北京大学
PEKING UNIVERSITY

人工智能处器理器设计

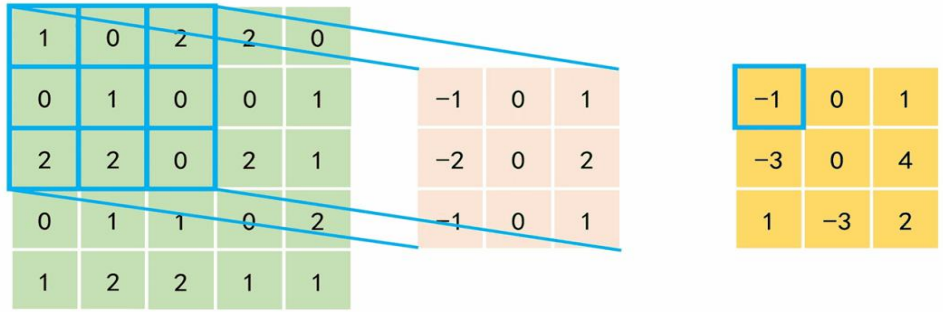
2019.8.18

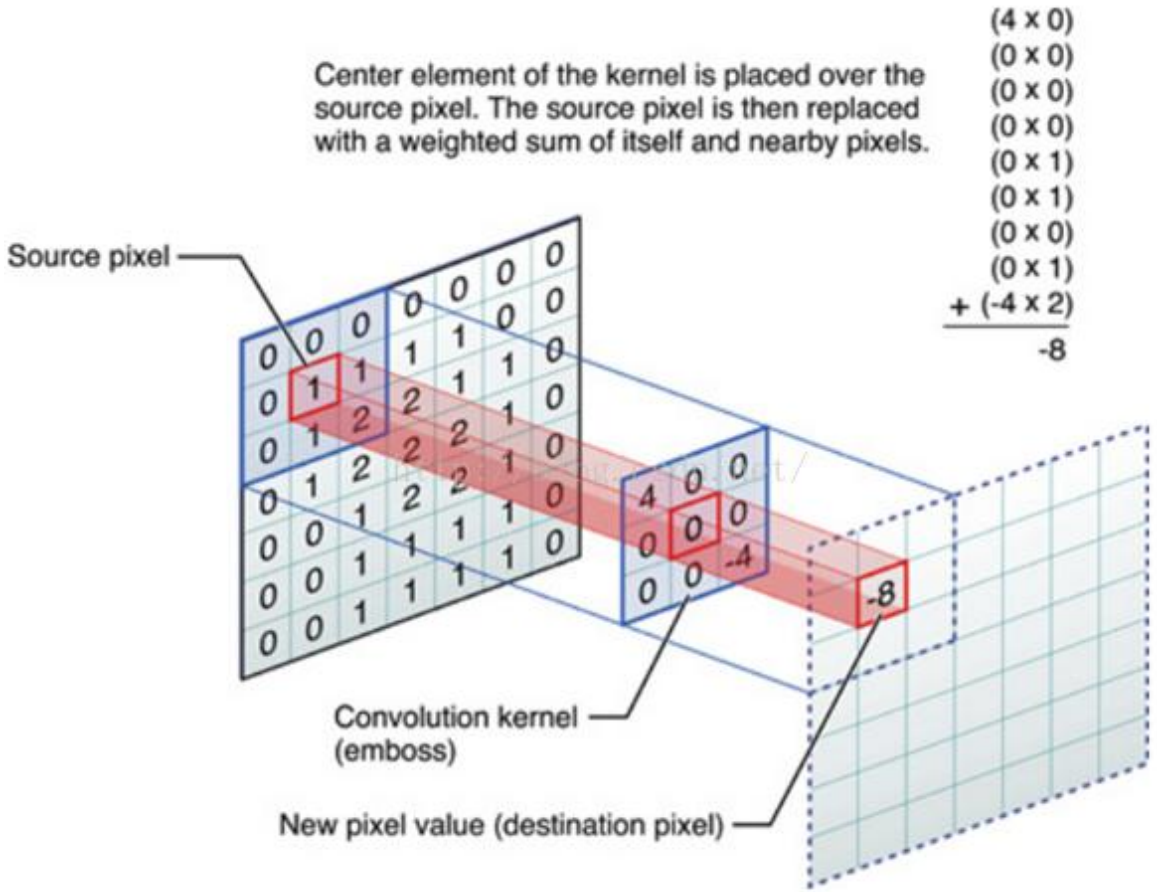
目录

- 卷积层
- 卷积层的代码
- 卷积层电路实现（TPU）
- 有代表性芯片
- 采用超导材料

卷积层

原输入图（source pixel）的多个点在卷积核的映射得到输出图的一个点，输入图和输出图之间称为卷积层。输入图和输出图之间通过卷积核和权重链接。

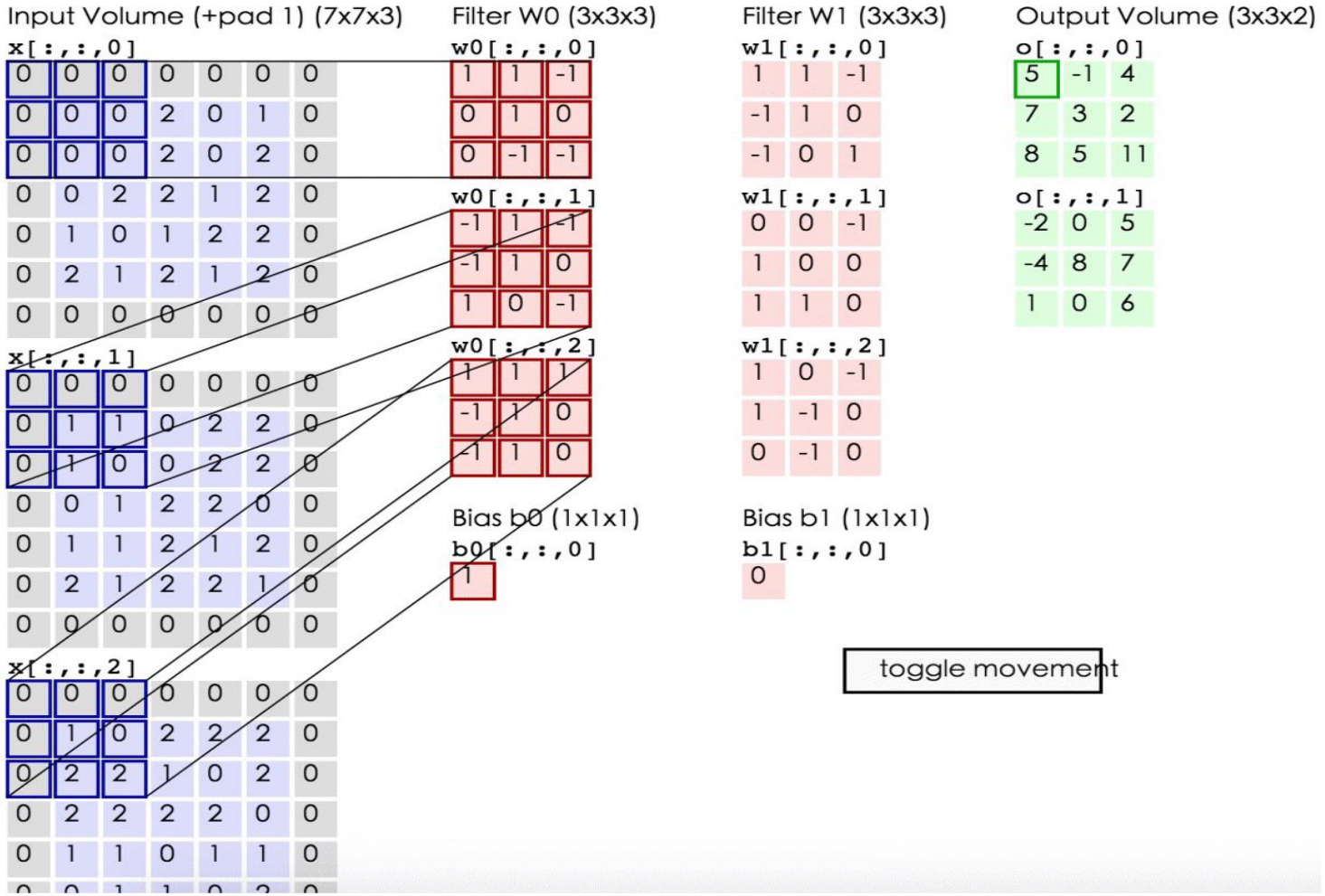




有填充（p=2）

步长为2

3通道并行



卷积层的代码

北京大学

PEKING UNIVERSITY

卷积核 conv1

对变量进行识别，所以通过define定义变量值。

```
#include "global.h"

#define K_SIZE 3
#define PAD 1
#define IC 3
#define IH 224
#define ID 224

void conv1(My_float *weight, My_float input[IC][IH + 2 * PAD][ID + 2 * PAD], My_float output[IH ][ID])
{
    int32_t  i, j, q, k1, k2;
    My_float sum = 0.0;
    int32_t offset_w = 0;

    for (i = 0; i < IH + 2 * PAD; i++)
    {
        for (j = 0; j < ID + 2 * PAD; j++)
        {
            sum = 0.0;
            offset_w = 0;
            for (q = 0; q < IC; q++)
            {
                for (k1 = 0; k1 < K_SIZE; k1++)
                {
                    for (k2 = 0; k2 < K_SIZE; k2++)
                    {
                    sum = sum + input[q][(i + k1)][(j + k2)] * weight[offset_w];
                    offset_w = offset_w + 1;
                    }
                }
            }
            output[i][j] = sum;

        }
    }

}
```

卷积层的HLS代码

根据caffemodel中定义的值来进行定义变量大小

- pad: 1， padding填充为1。
- kernel_size: 3， 卷积核的大小为3。
- IC：3，输入图像的层数为3，例如一般图像可分为RGB 3层。
- crop_size: 224，对图像进行裁剪，长宽值均为224。
- num_output: 64，此层输出图像的个数

```
#include "global.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PAD 1
#define K_SIZE 3
#define IC 3
#define IH 224
#define ID 224
#define OC 64

void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
    My_float output[OC][IH][ID], My_float bias[OC])

{
    unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };

    int32_t m, i, j, k;
    for(m = 0; m < IC; m++)
    {
        for (i = 0; i < IH ; i++)
        {
            for (j = 0; j < ID ; j++)
            {
                windows[m][i+PAD][j+PAD]= input[m][i][j];
            }
        }
    }
    My_float output_buf[OC][IH][ID];
    static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
    int offset_3x3 = 0;
    memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
    for (i = 0; i < OC; i++)
    {
        conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
        offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
    }

    for (i = 0; i < OC; i++)
    {
        for (j = 0; j < IH; j++)
        {
            for (k = 0; k < ID; k++)
            {
                output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
            }
        }
    }
}
```

卷积层的代码

- 调用此函数需要传入四个变量，变量类型分别为My_float、char。
- Weight——卷积核内具体数值。
- input[IC][IH][ID]——输入原始图像（未padding）。
- output[OC] [IH][ID]——输出经卷积运算的图像。
- Bias——偏置系数。

```
1   #include "global.h"
2   #include <string.h>
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   #define PAD 1
7   #define K_SIZE 3
8   #define IC 3
9   #define IH 224
10  #define ID 224
11  #define OC 64
12
13  void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
14      My_float output[OC][IH][ID], My_float bias[OC])
15
16  {
17      unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };
18
19      int32_t m, i, j, k;
20      for(m = 0; m < IC; m++)
21      {
22          for (i = 0; i < IH ; i++)
23          {
24              for (j = 0; j < ID ; j++)
25              {
26                  windows[m][i+PAD][j+PAD]= input[m][i][j];
27              }
28          }
29      }
30      My_float output_buf[OC][IH][ID];
31      static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
32      int offset_3x3 = 0;
33      memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
34      for (i = 0; i < OC; i++)
35      {
36          conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
37          offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
38      }
39
40      for (i = 0; i < OC; i++)
41      {
42          for (j = 0; j < IH; j++)
43          {
44              for (k = 0; k < ID; k++)
45              {
46                  output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
47              }
48          }
49      }
50  }
```

卷积层的代码

- 定义 char 类型的三维数组windows变量。
- 将输入的原始图像经过pad=1后，赋值给

  windows数组，共卷积函数conv1使用。

```
#include "global.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PAD 1
#define K_SIZE 3
#define IC 3
#define IH 224
#define ID 224
#define OC 64

void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
    My_float output[OC][IH][ID], My_float bias[OC])

{
    unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };

    int32_t m, i, j, k;
    for(m = 0; m < IC; m++)
    {
        for (i = 0; i < IH ; i++)
        {
            for (j = 0; j < ID ; j++)
            {
                windows[m][i+PAD][j+PAD]= input[m][i][j];
            }
        }
    }
    My_float output_buf[OC][IH][ID];
    static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
    int offset_3x3 = 0;
    memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
    for (i = 0; i < OC; i++)
    {
        conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
        offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
    }

    for (i = 0; i < OC; i++)
    {
        for (j = 0; j < IH; j++)
        {
            for (k = 0; k < ID; k++)
            {
                output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
            }
        }
    }
}
```

卷积层的代码

- 定义 My_float 类型的三维数组weight_but变量。
- 利用memcpy函数将卷积核weight内的数值复制给weight_buf。
- 复制的的字节数为： OC * K_SIZE * K_SIZE * IC * sizeof(My_float)

```
#include "global.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PAD 1
#define K_SIZE 3
#define IC 3
#define IH 224
#define ID 224
#define OC 64

void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
    My_float output[OC][IH][ID], My_float bias[OC])

{
    unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };

    int32_t m, i, j, k;
    for(m = 0; m < IC; m++)
    {
        for (i = 0; i < IH ; i++)
        {
            for (j = 0; j < ID ; j++)
            {
                windows[m][i+PAD][j+PAD]= input[m][i][j];
            }
        }
    }
    My_float output_buf[OC][IH][ID];
    static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
    int offset_3x3 = 0;
    memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
    for (i = 0; i < OC; i++)
    {
        conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
        offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
    }

    for (i = 0; i < OC; i++)
    {
        for (j = 0; j < IH; j++)
        {
            for (k = 0; k < ID; k++)
            {
                output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
            }
        }
    }
}
```

卷积层的代码

- 调用之前编写的卷积核conv1，进行运算。
- 此卷积层输出的64个图像分别进行卷积运算。

```
#include "global.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PAD 1
#define K_SIZE 3
#define IC 3
#define IH 224
#define ID 224
#define OC 64

void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
    My_float output[OC][IH][ID], My_float bias[OC])

{
    unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };

    int32_t m, i, j, k;
    for(m = 0; m < IC; m++)
    {
        for (i = 0; i < IH ; i++)
        {
            for (j = 0; j < ID ; j++)
            {
                windows[m][i+PAD][j+PAD]= input[m][i][j];
            }
        }
    }
    My_float output_buf[OC][IH][ID];
    static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
    int offset_3x3 = 0;
    memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
    for (i = 0; i < OC; i++)
    {
        conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
        offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
    }

    for (i = 0; i < OC; i++)
    {
        for (j = 0; j < IH; j++)
        {
            for (k = 0; k < ID; k++)
            {
                output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
            }
        }
    }
}
```

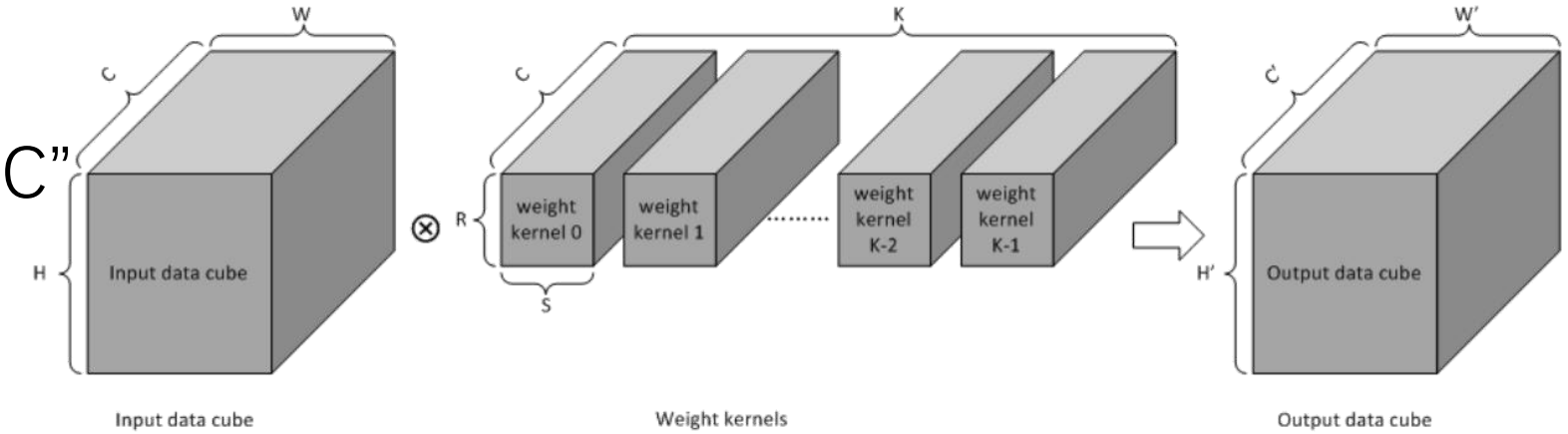
卷积层的代码

- 对每个输出图像加偏置系数bias。
- 加偏置系数过后，用激活函数Relu进行激活。
- 将激活后的值，赋值给output作为输出图像。

```
#include "global.h"
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define PAD 1
#define K_SIZE 3
#define IC 3
#define IH 224
#define ID 224
#define OC 64

void conv_1(My_float *weight, unsigned char  input[IC][IH][ID],
    My_float output[OC][IH][ID], My_float bias[OC])

{
    unsigned char  windows[IC][IH + 2 * PAD][ID + 2 * PAD] = { 0 };

    int32_t m, i, j, k;
    for(m = 0; m < IC; m++)
    {
        for (i = 0; i < IH ; i++)
        {
            for (j = 0; j < ID ; j++)
            {
                windows[m][i+PAD][j+PAD]= input[m][i][j];
            }
        }
    }
    My_float output_buf[OC][IH][ID];
    static My_float weight_buf[OC * K_SIZE * K_SIZE * IC];
    int offset_3x3 = 0;
    memcpy(weight_buf, weight, OC * K_SIZE * K_SIZE * IC * sizeof(My_float));
    for (i = 0; i < OC; i++)
    {
        conv1(&weight_buf[offset_3x3], &windows[IC][IH + 2 * PAD][ID + 2 * PAD], output_buf[i]);
        offset_3x3 = offset_3x3 + K_SIZE * K_SIZE * IC ;
    }

    for (i = 0; i < OC; i++)
    {
        for (j = 0; j < IH; j++)
        {
            for (k = 0; k < ID; k++)
            {
                output[i][j][k] = Relu(output_buf[i][j][k] + bias[i]);
            }
        }
    }
}
```

卷积操作

- 输入特征数据：W x H x C
- 权重数据：R x S x C
- 核心数：K
- 输出数据：W ' X H' X C”

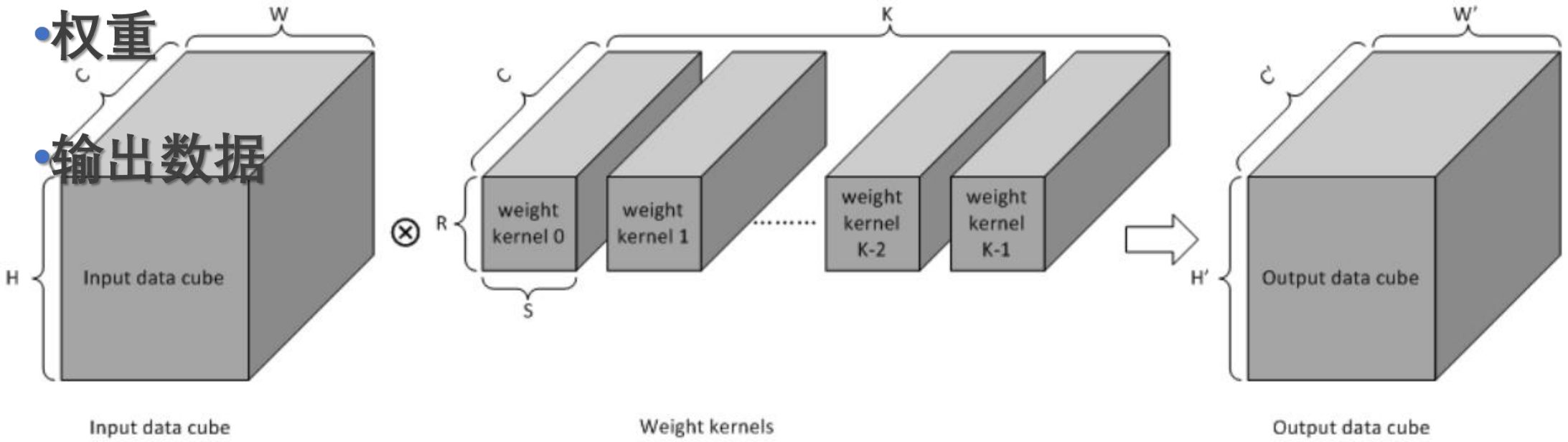


直接卷积

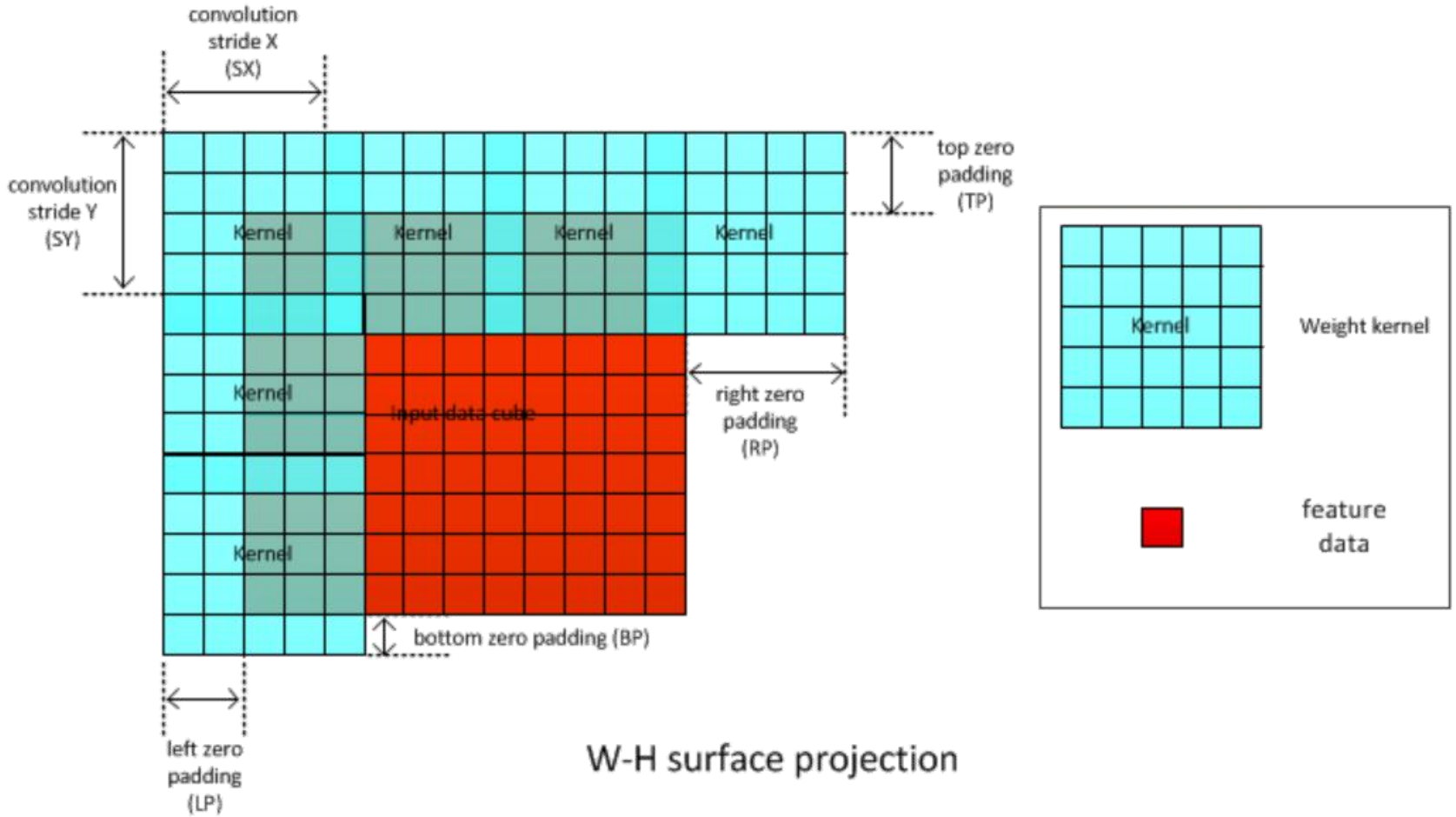
- 卷积流水线总是有两部分输入数据。
- 一个是输入激活数据，另一个是权重数据。

