HAIMA: A Hybrid SRAM and DRAM Accelerator-in-Memory Architecture for Transformer

汇报人：季贊杰

1.摘要

● 背景：在Transformer的矩阵与矩阵的乘法计算（$MMMO$）当中，大量的数据移动导致了计算延迟。目前有效的解决方案是将算术逻辑单元（ALU）嵌入到存储阵列当中，从而实现存储器体系结构中的加速器（AIM）。但是目前这个方向的工作没有考虑到Transformer层之间的并行性和资源需求的异构性，这导致了计算的延迟与资源的低利用率。

●目的：旨在提高$MMMO$计算的并行效率，并从端到端的角度减少Transformer推理延迟

●本文提出了名叫：$HAIMA$的一种混合加速结构，这是混合$AIMA$和Transformer的并行数据流，他利用了静态随机存储器（$SRAM$）和动态随机存储器（$DRAM$）之间的合作来加速不同的$MMMO$。与最先进的Newton和TransPIM相比，实现了1.4-1.5倍的加速，并解决了基于$DRAM$的$AIMA$执行轻量级$MMMO$资源利用不足的问题。

2.本文的主要贡献

- 提出了一种名为$HAIMA$的混合$AIMA$，它利用了$SRAM$和$DRAM$的硬件特性。基于$DRAM$的$AIMA$执行大规模$MMMO$。基于$SRAM$的$AIMA$充当基于$DRAM$的$AIMA$和主机之间的过滤器，并在数据移动期间处理轻量级$MMMO$。
- 提出了一种用于Transformer计算的并行数据流，该数据流考虑了不同层之间的不同并行性和资源需求。同时，数据流考虑了系统中Transformer的计算过程。
- 开发了支持$MMMO$高效计算的拟议架构和数据流的基于$SRAM$的$AIMA$（$SRAM-CIM$）和基于$DRAM$的$AIMA(DRAM-CNM)$。

