh结构代价较低， 但是灵活性比crossbar差。crossbar结构表示输入和输出之间两两相连。根据乘法原理， 这种结构的硬件开销是非常大的， 但是其层间灵活性又被算法DFG图映射过程普遍需要。当阵列规模增加到一定程度时，crossbar的全互连代价可能无法承受， 尤其在阵列的输入输出端。局域化的互连方式可能是一种出路。

另外，值得注意的是， 从PE阵列的角度来看，由于数据输入的要求，LSU单元与部分PE单元具有本征的配对关系，即位于PE阵列最左端和最右端的2×8个PE与LSU进行绑定，固定地对WSPM进行读取和写入。

#### 4.2 面向Scratchpad接口异构设计

根据可重构处理阵列中的处理单元 PE 支持的计算功能种类，可以分为同构可重构处理器和异构可重构处理器两种。前者由于所有的 PE 单元具有完全相同的功能，应用能够更好的在其上进行映射，对于编译器的实现来说也会更简单些。但是硬件实现开销大，有时候导致资源浪费。后者是不同的 PE 具有的不同的运算功能，比如访存运算和乘法运算等，尤其针对不同的应用和不同的阵列规模，能够在有效降低硬件开销的同时却能带来相同或者略微下降的计算性能。因此，异构可重构处理阵列设计已经成为可重构处理器设计的趋势。



图2.12 PE与DSPM数据关系

为了支持异构设计，我们将片上数据缓存DSPM（Data Scrachpad Memory）分割成了高灵活性的FSPM (Feature Scratchpad Memory) 和低灵活性的WSPM (Weight Scratchpad Memory), 在PEA中每一个PE单元都可以通过LSU访问WSPM的两套乒乓bank堆中的其中一个bank，在此基础上最左侧与最右侧的PE可以通过Crossbar访问FSPM的任意一个bank。这样的一个异构可重构处理阵列上的PE与DSPM的关系可以通过下图进行了进一步的描述。如图PEA最左侧与最右侧的PE，可以通过LSU访问FSPM的任何一个Bank, 此外每一个PE都能够访问WSPM两套乒乓bank中的对应bank。图中对这两类异构PE用颜色进行了区分.

这种实现的计算和访存异构的可重构处理单元阵列，也可以使得内部计算 PE 快速从边沿 PE 取得访存回来的数据，在多数情况下可以很好的实现“数据从一个边沿的访存 PE 输入、通过内部计算 PE 的处理、从另一边沿的访存 PE 输出”的空间计算模式。

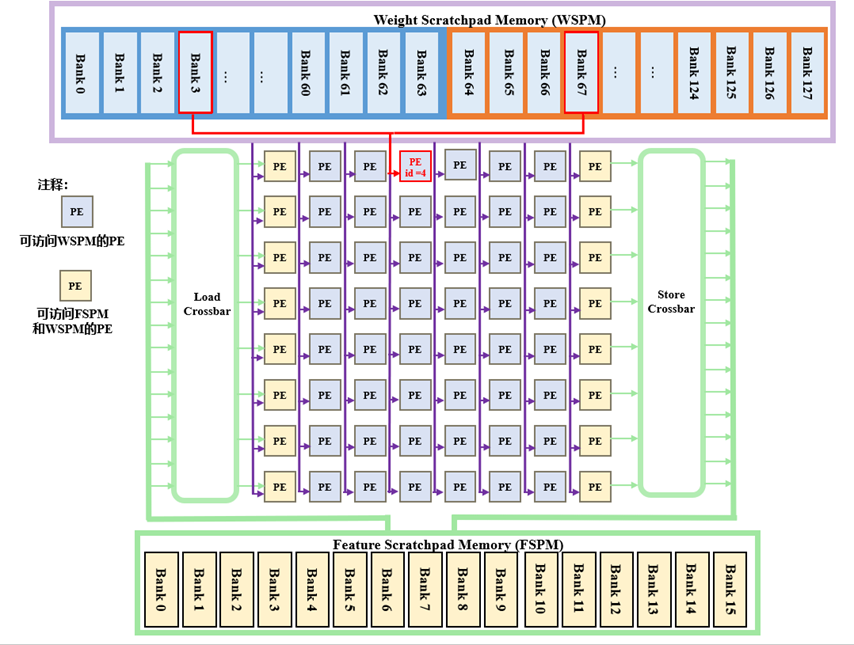


图2.13 面向LSU接口的异构PE设计

#### 4.3 位宽可重构设计

在架构中CGRA的PE是一种位宽可重构的模块，有Float Point 16bit模式和Integer 8bit模式，PE数据通道的物理位宽是16bit,在8bit模式下，计算被分解成了两个细粒度的8bit，在输出通道被重整成16bit。这样的设计可以满足存储位宽与输入位宽充分利用。图2.19和图2.20展示了16bit与8bit模式下PE的通路设计。

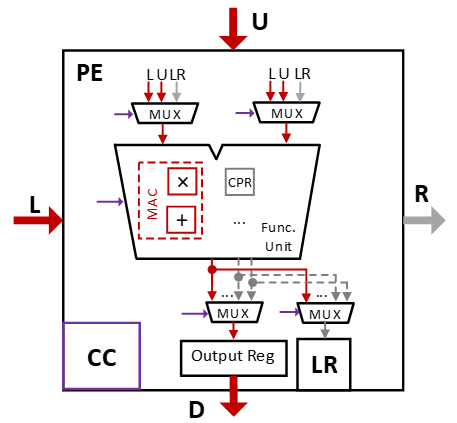


图2.14 16 bit模式

16Bit信号来自于PE的左方向或上方向，在进入ALU运算单元之前经过两个数据选择器，均受Context Controller发出的控制信号控制。经ALU进行运算处理之后的数据，若有迭代需求则存到局部寄存器LR中，若需要进行传递则经由Output Reg进行输出，至邻近的PE或LSU单元。

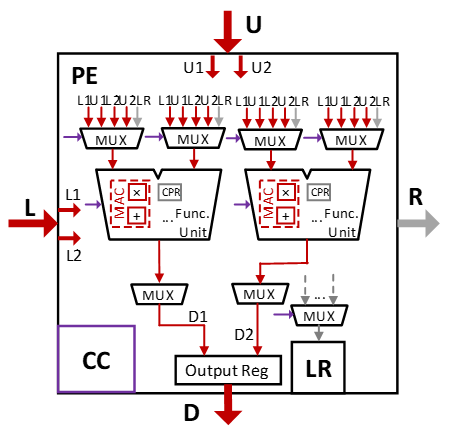


图2.15 8 bit 模式

8bit模式下的计算模式与16bit大致相同，不过由于粒度更细，所需要的配置信息更为复杂，消耗的硬件资源也更多。8bit模式下需要的数据选择器和ALU运算单元都是16bit的两倍，16bit信号在输入时被分解成了两个细粒度的8bit信号，经数据选择器选择、ALU运算后再Output Reg模块被重新整合为16bit，这样的设计实现了位宽的可重构，支持混合粒度的运算增加了灵活性，并且可以满足存储位宽与输入位宽充分利用。

#### 4.4 模拟器PE Context Memory配置

|  |  |  |
| --- | --- | --- |
| 结构体LOOP（外部输入INX start;INY step;INZ end） | | |
| unsigned int | passive\_beat | 被动模式的beat信号（触发beat通道 0~7 对应PE ；8, 9对应lsu in；10对应start接口，不使用被动模式则默认16） |
| int | step\_opcode | Opcode=00 add    01 >>    02 << |
| int | cpr\_mode | END条件的比较模式，  cpr\_mode=00 则判断<      01 判断>      02 判断== |
| bool | loop\_mode | 传递end，0意味着保持原来end的loop标记，1意味着把原来end的loop标记+1 |
| 结构体IF（inx a  iny b  inz c） | | |
| int | mode | 00 判断a< b      01 判断a<=b      02 判断a==b      03 判断a==1 |
| bool | if\_pattern | 是否输出beat  1是 0否 |
| int | result | 00 输出1,0      01 输出a,b      02 输出b,c      03 输出c,c |
| 结构体ROUTE（inx a） | | |
| uint8\_t | route\_delay | ROUTE延迟数 |
| 结构体ARITH（inx a  iny b） | | |
| int | mode | 00 add 加      01 sub 减      02 abs绝对值      03 NEG取负数      04 mul乘      05 mac乘加 |
| 结构体LOGIC（inx a  iny b） | | |
| int | mode | 00 AND 与      01 OR 或      02 NOT非（取反）      03 XOR 异或 |
| 结构体SHIFTER（inx a  iny b） | | |
| int | mode | 00 uSRL uA[31:0]>>uB[4:0]  01 sARL sA[31:0]>>sB[4:0]  02 uSLL uA[31:0]<<uB[4:0] |
| struct NONLINEAR | | |
| int | mode | 00 relu  01 sigmoid  02 leakyrelu |
| struct PECSPM\_TOP | | |
| unsigned int | CSPM\_total | CSPM指令数量  数据位宽是4bit |
| unsigned int | CSPM\_TOP\_renum | CSPMTOPָ指令重复次数 |
| struct PECSPM | | |
| bool | datawidth\_mode | 数据位宽模式  0: int8 1:fp16 |
| unsigned int | CSPM\_renum | CSPM指令循环次数 |
| unsigned int | enable\_beat | 触发beat  选项0123 对应PE右下左上4个通道；45对应lsu in1和2；  6对应start接口；  大于6越界  数据位宽3bit |
| unsigned int | INX | 选项 0123 对应PE右下左上的输入 ；  4对应lsu in1权重；  5对应lsu in2向量678对应lr [0][1][2]；  91011对应立即数imm；不使用则15，大于15越界数据位宽4bit |
| unsigned int | INY |
| unsigned int | INZ |
| unsigned int | PE\_OUT[0] | PE上方输出通道的选择，数据位宽4bit.  选项的0123对应复用左上右下的PE输入通道；45对应输出到lsu；67cal结果；15不输出 |
| unsigned int | PE\_OUT[1] |
| unsigned int | PE\_OUT[2] |
| unsigned int | PE\_OUT[3] |
| unsigned int | PE\_lr[0] | PE本地寄存器0的输出选择，数据位宽3bit，选项的0123对应复用左上右下的PE输入通道；45对应输出到lsu；67cal结果；15不输出 |
| unsigned int | PE\_lr[1] |
| unsigned int | PE\_lr[2] |
| unsigned int | PE\_lsu | PE的load/store单元的输出选择，数据位宽3bit，选项的0123对应复用左上右下的PE输入通道；45对应输出到lsu；67cal结果；15不输出 |
| unsigned int | PE\_gr | PE的全局寄存器链的输出选择，数据位宽3bit，选项的0123=复用左上右下方向的PE输入通道；45=输出到lsu；67=cal结果；15不输出 |
| bool | PE\_beat\_mode | 选择使能输出信号的模式，数据位宽1bit  0 =EN信号使能  1 =END信号使能 |
| unsigned int | PE\_opcode | 选择PE的运算模式，数据位宽3bit  0=循环Loop  1=路由数据Route  2=条件判断if  3=算术arith  4=逻辑运算logic  5=数据移位shift  6=非线性nl |
| LOOP | | PE\_LOOP |
| ROUTE | | PE\_ROUTE |
| IF | | PE\_IF |
| ARITH  LOGIC | | PE\_ARITH  PE\_LOGIC |
| SHIFTER | | PE\_SHIFTER |
| NONLINEAR | | PE\_NONLINEAR |
| struct CSPM | | |
| PECSPM\_TOP | | PE\_CSPM\_TOP |
| PECSPM | | PE\_CSPM |

