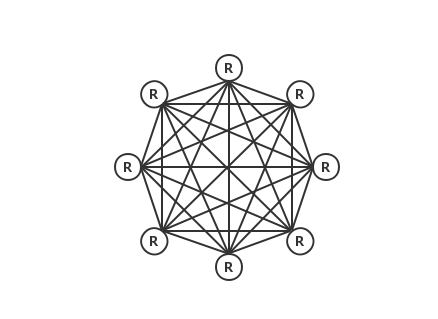
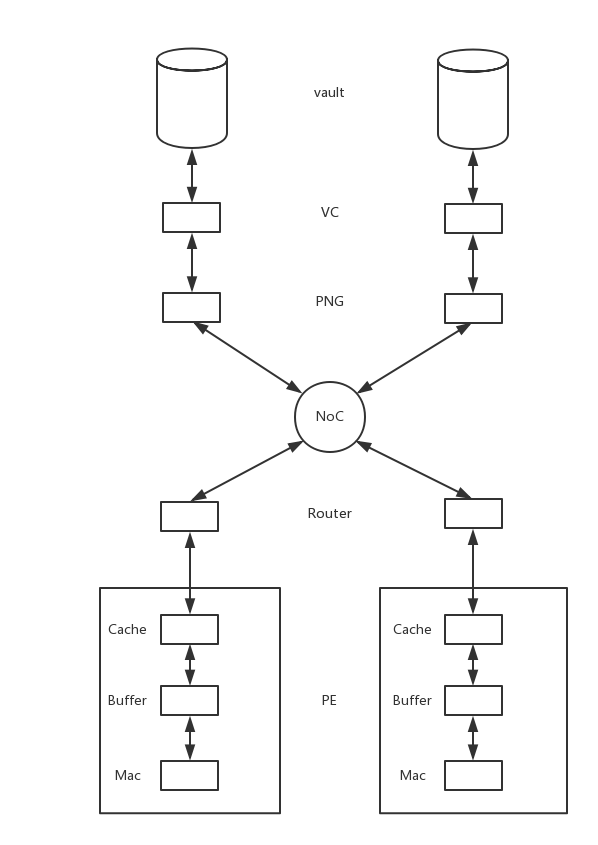
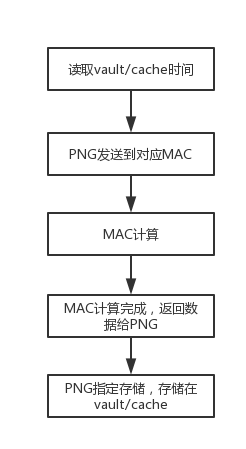
# 应用模型



通过输入的数据或是计算完的数据可以被存储vault（DRAM）和cache之中。众所周知，DRAM能够存储的数据远大于cache，但读取的速度缺慢了很多，所以**在存储数据的时候需要判断是存储在vault之中还是存储在cache之中**。由上一小节可知，PNG负责控制数据的流动，当计算开始时，PNG根据当前神经元所需要的数据去对应的存储位置取到数据，然后封装成对应的数据包，通过NoC网络传送给对应的PE进行计算。当PE中MAC计算完成后，将计算完成的结果传回给PNG，PNG判断后继需要这个数据的神经元，将其存储到对应PE的Cache之中，若Cache存储不下，则存储到对应的vault中。

时间消耗如下图所示。



# Algorithm

## 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。

## *Fast-Para-Conv*

*Fast-Para-Conv*分为如下几个步骤：

1. 计算每次发射所需要的核数和图循环次数。
2. 确定周期内图循环的组数和每个核任务种类。
3. 判断是否需要RETIMING。
4. 寻找“关键”节点集合（按照关键性排序）。
5. 从关键节点X开始，考虑其入度（出度）边，利用动态规划将部分放入cache。放不下的边，放入DRAM。放入cache的边对应的节点确定位置，放入DRAM的边对应的节点不确定位置。
6. 从已经确定位置的节点集合中，选取一个节点做与第5步一样的操作。若已经确定位置的节点集合为空，考虑未访问的节点中，满足1. 有入度边；2. 出度边上的节点与当前节点同核；其中一个条件，选择最长的节点进行第5步操作。
7. 循环第6步，直至所有节点都访问过。
8. 按照第5步到第7步中节点访问顺序，依次利用贪心思想确定放入DRAM的节点集合中的节点位置。
9. 计算每次发射所需要的核数和图循环次数。

若图的最大并发度是M，需要进行H=（K / M的上取整）次发射。其中若M整除K，

* 每次发射均需要的核数是M；
* 每次发射至少需要rmin= （T / H的下取整）次循环，至多需要rmax=（T / H的上取整）次循环。

若M不整除K，

* 前H-1次发射需要的核数是M，最后一次发射核数是K % M；
* 若设前H-1次发射每次循环x次，最后一次发射循环y次。所以在满足T=（H-1）\* x + y的情况下，选取总时间最小的一组（x，y）。

1. 确定周期内图循环的组数和每个核任务种类。

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。直接对所有任务进行排列具有很大的不确定性，每次任务的时间消耗不同，对最终的排列有着很大的影响。

