

实验四: Hexagon 矩阵乘法

姓名：潘梓月 学号：22551192 指导老师：赵新奎 实验日期：2025/10/13、2025/10/20

实验报告说明：

- 1环境准备 和 2编译运行 在md文件基础上，将部分bash替换为**实际运行的命令行**(用户名、文件位置等)，并附上相关步骤成功的**截图**。此外也标注了一些实验中遇到的问题（python和java缺失）
- 3优化矩阵乘法 是本人针对给出的初始代码进行优化的思路、部分代码展示、步骤以及优化结果分析，这是**10.13的中间实验记录**
- 4实验最终报告 完善了10.20新增的实验目标和记录表部分，并进行相关分析，这是**10.20的最终实验报告**

1 环境准备

1.1Hexagon SDK 安装

高通账号已申请通过

1.1.1 下载并安装 QPM

1. 下载 QPM

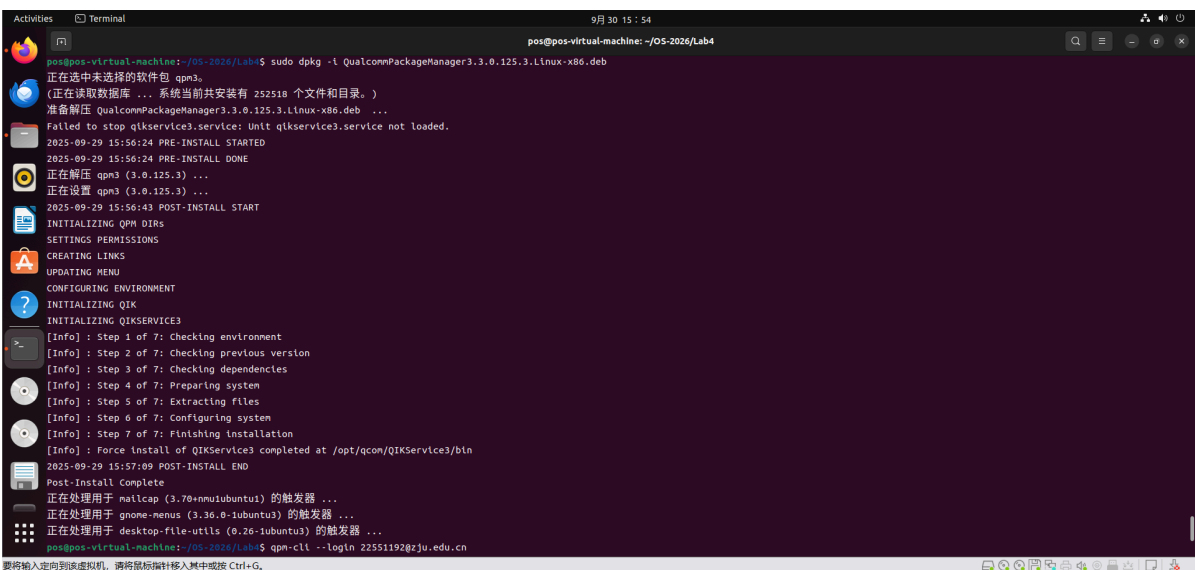
访问 [QPM 官方下载页面](#) 下载安装包。

2. 安装系统依赖

```
sudo apt update && sudo apt install -y bc xdg-utils
```

3. 安装 QPM

```
sudo dpkg -i QualcommPackageManager3.3.0.125.3.Linux-x86.deb
```



1.1.2 QPM 登录与配置

1. 登录 QPM

```
qpm-cli --login 22551192@zju.edu.cn
```

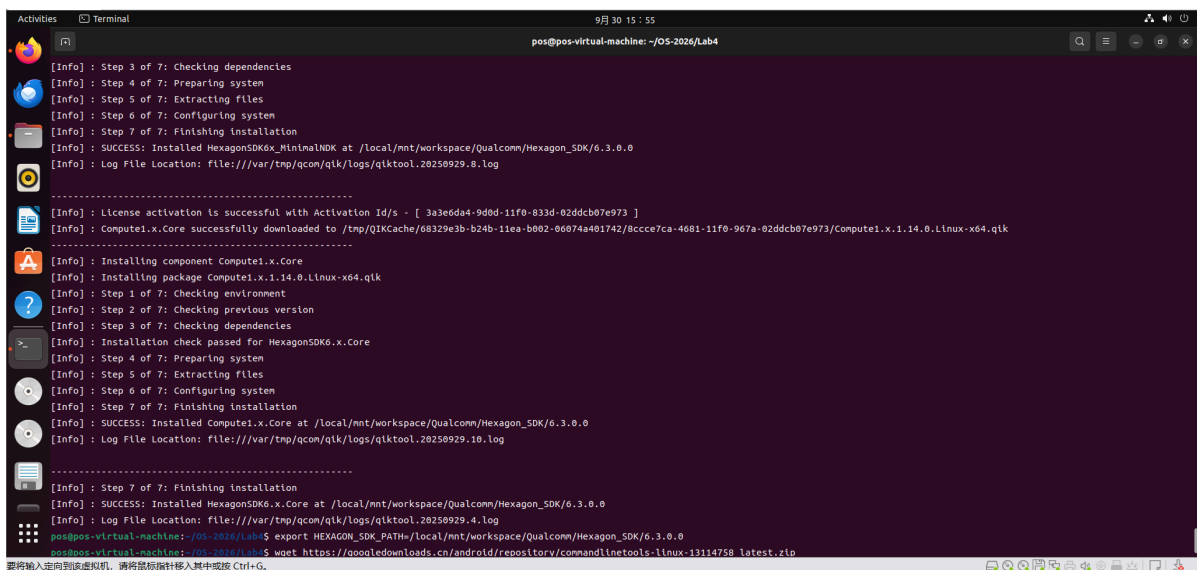
2. 激活许可证并安装 Hexagon SDK

```
qpm-cli --license-activate HexagonSDK6.x  
sudo qpm-cli --install HexagonSDK6.x
```