## （三）110\*512\*512\*512\*10 四层网络配置与验证

### 1、Layer\_0\_1\*110\*110\*512配置与验证

#### 1.1 Layer\_0\_PEA配置

（1）PEA与LSU连接关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **变量名称** | **类型** | **Load FSPM Support** | **Store FSPM Support** | **Load WSPM Support** |
| PE0 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W0 |
| PE1 | PE | Not Support | Not Support | LSU LOAD\_W1 |
| PE2 | PE | Not Support | Not Support | LSU LOAD\_W2 |
| PE3 | PE | Not Support | Not Support | LSU LOAD\_W3 |
| PE4 | PE | Not Support | Not Support | LSU LOAD\_W4 |
| PE5 | PE | Not Support | Not Support | LSU LOAD\_W5 |
| PE6 | PE | Not Support | Not Support | LSU LOAD\_W6 |
| PE7 | PE | Not Support | Not Support | LSU LOAD\_W7 |
| PE8 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W8 |
| PE9 | PE | Not Support | Not Support | LSU LOAD\_W9 |
| PE10 | PE | Not Support | Not Support | LSU LOAD\_W10 |
| PE11 | PE | Not Support | Not Support | LSU LOAD\_W11 |
| PE12 | PE | Not Support | Not Support | LSU LOAD\_W12 |
| PE13 | PE | Not Support | Not Support | LSU LOAD\_W13 |
| PE14 | PE | Not Support | Not Support | LSU LOAD\_W14 |
| PE15 | PE | Not Support | Not Support | LSU LOAD\_W15 |
| PE16 | PE | LSU LOAD\_F | Not Support | LSU LOAD\_W16 |
| PE17 | PE | Not Support | Not Support | LSU LOAD\_W17 |
| PE18 | PE | Not Support | Not Support | LSU LOAD\_W18 |
| PE19 | PE | Not Support | Not Support | LSU LOAD\_W19 |
| PE20 | PE | Not Support | Not Support | LSU LOAD\_W20 |
| PE21 | PE | Not Support | Not Support | LSU LOAD\_W21 |
| PE22 | PE | Not Support | Not Support | LSU LOAD\_W22 |
| PE23 | PE | Not Support | Not Support | LSU LOAD\_W23 |
| PE24 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W24 |
| PE25 | PE | Not Support | Not Support | LSU LOAD\_W25 |
| PE26 | PE | Not Support | Not Support | LSU LOAD\_W26 |
| PE27 | PE | Not Support | Not Support | LSU LOAD\_W27 |
| PE28 | PE | Not Support | Not Support | LSU LOAD\_W28 |
| PE29 | PE | Not Support | Not Support | LSU LOAD\_W29 |
| PE30 | PE | Not Support | Not Support | LSU LOAD\_W30 |
| PE31 | PE | Not Support | Not Support | LSU LOAD\_W31 |
| PE32 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W32 |
| PE33 | PE | Not Support | Not Support | LSU LOAD\_W33 |
| PE34 | PE | Not Support | Not Support | LSU LOAD\_W34 |
| PE35 | PE | Not Support | Not Support | LSU LOAD\_W35 |
| PE36 | PE | Not Support | Not Support | LSU LOAD\_W36 |
| PE37 | PE | Not Support | Not Support | LSU LOAD\_W37 |
| PE38 | PE | Not Support | Not Support | LSU LOAD\_W38 |
| PE39 | PE | Not Support | Not Support | LSU LOAD\_W39 |
| PE40 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W40 |
| PE41 | PE | Not Support | Not Support | LSU LOAD\_W41 |
| PE42 | PE | Not Support | Not Support | LSU LOAD\_W42 |
| PE43 | PE | Not Support | Not Support | LSU LOAD\_W43 |
| PE44 | PE | Not Support | Not Support | LSU LOAD\_W44 |
| PE45 | PE | Not Support | Not Support | LSU LOAD\_W45 |
| PE46 | PE | Not Support | Not Support | LSU LOAD\_W46 |
| PE47 | PE | Not Support | Not Support | LSU LOAD\_W47 |
| PE48 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W48 |
| PE49 | PE | Not Support | Not Support | LSU LOAD\_W49 |
| PE50 | PE | Not Support | Not Support | LSU LOAD\_W50 |
| PE51 | PE | Not Support | Not Support | LSU LOAD\_W51 |
| PE52 | PE | Not Support | Not Support | LSU LOAD\_W52 |
| PE53 | PE | Not Support | Not Support | LSU LOAD\_W53 |
| PE54 | PE | Not Support | Not Support | LSU LOAD\_W54 |
| PE55 | PE | Not Support | Not Support | LSU LOAD\_W55 |
| PE56 | PE | LSU LOAD\_F0 | Not Support | LSU LOAD\_W56 |
| PE57 | PE | Not Support | Not Support | LSU LOAD\_W57 |
| PE58 | PE | Not Support | Not Support | LSU LOAD\_W58 |
| PE59 | PE | Not Support | Not Support | LSU LOAD\_W59 |
| PE60 | PE | Not Support | Not Support | LSU LOAD\_W60 |
| PE61 | PE | Not Support | Not Support | LSU LOAD\_W61 |
| PE62 | PE | Not Support | Not Support | LSU LOAD\_W62 |
| PE63 | PE | Not Support | LSU STORE\_F1 | LSU LOAD\_W63 |

(2) CSPM配置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Context Memory | | | | |
| 实例化对象 | 结构体 | | 含义 | |
| PE\_TOP0 | CSPM\_TOP | | 初始化每个PE的Context Memory | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的Context Mememory配置  为MAC算子 | |
| PE\_CSPM1\_1 | PECSPM | | PEA非最左侧PE的Context Mememory配置为MAC算子 | |
| PE\_CSPM2 | PECSPM | | PEA所有PE公用的relu配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | LOAD\_FSPM配置 | |
| LOAD\_CSPM\_F1 | LSUCSPM | | LOAD\_FSPM配置 | |
| LSU\_TOP\_W0 | CSPM\_TOP | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W1 | LSUCSPM | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W2 | LSUCSPM | | LOAD\_WSPM配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | STORE\_FSPM配置 | |
| STORE\_CSPM1 | LSUCSPM | | STORE\_FSPM配置 | |
| Context Memory上述对象配置描述 | | | | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的CSPM : LSU支持Load FSPM，与WSPM的Load/Store | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| datawidth\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 – 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 – 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 5 | 0 – 15 | 4bit,输入通道，5对应lsu in |
| INZ | Unsigned Int | 6 | 0 – 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 5 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 – 5 | sMAC操作 |
| PE\_CSPM1\_1 | PECSPM | PEA非左侧PE的CSPM，LSU不具有访问FSPM的功能 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 – 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 – 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 5 | 0 – 15 | 4bit,输入通道，0对应左侧PE |
| INZ | Unsigned Int | 6 | 0 – 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 0 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 – 5 | sMAC操作 |
| PE\_CSPM2 | PECSPM | Relu,每一个PE都配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 6 | 0 - 6 | localreg使能，默认只有一拍end |
| INX | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 6 | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 6 | 0 – 6 | 3bit，6对应NONLINEAR |
| PE\_NONLINEAR.mode | Unsigned int | 0 | 0 - 5 | Relu操作 |
| LSU\_TOP\_F0 | CSPM\_TOP | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | Unsigned int | 0 | / | CSPMTOP指令重复次1次 |
| LOAD\_CSPM\_F1 | LSUCSPM | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| bank\_id | Unsigned int | 0 | 0-15 | FSPM区分ID |
| delay | Unsigned Int | 0 | / | 延迟 |
| length | Unsigned int | 109 | / | LOAD数据长度，从0计数，109表示Load 110个值 |
| CSPM\_renum | Unsigned Int | 7 | / | 重复操作次数 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned Int | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_W0 | CSPM\_TOP | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 1 | / | 4bit, CSPM指令数量为2 |
| CSPM\_TOP\_renum | Unsigned int | 3 | / | CSPMTOP指令重复次4次 |
| LOAD\_CSPM\_W1 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | Unsigned int | 动态配置 | 0- 63 | WSPM区分ID |
| delay | Unsigned Int | 动态配置 | / | / |
| length | Unsigned int | 109 | / | LOAD数据长度，从0计数，109表示Load 110个值 |
| CSPM\_renum | Unsigned Int | 0 | / | 重复操作次数，只执行1次 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LOAD\_CSPM\_W2 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 动态配置 | 0-63 | WSPM区分ID |
| delay | INT | 动态配置 | / | 根据PE在PEA位置单独配置 |
| length | INT | 109 | / | LOAD数据长度，从0计数，109表示Load 110个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_F0 | CSPM\_TOP | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | INT | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | INT | 0 | / | CSPMTOP指令重复次0次 |
| STORE\_CSPM1 | LSUCSPM | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 1 | 0-15 | FSPM区分ID |
| Data\_from | INT | 1 | / | Store模式下数据来源，0表示PE，1表示寄存器链 |
| length | INT | 511 | / | LOAD数据长度，从0计数，109表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 1 | 0, 1 | 0 load 1 store |

