# Abstract

随着图像处理和自然语言处理等领域的兴起，深度卷积神经网络在这些领域之中获得了巨大的成功。虽然深度卷积神经网络相比于其他方法具有更好的性能，但不容忽视的是它的高性能所带来的昂贵的代价——大量的数据移动。PIM架构的提出有效的解决了内存系统中的瓶颈（addressing the bottleneck in the memory subsystem）——低效的内存系统的利用，结合3D堆叠技术（3D-stacked），一些形如Neurocube的可扩展的、可编程的、计算性能高的新型架构逐渐被提出。如何利用卷积神经网络的高度并行性和PIM架构来提高神经网络的运行效率成为了一个热门的问题。

本文提出*Fast-Para-Conv*和*Prior-Conv*，都是基于PIM架构的并行任务级确定性的调度算法。

# Introduction

（深度卷积神经网络的火热->CNN的开销）

深度卷积神经网络已经表现出了十分优异的性能，并且已经普遍存在于机器学习和人工智能相关的应用之中，例如图像识别、自然语言处理等。[2-4, 6-10]尽管这些的应用有着不同的强大的各种功能，但都把计算效率作为首要的关注点。这些的CNNs在一层的计算中需要成百上千兆的内存来存储参数，同时还需要数以万计的数字计算操作。CNN模型中卷积占据了超过90%的计算任务，决定了主要的运行时间。[2, 7]这些数据在内存中的移动速度及计算速度决定了卷积神经网络的计算效率，也成为了卷积神经网络计算性能的主要瓶颈。

（PIM架构等）

（本文算法）

（文章组织结构）

# Background

（Neurocube？CNN？多发射？样例？）

## 卷积神经网络

卷积神经网络相比传统的机器学习算法在图像处理、自然语言处理等其他领域已经展现出了巨大的优势，它在处理这些问题时具有更高的精度。常见的经典卷积神经网络由卷积层（CONV）、池化层（POOL）和全连接层（FC）共四层组成。

### 卷积层

在卷积层中，对输入的多维数据即**张量**从中提取出代表着输入数据特点的特征映射（feature maps），用一个低维度的**矢量**表示。以*二维图像的提取的过程*为例，通过一个可以滑动的窗口即一个的二维**核**（kernel），对一个窗口内的张量与二维的核进行卷积运算，得到特征映射。

其中是激活函数（ReLU），通常采用非线性函数作为激活函数，用于把结果限制在一个合理的范围之内；是偏置的共享值。

### 全连接层

在全连接层中，所有输入层和输出层的神经元都*如下图*一样全连接着另一个。把一个神经元的所有输入与权重矩阵相乘得到输出，即：

### 池化层

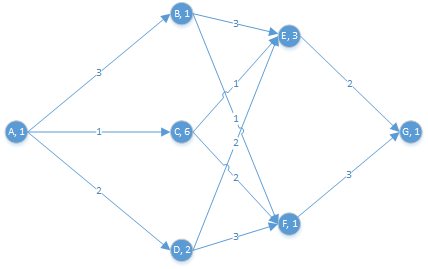
CNN中的池化层使用池化函数（pooling function）来进一步的调整这一层的输出。池化函数通常使用某一位置的相邻输出的总体统计特征来代替网络在该位置的输出，常见的池化函数有最大池化（max pooling）函数，即这个位置的相邻输出的最大值作为输出。