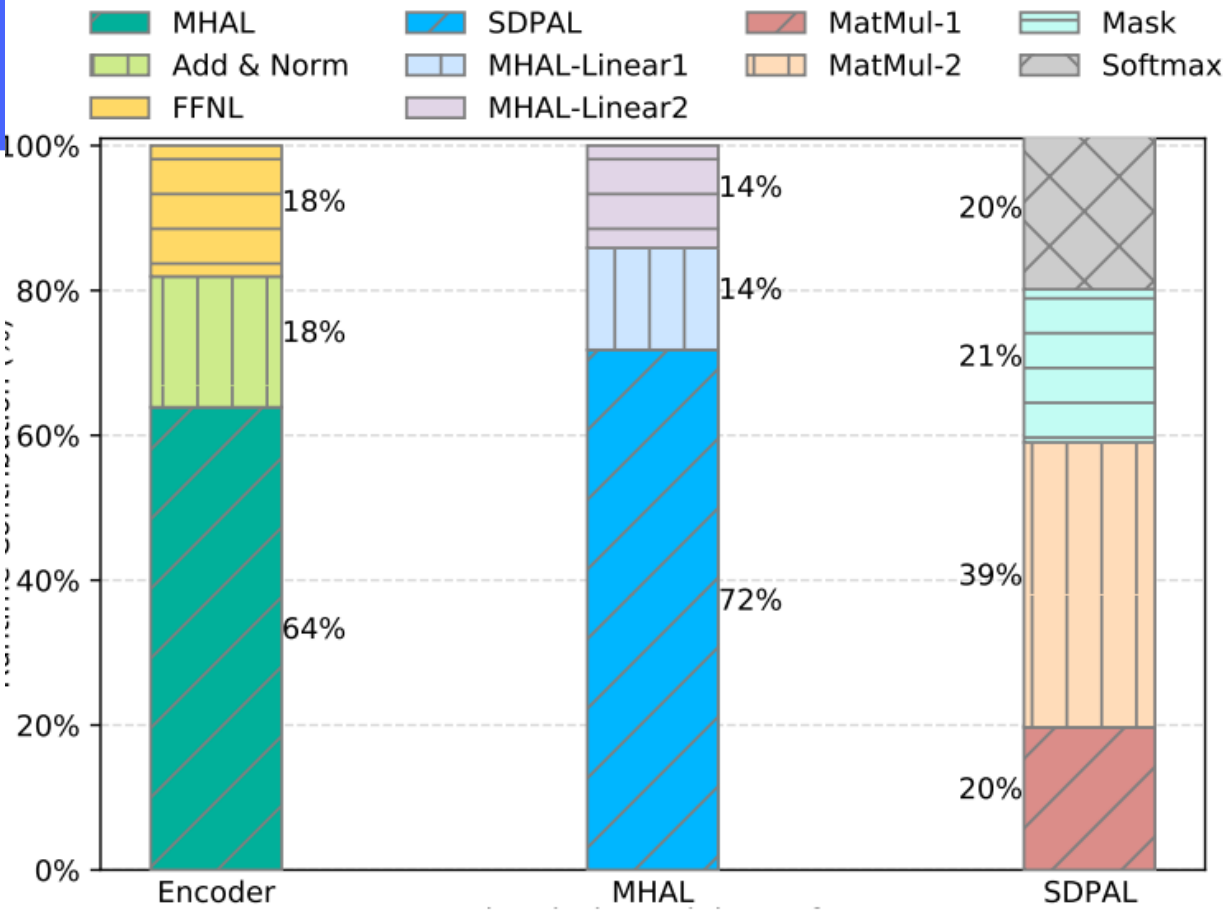


图1 Runtime Breakdown of Encoder

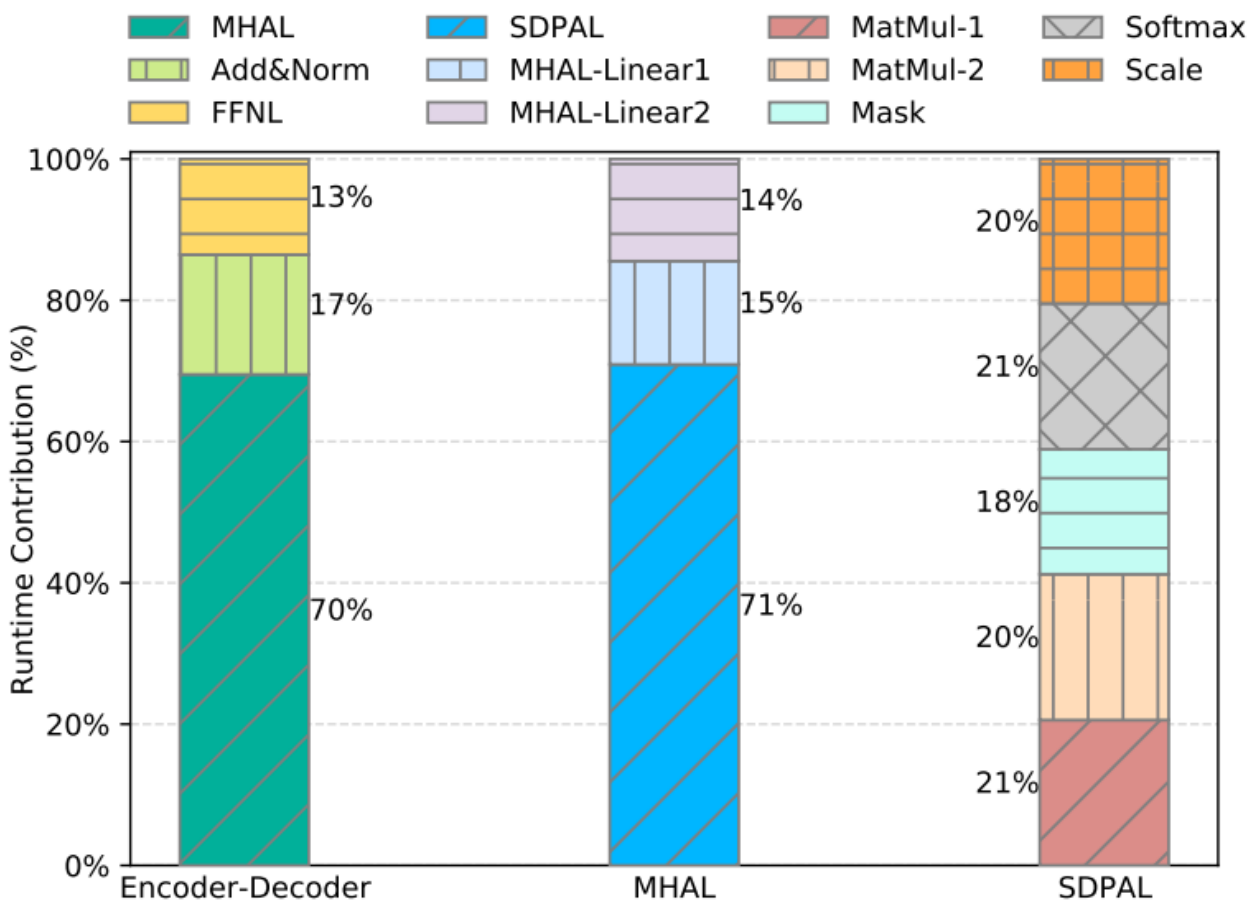


图2 Runtime Breakdown of Encoder-Decoder

图1和图2显示了$BERT$和$BART$模型运算时间时分解。如图所示，$MHAL$和$FFNL$是Transformer的两个关键模块，并主导计算延迟。造成这种现象的主要原因是在这两个模块中存在大量的$MMMO$计算。

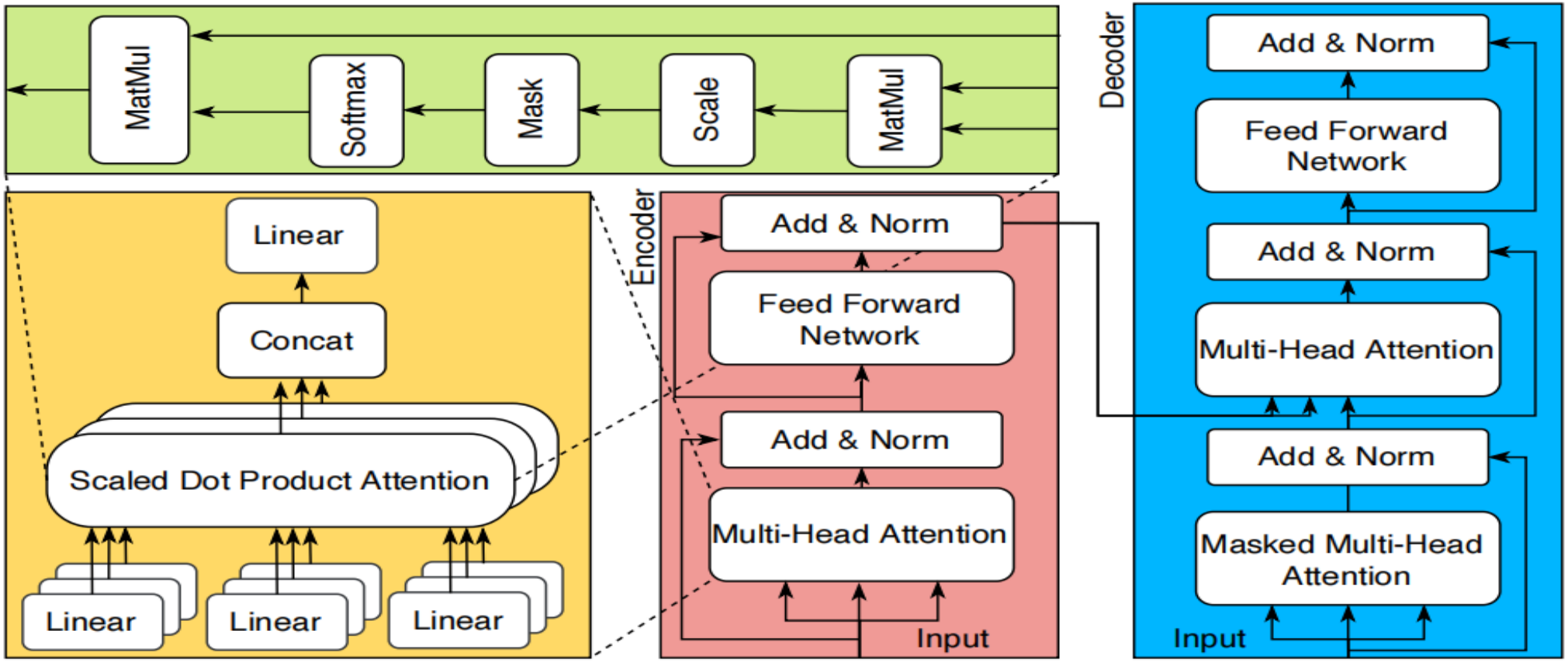


图3 Structure of Encoder/Decoder

对于具有N个标记和M个隐藏维度的序列，具有3个权重矩阵的3个全连接层（FCL），即$W^Q \in \mathbb{R}^{M*M}, W^K \in \mathbb{R}^{M*M}, W^V \in \mathbb{R}^{M*M}$，得到一个位置嵌入矩阵$X \in \mathbb{R}^{N*M}$，并分别生成Query($Q \in \mathbb{R}^{N*M}$)、Key($K \in \mathbb{R}^{N*M}$)和Value($V \in \mathbb{R}^{N*M}$) 矩阵。

假设MHAL中有H个头，那么Q,K,V,按列划分为H部分，即生成每个头部的权重子矩阵。我使用Q_h,K_h,V_h，其h∈[0:h−1]表示Q,K,V的权重子矩。

SDPAL主要由3个$MMMO$组成：$Y_h = Q_h K_h^T, S_h = Sotfmax(Q_h K_h^T/\sqrt{M}), A_h = S_h V_h$。最后一层FNL的权重矩阵用$W^L$表示，最后MHAL产生的输出$Z = AW^L + B^L, B^L \in \mathbb{R}^{1*M}$是偏置向量。

数据移动大约占计算延迟的60%。此外像$MMMO$这样的单指令多数据（$SIMD$）的操作类型正是$AIMA$能够有效处理的计算类型。Transformer中的所有$MMMO$在并行性以及计算和内存的资源需求方面都表现出显著的异构性。文章描述了Transformer中的4种$MMMO$类型，并描述了它们相应的并行策略。