#### 1.2 Layer\_0\_模拟器仿真及功耗分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Int 8bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 128 | 0.2433 | 880 | 963 | 28.45826791 |
| PE单元（执行比较） | 128 | 0.008 | 8 | 963 | 0.00850675 |
| load(带sram读) | 65 | 1.3851 | 880 | 963 | 82.2717757 |
| store(带sram写) | 1 | 1.4346 | 512 | 963 | 0.762736449 |
| rc | 64 | 0.033 | 512 | 963 | 1.122890966 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 118.5141778 |
| 不包含访存功耗/mw | | 29.58966563 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Float 16bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 64 | 0.8 | 880 | 963 | 46.78712357 |
| PE单元（执行比较） | 64 | 0.016 | 8 | 963 | 0.00850675 |
| load(带sram读) | 65 | 1.3851 | 880 | 963 | 82.2717757 |
| store(带sram写) | 1 | 1.4346 | 512 | 963 | 0.762736449 |
| rc | 64 | 0.033 | 512 | 963 | 1.122890966 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 136.8430334 |
| 不包含访存功耗/mw | | 47.91852129 |

注：以上数据为单个PEA阵列测算所得，400MHz下计算该层网络总耗时

### 2、Layer\_1\_1\*512\*512\*512配置与验证

#### 2.1 Layer\_1\_PEA配置

（1）PEA与LSU连接关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **变量名称** | **类型** | **Load FSPM Support** | **Store FSPM Support** | **Load WSPM Support** |
| PE0 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W0 |
| PE1 | PE | Not Support | Not Support | LSU LOAD\_W1 |
| PE2 | PE | Not Support | Not Support | LSU LOAD\_W2 |
| PE3 | PE | Not Support | Not Support | LSU LOAD\_W3 |
| PE4 | PE | Not Support | Not Support | LSU LOAD\_W4 |
| PE5 | PE | Not Support | Not Support | LSU LOAD\_W5 |
| PE6 | PE | Not Support | Not Support | LSU LOAD\_W6 |
| PE7 | PE | Not Support | Not Support | LSU LOAD\_W7 |
| PE8 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W8 |
| PE9 | PE | Not Support | Not Support | LSU LOAD\_W9 |
| PE10 | PE | Not Support | Not Support | LSU LOAD\_W10 |
| PE11 | PE | Not Support | Not Support | LSU LOAD\_W11 |
| PE12 | PE | Not Support | Not Support | LSU LOAD\_W12 |
| PE13 | PE | Not Support | Not Support | LSU LOAD\_W13 |
| PE14 | PE | Not Support | Not Support | LSU LOAD\_W14 |
| PE15 | PE | Not Support | Not Support | LSU LOAD\_W15 |
| PE16 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W16 |
| PE17 | PE | Not Support | Not Support | LSU LOAD\_W17 |
| PE18 | PE | Not Support | Not Support | LSU LOAD\_W18 |
| PE19 | PE | Not Support | Not Support | LSU LOAD\_W19 |
| PE20 | PE | Not Support | Not Support | LSU LOAD\_W20 |
| PE21 | PE | Not Support | Not Support | LSU LOAD\_W21 |
| PE22 | PE | Not Support | Not Support | LSU LOAD\_W22 |
| PE23 | PE | Not Support | Not Support | LSU LOAD\_W23 |
| PE24 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W24 |
| PE25 | PE | Not Support | Not Support | LSU LOAD\_W25 |
| PE26 | PE | Not Support | Not Support | LSU LOAD\_W26 |
| PE27 | PE | Not Support | Not Support | LSU LOAD\_W27 |
| PE28 | PE | Not Support | Not Support | LSU LOAD\_W28 |
| PE29 | PE | Not Support | Not Support | LSU LOAD\_W29 |
| PE30 | PE | Not Support | Not Support | LSU LOAD\_W30 |
| PE31 | PE | Not Support | Not Support | LSU LOAD\_W31 |
| PE32 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W32 |
| PE33 | PE | Not Support | Not Support | LSU LOAD\_W33 |
| PE34 | PE | Not Support | Not Support | LSU LOAD\_W34 |
| PE35 | PE | Not Support | Not Support | LSU LOAD\_W35 |
| PE36 | PE | Not Support | Not Support | LSU LOAD\_W36 |
| PE37 | PE | Not Support | Not Support | LSU LOAD\_W37 |
| PE38 | PE | Not Support | Not Support | LSU LOAD\_W38 |
| PE39 | PE | Not Support | Not Support | LSU LOAD\_W39 |
| PE40 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W40 |
| PE41 | PE | Not Support | Not Support | LSU LOAD\_W41 |
| PE42 | PE | Not Support | Not Support | LSU LOAD\_W42 |
| PE43 | PE | Not Support | Not Support | LSU LOAD\_W43 |
| PE44 | PE | Not Support | Not Support | LSU LOAD\_W44 |
| PE45 | PE | Not Support | Not Support | LSU LOAD\_W45 |
| PE46 | PE | Not Support | Not Support | LSU LOAD\_W46 |
| PE47 | PE | Not Support | Not Support | LSU LOAD\_W47 |
| PE48 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W48 |
| PE49 | PE | Not Support | Not Support | LSU LOAD\_W49 |
| PE50 | PE | Not Support | Not Support | LSU LOAD\_W50 |
| PE51 | PE | Not Support | Not Support | LSU LOAD\_W51 |
| PE52 | PE | Not Support | Not Support | LSU LOAD\_W52 |
| PE53 | PE | Not Support | Not Support | LSU LOAD\_W53 |
| PE54 | PE | Not Support | Not Support | LSU LOAD\_W54 |
| PE55 | PE | Not Support | Not Support | LSU LOAD\_W55 |
| PE56 | PE | LSU LOAD\_F1 | Not Support | LSU LOAD\_W56 |
| PE57 | PE | Not Support | Not Support | LSU LOAD\_W57 |
| PE58 | PE | Not Support | Not Support | LSU LOAD\_W58 |
| PE59 | PE | Not Support | Not Support | LSU LOAD\_W59 |
| PE60 | PE | Not Support | Not Support | LSU LOAD\_W60 |
| PE61 | PE | Not Support | Not Support | LSU LOAD\_W61 |
| PE62 | PE | Not Support | Not Support | LSU LOAD\_W62 |
| PE63 | PE | Not Support | LSU STORE\_F2 | LSU LOAD\_W63 |

(2) CSPM配置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Context Memory | | | | |
| 实例化对象 | 结构体 | | 含义 | |
| PE\_TOP0 | CSPM\_TOP | | 初始化每个PE的Context Memory | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的Context Mememory配置  为MAC算子 | |
| PE\_CSPM1\_1 | PECSPM | | PEA非最左侧PE的Context Mememory配置为MAC算子 | |
| PE\_CSPM2 | PECSPM | | PEA所有PE公用的relu配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | LOAD\_FSPM配置 | |
| LOAD\_CSPM\_F1 | LSUCSPM | | LOAD\_FSPM配置 | |
| LSU\_TOP\_W0 | CSPM\_TOP | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W1 | LSUCSPM | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W2 | LSUCSPM | | LOAD\_WSPM配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | STORE\_FSPM配置 | |
| STORE\_CSPM1 | LSUCSPM | | STORE\_FSPM配置 | |
| Context Memory上述对象配置描述 | | | | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的CSPM : LSU支持Load FSPM，与WSPM的Load/Store | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| datawidth\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 5 | 0 - 15 | 4bit,输入通道，5对应lsu in |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 5 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM1\_1 | PECSPM | PEA非左侧PE的CSPM，LSU不具有访问FSPM的功能 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 0 | 0 - 15 | 左侧PE |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 0 | 0-7, F | 左侧PE |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM2 | PECSPM | Relu,每一个PE都配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 6 | 0 - 6 | localreg使能，默认只有一拍end |
| INX | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 6 | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 6 | 0 – 6 | 3bit，6对应NONLINEAR |
| PE\_NONLINEAR.mode | Unsigned int | 0 | 0 - 5 | Relu操作 |
| LSU\_TOP\_F0 | CSPM\_TOP | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | Unsigned int | 0 | / | CSPMTOP指令重复次1次 |
| LOAD\_CSPM\_F1 | LSUCSPM | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| bank\_id | Unsigned int | 1 | 0-15 | FSPM区分ID |
| delay | Unsigned Int | 0 | / | 延迟 |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 7 | / | 重复操作次数 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned Int | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_W0 | CSPM\_TOP | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 1 | / | 4bit, CSPM指令数量为2 |
| CSPM\_TOP\_renum | Unsigned int | 3 | / | CSPMTOP指令重复次4次 |
| LOAD\_CSPM\_W1 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | Unsigned int | 动态配置 | 0- 63 | WSPM区分ID |
| delay | Unsigned Int | 动态配置 | / | / |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 0 | / | 重复操作次数，只执行1次 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LOAD\_CSPM\_W2 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 动态配置 | 0-63 | WSPM区分ID |
| delay | INT | 动态配置 | / | 根据PE在PEA位置单独配置 |
| length | INT | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_F0 | CSPM\_TOP | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | INT | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | INT | 0 | / | CSPMTOP指令重复次0次 |
| STORE\_CSPM1 | LSUCSPM | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 2 | 0-15 | FSPM区分ID |
| Data\_from | INT | 1 | / | Store模式下数据来源，0表示PE，1表示寄存器链 |
| length | INT | 511 | / | LOAD数据长度，从0计数，109表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 1 | 0, 1 | 0 load 1 store |

#### 2.2 Layer\_0\_模拟器仿真及功耗分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Int 8bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 128 | 0.2433 | 4096 | 4179 | 30.52387423 |
| PE单元（执行比较） | 128 | 0.008 | 8 | 4179 | 0.001960278 |
| load(带sram读) | 65 | 1.3851 | 4096 | 4179 | 88.2433654 |
| store(带sram写) | 1 | 1.4346 | 512 | 4179 | 0.175763388 |
| rc | 64 | 0.033 | 512 | 4179 | 0.25875664 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 125.0937199 |
| 不包含访存功耗/mw | | 30.78459115 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Float 16bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 64 | 0.8 | 4096 | 4179 | 50.18310601 |
| PE单元（执行比较） | 64 | 0.016 | 8 | 4179 | 0.001960278 |
| load(带sram读) | 65 | 1.3851 | 4096 | 4179 | 88.2433654 |
| store(带sram写) | 1 | 1.4346 | 512 | 4179 | 0.175763388 |
| rc | 64 | 0.033 | 512 | 4179 | 0.25875664 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 144.7529517 |
| 不包含访存功耗/mw | | 50.44382292 |

