实验过程

```
cd OS-2026
```

```
git pull
```

Hexagon SDK 安装

手动下载

```
wget https://apigwx-aws.qualcomm.com/qsc/public/v1/api/download/software/sdks/Hexagon_SDK/Linux/Debian/6.3.0.0/Hexagon_SDK.zip
```

创建文件夹后解压

```
unzip Hexagon_SDK.zip -d /home/qomolangma/Qualcomm
```

统一的环境变量

```
export HEXAGON_SDK_PATH=/home/qomolangma/Qualcomm/Hexagon_SDK/6.3.0.0
```

使配置生效：

```
source ~/.bashrc
```

Android NDK 安装

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
```

```
(base) qomolangma@LAPTOP-NVMISHDL:~$ mv cmdline-tools/* ~/Android/Sdk/cmdline-too
ls/latest/
(base) qomolangma@LAPTOP-NVMISHDL:~$ export PATH=~/Android/Sdk/cmdline-tools/late
st/bin/:$PATH
(base) qomolangma@LAPTOP-NVMISHDL:~$ source ~/.bashrc
(base) qomolangma@LAPTOP-NVMISHDL:~$
```

2. 安装OpenJDK 17

1. **安装 JDK**:

```
sudo apt update
sudo apt install openjdk-17-jdk
```

2. **找到 JDK 安装路径**:

```
sudo update-alternatives --config java
```

这会列出所有已安装的 Java 版本及其路径。路径通常是 `/usr/lib/jvm/java-17-openjdk-amd64`。

3. **编辑环境变量文件**:

```
sudo nano /etc/environment
```

4. **在文件中添加或修改**:

```
JAVA_HOME="/usr/lib/jvm/java-17-openjdk-amd64"
```

请将路径替换成你自己的实际路径。

5. **保存并退出** (在 `nano` 中是 `Ctrl+X`, 然后按 `Y`, 最后按 `Enter`)。

6. **重新加载环境变量**:

```
source /etc/environment
```

7. **验证**:

```
echo $JAVA_HOME
java -version
```

```
(base) qomolangma@LAPTOP-NVMISHDL:~$ sudo update-alternatives --config java
There is only one alternative in link group java (providing /usr/bin/java): /usr/lib/jvm/java-17-openjdk-amd64/bin/java
Nothing to configure.
(base) qomolangma@LAPTOP-NVMISHDL:~$ sudo nano /etc/environment
(base) qomolangma@LAPTOP-NVMISHDL:~$     source /etc/environment
(base) qomolangma@LAPTOP-NVMISHDL:~$     echo $JAVA_HOME
    java -version

/usr/lib/jvm/java-17-openjdk-amd64
openjdk version "17.0.16" 2025-07-15
OpenJDK Runtime Environment (build 17.0.16+8-Ubuntu-0ubuntu122.04.1)
OpenJDK 64-Bit Server VM (build 17.0.16+8-Ubuntu-0ubuntu122.04.1, mixed mode, sharing)
(base) qomolangma@LAPTOP-NVMISHDL:~$
```

```
sudo apt install sdkmanager
```

1. **为你的用户创建一个专门的 SDK 目录**：

```
mkdir -p ~/Android/Sdk
```

这个命令会在你的主目录 (`~`) 下创建 `Android/Sdk` 文件夹。

2. **设置 `ANDROID_SDK_ROOT` 环境变量**：
你需要告诉系统（以及 Android 开发工具），你的 SDK 安装在哪里。这个变量就是 `ANDROID_SDK_ROOT`。

   - 打开你的 shell 配置文件（取决于你使用的 shell）：

     ```
     nano ~/.bashrc
     ```

   - 在文件的末尾，添加下面两行：

     ```
     export ANDROID_HOME=$HOME/Android/Sdk
     export ANDROID_SDK_ROOT=$HOME/Android/Sdk
     ```

     > **注意**: `ANDROID_HOME` 是一个旧的变量名，但很多工具仍在使用。`ANDROID_SDK_ROOT` 是新的标准。建议两个都设置，指向同一个位置，以确保最大兼容性。

   - 保存文件并退出 (在 `nano` 中是 `Ctrl+X`, 然后按 `Y`, 最后按 `Enter`)。

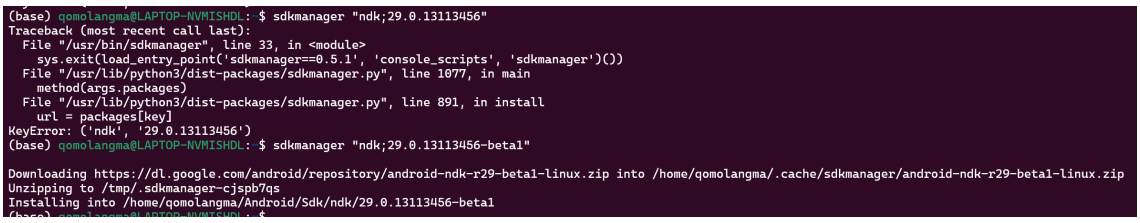
3. **让环境变量生效**：
关闭并重新打开一个终端，或者运行下面的命令：