3. 配置环境变量

在 `~/.bashrc` 中添加:

```
export HEXAGON_SDK_PATH=/local/mnt/workspace/Qualcomm/Hexagon_SDK/6.3.0.0
```



1.2 Android NDK 安装

1.2.1 下载并配置 Android SDK Command Line Tools

1. 下载 Command Line Tools

```
wget https://googledownloads.cn/android/repository/commandlinetools-linux-13114758_latest.zip
```

2. 解压并组织目录结构

```
unzip commandlinetools-linux-13114758_latest.zip  
mkdir -p ~/Android/Sdk/cmdline-tools/latest  
mv cmdline-tools/* ~/Android/Sdk/cmdline-tools/latest/
```

3. 配置环境变量

在 `~/.bashrc` 中添加:

```
export PATH=~/Android/Sdk/cmdline-tools/latest/bin/:$PATH
```

使配置生效：

```
source ~/.bashrc
```

1.2.2 安装 NDK

1. 查看可用版本并安装

```
sdkmanager --list
// 这个地方提示 JAVA_HOME not set and no 'java' command is found.
sdkmanager "ndk;29.0.13113456"
```

2. 配置 NDK 环境变量

在 `~/.bashrc` 中添加：

```
export ANDROID_NDK_ROOT=~/.Android/Sdk/ndk/29.0.13113456/
```

3. 安装 ADB 工具

```
sudo apt install android-tools-adb
```

2 编译与运行

2.1 编译 NPU 代码

首先设置 Hexagon SDK 环境并编译 DSP 代码：

```
source $HEXAGON_SDK_PATH/setup_sdk_env.source
// 提示未安装python
cd dsp
make hexagon BUILD=Debug DSP_ARCH=v79
```

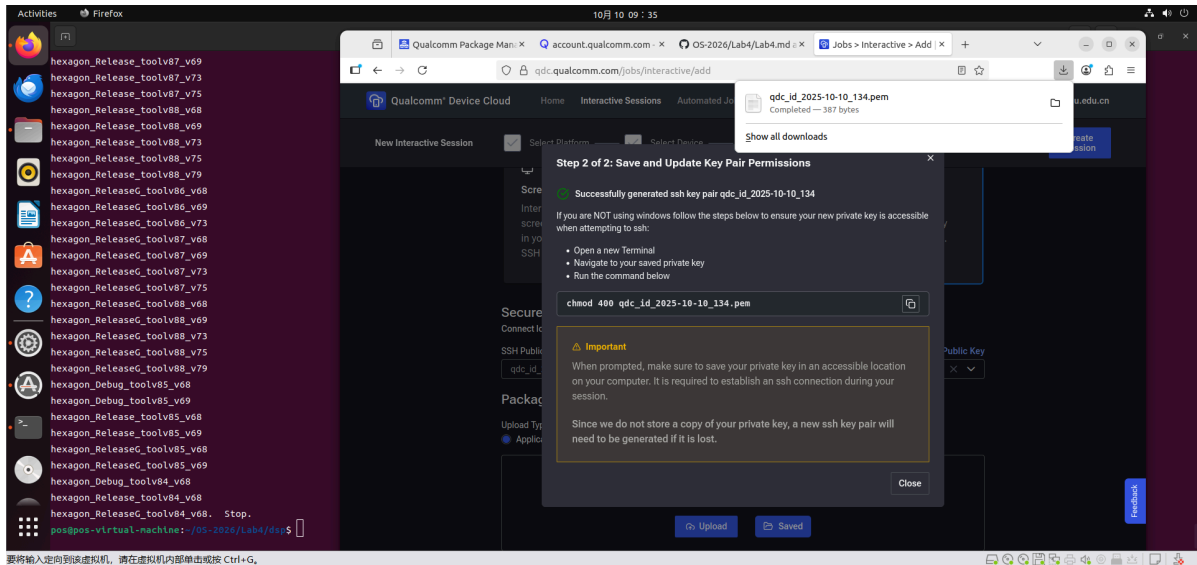


2.2 实体设备运行

2.2.1 申请 QDC 设备

1. 在 [高通 QDC 平台](#) 申请一台骁龙 8 Elite 手机
2. 选择 SSH 连接方式
3. 创建私钥并保存到 `~/qdc.pem`，修改权限：

```
chmod 400 ~/qdc.pem
```