注：以上数据为单个PEA阵列测算所得

### 3、Layer\_2\_1\*512\*512\*512配置与验证

#### 3.1 Layer\_2\_PEA配置

（1）PEA与LSU连接关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **变量名称** | **类型** | **Load FSPM Support** | **Store FSPM Support** | **Load WSPM Support** |
| PE0 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W0 |
| PE1 | PE | Not Support | Not Support | LSU LOAD\_W1 |
| PE2 | PE | Not Support | Not Support | LSU LOAD\_W2 |
| PE3 | PE | Not Support | Not Support | LSU LOAD\_W3 |
| PE4 | PE | Not Support | Not Support | LSU LOAD\_W4 |
| PE5 | PE | Not Support | Not Support | LSU LOAD\_W5 |
| PE6 | PE | Not Support | Not Support | LSU LOAD\_W6 |
| PE7 | PE | Not Support | Not Support | LSU LOAD\_W7 |
| PE8 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W8 |
| PE9 | PE | Not Support | Not Support | LSU LOAD\_W9 |
| PE10 | PE | Not Support | Not Support | LSU LOAD\_W10 |
| PE11 | PE | Not Support | Not Support | LSU LOAD\_W11 |
| PE12 | PE | Not Support | Not Support | LSU LOAD\_W12 |
| PE13 | PE | Not Support | Not Support | LSU LOAD\_W13 |
| PE14 | PE | Not Support | Not Support | LSU LOAD\_W14 |
| PE15 | PE | Not Support | Not Support | LSU LOAD\_W15 |
| PE16 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W16 |
| PE17 | PE | Not Support | Not Support | LSU LOAD\_W17 |
| PE18 | PE | Not Support | Not Support | LSU LOAD\_W18 |
| PE19 | PE | Not Support | Not Support | LSU LOAD\_W19 |
| PE20 | PE | Not Support | Not Support | LSU LOAD\_W20 |
| PE21 | PE | Not Support | Not Support | LSU LOAD\_W21 |
| PE22 | PE | Not Support | Not Support | LSU LOAD\_W22 |
| PE23 | PE | Not Support | Not Support | LSU LOAD\_W23 |
| PE24 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W24 |
| PE25 | PE | Not Support | Not Support | LSU LOAD\_W25 |
| PE26 | PE | Not Support | Not Support | LSU LOAD\_W26 |
| PE27 | PE | Not Support | Not Support | LSU LOAD\_W27 |
| PE28 | PE | Not Support | Not Support | LSU LOAD\_W28 |
| PE29 | PE | Not Support | Not Support | LSU LOAD\_W29 |
| PE30 | PE | Not Support | Not Support | LSU LOAD\_W30 |
| PE31 | PE | Not Support | Not Support | LSU LOAD\_W31 |
| PE32 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W32 |
| PE33 | PE | Not Support | Not Support | LSU LOAD\_W33 |
| PE34 | PE | Not Support | Not Support | LSU LOAD\_W34 |
| PE35 | PE | Not Support | Not Support | LSU LOAD\_W35 |
| PE36 | PE | Not Support | Not Support | LSU LOAD\_W36 |
| PE37 | PE | Not Support | Not Support | LSU LOAD\_W37 |
| PE38 | PE | Not Support | Not Support | LSU LOAD\_W38 |
| PE39 | PE | Not Support | Not Support | LSU LOAD\_W39 |
| PE40 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W40 |
| PE41 | PE | Not Support | Not Support | LSU LOAD\_W41 |
| PE42 | PE | Not Support | Not Support | LSU LOAD\_W42 |
| PE43 | PE | Not Support | Not Support | LSU LOAD\_W43 |
| PE44 | PE | Not Support | Not Support | LSU LOAD\_W44 |
| PE45 | PE | Not Support | Not Support | LSU LOAD\_W45 |
| PE46 | PE | Not Support | Not Support | LSU LOAD\_W46 |
| PE47 | PE | Not Support | Not Support | LSU LOAD\_W47 |
| PE48 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W48 |
| PE49 | PE | Not Support | Not Support | LSU LOAD\_W49 |
| PE50 | PE | Not Support | Not Support | LSU LOAD\_W50 |
| PE51 | PE | Not Support | Not Support | LSU LOAD\_W51 |
| PE52 | PE | Not Support | Not Support | LSU LOAD\_W52 |
| PE53 | PE | Not Support | Not Support | LSU LOAD\_W53 |
| PE54 | PE | Not Support | Not Support | LSU LOAD\_W54 |
| PE55 | PE | Not Support | Not Support | LSU LOAD\_W55 |
| PE56 | PE | LSU LOAD\_F2 | Not Support | LSU LOAD\_W56 |
| PE57 | PE | Not Support | Not Support | LSU LOAD\_W57 |
| PE58 | PE | Not Support | Not Support | LSU LOAD\_W58 |
| PE59 | PE | Not Support | Not Support | LSU LOAD\_W59 |
| PE60 | PE | Not Support | Not Support | LSU LOAD\_W60 |
| PE61 | PE | Not Support | Not Support | LSU LOAD\_W61 |
| PE62 | PE | Not Support | Not Support | LSU LOAD\_W62 |
| PE63 | PE | Not Support | LSU STORE\_F3 | LSU LOAD\_W63 |

(2) CSPM配置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Context Memory | | | | |
| 实例化对象 | 结构体 | | 含义 | |
| PE\_TOP0 | CSPM\_TOP | | 初始化每个PE的Context Memory | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的Context Mememory配置  为MAC算子 | |
| PE\_CSPM1\_1 | PECSPM | | PEA非最左侧PE的Context Mememory配置为MAC算子 | |
| PE\_CSPM2 | PECSPM | | PEA所有PE公用的relu配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | LOAD\_FSPM配置 | |
| LOAD\_CSPM\_F1 | LSUCSPM | | LOAD\_FSPM配置 | |
| LSU\_TOP\_W0 | CSPM\_TOP | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W1 | LSUCSPM | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W2 | LSUCSPM | | LOAD\_WSPM配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | STORE\_FSPM配置 | |
| STORE\_CSPM1 | LSUCSPM | | STORE\_FSPM配置 | |
| Context Memory上述对象配置描述 | | | | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的CSPM : LSU支持Load FSPM，与WSPM的Load/Store | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| datawidth\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 5 | 0 - 15 | 4bit,输入通道，5对应lsu in |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 5 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM1\_1 | PECSPM | PEA非左侧PE的CSPM，LSU不具有访问FSPM的功能 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 0 | 0 - 15 | 4bit,输入通道，0对应左侧PE |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 0 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM2 | PECSPM | Relu,每一个PE都配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 6 | 0 - 6 | localreg使能，默认只有一拍end |
| INX | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 6 | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 6 | 0 – 6 | 3bit，6对应NONLINEAR |
| PE\_NONLINEAR.mode | Unsigned int | 0 | 0 - 5 | Relu操作 |
| LSU\_TOP\_F0 | CSPM\_TOP | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | Unsigned int | 0 | / | CSPMTOP指令重复次1次 |
| LOAD\_CSPM\_F1 | LSUCSPM | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| bank\_id | Unsigned int | 0 | 0-15 | FSPM区分ID |
| delay | Unsigned Int | 2 | / | 延迟 |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 7 | / | 重复操作次数 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned Int | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_W0 | CSPM\_TOP | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 1 | / | 4bit, CSPM指令数量为2 |
| CSPM\_TOP\_renum | Unsigned int | 3 | / | CSPMTOP指令重复次4次 |
| LOAD\_CSPM\_W1 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | Unsigned int | 动态配置 | 0- 63 | WSPM区分ID |
| delay | Unsigned Int | 动态配置 | / | / |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 0 | / | 重复操作次数，只执行1次 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LOAD\_CSPM\_W2 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 动态配置 | 0-63 | WSPM区分ID |
| delay | INT | 动态配置 | / | 根据PE在PEA位置单独配置 |
| length | INT | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_F0 | CSPM\_TOP | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | INT | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | INT | 0 | / | CSPMTOP指令重复次0次 |
| STORE\_CSPM1 | LSUCSPM | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 3 | 0-15 | FSPM区分ID |
| Data\_from | INT | 1 | / | Store模式下数据来源，0表示PE，1表示寄存器链 |
| length | INT | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 1 | 0, 1 | 0 load 1 store |

#### 3.2 Layer\_2\_模拟器仿真及功耗分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Int 8bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 128 | 0.2433 | 4096 | 4179 | 30.52387423 |
| PE单元（执行比较） | 128 | 0.008 | 8 | 4179 | 0.001960278 |
| load(带sram读) | 65 | 1.3851 | 4096 | 4179 | 88.2433654 |
| store(带sram写) | 1 | 1.4346 | 512 | 4179 | 0.175763388 |
| rc | 64 | 0.033 | 512 | 4179 | 0.25875664 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 125.0937199 |
| 不包含访存功耗/mw | | 30.78459115 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Float 16bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 64 | 0.8 | 4096 | 4179 | 50.18310601 |
| PE单元（执行比较） | 64 | 0.016 | 8 | 4179 | 0.001960278 |
| load(带sram读) | 65 | 1.3851 | 4096 | 4179 | 88.2433654 |
| store(带sram写) | 1 | 1.4346 | 512 | 4179 | 0.175763388 |
| rc | 64 | 0.033 | 512 | 4179 | 0.25875664 |
| cpu | 1 | 5.89 | 4179 | 4179 | 5.89 |
|  | | | 包含访存功耗/mw | | 144.7529517 |
| 不包含访存功耗/mw | | 50.44382292 |