```
source ~/.bashrc
```

3. 安装 NDK

1. **查看可用版本并安装**

```
sdkmanager --list
```




```
sdkmanager "ndk;29.0.13113456-beta1"
```

2. **配置 NDK 环境变量**

在 `~/.bashrc` 中添加：

```
export ANDROID_NDK_ROOT=~/Android/Sdk/ndk/29.0.13113456-beta1/
```

3. **安装 ADB 工具**

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

代码编译与运行

编译 NPU 代码

执行下列命令前，确保自己处于 `/OS-2026/Lab4` 目录

```
cd OS-2026/Lab4
```

首先设置 Hexagon SDK 环境并编译 DSP 代码：

```
source $HEXAGON_SDK_PATH/setup_sdk_env.source
cd dsp
```

```
sudo apt install libtinfo5
```

```
make hexagon BUILD=Debug DSP_ARCH=v79
```

```
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/dsp$ make hexagon BUILD=Debug DSP_ARCH=v79
==== Building Debug variant of hexagon architecture v79 ====
==== Using Hexagon Tools at /home/qomolangma/Qualcomm/Hexagon_SDK/6.3.0.0/tools/HEXAGON_Tools/8.8.06 ====
==== Build output directory: /home/qomolangma/OS-2026/Lab4/dsp/hexagon_Debug_toolv88_v79/ship ====
python /home/qomolangma/Qualcomm/Hexagon_SDK/6.3.0.0/utils/telematics/hexagonsdk_telematics.py -e "2" "make.d,hexagon,dsp"
making .
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/dsp$
```

模拟器运行

> ⚠️ **注意**：模拟器运行性能较差，且不方便进行功能验证。

1. 修改必要的依赖库

```
sudo ln -s /usr/lib/x86_64-linux-gnu/libncurses.so.6 \
           /usr/lib/x86_64-linux-gnu/libncurses.so.5
```

2. 运行 Hexagon 模拟器

执行下列命令前，确保自己处于 `/OS-2026/Lab4/dsp` 目录且 NPU 代码编译成功

```
export DSP_BUILD_DIR=hexagon_Debug_toolv88_v79

${HEXAGON_SDK_ROOT}/tools/HEXAGON_Tools/8.8.06/Tools/bin/hexagon-sim \
    -mv79 \
    --simulated_returnval \
    --usefs ${DSP_BUILD_DIR} \
    --pmu_statsfile ${DSP_BUILD_DIR}/pmu_stats.txt \
    --cosim_file ${DSP_BUILD_DIR}/q6ss.cfg \
    --l2tcm_base 0xd800 \
    --rtos ${DSP_BUILD_DIR}/osam.cfg \
    ${HEXAGON_SDK_ROOT}/rtos/qurt/computev79/sdksim_bin/runelf.pbn \
    -- \

 ${HEXAGON_SDK_ROOT}/libs/run_main_on_hexagon/ship/hexagon_toolv88_v79/run_main_o
n_hexagon_sim \
    stack_size=0x400000 \
    -- \
```

```
    libtest_calculator_sim.so 64 64 64
```

```
====================================
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)
====================================



unloading ./libtest_calculator_sim.so:  HIGH:0x1E7:325:rtld.c
Main() returned 0:  HIGH:0x1EA:271:run_main_on_hexagon_sim.c

