USp_MV：基于稀疏的矩阵向量乘加速器设计

娄钰明
清华大学电子工程系
中国，北京 louym21@mails.tsinghua.edu.cn

***Abstract*— 稀疏性在神经网络中普遍存在，但是如何利用不规则的稀疏进行加速一直是一个问题。作为一个课程作业，本文参考前人的工作，设计了 USp_MV（Unstructured Sparse matrix-vector multiplication），一个可以利用权重与激活值稀疏性的矩阵乘法加速器，直接根据离线编码的权重与未编码的激活值输出计算结果。并利用 VGG-16 网络中的第三层卷积层验证了设计的合理性并实现了加速。同时尝试了不同稀疏率情况下的计算结果，在很大范围内获得了普遍加速，并分析了其他方法与未来改进方向。**

Keywords—稀疏、矩阵向量乘、神经网络加速器

I. INTRODUCTION

近些年来，神经网络发展极为迅速，已经成为人工智能的主流之一，其最普遍最重要的算子之一是矩阵向量乘（general matrix-vector multiplication, GEMV），卷积等操作经过 img2col 之后都可以转化分解为此。自 AlexNet [1] 首次采用 GPU 训练以后，神经网络的规模迅速上升，给硬件带来了极大挑战。特别是最近大语言模型的发展，一个普通的开源大模型就有十亿以上的参数，消费级显卡难以进行训练。为此，人们基于神经网络中的冗余，提出了一系列神经网络压缩算法，比如神经网络量化是用更少的比特数来表示神经网络的参数与激活值，从而降低数据搬移以及计算过程的代价。剪枝是利用神经网络的稀疏性，将一些不重要的权重与激活值直接置为零，不进行计算，从而降低开销。知识蒸馏是采用大模型的输出来训练小模型，从而获得更小但是性能优于普通训练的模型。低秩分解则是采用竖、横两个长条形的低秩矩阵相乘来近似原来的矩阵，从而降低开销。

虽然一系列神经网络压缩算法具有很好的效果与前景，然而一方面，这些算法往往是有损的，即会降低神经网络的精度；另一方面，这些算法往往依靠特殊的硬件支持才可以发挥效果。而本文提出了一种利用神经网络的稀疏性进行加速的硬件，即 USp_MV。在神经网络中，由于训练时可以引入 L1 或 L2 范数来诱导稀疏，以及卷积神经网络等普遍采用 ReLU 作为激活函数，这就导致神经网络的权重与激活值都有很多 0，即天然稀疏性。同时剪枝也可获得权重与激活值的稀疏性。本文就利用权重与激活值的稀疏性进行加速，以 8 bits 权重与激活值均量化后的模型为对象，设计了支持非结构化稀疏的神经网络矩阵乘法加速器，实现了加速。

目前，针对稀疏已经有不少加速器设计，比如英伟达的 A100 等显卡普遍使用了 2: 4 等 N: M 稀疏，即对矩阵分成大小为 M 的小块，在每块 M 个值中保留 N 个最大的非零值。还有在大语言模型中很多采用稀疏注意力机制，即对 attention map 进行分块，对其中不重要的块不予计算，并通过 cuda 编程实现结合硬件的加速，如 flash attention 的 block sparse[2][3]。然而这些硬件设计往往要求固定的稀疏形式，对于非结构化稀疏的数据需要进行近似，会导致精度损失，无法利用神经网络中由于正则化或者 ReLU 激活函数导致的天然非结构化稀疏性。由于数据稀疏形状的不确定性，直接设计针对非结构化稀疏的数据进行加速的硬件往往是困难的，而 USp_MV 实现了这一点，其在 8bits 精度下是完全无损的！具体而言，本文的主要贡献有：

（1）设计了 USp_MV，可以同时利用权重与激活值的非结构化稀疏性加速矩阵向量乘；

（2）利用 VGG-16 网络中的第三层卷积层验证了设计的合理性并实现了加速；

（3）尝试了不同稀疏率情况下的计算结果，在很大范围内获得了普遍加速；

II. METHOD

本文主要基于 2016 年韩松老师提出的 EIE 架构[4]，采用 compressed sparse column (CSC) 形式对权重进行离线编码，采用 4 个 PE 相互交织进行计算，从而防止冲突。总体设计如下图，具体设计分为 3 个部分进行介绍。

