上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

MHSA 加速器前端设计报告

学生姓名 汪子尧、黄超凡、张博文

学号 124039910018、124039910088、124039910094

学院 集成电路与学院

2025 年 5 月 27 日

目录

1 理论基础

Transformer 是一种广泛应用于自然语言处理（NLP）和其他序列建模任务的神经网络架构，它在 2017 年由 Vaswani 等人提出，并被广泛应用于机器翻译、文本生成、语言理解和问答等任务中。Transformer 多头自注意力机制（Multi-Head-Self-Attention，MHSA）的思想应用到了后续大量的网络模型中，核心思想是将输入序列映射为键（key）、值（value）和查询（query）向量，然后计算不同头的注意力权重来捕捉序列中各个位置之间的关联信息，使得模型能够更精准地理解文本等序列数据的语义。

然而，MHSA 的运算复杂度较高，尤其是在处理长序列时，这成为了一个显著的瓶颈。对于序列长度为 L、特征维度为 C 的输入，MHSA 的运算复杂度主要由以下几个部分组成：

- **线性变换计算**：
  - 查询（Q）、键（K）和值（V）的线性变换，每个变换的复杂度为 $O(C^2)$。
  - 由于有三个这样的变换，总复杂度为 $O(3C^2)$。
- **注意力权重计算**：
  - 每个查询向量与所有键向量的点积计算，复杂度为 $O(L \times C)$。
  - 由于有 L 个查询向量，总复杂度为 $O(L^2 \times C)$。
- **缩放和 Softmax**：
  - 缩放操作的复杂度为 $O(L^2)$。
  - Softmax 函数的复杂度为 $O(L^2)$。
- **值向量加权求和**：
  - 根据注意力权重对值向量进行加权求和，复杂度为 $O(L^2 \times C)$。

综合以上各部分，多头自注意力机制的总运算复杂度为：

$$O(3C^2 + 2L^2 + 2L^2 \times C)$$

在实际应用中，序列长度 L 和特征维度 C 通常都比较大。例如，在常见的自然语言处理任务中，序列长度 L 可能达到几百甚至上千，特征维度 C 通常为几百到几千。这使得多头自注意力机制的运算复杂度非常高，尤其是当需要处理长序列时，计算量会呈指数级增长。

由于运算复杂度的瓶颈，单纯的软件实现无法满足实时性要求较高的应用场景。因此，硬件加速器的设计变得至关重要。我们设计了专门针对多头自注意力机制的硬件加速器，并将其挂载在蜂鸟 E203 RISC-V 处理器上，以充分利用硬件的并行计算能力，显著降低运算复杂度，提高处理效率。

2 MHSA 加速器架构设计

2.1 异构加速系统设计

E203-MHSA-Accelerator 是一款基于芯来蜂鸟 E203 RISC-V 处理器自主开发的硬件加速模块。该模块作为独立外设直接挂载在 E203 SoC 的 ICB 总线上，专门针对 Transformer 模型中的多头自注意力（MHSA）计算单元进行硬件加速。该方案通过在 SoC 系统中集成专用硬件加速模块，针对当前 Transformer 模型在边缘设备部署时面临的计算瓶颈问题，提出了轻量化异构计算解决方案。异构加速系统架构如图 1所示：

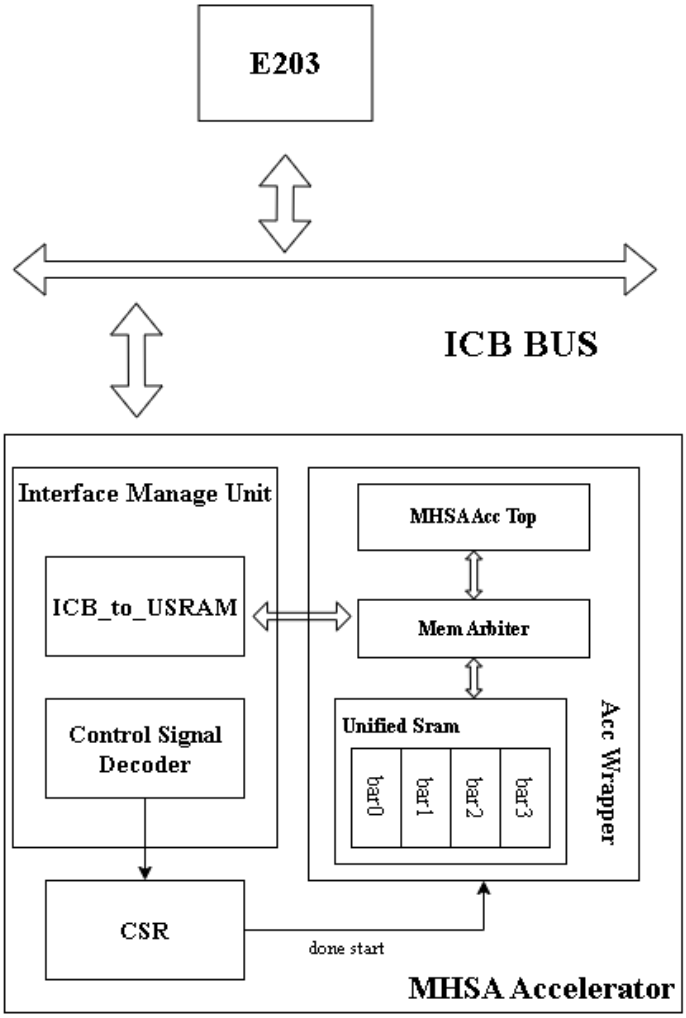

图 1: E203-MHSA-Accelerator 异构计算加速系统

图 1中，Acc Wrapper 模块即为 MHSA 加速器顶层模块，加速器顶层模块内主要包含内部 Memory 、计算模块与其他相应的控制模块。我们将 MHSA 加速器的顶层架构设计安排在 2.2 节中详细阐述，并在第 3 章中深入探讨加速器的前端设计。