Done!
        T0: Insns=30893709 Packets=28936657
        T1: Insns=241461 Packets=110368
        T2: Insns=14371 Packets=6195
        T3: Insns=198 Packets=103
        T4: Insns=198 Packets=103
        T5: Insns=198 Packets=103
        T6: Insns=0 Packets=0
        T7: Insns=0 Packets=0
        Total: Insns=31150135 Pcycles=116176306
```

设备运行 (推荐)

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


```
chmod 400 qdc_id_2025-10-20_62.pem
```

3. 创建私钥并保存到 `~/qdc.pem`，修改权限：

```
chmod 600 qdc_id_2025-10-20_62.pem
```

2. 建立设备连接

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_id_2025-10-20_62.pem -L 5037:sa341225.sa.svc.cluster.local:5037 -N sshtunnel@ssh.qdc.qualcomm.com

ssh -i qdc_id_2025-10-29_557.pem -L 5037:sa341296.sa.svc.cluster.local:5037 -N sshtunnel@ssh.qdc.qualcomm.com
```

2. **验证设备连接**

```
adb devices
```

   应该能看到已连接的设备。

```
(base) qomolangma@LAPTOP-NVMISHDL:~$ adb devices
List of devices attached
b84a556d        device

(base) qomolangma@LAPTOP-NVMISHDL:~$
```

3. 编译 Android 测试工具

执行下列命令前，确保自己处于 `/OS-2026/Lab4` 目录

```
export HEXAGON_SDK_PATH=/home/qomolangma/Qualcomm/Hexagon_SDK/6.3.0.0
export ANDROID_NDK_ROOT=~/Android/Sdk/ndk/29.0.13113456-beta1/
```

```
source $HEXAGON_SDK_PATH/setup_sdk_env.source
mkdir build
cd build
```

```
sudo apt  install cmake
```

```
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
```

```
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ export ANDROID_NDK_ROOT=~/Android/Sdk/ndk/29.0.13113456-beta1/
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/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 ..
-- The C compiler identification is Clang 20.0.0
-- The CXX compiler identification is Clang 20.0.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /home/qomolangma/Android/Sdk/ndk/29.0.13113456-beta1/toolchains/llvm/prebuilt/linux-x86_64/bin/clang - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /home/qomolangma/Android/Sdk/ndk/29.0.13113456-beta1/toolchains/llvm/prebuilt/linux-x86_64/bin/clang++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/qomolangma/OS-2026/Lab4/build
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ make
[ 25%] Building CXX object CMakeFiles/npu_gemm_test.dir/npu_gemm_cli.cpp.o
[ 50%] Building CXX object CMakeFiles/npu_gemm_test.dir/calculator-api.cpp.o
[ 75%] Building C object CMakeFiles/npu_gemm_test.dir/dsp/hexagon_Debug_toolv88_v79/calculator_stub.c.o
[100%] Linking CXX executable npu_gemm_test
[100%] Built target npu_gemm_test
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$
```

4. 部署并运行测试

执行下列命令前，确保自己处于 `/OS-2026/Lab4/build` 目录

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
```

```
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ adb push npu_gemm_test /data/local/tmp/
npu_gemm_test: 1 file pushed. 0.5 MB/s (2656128 bytes in 4.983s)
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ adb push ../dsp/hexagon_Debug_toolv88_v79/ship/libcalculator_skel.so
adb: push requires an argument
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ /data/local/tmp/
-bash: /data/local/tmp/: No such file or directory
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ adb push npu_gemm_test /data/local/tmp/
adb push ../dsp/hexagon_Debug_toolv88_v79/ship/libcalculator_skel.so /data/local/tmp/
npu_gemm_test: 1 file pushed. 0.8 MB/s (2656128 bytes in 3.356s)
../dsp/hexagon_Debug_toolv88_v79/ship/libcalculator_skel.so: 1 file pushed. 0.1 MB/s (31568 bytes in 0.293s)
(base) qomolangma@LAPTOP-NVMISHDL:~/OS-2026/Lab4/build$ adb shell
sun:/ # cd /data/local/tmp
sun:/data/local/tmp # chmod +x npu_gemm_test
sun:/data/local/tmp # ./npu_gemm_test 64 64 64 --cpu-check

==============================
Starting NPU GEMM test with dimensions: C(m,n) = A(m,k) * B^T(n,k)
M=64, K=64, N=64
Initializing input matrices...
Calling NPU for GEMM calculation (A * B^T)...