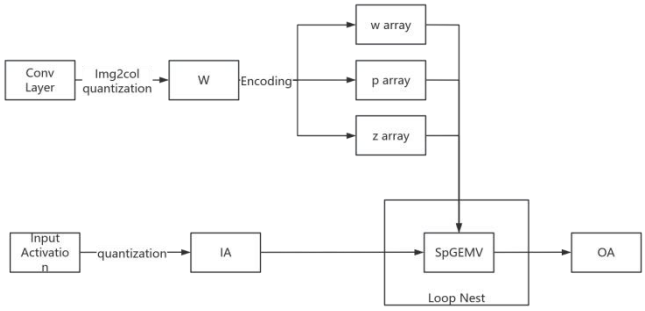

图 1 总体流程图

A. 量化

本文首先将权重与激活值同时均匀量化为 8 bits，此时数值区间为$\{-128, -127, -126, \cdots\cdots, 126, 127\}$。之后再进行计算，加速器中部分和的精度为 32 bits，保证足够的计算精度。同时计算完矩阵乘法之后，还需要对结果进行量化，具体为将结果除以 scale（采用整除 256，即右移 8 位），之后进行截断到量化区间。即：

$$Q(OA(i,j)) = Q(\frac{1}{256}\sum_{k=0}^{M-1} Q(W(i,k)) \times Q(IA(k,j)))$$

其中 $Q(\cdot)$为量化函数，M 为 IA 的高度。

B. 计算

USp_MV 的输入包括权重编码获得的 w、p、z 三个向量以及未经过编码的 IA。由于 USp_MV 仅仅针对神经网络的推理阶段，所以对于权重的编码采用软件离线完成。

图 3 展示了具体的 PE 任务分配与编码方式，采用了 4 个 PE 计算 16×8 与 8×1 的 GEMV，这与设计的初始参数也是一致的。计算时，首先找到向量的非零值，如果没有直接输出 all_finished 信号进行下一轮计算。之后PE_i负责矩阵中所有与 i 模 4 同余的行的计算，根据编码后的权重矩阵找到对应位置，如果第 j 个位置的向量非零，则就将 $W[p[j]: p[j+1]]$的 $p[j+1]-p[j]$个权重填充到对应的长为 4 的队列中，与此同时，还要根据 z 向量与 i 确定这些权重是第几行的。计算时每次将队列首的权重与激活值相乘，累加到对应行号的部分和结果上。当所有 PE 的队列都为空时，就会激活 finished 信号，寻找下一个非零的激活值，直到算完所有激活值，最终输出 all_finished 信号，完成一轮计算。队列的填充与 PE 的对应关系如图 2。

