移动端AI模型部署相关

FastRPC

资料:

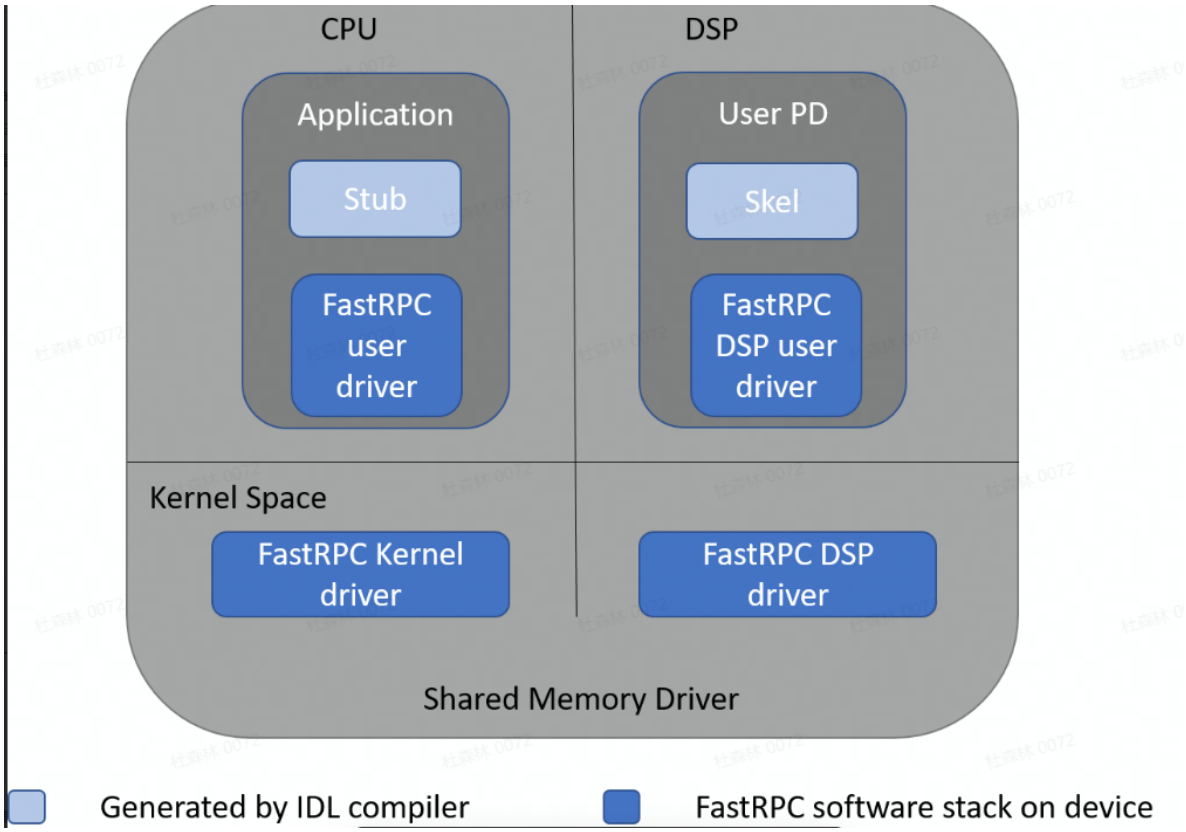
Hexagon_SDK/4.5.0.3/docs/software/ipc/rpc.html

FastRPC: FastRPC 是用于启用 CPU 和 DSP 之间的远程函数调用的 RPC 机制。

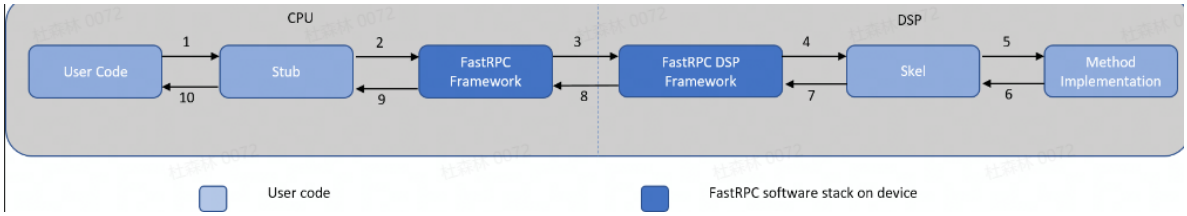
作用：FastRPC 框架将大型处理任务卸载到DSP上，DSP 可以利用其内部处理资源（例如 HVX）以比 CPU 更具有计算和功率效率的方式执行任务。

