测试结果

实验编号	实现方式	设备/模拟器	矩阵尺寸 (M×K×N)	计算耗时 (ms)	备注
1	朴素 baseline		64×64×64	50.654	
2	HVX 内积 (A * B^T)		64×64×64	42.119	
3	HVX 外积 (A * B)		64×64×64	33.300	
4	朴素 baseline		256×256×256	614.851	
5	HVX 内积 (A * B^T)		256×256×256	84.069	
6	HVX 外积 (A * B)		256×256×256	79.203	
7	朴素 baseline		512×512×512	6249.638	
8	HVX 内积 (A * B^T)		512×512×512	328.263	
9	HVX 外积 (A * B)		512×512×512	340.416	
10	朴素 baseline		88×99×66	66.296	
11	HVX 内积 (A * B^T)		88×99×66	44.099	
12	HVX 外积 (A * B)		88×99×66	45.236	

以上所有数据在设备中测试

内积与外积对比分析

总体上来说，外积计算在性能上优于内积。

数据复用

内积计算法:
对矩阵 A: 在计算结果矩阵 C 的某一行（例如第 i 行）的所有元素时，矩阵 A 的第 i 行会被重复使用 n 次。这具有良好的时间局部性。
对矩阵 B: 在计算结果矩阵 C 的某一列（例如第 j 列）的所有元素时，矩阵 B 的第 j 列会被重复使用 m 次。
问题: 当缓存较小时，计算 C 的下一行元素 C(i+1, j) 时，可能 A 的第 i 行已经被逐出缓存。传统的朴素

三层循环实现（i, j, k）在计算 C(i, j) 和 C(i, j+1) 时，虽然复用了 A(i, :)，但 B 的数据是按列访问的，复用性较差，尤其是当矩阵按行存储时。

外积计算法:

在每次外积的累加中（例如第 l 次循环），算法会读取 A 的第 l 列和 B 的第 l 行。这两个向量会被用来更新 整个 结果矩阵 C 的所有元素。

优势: A 的一列和 B 的一行数据被充分利用，参与了 m * n 次乘法运算，然后才被丢弃。这种方式将对输入矩阵的数据复用最大化。相比内积法，它可以减少对主内存的读取请求。

内存访问模式

内积计算法:

访问矩阵 A: 在计算 C(i, j) 时，对 A 的第 i 行的访问是 顺序访问，这是高效的。

访问矩阵 B: 对 B 的第 j 列的访问是 跨步访问 (Strided Access)。因为内存中连续存放的是一行的数据，要访问一整列就需要跳过一整行的字节数才能读到下一个元素。这是内积法性能不佳的主要原因之一[2]。

访问矩阵 C: 访问 C(i, j) 是一个点，但通常会按行计算，所以对 C 的写入是顺序的。

外积计算法:

访问矩阵 A: 访问 A 的第 l 列是 跨步访问，这同样是低效的。

访问矩阵 B: 访问 B 的第 l 行是 顺序访问，这是高效的。

访问矩阵 C: 在每次外积累加中，需要读写 整个 C 矩阵。这对缓存来说是一个巨大的压力，因为 C 矩阵可能无法完全装入缓存。

改进: 尽管朴素的外积法看起来也有跨步访问的问题，但它可以通过分块（Tiling）技术进行优化。通过将矩阵划分为小的子块，可以确保每次计算的子矩阵（A 的列块、B 的行块和 C 的目标块）都能放入缓存，从而将跨步访问限制在小的缓存块内，并极大提升了数据复用。这种分块策略是现代高性能计算库（如 BLAS）的核心思想。

向量指令

内积计算法:

内积（点积）的计算过程可以很好地向量化。可以使用 SIMD 指令同时加载 A 的一行中的多个元素和 B 的一列中的多个元素，进行乘法和累加操作。

挑战: B 的列访问依然是问题。为了解决这个问题，一个常见的优化是在计算前先将矩阵 B 转置。转置后，对 B 的列访问就变成了对转置后矩阵 B' 的行访问，从而变成顺序访问，非常有利于 SIMD 指令的加载和计算。

外积计算法:

外积的计算模式与 SIMD 指令非常契合。在计算 column_l(A) * row_l(B) 时，可以从 column_l(A) 中取出一个标量 A(i, l)，然后使用 SIMD 的 广播 (Broadcast) 指令将其复制到向量寄存器的所有通道中。

接着，用这个广播后的向量与 row_l(B) 的一部分（一个向量长度）进行 SIMD 乘法，结果再与 C 矩阵

对应位置的向量进行 SIMD 加法。这个过程被称为 向量-标量乘加 (Vector-Scalar Multiply-Add) 操作，效率极高。
这种方式避免了内积法中对 B 的跨步访问问题，使得数据加载和计算都能高效地流水线化