在 loop nest 中采用 OA 不变，即不断更换对应的 W 与 IA，直到得到 OA 的最终结果再更新到主存中，接着清空部分结果并启动下一轮计算。

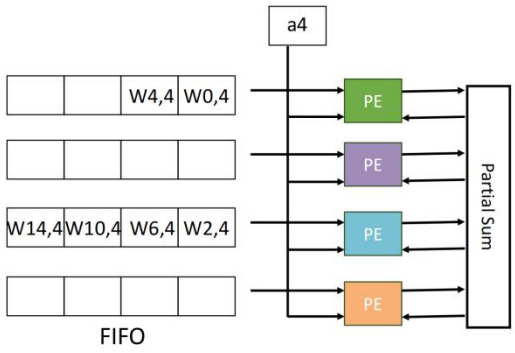

图 2 队列填充与 PE 计算

C. 编码

为了降低计算时的权重数据搬移代价，对权重的编码将是极为重要的。此处介绍 w、p、z 向量如何获得。

本文采用的编码方式为 CSC，将一个 16×8 的稀疏矩阵编码为 4 个不定长的 w、z 向量与 4 个长度固定为 9 的 p 向量，分别对应 4 个 PE。以PE_0为例，则先取出权重的 4 的整数倍行，之后从上到下，从左到右将所有非零权重取出排列为 w 向量。z 向量与 w 向量等长，含义为在每列中对应 w 值与前一个 w 值之间 0 的个数。例如 z 在某位为零，就表示该 w 位于第 0 行或者该 w 列的行数-4 处权重非零。对于 p 向量，$p[j+1]-p[j]$就表示第 j 列PE_0负责的非零权重个数。如此以来可以将权重大大压缩。理论上 p 只需要 6 bits，z 需要 2 bits，w 需要 8 bits。但是为了判断方便，本文对于 p、z、w 都使用了 signed 类型，所以分别为 7、3、8 bits。对于实验数据而言，w 与 z 最长为 23，但是由于硬件的端口是固定的，所以本文不得不将 w 与 z 都补零到长度为 32，此处可以直接硬性截断，但是有损。

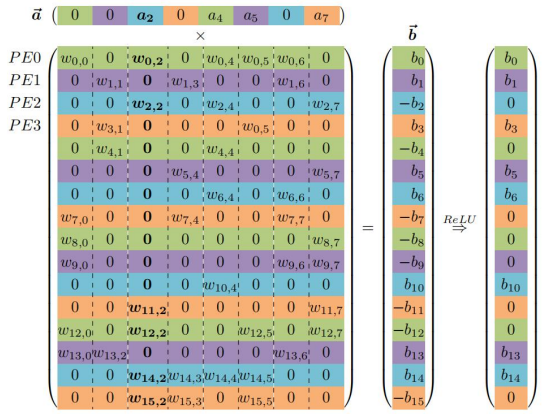

Figure 2. Matrix W and vectors a and b are interleaved over 4 PEs. Elements of the same color are stored in the same PE.

Virtual Weight	$W_{0,0}$	$W_{8,0}$	$W_{12,0}$	$W_{4,1}$	$W_{0,2}$	$W_{12,2}$	$W_{0,4}$	$W_{4,4}$	$W_{0,5}$	$W_{12,5}$	$W_{0,6}$	$W_{8,7}$	$W_{12,7}$
Relative Row Index	0	1	0	1	0	2	0	0	0	2	0	2	0
Column Pointer	0	3	4	6	6	8	10	11	13				

Figure 3. Memory layout for the relative indexed, indirect weighted and interleaved CSC format, corresponding to PE_0 in Figure 2

图 3 编码与 PE 任务分配

III. RESULTS

A. 数据准备

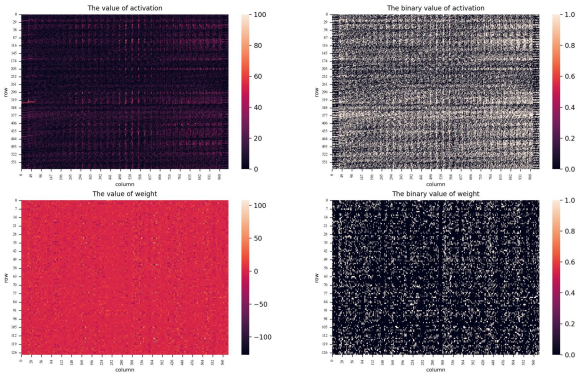

图 4 数据可视化

实验数据采用 8bit 量化的 VGG-16 网络中的第三层卷积，经过 img2col 展开后为 128×576 的权重矩阵与 576×1024 的输入激活值矩阵相乘获得 128×1024 的输出矩阵。图 4 画出了 img2col 展开后的输入激活值以及权重的热图。其中左侧是原始数据，右图是二值化后的数据，可以看到非零值分布较为均匀，并且输入激活值比权重更加稀疏。同时，输入激活值、权重与输出激活值（尚未经过激活函数）的稠密度如表 1，其中 col 后缀表示进行 img2col 展开的数据，也是真正用来计算和求出的数据。