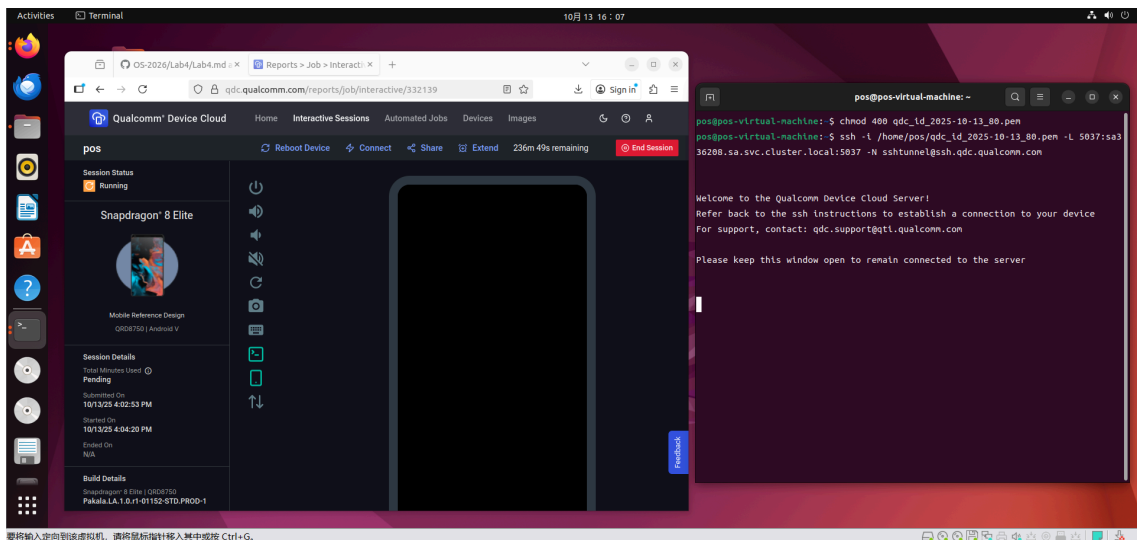


2.2.2 建设设备连接

1. 创建 SSH 隧道

点击 QDC 页面右上角的【Connect】按钮，复制连接命令，例如：

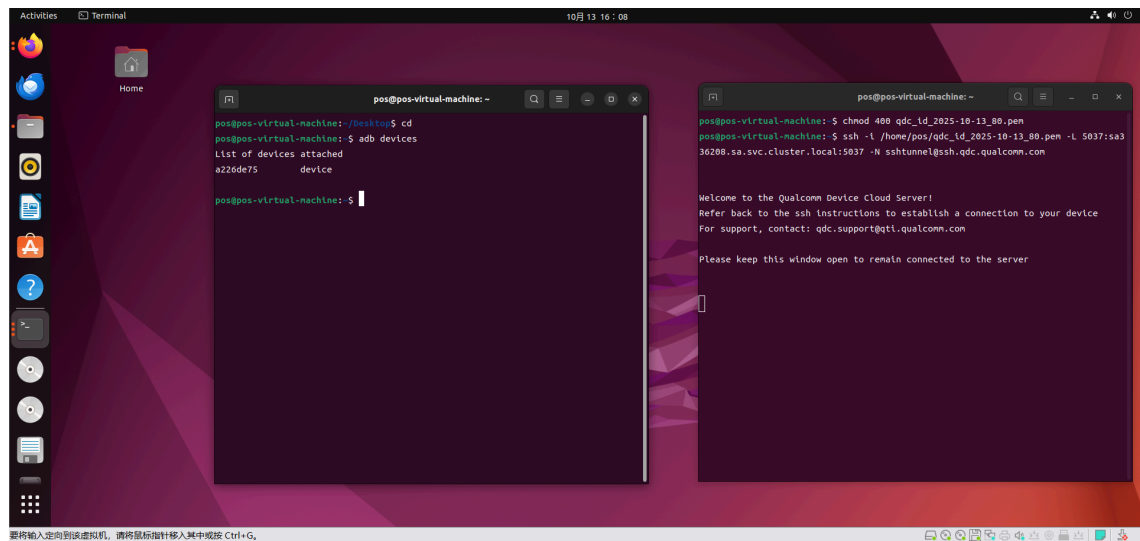
```
ssh -i ~/qdc.pem -L 5037:sa324277.sa.svc.cluster.local:5037 -N  
sshtunnel@ssh.qdc.qualcomm.com
```