（i）用于获得Q、K、V的$MMMO$:Q、K、V（分别标记为$MMMO-Q/K/V$）

（ii）用于获得Y和A的$MMMO$（分别标记为$MMMO-Y/A$）

（iii）用于获得Z的$MMMO$（标记为$MMMO-Z$）

（iv）$FFNL$中的$MMMO$（标记为$MMMO-FFNL$）

（i）$MMMO-Q/K/V$:采用的并行策略是矩阵分割乘法（**MPM**）。具体而言，W^Q, W^K, W^V分别按列等分为H个子矩阵。每个头部的线性投影权重子矩阵可以表示为$W_h^Q \in \mathbb{R}^{M*(M/H)}, W_h^K \in \mathbb{R}^{M*(M/H)}, W_h^V \in \mathbb{R}^{M*(M/H)}$，因此$Q_h = XW_h^Q + B_h^Q$，$K_h = XW_h^K + B_h^K$，$V_h = XW_h^V + B_h^V$ 其中$B_h^{Q/K/V} \in \mathbb{R}^{1*(M/H)}$是偏置向量。

（ii）$MMMO-Y/A$:目前大多数$MMMO-Y/A$工作采用的并行策略只是独立计算H个头中的Y_h和A_h，并没有进一步探索一个头内$MMMO$的并行性。由于$AIMA$无法以具有成本效益的方式处理像$Sotfmax(\cdot)$这样相对复杂的操作。因此，为了得到A_h，首先将Y_h发送到主机以完成S_h=$Sotfmax(Q_h K_h^T)$的计算，然后将结果矩阵S_h发送回$AIMA$以完成A_h=S_h V_h的计算。然而在计算S_h以及主机和片外存储器之间的数据通信期间 V_h只能处于等待状态。

（iii）$MMMO-Z$：与$MMMO-Q/K/V$相比，虽然$MMMO-Z$不具有多个头之间的并行性，但其计算规模仅为它们的三分之一。也就是说$MMMO-Z$需要更少的计算和内存资源。因此，如果$MMMO-Z$仍然在具有足够资源的片外存储器上执行，则将显著降低资源利用率，并导致系统不必要的能耗。MPM并行策略可以加速$MMMO-Z$的计算。

（iv）$MMMO-FFNL$:$MMMO-FFNL$在多个头之间没有并行性，但其计算负载规模较大，因此比其他$MMMO$需要更多的计算和内存资源。现有工作通常采用MPM并行策略和基于DRAM的AIMA来加快MMMO-FFNL的计算速度。

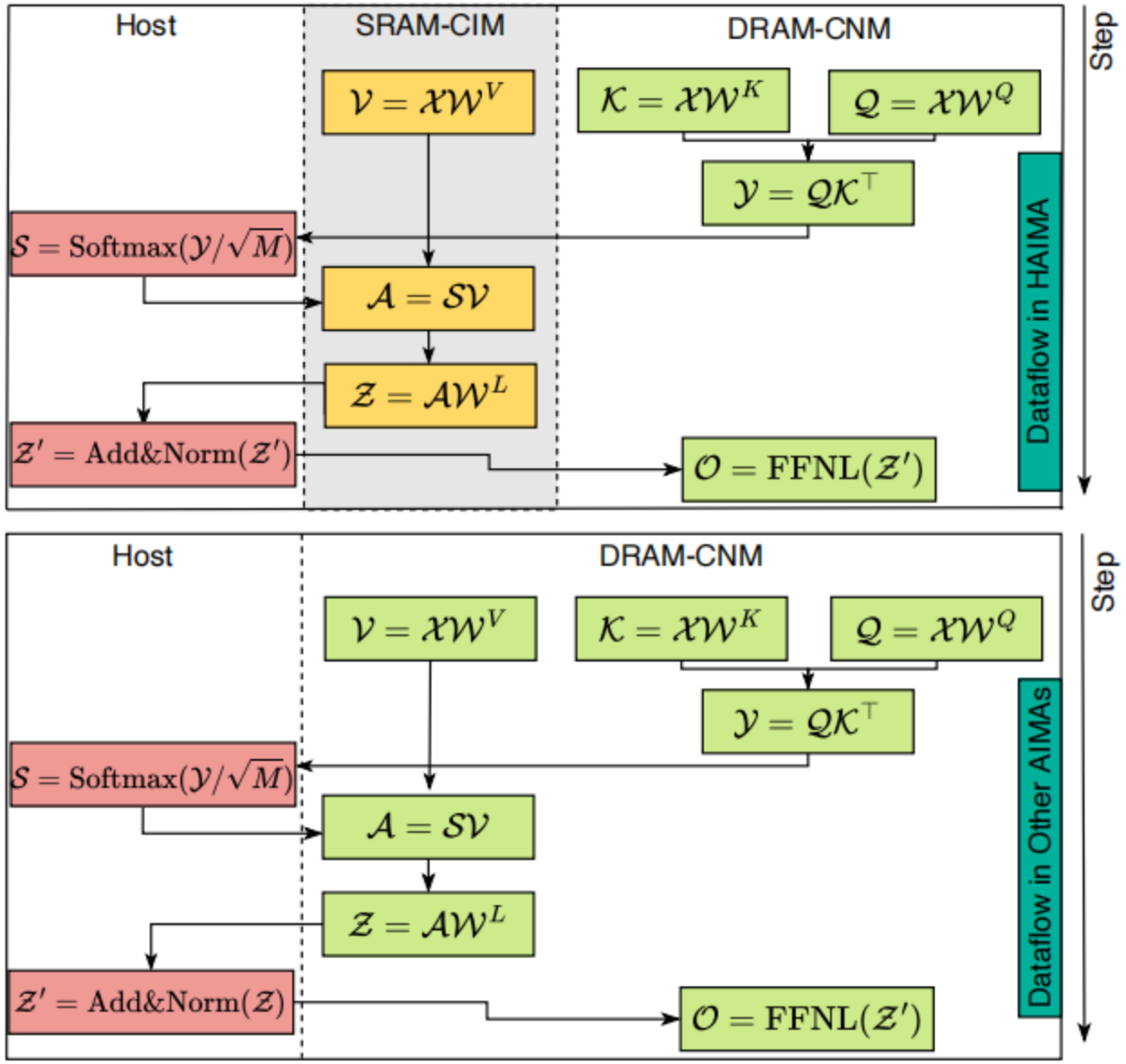
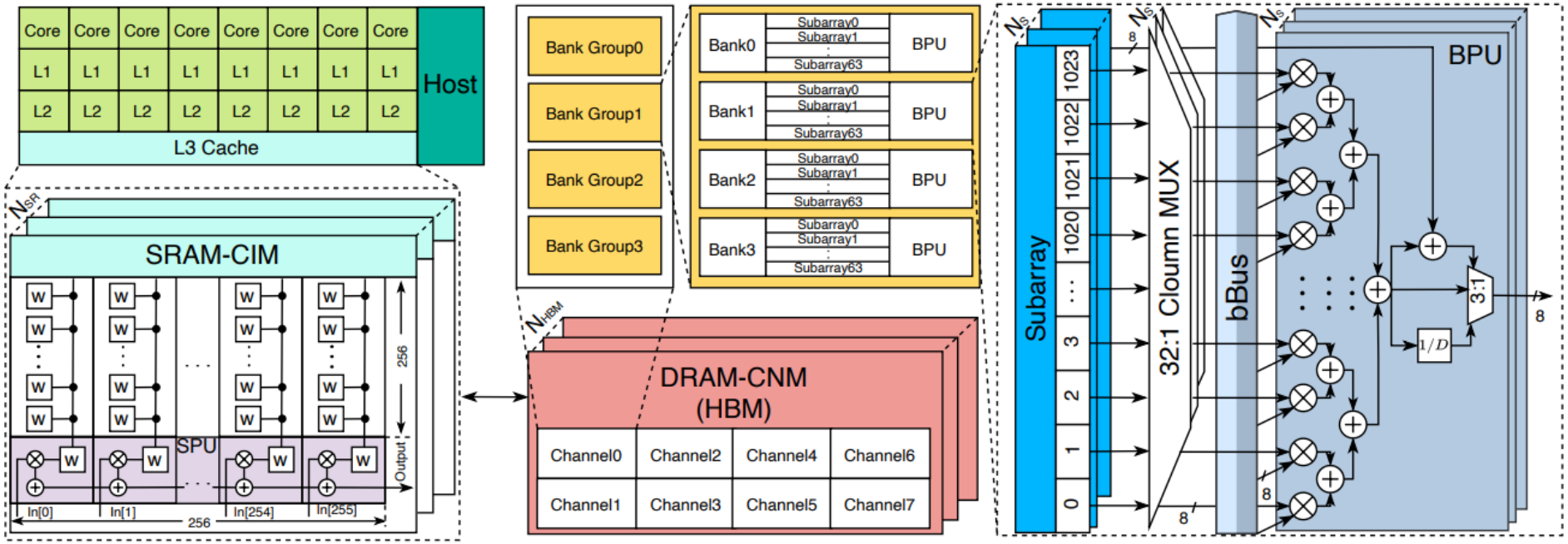