其中所有数据的准备通过 prepare.py 文件实现；数据的可视化与稠密度计算通过 visualization.py 文件实现。

数据	稠密度
IA	46.61%
IA_col	44.52%
W	28.00%
OA	99.98%

表 1 数据稠密度

B. 基础验证

本文采用的基线(baseline)是 lab2 中所给的 PE_array，将 PE 的个数统一为 4，时钟周期统一为 10ns 进行实验，在同样 100%正确率的前提下，USp_MV 用时 123.287ms，而 PE_array 用时 189.727ms，实现了 1.54×的加速。

具体的参数对比如下：

指标	USp_MV	baseline
算力	0.4 Gops	0.4 Gops
IA 带宽需求	6.4 Gbps	**3.2 Gbps**
W 带宽需求	166 Gbp	**0.8 Gbps**
OA 带宽需求	12.8 Gbps	**3.2 Gbps**
IA 片上 buffer	64 bits	**32 bits**
W 片上 buffer	1660 bits	**8 bits**
OA 片上 buffer	128 bits	**32 bits**
IA 片外访问总 bit 数	**18 MB**	72 MB[1]
W 片外访问总 bit 数	**7.28 MB**	18 MB
OA 片外写入总 bit 数	1 MB	1 MB
计算完成用时	**123.287ms**	189.727ms
等效阵列利用率	**153.093%**	99.48%
最高时钟频率	216.64MHz	**239.46MHz**

表 2 参数对比

可以看到 USp_MV 利用了稀疏性产生了大于 1 的等效利用率。同时其对主存的访存次数也降低了 2~4 倍。代价是更高的带宽需求、更大的面积[2]以及更大的 buffer。由于实验的主存是软件控制的，如果采用真正的 DRAM，则 USp_MV 加速将会更多。

C. 不同稀疏度

这部分本文考虑在权重和激活值具有不同稀疏性的情况下对比 USp_MV 的加速效果。为了方便，本文采用了更小的数据：随机生成的 128×32 的权重矩阵与 32×32 的输入激活值矩阵相乘获得 128×32 的输出矩阵。此

[1] 注：1B=8 bits，1GB=1024MB=1024*1024KB=1024*1024*1024B。
[2] 注：面积见附录。

时测试得到 baseline 为 358.430 μs，而 USp_MV 结果如下：

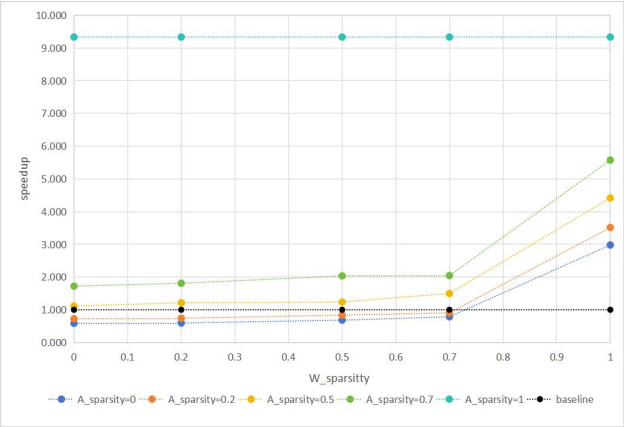

图 5 不同稀疏度下的加速效果

可以看到在很大范围内 USp_MV 都可以实现加速，并且加速比最高可以接近 10 倍。即使面对完全稠密的矩阵，也仅仅慢了不到一倍，效果还是比较明显的。另外，可以看到对激活值的稀疏对加速的影响更大，这是比较好解释的，因为计算时是先找的非零激活值，之后才看对应的权重。而神经网络中激活值的稀疏会比权重的稀疏更好获得，因为前者只需要一个 ReLU 激活函数，而后者往往需要更改训练的 loss 进行重新训练或进行剪枝降低精度。当然，如果真的是 W 更稀疏，可以更改 mapping，使输入矩阵是 A，输入向量时 W，但是需要对 A 进行离线编码。