卷积操作

- **输入数据**
- **权重**
- **输出数据**

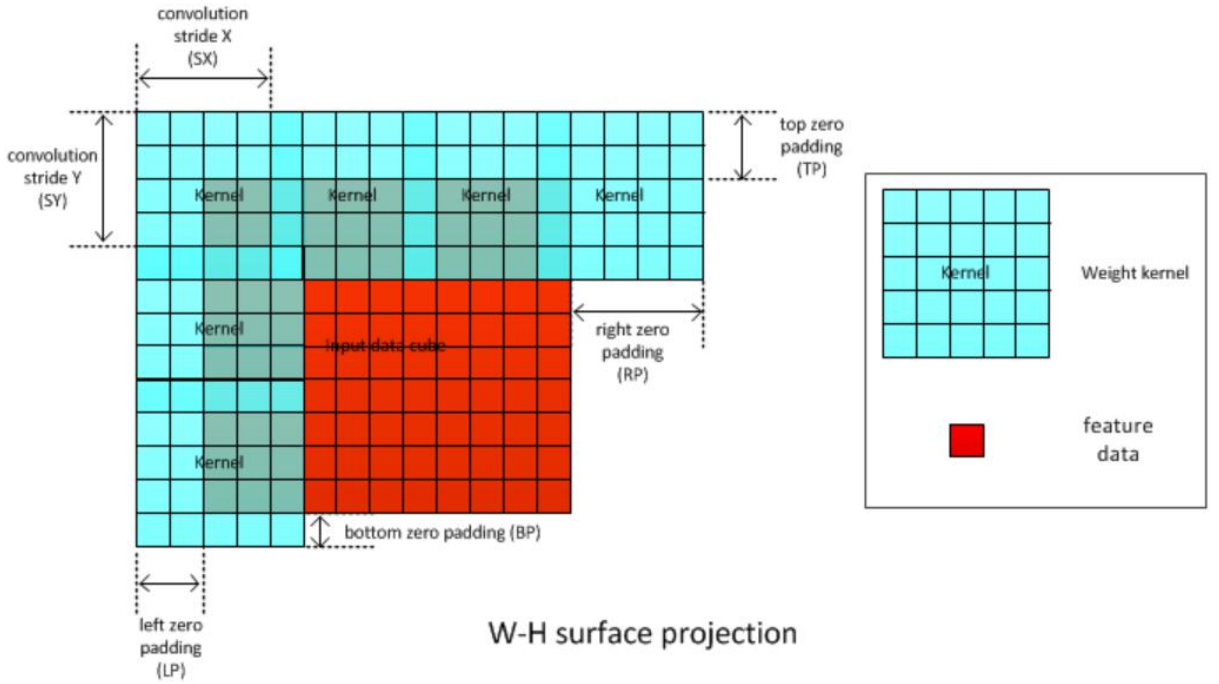


补零与步长


convolution
stride X
(SX)
convolution
stride Y
(SY)
top zero
padding
(TP)
Kernel
Kernel
Kernel
Kernel
Kernel
Kernel
Kernel
Input data cube
right zero
padding
(RP)
bottom zero padding (BP)
left zero
padding
(LP)
W-H surface projection
Kernel
Weight kernel
feature
data

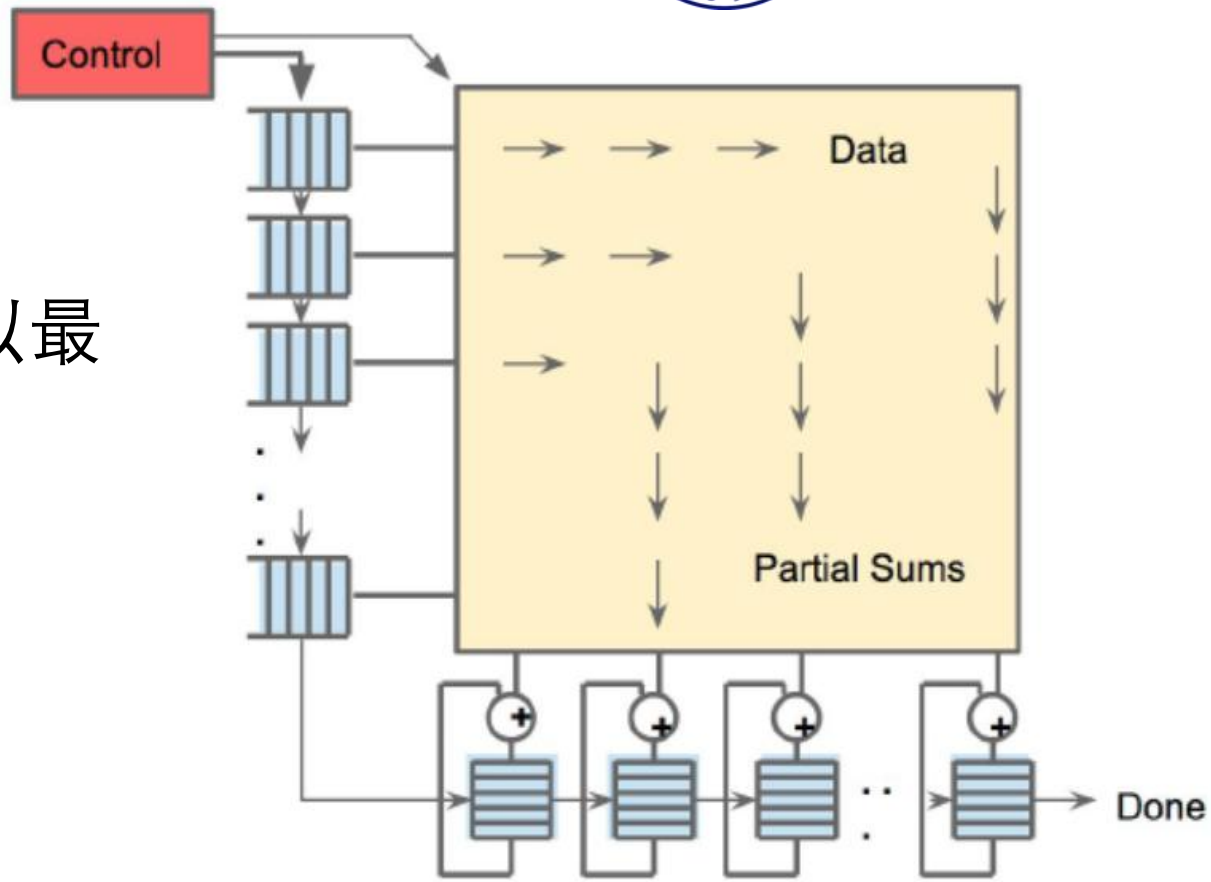
补零与步长

- 补零：左边界处的LP，右边界处的RP，顶部边界处的 TP，底部边界处的BP
- 卷积步长：X维度为SX，Y维度为SY



乘法矩阵（TPU）

乘法矩阵是一个脉动阵列，以最少的内存访问次数得出结果。



乘法矩阵

- 激活输入加载器(value loader)：从统一缓冲区接受激活输入，并将其发送至行阵列左侧的乘加单元，也可以将激活输入发送至相邻的激活输入加载器。
- 乘加单元(cell)：在一个时钟周期内，可以由激活输入和权值生成输出和，并将其沿权值传播方向发送至相邻乘加单元。
- 权值提取接口(weight fetcher interface)：从存储单元（例如动态存储器）接受权值输入，并将其发送至列阵列最顶部的乘加单元。

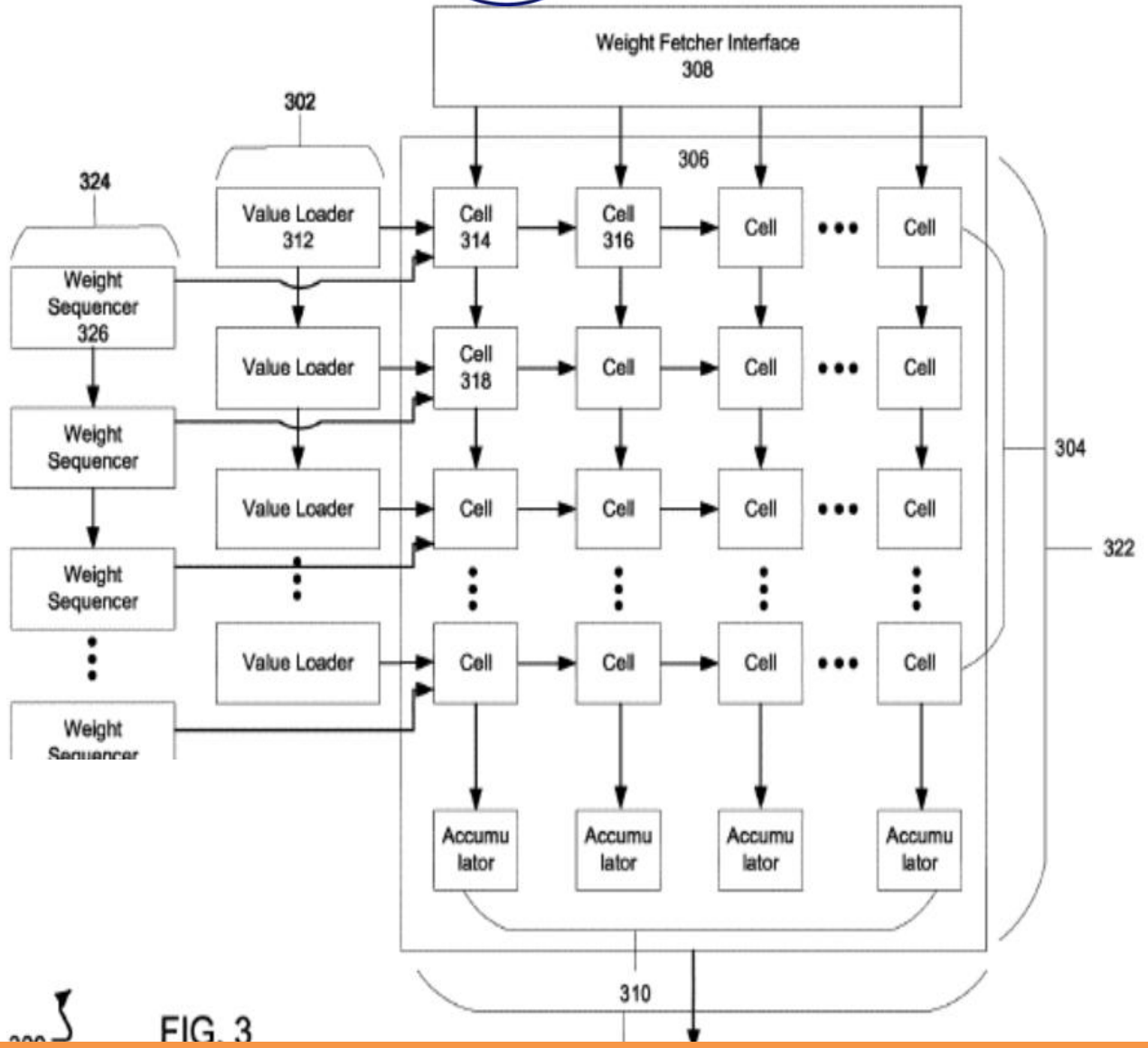
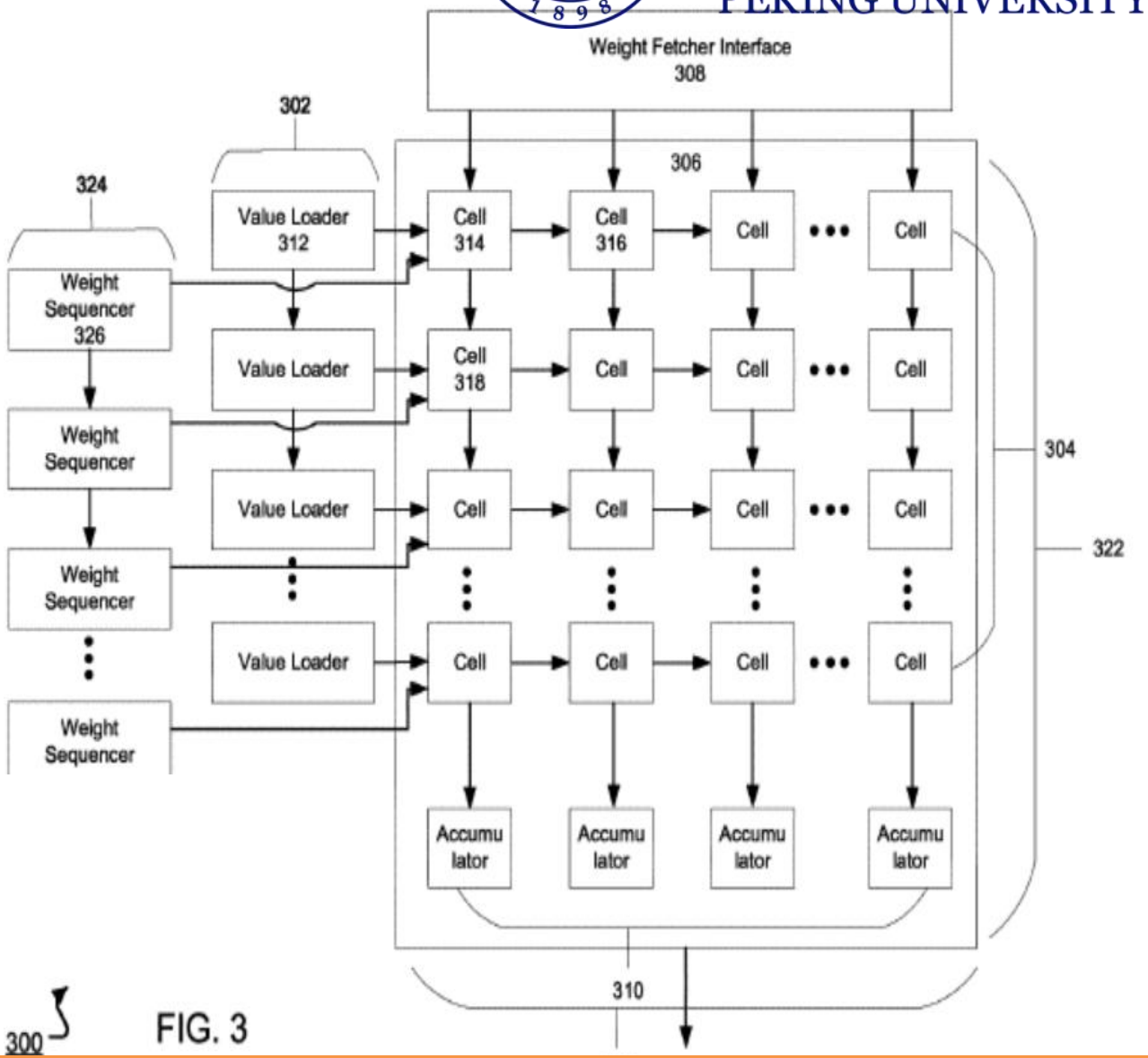


FIG. 3

乘法矩阵

- 累加单元(accumulator unit)。存储并累加乘加单元的输出和，并将其发送至向量计算单元。当并行输入小于行/列数时，可以不经过累加计算直接将乘加单元的结果发送至向量计算单元。
- 权值序列发生器(weight sequencer)。控制脉动阵列中的权值传输。



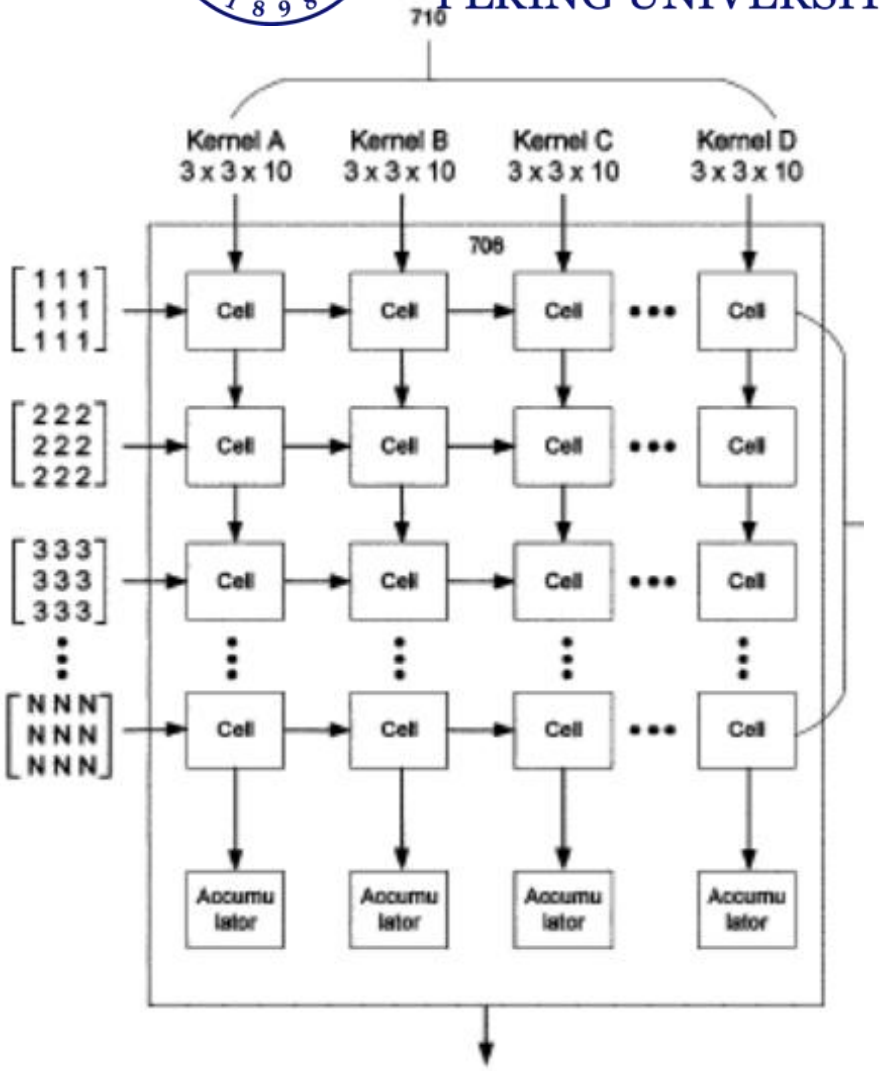
乘法矩阵

1.每次输入变换可以花费多个时钟周期，直到全部输入达到指定位置。

2.若并行输入的数量小于行/列数，可将输入复制并送至空的行/列，以便脉动矩阵同时进行多个卷积计算。

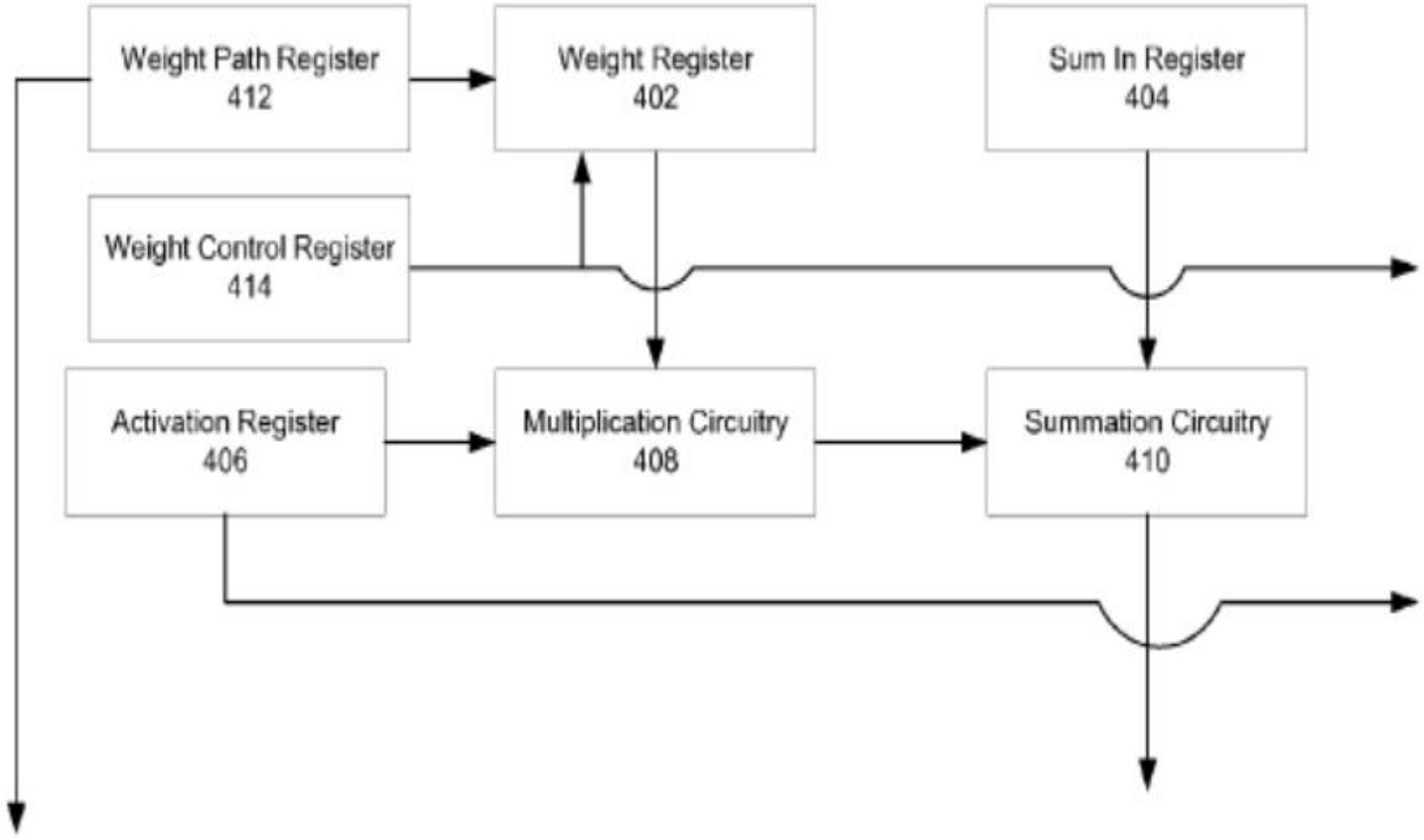
3.激活输入可以是一层，也可以是一层中的部分窗口集合。卷积核同理。

4.对每一个存储到脉动矩阵中的单元的输入，其行列地址可以存储在统一缓冲中。



乘加单元

- 权值寄存器(weight register)
- 和寄存器(sum in register)
- 激活输入寄存器(activation register)
- 乘法电路(multiplication circuitry)
- 求和电路(summation circuitry)：
- 权值路径寄存器(weight path register)：权值可以在多个时钟周期内不发生变化，可与多个激活输入进行计算。
- 权值控制寄存器(weight control register)：控制权值何时从权值路径寄存器发送至权值寄存器。