接口管理模块（Interface Manage Unit，IMU）负责桥接 E203 主控核的 ICB 总线协议与加速器内部逻辑。该单元将 ICB 总线输入的读写请求实时解析为两类信号：面

向 SRAM 存储体的统一访存接口（Unified Sram Interface，USI）的访存通道，以及对控制状态寄存器（Control State Register，CSR）的硬件控制通道。

对于 Host 端而言，不必关心加速器内部的具体实现细节，只需要关注加速器的 Memory Map 即可。Memory Map 定义了 Host 端与加速器交互时需要访问的寄存器地址和对应的功能描述。通过 Memory Map，Host 端可以方便地控制加速器的启动、监控加速器的状态以及读写加速器的数据缓冲区。Memory Map 表格如表 1所示：

表 1: Memory Map

Offset Address	Memory Type	R/W	功能描述
0x00000000-0x00004000	Data Buffer	R/W	统一编址的 SRAM 存储体
0x00004000	Control Register	R/W	bit[0]=1：加速器使能
0x00004004	Status Register	R	bit[0]=1：运算完成
0x00004008	Configuration Register	R/W	输入数据起始地址
0x0000400C	Configuration Register	R/W	输出数据起始地址
0x10000000-0xFFFFFFFF	Reserved	-	保留地址空间

2.2 MHSA 加速器顶层设计

为提升加速器系统的硬件兼容性，外设采用标准化的接口控制体系。接口信号如表 2所示：

表 2: 接口信号定义

信号类型	信号名称	方向	位宽	描述
Global signal	clk	I	1	系统时钟信号
	rst_n	I	1	低有效异步复位信号
Control signal	done	O	1	加速器计算完成标志
	start	I	1	加速器启动触发信号
	input_base	I	[31:0]	输入数据基地址（32 位地址空间）
	output_base	I	[31:0]	输出数据基地址（32 位地址空间）
Unified Sram Interface	soc_write_en	I	1	SRAM 接口写使能信号
	soc_data_in	I	[WIDTH-1:0]	输入数据总线
	soc_addr	I	[31:0]	统一寻址总线（32 位地址空间）
	soc_data_out	O	[WIDTH-1:0]	输出数据总线

加速器顶层模块内例化 MHSA_Acc_Top ， Mem_Arbiter , Unified_Sram 三个模块：

(1) **MHSA_Acc_Top**：

本模块为加速器的核心计算模块，负责执行多头自注意力机制的计算。我们将在第 3 章节中详细解释加速计算原理与前端设计。

(2) **Mem_Arbiter**：

- **Host 端与加速器运算单元对 Unified Sram 的访问仲裁：** MHSA 任务启动前，Host 端通过 DMA 或 Direct-IO 的方式从外部存储向加速器内部存储单元完成输入数据与权重数据的数据搬运；MHSA 任务进行时，加速器运算单元需要从内部存储器读取输入数据与权重数据执行运算过程。
- **Host 端对不同内存条带(bar)的访问仲裁：**由于 ICB 转 Unified_Sram_Interface 的数据带宽受 ICB 总线限制（32bit），因此对 4 块 bar 进行统一地址编码，仲裁单元根据地址映射和访问请求，决定哪个内存条带应该被访问。统一地址编码如表 3所示：

表 3: 统一地址编码

内部存储 bar	统一编码地址空间
bar0	0x0000_0000 - 0x0000_1000
bar1	0x0000_1000 - 0x0000_2000
bar2	0x0000_2000 - 0x0000_3000
bar3	0x0000_3000 - 0x0000_4000

(3) **Unified_Sram**：

- Unified_Sram 模块为 Host 端提供了一个统一接口，用于访问多个 SRAM 存储体。
- MHSA 计算单元频繁读取输入数据、权重数据以及中间数据。为最大化 Unified_Sram 的访问带宽，四块 bar 的输入输出接口被并行连接至计算模块，而非提供统一的编址空间。这种设计允许计算模块同时对四块 bar 进行读写操作，从而显著提升数据处理效率。