向量指令

1. 标量广播指令 - Q6_V_vsplat_R()： 在外积法中，需要将矩阵A中的一个标量 A(i, k) 与矩阵B的一整行（或其中一段向量）的所有元素相乘。如果用常规方法，需要循环32次，每次都加载同一个标量 A(i, k)。vsplat 指令解决了这个瓶颈。它在一个指令周期内，就将单个标量 A(i, k) 加载并“广播”到一个完整的向量寄存器中，生成一个所有元素都等于 A(i, k) 的向量。
2. 零向量创建 - Q6_V_vzero()：矩阵乘法本质上是一个大规模的累加过程。在计算开始前，用于存储结果矩阵C的子块（通常是几个向量）的寄存器（累加器）必须被初始化为零。vzero 指令提供了一种最快的方式来清空一个向量寄存器。它不需要从内存读取任何数据，也无需循环，一条指令即可生成一个包含32个零的向量。
3. 向量乘法指令 - Q6_Vqf32_vmpy_VsfVsf()：矩阵乘法的核心是海量的乘法运算。传统的标量计算一次只能执行一个乘法。vmpy 指令接收两个向量（例如，一个来自vsplat的广播向量，另一个是从B矩阵加载的行向量），并对它们的对应元素进行并行乘法。一条vmpy指令可以同时完成32次浮点乘法。这直接带来了理论上高达32倍的计算吞吐量提升。
4. 向量加法指令 - Q6_Vqf32_vadd_Vqf32Vqf32()： 乘法之后是累加。在外积法中，C += column_k(A) * row_k(B)。每次vmpy计算出的部分积向量，都需要被加到最终结果C的对应向量上。 一条vadd指令可以同时完成32次浮点加法，与vmpy共同构成了“乘-加”的核心计算流。
5. 向量旋转指令 - Q6_V_vror_VR()：这个指令用于一个相对特殊的场景——向量归约（Reduction），也叫水平求和（Horizontal Sum）。在某些算法实现中（尤其是内积法），我们可能会得到一个向量，其中包含了计算单个目标元素C(i, j)所需的所有部分积，例如[p_1, p_2, ..., p_{32}]。为了得到最终的标量结果，需要将这个向量内的所有元素相加。直接对向量内的元素进行求和很低效。vror (vector rotate right) 提供了一种在寄存器内部高效重排数据的方法。通过一系列的旋转和相加操作，可以并行地完成求和，整个求和过程完全在寄存器内部完成，避免了将向量写回内存再逐个读取相加的巨大开销。

进一步优化

尾部问题

1. 单独循环处理：将循环分为两部分：一个处理大部分能被整除的数据的主循环，和一个处理剩余尾部数据的次循环。主循环可以进行深度优化，如循环展开和向量化，而尾部循环则单独处理。
2. 填充0：在内存中分配矩阵时，可以额外多分配一些空间，将矩阵的维度向上取整到最适合块大小或向量长度的倍数。多出来的“填充”部分可以用0来补齐。这样做的好处是所有计算都可以统一在

高效的主循环中完成，消除了分支预测失败和处理尾部的开销。
3. 掩码（Masking）处理: 许多现代SIMD指令集（如AVX2和AVX-512）支持掩码操作。即使一个向量寄存器中只有部分数据是有效的，也可以通过掩码来指定只对有效的数据进行加载、存储和计算，而忽略无效的“尾部”数据。

对齐问题

1. 使用对齐内存分配器：使用专门的函数来分配对齐的内存，例如C++11的aligned_alloc、Windows的_aligned_malloc或POSIX的posix_memalign。
2. 动态对齐检查与代码分支: 在计算核心代码之前，可以检查关键数据（如矩阵A的行、矩阵B的列）的地址是否对齐。根据检查结果，执行两条不同的代码路径：一条是为对齐数据优化的最高速路径（，另一条是为非对齐数据准备的通用路径。
3. 数据重排 : 在主计算开始前，可以将输入矩阵（或其子块）复制到一个对齐的临时缓冲区（packed buffer）中。虽然这会引入一次数据复制的开销，但后续在计算密集型的内层循环中，所有内存访问都将是对齐且连续的，这种开销通常是值得的。

缓存与内存带宽

1. 分块 : 这是针对缓存最重要的优化。其核心思想是将大矩阵划分为能够完全装入某一级缓存（如L1或L2 Cache）的小子矩阵（Block）。然后，计算的单位从单个元素变为子矩阵。
2. 利用多级缓存: 设计多层次的分块策略。可以设计一个大的块来适应L3缓存，在这个大块内部再划分为更小的块来适应L2或L1缓存。
3. 减少数据移动：如果矩阵乘法是一系列计算中的一步,可以将加法操作与乘法操作融合在同一个循环中。这样，矩阵C的元素被读入后，可以立即与AB的结果相加，而不需要将AB的完整结果写回主存再读出.