矩阵变换

1.矩阵划分：以3*3的卷积核为例，将激活矩阵划分为5*5的矩阵。

2.向量输入：将激活输入矩阵划分四个象限，对于每个象限根据从上到下、从左到右的顺序转化成向量输入。

3.旋转核的生成：将卷积核结构的元素沿X维或y维移动。

4.产生累计输出。

5.产生层输出。

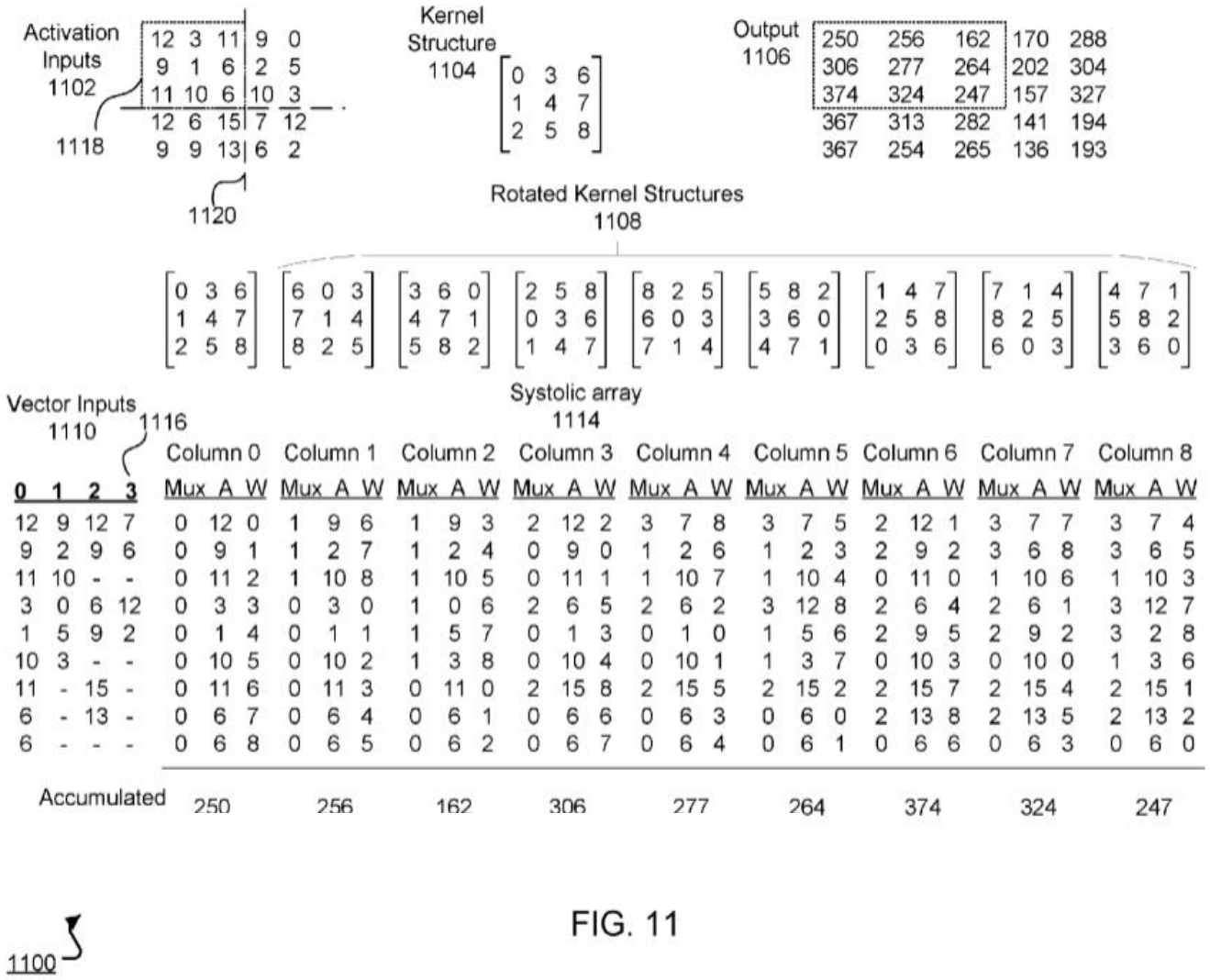


FIG. 11

矩阵变换

北京大学
PEKING UNIVERSITY

将任意步长的卷积网络变换为步长1*1的卷积网络，以减少控制电路。

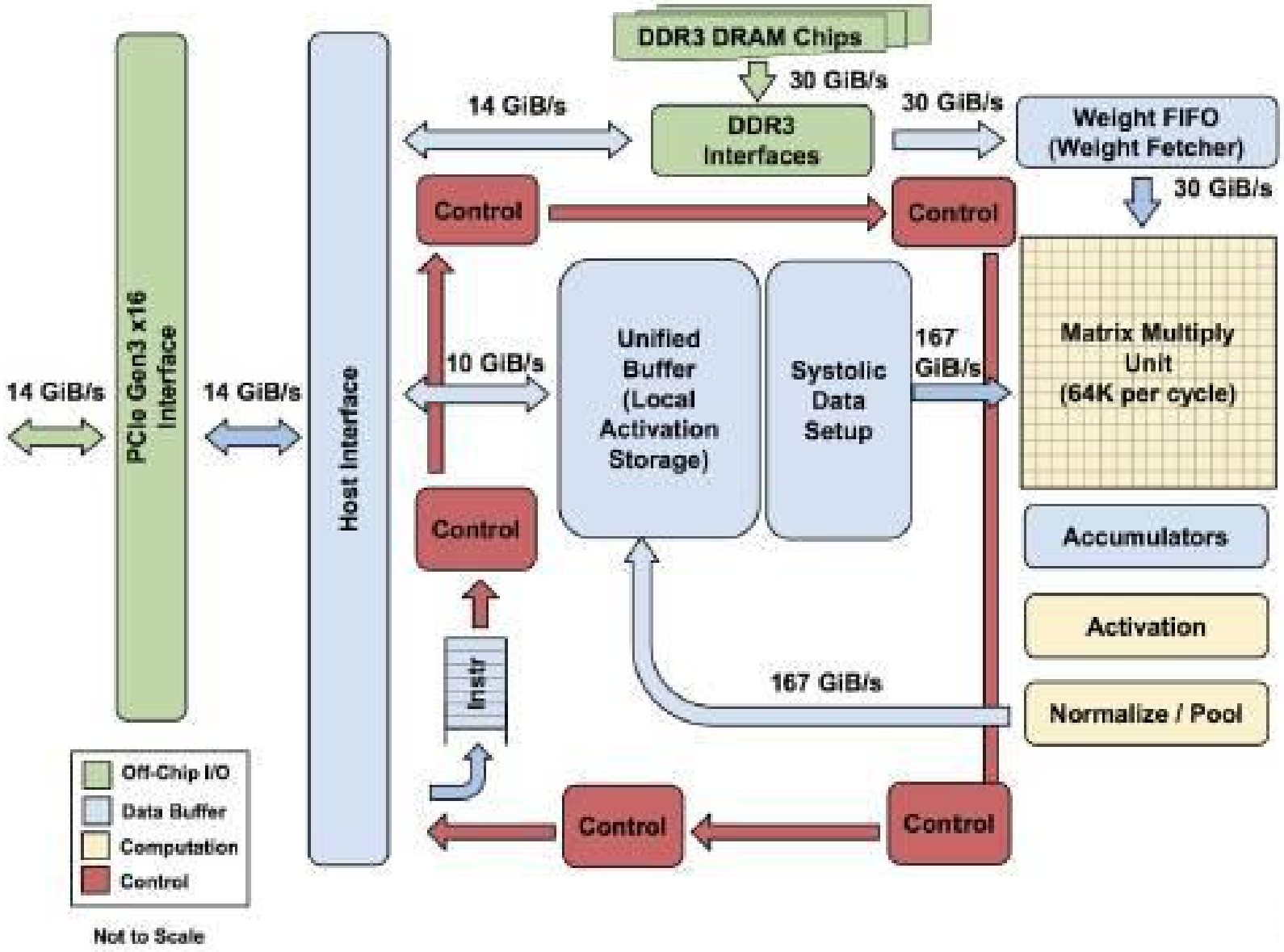
输入					权值		
1	2	3	4	5	1	2	3
6	7	8	9	10	4	5	6
11	12	13	14	15	7	8	9
16	17	18	19	20			
21	22	23	24	25			

变换后的输入											
1	3	5	2	4	0	6	8	10	7	9	0
11	13	15	12	14	0	16	18	20	17	19	0
21	23	25	22	24	0	0	0	0	0	0	0

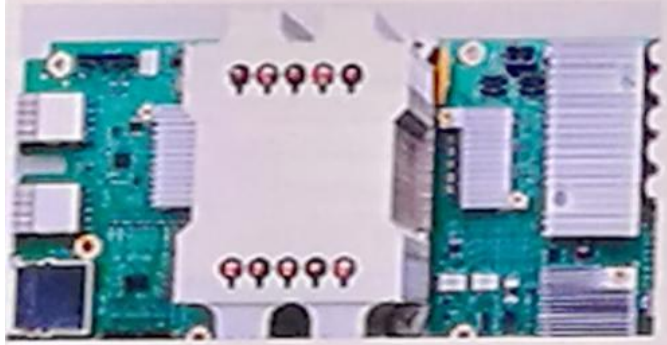
变换后的权值											
1	3		2	0		4	6		5	0	
7	9		8	0		0	0		0	0	

TPUv1架构

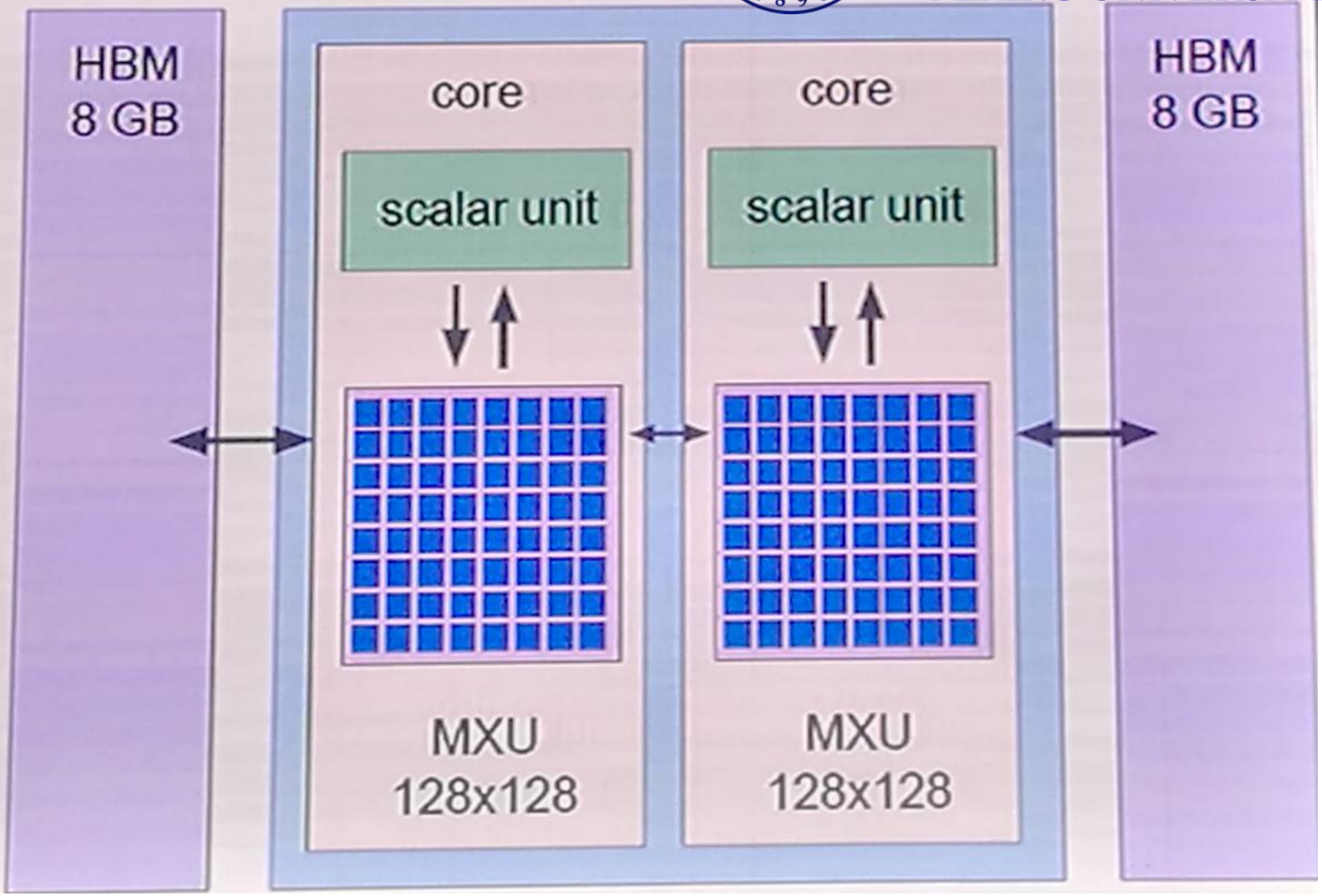
- 核心：64K的8位矩阵乘单元阵列、片上28MB的软件管理存储器。
- 峰值计算能力：每秒92 TOPS。
- 指令：通过PCIE Gen3的16 lane总线送入指令缓冲。
- 内部模块：256字节宽（即1024位宽）的通路相连。
- 乘加单元：256x256个。
- 累加器：4MiB 32位。
- 乘法矩阵所需的权重分阶段从片外8GiBDRAM读入到片上的权重队列。
- 权重队列深度：4个数据片段。
- 中间结果被保存在24MiB的片上统一缓冲区UnifiedBuffer，也作为矩阵单元的输入。



TPUv2 Chip

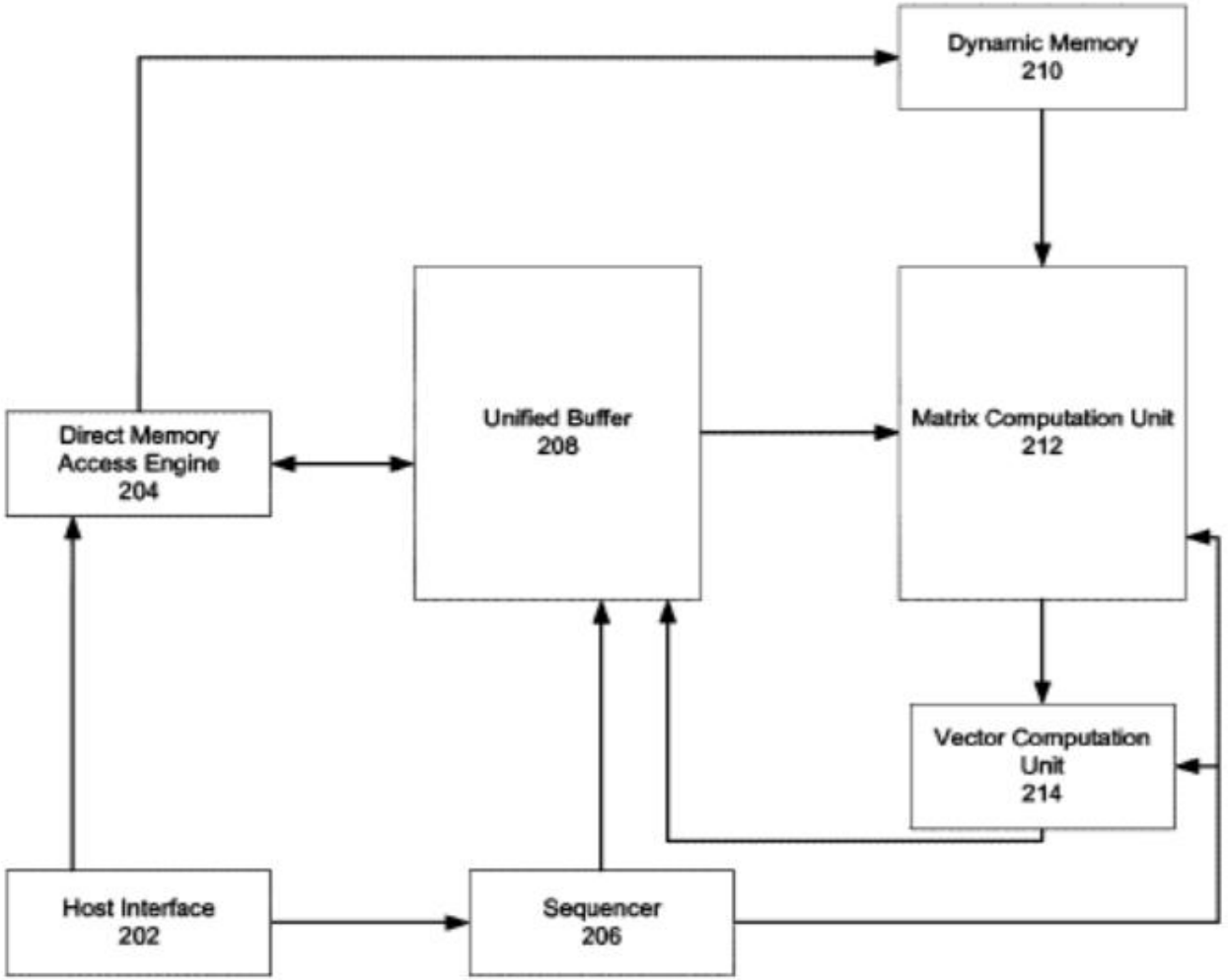


- 16 GB of HBM
- 600 GB/s mem BW
- Scalar unit: 32b float
- MXU: 32b float accumulation but reduced precision for multipliers
- 45 TFLOPS



TPUv1

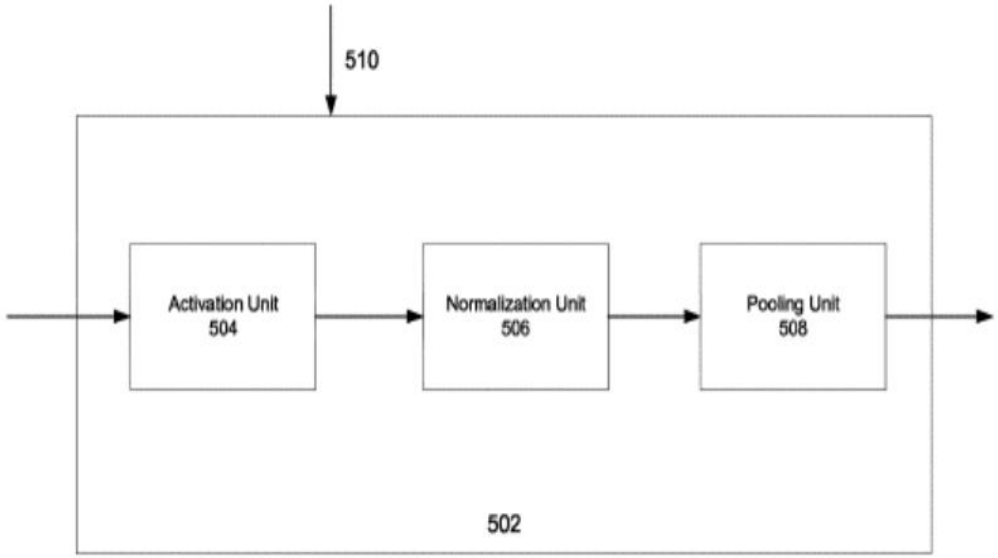
- 主机接口(host interface)：接受指令、神经网络参数。
- 乘法矩阵(matrix computation unit)。一个二维脉动阵列。从动态存储器和统一缓冲区读入权值和激活输入。将权值和激活函数处理后的结果向量发送至向量计算单元。
- 向量计算单元(vector computation unit)。处理乘法矩阵的结果向量，并将结果发送至统一缓冲区。



向量计算单元

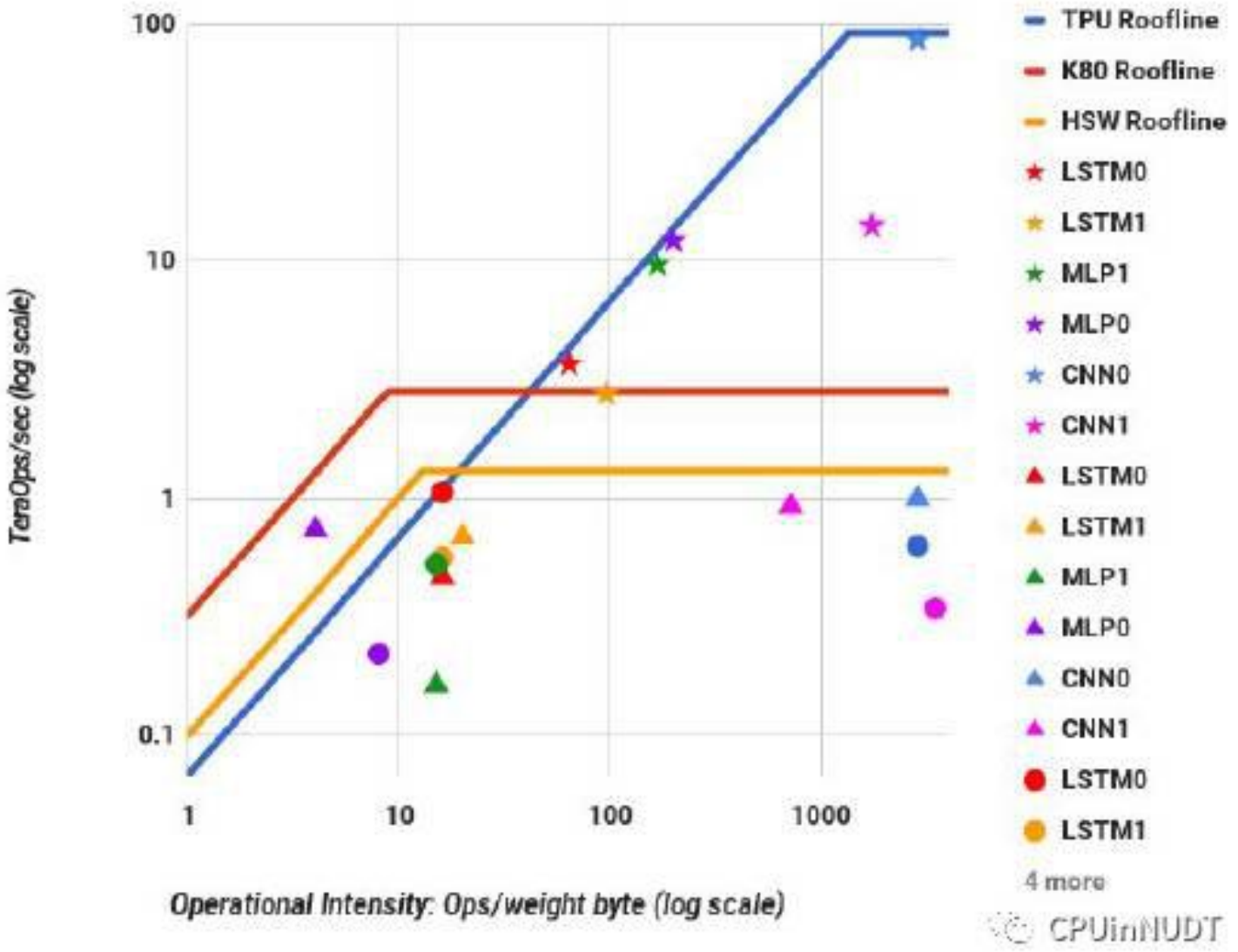
向量计算单元(vector computation unit)：从乘法矩阵接收和向量。将激活函数、标准化、池化的结果发送至统一缓冲区。标准化单元和池化单元的位置可以互换。