注：以上数据为单个PEA阵列测算所得

### 4、Layer3\_1\*512\*512\*10配置与验证

#### 4.1 Layer\_3\_PEA配置

（1）PEA与LSU连接关系

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **变量名称** | **类型** | **Load FSPM Support** | **Store FSPM Support** | **Load WSPM Support** |
| PE0 | PE | Not Support | Not Support | Not Support |
| PE1 | PE | Not Support | Not Support | Not Support |
| PE2 | PE | Not Support | Not Support | Not Support |
| PE3 | PE | Not Support | Not Support | Not Support |
| PE4 | PE | Not Support | Not Support | Not Support |
| PE5 | PE | Not Support | Not Support | Not Support |
| PE6 | PE | Not Support | Not Support | Not Support |
| PE7 | PE | Not Support | Not Support | Not Support |
| PE8 | PE | Not Support | Not Support | Not Support |
| PE9 | PE | Not Support | Not Support | Not Support |
| PE10 | PE | Not Support | Not Support | Not Support |
| PE11 | PE | Not Support | Not Support | Not Support |
| PE12 | PE | Not Support | Not Support | Not Support |
| PE13 | PE | Not Support | Not Support | Not Support |
| PE14 | PE | Not Support | Not Support | Not Support |
| PE15 | PE | Not Support | Not Support | Not Support |
| PE16 | PE | Not Support | Not Support | Not Support |
| PE17 | PE | Not Support | Not Support | Not Support |
| PE18 | PE | Not Support | Not Support | Not Support |
| PE19 | PE | Not Support | Not Support | Not Support |
| PE20 | PE | Not Support | Not Support | Not Support |
| PE21 | PE | Not Support | Not Support | Not Support |
| PE22 | PE | Not Support | Not Support | Not Support |
| PE23 | PE | Not Support | Not Support | Not Support |
| PE24 | PE | Not Support | Not Support | Not Support |
| PE25 | PE | Not Support | Not Support | Not Support |
| PE26 | PE | Not Support | Not Support | Not Support |
| PE27 | PE | Not Support | Not Support | Not Support |
| PE28 | PE | Not Support | Not Support | Not Support |
| PE29 | PE | Not Support | Not Support | Not Support |
| PE30 | PE | Not Support | Not Support | Not Support |
| PE31 | PE | Not Support | Not Support | Not Support |
| PE32 | PE | Not Support | Not Support | Not Support |
| PE33 | PE | Not Support | Not Support | Not Support |
| PE34 | PE | Not Support | Not Support | Not Support |
| PE35 | PE | Not Support | Not Support | Not Support |
| PE36 | PE | Not Support | Not Support | Not Support |
| PE37 | PE | Not Support | Not Support | Not Support |
| PE38 | PE | Not Support | Not Support | Not Support |
| PE39 | PE | Not Support | Not Support | Not Support |
| PE40 | PE | Not Support | Not Support | Not Support |
| PE41 | PE | Not Support | Not Support | Not Support |
| PE42 | PE | Not Support | Not Support | Not Support |
| PE43 | PE | Not Support | Not Support | Not Support |
| PE44 | PE | Not Support | Not Support | Not Support |
| PE45 | PE | Not Support | Not Support | Not Support |
| PE46 | PE | Not Support | Not Support | Not Support |
| PE47 | PE | Not Support | Not Support | Not Support |
| PE48 | PE | LSU LOAD\_F3 | Not Support | Not Support |
| PE49 | PE | Not Support | Not Support | Not Support |
| PE50 | PE | Not Support | Not Support | Not Support |
| PE51 | PE | Not Support | Not Support | Not Support |
| PE52 | PE | Not Support | Not Support | Not Support |
| PE53 | PE | Not Support | Not Support | Not Support |
| PE54 | PE | Not Support | Not Support | LSU LOAD\_W54 |
| PE55 | PE | Not Support | Not Support | LSU LOAD\_W55 |
| PE56 | PE | LSU LOAD\_F3 | Not Support | LSU LOAD\_W56 |
| PE57 | PE | Not Support | Not Support | LSU LOAD\_W57 |
| PE58 | PE | Not Support | Not Support | LSU LOAD\_W58 |
| PE59 | PE | Not Support | Not Support | LSU LOAD\_W59 |
| PE60 | PE | Not Support | Not Support | LSU LOAD\_W60 |
| PE61 | PE | Not Support | Not Support | LSU LOAD\_W61 |
| PE62 | PE | Not Support | Not Support | LSU LOAD\_W62 |
| PE63 | PE | Not Support | LSU STORE\_F4 | LSU LOAD\_W63 |

(2) CSPM配置

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Context Memory | | | | |
| 实例化对象 | 结构体 | | 含义 | |
| PE\_TOP0 | CSPM\_TOP | | 初始化每个PE的Context Memory | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的Context Mememory配置  为MAC算子 | |
| PE\_CSPM1\_1 | PECSPM | | PEA非最左侧PE的Context Mememory配置为MAC算子 | |
| PE\_CSPM1\_2 | PECSPM | | PEA最左侧PE的Context Mememory配置  Route算子 | |
| PE\_CSPM1\_3 | PECSPM | | PEA非最左侧PE的Context Mememory配置  Route算子 | |
| PE\_CSPM2 | PECSPM | | PEA所有PE公用的relu配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | LOAD\_FSPM配置 | |
| LOAD\_CSPM\_F1 | LSUCSPM | | LOAD\_FSPM配置 | |
| LSU\_TOP\_W0 | CSPM\_TOP | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W1 | LSUCSPM | | LOAD\_WSPM配置 | |
| LOAD\_CSPM\_W2 | LSUCSPM | | LOAD\_WSPM配置 | |
| LSU\_TOP\_F0 | CSPM\_TOP | | STORE\_FSPM配置 | |
| STORE\_CSPM1 | LSUCSPM | | STORE\_FSPM配置 | |
| Context Memory上述对象配置描述 | | | | |
| PE\_CSPM1\_0 | PECSPM | | PEA最左侧PE的CSPM : LSU支持Load FSPM，与WSPM的Load/Store | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| datawidth\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 5 | 0 - 15 | 4bit,输入通道，5对应lsu in |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 5 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM1\_1 | PECSPM | PEA非左侧PE的CSPM，LSU不具有访问FSPM的功能 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 4 | 0 - 6 | 触发beat通道，load\_w使能 |
| INX | Unsigned Int | 4 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 0 | 0 - 15 | 4bit,输入通道，0对应左侧PE |
| INZ | Unsigned Int | 6 | 0 - 15 | 4bit,输入通道，local\_reg[0] |
| PE\_OUT[0] | Unsigned Int | 0 | 0-7, F | 3bit |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 6 | 0-7, F | 3bit, 输出选择，local\_reg[0] |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_beat\_mode | Bool | 1 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 3 | 0 – 6 | 3bit，3对应ARITH |
| PE\_ARITH.mode | Unsigned int | 5 | 0 - 5 | sMAC操作 |
| PE\_CSPM1\_2 | PECSPM | Route,最左侧PE | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 5 | 0 - 6 | localreg使能，load\_w |
| INX | Unsigned Int | 5 | 0 - 15 | 4bit,输入通道，load\_w |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 5 | 0-7, F | 3bit, 输出load\_f |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 0 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 1 | 0 – 6 | 3bit，Route |
| PE\_Route.route\_delay | Unsigned int | 0 |  | Route延迟 |
| PE\_CSPM1\_3 | PECSPM | Route,非最左侧PE | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 0 | 0 - 6 | localreg使能，左侧PE |
| INX | Unsigned Int | 0 | 0 - 15 | 4bit,输入通道，左侧PE |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 0 | 0-7, F | 3bit, 左侧PE |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 0xf | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 0 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 1 | 0 – 6 | 3bit，Route |
| PE\_Route.route\_delay | Unsigned int | 0 |  | Route延迟 |
| PE\_CSPM2 | PECSPM | Relu | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| Data\_width\_mode | Bool | 0 | 0/1 | 数据位宽模式 0: int8 1:fp16 |
| CSPM\_renum | Unsigned Int | 0 | / | 循环次数，0表示1次 |
| enable\_beat | Unsigned Int | 6 | 0 - 6 | localreg使能，默认只有1拍end |
| INX | Unsigned Int | 6 | 0 - 15 | 输入来源localreg[0] |
| INY | Unsigned Int | 0xf | 0 - 15 | 4bit, 不使用 |
| INZ | Unsigned Int | 0xf | 0 - 15 | 4bit,不使用 |
| PE\_OUT[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_OUT[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_OUT[3] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[0] | Unsigned Int | 0xf | 0-7, F | 3bit, 不使用 |
| PE\_lr[1] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lr[2] | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_lsu | Unsigned Int | 0xf | 0-7, F | 3bit，不使用 |
| PE\_gr | Unsigned Int | 6 | 0-7, F | 3bit，输出选择，6对应localreg[0] |
| PE\_beat\_mode | Bool | 0 | 0,1 | 0 EN 使能, 1 END 使能 |
| PE\_opcode | Unsigned Int | 6 | 0 – 6 | 3bit，6对应NONLINEAR |
| PE\_NONLINEAR.mode | Unsigned int | 0 | 0 - 5 | Relu操作 |
| LSU\_TOP\_F0 | CSPM\_TOP | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | Unsigned int | 0 | / | CSPMTOP指令重复次1次 |
| LOAD\_CSPM\_F1 | LSUCSPM | LOAD\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| bank\_id | Unsigned int | 3 | 0-15 | FSPM区分ID |
| delay | Unsigned Int | 0 | / | 延迟 |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 7 | / | 重复操作次数 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned Int | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_W0 | CSPM\_TOP | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | Unsigned Int | 1 | / | 4bit, CSPM指令数量为2 |
| CSPM\_TOP\_renum | Unsigned int | 3 | / | CSPMTOP指令重复次4次 |
| LOAD\_CSPM\_W1 | LSUCSPM | LOAD\_W配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | Unsigned Int | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | Unsigned int | 动态配置 | 0- 63 | WSPM区分ID |
| delay | Unsigned Int | 动态配置 | / | / |
| length | Unsigned int | 511 | / | LOAD数据长度，从0计数，511表示Load 512个值 |
| CSPM\_renum | Unsigned Int | 0 | / | 重复操作次数，只执行1次 |
| mode | Unsigned int | 0 | 0, 1 | 0 load 1 store |
| renum\_delay | Unsigned INT | 1 | / | 重复操作间隔延时，延时1拍 |
| LSU\_TOP\_F0 | CSPM\_TOP | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| CSPM\_total | INT | 0 | / | 4bit, CSPM指令数量为1 |
| CSPM\_TOP\_renum | INT | 0 | / | CSPMTOP指令重复次0次 |
| STORE\_CSPM1 | LSUCSPM | STORE\_F配置 | | |
| 参量 | 类型 | 取值 | 取值范围 | 含义 |
| addr\_start | INT | 0 | 0-511 | Bank内起始地址 |
| Bank\_id | INT | 4 | 0-15 | FSPM区分ID |
| Data\_from | INT | 1 | / | Store模式下数据来源，0表示PE，1表示寄存器链 |
| length | INT | 9 | / | LOAD数据长度，从0计数，109表示Load 512个值 |
| CSPM\_renum | INT | 0 | / | 重复操作次数，只执行1次 |
| mode | INT | 1 | 0, 1 | 0 load 1 store |