## 基于3D-PIM架构

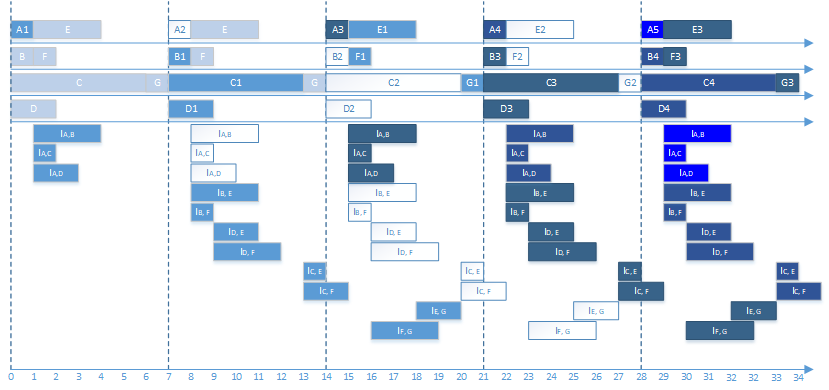
PIM架构作为当下最流行的一种利用3D堆叠技术融合了多个DRAM层和一个逻辑控制层为内存效率的瓶颈提供了一种很好的解决方案。图中展示了一种高效率低能耗的PIM架构——Neurocube。这种架构整合了高度并行的HMC，以内存为中心，利用数据驱动的性质和神经启发算法实现了可编程的内存系统来驱动数据流动（the data-driven nature and neuro-inspired algorithms to implement a programmable memory system to drive data flow）。Neurocube可以分为存储层和逻辑层两个层次。存储层将多个DRAM利用3D堆叠（3D-stacking）技术重叠在一起，不同的DRAM之间利用高速传输介质（through-silicon via-TSV）连接在一起。这些DRAM被划分成多个vault，每个vault在功能和操作上是相互独立的。在逻辑层中，每个vault由一个vault控制器（VC）控制，它控制vault中内存的引用操作（manage all memory reference operations within that vault）。每个VC都与一个可编程的神经序列生成器（programmable neurosequence generator）PNG相连，PNG通过2D网格网络（2D mesh network）与其中的路由和计算单元通信。PNG负责控制神经计算中所需要的数据流动，生成在前一层网络中神经计算在DRAM中的地址和权重，并封装成一个数据包发送到指定计算单元中的乘法累加器中，其中数据包包含了MAC-ID，source ID——Vault ID，destination ID——PE ID。PE是这个架构中主要的计算单元，包含许多乘法累加器、一个cache，两个buffer。MAC负责进行计算，两个buffer分别用来接受收到的数据包和权重。对小的网络来说，权重矩阵不大时，直接存储在PE中的buffer中，但对节点多的网络来说，权重矩阵较大，PE中的buffer存不下，所以需要将权重矩阵拆分后，存储在DRAM之中。cache用于存储不是当前轮次操作的数据包，直到操作计数器发生变化时，再将其中存储的数据包放入buffer中。

## 样例

## 样例1

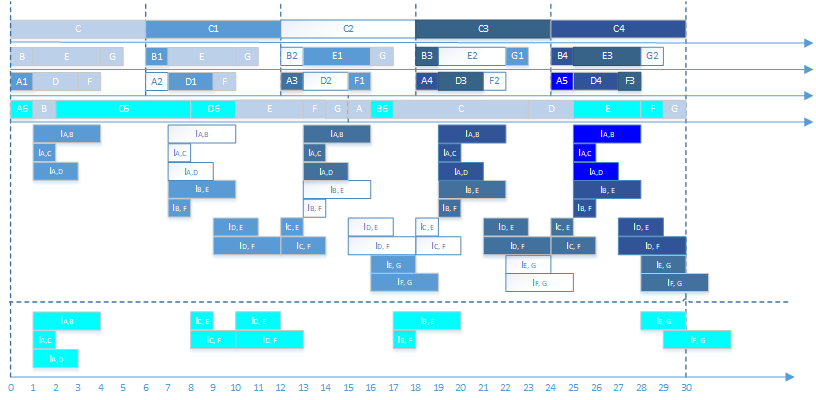


因为考虑CNN中卷积层有许多节点的运行时间很长，故设计出的样例图G如上，其中节点中的数字表示节点的运行时间，边上的数字表示边所需要消耗的时间。G图中C节点的运行时间远大于其他的节点的运行时间。



