

# Lab4文档

## 实验数据记录表

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	模拟器	64×64×64	37ms	/
2	HVX 内积 (A * B^T)	模拟器	64×64×64	4ms	/
3	HVX 外积 (A * B)	模拟器	64×64×64	2ms	/
4	朴素 baseline	模拟器	256×256×2 56	2390ms	/
5	HVX 内积 (A * B^T)	模拟器	256×256×2 56	192ms	/
6	HVX 外积 (A * B)	模拟器	256×256×2 56	157ms	/
7	朴素 baseline	模拟器	512×512×5 12	19083ms	/
8	HVX 内积 (A * B^T)	模拟器	512×512×5 12	1440ms	/
9	HVX 外积 (A * B)	模拟器	512×512×5 12	1260ms	/
10	朴素 baseline	模拟器	88×99×66	82ms	/
11	HVX 内积 (A * B^T)	模拟器	88×99×66	23ms	/
12	HVX 外积 (A * B)	模拟器	88×99×66	14ms	/

# 实验分析

## 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

在HVX向量化矩阵乘法中，内积法使用ijk循环顺序，通过向量化计算A的一行向量与B转置后的一行向量的点积；外积法使用ikj循环顺序，通过向量化j维度将A的一个标量与B的一行向量相乘，并累加到C的行向量上。

### (1) 数据复用与缓存效率对比

#### 内积法：

在寄存器复用方面，内积法仅vAcc\_qf32累加器被高效利用；在缓存复用方面，其j循环需要重复加载A矩阵的第i行，导致其同一行数据被j循环遍历n次，在此行数据无法完全保留在L1缓存中时，会导致大量的缓存未命中或DRAM带宽压力。

#### 外积法：

在寄存器复用方面，C[i][j:j+31]的累加和acc\_qf32被保持在HVX向量寄存器中，并跨越了整个k循环，使得k次累加发生在寄存器内部，完全避免了内存读写；在缓存复用方面，A矩阵的第i行仅在最外层循环被访问一次，其标量元素在k循环中被顺序读取，具有良好的空间局部性。

### (2) 内存访问模式对比

#### 内积法：

在内存读请求时，l循环内部每次迭代需要2次128字节向量加载（用于vA\_sf和vB\_T\_sf）；在内存写请求时，j循环结束后执行一次4字节标量写入以记载最终的点积结果，读写带宽需求严重不平衡。

#### 外积法：

在内存读请求时，l循环内部每次迭代需要1次4字节标量加载和1次128字节向量加载；在内存写请求时，k循环结束时执行一次128字节的向量写入，读写相对均衡特别是向量写入效率较高

### (3) 向量指令使用与额外开销对比

#### 内积法：

使用Q6\_Vqf32\_vmpy\_VsfVsf (向量\*向量)和Q6\_Vqf32\_vadd\_Vqf32Vqf32 (向量+向量)作为核心指令；额外开销方面，l循环结束后，累加器中包含的是32个部分和，为了得到最终的标量点积，必须执行一次“水平求和”归约（代价是5次的Q6\_V\_vror\_VR向量旋转和5次的Q6\_Vqf32\_vadd\_Vqf32Vqf32向量相加）

#### 外积法：

使用Q6\_V\_vsplat\_R (标量广播), Q6\_Vqf32\_vmpy\_VsfVsf (标量\*向量)和Q6\_Vqf32\_vadd\_Vqf32Vqf32 (向量+向量)作为核心指令；额外开销方面，k循环结束后，累加器中存储的**直接就是最终的结果向量**，不需要任何规约操作，只需要一次Q6\_Vsf\_equals\_Vqf32格式转换即可写回内存。

## 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

见下文“代码实现”中注释内容，这里不再进行赘述

## 3.针对尾部、对齐、缓存与内存带宽瓶颈提出优化建议

### （1）关于尾部处理瓶颈

在分配矩阵内存时，对于非32倍数的维度应将其向上取整到32倍数，空余位置使用全0进行填充；此改进可使hvx矩阵乘法中尾部的标量乘法代码永远不被执行，所有数据都由高效的HVX主循环处理，并改善代码的分支预测。

