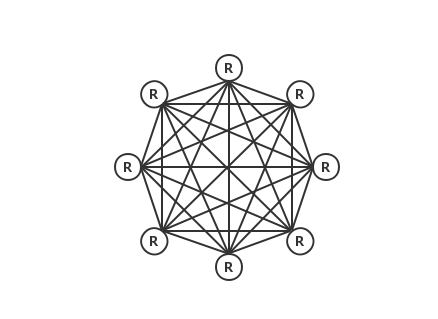
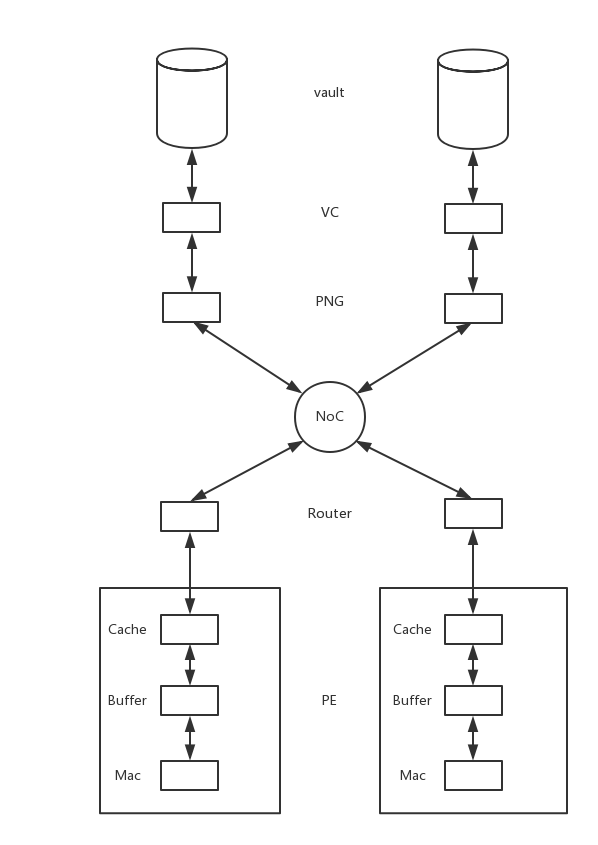
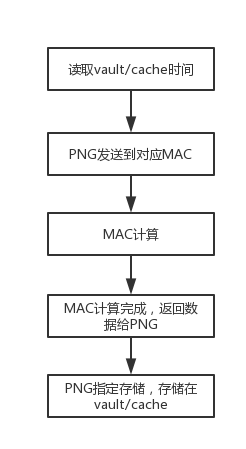
# 应用模型



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，，其中表示节点集合，节点个数是；表示边集合，边个数是；表示PE个数；表示图的重定时次数。

定义表示需要循环的次数。

## *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。

**Step One**：**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，定义，则需要次发射。其中若整除，

* 每次发射均需要的核数是；
* 每次发射最多需要次循环。

若不整除，

* 前次发射需要的核数是，最后一次发射核数是；
* 若设前次发射每次循环次，最后一次发射循环次。故满足，对所有的选取使得总时间最小的一组。

**Step Two：确定周期内图循环的组数和每个核任务种类。**

将所有的任务节点首先在不考虑数据依赖的条件下安排在各个PE的每个时间段。因为图中包含一些执行时间远大于其他的任务节点，若直接按照拓扑序列对所有任务节点进行排列后依次插入各个PE，最终消耗的总时间会很大，主要的原因是在于这些执行时间特别长的任务节点在它们所在的PE上运行时，其他的PE总是空闲状态，所以本算法考虑这些长执行时间的任务节点所带来的影响，采取贪心的策略对任务节点进行排列。这个问题抽象出来就是最小化最大值问题：把PE看在箱子，PE上所有运行的任务节点的执行时间之和为箱子的体积，任务节点看作物品，任务节点的执行时间可以看作物品的体积，那么需要将这些物品放入箱子之中，最终使得这些箱子中体积最大的箱子的体积尽量小。解决方案是对所有的物品按照体积大小从大到小排序，每次将物品放在当前所有箱子中体积最小的箱子里面，直至所有物品都放入箱子中。

每个周期内并不一定只循环一轮图，所以同时也需要确定每个周期内图的循环次数。定义一个周期内所有PE的利用率为，一个周期内所有PE的最低利用率为，同时定义一个周期内图的循环次数为。那么的值时满足的最小循环次数。但为了避免循环次数过大，定义最大循环次数为，若在次循环内，均不能被满足，则等于在次循环内，最大的循环次数。图

为了提高算法效率，每次都在前一次循环中任务节点安排的结果上直接排列下一次循环的任务节点。当得到后，再对当前所有PE上的任务按照贪心策略重新摆放。

伪代码：

*GenerateArrange():*



**Step Three：判断是否需要RETIMING。**

**Step Four：寻找“关键”节点集合（按照关键性排序）。**

定义关键节点集合，参数。对任务节点，若它的执行时间满足，则认为任务节点是关键节点，将它加入关键节点集合。是一棵二叉平衡排序树，按照任务节点的时间大小，从大到小进行排序。

伪代码：

*GetKeyNodeySet():*

**Step Five：确定关键节点的位置，通过其入度（出度）边，利用动态规划确定入度（出度）边上传输的数据存储的位置（cache/DRAM），同时确定边所连接的节点的位置。**

对于“关键”节点集合中的任务节点，根据它所在的核上已经排列的情况，将它放在能放置的位置的最前面。如下图

对的每一条入度边即从到有一条边，从向传输的数据需要存储在核为的PE上的Local Cache中。因为Local Cache的容量有限，所以如果所有的入度边不能全部放入Local Cache中，那么就需要用动态规划的方法选出一些任务节点传输的数据放入Local Cache，剩下的则放入DRAM中。对于传输数据已经放入Local Cache的任务节点，它的位置只需要离间隔数据在Local Cache上传输的时间即可。

对的出度边即从到有一条边，仅考虑的任务节点。对满足条件的，从向传输的数据也需要存储在核为的PE上的Local Cache中。类似入度边时所做的处理，考虑到Local Cache的容量有限，对传输的数据同样需要用动态规划的方法来挑选一些数据放入Local Cache，而剩下的放入DRAM。

对任务节点的条入度（出度）边上所需要传输的数据将他们放入容量为的Local Cache中，采用动态规划的策略来解决。算法导论

伪代码：

*ArrangeInCache():*



对于关键节点，依次从前往后核为的PE上的个空闲区间，对第个空闲区间，是否满足。若满足，则任务节点的开始时间。分别考虑任务节点的入度边集合和出度边集合。

伪代码：

*ArrangeKeyNode():*

1. 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。
2. 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。
3. CanSetIn = BinPacking(KeyNode.InEdgeSet, MaxCache - HIn)
4. CanSetOut = BinPacking(KeyNode.OutEdgeSet, MaxCache - HOut)
5. 将KeyNode放入QCertain，将CanSetIn、CanSetOut中的节点均放入QCertain。
6. 将KeyNode.InEdgeSet – CanSetIn和KeyNode.OutEdgeSet – CanSetOut中的节点加入QUncertain。
7. 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。