- ✓激活函数单元(activation unit)
- ✓标准化单元(normalization unit)
- ✓池化单元(pooling unit)。
- ✓控制信号(control signal)：从序列发生器读入。控制激活函数的结果是否需要标准化或池化，明确标准化或池化参数（例如步长）。



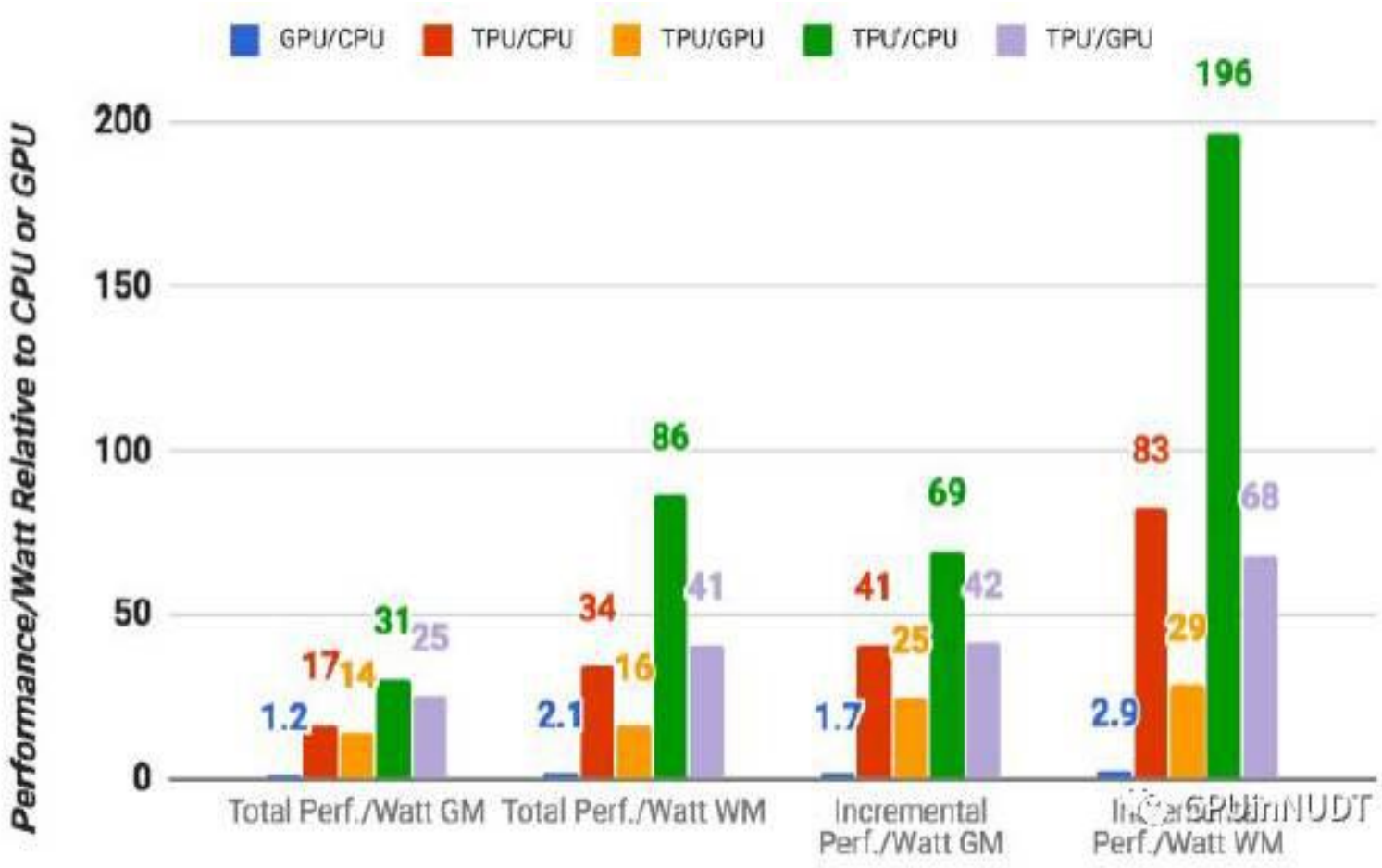
TPUv1速度

- TPU的峰值速度是GPU的15.3倍。
- TPU对于各种神经网络的计算速度都在GPU的roofline之上。



TPUv1的能效比

- GPU服务器（蓝色条柱）和TPU服务器（红色条柱）相对于CPU服务器的能效比（功耗/瓦TDP），以及TPU服务器（黄色条柱）相对于GPU服务器的能效比。
- TPU'是改进的TPU。绿色条柱显示了与CPU服务器对比的性能，薰衣草紫色部分显示了与GPU服务器对比性能。
- 总和指标包含了主机服务器的功耗，而增量数据不包含。GM和WM是几何与加权平均数据。



北京大学
PEKING UNIVERSITY

型号	应用领域	特点	发布时间	能效比
TPU 1.0	面向终端的DNN推理云服务器	高速	2016	92TOPS/40W
TPU 2.0/Cloud TPU	面向终端的DNN推理/训练云服务器	可支持训练	2017	180TFLOPS
TPU 3.0	面向终端的DNN推理/训练云服务器	高性能、高功耗	2018	100PFLOPS

人工智能处理器设计三大矛盾

北京大学
PEKING UNIVERSITY

- 有限规模的硬件vs无限规模的算法
- 结构固定的硬件vs千变万化的算法
- 能耗受限的硬件vs精度优先的算法

解决方法

- 硬件神经元虚拟化
- 深度学习指令集
- 稀疏神经网络处理器结构

DianNao

通用性
速度

神经网络的硬件实现：

每个神经元用逻辑电路实现，每个突触用锁存器或者RAM来实现。

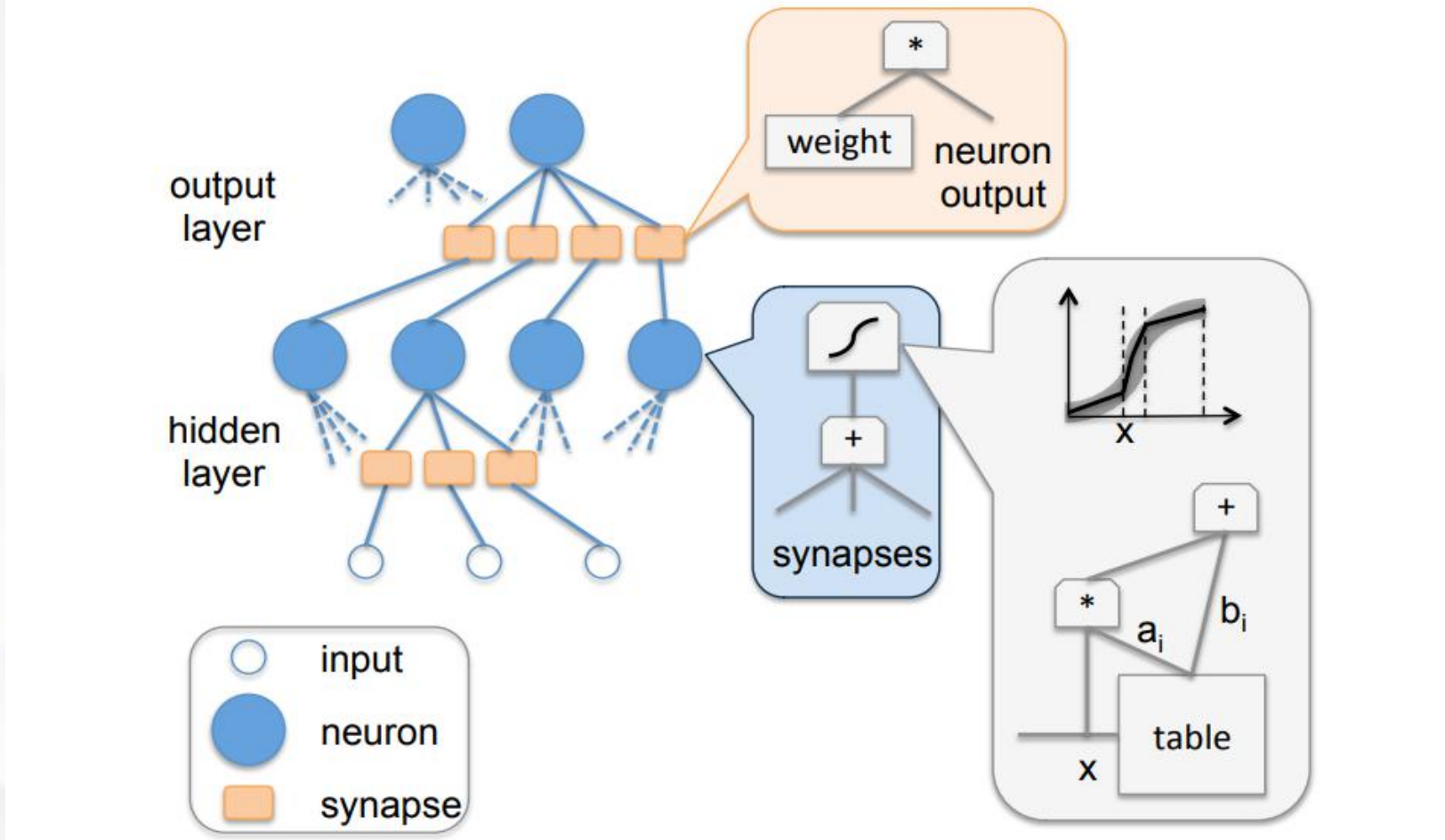


Figure 9. *Full hardware implementation of neural networks.*

- 将每个神经元和突出都用电路实现，面积和功耗随规模的增加大幅上升。
- 使用Synopsys ICC布局布线，TSMC 65nm标准工艺。
- 横坐标Tn×Ti表示有Tn个输入神经元、Ti个输出神经元的带Sigmoid激活函数的全连接层。

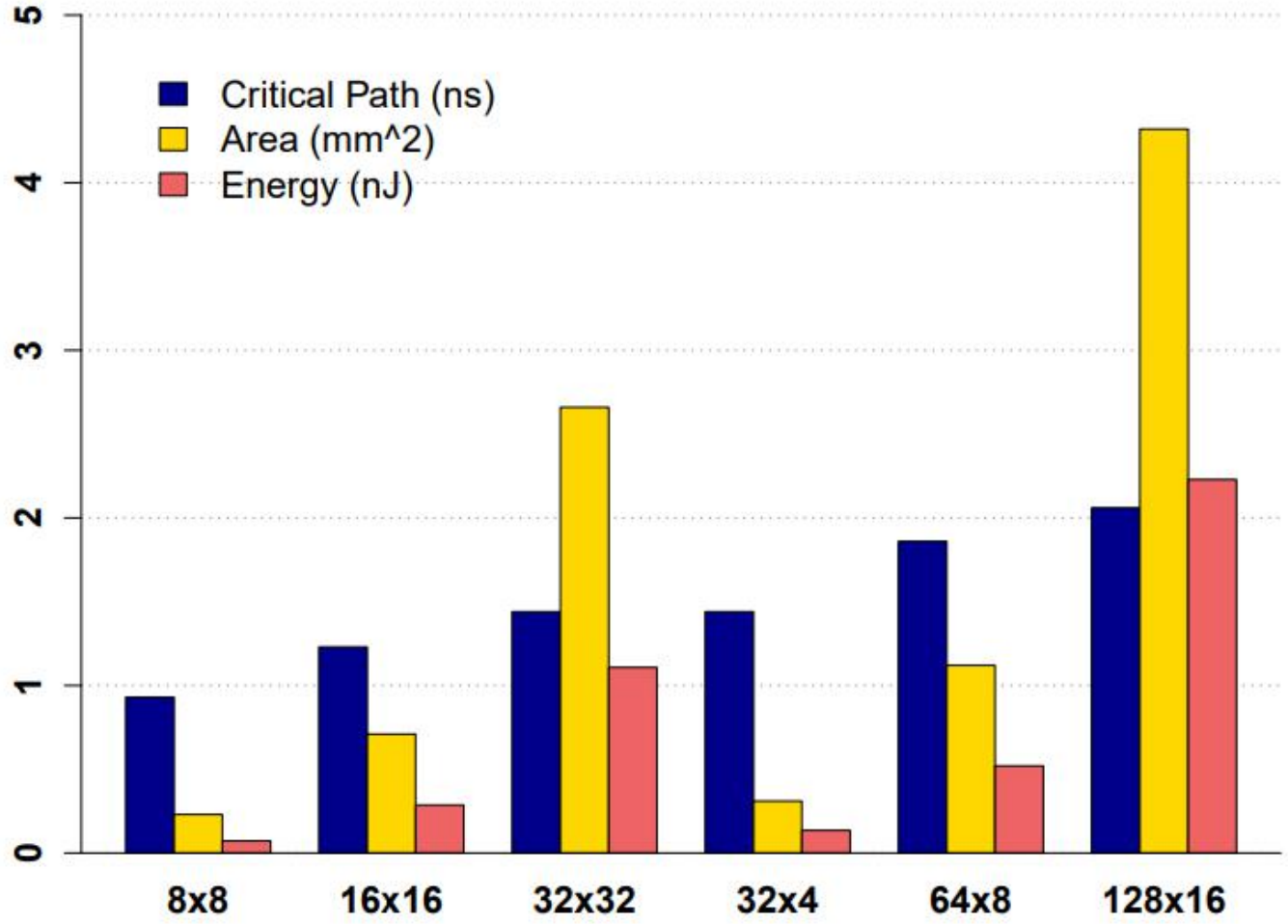


Figure 10. *Energy, critical path and area of full-hardware layers.*

卷积层庞大的输入数据通过线性叠加的方式进行处理，因此可以采用分块计算的方式来减少对内存带宽的占用。

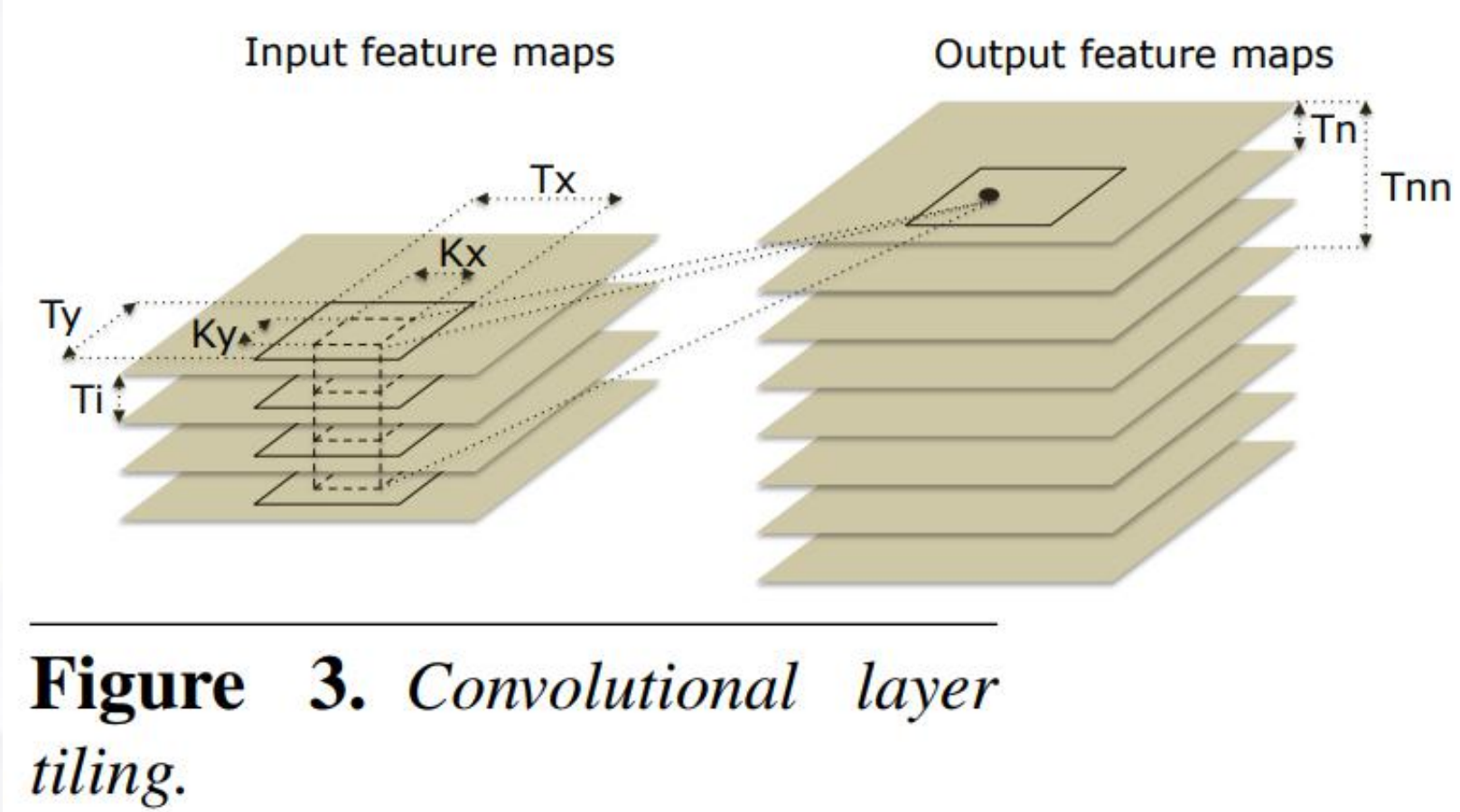


Figure 3. *Convolutional layer tiling.*

- NFU：神经功能单元
- Nbin, Nbout, SB：缓存区
- DMAs：通过直接内存访问来实现预加载
- Control Processor：进程控制，驱动三个缓存区和DMAs的运行