2.3 MHSA 加速系统运行流程

(1) Host 端完成任务定义与划分

(2) Host 端通过 DMA 或 DIRECT-IO 的方式通过 ICB 总线向加速器 Unified Sram 写入初始数据（输入参数/输入权重）

(3) Host 端通过配置加速 CSR 使能加速器

(4) MHSA 加速器进行运算任务

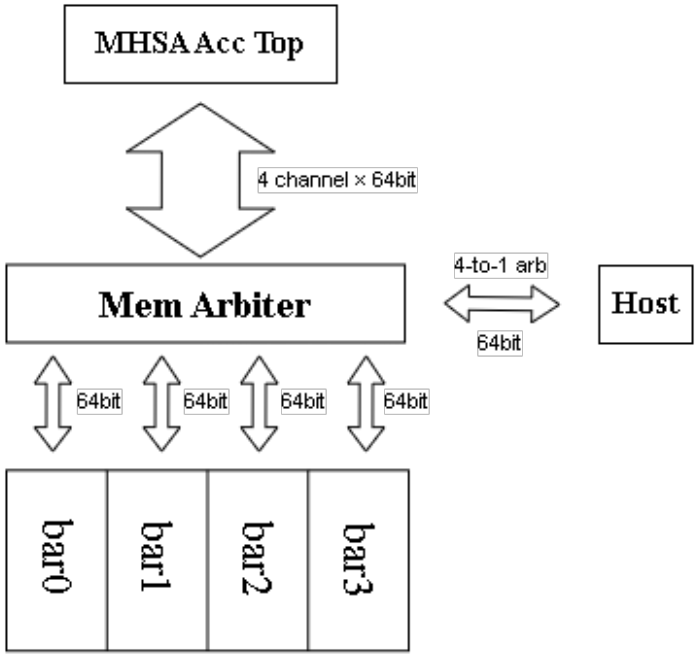

图 2: 仲裁位宽示意图

(5) Host 间隔一定时间轮询读取加速器状态寄存器结束位

3 MHSA 前端设计

3.1 计算原理

3.1.1 基本计算原理

MHSA（Multi-Head Self-Attention，多头自注意力）计算是 Transformer 中重要的计算单元。基本计算原理简化为以下三个步骤：

1. 输入投影与分解

   - 特征维度（B, L, C）的输入向量 X 通过三组可训练参数矩阵（$W_Q/W_K/W_V$）生成查询（Q）、键（K）、值（V）三元组，沿特征维度拆分为 h 个子矩阵。

2. 注意力权重生成

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

3. 加权融合与重构

   - 注意力权重与 V 矩阵相乘后，通过拼接单元将 h 个头输出沿特征维度重组，最终线性变换层后输出。

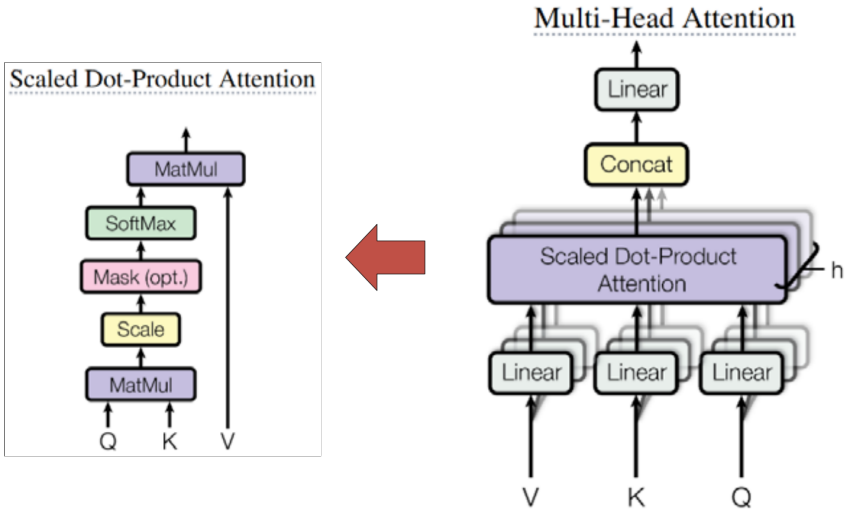

图 3: MHSA 计算原理

3.1.2 加速计算原理

基于对多头自注意力（MHSA）计算特性的深入分析，我们针对 MHSA 计算的计算密集型和访存密集型两种特性提出三种可能的加速策略，优化策略如下：

(1) **Pipeline 流水线化**

通过对 MHSA 计算图的时间局部性特征进行分析，我们可以建立 MHSA 运算的流水线架构：第一级完成序列拆分与 Q/K/V 矩阵计算，第二级执行 Scaled Dot-Product Attention 的并行计算，第三级处理多头结果的拼接与线性变换。其中，Scaled Dot-Product Attention 内部可继续流水线化，拆分为 QK-MatrixMulti 层，Scale 层与 Softmax 层。同时，本次设计针对特定的任务要求，即输入特征维度（B，L，C）=（1，32，128），由于 B=1，加速器只会进行一轮 MHSA 的运算，因此流水线化对性能的提升并不明显，但我们为了保留这一特性的体现，在实现中采用顺序状态机的方式，对可以顺序进行的运算层进行分解，如 3.3 节中所示。

(2) **矩阵乘运算优化**

考虑特征维度（B，L，C）=（1，32，128）的情况，我们对 MHSA 的计算复杂度进行具体分析如下：

i. **QKV 投影**

$$3LC^2 = 3 \times 32 \times 128^2 = 1572864 \quad （占比\ 66.6\%） \tag{2}$$

ii. **自注意力矩阵乘**

$$2L^2C = 2 \times 32^2 \times 128 = 262144 \quad （占比\ 11.1\%） \tag{3}$$

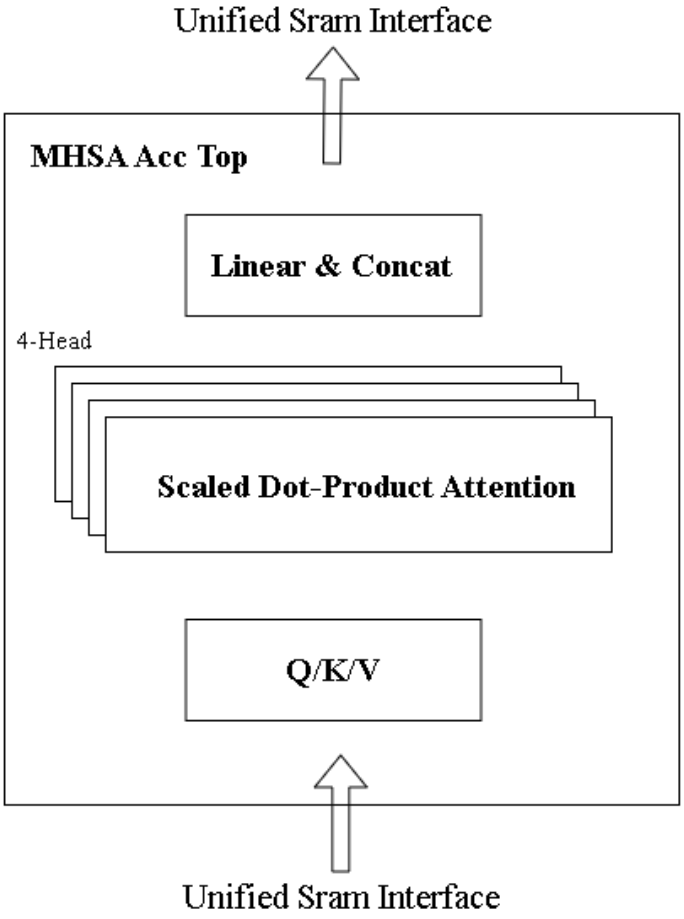

图 4: Pipelined-MHSA (3-level)