D. 不同的 PE 个数

时间原因，这部分没有进行实验，仅仅做理论分析。USp_MV 可以加速的本质一方面来源于跳过了为零的激活值，不进行计算；另一方面源自权重的稀疏，即 PE 的队列空了之后就可以直接进行下一轮计算。而增加 PE 个数之后，如果每个 PE 负责的依然是 4 行，则会导致很多 PE 等待某个 PE 的情况，降低阵列利用率，不利于加速。而如果使每个 PE 负责的行数相应减少，比如 8 个 PE 每个负责 2 行。一方面会导致计算时间变短，控制反而成为瓶颈，降低利用率；另一方面由于 PE 等待的周期上限减少，反而会提高利用率。两方面相互权衡，可以找到一个最佳的 PE 个数。

IV. DISCUSSION

A. 编码

在 USp_MV 的编码中，p 与 z 的位宽以及 w 与 z 的长度均是考虑的最坏的情况。然而如实验发现，w 与 z 的长度最多只有 23 个，但是表示范围却达到了 32。在某种意义上可以直接截断，由于激活值的稀疏性，最后未必会对计算结果有太大影响，有点类似于结构化稀疏。同时目前的实现是采用补 0 的方式，但是也可以不补 0，在 tb 文件中做简单修改，这样就真正实现了对权重的压缩，减少了数据搬移开销，但是因为需要考虑最坏的情况，所以无法降低带宽需求。之后本文考虑其余一些编码方法。

首先是与 CSC 相对的 CSR，即对每一行进行编码。二者的计算其实差不多，可以理解为把矩阵做了转置，差别不大。这两种编码的好处是不需要遍历，易于寻址。

之后是 bitmask，即存储矩阵同等大小的 1 bit mask 矩阵与所有非零值。这样的好处是编码简单，但是寻址很困难。在硬件设计时每次都要遍历 mask 到下一个 1。

最后是游程编码，即记录所有非零值与前面的零个数。类似于只有一行的 CSR。如果矩阵比较稀疏，中间可能会连续有很多 0，游程编码就需要更多 bit 的空间来存储游程，或者需要插入 0，导致增加了不必要的计算。

总体而言，在 USp_MV 的实现中，CSC 是比较合适的。

B. 三种计算对比

本次实验除了 USp_MV 的实现方式即 interleave 外，还有其他两种，这里试做一个对比。

模式一采用类似外积的方式，直接实现 GEMM。会需要更大的 buffer 以及更高的带宽。同时由于会存在哈希冲突不太好并行，如果涉及很多个 MAC 计算，虽然可以提高并行，但是会导致利用率下降。

模式二则是用一个周期匹配，实现 GEVV。可以使用加法树实现，而且不存在冲突，易于并行与扩展。

总体而言，模式一的粒度更大，但是也带来了更多问题；模式二粒度最小，并且易于扩展；模式三介于二者之

间，利用同余的方式避免了冲突，很巧妙，但是没有模式二那样易于扩展，但是比模式一更高效一点。

V. Conclusion

本文设计了 USp_MV，一个可以同时利用权重与激活值稀疏的矩阵乘法加速器，直接根据离线编码的权重与未编码的激活值输出计算结果。并研究了不同的权重、激活值稀疏以及不同的 PE 个数对最终效果的影响。同时 USp_MV 在设计之初就考虑到了一些扩展的问题，可以很方便的集成。

未来，希望可以根据不同的数据特点设计不同稀疏模式的硬件。比如对于稠密矩阵直接计算，可以考虑在器件中加一个信号统计当前数据稀疏的程度，以此来决定下一轮计算采用的计算模式，从而实现更好的加速。同时还可以结合神经网络压缩方法，软硬件协同设计实现更好的加速。

本文仅作为一个课程作业，不用于发表！

VI. 附录

A. 基础验证截图

```
run all
wrong num:            0
$finish called at time : 123287460 ns
run: Time (s): cpu = 00:01:11 ; elapsed = 00:02:35 . Memory (MB): peak = 1774.121 ; gain = 43.820
```