图4 Dataflow Comparison between *AIMAs*

与现有工作不同，本文利用不同$MMMO$中的头间并行性和头内并行性来加速Transformer的计算。此外，在所提出的并行数据流中考虑了不同$MMMO$的资源需求。具体而言，$SRAM-CIM$处理小型$MMMO$，而$DRAM-CNM$处理大型$MMMO$。图4显示了$HAIMA$和其他基于$DRAM$的$AIMA$之间的数据流比较。如图4顶部所示，$HAIMA$数据流利用$SRAM-CIM$有限的计算能力来完成$MMMO-V/A/Z$的计算。因此，可以在S的移动期间获得A。

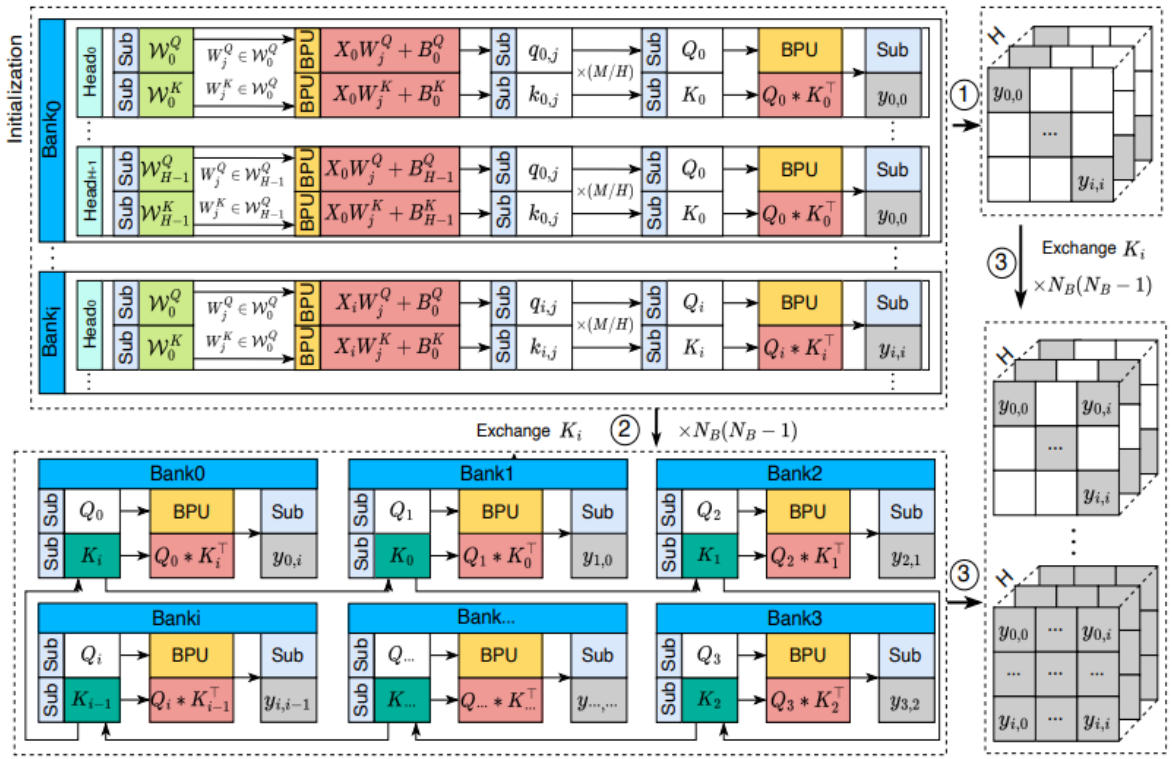
这种机制不仅可以实现通信和计算的重叠，减少不必要的V等待等待时间，还可以解决$DRAM-CNM$执行轻量级$MMMO$时资源利用率低、能耗高的问题。

HAIMA数据流



- 数据流底层架构基于HAIMA。为了充分利用DRAM-CNM上存储的并行特性，采用数据并行和基于令牌的数据映射的联合策略来提高内存数据局部性。而利用脉动阵列的思想在SRAM-CIM上构建高效的轻量级MMMO架构。
- DRAM-CNM执行MMMO-Q/K/Y/FFNL，而SRAM-CIM执行MMMO-V/A/Z。