iii. **输出投影**

$$LC^2 = 32 \times 128^2 = 524288 \quad (\text{占比 } 22.2\%) \tag{4}$$

iv. **Softmax**

$$3L^2 = 3 \times 32^2 = 3072 \quad (\text{占比 } 0.13\%) \tag{5}$$

矩阵乘操作占比为:

$$\eta_{\text{matmul}} = \frac{4LC^2 + 2L^2C}{C_{\text{total}}} \times 100\% \approx 99.87 \tag{6}$$

基于上述计算量分析,矩阵乘法运算作为 MHSA 的核心计算瓶颈(占比 99.87%),其加速优化成为提升整体性能的关键路径。为此，本设计采用基于空间并行架构的脉动阵列（Systolic Array）实现方案，我们将在 3.2 小节中详细介绍脉动阵列的前端设计实现。

(3) **量化**

在本次设计中，考虑到输入数据为 8 位整数（8bit int 型），我们决定充分利用这一特性，对中间计算结果也采用 8 位量化处理，以确保数据处理过程的一致性和高效性。

3.2 脉动阵列设计

脉动阵列作为本设计的底层核心架构，是整个 MHSA 加速器运算的基础。因此，本小节将专门对脉动阵列的前端设计进行深入且详细的阐述。

在确定脉动阵列的规模时，我们首先考虑了 MHSA（多头自注意力机制）运算的需求。由于 Q/K/V 的线性层运算可以完全并行执行，因此需要 3 个独立的脉动阵列矩阵乘单元来分别处理 Q、K、V 的线性变换。此外，QK 矩阵乘法层、注意力权重矩阵乘法层以及输入投影连接层各自也需要 1 个脉动阵列矩阵乘单元。这意味着整个系统总共需要 6 个脉动阵列矩阵乘单元。

同时，设计要求对乘法器的数量有限制，**总加速器使用的乘法器数量不得超过 480 个**。基于这个约束，我们对脉动阵列的规模进行了优化选择。经过评估和计算，确定了最大的脉动阵列规模为 8×8 的脉动阵列，在满足运算需求的同时，有效地控制乘法器的使用数量。

接下来，我们对脉动阵列的数据流进行规划。在使用脉动阵列实现两个二维矩阵乘法时，通常有以下两种策略:

1. **(i) 数据流动方式一：**在脉动阵列中，流动的数据是矩阵 X 和矩阵 W，而每一步的中间结果 Y 则存储在每个处理单元（PE）中，随着数据的逐级传递进行累加。

2. **(ii) 数据流动方式二：**在脉动阵列中，流动的数据是输入矩阵 X 以及部分和 Y 的中间结果，而权重矩阵 W 则固定存储在每个处理单元（PE）中，每个 PE 利用本地的 W 对流动的 X 进行运算，并将更新后的部分和传递到下一个单元