2. 验证设备连接

```
adb devices
```

能看到已连接的设备。



2.2.3 编译 Android 测试工具

```
export HEXAGON_SDK_PATH=/local/mnt/workspace/Qualcomm/Hexagon_SDK/6.3.0.0
export HEXAGON_SDK_ROOT=/local/mnt/workspace/Qualcomm/Hexagon_SDK/6.3.0.0
export ANDROID_NDK_ROOT=~/.Android/sdk/ndk/29.0.13113456/
source ~/.bashrc

source $HEXAGON_SDK_PATH/setup_sdk_env.source
mkdir build
cd build
cmake -DANDROID_ABI=arm64-v8a -DANDROID_PLATFORM=android-24 \
      -DCMAKE_TOOLCHAIN_FILE=$ANDROID_NDK_ROOT/build/cmake/android.toolchain.cmake \
      -DHEXAGON_SDK_ROOT=$HEXAGON_SDK_ROOT ..
make
```

- **libinfo.so.5报错**: 如果遇到缺少libinfo5的报错, 执行 `sudo apt install libtinfo5` 或者执行 `ln -s /usr/lib/x86_64-linux-gnu/libtinfo.so.6 /usr/lib/x86_64-linux-gnu/libtinfo.so.5`

2.2.4 部署并运行测试

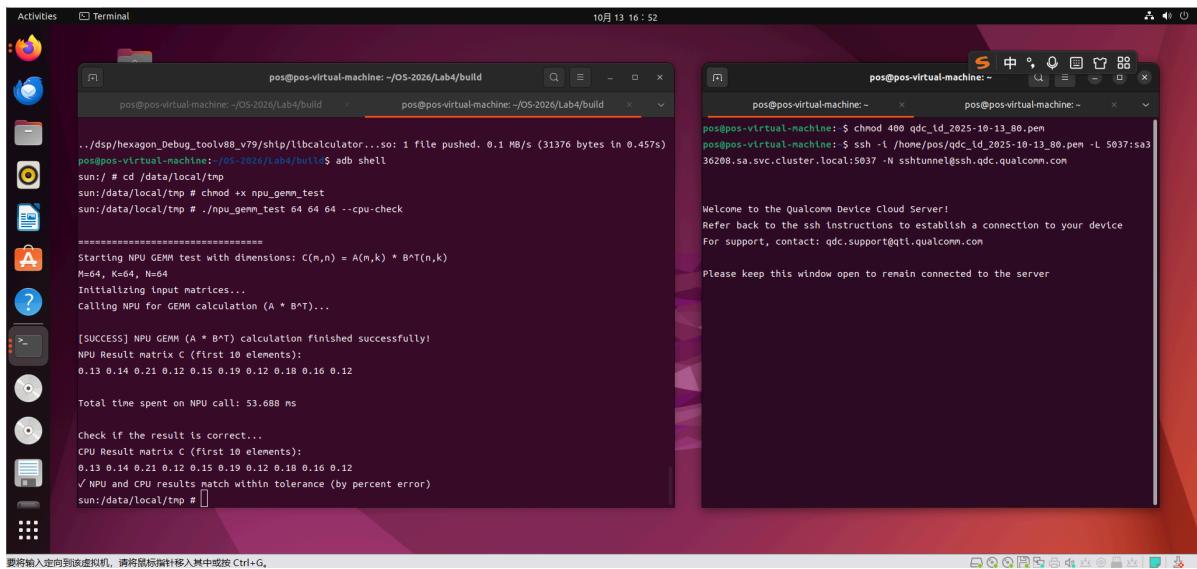
1. 推送文件到设备

```
adb push npu_gemm_test /data/local/tmp/
adb push ../dsp/hexagon_Debug_toolv88_v79/ship/libcalculator_skel.so
/data/local/tmp/
```

2. 在设备上执行测试

```
adb shell
cd /data/local/tmp
chmod +x npu_gemm_test
./npu_gemm_test 64 64 64 --cpu-check
```

可以看到原本算法执行的Total time spent on NPU call: **53.688 ms**



3 优化矩阵乘法

3.1 思路

【向量化】HVX 指令可以一次性处理 32 个浮点数（一个向量）。我重写了两个关键的矩阵乘法函数，让它们尽可能地使用向量指令来代替原来的 `for` 循环。

1. 对于 $A \times B^T$ （转置乘法），采用内积策略：将矩阵 A 和 B 的两行数据（各 32 个元素）加载到向量寄存器中，进行向量乘法和累加。最后通过一系列“归约”操作，将向量中累加的 32 个部分和汇总成一个最终结果。
2. 对于 $A \times B$ （标准乘法），采用更高效的外积策略：将矩阵 A 的一个元素“广播”（复制）成一个包含 32 个相同值的向量，然后与矩阵 B 的一整行进行向量乘法和加法，一次性更新 32 个输出结果。

