# 介绍

（为什么要优化？新的技术？本文的技术？）

在深度学习的系统中，计算能力的优化是极为重要的。在高维卷积，成百上千的过滤器和通道可以同时处理。目前最先进的卷积神经网络（CNN）需要几百兆的过滤器权值的存储和30k-600k的操作对于每像素的输入。在这样的卷积网络中，重要数据的移动所带来的吞吐量挑战着底层的计算存储硬件。

随着内存计算（PIM）或临近数据处理（NDP）的产生，这些数据移动所带来的挑战得到了很好的解决。它们的核心思想是让计算器尽可能地靠近内存，从而达到减少数据移动时所带来的影响。PIM的最新进展，提出了三维堆叠内存架构，使处理引擎（PE）接近内存中的数据。这种新的体系结构允许堆叠内存提供计算和存储功能。另一方面，神经网络是数据密集型和高度并行的，有很多机会可以利用不同层次的并行性缩减神经网络的时间总消耗。神经网络在PIM结构上运行，可以充分利用卷积的平行度。然而，大量的中间处理结果（即部分）是由卷积并行生成，这需要相当大的额外的内存来存储这些中间数据。

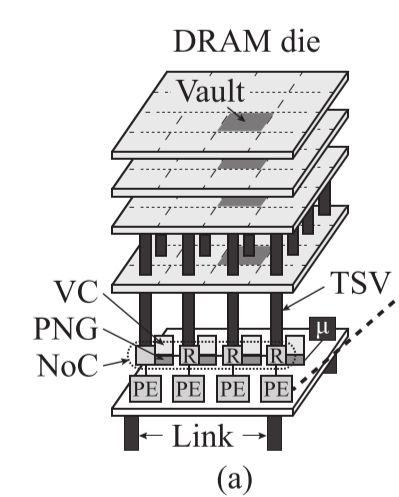
本文提出了两种新颖的数据调度算法，它们都采用了高效的PIM架构，实现了卷积神经网络的并行计算。它们重新分配了卷积神经网络中的中间数据。它们的目标是最大化应用吞吐量同时最小化总时间的前提下生成新的任务安排序列。利用优化算法重新改造后的卷积神经网络，不仅仅充分利用了已有的硬件设施，同时将会大大的提高其吞吐量。

# 系统模型和概念

（Neurocube？CNN？多发射？样例？）

## Neurocube

Neurocube是一种可编程、可扩展、功耗高的数字体系结构，它整合了具有高度并行计算的HMC，可编程的神经序列生成器。Neurocube是以内存为中心的神经计算机构，利用数据驱动的性质和已知内存访问模式的神经启发算法实现了驱动数据流动的计算单元。