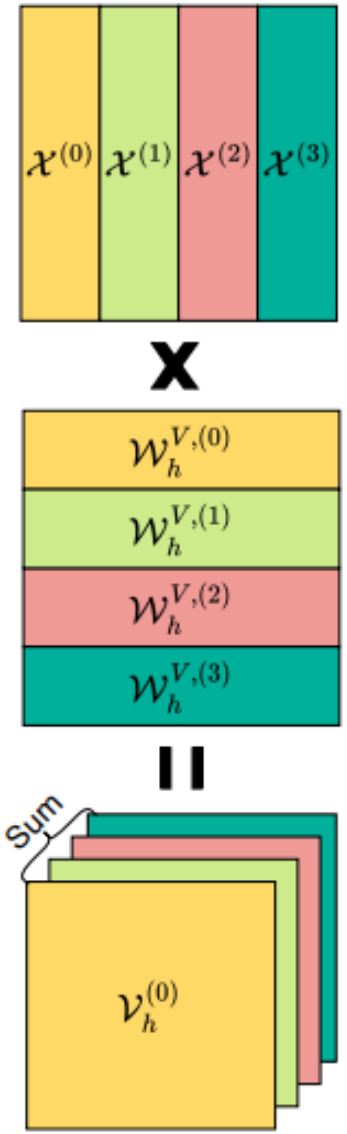
DRAM-CNM中的数据流



(a) Dataflow in DRAM-CNM

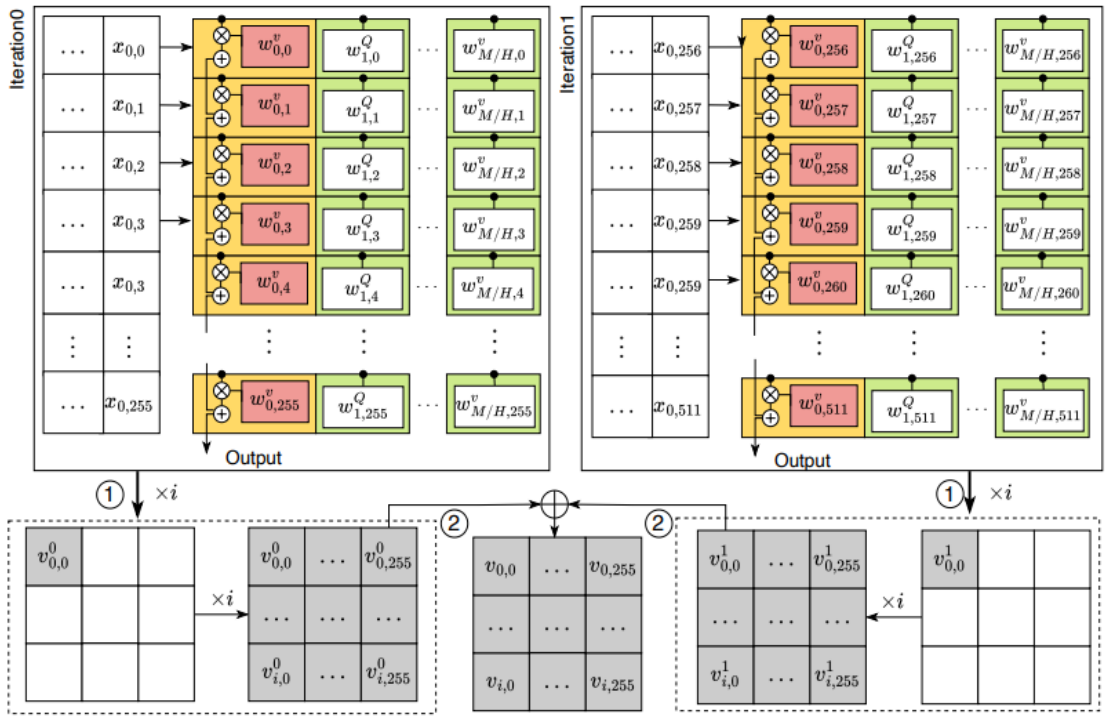
1) MMMO-Q/K/Y：将权重矩阵W^Q和W^K加载到每个内存中。同时W^Q和W^K按列平均分为H个子矩阵。每个头的权重子矩阵单独存储在内存的子数组里。由于访问子数组一次只能检索子数组中的一行数据，因此，对于一个头，X_i (i $\in$ [0, N − 1])的有关向量$Q_i \in R^{1\times M/N}$和$K_i \in R^{1\times M/N}$可以通过分别访问M/H行获得。得到Q_i和K_i后，可以立即进行$y_{i,i}=Q_iK_i^T$的计算。受益于内存的并行处理能力，可以获得所有头的注意力分数。此外，为了获得所有代币中的分数内存以环形方式交换存储的K_i。在每次交换迭代中，最初存储在内存中的Q_i乘以交换后的K_i，从而得到X_i和$X_{i'}$之间的注意力分数。经过$N_B(N_B-1)$次交换迭代后，最终可以得到所有头中N个代币中的注意力分数，即Y。接下来，Y将被发送到主机且S=Softmax(Y)的计算将由主机执行。

2) MMMO-FFNL:矩阵乘法实现了MPM的形式的并行计算。在获得主机/SRAM-CIM发送的矩阵Z' $\in R^{N\times M}$后，DRAM-CNM以MPM的形式完成MMO-FFNL的计算。Z'和FFNL第一权重的矩阵按行和列分别平均分为N_B个子矩阵，并存储在N_B个内存中。这样就并行完成了子矩阵之间的计算，生成了N_B个中间结果矩阵。FFNL的第一层权重矩阵可以通过对所有中间矩阵求和得到。重复上述过程两次得到O。



SRAM-CIM中的数据流

1) MMMO-V:将权重矩阵 W^V 加载到SRAM-CIM中。首先将W^V按列分为H个子矩阵，然后将子矩阵W_{h}^V进一步按行分为M/256个子矩阵，即$W_{\mathrm{h}}^{V,(0)}$和$W_{\mathrm{h}}^{V,(1)}$。输入矩阵X并按列分为M/256个子矩阵，即$X_{\mathrm{h}}^{(0)}$和$X_{\mathrm{h}}^{(1)}$。然后，输入和部分和都通过一个二维SRAM处理单元(SPU)同时执行流水线并行MMMO。虽然一个SPU完成$V_{\mathrm{h}} = XW_{\mathrm{h}}^V$的延迟是M/256个SPU完成$V_{\mathrm{h}} = XW_{\mathrm{h}}^V$的M/256倍，但这并不影响HAIMA完成A=SV的性能。因为在A到达SRAM-CIM前，MMMO-V的计算已经完成。由于面积和功耗限制，SRAM-CIM中没有添加支持矩阵加法的ALU。因此中间结果矩阵$V_{\mathrm{h}}^{(0)}$和$V_{\mathrm{h}}^{(1)}$将被传入主机中求和，然后最终结果矩阵V_{h}将被写入SRAM-CIM中。V_{h}的存储格式与W_{h}^V相同，以方便后续MMMO-A的计算。



(c) Dataflow in SRAM-CIM

2) MMMO-A/Z:在S_h计算完成后，主机将S_h发送到保留V_h的SRAM-CIM。类似地，S_h按列分为M/256个子矩阵，然后将每个子矩阵逐行发送到SPU，完成MMMO-A的计算。在A_h的计算过程中，W^L被加载到SRAM-CIM中。注意为了减少SRAM-CIM在CIM模式和内存模式之间切换带来的延迟，HAIMA始终确保如果存在一个CIM模式的SRAM-CIM，则必须存在一个内存模式的SRAM-CIM。我们首先将W^L按列分为H个子矩阵，然后将子矩阵W_h^L按行分为M/256个子矩阵。以这种方式，MMMO-Z可以按照与MMMO-V/A相同的方式执。主机将Z作为Add和Norm层的输入来生成结果矩阵Z'。然后Z'被发送回DRAM-CNM以完成MMMO-FFNL的计算。