3.2 部分代码展示

```
79 #ifdef __hexagon__
80     memset(output, 0, m * n * sizeof(float));
81     const int hvx_width = sizeof(HVX_Vector) / sizeof(float);
82     int n_vec = n - (n % hvx_width);
83
84     for (int i = 0; i < m; i++) {
85         for (int l = 0; l < k; l++) {
86             HVX_Vector vA_splat = Q6_Vvsplat_R(float_to_bits(input_matrix1[i * k + l]));
87             for (int j = 0; j < n_vec; j += hvx_width) {
88                 HVX_Vector* pB = (HVX_Vector*)&input_matrix2[l * n + j];
89                 HVX_Vector* pC = (HVX_Vector*)&output[i * n + j];
90
91                 // 使用標準浮點乘法，結果也是標準浮點向量
92                 HVX_Vector vprod = Q6_Vsf_vmpy_VsfVsf(vA_splat, *pB);
93                 // 直接進行標準浮點向量加法
94                 *pC = Q6_Vsf_vadd_VsfVsf(*pC, vprod);
95             }
96             for (int j = n_vec; j < n; j++) {
97                 output[i * n + j] += input_matrix1[i * k + l] * input_matrix2[l * n + j];
98             }
99         }
100     }
```

```

122 #ifdef __hexagon__
123     const int hvx_width = sizeof(HVX_Vector) / sizeof(float);
124     int k_vec = k - (k % hvx_width);
125
126     for (int i = 0; i < m; i++) {
127         for (int j = 0; j < n; j++) {
128             // 累加器使用標準的 HVX_Vector
129             HVX_Vector v_sum = Q6_V_vzero();
130             for (int l = 0; l < k_vec; l += hvx_width) {
131                 HVX_Vector* pA = (HVX_Vector*)&input_matrix1[i * k + l];
132                 HVX_Vector* pB = (HVX_Vector*)&input_matrix2[j * k + l];
133                 // 使用標準浮點乘加
134                 v_sum = Q6_Vsf_vadd_VsfVsf(v_sum, Q6_Vsf_vmpy_VsfVsf(*pA, *pB));
135             }
136
137             // 對標準浮點向量進行歸約
138             for (int r = hvx_width / 2; r > 0; r /= 2) {
139                 v_sum = Q6_Vsf_vadd_VsfVsf(v_sum, Q6_V_vror_VR(v_sum, r * sizeof(float)));
140             }
141
142             float sum;
143             memcpy(&sum, &v_sum, sizeof(float));
144
145             for (int l = k_vec; l < k; l++) {
146                 sum += input_matrix1[i * k + l] * input_matrix2[j * k + l];
147             }
148             output[i * n + j] = sum;
149         }
150     }

```

3.3 步驟

我们发现项目存在两套独立的编译系统（CPU端和DSP端）。最初只编译了 CPU 测试程序，而没有重新编译 DSP 库。正确的做法是必须进入 `dsp` 目录单独执行 `make` 命令，这样才能生成修改后的 `libcalculator_skel.so` 新文件。

3.3.1 设定环境

```

cd /local/mnt/workspace/Qualcomm/Hexagon_SDK/6.3.0.0
source ./setup_sdk_env.source

```

3.3.2 重新编译

```

cd ~/OS-2026/Lab4/dsp
make clean V=hexagon_Debug_toolv88_v79
make tree V=hexagon_Debug_toolv88_v79

```

3.3.3 部署运行

重复2.2.3和2.2.4中的步骤，注意先清除build文件

```
rm -rf build
```

3.3.4 优化结果分析

由于代码中只输出NPU调用的总时间，为精确对比DSP/NPU计算耗时，执行如下命令：

```
adb logcat -s calculator
```

观察到初始代码的DSP/NPU计算耗时为7.943ms，函数总执行耗时50.454ms


```
10-13 05:35:45.376 12445 12445 D calculator: 初始化 DSP 环境
10-13 05:35:45.376 12445 12445 I calculator: DSP_LIBRARY_PATH set to: /data/local/tmp
10-13 05:35:45.376 12445 12445 I calculator: [PROFILING] DSP 环境初始化耗时: 0.005 ms
10-13 05:35:45.376 12445 12445 I calculator: GEMM: m=64, k=64, n=64
10-13 05:35:45.376 12445 12445 I calculator: GEMM: Allocating ION memory for inputs and output...
10-13 05:35:45.376 12445 12445 I calculator: [PROFILING] ION 内存分配耗时: 0.062 ms
10-13 05:35:45.376 12445 12445 I calculator: [PROFILING] 数据传输 (Host -> Device) 耗时: 0.006 ms
10-13 05:35:45.376 12445 12445 I calculator: GEMM: Opening handle...
10-13 05:35:45.413 12445 12445 I calculator: [PROFILING] DSP 句柄准备耗时: 36.889 ms
10-13 05:35:45.413 12445 12445 I calculator: GEMM: Calling remote function calculator_gemm...
10-13 05:35:45.421 12445 12445 I calculator: [PROFILING] DSP/NPU 计算耗时: 7.943 ms
10-13 05:35:45.421 12445 12445 I calculator: GEMM: Remote call successful.
10-13 05:35:45.421 12445 12445 I calculator: [PROFILING] 数据传回 (Device -> Host) 耗时: 0.004 ms
10-13 05:35:45.421 12445 12445 I calculator: GEMM: Cleaning up resources...
10-13 05:35:45.426 12445 12445 I calculator: [PROFILING] 资源清理耗时: 5.247 ms
10-13 05:35:45.426 12445 12445 I calculator: [PROFILING] 函数总执行耗时: 50.454 ms
```