两种数据流如图 5所示：

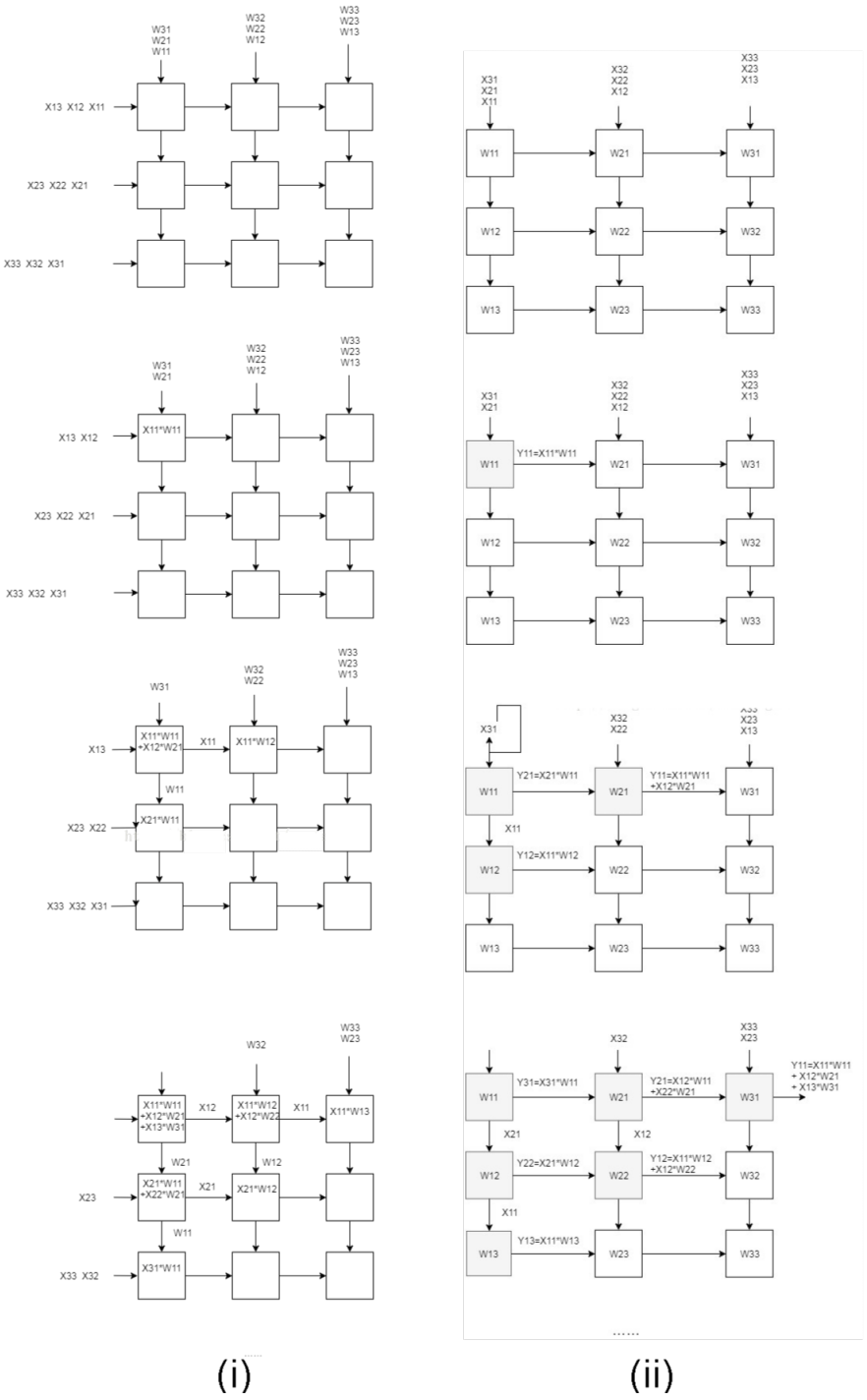

图 5: 脉动阵列实现矩阵乘的两种数据流

在方案 (ii) 中，由于我们采用的是 8×8 的脉动阵列，无法在一次操作中完成 32/128 规模的矩阵运算。因此，每次运算都需要重新将权重数据加载到每个 PE 单元中。这相

当于为权重数据开辟了一组额外的寄存器用于缓存。与方案 (i) 相比，这种方式在时间和空间上的开销都显著增加。因此，从资源利用效率的角度来看，方案 (ii) 并不如方案 (i)。

然而方案 (i) 也对数据输入带宽提出了更高的要求。它不仅需要输入矩阵 X 的数据，还要求同步输入权重数据。针对这一情况，我们考虑内部统一存储器 Unified SRAM 的带宽是否能够满足需求。对于一个 8×8 的脉动阵列，若要实现不间断的流水线操作，最少需要在一个时钟周期内输入 8 个 8 位的输入数据以及 8 个 8 位的权重数据。在流水化后的 MHSA 架构中，对数据带宽要求最高的部分是 Q/K/V 投影的线性层，这部分需要同时输入三组权重数据。然而，输入数据 X 可以被广播到三个脉动阵列中。因此，最少需要 4 块位宽为 64 位的数据总线（bar）。经过评估，这样的设计符合整体架构的要求。

最后值得注意的一点，例如：方案 (i) 的数据流中，x11，x21，x31 ... 不会同时输入脉动阵列，因此这里需要进行延时控制，对于第 8 列的脉动阵列，需要延时 7 个 clock cycle 后输入脉动阵列。

脉动阵列的接口定义如下：

```
module mm_systolic(
    input logic clk,
    input logic rst_n,

    input logic [63:0] row_bar,
    input logic [63:0] col_bar,
    input logic bar_valid,

    output logic [31:0] res [0:7][0:7],

    input logic flush // pull up one cycle to flush PE result
);
```

其中 row_bar 与 col_bar 分别为一次从存储器中独处的 8 个（1 组）8bit 的输入数据 X 与权重数据 W，flush 信号用于冲刷一轮运算完成后脉动阵列每个 PE 内的累加和数据。

3.3 MHSA 计算模块

在顶层模块中，我们采用顺序状态机对流水线化后的 MHSA 进行控制。由于状态机 MHSA 计算逻辑基本顺序实现，因此状态机逻辑较为简单，当下一层的运算完成标志信号被拉高时，状态机将自动进入下一层的运算控制。

在采用顺序状态机的设计框架下，计算模块与统一存储器 Unified Memory 之间的

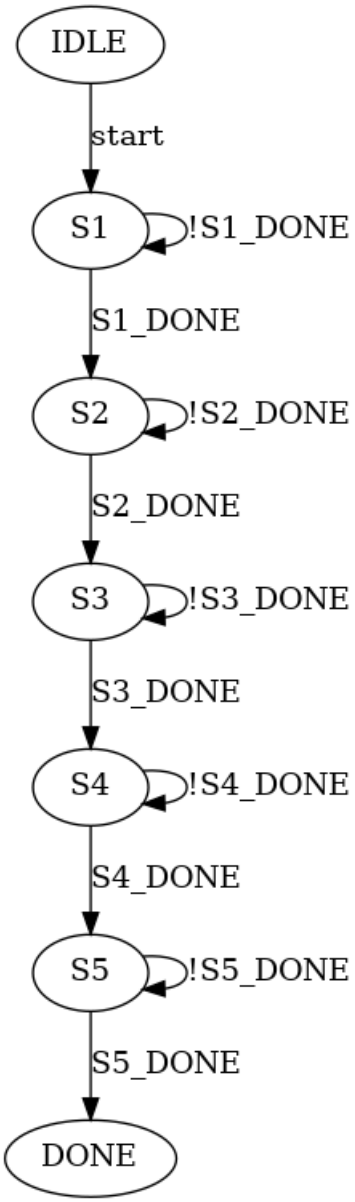

图 6: MHSA 顺序状态机

交互得以实现更为精细且高效的控制。具体而言，当状态机处于某一特定运算状态时，例如处于第 i 层运算状态，该层的计算模块将自动获得对 4 块数据总线（bar）的存储总线仲裁权，并完全独占这些总线的读写控制权。这使得计算模块能够高效地读取输入数据和权重数据，并将中间结果写入存储器，而无需担心与其他模块的潜在冲突。一旦当前层的运算任务完成，计算模块会立即释放对存储总线的控制权。此时，状态机根据预设的逻辑自动进入下一状态，相应地，下一层的计算模块随即获得仲裁权，接管总线的读写控制权，继续执行其运算任务。这一过程会按照既定的顺序依次进行，直至所有运算层完成，状态机进入最终状态。