[SUCCESS] NPU GEMM (A * B^T) calculation finished successfully!
NPU Result matrix C (first 10 elements):
0.13 0.14 0.21 0.12 0.15 0.19 0.12 0.18 0.16 0.12

Total time spent on NPU call: 56.217 ms

Check if the result is correct...
CPU Result matrix C (first 10 elements):
0.13 0.14 0.21 0.12 0.15 0.19 0.12 0.18 0.16 0.12
√ NPU and CPU results match within tolerance (by percent error)
sun:/data/local/tmp #
```

10.20

向量化与对齐

HVX 向量宽度

HVX 向量长度是 128 字节，也就是一次可以并行操作 32 个 `float`，代码里显式写了：

```
#define HVX_BYTES 128
#define HVX_F32_PER_VEC 32
```

所有 HVX 块的循环都是以 32 为步长处理数据。

对齐与加载策略

在HVX加载时使用了 `vloadu_f32()`：

```
static inline HVX_Vector vloadu_f32(const float *p) {
    HVX_Vector v;
    memcpy(&v, p, HVX_BYTES);
    return v;
}
```

“统一用 memcpy 搬 128B 进 HVX_Vector”策略，可以处理“只要有 32 个 float 连续在内存中”就能读，不要求调用点一定是 128B 对齐的指针，从而避免未对齐 load 指令的限制。同理，`vstoreu_f32()` 用 memcpy 把计算结果写回内存。

在测试分配矩阵时也使用了对齐分配：