观察到优化后代码的DSP/NPU计算耗时为3.984ms，函数总执行耗时37.590ms

```
10-13 06:13:09.539 17873 17873 D calculator: 初始化 DSP 环境
10-13 06:13:09.539 17873 17873 I calculator: DSP_LIBRARY_PATH set to: /data/local/tmp
10-13 06:13:09.539 17873 17873 I calculator: [PROFILING] DSP 环境初始化耗时: 0.038 ms
10-13 06:13:09.539 17873 17873 I calculator: GEMM: m=64, k=64, n=64
10-13 06:13:09.539 17873 17873 I calculator: GEMM: Allocating ION memory for inputs and output...
10-13 06:13:09.539 17873 17873 I calculator: [PROFILING] ION 内存分配耗时: 0.062 ms
10-13 06:13:09.539 17873 17873 I calculator: [PROFILING] 数据传输 (Host -> Device) 耗时: 0.006 ms
10-13 06:13:09.539 17873 17873 I calculator: GEMM: Opening handle...
10-13 06:13:09.567 17873 17873 I calculator: [PROFILING] DSP 句柄准备耗时: 28.145 ms
10-13 06:13:09.567 17873 17873 I calculator: GEMM: Calling remote function calculator_gemm...
10-13 06:13:09.571 17873 17873 I calculator: [PROFILING] DSP/NPU 计算耗时: 3.984 ms
10-13 06:13:09.571 17873 17873 I calculator: GEMM: Remote call successful.
10-13 06:13:09.571 17873 17873 I calculator: [PROFILING] 数据传回 (Device -> Host) 耗时: 0.004 ms
10-13 06:13:09.571 17873 17873 I calculator: GEMM: Cleaning up resources...
10-13 06:13:09.576 17873 17873 I calculator: [PROFILING] 资源清理耗时: 4.975 ms
10-13 06:13:09.577 17873 17873 I calculator: [PROFILING] 函数总执行耗时: 37.590 ms
```

由此可见，优化后的代码计算速度提升了99.4%，几乎达到了2倍的性能飞跃；总调用延迟降低了12.864ms，端到端的整体性能提升了25.5%。

4 实验最终报告

4.1 任务目标

在 `Lab4/dsp/calculator_imp.c` 中实现并比较多种矩阵乘法实现：

- 朴素标量实现 (baseline)：直接三重循环实现的 C 语言矩阵乘法，用作基线性能对比；
- 基于 HVX 的内积实现 ($A * B^T$)：对 B 做转置，使得点积 (dot-product) 可以用向量化内积 (内积法) 计算；
- 基于 HVX 的外积实现 ($A * B$)：采用外积法，利用标量广播将 A 的单个元素与 B 的一整段向量相乘并累加到 C 的子向量。

4.2 实现要求与验收准则

- 功能等价：对任意合法输入（浮点矩阵）都应输出误差在浮点容差内的结果；
已满足，见4.3
- 向量化与对齐：HVX 代码应处理 128 字节 (32 float) 对齐，说明如何处理尾部不对齐或非 32 倍长度的情形

已说明，见4.4.4

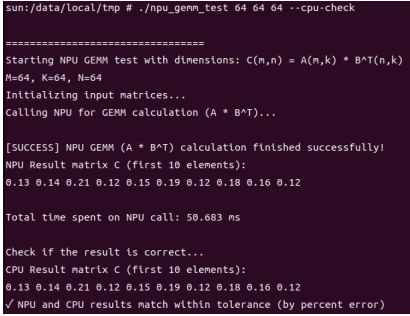
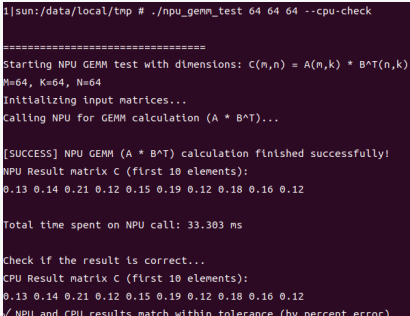
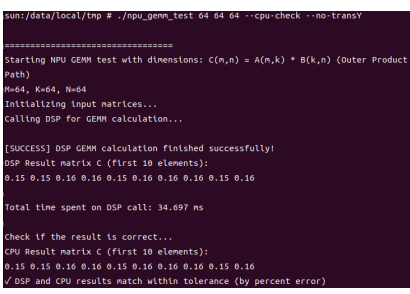
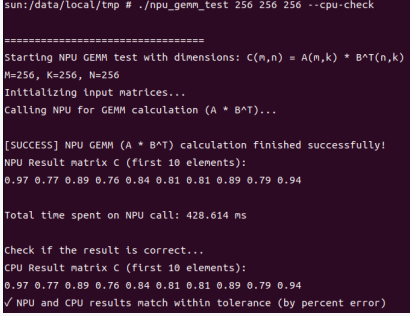
3. 性能测量：对不同矩阵尺寸执行并记录