3.3.1 Unified Memory 数据地址映射

这里我们需要对原始输入数据，原始权重数据，以及中间数据在统一存储器中 BAR 与地址的分布进行特别说明，因为这决定模块对读写的 bar 选择与地址产生。为了最大化利用存储器的带宽（4 bar × 64 bit），使用脉动阵列执行矩阵乘运算的模块需要将待输入的两组数据分别存储在不同的 bar 上以便可以同时输入脉动阵列而不需要中间缓存。因此需要对数据流依赖进行分析后，写入到合适的 bar 上，这里我们将 BAR 分为 bar0，bar1，bar2，bar3。本设计中，数据的排布关系如图 7所示：

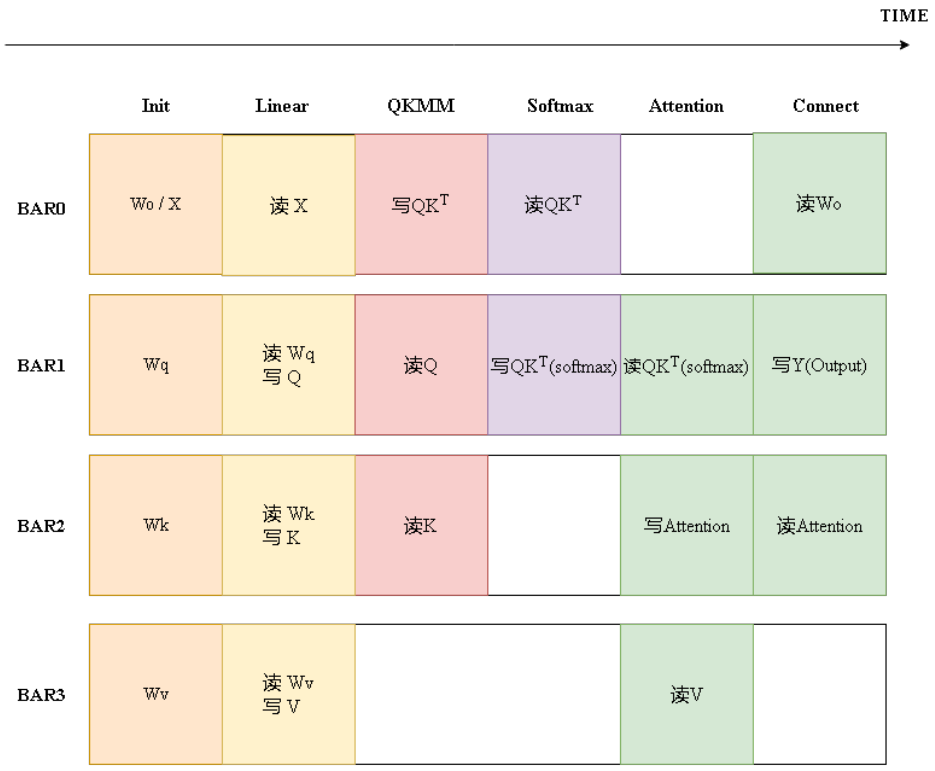

图 7: 原始/中间数据在 Bar 的分布

从对 Unified Memory 的读写时空分布图中我们可以看到本设计访存的效率较高，且最大化利用了存储带宽，提高了运算模块的访存读写效率。其中利用率较低的模块为

Softmax，该模块本身即为输入输出的单映射，且不需要进行矩阵乘运算，因此读写各自只需要 1 块 bar 即可。

3.3.2 linear 线性层

在 MHSA 的线性层中，需要完成输入矩阵 X 与权重矩阵 Q、K、V 的矩阵乘法运算，其运算规模为 (32,128)×(128,128)。由于我们采用的是 8×8 的脉动阵列，其单次运算规模有限，因此必须通过分时复用的方式，将大矩阵分解为多个小块进行分块矩阵乘法运算。具体而言，我们将 32×128 的输入矩阵和 128×128 的权重矩阵分别划分为多个 8×8 的子矩阵块。根据矩阵的维度，分块的次数为 32/8 × 128/8 =64 次。每次分块运算后，脉动阵列会输出一个 8×8 的中间结果矩阵。通过依次执行这 64 次分块矩阵乘法运算，并将每次得到的中间结果矩阵逐步拼接，最终能够重构出完整的 (32,128) 的结果矩阵。