```
float* A = (float*)memalign(128, ...);
```

这样矩阵起始地址本身是128B对齐的，大部分主循环访问天然是 aligned，进一步提升性能。

尾部处理

HVX 向量一次只能很干脆地处理 32 个元素，本实验选择显式处理的“尾巴”：

- 在外积实现 `gemm_hvx_outer()` 里：
  - 主循环只跑到 `n_vec = (n/32)*32`，即完整的 32 倍数的列块；
  - 剩下的 `j = n_vec .. n-1` 用标量回退。
- 在内积实现 `gemm_hvx_dot()` 里：
  - 主循环只跑到 `k_vec = (k/32)*32`，即完整的 32 倍数的 K 段；
  - 剩下的 `l = k_vec .. k-1` 用标量回退。

这满足实验任务要求的第二项：“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”。

性能测量

三次实验

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	设备	64×64×64	**46.097**	
2	HVX 内积 (A * B^T)	设备	64×64×64	**44.166**	
3	HVX 外积 (A * B)	设备	64×64×64	**38.930**	
4	朴素 baseline	设备	256×256×256	**612.028**	
5	HVX 内积 (A * B^T)	设备	256×256×256	**189.787**	
6	HVX 外积 (A * B)	设备	256×256×256	**131.425**	
7	朴素 baseline	设备	512×512×512	**6266.126**	
8	HVX 内积 (A * B^T)	设备	512×512×512	**996.740**	
9	HVX 外积 (A * B)	设备	512×512×512	**722.799**	
10	朴素 baseline	设备	88×99×66	**62.872**	
11	HVX 内积 (A * B^T)	设备	88×99×66	**44.846**	
12	HVX 外积 (A * B)	设备	88×99×66	**41.422**	

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	设备	64×64×64	**52.070**	
2	HVX 内积 (A * B^T)	设备	64×64×64	**41.389**	
3	HVX 外积 (A * B)	设备	64×64×64	**32.150**	
4	朴素 baseline	设备	256×256×256	**608.593**	
5	HVX 内积 (A * B^T)	设备	256×256×256	**197.154**	
6	HVX 外积 (A * B)	设备	256×256×256	**123.350**	
7	朴素 baseline	设备	512×512×512	**6250.330**	
8	HVX 内积 (A * B^T)	设备	512×512×512	**985.887**	
9	HVX 外积 (A * B)	设备	512×512×512	**719.168**	
10	朴素 baseline	设备	88×99×66	**62.344**	
11	HVX 内积 (A * B^T)	设备	88×99×66	**48.354**	
12	HVX 外积 (A * B)	设备	88×99×66	**38.185**	

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	设备	64×64×64	**44.012**	