考虑到一个周期只安排一轮图的话，可能会造成利用率较低，所以在一个周期内可以安排多轮，直至利用率达到预期或达到最大轮数后才终止安排。若安排了S轮，则可以认为这S轮为一个周期。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。

这个问题可以类比最小化最大值问题，可以用贪心的思想来解决。把PE看在箱子，消耗时间可以看作占用的体积，需要将计算任务放入箱子之中，让体积最大的箱子的体积尽可能小。对于需要安排多少轮任务在一个周期内，可以在安排第S轮任务时，在第S-1轮任务安排的结果上直接排列下一轮任务。伪代码：

*GenerateArrange(A, N)*

1. 定义S=0，Ratio = 0, Round = 0。LimitedRatio为最低利用率，LimitedRound为最大轮数。
2. 构建一个以结束时间短为优先级的优先队列Q。
3. 初始N个PE的结束时间pe.EndTime=0，加入队列Q。
4. While Ratio < LimitedRatio and Round < LimitedRound:
   1. Round = Round + 1
   2. 对保存了一轮任务的数组A以任务的时间长为第一优先级，拓扑序小为第二优先级进行排序。
   3. for i = 0 to A.length :
      1. 从队列取出队首PE，更新其结束时间=PE.EndTime + A[i].Cost，再将其加入队列。
   4. 计算当前利用率NowRatio
   5. if NowRatio > Ratio:
      1. S = Round
      2. Ratio = NowRatio
5. return S
6. 判断是否需要RETIMING。
7. 寻找“关键”节点集合（按照关键性排序）。

对所有任务节点，算法认为执行时间超过最长任务节点执行时间的80%的节点为“关键节点”，其中时间越长的关键性越大。

1. 从关键节点X开始，考虑其入度（出度）边，利用动态规划将部分放入cache。

对于“关键”节点集合中的节点X，首先它的位置需要尽量靠前，其次分别来考虑入度边和出度边。

对入度边来说，若节点X所在PE核为P，所有入度边的传输内存需要都放在P中的Local Cache中。若放得下，则全部放入，同时这些边所连的节点都排列的尽可能离X近。若放不下，需要挑选出几个任务放入DRAM之中。

对出度边来说，仅考虑出度边所连接的节点也在核P上的出度边。这些边的传输内存也需要都放在P中的Local Cache中。具体放置的安排和入度边放置时同理。

伪代码：

*BinPacking(Goods, Size):*

1. 构建数组DP，长度为Size，初始值为0；构建数组Trace，长度为Goods.length，初始Trace[i]=i。
2. for good in Goods:
   1. for i = Size to good.size:
      1. if DP[i – good.size] + good.size > DP[i]:
         1. DP[i] = DP[i – good.size] + good.size
         2. Trace[i] = i – good.size
3. 构建能放下的节点集合CanSet。j = Size
4. While Trace[j] != j:
   1. CanSet.add(j-Trace[j])
   2. j = Trace[j]
5. return CanSet

*ArrangeKeyNode(KeyNode, QCertain, QUncertain, Trace):*

1. KeyNode放置其所在核能放置的最前位置。
2. 确定前和后驱点的选择和位置，从而确定入度边和出度边的内存消耗长度，获取入度边中内存消耗最大长度LIn和出度边中内存消耗最大长度LOut。
3. 查询区间[KeyNode.StartTime – L, KeyNode.StartTime]内的内存峰值HIn和区间[KeyNode.EndTime, KeyNode.EndTime + LOut]内的内存峰值HOut。
4. CanSetIn = BinPacking(KeyNode.InEdgeSet, MaxCache - HIn)
5. CanSetOut = BinPacking(KeyNode.OutEdgeSet, MaxCache - HOut)
6. 将KeyNode放入QCertain，将CanSetIn、CanSetOut中的节点均放入QCertain。
7. 将KeyNode.InEdgeSet – CanSetIn和KeyNode.OutEdgeSet – CanSetOut中的节点加入QUncertain。
8. Trace.add(KeyNode)

*SpreadFromKeyNode(KeyNodeSet, UnCheckedNodeSet):*

1. 构建队列QUncertain，QCertain。构造数组Trace。
2. for KeyNode in KeyNodeSet:
   1. ArrangeKeyNode(KeyNode, QCertain, QUncertain, Trace)
   2. UnCheckedNodeSet.remove(KeyNode.InEdgeNodeSet)
   3. UnCheckedNodeSet.remove(KeyNode.OutEdgeNodeSet)
3. While UnCheckedNodeSet.empty() == false:
   1. While QCertain.empty() == false:
      1. KeyNode = QCertain.pop()
      2. ArrangeKeyNode(KeyNode, QCertain, QUncertain, Trace)
   2. If UnCheckedNodeSet.empty() == false:
      1. KeyNode = UnCheckedNodeSet.pop()
      2. ArrangeKeyNode(KeyNode, QCertain, QUncertain, Trace)
4. return Trace and QUncertain
5. 依次利用贪心思想确定放入DRAM的节点集合中的节点位置。

对于存在不确定节点的同一个不同循环的节点X的集合，即仅包括不同循环的节点X，依次按照这些节点的开始时间，对前驱节点，优先给开始时间晚的节点安排结束时间晚的前驱节点；对后继节点，优先给开始时间早的节点安排开始时间早的后继节点。

Q：是否覆盖所有边？