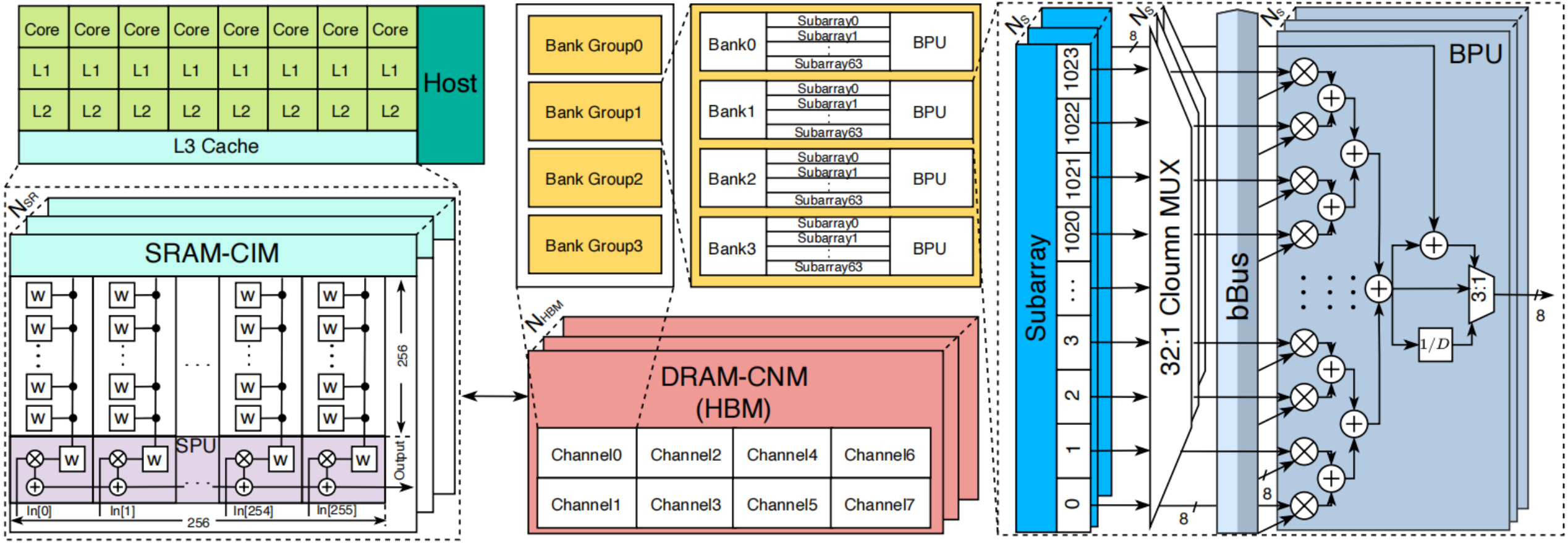


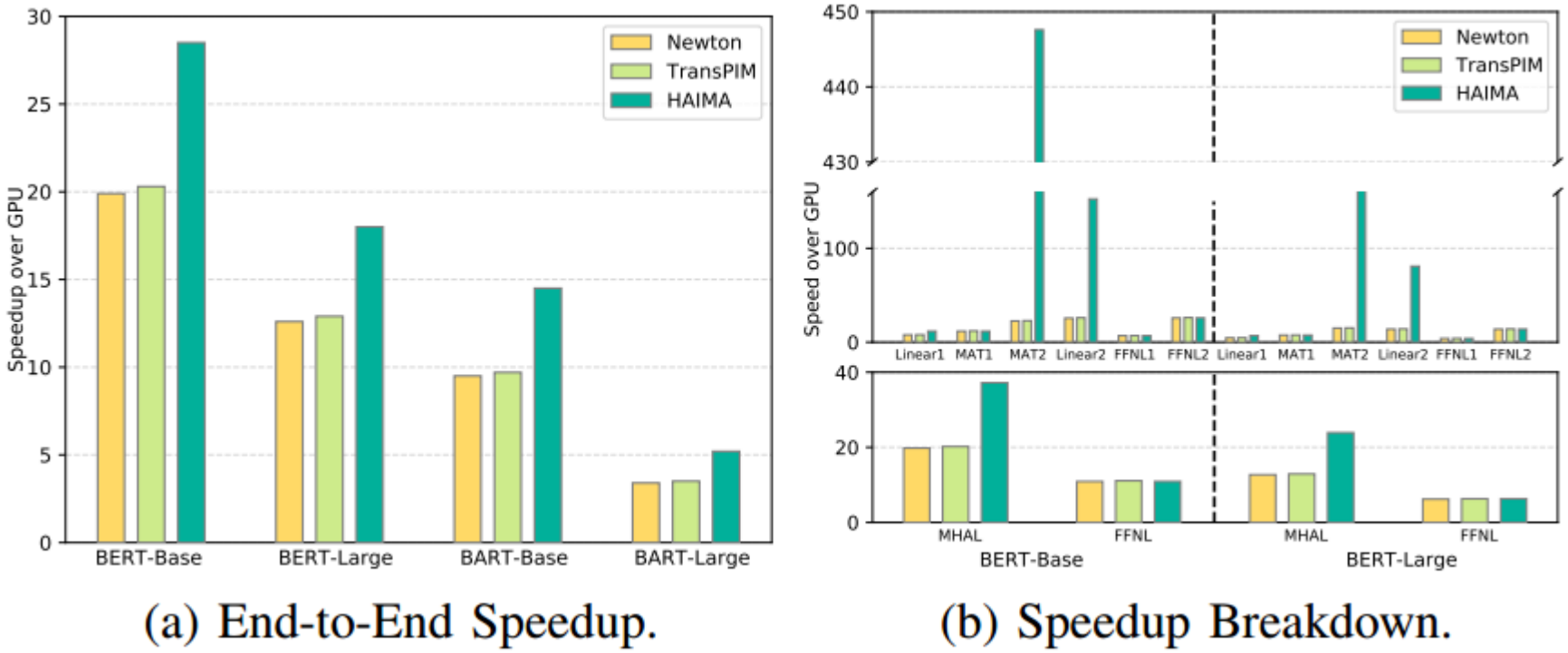
图5 Overview of *HAIMA*

$DRAM-CNM$：设计了BPU来支持$MMMO$的计算。HAIMA支持同时激活每个库中的N_S个子阵列，以提高计算的并行性。我在每个子阵列中保留一个连接到BPU的计算行（CR），需要执行的数据必须首先复制到该行。以这种方式，可以在访问CR中存储的数据时执行该数据。为了避免昂贵的面积和功率开销我在BPU和CR之间添加了一个32-To-1的MUX。BPU设计用于支持2个向量之间的点积。

$SRAM-CIM$：每个 SPU有256个计算单元，每个计算单元由8个6晶体管SRAM单元、一个乘法器和一个全加法器组成。类似地，为了避免昂贵的面积和功率开销，SPU被设计为支持2个矢量之间的点积。此外，$SRAM-CIM$中存储的需要执行的数据首先被复制到SPU的计算阵列中。然后，当流水线并行$MMMO$完成时，输入和部分和都通过SPU。

4.评价

1.Performance Comparison



(a) End-to-End Speedup. (b) Speedup Breakdown.