### （2）关于内存对齐瓶颈

不在乘法函数中使用memcpy进行非对齐内存的补救，而应当先天确保送入函数内存的128字节对齐，如调用posix\_memalign或类似功能函数，从而直接使用对齐的指针加载。

### （3）关于缓存瓶颈

矩阵尺寸增大时，数据无法全部装入L1或L2缓存；因此在大矩阵计算时将目标矩阵C分为若干个小的tile，以tile大小64\*64为例，一次只需要矩阵A一个64\*K的行tile和矩阵B一个K\*64的列tile；为防止依旧装不下，可将K维度也进行分块，此时最终循环变为6层，如果A、B、C的tile能够完全装入L2缓存，将极大降低对DRAM内存带宽需求。

### （4）关于内存带宽瓶颈

可在计算当前步骤k的同时，异步对k+N步骤的数据进行预取，或在外积法内展开j循环提高计算密度（通过提升(vmpy+vadd)指令数与load指令数的比例）

## 实验结果展示

（1）在矩阵维度为64 64 64时，朴素baseline（左）和HVX内外积（右）的性能展示如下：

<pre>===== Starting GEMM test in simulator: M=64, K=64, N=64  Calling calculator_gemm (transY=0)... GEMM executed successfully. Time: 37 ms Verification: PASSED (transY=0)  Calling calculator_gemm (transY=1)... GEMM executed successfully. Time: 36 ms Verification: PASSED (transY=1) =====</pre>	<pre>===== Starting GEMM test in simulator: M=64, K=64, N=64  Calling calculator_gemm (transY=0)... GEMM executed successfully. Time: 2 ms Verification: PASSED (transY=0)  Calling calculator_gemm (transY=1)... GEMM executed successfully. Time: 4 ms Verification: PASSED (transY=1) =====</pre>
--	--

（2）在矩阵维度为256 256 256时，朴素baseline（左）和HVX内外积（右）的性能展示如下：

```

=====
Starting GEMM test in simulator: M=256, K=256, N=256

Calling calculator_gemm (transY=0)...
GEMM executed successfully. Time: 2390 ms
Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 2298 ms
Verification: PASSED (transY=1)
=====

```

```

=====
Starting GEMM test in simulator: M=256, K=256, N=256

Calling calculator_gemm (transY=0)...
GEMM executed successfully. Time: 157 ms

Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 192 ms
Verification: PASSED (transY=1)
=====

```

(3) 在矩阵维度为512 512 512时，朴素baseline（左）和HVX内外积（右）的性能展示如下：

```

=====
Starting GEMM test in simulator: M=512, K=512, N=512

Calling calculator_gemm (transY=0)...

GEMM executed successfully. Time: 19083 ms
Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 18350 ms
Verification: PASSED (transY=1)
=====

```

```

=====
Starting GEMM test in simulator: M=512, K=512, N=512

Calling calculator_gemm (transY=0)...
GEMM executed successfully. Time: 1260 ms

Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 1400 ms
Verification: PASSED (transY=1)
=====

```

(4) 在矩阵维度为88 99 66时，朴素baseline（左）和HVX内外积（右）的性能展示如下：

```

=====
Starting GEMM test in simulator: M=88, K=99, N=66

Calling calculator_gemm (transY=0)...
GEMM executed successfully. Time: 82 ms
Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 79 ms
Verification: PASSED (transY=1)
=====

```

```

=====
Starting GEMM test in simulator: M=88, K=99, N=66

Calling calculator_gemm (transY=0)...
GEMM executed successfully. Time: 14 ms
Verification: PASSED (transY=0)

Calling calculator_gemm (transY=1)...
GEMM executed successfully. Time: 23 ms
Verification: PASSED (transY=1)
=====

