1. 实验数据记录表

实验编号	实现方式	设备	矩阵尺寸 (M×K×N)	计算耗时 (ms)	备注
1	朴素 baseline	设备	64×64×64	42.088 ms	
2	HVX 内积 (A * B^T)	设备	64×64×64	44.105 ms	
3	HVX 外积 (A * B)	设备	64×64×64	34.979 ms	
4	朴素 baseline	设备	256×256×256	481.815 ms	
5	HVX 内积 (A * B^T)	设备	256×256×256	109.480 ms	
6	HVX 外积 (A * B)	设备	256×256×256	134.429 ms	
7	朴素 baseline	设备	512×512×512	3600.657 ms	
8	HVX 内积 (A * B^T)	设备	512×512×512	454.909 ms	
9	HVX 外积 (A * B)	设备	512×512×512	670.457 ms	
10	朴素 baseline	设备	88×99×66	58.547 ms	
11	HVX 内积 (A * B^T)	设备	88×99×66	49.773 ms	
12	HVX 外积 (A * B)	设备	88×99×66	41.639 ms	

2. 分析要点

2.1. 对比内积与外积在数据复用、内存访问模式与向量指令使用上的差异；

1. 数据复用

- 外积法采用标量广播策略：将 A 矩阵的单个元素 $A[i][l]$ 广播到向量寄存器，在内层循环中被复用 $\lceil \frac{n}{32} \rceil$ 次，实现高效的寄存器级复用。但代价是 C 矩阵的同一位置需要被读-改-写 k 次，频繁的 load&store 操作增加内存带宽压力。
- 内积法则采用向量点积策略：A 矩阵的一行向量 $A[i][l:l+31]$ 在 j 循环中被重复读取 n 次，依赖 L1 Cache 的缓存复用；而 C 矩阵的累加完全在寄存器中完成，最终仅需一次写入。这使得内积法在大 k 值时能充分利用寄存器累加器，避免 C 矩阵的频繁内存访问，但对 A 矩阵的 Cache 命中率有较高要求。

2. 内存访问模式

- 外积法的内存访问特征是“写入密集”：A 与 B 均为行优先顺序访问，Cache 友好；但 C 矩阵在 k 维度的每次迭代中都需要先读取旧值、计算后写回，同一 Cache Line 被反复污染。这种读-改-写模式在小 k 值时影响较小，但 k 值增大时会成为性能瓶颈。
- 内积法的内存访问特征是“读取密集”：B^T 顺序访问，C 仅有一次写入，无读取开销；但 A 的同一行在 j 循环中被重复读取 n 次，若 `k * sizeof(float)` 超过 L1 Cache 容量，则会导致多次从主存读取。因此内积法性能高度依赖硬件 Cache 大小，而外积法对 Cache 的敏感度较低，但受限于内存写入带宽。

3. 向量指令使用差异

- 外积法指令序列：每次处理 32 个 C 元素仅需 1 次标量广播、1 次向量乘法、1 次向量加法与若干格式转换，总计少量指令，适合任意 k 值。
- 内积法需要额外的向量归约步骤：完成向量乘加后，需通过多次向量旋转与加法将 32 个部分和归约为标量，这些归约指令在小 k 值时占比显著。但内积法的优势在于累加器常驻寄存器，最终写回内存次数更少。

2.2. 关键 HVX 指令详解：指出在代码中使用到的每种 HVX 指令（例如 Q6_V_vsplat_R、Q6_Vqf32_vmpy_VsfVsf、Q6_Vqf32_vadd_Vqf32Vqf32、Q6_V_vror_VR 等）并解释它们在你的实现中如何改善性能；