#### 4.2 Layer\_3\_模拟器仿真及功耗分析

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Int 8bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 10 | 0.2433 | 512 | 534 | 2.332764 |
| PE单元（执行比较） | 10 | 0.008 | 8 | 534 | 0.001199 |
| load(带sram读) | 11 | 1.3851 | 512 | 534 | 14.6084 |
| store(带sram写) | 1 | 1.4346 | 10 | 534 | 0.026865 |
| rc | 10 | 0.033 | 10 | 534 | 0.00618 |
| cpu | 1 | 5.89 | 534 | 534 | 5.89 |
|  | | | 包含访存功耗/mw | | 22.8654 |
| 不包含访存功耗/mw | | 2.340142322 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Float 16bit模式 | | | | | |
| 硬件底层单元 | 个数 | 功耗/mw | cycle个数 | cycle总计 | 总计/mw |
| PE单元（执行乘累加） | 10 | 0.8 | 512 | 534 | 7.670411985 |
| PE单元（执行比较） | 10 | 0.016 | 8 | 534 | 0.002397004 |
| Load  (带sram读) | 11 | 1.3851 | 512 | 534 | 14.60839551 |
| Store  (带sram写) | 1 | 1.4346 | 10 | 534 | 0.026865169 |
| rc | 10 | 0.033 | 10 | 534 | 0.006179775 |
| cpu | 1 | 5.89 | 534 | 534 | 5.89 |
|  | | | 包含访存功耗/mw | | 28.20424944 |
| 不包含访存功耗/mw | | 7.678988764 |

注：以上数据为单个PEA阵列测算所得

### 5、CGRA性能要求评估

下表为Layer1与Layer2两个网络下PEA阵列上所有PE都映射为MAC算子，我们考虑这种情况下所反映出的CGRA性能情况，INT8处理精度下，不考虑访存能效比能够达到3.26TOPs/w, 在FP16处理精度下能效比有所下降，两种模式下功耗均小于1w。根据本章前4节的仿真结果计算，110\*512\*512\*512\*10规模的神经网络处理时间在400MHz下为24.6375us。

|  |  |
| --- | --- |
| 神经网络层 | 模拟器仿真周期数 |
| Layer\_0\_1\*110\*110\*512 | 963 |
| Layer\_1\_1\*512\*512\*512 | 4179 |
| Layer\_2\_1\*512\*512\*512 | 4179 |
| Layer\_3\_1\*512\*512\*10 | 534 |
| 处理时间合计 | 9855 |

|  |  |  |  |
| --- | --- | --- | --- |
| 8bit算力评估 | 128个mac | 250.9 | OPS/cycle |
|  | 频率 | 400 | Mhz |
|  | 性能 | 100.36 | GOPS/s |
|  | 能效(不计算访存) | 3.260072532 | TOPS/w |
|  | 包含访存功耗/mw | 125.0937199 |  |
|  | 不包含访存功耗/mw | 30.78459115 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| 16bit算力评估 | 64个mac | 125.5 | FLOPS/cycle |
|  | 频率 | 400 | Mhz |
|  | 性能 | 50.2 | GFLOPS/s |
|  | 能效(不计算访存) | 0.995166446 | TFLOPS/w |
|  | 包含访存功耗/mw | 144.7529517 |  |
|  | 不包含访存功耗/mw | 50.44382292 |  |

## （四）未来工作展望

CGRA可以根据应用领域的不同去调整，比如面向深度学习领域的神经网络应用，特别是计算规律的卷积神经网络（CNN）。这时重点就是限制传统CGRA的通用性和可重构性，以适应CNN的计算模式，并且将牺牲一部分通用性换来的逻辑，用于支持预定的深度学习工作负载的专门操作（如压缩、多播等）。此外，这些架构往往更青睐较小的（或混合的）数字表示，因为深度学习往往可以修正为低精度的计算。

粗粒度的PE可以包括可编程的延迟控制线，以便更容易映射时间上接近的数据。PE内部的数据通过令牌同步；对多种常见的深度学习访问模式（包括跨度、类型等）的支持，通过定制的buffer单元来实现，这些单元以VLIW（超长指令字）的方式可编程，并产生对外部存储器的访问。

针对CNN，可以利用激活和内核权重中的稀疏性。针对强化学习的CGRA加速器。处理元件本身是相当静态的，支持加法、乘法或两者的融合。此外，一些不同的激活函数（ReLu、sigmoid和tanh）可以通过配置寄存器来选择，数据可以暂时存储在本地的刮板上。目前大多数CGRA将地址发生器放在PE之外的离散单元中，也可以将地址发生器放在PE里面。通过全局通信互联线，支持用户控制系统的强化训练经验。

面向低功耗应用领域，如嵌入式和传感器，CGRAs也被证明具有竞争力，特别是与现有的（低功耗）处理器和DSP引擎相比。该领域的CGRAs遵循与早期CGRA设计相同的概念，但侧重于技术和架构的改进，以降低结构的静态和/或动态功率。

这些CGRAs倾向于关注尽可能地降低频率和电压。由于系统的动态功耗是频率和电压的函数（Pdynamic=C∗V2∗fclk），降低频率可以对功耗产生巨大的影响。该领域的几个CGRAs在接近MHz的水平上运行，有些甚至完全取消了时钟。

我们知道，(i) 时钟（时钟树、触发器、状态等）是现代芯片上大部分消耗功率的罪魁祸首，以及(ii) 应用程序在CGRA上有足够的并行性，可以在需要时自由地用芯片面积换取性能。

一种方式：我们可以像Cool Mega-Array（CMA-1和CMA-2）那样，让架构侧重于流计算，其中处理器向CGRA提供输入，在适当的时候，使用无时钟结构进行计算。这样的间歇性无时钟的架构是高能效的，该芯片的漏电功率可以低至1毫瓦。CMA架构在使用24位数据路径时，能够达到89.28 GOPS/Watt。CMA架构仍在研究中，最近的工作集中在提高性能（通过VPCMA中的可变延迟流水线）或通过体偏压进一步降低功耗。

另一种方式：利用电压和频率的动态缩放（DVFS）来获得功耗优势，使用定点（而不是浮点）数字表示。或是采取分立的电源控制器（实现时钟门控），以减少空闲时的能量使用。

### 1、架构硬件特性

首先分析本课题中面向机载智能算法的可重构架构所具备的硬件特性：

1、阵列具备了片上高速缓存。

本方案中配置信息的高速缓存结构将可重构处理器近期将要执行的配置信息存储在片上高速缓存中，当可重构处理器执行这些配置时，可以从片上配置信息高速缓存中获得，不需要去外部存储器中读取。由于片上高速缓存的访问延时（一般为几个时钟周期）要远远低于外部存储器，因此配置信息的高速缓存结构可以减小配置信息传输时间对可重构处理器性能的影响。在配置信息的高速缓存中，只存储当前使用的部分配置信息，有利于减小存储在片上的配置信息数据量可以减小配置信息存储器的面积和功耗。在可重构处理器运行过程中，配置信息的高速缓存结构可以实时地为可重构处理器读取新的配置信息，不需要重新初始化可重构处理器，保持了可重构处理器的灵活性。针对发生缓存缺失的情况（可重构处理器执行的配置信息不在片上缓存中，需要从外部存储器中读取该配置信息到可重构处理器内部，可重构处理器需要等待配置信息传输完成才能开始执行，这会降低可重构处理器的性能），本课题进一步提出了配置信息的分组预取方法，将配置信息按照不同的计算任务进行分组，对于完成同一个计算任务的一组配置信息进行预取，可以减少配置信息的缓存结构中发生缓存缺失的次数。

2、能够动态调整访存带宽，降低功耗而不降低性能。

因为神经网络计算过程中对存储容量和带宽有巨大的需求，因此计算架构需要很高的DRAM带宽接口。但是神经网络在运行不同层的时候DRAM带宽需求会发生波动。通常全连接层的带宽需求很高，而卷积层因为有大量权重复用，带宽需求较低。如果DRAM一直保持高带宽，将会造成功耗的浪费。给定带宽下，不同的线宽和电压下的功耗不同，并存在一个功耗极小点。本课题将根据神经网络在架构上的执行时间和访存量，评估每一神经网络层的带宽需求，再调节线宽和电压/频率来最优化DRAM功耗，其中频率是正相关于电压的。本课题将研究可编程的存储接口设计：包括研究物理上如何设计可编程接口，能够对线宽、电压和频率同时调节以降低功耗；并研究动态配置机制以快速调节DRAM带宽，而不影响系统性能。

3、支持可变精度，提高计算能效和阵列利用率。

尽管CGRA擅长整数和浮点数运算，但不支持可变的数据精度就会造成大量的性能和能耗浪费。多年来许多CGRAs的报告显示，他们粗粒度可重构架构擅长8/16/32位整数运算（相较于Nvidia GPU），也因此引起深度学习推理界的兴趣；并且那些包含浮点单元的CGRA可以与现代GPU的性能竞争，有时甚至超过它们（例如，Plasticine架构能够提供12.3 TFLOP/s的性能，可与那一代的GPU相媲美，而早期的Redefine和SGMF架构可以分别提供300和840 GFLOP/s。甚至更早，WaveScalar架构能够达到128 GFLOP/s，这在当时远远领先于GPU）。

对于运算单元层面来说，除了配置的缓存问题与可编程存储接口的设计适配程度，则需要通过重构选择不同的计算位宽、计算通道、组合不同的功能模块，提高芯片灵活性。传统的乘加单元设计采用固定的数据精度（通常为16比特），如果仍执行低比特的计算任务就会造成大量的性能和能耗浪费。而精度自适应的可重构乘累加计算单元为了减小神经网络的规模，在实际使用时常常对网络进行定点化压缩处理，使得网络的数据位宽和权重可以是8比特、4比特、2比特甚至1比特。精度可配置的计算单元则可以根据需求的不同，支持可变位宽神经网络运算（int8和fp16），实现硬件资源的最大化利用。并且针对低比特的网络，往往具有更高的计算并行度，提高了计算能效。

此外，因为不同神经网络（如卷积神经网络、递归神经网络）有各自的特殊的计算，并且除卷积操作以外还含有全连接操作、池化、门控、激活等计算，运算单元需要复用多种硬件资源，提高芯片的灵活性。因此才需要研究高度灵活的多精度、功能可配的运算单元。

4、利用近似计算技术，精简化计算单元的实现逻辑以提升能效比。

在神经网络的层内运算中，除了大量密集计算的特性，神经网络计算还有一个很重要的特点是容错性，即某个神经元输出的微小偏差对输出结果影响不大。这是因为神经网络的输出通常是分类或者识别的结果，并不需要特别精确的数值表示，某个神经元的误差对最终分类/识别结果的影响不大。因为神经网络的容错性，我们利用近似计算技术，通过精简计算单元的实现逻辑来降低功耗，并带来能效上的提升。本课题设计了一个近似精度可配置的乘加单元。在乘加单元内部设计多个精度控制开关，在计算过程中根据计算任务的需求关闭部分数据通路以节省能耗。同时，设计误差修复的神经网络训练方法，在训练神经网络过程中引入计算单元的近似误差，通过重训练保证网络收敛。因此，网络精度不会应为近似误差造成损失，同时因为不必要的计算逻辑关闭，节省了大量能耗，大大提升了计算能量效率。