2	HVX 内积 (A * B^T)	设备	64×64×64	**43.631**	
3	HVX 外积 (A * B)	设备	64×64×64	**35.083**	
4	朴素 baseline	设备	256×256×256	**606.213**	
5	HVX 内积 (A * B^T)	设备	256×256×256	**196.767**	
6	HVX 外积 (A * B)	设备	256×256×256	**135.833**	
7	朴素 baseline	设备	512×512×512	**6255.987**	
8	HVX 内积 (A * B^T)	设备	512×512×512	**1010.313**	
9	HVX 外积 (A * B)	设备	512×512×512	**715.489**	
10	朴素 baseline	设备	88×99×66	**58.825**	

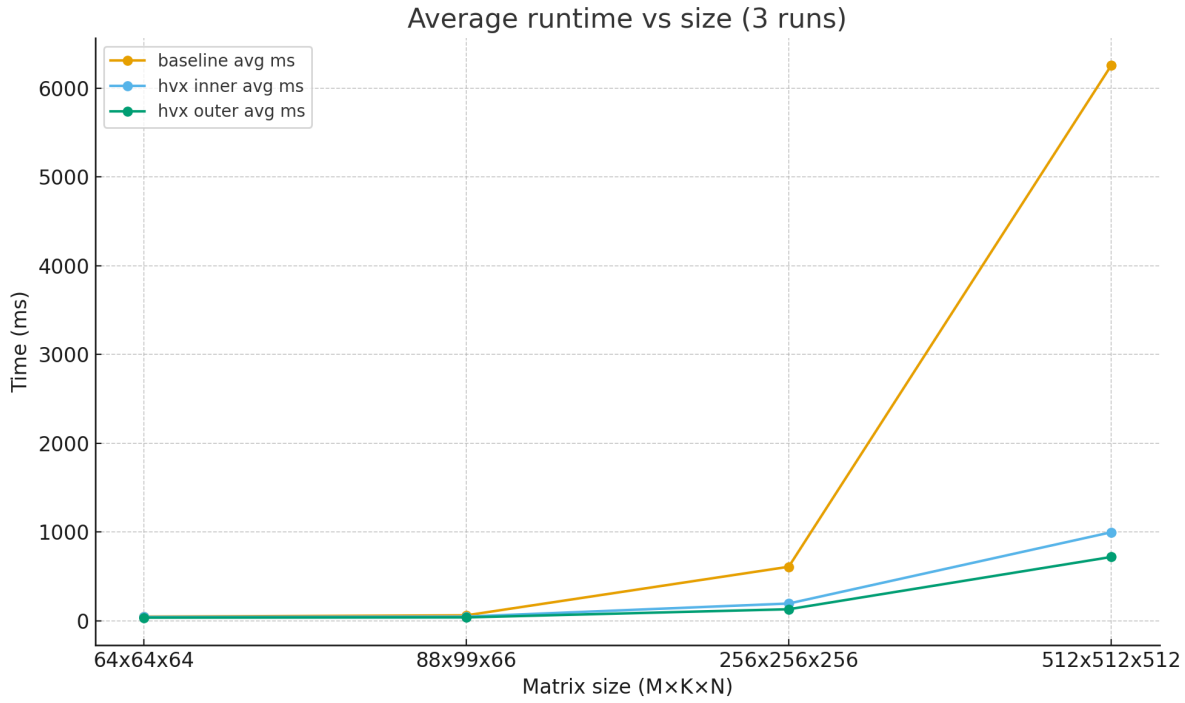
实验编号	实现方式	设备/模拟器	矩阵尺寸 (M×K×N)	计算耗时 (ms)	备注
11	HVX 内积 (A * B^T)	设备	88×99×66	**48.375**	
12	HVX 外积 (A * B)	设备	88×99×66	**34.922**	

结果分析

计算效率

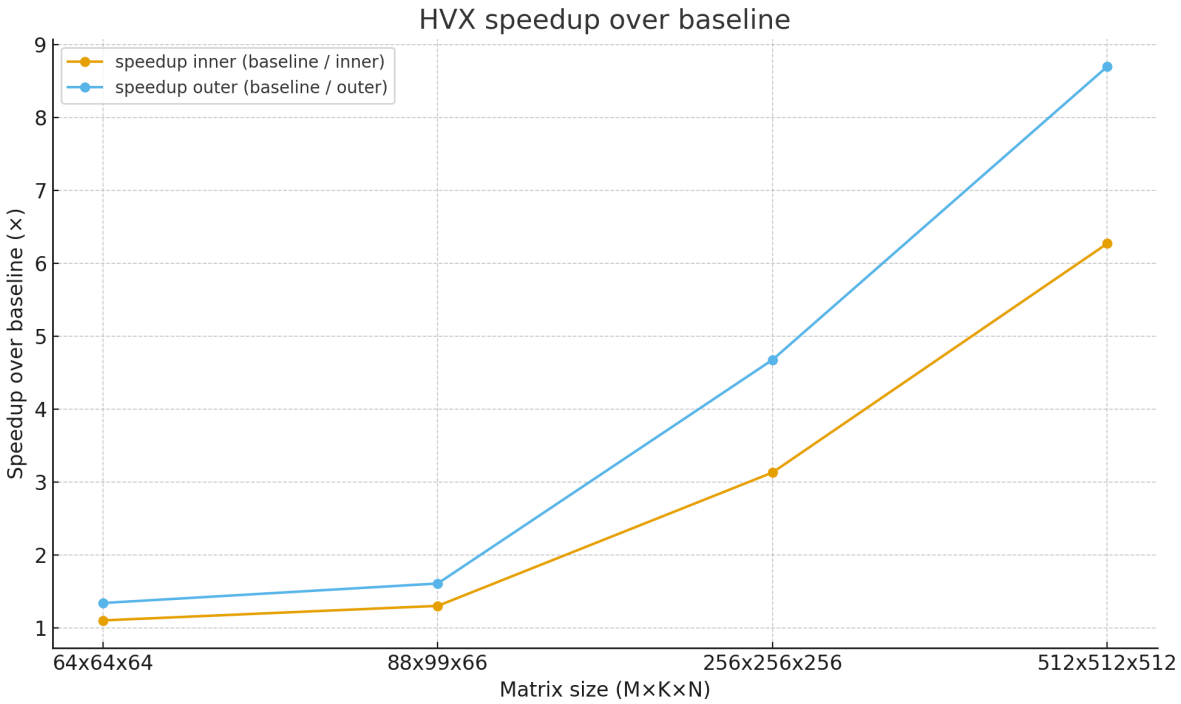
最终平均结果：

矩阵尺寸	baseline 平均耗时	HVX 内积 平均耗时	HVX 外积 平均耗时
64×64×64	47.393 ms	43.062 ms	35.388 ms
88×99×66	61.347 ms	47.192 ms	38.176 ms
256×256×256	608.945 ms	194.569 ms	130.203 ms
512×512×512	6257.481 ms	997.647 ms	719.152 ms



我也算了 HVX 相对于 baseline 的平均加速倍数：

矩阵尺寸	HVX 内积 加速比	HVX 外积 加速比
64×64×64	1.10×	1.34×
88×99×66	1.30×	1.61×
256×256×256	3.13×	4.68×
512×512×512	6.27×	8.70×



观察到，规模越大，加速越夸张。512^3上 HVX 外积平均加速8.7倍，已经是数量级的提升。

HVX 指令及作用

- `Q6_V_vsplat_R()`

  将单个标量浮点值扩展（splat）成一个完整的 HVX 向量寄存器，即 128 字节的大向量，其中 32 个 float 全部是同一个值。本实验在 `gemm_hvx_outer()`（外积法）里使用它，把 `A[i,l]` 这个标量广播成向量 `a_brd`。这样后续的计算就可以拿这一整条广播向量同时去乘 B 的 32 个相邻元素块 `[B[l, j], ..., B[l, j+31]]`，从而一次性更新 C 的 32 个输出位置 `[C[i, j], ..., C[i, j+31]]`。这一点非常关键，因为它把"标量×向量"的外积更新，变成了"向量×向量"的并行乘法和累加，这是外积法高效的基础。

- `Q6_Vqf32_vmpy_VsfVsf()`

  对两个向量做逐元素乘法，并把结果保存在 qf32 精度的向量寄存器里。在外积法中用它计算 `a_brd * b_vec`，相当于在一个时钟片段里同时完成 32 次浮点乘法；在内积法 `gemm_hvx_dot()` 里也用它对 `va` 和 `vb` 做逐元素相乘，其中 `va` 是 A 的一段长度为 32 的切片，`vb` 是 B^T 的对应切片。这一步可以被认为是"向量化的部分点积贡献"，它直接把 baseline 中最内层的标量乘法循环展开成 32 路 SIMD。