假设共有4个PE，上图为一种任务调度的策略，称之为*Para-Conv*。它首先将任务按照ID顺序依次摆放，即得到了第一个周期的摆放，每个虚线分割的为一个周期。因为如此摆放时并没有考虑图中各个任务的依赖关系，故接下来对任务进行重定时。在第一个周期内A没有前序任务，故可以直接执行任务A。任务A执行完成后，数据传输到任务B、C、D需要等待一定的时间即，所以需要等到第二个周期才能运行任务B、C、D。同理可得，由于任务C的时间很长，所以他直接决定了任务E、F最早开始时间。第二个周期的任务E和任务F均不满足条件，所以直到第三个周期才能运行。第一轮的最后一个任务G在第四个周期运行。故运行完一轮耗时26单位时间。

前四个周期被称为前序，从第五个周期开始，每个周期内按照初始的排列的任务都会被得到执行，但不一定在同一个轮次内执行。这种方法在一个图的执行次数很多的时候，能达到很高的效率。但它也有一定的缺陷，初始的排列对总时间的消耗会产生很大影响。如下例，

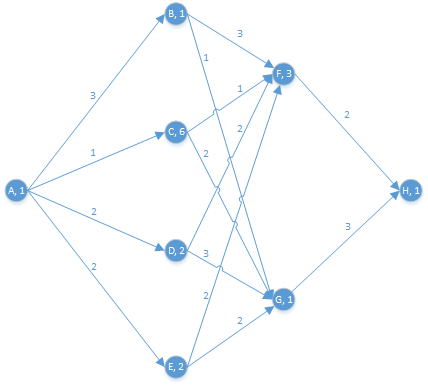


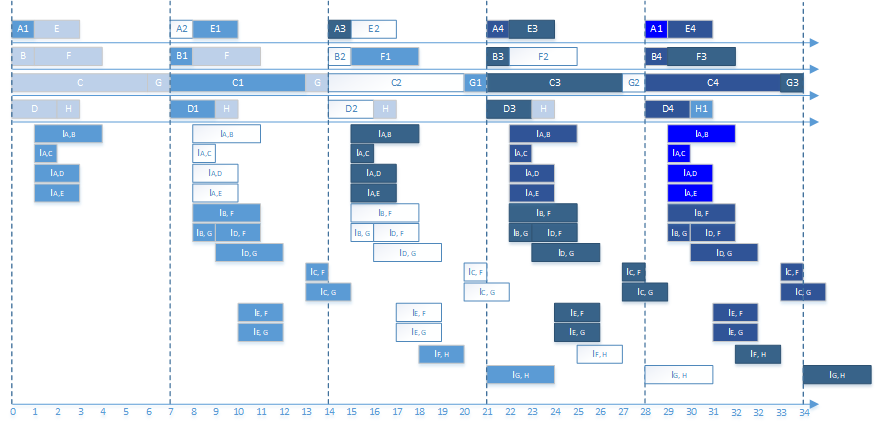
在前一种做法中，我们发现一个周期内的PE利用率不是很高，从而导致了整体的CPU利用率低，时间较长。故本文提出*Fast-Para-Conv*对初始的任务排列进行了重新的安排，尽可能得使一次周期内每个PE的任务时间总和都接近平均值。如上图所示，考虑到任务C的时间十分的长，故将任务C单独放在一个PE上，其他任务相互组合，且尽可能平均，最后每个PE上的安排的任务按照拓扑序进行排序。同时，*Fast-Para-Conv*结合了多发射技术，充分利用了每一个PE。*Fast-Para-Conv*中每次发射所需要的PE数量为图G的最大并发度，如若PE个数不够时，依据拥有的PE个数，自适应的调度安排策略。如上图中，图G的最大并发度是3，故第一次发射需要3个PE，第二次发射时只剩下1个PE，故重新调整了安排策略。通过*Fast-Para-Conv*的调度PE的利用率得到大幅提升，且时间也相对缩短了很多，尤其在对拥有长时间计算的节点的图有着不错的表现。