```

## 代码实现

### 代码块

```

1  //指令加速原理
2  /*
3   1.Q6_V_vsplat_R: 使标量值能够与B矩阵的一整行向量（32个元素）同时进行运算
4   2.Q6_Vqf32_vmpy_VsfVsF: 将内层循环中32次标量乘法压缩为一条指令以实现32倍理论吞吐量
5   3.Q6_Vqf32_vadd_Vqf32Vqf32: 一次性执行32次浮点加法，将32个乘法结果高效地累加到累加器向量
6   4.Q6_V_vror_VR: 通过高效的指定字节数循环右移使vadd指令能将32个部分和快速汇总为1个总和
7  */
8  #define VLEN 32
9  #define VLEN_BYTES 128
10 //这个宏定义用于检查内存是否128字节对齐，如果不对齐再使用memcpy
11 #define IS_ALIGNED(ptr) (((uintptr_t)(ptr) & (VLEN_BYTES - 1)) == 0)
12 static inline int32_t float_to_bits(float input)
13 {
14     union {
15         float f;
16         int32_t i;

```

```

17     } fp32 = {.f = input};
18     return fp32.i;
19 }
20 /*HVX外积实现*/
21 static inline void matmul_ijk(float *restrict input_matrix1,
22                               float *restrict input_matrix2,
23                               float *restrict output,
24                               uint32_t m,
25                               uint32_t k,
26                               uint32_t n) {
27     for (int i = 0; i < m; i++) {
28         int j = 0;
29         const int n_vec_loops = n / VLEN;
30         for (j = 0; j < n_vec_loops * VLEN; j += VLEN) {
31             //使用Q6_V_vzero初始化累加器 (QF32格式, 用于累加 C[i][j:j+31] 的结果
32             HVX_Vector vC_acc_qf32 = Q6_V_vzero();
33             for (int l = 0; l < k; l++) {
34                 // 加载 A[i][l], 并利用Q6_V_vsplat_R将其广播到 32 个通道
35                 float a_scalar = input_matrix1[i * k + l];
36                 HVX_Vector vA_scalar_sf =
37                 Q6_V_vsplat_R(float_to_bits(a_scalar));
38                 float *pB = input_matrix2 + l * n + j;
39                 HVX_Vector vB_loaded_sf;
40                 if (IS_ALIGNED(pB)) {
41                     vB_loaded_sf = *((HVX_Vector *)pB);
42                 } else {
43                     __attribute__((aligned(VLEN_BYTES))) float b_buf[VLEN];
44                     memcpy(b_buf, pB, VLEN_BYTES);
45                     vB_loaded_sf = *((HVX_Vector *)b_buf);
46                 }
47                 //使用Q6_Vqf32_vmpy_VsfVsf进行向量逐元素相乘
48                 HVX_Vector vMul_qf32 = Q6_Vqf32_vmpy_VsfVsf(vA_scalar_sf,
49                 vB_loaded_sf);
50                 //使用Q6_Vqf32_vadd_Vqf32Vqf32进行向量逐元素相加
51                 vC_acc_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(vC_acc_qf32, vMul_qf32);
52             }
53             //使用Q6_Vsf_equals_Vqf32将高精度累加结果转回标准 float 格式
54             HVX_Vector vC_final_sf = Q6_Vsf_equals_Vqf32(vC_acc_qf32);
55             float *pC = output + i * n + j;
56             if (IS_ALIGNED(pC)) {
57                 *((HVX_Vector *)pC) = vC_final_sf;
58             } else {
59                 __attribute__((aligned(VLEN_BYTES))) float c_buf[VLEN];
60                 *((HVX_Vector *)c_buf) = vC_final_sf;
61                 memcpy(pC, c_buf, VLEN_BYTES);
62             }
63         }
64     }
65 }