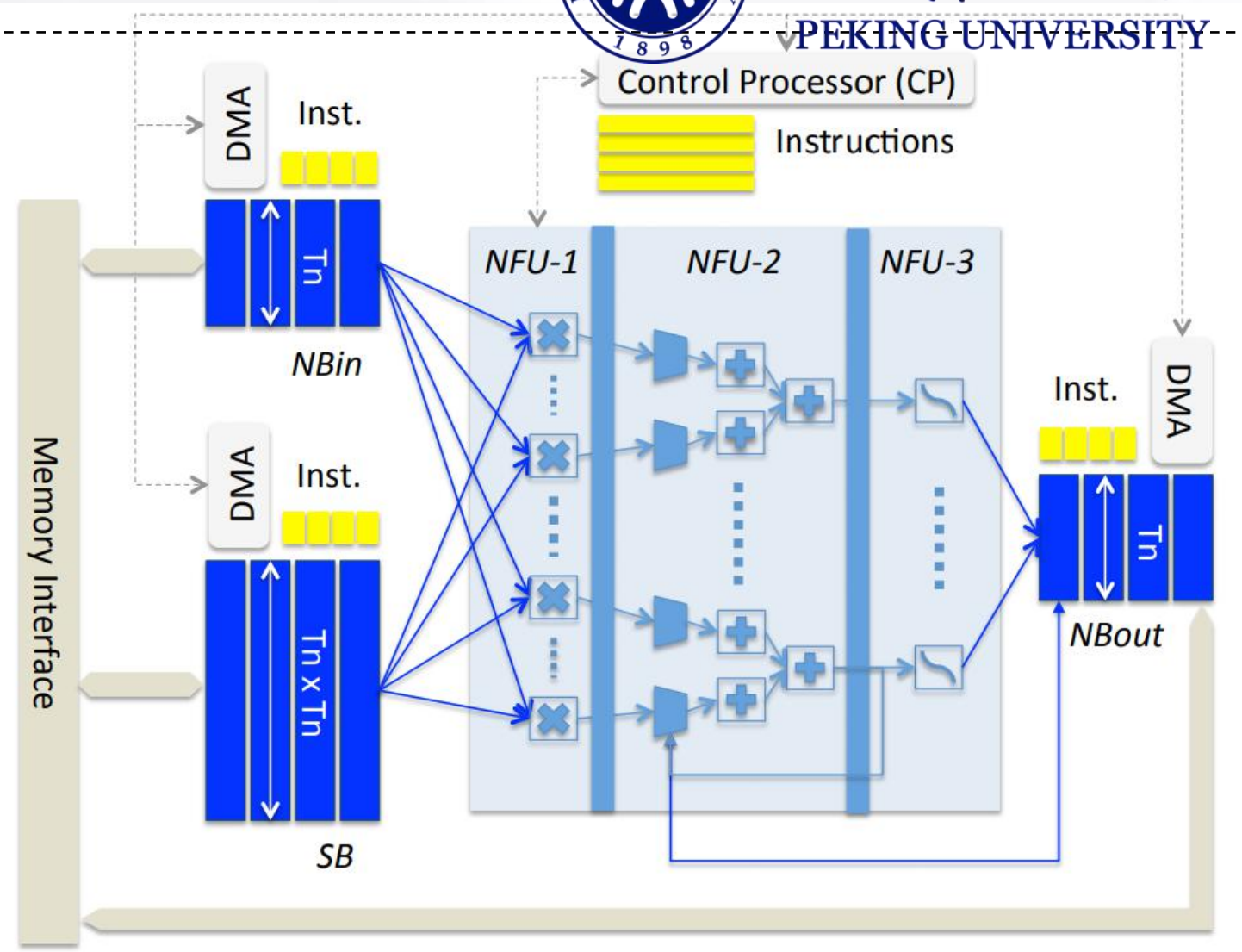


Figure 11. *Accelerator.*

- 面积3.2mm^2
- 功耗480mW
- 与128位2GHzSIMD处理器对比
  实现117.87倍的加速
- 功耗同比减少了21.08倍

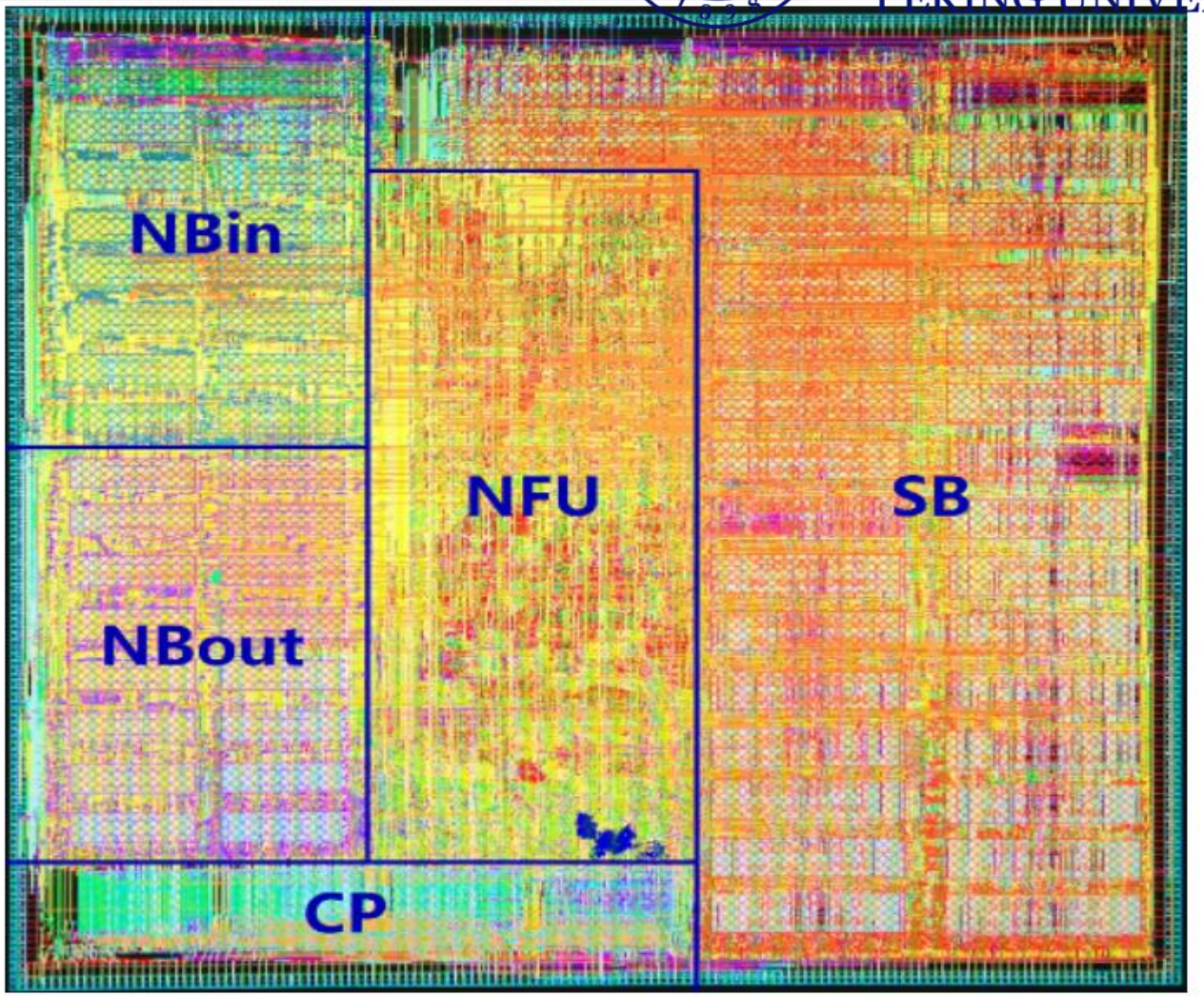


Figure 15. *Layout (65nm).*

加速器平均能量比同类型加速器降低一个数量级以上

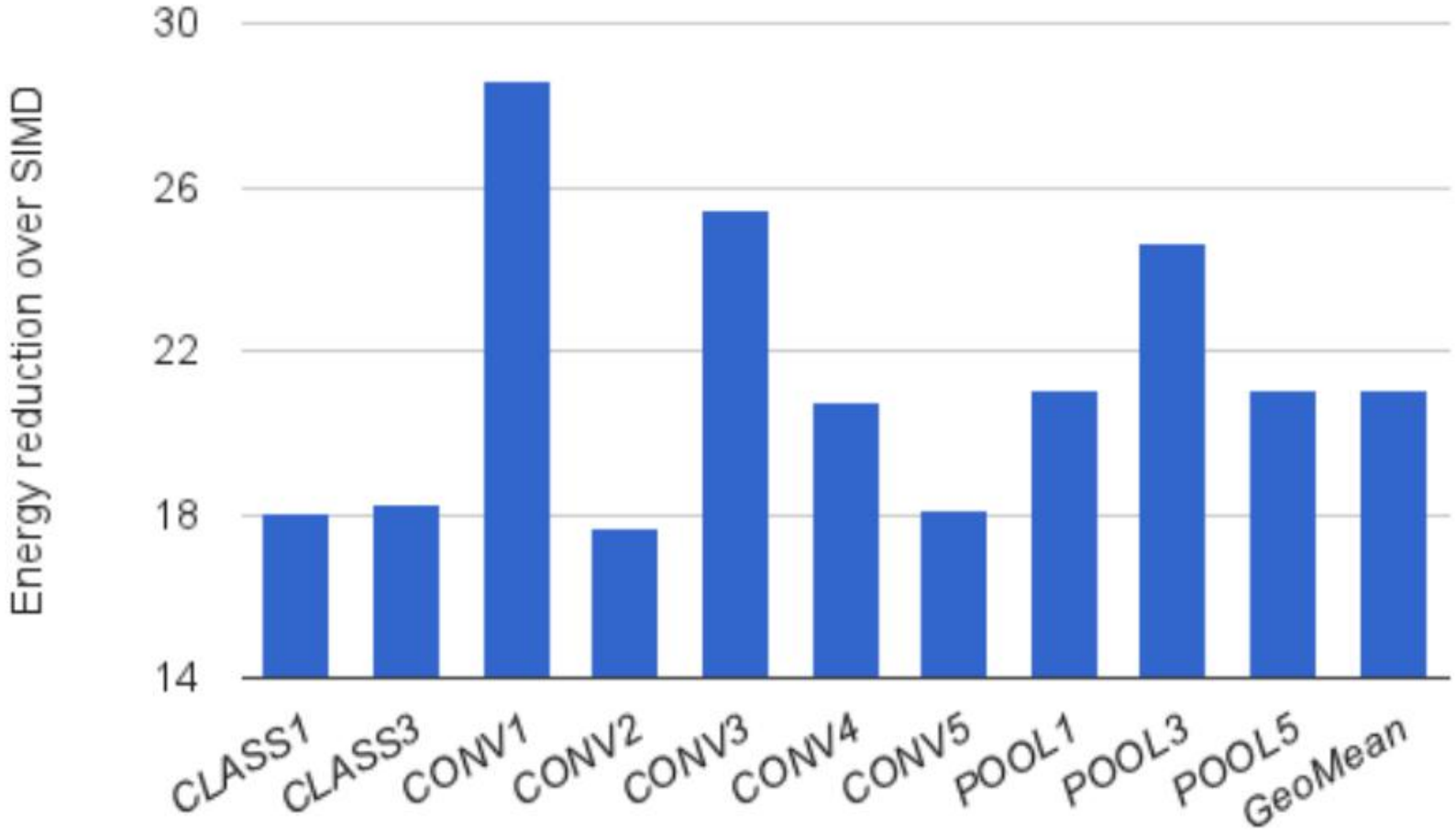


Figure 17. *Energy reduction of accelerator over SIMD.*

- 消耗能量比例：
- DianNao中主内存访问的能量占主导地位；
- SMID中大约2/3的能量用于计算，只有1/3用于内存访问。

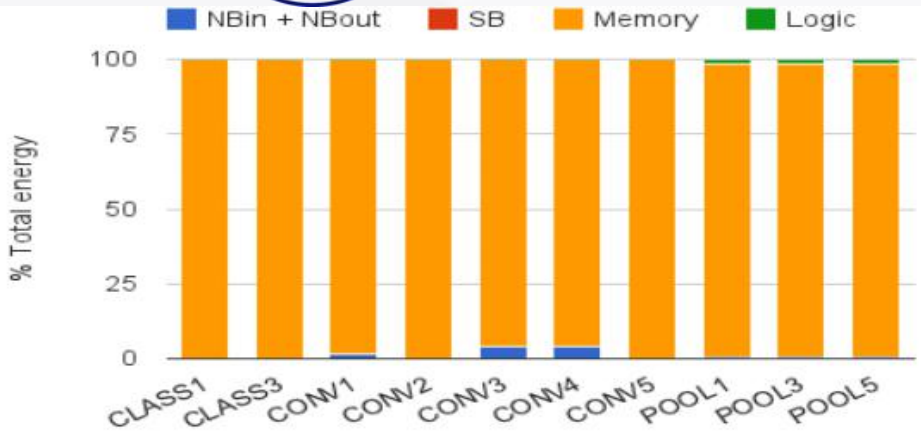


Figure 18. *Breakdown of accelerator energy.*

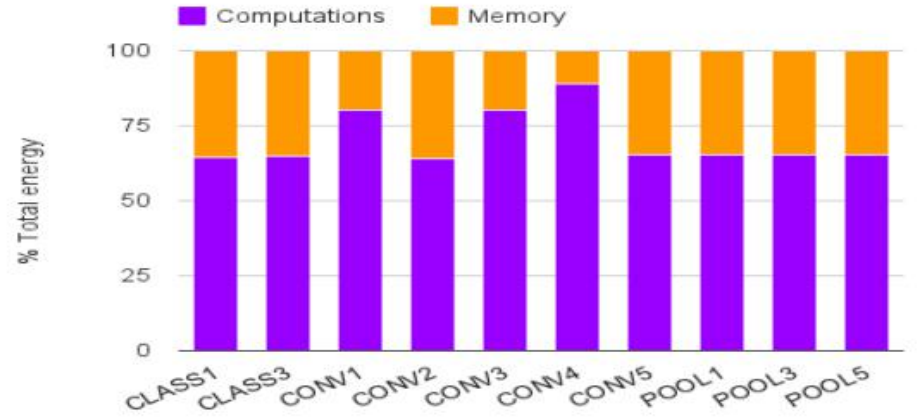


Figure 19. *Breakdown of SIMD energy.*

北京大学
PEKING UNIVERSITY

DaDianNao

卷积层

归一化层

池化层

全连接层

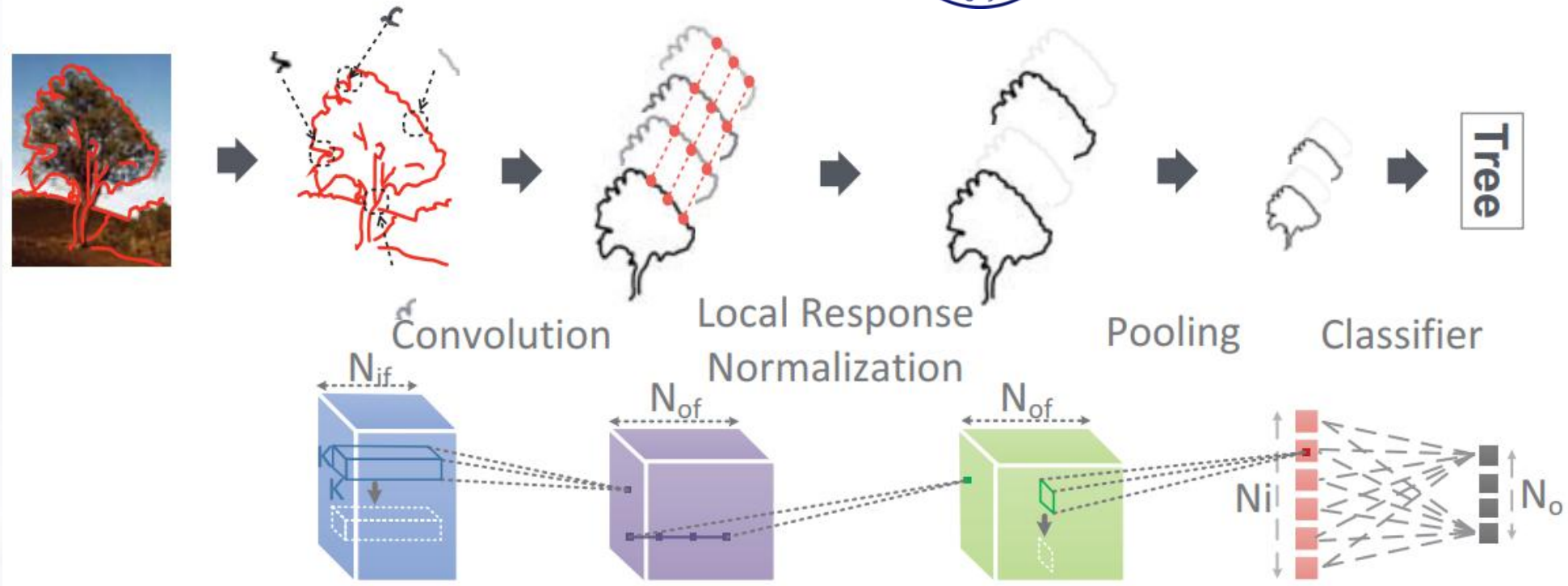


Figure 1: *The four layer types found in CNNs and DNNs.*

单纯扩大NFU的局限：

导线占据了芯片上的较大面积

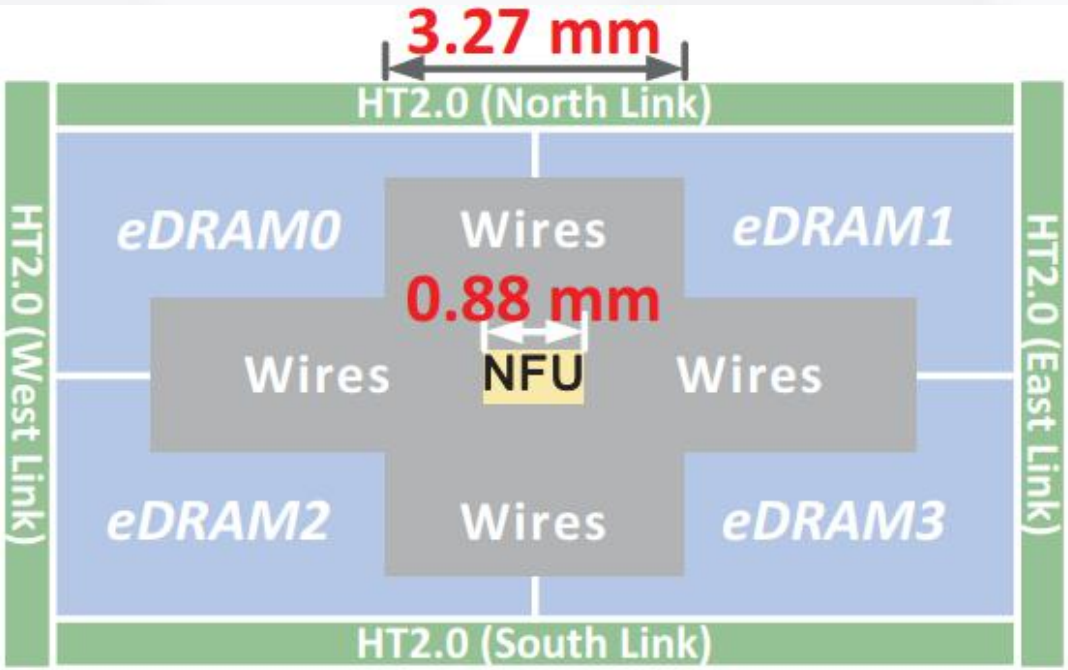


Figure 4: *Simplified floorplan with a single central NFU showing wire congestion.*

解决方法：

将输出神经元分散在不同的块中

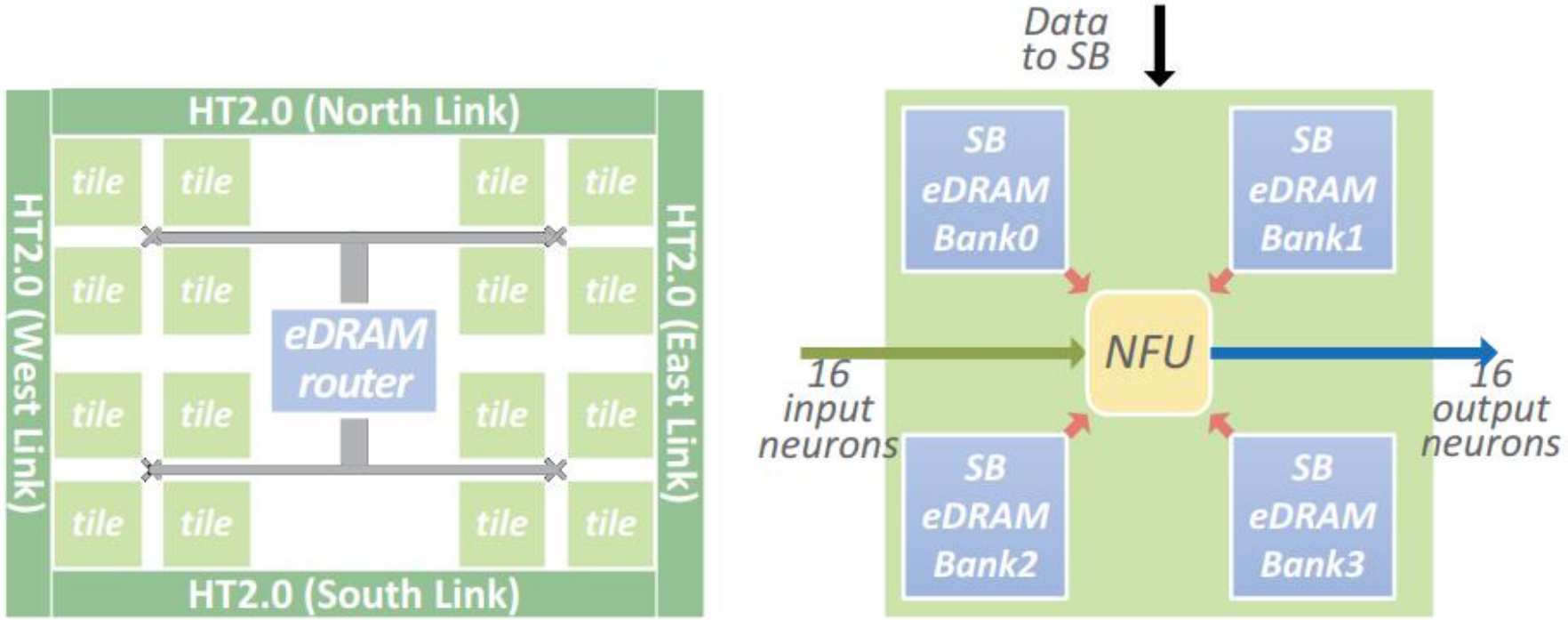


Figure 5: *Tile-based organization of a node (left) and tile architecture (right). A node contains 16 tiles, two central eDRAM banks and fat tree interconnect; a tile has an NFU, four eDRAM banks and input/output interfaces to/from the central eDRAM banks.*

DaDianNao的NFU新增：

- Multipliers
- Adders
- Mux
- Transfer function：增加LRN, LCN，反向传播等的处理能力

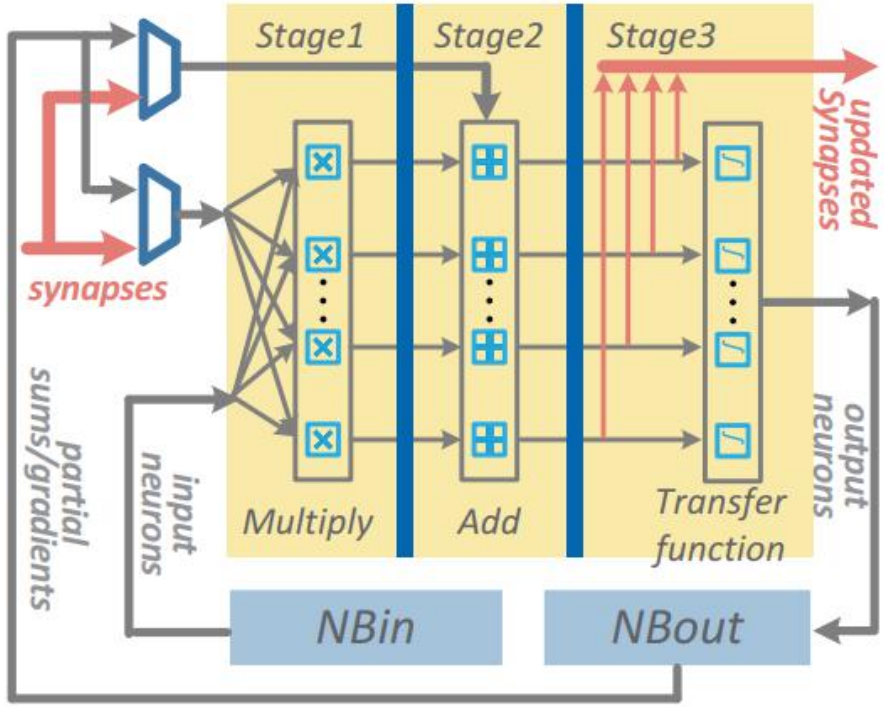


Figure 6: *The different (parallel) operators of an NFU: multipliers, adders, max, transfer function.*

在不同的条件下，

NFU可以采取不同的策略

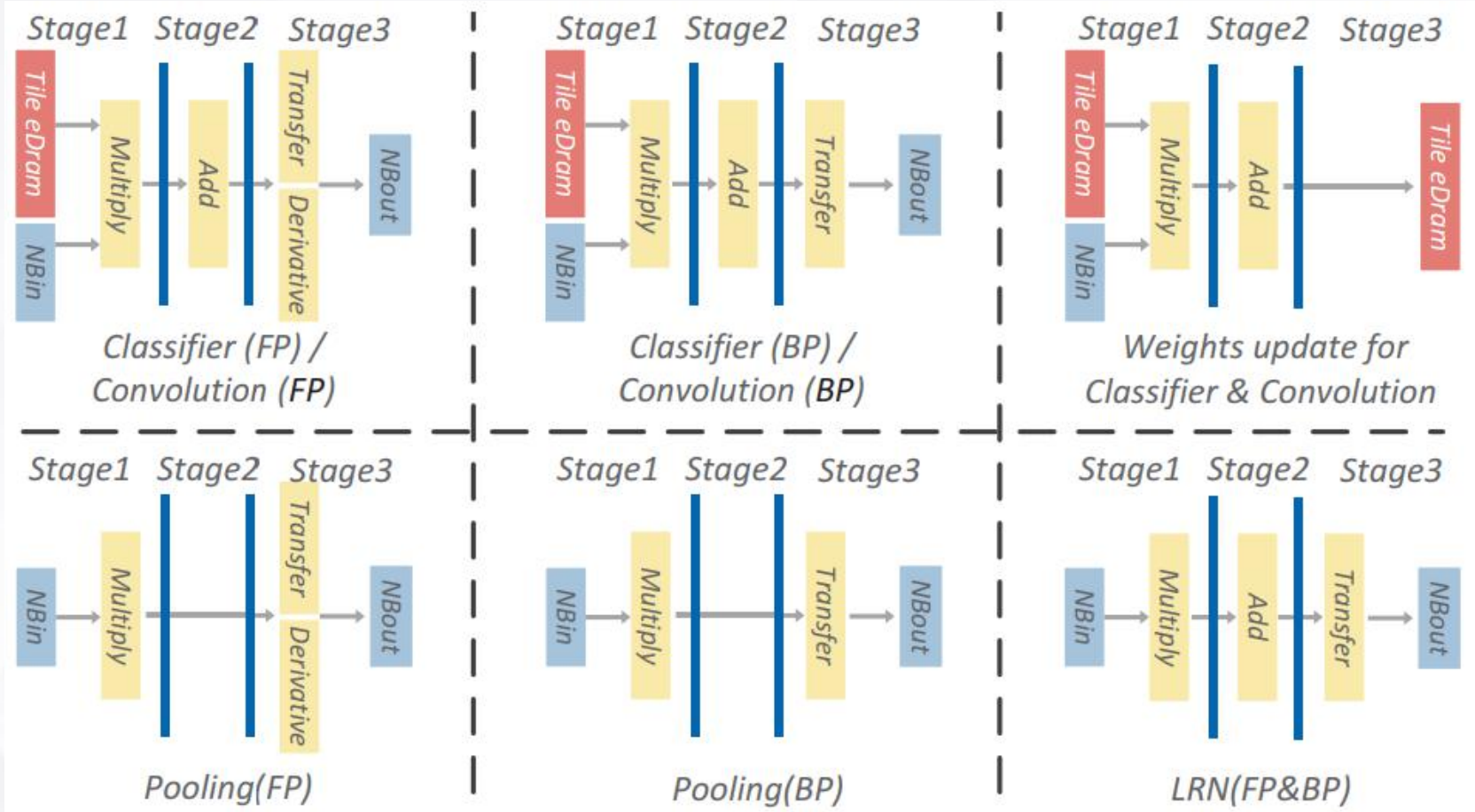


Figure 7: *Different pipeline configurations for CONV, LRN, POOL and CLASS layers.*