做性能对比时使用以下基于$DRAM$的$AIMA$或硬件作为对照:Newton,TransPIM，和Nvidia Tesla V100。此外，两个具有代表性的基于Transformer的$LSDNNs$，即$BERT$和$BART$，用于评估不同的$AIMA$。

2.Area and Power Comparison

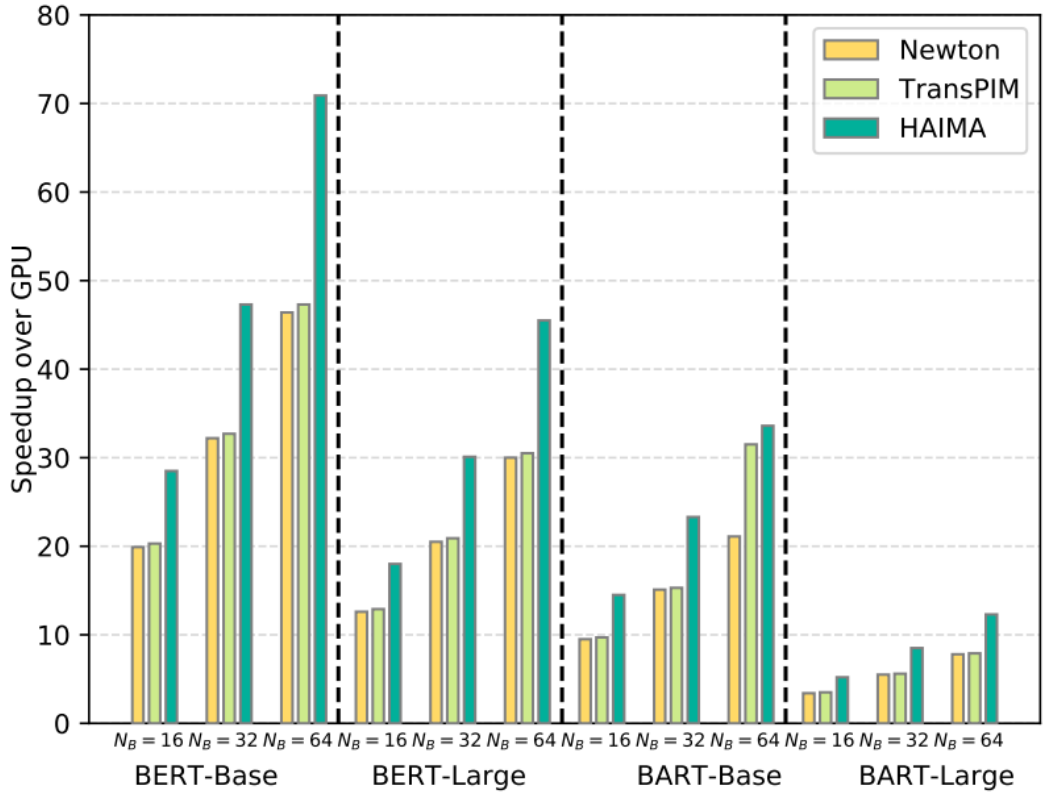
与具有8GB存储器（53.15mm2）的原始HBM2相比，在$HAIMA$中，每个CNM单元具有一个BPU和一个$32-to-1\ MUX$，因此导致约1.4%的面积开销。同时，与具有64KB内存（0.26mm2）的原始$SRAM$相比，每个$SRAM-CIM$中都有一个SPU，导致约48.1%的面积开销。

同时，由于Newton和TransPIM没有$SRAM-CIM$，文章只比较了3个$AIMA$的CNM单元之间的面积和功率。如表所示，$HAIMA$的CNM单元的面积（0.047mm2）分别是Newton和TransPIM的1.07倍和2.94倍。$HAIMA$的CNM单元（3138mW）的功率分别是Newton和TransPIM的1.01倍和3.92倍。

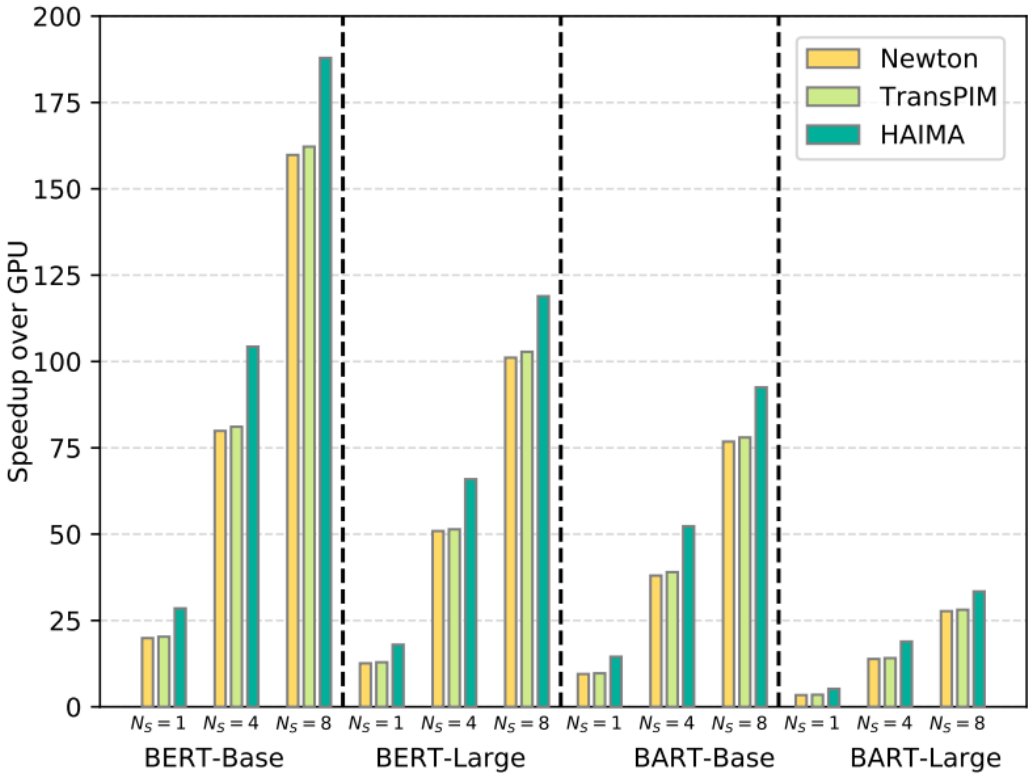
Unit	Area/Unit (mm^2)	Power/Unit (mW)
CNM Unit of TransPIM	0.016	800.6
CNM Unit of Newton	0.044	3,110.7
BPU of HAIMA	0.014	148.0
32-to-1 MUX of HAIMA	0.033	2,990
SPU of HAIMA	0.125	1,554.3

3.Scalability Comparison

如图a,b所示，随着N_B和N_S的增加，3个$AIMA$的加速性能显著提高。从中也可以看出，$HAIMA$是表现最好的。两个模型的加速度增益差异的原因是BART的参数规模大于BERT的参数规模。这证明了$DRAM-CNM$在加速$LSDNN$方面的优势。如图所示，增加N_S的速度提升幅度比增加N_B的速度提升幅度大。这是因为增加N_B会增加库与库之间的通信，因此加速受到影响。