5、针对MCTS算法提高硬件契合度

MCTS（蒙特卡洛树搜索算法） 的并行方法主要分为三种。在当前场景下，叶节点的并行并不适用。树并行相对复杂和繁琐，涉及到多线程之间的通信等待，并行效果也不一定好，推荐使用根并行的方法。根并行方法也分为single-run和multiple-runs。其中 single-run，每个线程采用不同的随机种子各自分别搜索一棵树，到底一定一时间后，将结果合并。multiple-runs同样是每个线程采用不同的随机种子各自分别搜索一棵树，但每隔一段时间，会将搜索结果合并，其中访问次数的加和需要除以线程数，这样会有更好的效果。

### 2、优化方向

计算机体系结构的发展过程中，逐渐形成了由编程语言、编译器中间表示、指令级架构、寄存器传输级逻辑等抽象层次构成的层次化中间层。而中间层是计算行业增长和生产力进步的主要驱动力。尽管如今大多数计算机科学专业的从业者可能对现代微处理器的工作原理和芯片制造的工艺流程一知半解，但是通过维护这些层层相扣的中间层，计算机专业从业者得以在更高的抽象层次开展工作。

然而每一个中间层的引入，都会对应用在芯片上的性能造成损失。当芯片的性能不再随着摩尔定律的前进而增长时，这部分性能就成为了尚未充分发掘的金矿。如今学术研究中火热的领域定制，如领域定制算法库、编译器、架构、器件等等，都是跨越多个中间层的尝试。

那么要想做好领域定制算法、硬件的定制化设计，以达到挖掘性能的提升空间的目的——软硬件协同设计（Software-hardware Codesign）是很重要、也越来越受欢迎的设计方式，也可以理解为系统级的合作设计——根据算法需求定制硬件、修改硬件架构以更适合专用领域；同时算法工程师根据硬件设计者反馈回的硬件资源或架构特性的约束，去修改和优化算法，以更好地适应可重构硬件的特点和发挥其优势。

近年已经有许多基于软硬协同设计的算法优化研究，很好地向大家展示了如何充分利用硬件资源和架构特点，去优化传统算法、突破传统算法理论的限制，极大地降低系统的部署成本，显著地提升算法的效率。

### 算法-硬件协同设计（Co-design）

那么根据可重构架构所具备的硬件特性，在此提出一些可能的未来优化思路：

1、优化算法多层嵌套循环

在蒙特卡洛树搜索以及神经网络算法中，都有可能出现多层级的嵌套循环流水问题，当对多层循环尤其是非完美循环进行软件流水时，流水核心包含的内容较多，可能会导致动态重构的配置信息量超出存储能力。因此，需要逐层级地提取多层流水中重复的计算过程，得到多层流水核心中的最小重复单元，从而减少配置信息量。在执行时采用多级索引的方式，相同计算模式可复用同一套配置信息单元，通过拼接各个外层迭代的配置信息元素，来实现完整的计算过程。但是，随着压缩的程度增大，会带来两方面的代价——首先，配置信息所能表达的灵活度会降低，导致更难搜索到从算子到计算阵列的合法映射；其次，索引系统的硬件代价会增加。

因此，除了将配置信息的压缩方案建模为有约束最优化问题、采取更优的压缩方案以外，被映射的算法也应当尝试在源头去解决动态重构的配置信息量超出存储能力的问题。也就是端硬件存储资源的限制，指示着神经网络算法的优化方向——对于多层级的嵌套循环流水问题，尽量减少循环（尤其是非完美循环）的层数，以降低流水核心包含的信息量；进而避免压缩各个外层迭代的配置信息，所导致的搜索映射的难度、时间复杂度和硬件资源开销上升。

2、提升CGRA的利用率

CGRA本身是一个网状结构，有点对点和广播操作的多种选择，高度适用于深度学习卷积模式。可重构数据通路一般由处理单元（Processing Unit, PE）和互联两部分组成。与超标量处理器的数据通路类似，可重构数据通路也有多个功能单元即 PE，而且数目更多。PE不是独立的处理器，而是类似于单核处理器执行单元的硬件结构，用于完成各种计算操作。空间并行的 PE 利用并行化执行方式（例如 ILP）来提升计算能力。互联（也常称为路由）的功能是在不同 PE 之间以及 PE 与外部模块之间进行快速通信，例如将一个 PE 的计算结果直接传递到另一个 PE 的输入。互联的结构通常是开关网络或多路选择器网络。

用户（或编译器）正是通过这个网状结构决定数据如何在系统中流动。有多种方式将数据带入/带出结构。一种常见的方式是在主处理器的内存中映射设备（内存映射），并由主处理器协调执行。另一种方法是包括（通用）地址生成器（AG），它可以被配置为使用某种模式（通常对应于应用程序的嵌套循环）访问外部存储器，并通过阵列推动加载的数据。第三种选择是让PE同时既做计算又做地址生成。图1:b显示了一个RC元件的内部，包括一个ALU（整数和/或浮点运算能力），两个多路复用器（MUX），以及一个用于存储的本地SRAM。两个多路复用器决定对哪个外部输入进行操作。这些输入通常是相邻RC的输出、本地SRAM、一个常数或之前的输出（例如，用于累积）。ALU的输出也同样连接到相邻的RC、本地SRAM，或回到一个MUXes。

例如，早期的CGRAs通常包括结构内的细粒度可重构元素（查找表，LUT）。虽然网状拓扑结构是迄今为止最常用的，但有些设计选择了环状或线性阵列拓扑结构。最后，网络中的数据流控制可以有不同的复杂性（例如，令牌（token）或标签令牌(tagged-token)）。以Eyeriss为例，它作为CGRA深度学习推理引擎，计算大部分是基于乘积操作，十分注重重新配置网络访问模式，并利用中间激活的稀疏性来增加观察带宽。因此在推断AlexNET时可以利用几乎100%的CGRA资源。

3、混合粒度的可重构系统

与FPGA等相结合，比如最近赛灵思Versal [150], [151]系列将很大一部分硅片分配给可编程的、邻接的、处理元素的网状CGRA结构。FPGA支持深度学习CGRA，但为了支持人工智能引擎未来可能的新的深度学习功能，人工智能引擎可以直接连接可重构硅片的剩余FPGA部分，这是赛灵思闻名的细粒度可重构单元的形式。该系统本身就是一种尝试，结合了细粒度和粗粒度可重构的优点。

现代的CGRAs（例如Plasticine，SIMD-RA）确实已经做到了SIMD级别的并行性，以支持并行提取指令。因此可以考虑多线程、多任务的编程模型，而恰恰任务依赖性与CGRA中PE所拥有的计算和存储资源非常匹配，这些CGRAs有算子簇和scratchpad，其中任务依赖性，就决定了数据如何在这些CGRAs上流动（利用任务间和任务内的并行性和数据定位）。

对于FPGA和高层次综合来说，总是存在着这些大型内核可能无法装入单个FPGA的情况；然而，CGRAs可以存储多个context和内核，在它们之间切换的开销很小，为执行整个应用程序以及利用内核间时间和空间数据定位的机会提供了可能性。这也可以考虑多加以利用。

4、MCTS算法执行方式

实际上无论是single-run方式还是multiple-runs方式，和单线程进行同样的次数搜索，尽管并行的方法会进行一些重复的搜索，看起来总的有效搜索次数其实低于单线程的方法，但是在一些实验中反而效果更好。主要原因可能是单线程的搜索更容易进入局部最优，而多线程的搜索能从某种程度上规避这一点。

在该问题中，无论是采用single-run，multiple-runs的方法，都需要对搜索次数或时间以及并行的线程数进行重新测定，以满足最后的效果要求。同时，如果是multiple-runs还需要对什么时候进行同步效果更好进行进一步测算研究。

5、利用CGRA配置速度优势实现算法多样性

机载智能计算单元需要设计高性能CPU主控单元来支持决策树类算法（如MCTS）搜索过程的处理，并设置面向神经网络运算加速的专用协处理器来支持神经网络的快速处理。综合考虑已有技术成熟度，本课题选用高性能CPU主控，多核CGRA联协处理神经网络的方式，进行机载智能计算单元设计。

我们知道MCTS搜索过程是较为典型的决策树运算过程。而决策树算法的特点是根据不同的条件进行判断和分支选择，并且有非常多的条件规则和分支选择规则。在机载应用中，决策树算法的树生成过程和树剪枝过程，即决策树的学习过程，通常在地面服务器中进行，只将优化后的树形结构移植到机载环境中执行。但在某些情况下，同样需要在机载计算系统中实时进行决策树算法的动态生成和调整。

决策树结构可以和if-then规则对应，对于if-then形式的规则，体现在处理器中的操作包括：

1）从存储空间（外部存储器或者内存）中获取规则条件；

2）根据规则条件，从规则数据库中搜索相应规则；

3）依据规则条件，执行逻辑判断，按照规则结果进行分支选择；

因此，决策树算法运行时的计算需求涉及两个方面，一方面是处理器执行数据读写和执行分支跳转类指令的能力；一方面是计算机中的数据库管理能力，反映在硬件的存储器性能和操作系统的数据库系统或文件系统支持。

其中，处理器执行能力，依赖于处理器架构，并受编译器能力影响。处理器的处理能力一方面在很大程度上依赖流水线处理技术，将指令处理流水化，以充分利用运算器和存储器的资源，加快处理速度。而在执行跳转类指令时，由于跳转类指令会破坏流水线，导致流水中断。另一方面，跳转类指令往往需要跨地址进行数据寻址和指令寻址，会破坏存储器中指令和数据的时间局部性和空间局部性，导致取指和取数据延迟增加。因此，处理器的指令处理瓶颈，往往在跳转类指令。处理器执行跳转类指令的能力，是影响决策类算法计算速度的重要指标。除处理器的执行能力外，编译器对决策类算法的计算效率也有影响。决策类算法通常采用高级语言实现，编译器将其编译为机器语言，编译过程中对高级程序语言的优化方式、优化水平，影响生成的机器语言的执行效率，最终反映在算法的指令规模和处理器执行各条指令的效率。