- `Q6_Vqf32_vadd_Vqf32Vqf32()`

  执行向量加法，并将两个 qf32 向量的结果逐元素相加。在外积法中，这个加法扮演"累加器"的角色：把每次 `a_brd * b_vec` 的乘积结果累加到同一个 `acc_q` 寄存器中，这意味着在不断更新 C[i, j:j+31] 这一整段的部分和，但这一切都还停留在寄存器而不是内存里，只有在循环末尾才回写内存。在内积法中，这条指令被用来把一小段（32 元素）点积的乘积结果，逐块累加到一个临时的向量累加器 `acc_q`，为后续的水平归约做准备。可以说它让"多轮乘法结果"在向量寄存器里不断累加，而不需要频繁访存。

- `Q6_V_vror_VR()`

  把一个向量按字节数进行循环右移（rotate right）。在 `hvx_reduce_add_qf32()` 中反复使用它，把向量后半部分旋转到前半部分，然后用 `Q6_Vqf32_vadd_Vqf32Vqf32()` 把两半加在一起。第一次旋转 16 个 float 的距离，可以理解为把 32-lane 向量的后 16 个元素搬到前 16 个元素的位置并相加，得到一个"每两元素折半"的中间结果。之后我们再旋转 8 个 float、4 个 float、2 个

float、1 个 float，逐次叠加。经过这棵“二叉树式”折叠过程，原本分布在 32 个 lane 里的 32 个部分和，最终被压缩到同一个 lane 上。`Q6_V_vror_VR()` 让我们可以在向量寄存器内部完成归约求和，而不需要把整条向量拆成标量循环；它是把 SIMD 并行结果“收束成一个标量”的桥梁。

HVX 指令的作用不仅仅是“把标量循环并行化成 32 路 SIMD”。更重要的是，它们让我们能够以“广播+向量乘加+并行累加+本地归约”的方式重写整个 GEMM 的计算结构，从而在大矩阵规模下大幅降低总线往返和标量循环的开销。这也解释了实验里非常显著的加速比：对于 512×512×512 的矩阵乘法，baseline 的执行时间在 6 秒以上，而 HVX 外积法将其压缩到了约 0.7 秒量级，接近 **9 倍**的提速。

瓶颈、优化建议

在 64×64×64 中，HVX 版本只比 baseline 快 1.1 ~ 1.34倍，这是因为DSP RPC 调用 + 初始化开销在小规模下占比较大，同时向量化准备（广播、acc_q初始化、归约）本身也有固定开销，小矩阵没有足够的工作量去摊销这些固定成本。HVX 优势随矩阵规模增大而放大，这是正常现象。

对于 n 或 k 不是 32 的倍数，比如 88×99×66 这种模型，不满 32 的尾部列/尾部通道会走标量回退。从数据看，即使在这种“不整齐”的情况下，HVX 外积依旧能比 baseline 快 1.6× 左右，说明尾部处理是正确而且高效的。

当前实现还没有显式使用 `l2fetch`（Hexagon SDK 里提供的 L2 预取机制）或 VTCM（本地 scratchpad）优化。 未来可以尝试在处理下一块 B 和 C 之前，提前发起 `l2fetch` 把那一块调进更快的本地存储；把 A 的一行和 B 的一部分 tile 到本地 VTCM，减少重复从外存加载的次数；进一步分块 (tiling) 避免大矩阵直接访问 DDR/L2 带来的带宽瓶颈。外积法已经把 `C[i,j:j+31]` 整段在本地寄存器中累加到 `acc_q`，再统一回写。再往前走一步，我们可以把 C 的 tile 暂存在 VTCM 中，做多次 `l` 的累加后再回写，这能继续减少总线写流量。