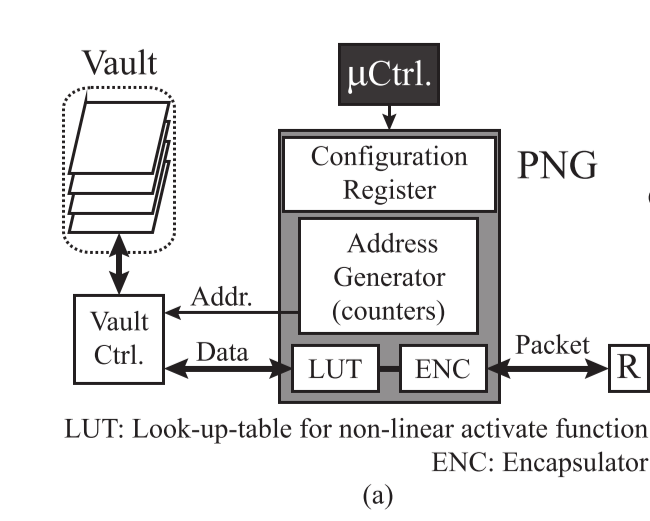
Neurocube的基本结构如上图所示，可以将分为两个层次：存储层和逻辑层。

### 存储层

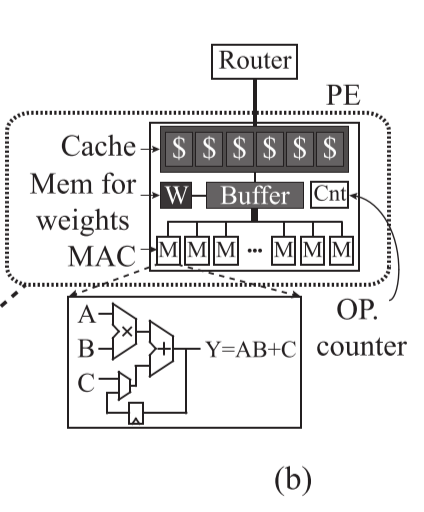
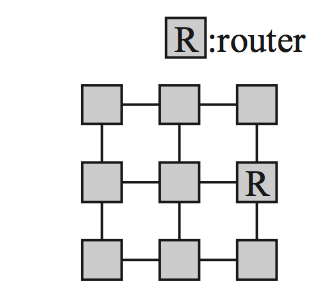
存储层的结构设计和HMC的结构相同，是由多个内存块重叠在一起构成，它们之间利用高速传输介质硅通孔（through-silicon via-TSV）连接在一起。在存储层中，内存块被划分为多个vault，通常划分为16份。每个vault在功能和操作上都是相互独立的。

每个vault都与一个内存控制器（VC）连接，其中内存控制器位于逻辑层，负责数据的存储管理。

### 逻辑层



逻辑层的基本结构如上图所示，每个VC与一个可编程的神经序列生成器（PNG）相连接，而这个序列器通过由2D网格网络连接的路由（Router）与PE通信。PNG负责控制神经计算所需的数据流动，生成在前一层连接的神经计算在内存中的地址和权值。数据包在PNG中被封装了源ID（Vault ID）和目的ID（PE ID），通过在Router发送给对应的计算单元（PE）。



PE的结构如上图所示，它是主要的计算单元，包含许多乘法累加器（MAC），一个cache内存，一个存放数据包的buffer，一个存放权值的buffer。

当对应MAC的数据包到达PE时，需要考虑两种情况

* 若它的OP-ID大于当前OP计数器的计数值时，它将被存入cache
* 否则直接放入Buffer，以供MAC使用和计算。

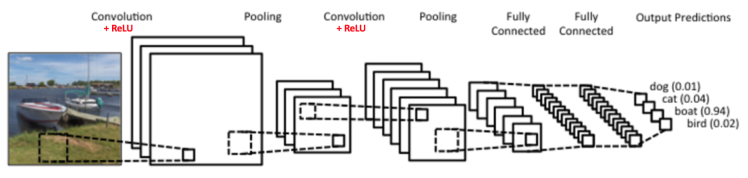
当当前轮次操作全部完成后，OP计数器发生变化时，同时将cache中存放的提前抵达的数据包放入对应的Buffer。

## CNN

S. Haykin, “Neural Networks and Learning Machines”, 3rd ed., McMaster University, Hamilton, Ontario, Canada, 2009

I. Goodfellow, Y. Bengio, A. Courville, “Deep learning”, Cambridge, Massachusetts, The MIT Press, 2016

卷积神经网络是一种专门用来处理具有类似网格结构的数据的神经网络。例如图像数据，即可以看作二维的像素网格。



如上图所示，一个标准CNN的层级可以被划分为卷积层，池化层和全连接层。卷积层是CNN中负责主要计算的层，它将输入的图像数据不同局部的矩阵和卷积核矩阵利用一个滑动窗口进行相乘相加操作，得到新的张量。对于卷积层的输出，还要通过ReLU激活函数，将输出的张量中的小于0的位置对应的元素值都变为0。池化层是对输入张量的各个子矩阵进行压缩，每次池化操作通常对对应区域的进行最大化或者平均化操作，得到池化后的结果。全连接层计算它的输入的内积和权值，可以被看作是一种特殊的卷积层。

一个CNN结构可以被抽象成一个有向无环图（DAG）G=(V, E, P, R)，其中V={T\_1, …, T\_n}是一个节点集合，每个节点表示一次卷机或池化操作。E\subseteq V\times V是一个边集合，每条有向边(V\_i, V\_j)\in E(V\_i, V\_j\in V)表示节点V\_i和节点V\_j之间的数据依赖关系，即节点V\_j需要节点V\_i的处理结果，定义这个处理结果为I\_{i,j}。P表示这个图需要循环的次数。