```

```

62         // 7. 利用标量实现，边界处理N维度上剩余的n%32个元素
63         for (; j < n; j++) {
64             float sum = 0.0f;
65             for (int l = 0; l < k; l++) {
66                 sum += input_matrix1[i * k + l] * input_matrix2[l * n + j];
67             }
68             output[i * n + j] = sum;
69         }
70     }
71     return;
72 }
73 /*辅助函数：HVX向量规约，将一个QF32的所有32个元素相加，并返回float标量*/
74 /*核心思想是使用5次旋转和5次求和得到整个向量之和*/
75 static inline float hsum_qf32_reduction(HVX_Vector v_qf32) {
76     HVX_Vector vRot;
77     // 使用Q6_V_vror_VR旋转16个float并使用Q6_Vqf32_vadd_Vqf32Vqf32执行向量加法
78     vRot = Q6_V_vror_VR(v_qf32, 16 * sizeof(float));
79     v_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(v_qf32, vRot);
80     vRot = Q6_V_vror_VR(v_qf32, 8 * sizeof(float));
81     v_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(v_qf32, vRot);
82     vRot = Q6_V_vror_VR(v_qf32, 4 * sizeof(float));
83     v_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(v_qf32, vRot);
84     vRot = Q6_V_vror_VR(v_qf32, 2 * sizeof(float));
85     v_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(v_qf32, vRot);
86     vRot = Q6_V_vror_VR(v_qf32, 1 * sizeof(float));
87     v_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(v_qf32, vRot);
88     // 此时，v_qf32 的所有元素都等于总和 (QF32 格式)
89     HVX_Vector vSum_sf = Q6_Vsf_equals_Vqf32(v_qf32);
90     __attribute__((aligned(VLEN_BYTES))) float temp_arr[VLEN];
91     *((HVX_Vector *)temp_arr) = vSum_sf;
92     return temp_arr[0];
93 }
94 /*HVX内积实现*/
95 static inline void matmul_ikj_transposed_b(float *restrict input_matrix1,
96                                             float *restrict input_matrix2,
97                                             float *restrict output,
98                                             uint32_t m,
99                                             uint32_t k,
100                                             uint32_t n) {
101     for (int i = 0; i < m; i++) {
102         for (int j = 0; j < n; j++) {
103             //使用Q6_V_vzero初始化累加器
104             HVX_Vector vAcc_qf32 = Q6_V_vzero();
105             int l = 0;
106             const int k_vec_loops = k / VLEN;
107             for (l = 0; l < k_vec_loops * VLEN; l += VLEN) {
108                 float *pA = input_matrix1 + i * k + l;

```

```

109         HVX_Vector vA_sf;
110         if (IS_ALIGNED(pA)) {
111             vA_sf = *((HVX_Vector *)pA);
112         } else {
113             __attribute__((aligned(VLEN_BYTES))) float bufA[VLEN];
114             memcpy(bufA, pA, VLEN_BYTES);
115             vA_sf = *((HVX_Vector *)bufA);
116         }
117         float *pB_T = input_matrix2 + j * k + l;
118         HVX_Vector vB_T_sf;
119         if (IS_ALIGNED(pB_T)) {
120             vB_T_sf = *((HVX_Vector *)pB_T);
121         } else {
122             __attribute__((aligned(VLEN_BYTES))) float bufB[VLEN];
123             memcpy(bufB, pB_T, VLEN_BYTES);
124             vB_T_sf = *((HVX_Vector *)bufB);
125         }
126         //使用Q6_Vqf32_vmpy_VsfVsf执行向量乘法
127         HVX_Vector vMul_qf32 = Q6_Vqf32_vmpy_VsfVsf(vA_sf, vB_T_sf);
128         //使用Q6_Vqf32_vadd_Vqf32Vqf32执行向量加法
129         vAcc_qf32 = Q6_Vqf32_vadd_Vqf32Vqf32(vAcc_qf32, vMul_qf32);
130     }
131     // 使用上面的辅助函数进行规约，将vAcc中的32个元素相加
132     float vec_sum = hsum_qf32_reduction(vAcc_qf32);
133     // 7. 边界处理（尾部循环），使用标量乘法处理尾部的k%32个元素
134     float tail_sum = 0.0f;
135     for (; l < k; l++) {
136         tail_sum += input_matrix1[i * k + l] * input_matrix2[j * k +
137         l];
138     }
139     output[i * n + j] = vec_sum + tail_sum;
140 }
141 return;
142 }

```