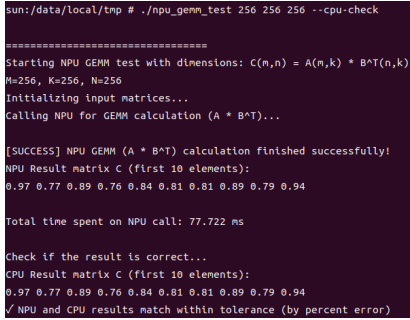
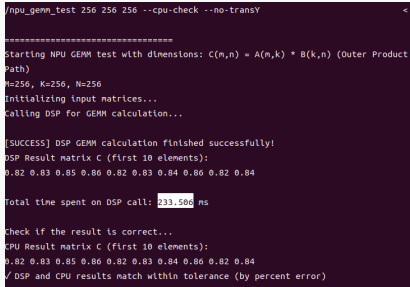
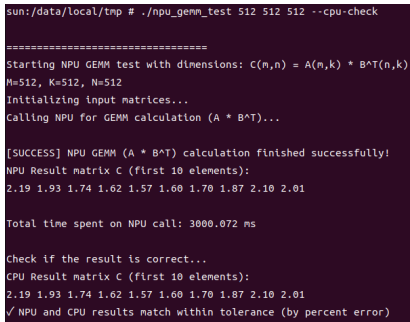
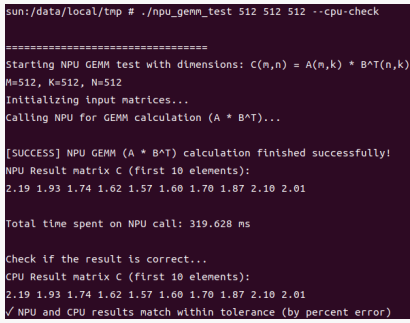
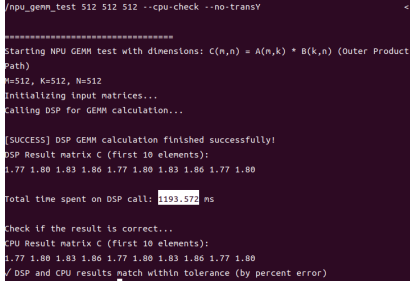
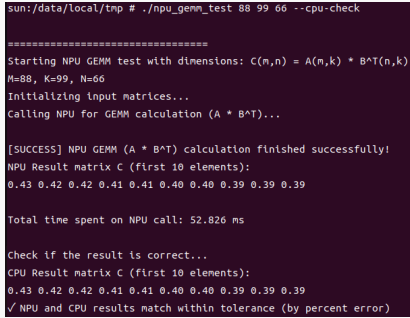
已记录，见4.3

4. 结果分析：比较三种实现的运行时间与计算效率，并说明在实现中使用到的主要 HVX 指令与它们的作用（例如 vsplat、vmpy、vadd、vror 等）

已分析，见4.4.1、4.4.2

4.3 实验数据记录表

编号	实现方法	设备	矩阵尺寸	计算耗时 (ms)	运行截图
1	baseline		64×64×64	50.683	
2	HVX 内积		64×64×64	33.303	
3	HVX 外积		64×64×64	34.697	
4	baseline		256×256×256	428.614	

编号	实现方法	设备	矩阵尺寸	计算耗时 (ms)	运行截图
5	HVX 内积		256×256×256	77.722	
6	HVX 外积		256×256×256	233.506	
7	baseline		512×512×512	3000.072	
8	HVX 内积		512×512×512	319.628	
9	HVX 外积		512×512×512	1193.572	
10	baseline		88×99×66	52.826	

编号	实现方法	设备	矩阵尺寸	计算耗时 (ms)	运行截图
11	HVX 内积		88×99×66	38.224	
12	HVX 外积		88×99×66	40.938	

4.4 分析要点

4.4.1 内外积差异

从实验数据中可见，对于大规模矩阵，HVX 内积 ($A * B^T$) 的性能显著优于 HVX 外积 ($A * B$)。

- 在 512x512x512 尺寸下，内积实现 (319.6ms) 比外积实现 (1193.6ms) 快了近 3.7 倍；
- 在 256x256x256 尺寸下，内积实现 (77.7ms) 也比外积实现 (233.5ms) 快了近 3.0 倍。
- 在 64x64x64 和 88x99x66 这样的小尺寸矩阵上，两者性能接近。这是因为矩阵太小，RPC 调用、内存分配和线程调度的固定开销 (3.3.4中有说明) 占了总耗时的大头，导致计算本身的优化效果被掩盖了。