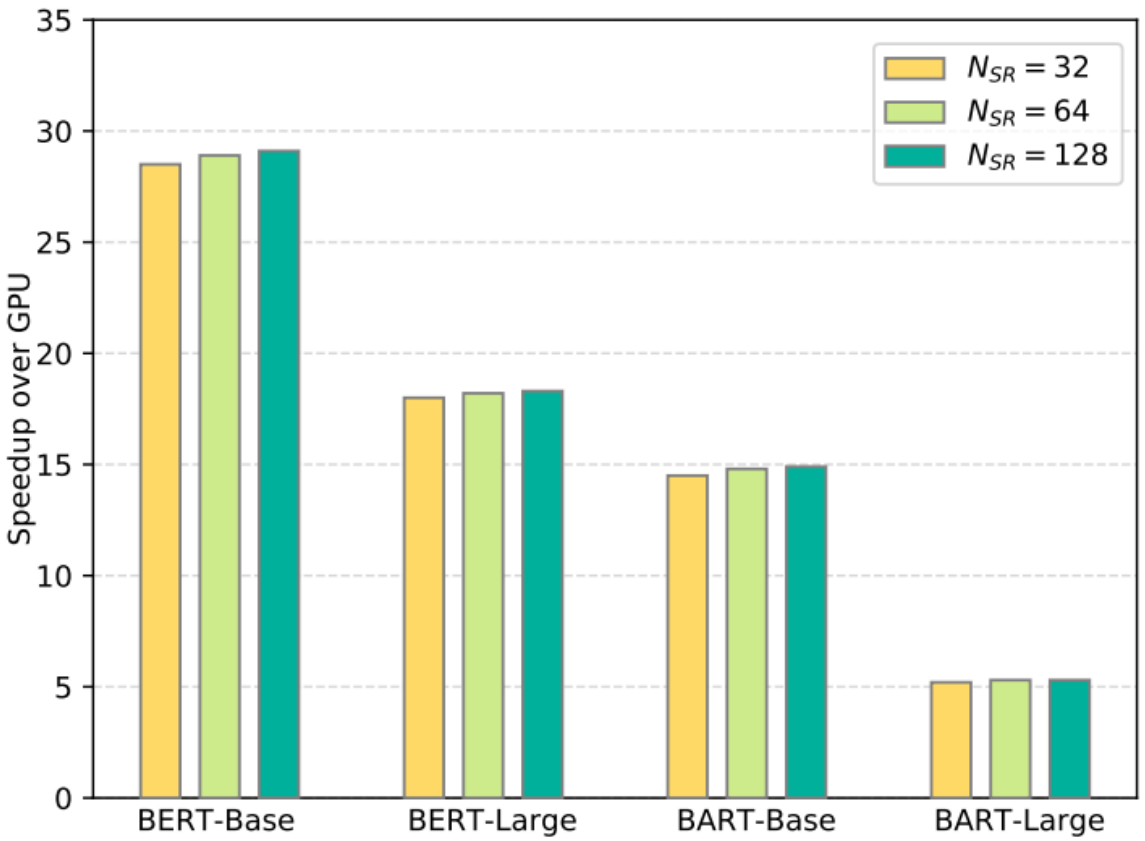


(a) Impact of N_B on Speedup

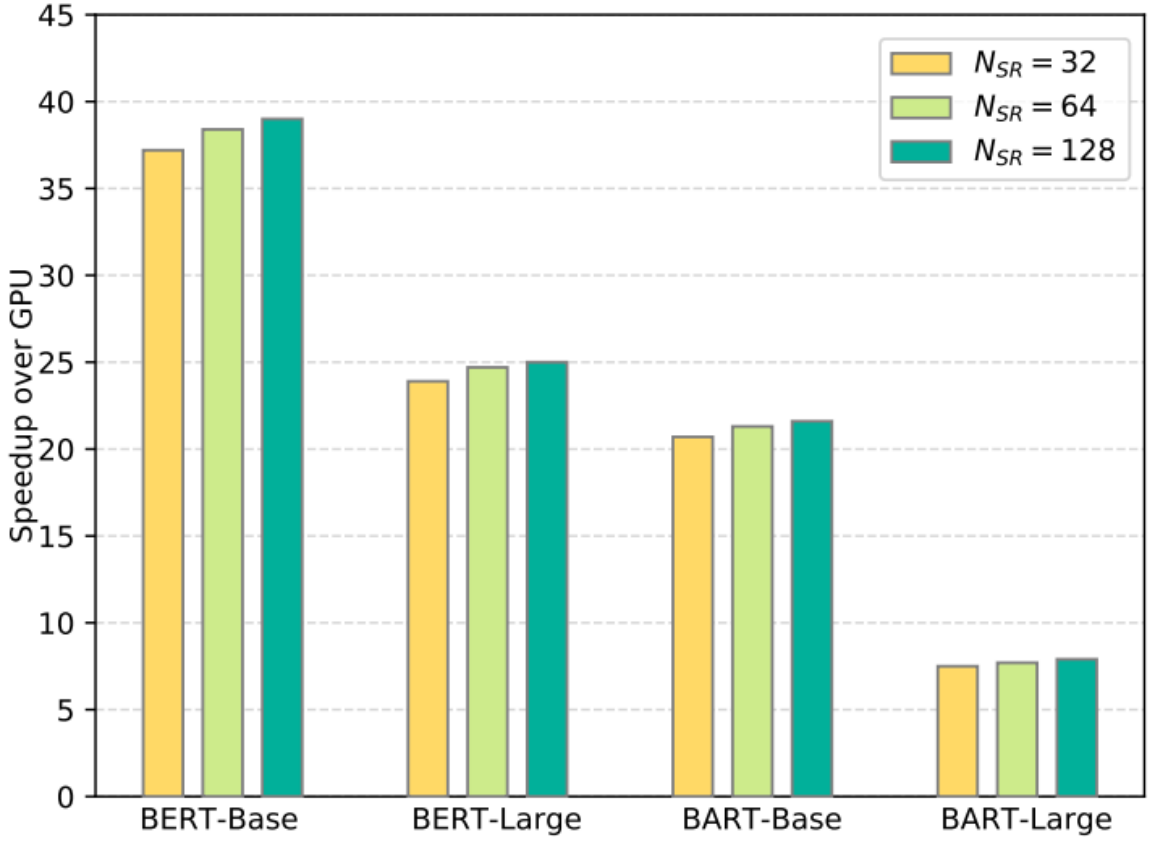


(b) Impact of N_S on Speedup

尽管增加$SRAM-CIM$的数量也可以提高$HAIMA$的性能，但是改进是非常有限的。这是因为$MMMO-V/A/Z$的工作负载在Transformer的总工作负载中占比较小的部分，从而限制了$SRAM-CIM$的增益加速。此外，等待S的V也会影响HAIMA的性能。当然，根本原因是$SRAM$的资源非常有限。实验结果证明了$SRAM-CIM$和$DRAM-CNM$合作加速$LSDNN$的必要性



(a) End-to-End Speedup



(b) MHAL Speedup

5.总结

本文提出了用于Transformer的$HAIMA$和并行数据流，利用$SRAM$和$DRAM$之间的协作来加速不同的$MMMO$。与其他基于$DRAM$的$AIMA$相比，本文提出的软硬件协同设计实现了1.4x-1.5倍的速度提升，并解决了基于$DRAM-AIMA$执行轻量级$MMMO$时资源利用不足的问题。本文的工作不仅展示了基于$SRAM$和$DRAM$的$AIMA$自的特性，而且为利用混合$AIMA$加速$LSDNN$开辟了机会。

THANK YOU