多节点映射：

- 标准化层具有极强的局部特性，不需要节点之间的数据交互。
- 卷积层与池化层具有一定的局部特性，feature map中的一个元素只与其周围的矩形元素有运算关联，恰当分配节点数据，使得只有相邻节点存在数据交互。
- 全连接层不具备局部特性，需要设计广播策略以提升能效。

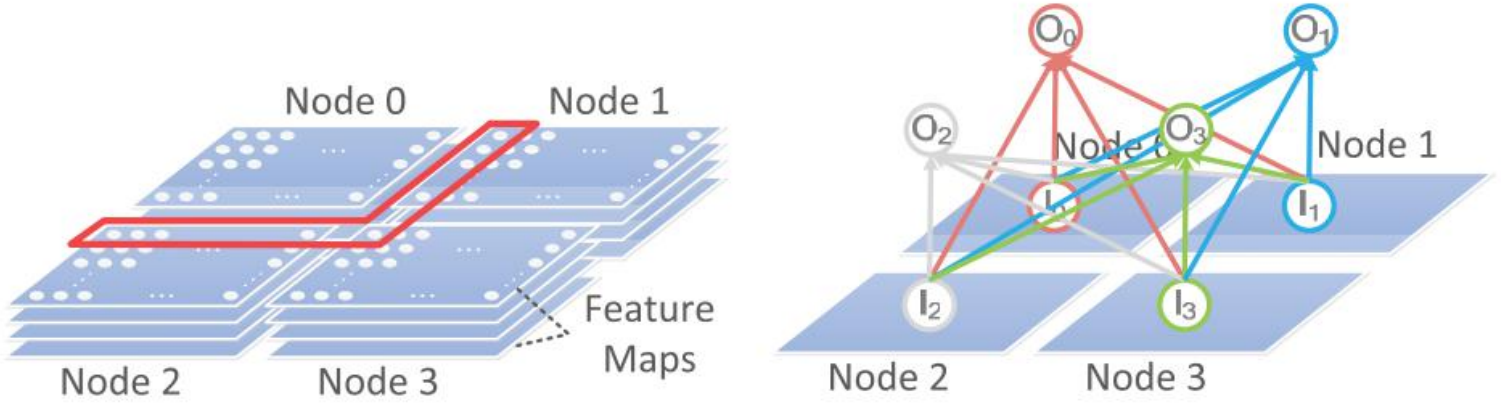


Figure 8: *Mapping of (left) a convolutional (or pooling) layer with 4 feature maps; the red section indicates the input neurons used by node 0; (right) a classifier layer.*

- 28nm制程，每个节点 $67.73mm^2$
- 芯片峰值功率为15.97W
- 在使用64个节点的情况下，性能可超过单个GPU的450.65倍，能耗降低150.31倍

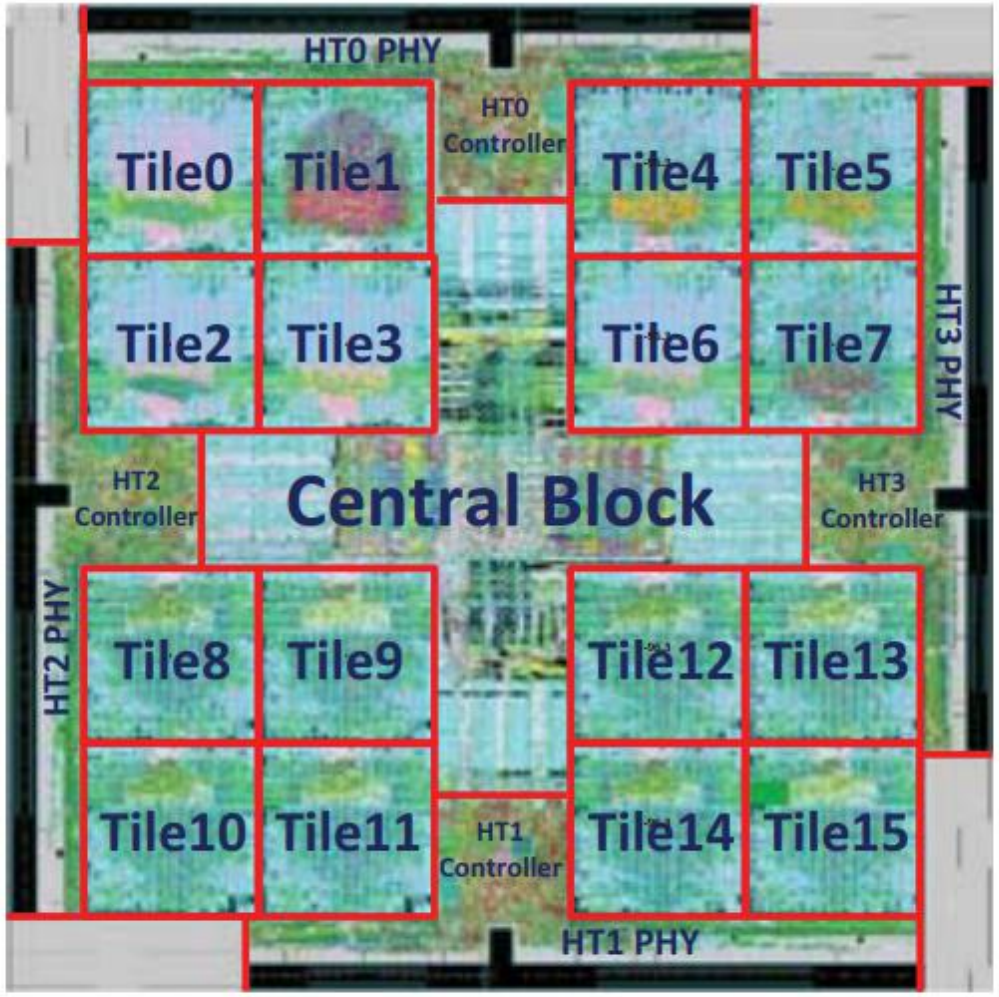


Figure 9: *Snapshot of the node layout.*

实际加速比率：

与GPU相比，DaDianNao有着显著的加速率

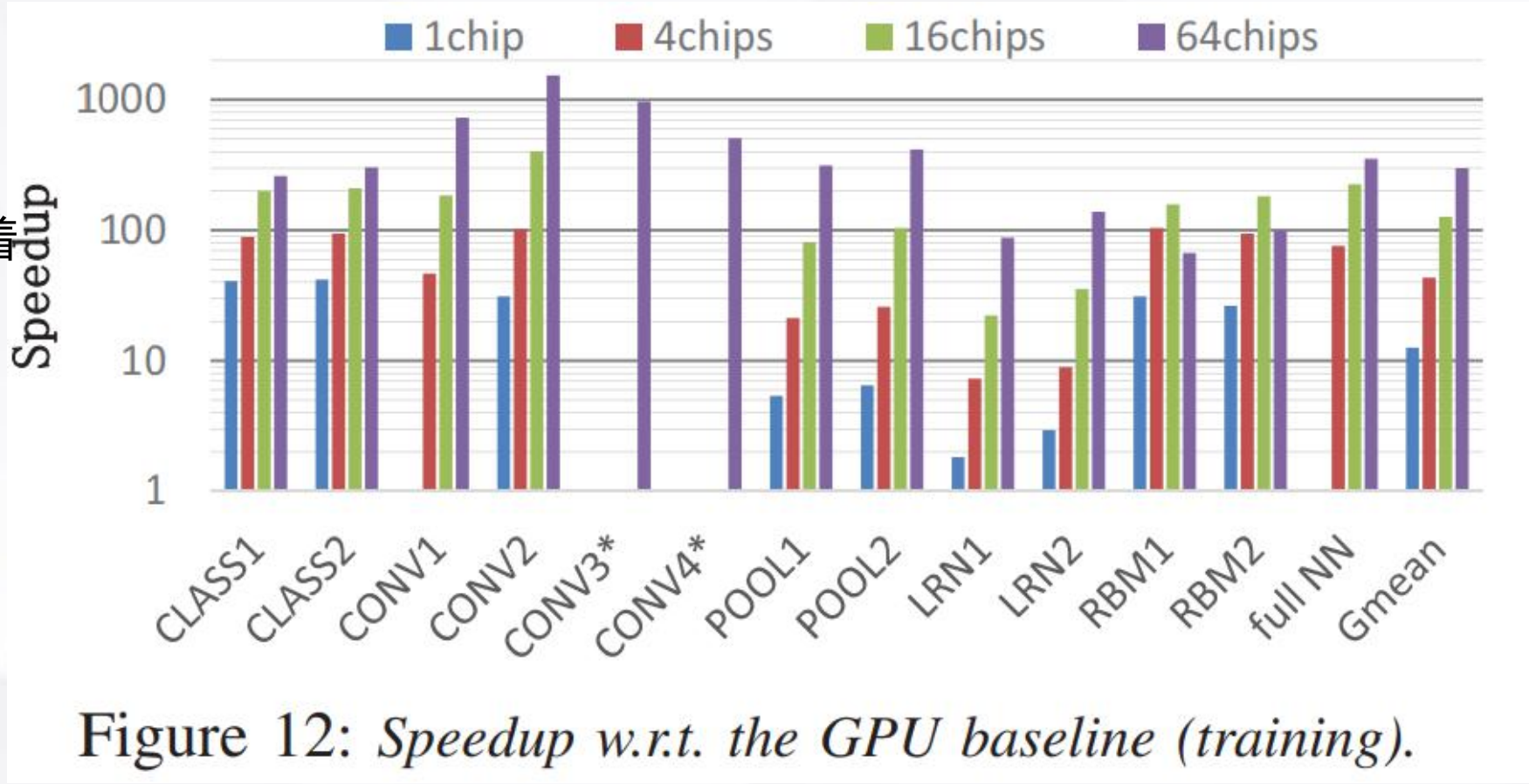


Figure 12: *Speedup w.r.t. the GPU baseline (training).*

节能比率：

与GPU相比，芯片实现了显著的耗能降低

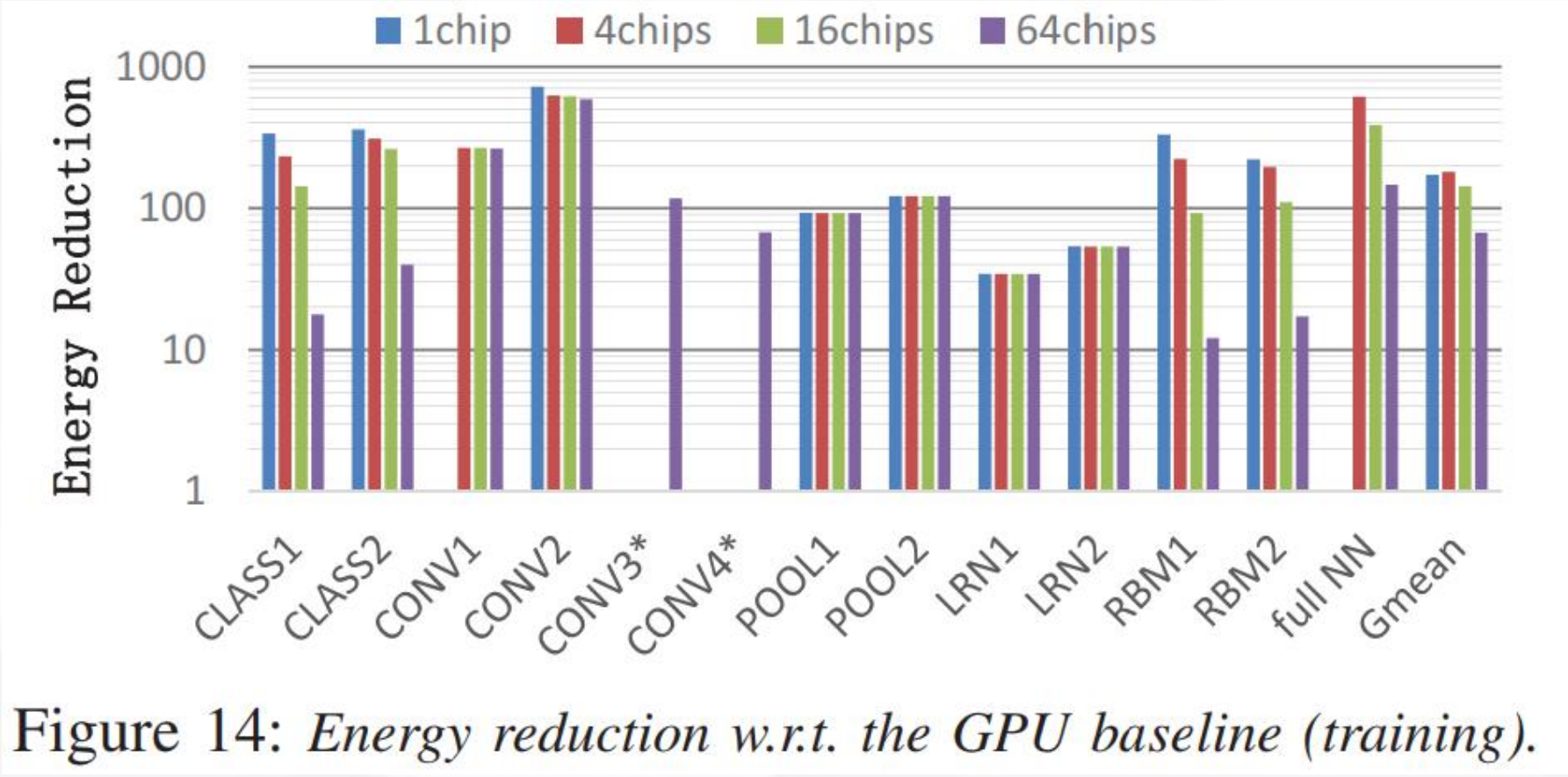


Figure 14: *Energy reduction w.r.t. the GPU baseline (training).*

ShiDianNao

- 通过加速器和摄像头的直连来绕过内存。
- 拒绝与内存交互，提升能效比
- 应用于摄像头

ShiDianNao架构：

- NBin, Nbout, SB: 缓冲区
- IB and Decoder:译码器
- ALU：算术单元
- Buffer Controller：缓冲控制器
- NFU：神经功能单元

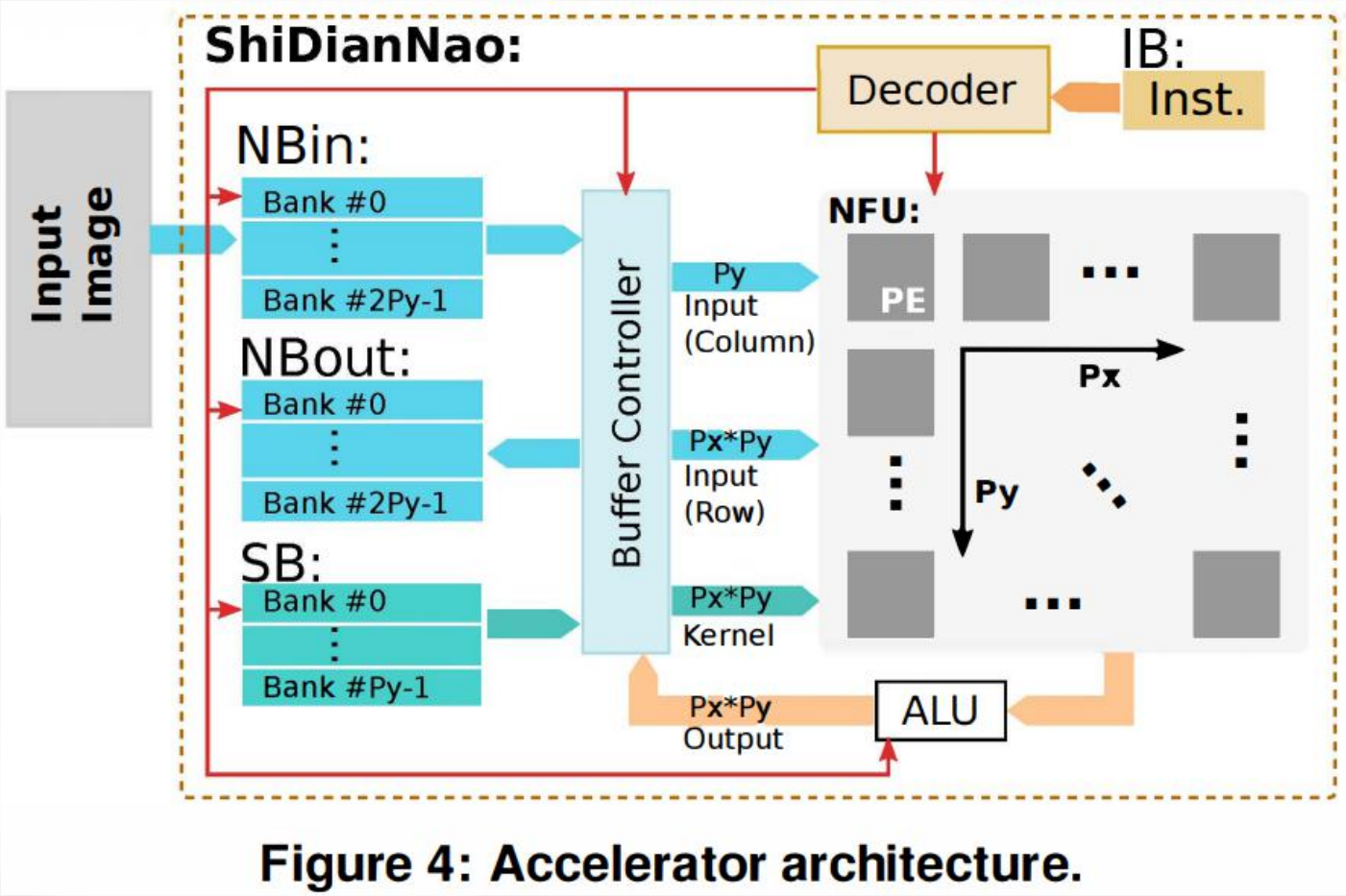


Figure 4: Accelerator architecture.

PE单元架构

输入数据可以在相邻处理单元传播，

提高输入数据的复用率

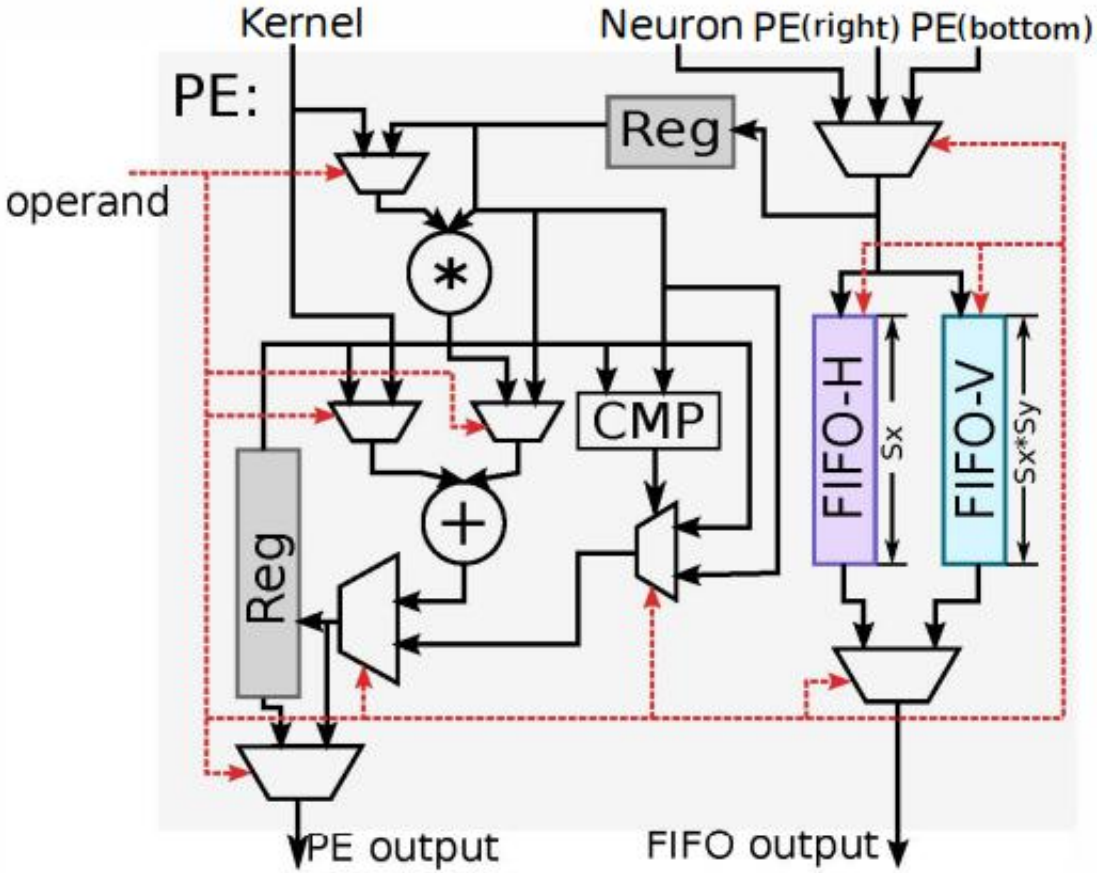


Figure 6: PE architecture.