造成这种巨大性能差异的核心原因在于内存访问模式和数据复用：

1. HVX 内积 ($A * B^T$):

- 内存访问：此实现计算 `C[i][j]` 的值。在最内层循环 (`k` 维度) 中，它连续加载 `A[i][k...k+31]` 和 `B[j][k...k+31]` 的数据。由于 `B` 矩阵已经转置，`B[j][k...k+31]` 在内存中也是连续存储的。这种双流连续读取的方式对缓存极其友好。
- 数据复用：向量累加器 `v_sum` 在整个 `k` 循环中都保持在 DSP 的向量寄存器中，进行了大量的计算 (`vmpy / vadd`)，直到循环结束才将最终的标量和写回主内存的 `C[i][j]` 位置。内存写操作被降到了最低。

2. HVX 外积 ($A * B$):

- 内存访问：此实现 (`ikj` 循环顺序) 在最内层循环 (`j` 维度) 中，重复加载 `B[k][j...j+31]` 的数据，并反复读写 `C[i][j...j+31]` 的数据。
- 性能瓶颈：问题出在对 `C` 矩阵的访问上。在 `k` 循环的每一次迭代中，都需要将 `C` 矩阵的一整行读入向量寄存器，与 `A*B` 的积相加后，再写回内存。这导致了 `M * K` 次的内存读和 `M * K` 次的内存写操作，造成了灾难性的内存带宽瓶颈，严重拖慢了计算速度。

结论：内积法通过转置矩阵 `B`，巧妙地将内存访问优化为两次连续读取和一次最终写入，最大化了计算密度和寄存器复用。而外积法受限于其算法，在 `k` 循环中产生了大量的内存读写，导致其性能远逊于内积法。

4.4.2 关键 HVX 指令详解

1. `vsplat` (标量广播)

- 作用：将一个 32 位的标量浮点数复制 32 次，“广播”成一个 128 字节的 HVX 向量。
- 应用：在 HVX 外积实现中，用它将 `A[i][k]` 的单个元素广播成一个向量 `vA_splat`，以便后续能与 `B` 矩阵的一整行（32 个元素）进行并行乘法。

2. `vmpy` (向量浮点乘法)

- 作用：获取两个 HVX 向量，对它们各自对应的 32 个浮点数进行逐元素相乘，返回一个包含 32 个乘积的 HVX 向量。
- 应用：在内积和外积中都用作主要的计算单元。

3. `vadd` (向量浮点加法)

- 作用：获取两个 HVX 向量，对它们各自对应的 32 个浮点数进行逐元素相加。
- 应用：在内积中，用它将每一步的乘积累加到向量累加器 `v_sum` 中；在外积中，用它来将计算结果累加到输出矩阵 `C` 的对应位置。

4. `vrrot` (向量旋转)

- 作用：将一个 HVX 向量中的所有字节（128 字节）进行循环右移。
- 应用：在内积中，用于实现“归约”。当 `k` 循环结束后，`v_sum` 中包含了 32 个部分和。我们通过 `vrrot` 将向量旋转一半（64 字节），然后与自身相加，这样 32 个和就变成了 16 个；重复此过程（旋转 32 字节、16 字节...），最终可以将所有 32 个部分和高效地累加为向量中的第一个元素，得到最终的标量和。

4.4.3 优化建议

1. 尾部处理

当前我们采用了“主循环 + 标量尾部循环”的策略。对于追求极致性能的场景，这个标量尾部循环可以被“向量掩码 (Vector Masking)”所取代。即我们仍然加载 32 个元素，但使用一个特殊的掩码 (Mask) 告诉 HVX 单元只计算并写回前 3 个元素的结果，忽略后 29 个。这避免了切换回标量计算的开销。

2. 内存对齐

当前实现：在 HVX 函数中，我们使用了 `memcpy` 来安全地将数据从可能未对齐的内存地址加载到一个栈上分配的、保证对齐的 `HVX_Vector` 临时变量中。但 `memcpy` 带来了额外的拷贝开销。最好的优化是在 CPU 端，即在 `calculator-api.cpp` 中分配 `rpcmem` 内存时，就保证每一行 (row) 的起始地址都强制 128 字节对齐。如果能做到这一点，DSP 端的 `memcpy` 就可以被移除，换回更高效的直接指针加载，从而获得显著的性能提升。

3. 缓存与带宽瓶颈

外积实现存在严重的带宽瓶颈。内积实现虽然已经好很多，但在处理超大矩阵时（如 $512 \times 512 \times 512$ ），依然会受限于 L2 缓存大小和内存带宽。可以考虑进行循环分块：不要一次计算一整行或一整个元素，而是将 `A`、`B`、`C` 矩阵都切成小块计算。这样做可以把所需 `A` 和 `B` 的数据块加载到 L1/L2 缓存中，并高度复用它们来计算 `C` 小块中的所有 16 个元素，极大地降低对主内存的访问次数。