CPU处理器架构核心在于设置了运算器、控制器和存储器，利用专门的控制器控制指令的执行，同时设置了大量的数据寄存器和状态寄存器进行指令的控制和运行。因此，CPU架构特性加上指令集的设置，使其能够较好的支持控制类指令的执行。

另一方面，决策类算法通常需要构建决策树，或者生成if-then规则来存储整个决策算法，而决策树自身的结构复杂度，或者由决策树生成的规则的规模，都非常高。在决策类算法执行过程中，根据每一步的输入条件，需要从众多的分支或规则中搜索满足条件的规则。因此，计算系统对决策树的存储和管理能力，以及对大量规则的管理能力，直接影响每一步的决策效率。具体的，若用树形结构存储决策树，则内存空间大小能否满足决策树的存储规模需求，是影响决策树搜索速度的最大限制。若内存空间不足，则需将决策树的树形结构存储至外部存储器，而外部存储器的访存速度相比于内存，平均会下降一至两个数量级。因此，从内存空间的指标，决策树算法对计算系统的存储空间，尤其是内部存储空间的能力需求较大。若用规则形式存储决策树，则需用文件系统或数据库系统存储和管理众多的规则，规则的存储和管理方式，以及规则查询方法，直接影响规则搜索的效率。因此，计算系统是否具备高效的数据库管理能力，是决策规则能否快速查找的关键。

综上所述，决策树类算法对计算系统的能力需求，主要包括两个方面：计算能力，尤其是处理大量复杂的分支跳转类控制指令的能力；存储能力，尤其是对大量离散化数据的高效管理能力。

（2）神经网络运算需求

神经网络算法的两大特性即计算密集性和存储密集性。计算密集性体现在神经网络算法的三个方面：（1）计算规模十分庞大，如卷积神经网络算法中较为简单的AlexNet算法，其乘加运算的次数就有几十亿之多；（2）计算指令占整个算法指令数的比例非常高，在神经网络算法中，基于神经网络的简单且标准的网络模型和结构，算法的绝大部分操作都是卷积运算，而卷积运算仅包括乘法和加法运算。（3）深度神经网络算法在执行前向计算时，计算指令分布密集，由深度神经网络的结构，决定了网络的较低层都是标准卷积运算，中间夹杂的诸如dropout等概率选择，也可以转换为乘法运算；同时网络的较高层可能会有一层或多层全连接运算，其运算核心也是乘法和加法。在整个网络进行前向过程中，几乎没有控制类指令。因此，神经网络算法的典型计算密集性特性，使得该类算法十分适合利用具备大规模并行结构的处理器实现。

神经网络算法的存储密集性体现在算法模型的规模。一方面，通过对多种不同应用和不同架构的深度神经网络算法分析可知，深度神经网络算法模型的规模普遍较大，较简单的网络模型，其参数数量都在百万以上，而在计算机中存储时，若采用浮点数存储，则需百兆以上的存储空间。而深度神经网络算法复杂度不断增加，其模型规模必将不断增大，这对计算系统的存储资源提出了很大的需求。另一方面，通过对深度神经网络算法执行过程分析可知，深度神经网络算法执行中，需要频繁访问存储空间以获取卷积核、特征图等矩阵形式的数据，同时需要将每一步运算中产生的新特征图以及更新的卷积核参数写回至存储空间中。因此在网络运行过程中，处理器与存储器之间将会进行频繁且巨量的数据传输，这对处理器内部数据总线的带宽，提出了非常高的需求。

综上所述，神经网络算法对计算系统的能力需求，主要包括两个方面：计算能力，尤其是处理大规模并行化的乘加运算的能力；存储能力，尤其是高速存储空间大小以及处理器与存储器之间的数据总线带宽。

作为神经网络加速的协处理器，CGRA弥补了FPGA的两个大的性能和可用性问题：编译时间和重新配置的开销。众所周知，将设计编译到FPGA上是一项耗时的任务。现代的FPGA体积较大，具有许多独特的特性，使得在其上进行放置和布线不是一件容易的事。随着FPGA门数规模的增长，占用综合工具的内存容量也在增长，针对新器件（如英特尔Stratix 10）进行编译的工具，会消耗系统内存的很大一部分，十分限制一台机器中可能的并行编译的数量。通过粗化和减少可重构单元的数量，编译时间可以大大减少（例如，近三个数量级）。此外，FPGA一旦被编程通常被期望运行很长时间，因为context切换会产生相对较大的（数秒）开销。

因此可以考虑支持更多的决策树类算法，多套算法取长补短搭配使用，从而发挥CGRA配置速度的优势。

### 3、 总结

近年来，人工智能技术已在各类民用场景中发挥着愈发重要的作用，被广泛应用于机器视觉、语音识别、自动驾驶等各个领域。信息化和智能化也正在重塑未来战争的形态，军事智能成为竞争的焦点。为适应智能化战争的需求，重大装备（如各型地基雷达、机载雷达、无人作战平台等）需要具备高灵敏、高性能、强实时、智能化的信息处理能力，以面对现代战场环境的高度复杂性、动态性和不确定性，不断获得情报信息。这就要求核心信息处理芯片必须兼具高密度的智能计算能力，具备高能效、可编程、高灵活、自适应等特点。

高密度计算与人工智能算法各具特点。有一些神经网络算法运算规整、对计算精度要求高，数据速率高、数据量大，一般采用16位/32位定点计算；也有些算法在数据处理中，算法灵活多变、动态范围较大，需要采用单精度和双精度浮点计算。而人工智能算法具有内生容错性，权重模型普遍采用8位定点，有些场景下甚至可以被压缩为1位；神经网络模型具有较大冗余性，通过剪枝等方法处理后，运算不再规整。而另一方面，神经网络与智能算法又紧密联系，在一个完整的人工智能应用中，这二者缺一不可。以蒙特卡洛树搜索算法为例，节点拓展与节点选择评估就有着明显的差别，节点拓展的运算较为复杂但是并行度的要求并不高，而节点选择评估需要用到神经网络算法，有着极为大量的卷积操作，但是操作却又单一，只有乘加操作，这就要求硬件计算平台必须兼顾这两类差异明显的计算任务。

此外，由于军事环境具有边界不确定性和信息不完整性的特点，人工智能算法的训练数据集必然无法充分反映应用环境的情况，这就需要信息系统具有自适应、自学习的能力，通过应用过程中的增量学习，提高针对实际环境的信息处理效果。这就要求硬件计算平台能够支持人工智能算法的在线训练。

回顾近三十年信息处理芯片的发展历史，其性能得益于集成电路工艺的进步和计算架构的演进。传统的以冯·诺依曼架构为基础的通用处理器具有极强的灵活性，但是其性能和功耗并不理想。相对的，专用计算执行速度快，功耗低。但代价是灵活性和扩展性很差。高密度神经网络算法的计算量和数据量都十分庞大，而且处于快速演进之中。使用通用处理器或者完全定制化的专用电路都没有办法同时满足性能、能效和灵活性要求。当前，集成电路制造工艺已进入“后摩尔时代”，工艺进步带来的性能提升已非常有限，探索新型计算架构成为唯一选择。

可重构计算（Reconfigurable Computing）将软件的灵活性和硬件的高效性结合在一起，在性能、功耗和灵活性等关键指标之间取得更好的平衡，因此更适合处理这些任务。近年来人工智能芯片的发展历程以充分说明了这一趋势。针对神经网络算法，传统的CPU器件(如Intel Xeon E5)能效只有0.2~1GOPS/W，而采用可重构架构的神经网络计算芯片（如清华大学Thinker芯片）的能效则可达到TOPS/W量级。

面向重大装备的复杂数据处理和智能计算需求，研究动态可重构的多模态智能处理器，将能够提升处理器件效能和硅资源利用率，解决支持领域单一的问题，有助于实现重大装备的网络化、小型化、一体化和智能化。

虽然现有工作已经具备远超通用处理器的能效和高于专用计算器件的灵活度，但是，对于树搜索和神经网络融合的任务来说，仍然有关键性不足。首先，多模态处理能力不足，无法同时适应节点搜索和神经网络的不同位宽计算，以及张量类型数据和时间序列数据同时处理等需求。第二，该领域大部分工作还不具备成熟的片上学习能力。第三，在编译技术上，目前还没有面向高密度计算与人工智能融合的统一编译方法。

因此，我们设计了一种高密度计算和人工智能融合共存的多模态处理器架构，并开发配套的异构编译方法。该架构支持资源复用和多层级动态重构，通过异构的计算阵列，满足不同位宽和计算类型的需求，同时实现树搜索算法、神经网络推理和训练的高效执行。根据设计特点分析，机载智能计算单元采用由高性能粗粒度可重构计算阵列集成得到的可重构处理器作为协处理器提供神经网络加速处理功能。

我们通过对MCTS等决策树类典型算法在CGRA中映射实现的分析，验证了CGRA中运行决策树类典型算法的可行性。我们经过进一步的梳理与验证，明确了适合在 CGRA 中进行加速的算法模块，和由于自身算法特点而在 CGRA 中实现效率低下的物理层模块，为下一步更有针对性的深入研究与验证工作打下扎实 基础。此外，根据决策树算法位宽变化、访存频繁、循环嵌套多等特点，研究团队还对 CGRA 的架构进行了并行化、访存方式改进等优化设计，进一步提高了算法与硬件架构的契合度。

总结起来， 我们完成了对机载智能算法蒙特卡洛树搜索算法，以及其配套需要使用的深度神经网络的整体计算架构的适配性研究和设计工作。我们介绍了可重构架构的历史和发展现状，并且阐述了可重构架构的设计流程，分为算法设计、架构设计、工具链设计以及调度优化。我们分析了整套 MCTS算法的实现思路，着重介绍了在我们提出的架构上实现运算的 各类算法原理和可重构架构的算子映射方式。我们设计并完善了架构各方面细节，具体包括但不限于 PE 模块算子、 PE 模块互联通道、 IO、存储模块等等。最终还针对上述分析的算法在模拟器上进一步仿真进行了功能验证。

基于对决策树等智能终端应用中采用的神经网络的多样性、共通性分析，实现了算子与运行模式的分离，加速其中并行逻辑，优化了串行执行逻辑的数据通路，从而提高了处理速度，同时构建了动态可配置机制，降低了芯片面积，提高了能效比。

CGRA 由于其出色的性能、功耗与灵活性，是智能装备信息处理核心层的重要实现方式。未来有机会我们还将设计完成面向该架构的指令格式以及编译器功能，实现更高效能高效率算法映射，为智能装备领域提供更加高效灵活的硬件技术基础。