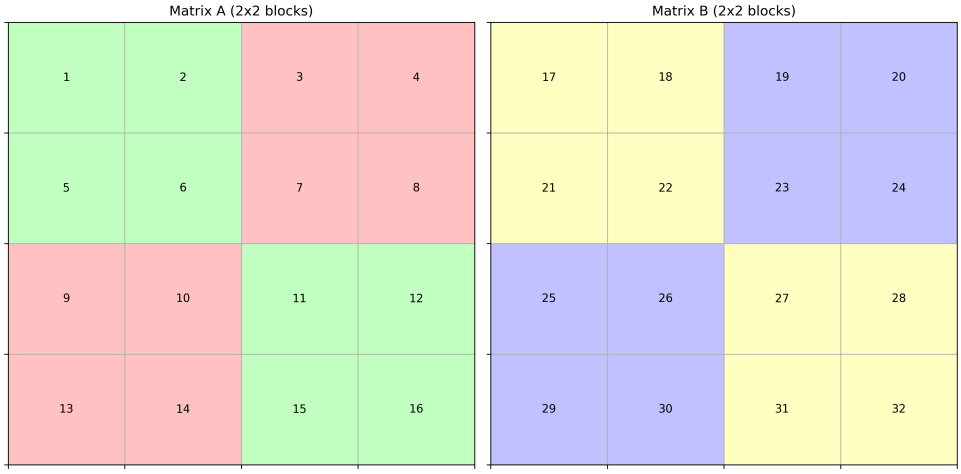

图 8: 脉动阵列分时复用实现分块矩阵乘法

为了高效地实现 64 轮分块矩阵乘运算，我们将整个运算过程进一步分解为两层嵌套循环。具体来说，外层循环对应于输入矩阵 X 的行方向，循环次数为 32/8 = 4 次；内层循环对应于权重矩阵 W 的列方向，循环次数为 128/8 = 16 次。在具体操作中，我们首先固定输入矩阵 X 的行数据。在每一外层循环中，依次输入 X 的 1-8 行、9-16 行、17-24 行和 25-32 行数据。对于每一段固定的 X 数据，内层循环负责遍历权重矩阵 W 的列数据，依次输入 W 的 1-8 列、9-16 列、17-24 列，直至 121-128 列。当内层循环完成一轮（即输入完 W 的 121-128 列数据）后，外层循环触发，移动 X 的输入范围，进入下一组 X 数据的处理。

为了控制这一复杂的循环过程，我们引入了一个循环计数器（loop counter），其低 4 位用于指示当前内层循环的进度，即当前处理的权重矩阵 W 的列位置；而第 5-6 位则用于指示外层循环的进度，即当前处理的输入矩阵 X 的行位置。值得一提的是这个 Design Tip 使得我们只需要一个计数器，就可以同时跟踪两个循环的进度，简化了控制逻辑。

为了精确控制每一轮循环中的数据读取、运算以及结果写回操作，我们设计了一个包含 IDLE、READ、WAIT 和 WRITE 四个状态的状态机。这四个状态分别负责以下任务：

i. IDLE：空闲状态，等待启动信号。

ii. READ：从统一存储器（Unified Memory）中读取数据，并将其输入到脉动阵列中。

iii. WAIT：等待脉动阵列完成当前运算。由于脉动阵列的流水线特性，需要确保所有数据完成计算。

iv. WRITE：将脉动阵列中每个处理单元（PE）的结果写回到统一存储器中，缓存中间结果。

在每一轮循环中，模块首先进入 READ 状态，从统一存储器中读取数据，并按照 3.2 节中描述的方式将其输入到脉动阵列中。随后，模块进入 WAIT 状态，等待脉动阵列完成当前运算。**需要注意的是，根据 3.2 节中的描述，脉动阵列的第 n 行或列的初始数据需要延迟 n−1 个时钟周期后才能输入。即使所有数据完全输入后，最后一个数据在脉动阵列中通过流水线传递到最后一个 PE 仍然需要经历 8 个时钟周期。因此，整个过程的最大延迟为 7+8=15 个时钟周期**。这意味着在状态机中必须添加 WAIT 状态，以确保脉动阵列完全完成计算后再执行写回操作。完成等待后，模块进入 WRITE 状态，将脉动阵列中每个 PE 的结果值写回到存储器中，缓存中间结果。

从 3.3.1 小节中，我们可以知道，线性层的中间结果会被写回到存放各自权重数据的 bar 上，因此例化 3 个线性层模块互不干扰，并行运算 Q/K/V 矩阵后写回，原始数据 X 的输入被广播到 3 个模块同步运算。

3.3.3 QK-MatrixMulti 层

在 3.3.2 节中，矩阵 Q、K 和 V 分别被写入到不同的存储区域（bar1、bar2 和 bar3）。在处理 4-Head 情况下的 QK 矩阵乘法时，尽管理论上可以并行完成所有 4 个 Head 的运算，但由于硬件资源（特别是乘法器数量）的限制，我们选择使用一个 QK-MatrixMulti 层，依次对每个 Head 执行 QK 矩阵乘法运算。

对于单个 Head，其矩阵乘法运算规模为 (32,32)×(32,32)。为了在有限的硬件资源下高效完成这一运算，我们依然采用 8×8 的脉动阵列，并通过分时复用的方式将其划分为更小的子矩阵进行运算。具体来说，整个运算需要进行 32/8 * 32/8 =16 次分块矩阵乘法运算。

由于 4 个 Head 的完整 Q（以及 K 和 V）矩阵均存储在同一块 bar 上，我们可以通过修改偏移地址来顺序访问每个 Head 的数据，从而依次完成 4 个 Head 的 QK 矩阵乘法运算。从本质上讲，这一过程相当于在原有的两层循环（分块矩阵乘法的外层和内层循环）基础上，再添加一层针对不同 Head 的遍历循环。

在内部实现逻辑上，这一过程与线性层（Linear Layer）的设计类似，主要通过循环计数器和状态机来控制数据的读取、运算和写回。由于其逻辑与线性层高度相似，这里不再重复赘述。

这里仅提供不同于 Linear 的 4-Head 地址偏移的相关参考代码：

```
assign addr_bar0 = LINEAR_OUTPUT_BASE + head_dim * 'd128 +
    read_addr_q;
assign addr_bar1 = LINEAR_OUTPUT_BASE + head_dim * 'd128 +
    read_addr_k;
assign addr_bar2 = QKMM_OUTPUT_BASE + head_dim * 'd128 + write_addr;
```