卷积层计算示例

减少PE间数据访存44%~75%

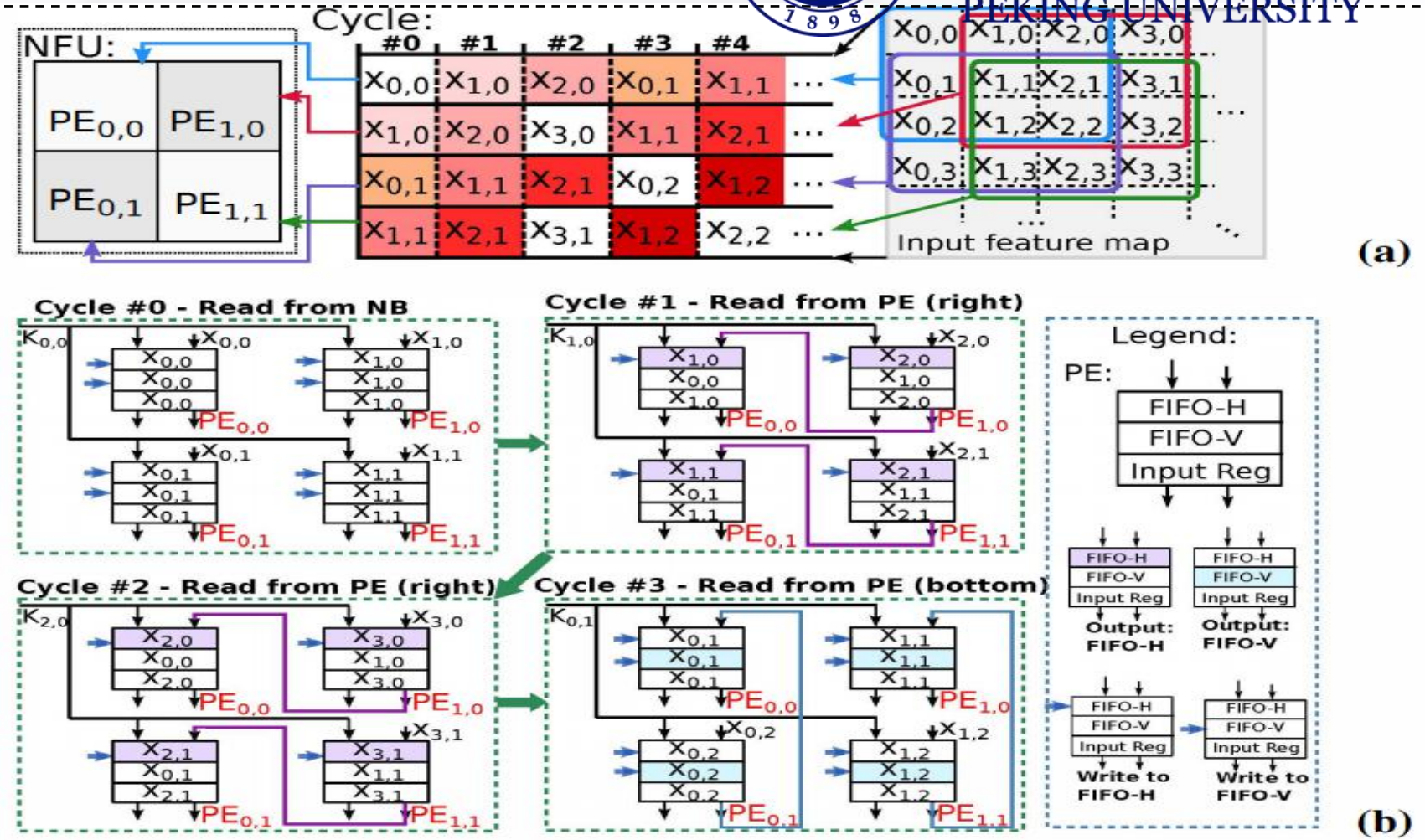


Figure 13: Algorithm-hardware mapping between a convolutional layer (convolutional window: 3×3; step size: 1×1) and an NFU implementation (with 2×2 PEs).

控制模式：

A 读banks0~Py-1

B 读banksPy~2Py-1

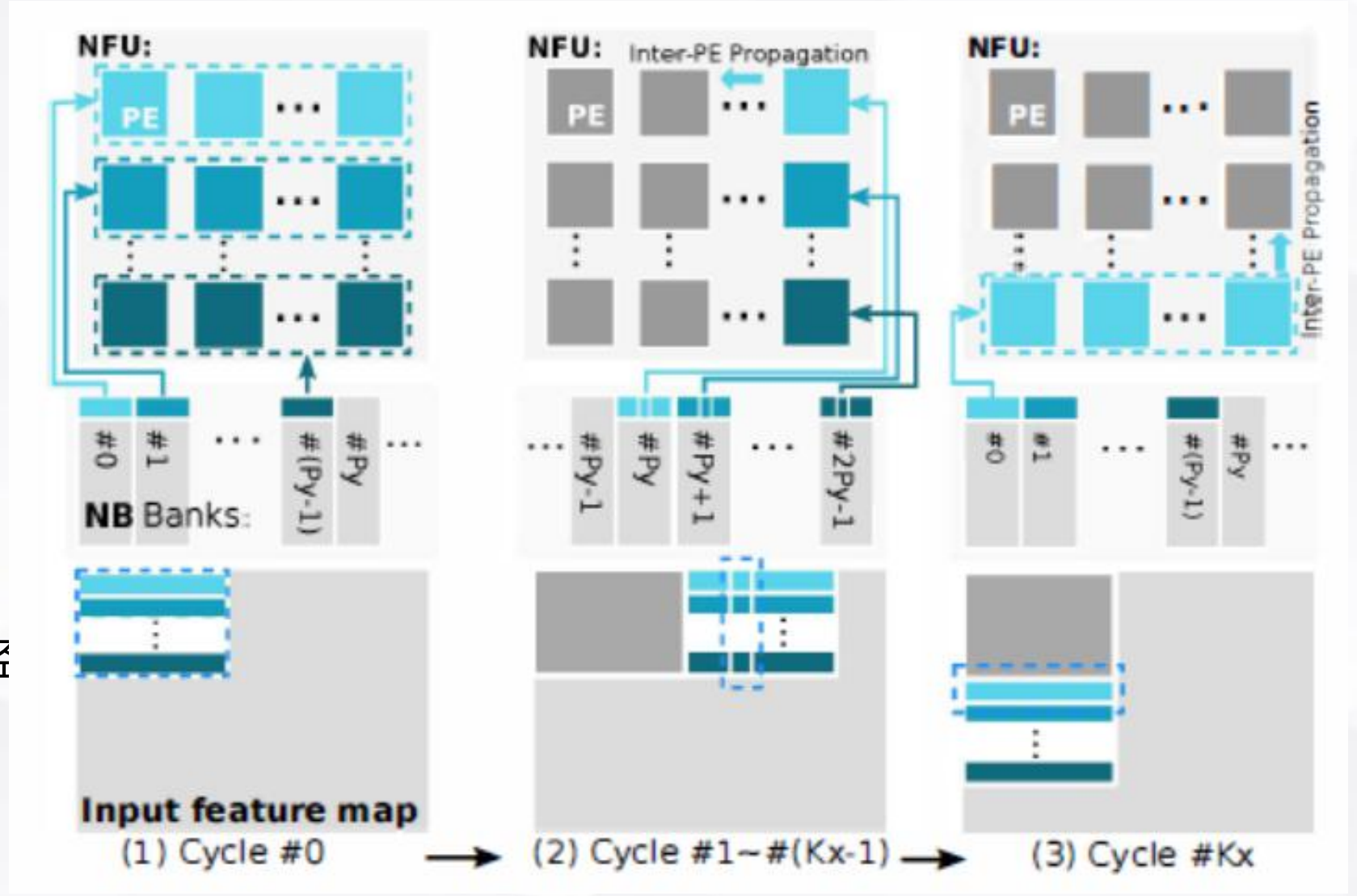
C 读1个bank

D 读1个神经元

E 以一定步长读取神经元

F 在Py个bank中分别读取1个神经元

对于右图：a/b/e—f—c



LRN层的分解图示：

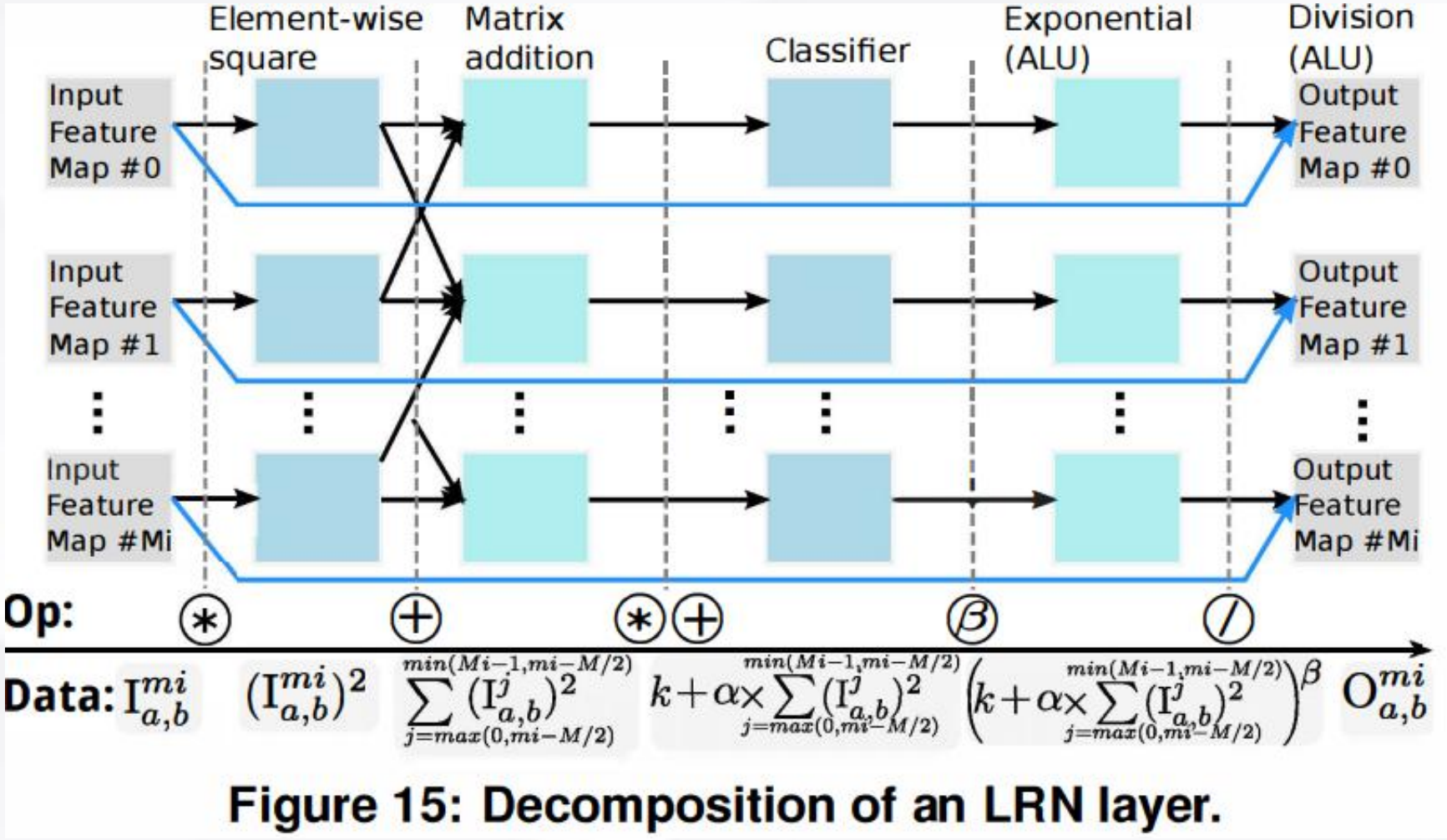


Figure 15: Decomposition of an LRN layer.

LCN层的分解图示：

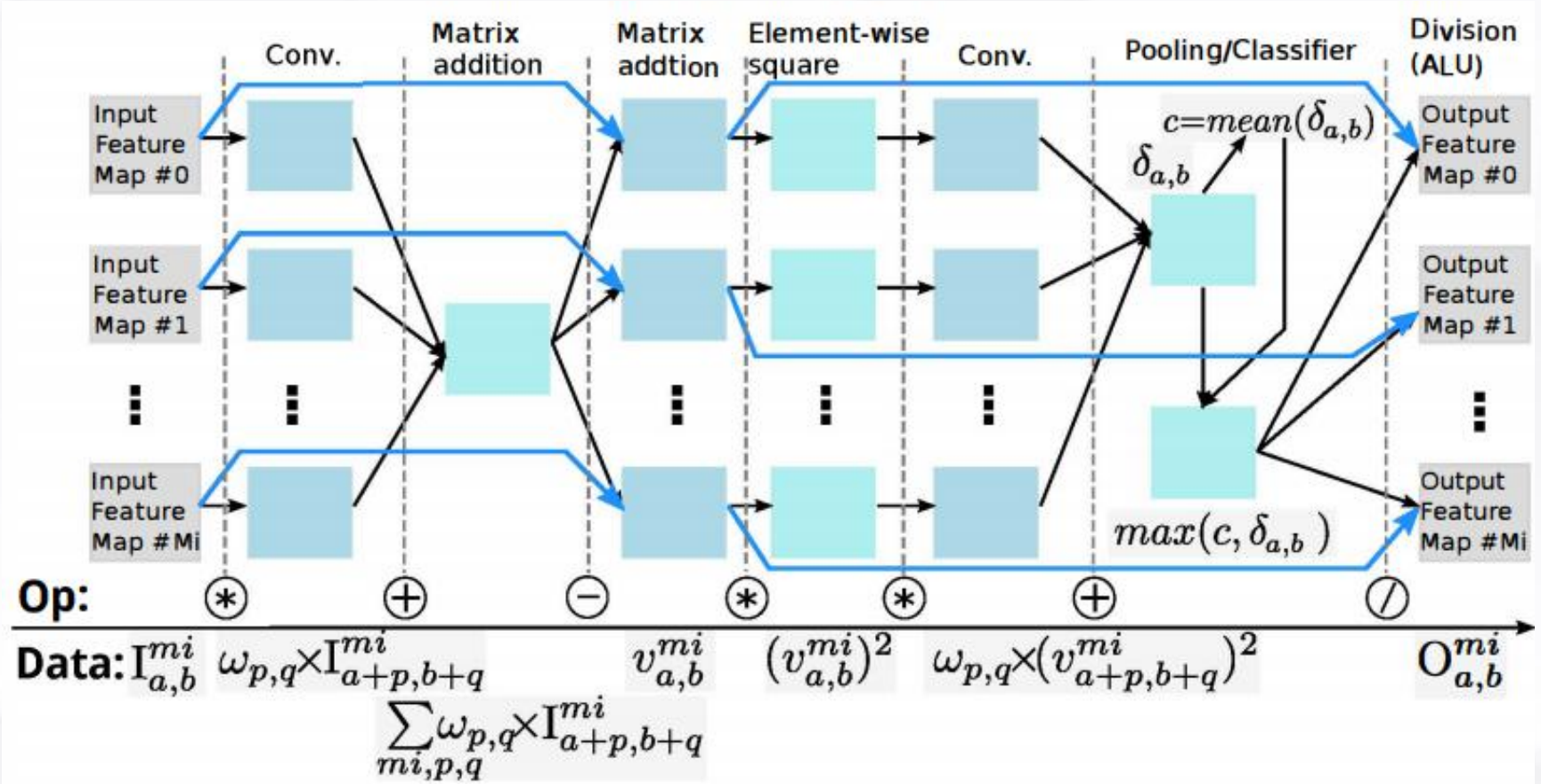


Figure 16: Decomposition of an LCN layer.

- 65nm制程
- 面积为4.86 mm^2
- 总功耗320.10mW
- 平均比CPU快50倍，比GPU快30倍
- 比GPU节能4700倍

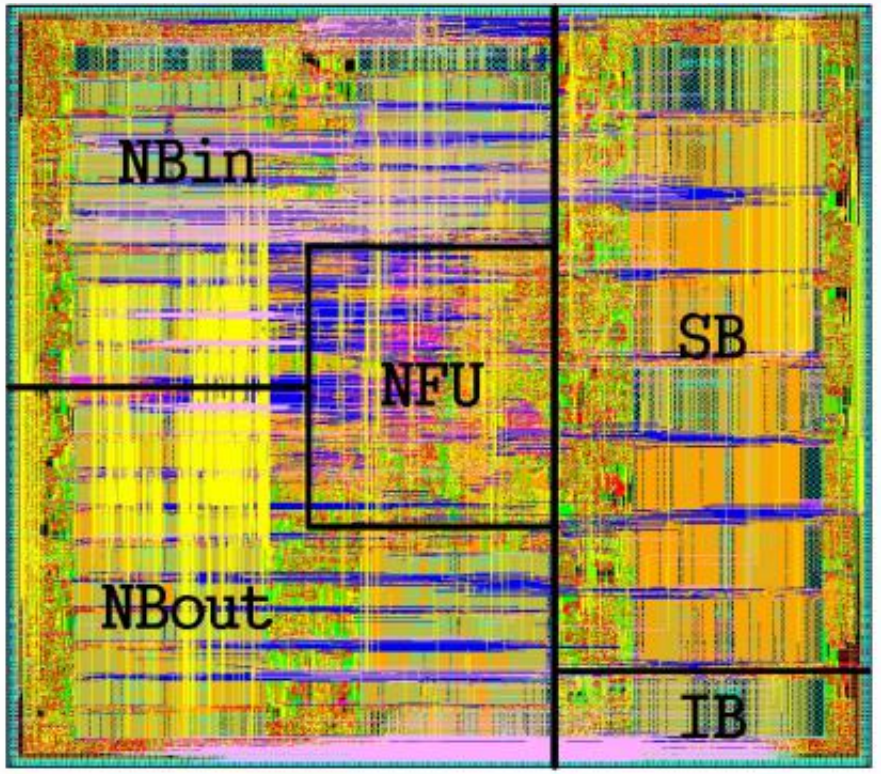


Figure 17: Layout of ShiDianNao (65 nm).

ShiDianNao速率：

优于GPU和DianNao

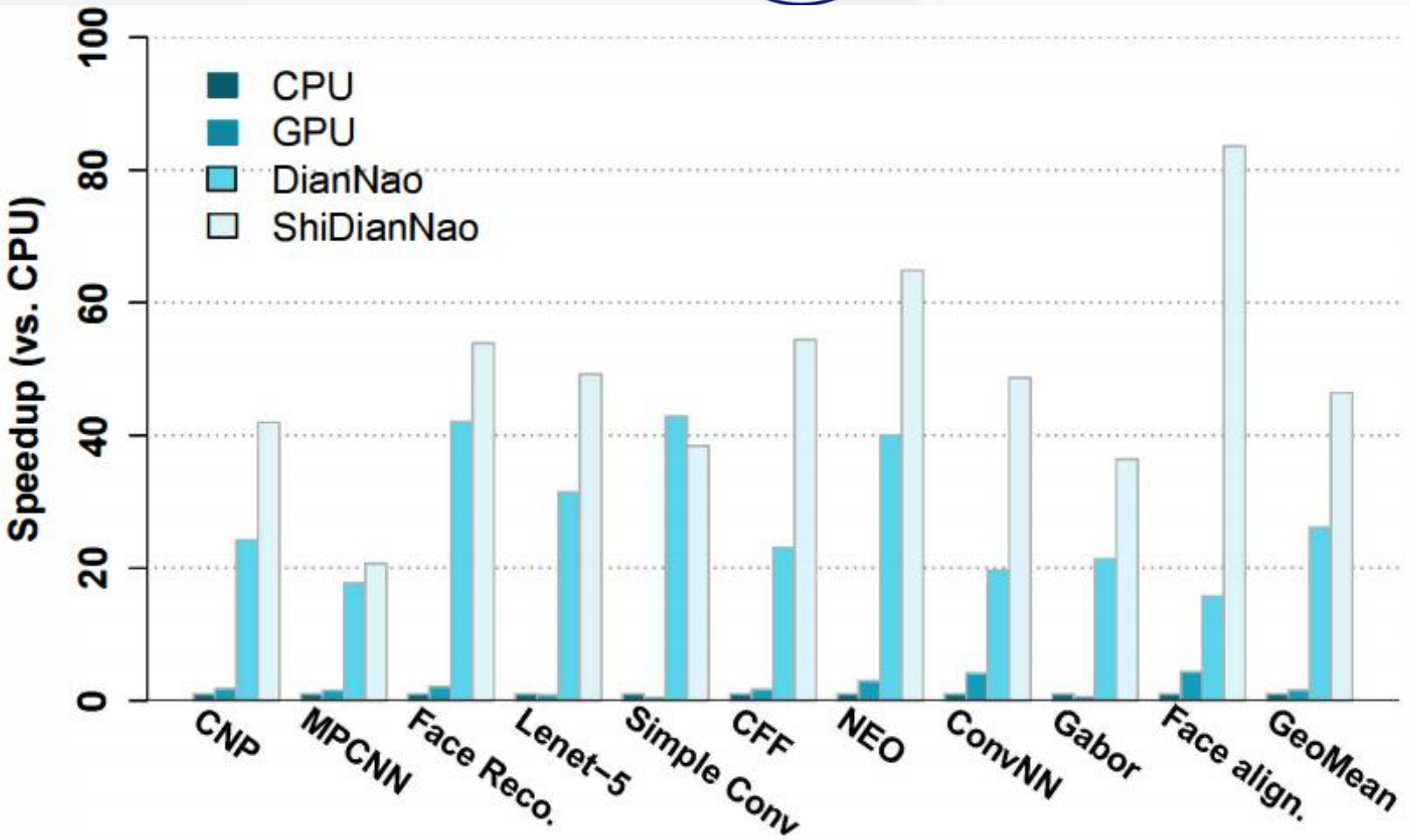


Figure 18: Speedup of GPU, DianNao, and ShiDianNao over the CPU.

ShiDianNao能耗：

同GPU和DianNao相比具有极低能耗

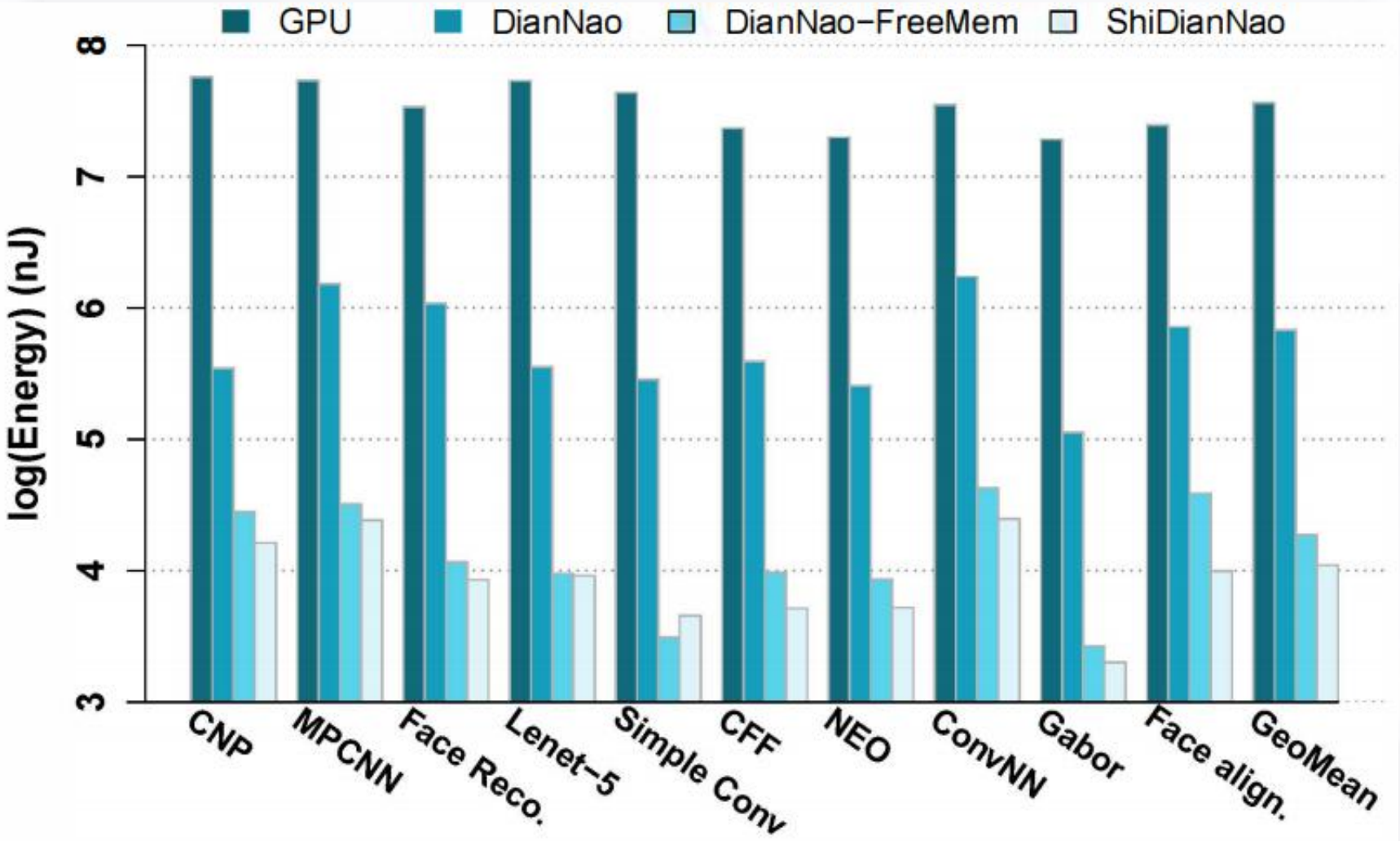


Figure 19: Energy cost of GPU, DianNao, and ShiDianNao.