- `Q6_V_vsplat_R`：将 32 位标量值复制到 HVX 向量的所有 32 个元素中（每个元素 4 字节）。在外积法中，每处理 32 个 C 矩阵元素仅需 1 次广播，相当于 32× 数据准备效率提升
- `Q6_Vqf32_vmpy_VsfVsf`：执行 32 个单精度浮点数的并行乘法（SIMD）。1 条指令完成 32 次浮点乘法，可以带来 8-32× 计算吞吐量提升
- `Q6_Vqf32_vadd_Vqf32Vqf32`：累加器更新（内积法）和向量归约，配合 Q6_V_vror_VR 实现树形归约，带来性能提升。
- `Q6_Vsf_equals_Vqf32`：将 QF32 格式（内部高精度）转换为 SF 格式（标准 IEEE 754 单精度）。转换指令延迟 1-2 cycles，远低于重新计算的成本。
- `Q6_V_vror_VR`：向右旋转向量元素（循环移位），用于实现高效的向量归约（Reduction）。每次旋转后的加法可并行执行，延迟约 10-15 cycles，带来 2-3× 归约速度提升。
- `Q6_V_vzero`：创建全零向量，用作累加器的初始值。避免内存操作：无需从内存加载 0 值。

外积实现的指令组合效率

```c
// 每处理 C 矩阵的 32 个元素：
  Q6_V_vsplat_R          // 1 cycle  - 广播
  Q6_Vqf32_vmpy_VsfVsf   // 1 cycle  - 32 次乘法
  Q6_Vsf_equals_Vqf32    // 1 cycle  - 格式转换
  Q6_Vqf32_vadd_VsfVsf   // 1 cycle  - 32 次加法
  Q6_Vsf_equals_Vqf32    // 1 cycle  - 格式转换
  // 总计：5 cycles 完成 32 次 FMA（乘加融合）
```

```c
// 对比标量执行：
  for (int j = 0; j < 32; j++) {
      C[j] += a_val \* B[j];
      // 32 cycles（乘法）+ 32 cycles（加法）= 64 cycles
  }
```

加速比：64 / 5 = 12.8× 理论加速

内积实现的指令组合效率

```c
// 主循环（每处理 32 个点积元素）：
  Q6_Vqf32_vmpy_VsfVsf      // 1 cycle  - 32 次乘法
  Q6_Vqf32_vadd_Vqf32Vqf32  // 1 cycle  - 32 次累加
  // 每次迭代：2 cycles 完成 32 次 FMA

// 归约阶段（每个 C 元素）：
  5 × Q6_V_vror_VR           // 5 cycles  - 旋转
  5 × Q6_Vqf32_vadd_Vqf32Vqf32  // 5 cycles  - 归约加法
  // 总计：10 cycles 完成 32 元素归约
```

```c
// 对比标量执行（k=32 时）：
  float sum = 0;
  for (int l = 0; l < 32; l++) {
      sum += A[l] \* B[l];
      // 64 cycles（乘加）+ 31 cycles（归约）= 95 cycles
  }
```

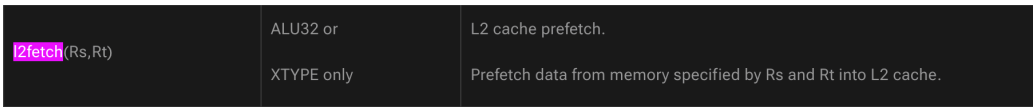
加速比：95 / (2 + 10) = 7.9× 理论加速

2.3. 针对尾部、对齐、缓存与内存带宽瓶颈提出优化建议。

当前实现的最大瓶颈在于 `outer_product` 策略对内存带宽的极端消耗，因为它在 l 循环内部反复读写了整个 C 矩阵。最关键的优化是必须在内层 j 循环中使用 HVX 向量寄存器作为累加器，直到 l 循环完全结束后才将最终结果一次性写回内存。其次，为了消除内层循环中代价高昂的对齐检查分支，应强制要求调用者提供 128 字节对齐的内存（使用 `memalign`），让内核始终执行最高效的对齐加载/存储。同时，应使用 HVX Predicate 来处理尾部数据，以取代低效的标量尾部循环。最后，为处理大矩阵的缓存瓶颈，应采用分块策略，并优先将计算块加载到 VTCM 中执行，以实现零延迟访问。

2.4. 延伸讨论（可选）：

翻阅硬件手册，查看 l2fetch 函数的作用，并尝试用在代码实现中，观察变化



翻阅硬件手册后，了解到 `l2fetch` 函数是一种非阻塞的数据预取指令，其核心作用是在处理器核心实际需要数据之前，异步地将指定内存地址（通常位于主存 DDR）的数据块提前加载到 L2 缓存中，从而降低后续访问延时。