## 样例

# 算法

## 输入

定义一个有向无环图DAG，G=(V, E, P, R)，R表示图G的retiming，用来计算retiming。

定义K表示PE的总个数。

定义T表示G需要循环的次数。

## 算法1

当顾客刚进麦当劳准备买食物的时候，顾客都会优先去排人数少的窗口的队伍，因为人数少就意味着等待的时间短，而如果有空闲的窗口，顾客肯定会优先选择空闲的窗口。而这个算法就如同顾客买食物的心理一样，在调度的任务的时候，哪个PE空闲了就把下一个任务节点直接放到那个空闲的PE上去计算，直到所有的任务都计算完成。而算法的关键在于如何在等待运行的任务中选择下一个任务能使最终的结果得到局部最优解：

* 如果下一个任务节点为随机选取，那么考虑到数据依赖关系时，会造成很大的延迟，得到最优解的概率很低，算法效率低。
* 如果下一个任务节点为选取耗时短的节点，那么将会造成运行时间长的节点在等待队列等待的时间较长。
* 如果下一个任务节点为选取开始时间早的节点，那么对于数据传输时间长的节点也将会在等待队列中等待较长的时间。

随机选取很显然是不可取，因为下一个任务的选取应该遵循拓扑结构。选取耗时较短的节点和选取开始时间早的节点都可以作为备选条件，因为考虑到节点依赖的关系，每个节点V\_{i}在等待队列中等待的时间不会超过T\times N，N表示与V\_{i}拓扑序列相同的节点个数。

算法的具体执行步骤如下（以下一个节点的选取为耗时较短优先为例）：

1. 获取有向无环图G的拓扑序列。
2. 构造一个长度为K的空闲PE优先队列Q\_{freepe}，以PE的ID小为优先级，初始化丢入K个空闲PE的id。
3. 构造一个运行节点优先队列Q\_{running}，以节点结束时间早为优先级。构造一个节点等待队列Q\_{waiting}，以节点运行时间短为优先级。
4. 根据拓扑序列，对T\_{j}轮所有入度为0的节点V\_{i}，
5. 如果Q\_{freepe}不为空：从Q\_{freepe}取出空闲PE的peid，构建运行节点V\_{running}，赋值其运行的PE为peid，将其丢入Q\_{running}。
6. 如果Q\_{freepe}为空： 构建等待节点V\_{waiting}，将其丢入Q\_{waiting}。
7. while Q\_{running}不为空：
8. 丢出队首节点V\_{top}，将队首节点的peid丢入Q\_{freepe}。
9. 获取节点V\_{top}在拓扑序列中的下一个（可能为多个，多个拓扑序相同的节点）节点V\_{j}。V\_{j}的轮次设置与V\_{top}相同，并设置其开始时间为V\_{top}.endtime + edge[V\_{top}][V\_{j}].cost，将其丢入Q\_{waiting}。
10. while Q\_{waiting}不为空：
11. 如果Q\_{freepe}为空，break。
12. 取出Q\_{waiting}队首节点V\_{wtop}，取出Q\_{freepe}队首peid，构建运行节点V\_{running}，丢入Q\_{running}。

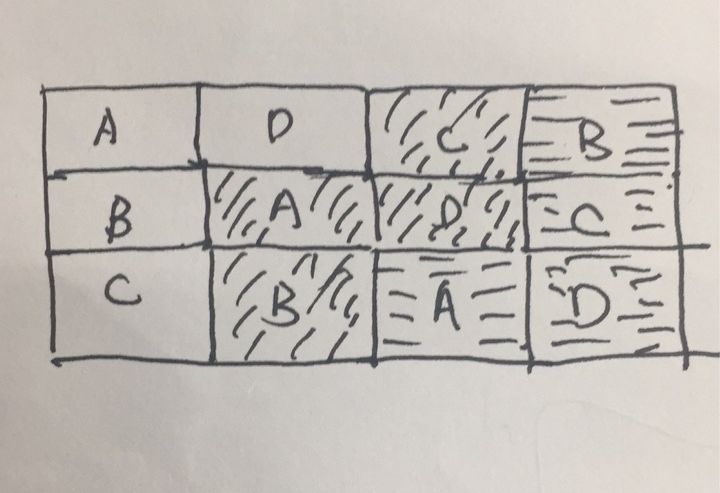
时间复杂度的分析：每轮中，每条边都只循环了一次，而从等待队列中取节点几乎每次最多只取K次，故时间复杂度可以认为是 O（T\times (E + K)），其中E表示边的数量。因为在CNN中，K远小于E，故时间复杂度可以认为是O（T\times E）。