北京大学
PEKING UNIVERSITY

PuDianNao

支持多种神经网络算法:

k-NN

k-Means

DNN

Linear Regression

SVM

Naive Bayes

Classification Tree

PuDianNao架构：

FUs: 功能单元

InstBuf:指令缓冲区

HotBuf, ColdBuf, OutputBuf：数据缓冲区

Control Module：控制模块

DMA：直接存储器读取

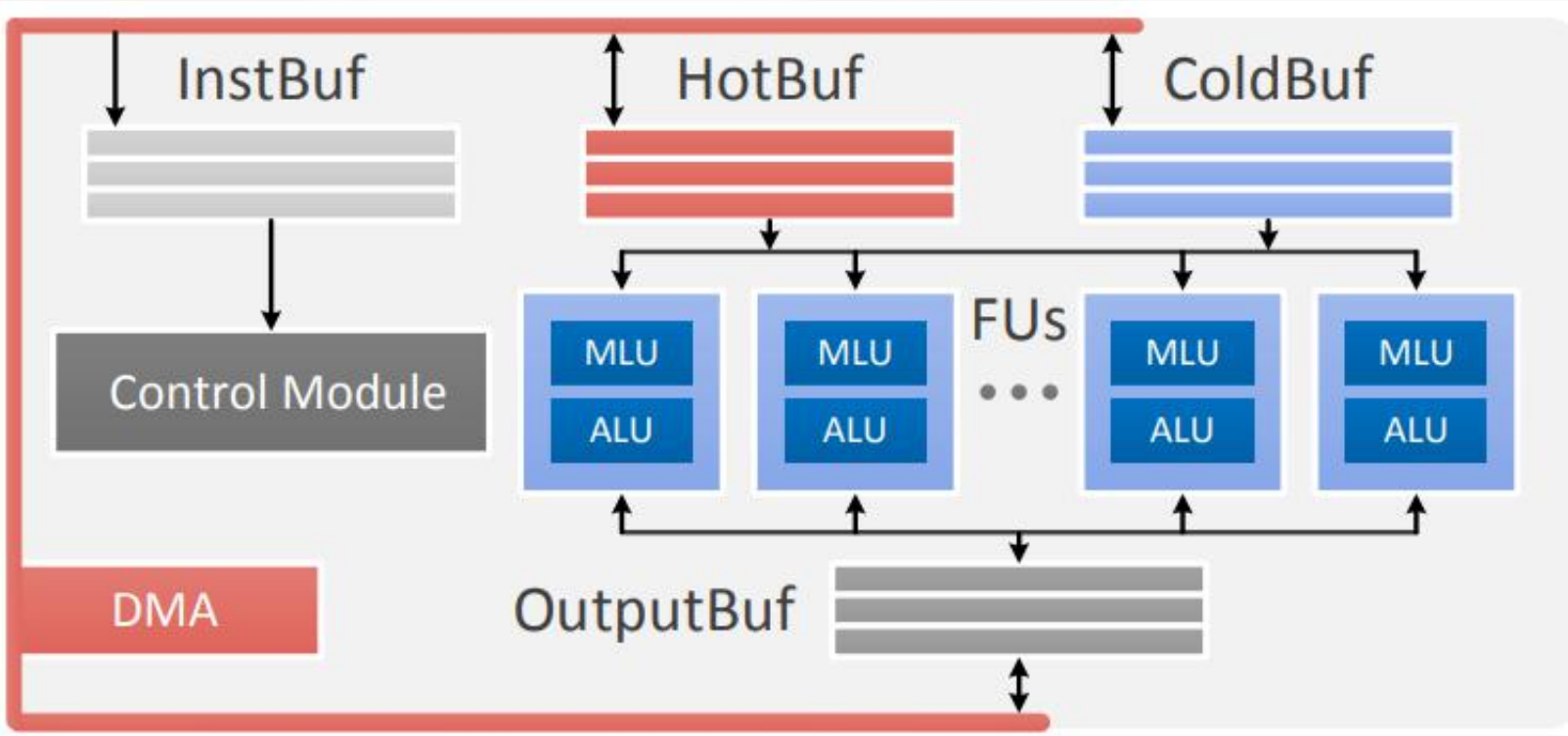


Figure 11: Accelerator architecture of PuDianNao.

MLU:

将多种神经网络算法抽象成少量的运算元素。

增加额外的阶段来满足不同机器学习方法的要求

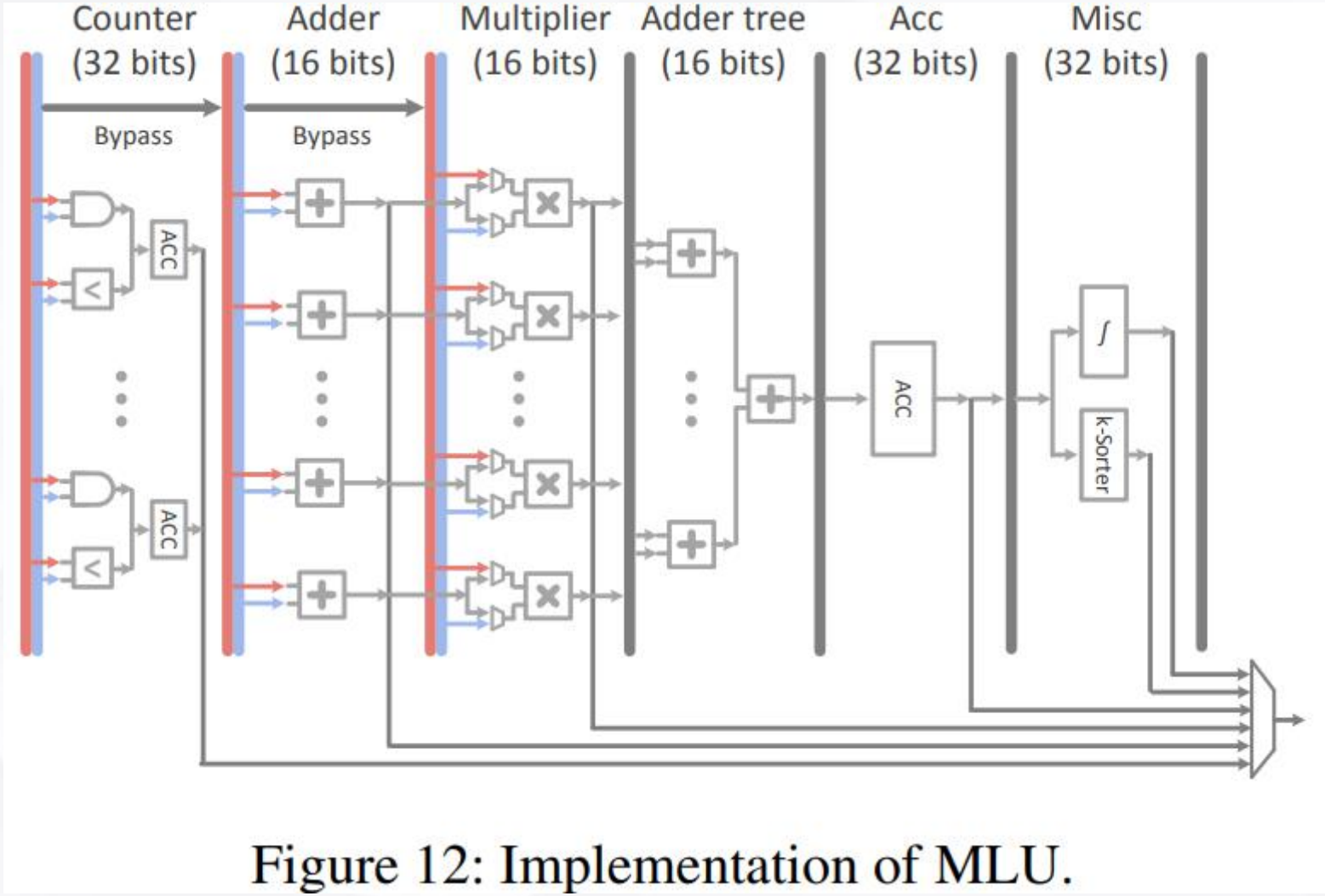


Figure 12: Implementation of MLU.

- 65nm制程
- 面积为3.51 mm^2
- 总功耗596mW
- 平均比GPU快1.2倍
- 节能128.41倍

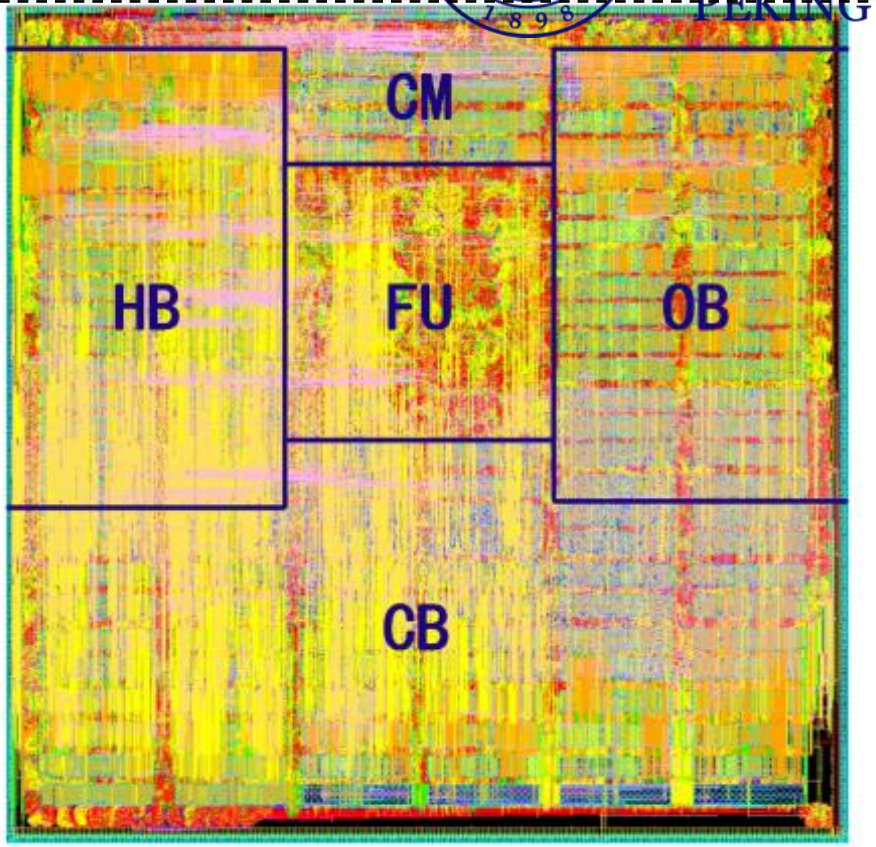


Figure 14: Layout of PuDianNao. CM, FU, HB, CB, and OB stand for Control Module, Functional Unit, HotBuf, ColdBuf, and OutputBuf, respectively.

性能表现：

PuDianNao在许多项训练中优于GPU

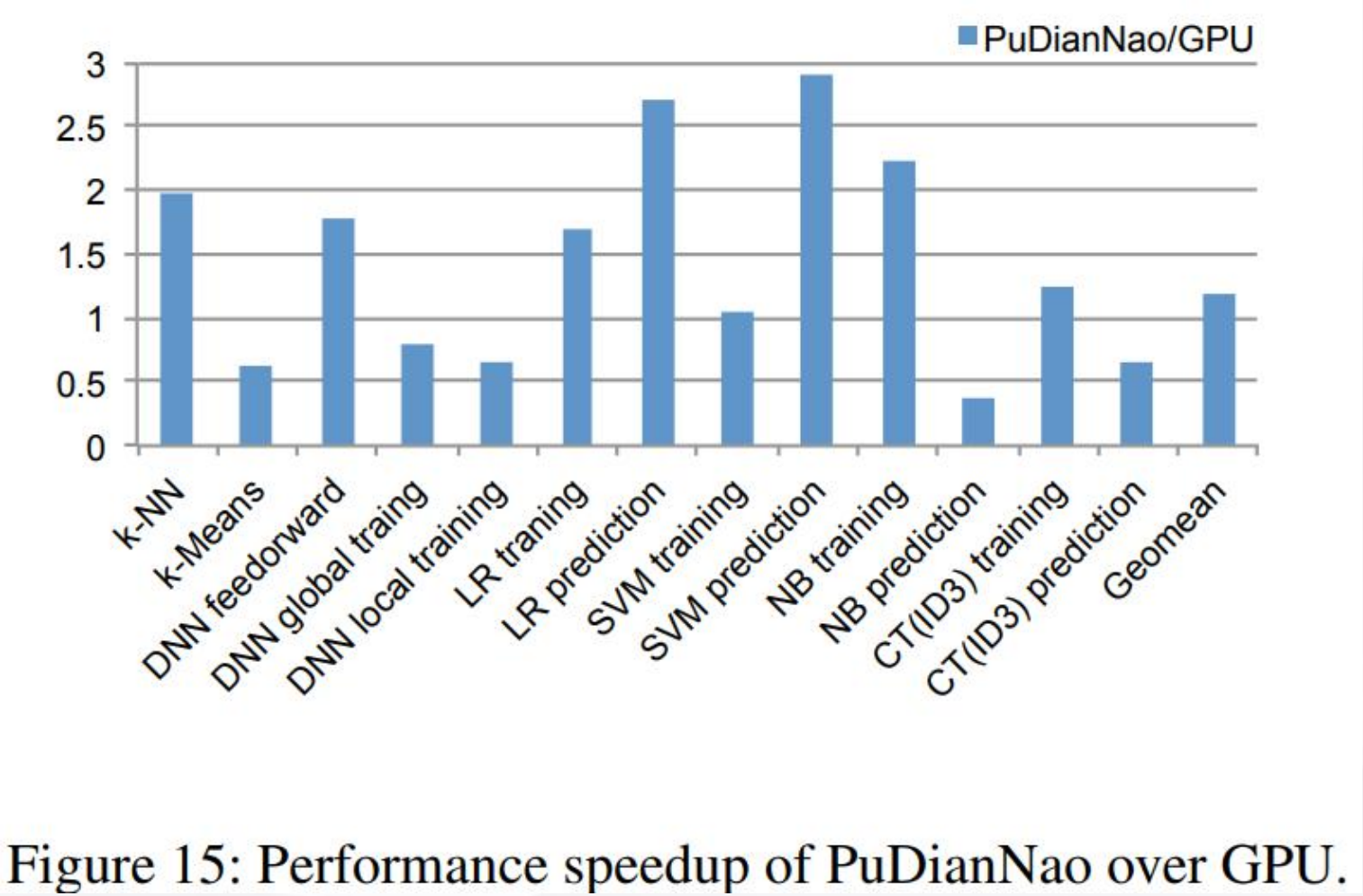


Figure 15: Performance speedup of PuDianNao over GPU.

能耗表现：

PuDianNao耗能明显低于GPU

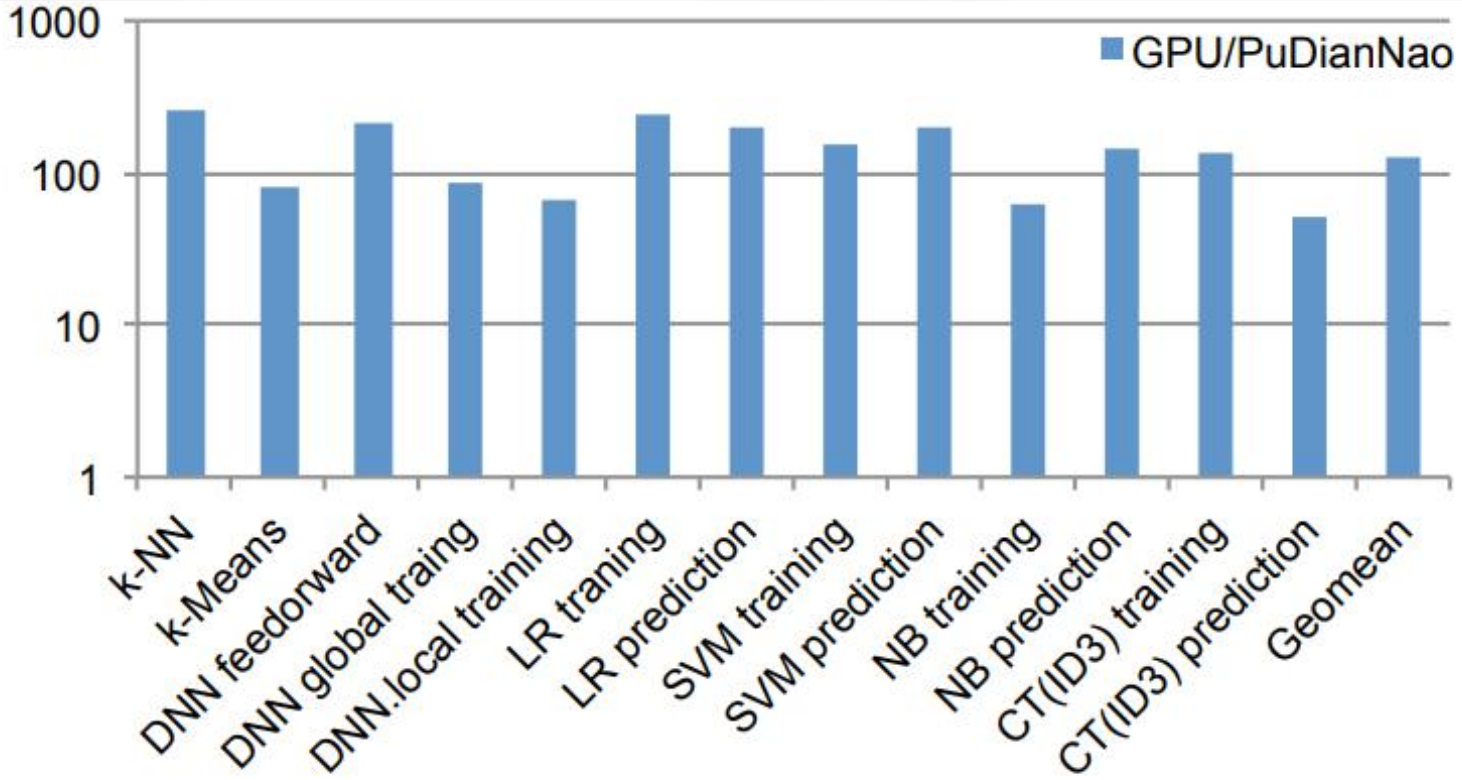


Figure 16: Energy reduction of PuDianNao over GPU.

所有者	型号	应用领域	特点	发布时间	架构	工艺	能效比
寒武纪	1A	多种终端应用	高性能	2016	?	----	----
	1H8	视觉处理终端	低功耗、小面积	2017	?	----	----
	1H16	多种终端应用	通用性、高性能	2017	?	----	----
	1M	多种终端应用	支持更多算法、高性能	2018	?	7nm	5TOPS/W
	MLU100	云端	通用性	2018	MLUv01/Cambricon ISA	16nm	128TOPS/80W
Google	TPU 1.0	面向终端的DNN推理云服务器	高速	2016	----	28nm	92TOPS/40W
	TPU 2.0/Cloud TPU	面向终端的DNN推理/训练云服务器	可支持训练	2017	?	?	180TFLOPS
	TPU 3.0	面向终端的DNN推理/训练云服务器	高性能、高功耗	2018	?	?	100PFLOPS
华为昇腾	Max 910	全栈全场景		2018	达芬奇	7nm	256TFLOPS/350W
	Mini 310	全栈全场景	高速低功耗	2018	达芬奇	12nm	16TFLOPS/8W
	Lite	----	----	未发布	----	----	----
	Tiny	----	----	未发布	----	----	----
	Nano	----	----	未发布	----	----	----
苹果	A12神经网络引擎	?	?	?	?	?	?

北京大学
PEKING UNIVERSITY

智能云服务器芯片 Cambricon MLU100

寒武纪MLU100智能处理卡搭载了寒武纪MLU100云端智能芯片，为云端推理提供强大的运算能力支撑。与传统架构处理器相比，MLU100在处理人工智能任务时可获得巨大的性能和能效提升，是真正适合人工智能的处理器。

MLU100云端智能芯片的等效理论计算能力高达128TOPS，支持4通道64bit ECC DDR4内存，并支持多种容量，可满足各类推理场景的计算和存储需求。

通用性好

通用云端智能处理器，支持各类深度学习技术，支持多模态智能处理（视觉、语音和自然语言处理），应用领域广泛。

高能效

相比GPU处理器，采用了针对深度学习和人工智能应用特点定制的指令集和处理器架构，具有更优的能效比。

支持稀疏化

将稀疏化技术用于智能芯片，通过稀疏化处理，可以达到128TOPS的INT8运算能力。

完善的软件开发环境

寒武纪为MLU100提供了一整套成熟的开发环境Cambricon NeuWare，包括应用开发、功能调试、性能调优等在内的一系列工具。

北京大学
PEKING UNIVERSITY

产品型号	MLU100-C	MLU100-X
核心架构	Cambricon MLUv01	
核心频率	1 GHz	
半精度浮点运算速度（FP16）	16 TFLOPS（关闭稀疏模式时峰值） 64 TFLOPS（打开稀疏模式时峰值）	
整数运算速度 (INT8)	32 TOPS（关闭稀疏模式时峰值） 128 TOPS（打开稀疏模式时峰值）	
内存容量	8GB/16GB	
内存位宽	256-bit	
内存带宽	102.4 GB/s	
系统接口	PCI Express 3.0 x16	
外形	全高全长，单 slot	半高半长，单 slot
是否支持解码	支持解码	不支持解码
功耗	110W	75W
ECC 保护	是	
散热方式	被动散热	

应用场景

视觉智能处理

内置硬件编解码引擎，强大的INT8算力，超低的计算延迟，可以支持最高32路高清视频或480fps高清图像的解码和智能处理。

大数据处理

相比普通CPU/GPU服务器，具有更高的吞吐量和更低的延迟。

北京大学
PEKING UNIVERSITY

Cambricon Neuware Machine Learning Library

- 1. 支持丰富的基本算子，通过基本算子的组合实现多样的机器学习/深度学习算
- 法，从而满足通用性、灵活性、可扩展性需求。目前已支持的基本算子包含
- 以下几类：
- 常见神经网络算子
- 卷积/反卷积
- 池化
- 激活（ReLU、Sigmoid、TANH 等）
- LRN、批规范化
- Softmax
- 全连接
- 矩阵、向量、标量算子
- 矩阵乘
- 张量加、减
- 张量逻辑运算
- Tensor 变换（Crop、Reshape、Slice、Concat 等）
- ROIPooling
- Proposal
- 循环神经网络算子
- LSTM
- BasicRNN、RNN
- SVDF