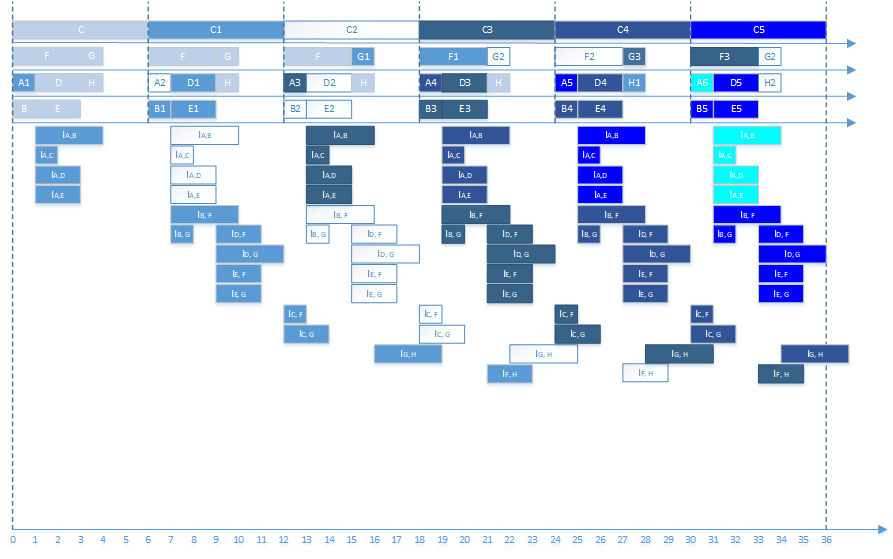
考虑到*Fast-Para-Conv*和*Para-Conv*不论如何都需要一个短暂的前序过程，而且在前序的过程中PE的利用率从一个很低的值开始慢慢增长。为了避免前序过长所导致的总时间长和PE利用率低，本文提出了另外一种算法——*Prior-Conv*，它并没有采用重定时的技术，而是每次都让下一个任务在先结束的PE上运行。在初始的时候，一起把所有轮数的拓扑序第一任务一起执行，这样可以充分利用靠前轮数的任务执行完成后的数据传输的时间，在这个时间直接执行后面轮数的任务。*Prior-Conv*中最关键的是下一个任务的选取。上图展示的策略是优先选取拓扑序高的且运行时间短的，选取拓扑序高的是为了不打乱数据的依赖关系。