## 算法2

通过测试数据的输出结果发现，在数据量大的情况下，算法1的总时间消耗不能达到一个较优的解，且 CPU的利用率不高。主要的原因在于算法1没能把有数据依赖的两个节点在传输数据的时所消耗的时间有效的利用起来。为了提高在数据量大的情况下CPU的利用率同时也不影响总时间消耗，提出了算法2。在介绍算法2之前先介绍一个基础算法。这个基础算法可以分为两个步骤：规则排列，恢复依赖。

* 规则排列

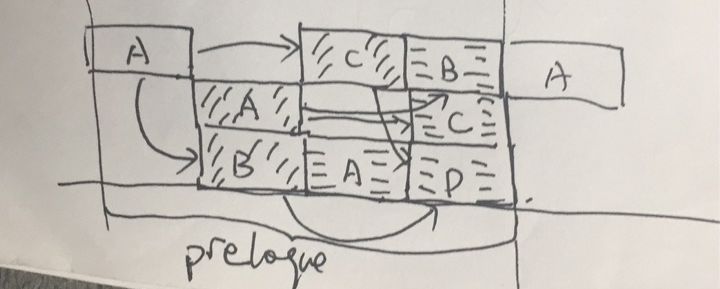
将所有的任务节点不考虑数据依赖的条件下按照一定的规则安排在各个PE的每个时间段，为了达到局部的最优解，每个PE的每个时间段前后的任务之间没有时间等待。最简单的排列规则是直接按照顺序依次安排，如下图所示，横坐标表示时间线，纵坐标表示PE：



需要注意的是，这样的安排可以不止安排一轮，可以安排多轮，直至满足了特定的条件后才终止安排。若安排了S轮，则可以认为这S轮为一个周期。对于刚刚提到的“特定的条件”，即一个周期结束的判定条件，可以人为的设定最大轮数，也可以依据PE的利用率来限制，当利用率达到预期值时，就停止安排。需要注意的是，为了保证完整性，利用率的计算应该在每一轮安排结束后再进行计算。

* 恢复依赖

因为第一步的时候忽略了节点之间的数据依赖关系，但最后在计算总时间的时候还是要考虑数据以来关系的，故每个节点需要依据依赖关系找到它后面满足依赖关系的后续节点。如下图所示，



当一个节点在同一个周期内找不到满足依赖关系的后续节点，需要跨几个周期才能找到后续节点时，此时就需要利用重定时（retiming）操作来确定。

算法2利用上述的基础算法，结合多发射技术，重新对任务节点进行调度，并且在第一步的规则排列中，设计合理的规则以提高CPU的利用率。规则如下：对于任务安排， 尽可能让每个PE的总任务时间相差最小，同时每个PE里面安排的任务按照拓扑序列先后排序。换句话来说，就是让一次发射的m个PE中，安排后总时间最长的PE的运行时间尽可能少。

算法2的实现可以很多的方法，故本文只介绍算法2的大体方向，具体步骤如下：

1. 获取有向无环图G的拓扑序列。
2. while 不满足周期结束条件：
3. 依据规则在PE上安排G的各个节点。
4. 计算每次发射需要循环的轮数t。
5. for i = 0; i < t; i ++：
6. for V\_{j} in {Topology of G}：
7. for e in Edge[V\_{j}]：
8. V\_{k}为V\_{j}经过边e所到达的在安排的节点。
9. 获取V\_{k}在规则排列图中的具体位置，即获取开始时间，结束时间和PEID等信息。
10. 更新V\_{k}的开始时间为min(V\_{j}.endtime + e.cost, V\_{k}.starttime)。

虽然算法2从伪代码上来看比算法1短，但是如果考虑到代码运行效率前提下，还是需要一定的难度。

时间复杂度分析：从伪代码来看，每轮中，每条边都只循环了一次，所以时间复杂度为O（t\times E），其中E表示边的数量，t表示每次发射的循环次数。但在具体实现的过程中，尤其是伪代码的第9行可能会对最终的时间复杂度有所影响，一定程度上提高时间复杂度。

## 输出

算法最终输出总运行时间，CPU利用率，每个节点每轮的开始运行时间、结束运行时间、运行PE的ID。

# 实验