图 6 USp_MV 结果

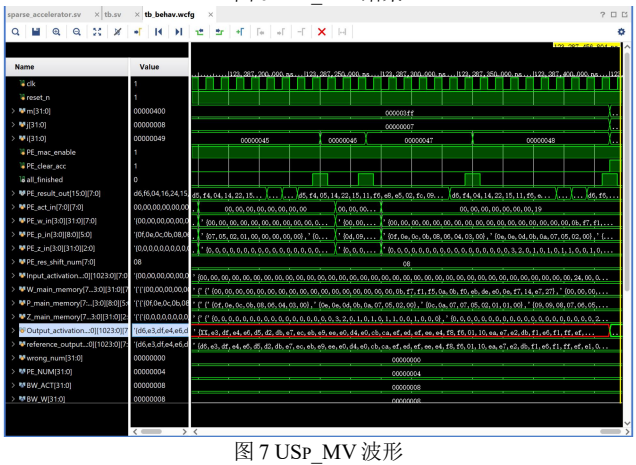

图 7 USp_MV 波形

图 8 USp_MV 时序分析

Resource	Utilization	Available	Utilization %
LUT	1360	78600	1.73
FF	327	157200	0.21
IO	413	250	165.20

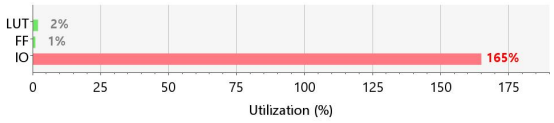

图 9 USp_MV 面积情况

```
wrong num:            0
$finish called at time : 189726750 ns
run: Time (s): cpu = 00:01:04 ; elapsed = 00:01:11 . Memory (MB): peak = 1755.422 ; gain = 0.000
```

图 10 PE_ARRAY 结果

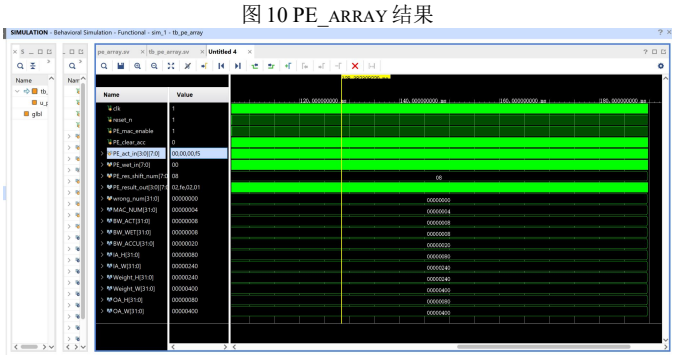

图 11 PE_ARRAY 波形

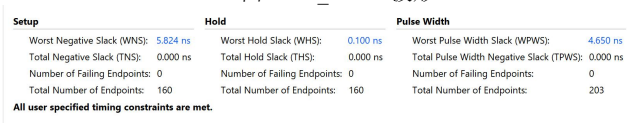

图 12 PE_ARRAY 时序分析

Resource	Utilization	Available	Utilization %
LUT	875	78600	1.11
FF	202	157200	0.13
IO	84	250	33.60

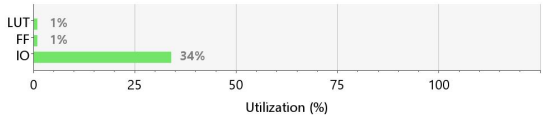

图 13 PE_ARRAY 面积情况

B. 文件清单

数据准备：

visualization.py(绘图)、prepare.py(编码与转为二进制)、prepare_small_scale.py(小数据 debug 的 prepare 文件)、diverse_sparsity.py(生成指定稀疏度的数据)

实验：

USp_MV(一个 source 一个约束一个 tb)、baseline(同上)

[1] Krizhevsky, A., Sutskever, I., and Hinton, G. E. ImageNet classification with deep convolutional neural networks. In NIPS, pp. 1106–1114, 2012.

[2] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems, 35:16344–16359, 2022.

[3] Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. arXiv preprint arXiv:2307.08691, 2023.

[4] S. Han et al., “EIE: Efficient inference engine on compressed deep neural network,” in Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA), Jun. 2016, pp. 243–254.