## 样例2

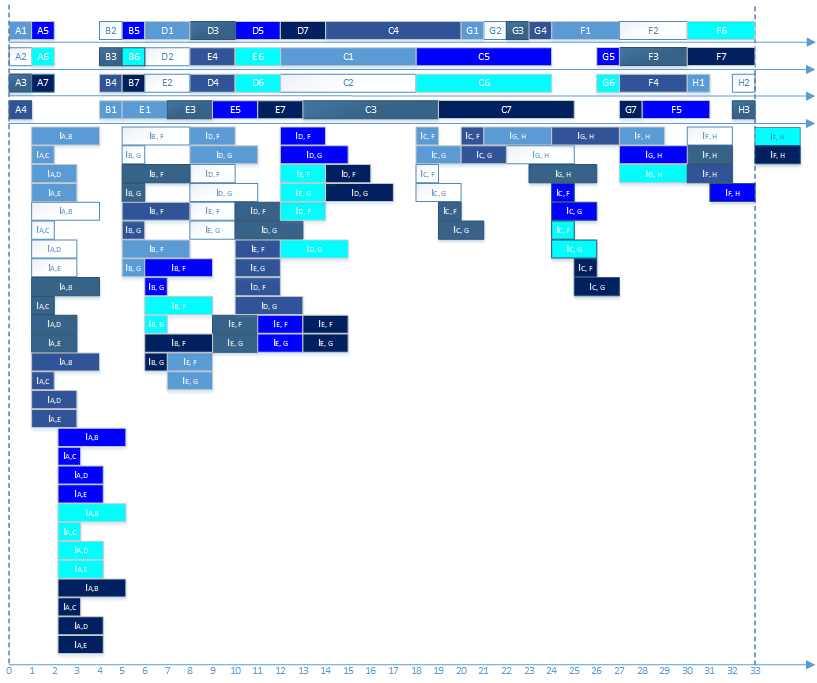




上图为*Para-Conv*在第二个图上的调度。



上图为*Fast-Para-Conv*在第二个图上的调度



上图为*Prior-Conv*在第二个图上的调度

综合上述三种策略，*Fast-Para-Conv*的相比于*Para-Conv*提高了PE的利用率，并且结合多发射，总时间的消耗更短。*Prior-Conv*具有高度的并行性，避免了重定时技术前序中所浪费的时间，而是直接让后面轮数的无依赖任务先执行，从而提高了PE的利用率，缩短了运行时间。*Fast-Para-Conv*和*Prior-Conv*比较，*Fast-Para-Conv*受限于一个周期的图循环轮数，*Prior-Conv*受限于下一个任务的选取，总的来看，*Prior-Conv*的性能更优，速度更快，PE的利用率越高。

# 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。

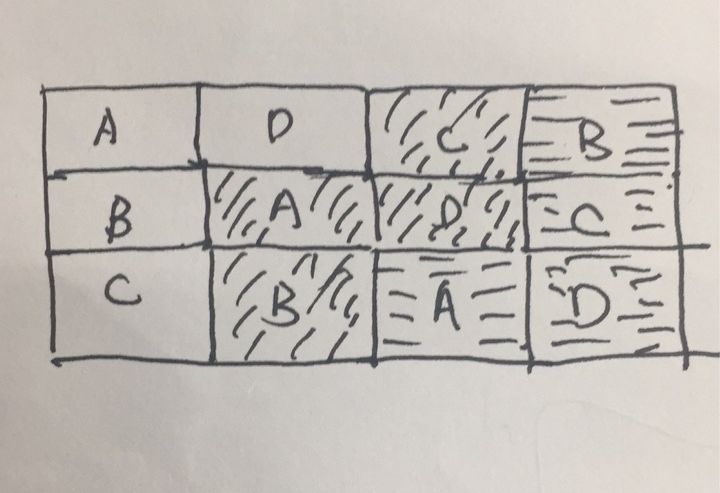
定义T表示G需要循环的次数。

## *Fast-Para-Conv*

在介绍*Fast-Para-Conv*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

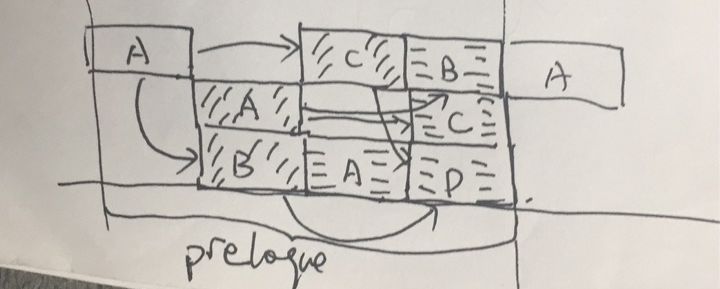
将所有的任务节点不考虑数据依赖的条件下按照一定的规则安排在各个PE的每个时间段，为了达到局部的最优解，每个PE的每个时间段前后的任务之间没有时间等待。最简单的排列规则是直接按照顺序依次安排，如下图所示，横坐标表示时间线，纵坐标表示PE：



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。

*Fast-Para-Conv*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。

*Fast-Para-Conv*的实现可以很多的方法，故本文只介绍*Fast-Para-Conv*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 Vj in {Topology of G}：
7. for e in Edge[Vj]：
8. Vk为Vj经过边e所到达的在安排的节点。
9. 获取Vk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Vk的开始时间为min(Vj.endtime + e.cost, Vk.starttime)。

虽然从伪代码上, *Fast-Para-Conv*的代码量并不大，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，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。

## *Prior-Conv*

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，每个节点在等待队列中等待的时间不会超过，N表示与拓扑序列相同的节点个数。

*Prior-Conv*的具体执行步骤如下（以下一个节点的选取为拓扑序列高且耗时较短优先为例）：

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，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，以节点结束时间早为优先级。构造一个节点等待队列，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. 如果不为空：从取出空闲PE的peid，构建运行节点，赋值其运行的PE为peid，将其丢入。
6. 如果为空： 构建等待节点，将其丢入。
7. while 不为空：
8. 丢出队首节点，将队首节点的peid丢入。
9. 获取节点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。的轮次设置与相同，并设置其开始时间为，将其丢入。
10. while 不为空：
11. 如果为空，break。
12. 取出队首节点，取出队首peid，构建运行节点，丢入。