3.3.4 scale 层

在 MHSA 计算中，Q 与 K 相乘得到的注意力权重矩阵通常具有较大的数值波动范围。为了避免 softmax 函数中出现数值不稳定或梯度消失等问题，需要在 softmax 运算前对注意力矩阵进行缩放处理。每个注意力权重应除以向量维度的平方根（即 $\sqrt{d_k}$，其中 d_k 是 key 向量的维度），从而保持数值稳定性并提升训练效率。

该过程的数学表达式如下：

$$Attention(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) \tag{7}$$

由于 $d_k = 32$，则有：

$$\frac{1}{\sqrt{32}} \approx 0.1768 \tag{8}$$

同时，由于使用 8-bit 定点数（[0, 255]）进行量化，因此将 $\frac{1}{\sqrt{32}}$ 乘以 4，保证数据范围能保持在 [0, 255] 之间。**在硬件中该步骤进一步简化，将缩放操作通过移位实现，即近似缩放因子为 0.703125**，表示为：

$$\frac{1}{\sqrt{32}} \times 4 \approx 0.703125 = \left(\frac{1}{2} + \frac{1}{8} + \frac{1}{16} + \frac{1}{64}\right) \tag{9}$$

本模块的主要输入为 QK 矩阵乘法层的中间结果，输入为 64-bit 总线，每 8-bit 表示一个量化的 attention 分数，共 8 个数据组成一个 vector。

模块实现中，采用定点近似乘法方式将每个 8-bit attention 分数与缩放因子 0.703125 相乘，该步骤转化为了移位操作。该计算方式避免了乘法器资源的使用，完全由移位器和加法器组成，硬件资源占用极低，减少了对面积和功耗的消耗。

3.3.5 softmax 层

Softmax 是 Transformer 中多头自注意力(MHSA)机制的关键组成，用于将 attention logits 映射为概率分布，使得每个注意力得分均在 $[0, 1]$ 范围内，且一行内总和为 1，保

证注意力机制的归一性。其标准定义为：

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{10}$$

在 MHSA 中，Softmax 应用于 $QK^T/\sqrt{d_k}$ 的结果矩阵，如式（7）所示，使其每一行作为对各 key 向量的注意力分布，用于加权求和。然而，Softmax 涉及两个计算复杂度较高的操作：

i. 指数函数 e^x

ii. 除法运算 $\frac{1}{\sum e^{x_j}}$

若直接用定点运算实现，所需的乘法器与除法器将大大增加芯片面积和功耗。为此，**本设计采用 LUT（查找表）替代上述操作 [1]，通过查表代替复杂运算，在资源受限的嵌入式 SoC 上实现高效近似 Softmax 运算**。整个 softmax 分两步：

1. 1 维 LUT：对每个 attention score 进行指数映射，近似计算 e^{x_i}，由一维 LUT 完成；
2. 2 维 LUT：将每个 e^{x_i} 与当前行的 $\sum_j e^{x_j}$ 进行除法计算，由二维 LUT 完成归一化。

(1) **1 维 LUT：指数逼近**

本设计使用 8-bit 定点输入，数据范围在 $[0, 255]$，其指数查找表按以下公式预先离线计算：

$$\text{lut1D}[i] = \left(\frac{1}{e^{(255-i)/96}} \right) \times 255 \tag{11}$$

由于指数函数增长极快，若直接使用输入 $x_i \in [0, 255]$，会导致指数输出严重失衡，几乎所有较小输入对应的输出都接近 0，从而使 softmax 函数的归一化作用失效。为此，需要压缩指数函数的输入动态范围，使得 LUT 的输出值能够在整个输入区间内较均匀地分布，保持足够的区分度。因此需要除以 96 ，人为缩小指数函数的指数输入范围，以确保指数值不会迅速饱和，对小值的指数结果不会一律变成 0，避免精度丢失。

(2) **2 维 LUT：归一化逼近**

完成指数计算后，需将每个元素的指数值除以当前行的指数和以实现归一化操作，设计了二维查找表来近似替代上述除法过程。

为了使查找表可控且查表高效，不直接使用全 8-bit 的分子和分母值作为查表索引，而是对其输入进行了位宽压缩与抽象：分子表示当前元素的指数值 e^{x_i}，使用

其高 4 位（bit[7:4]）作为 LUT2D 的行索引，映射范围 $[0, 15]$。分母表示当前行所有元素指数值之和 $\sum_j e^{x_j}$，使用其高 6 位（bit[12:7]）作为列索引，映射范围 $[1, 63]$（避免除以 0）。该压缩方案能显著降低 LUT 存储开销，仅需 $16 \times 64 = 1024$ 个条目。如下式所示：

$$\text{lut2D}[i][j] = \frac{i/16}{j/128} \times 16 \tag{12}$$

归一化放大 16 倍以保留精度，使得结果范围保持在 [0, 255]，同时使用 2 维 LUT 从而避免除法器，实现近似归一化。这种设计极大简化了计算复杂度，只需查两张表即可完成，整体公式如下：

$$\text{softmax}(x_i) \approx \text{lut2D}[\text{lut1D}[x_i]/16][\sum \text{lut1D}[x_j]/128] \times 16 \tag{13}$$

总体而言，Softmax 层通过 LUT 实现近似计算，完全去除乘法器与除法器，节省大量面积与功耗；查表替代指数与归一化，大幅降低时序路径与延迟。

3.3.6 Attention-MatrixMulti 层

同 QK-MatrixMulti 层运算逻辑完全一致，仅对于输入输出 bar 的选择以及数据基地址不同，具体见 3.3.3 节。

3.3.7 connect 层

同 Linear 层运算逻辑完全一致，仅对于输入输出 bar 的选择以及数据基地址不同，具体见 3.3.2 节。

参考文献

[1] Vasyltsov I, Chang W. Efficient softmax approximation for deep neural networks with attention mechanism[J]. arXiv preprint arXiv:2111.10770, 2021.