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，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。

## 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。

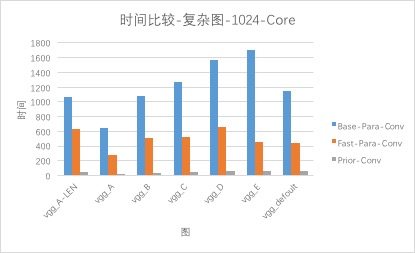
# Experiment

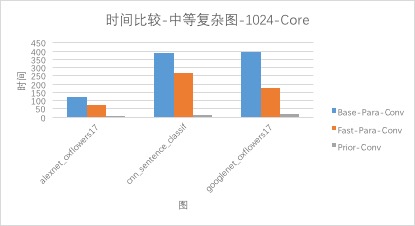
实验数据如下：

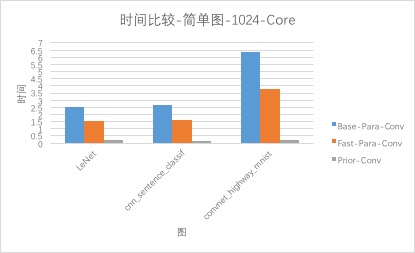


## 运行时间比较

所有图均迭代500次。



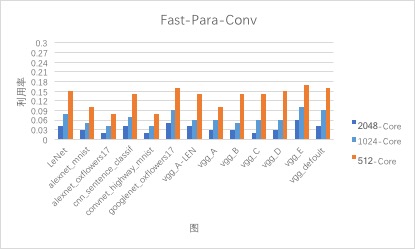




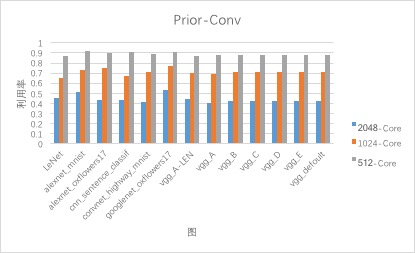
从上面3个图可以明显的看出，不管是复杂图，中等复杂图还是简单图，*Fast-Para-Conv*和*Prior-Conv*都明显优于*Para-Conv*，其中*Fast-Para-Conv*的速度基本是*Para-Conv*的1.5-2倍左右，*Prior-Conv*的速度基本是*Para-Conv*的20倍左右。

## 利用率比较

*Fast-Para-Conv*



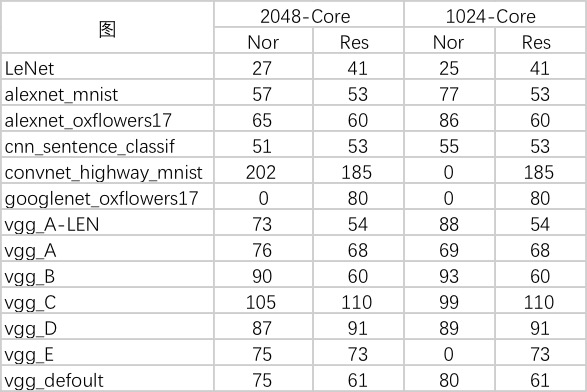
*Prior-Conv*



从上面两个图可以明显看出，随着PE的个数增加，PE的利用率都是在逐渐降低。但总体来说，*Prior-Conv*的PE利用率都远大于*Fast-Para-Conv*的利用率，这也是*Prior-Conv*的速度快的原因之一。

## Retiming 次数比较

因为*Prior-Conv*不需要Retiming，故此处只分析*Fast-Para-Conv*的retiming次数。



上表中，因为*Fast-Para-Conv*利用了多发射技术，其中可以分为两类发射Nor和Res，Nor表示以图最大并发度为PE个数的发射，Res表示PE个数不够图最大并发的发射。