FastRPC接口：FastRPC 接口在 **IDL 文件中**定义，并使用 QAIC 编译器进行编译以生成头文件以及存根和 skel 代码。头文件和存根应该被构建并链接到 CPU 可执行文件中，而头文件和 skel 应该被构建并链接到 DSP 库中。

1、FastRPC架构：



2、FastRPC工作流程：



3、Android 软件组件：

有些库与android 版本有关。

4、DSP保护域：

dsp中的各种保护域，FastRPC 客户端程序在用户 PD （protection domains）中运行。

> 动态与静态PD： 用户 PD 可以在启动时静态创建，但它们更经常由 CPU 应用程序在运行时动态创建，这些应用程序将模块卸载到 DSP。静态和动态 PD 都支持共享对象的动态加载。
>
> --静态PD:静态 PD 在 DSP 上创建，以支持音频和传感器等特定用例。静态 PD 允许在 CPU 上运行的守护进程的帮助下动态加载共享对象。
>
> --动态用户PD:每个通过 FastRPC 使用 DSP 的 CPU 用户进程在 DSP 上都会有一个对应的动态用户 PD； 每个 DSP 只能支持有限数量的并发动态用户 PD
>
> --签名和未签名的PD：签名的 PD 在所有 DSP 上都可用，未签名的仅在cDSP中受支持。
>
> --未签名的PD支持：要检查设备是否支持未签名的 PD，请使用 DSP 属性 UNSIGNED_PD_SUPPORT 执行能力查询。
>
> --未签名的PD可用服务：线程创建和线程服务；HVX 上下文；时钟频率控制；VTCM；缓存操作；映射相应HLOS应用程序分配的HLOS内存
>
> --未签名的PD限制：
> --请求免签名卸载：将动态共享对象卸载到 cDSP 而无需签名

5、IDL 编译器：

DSP 平台和所有 FastRPC 程序的接口都用一种称为 IDL 的语言进行描述。DL 提供软件实施的灵活性，同时为软件模块保持一致的接口。

典型的IDL头文件：

```
#include "AEEStdDef.idl"
#include "remote.idl"

interface calculator : remote_handle64 {
   long sum(in sequence<long> vec, rout long long res);
   long max(in sequence<long> vec, rout long res);
};
```

包含实例：.../4.5.0.3/docs/examples/calculator/index.html

6、多域：

域是用于加载和执行代码的远程环境 (DSP)。Snapdragon 产品包括多个 Hexagon DSP（例如，cDSP、aDSP、mDSP 或 sDSP）。在许多目标上，这些域中的一个以上可供 FastRPC CPU 用户进程使用。多域的优势：... ..., 包含实例。

7、FastRPC线程与进程：

客户端可以使用以下 API 和数据结构（暴露在 remote.h 头文件中）来配置他们的 DSP 线程参数：

```
struct remote_rpc_thread_params {
            int domain;
            int prio;
            int stack_size;
};
```

处理异常：

子系统重启 (SSR)

PD restart: example for `calculator_test.c`

限制：

Example:

在 cDSP 上将优先级设置为 100，堆栈大小设置为 4 MB：

```
#include "remote.h"
...
main() {
        struct remote_rpc_thread_params th_data;
        th_data.domain = CDSP_DOMAIN_ID;
        th_data.stack_size = 4*1024*1024;
        th_data.prio = 100;
        nErr = remote_session_control(FASTRPC_THREAD_PARAMS, (void*)&th_data,
sizeof(th_data));
        if (nErr) {
                printf("ERROR 0x%x: remote_session_control failed to set thread
params\n", nErr);
        } else {
                printf(" - Successfully set CDSP user process thread
parameter(s)\n");
        }
        ....
     remote_rpc_notif_register status_notif;   my_first_RPC_call();
}
```

DSP用户进程的状态通知：

客户端可以使用此 API 和以下数据结构（暴露在 remote.h 头中）来注册 DSP 用户进程的状态通知。

```
typedef struct remote_rpc_notif_register {
            void *context;
            int domain;
            fastrpc_notif_fn_t notifier_fn;
} remote_rpc_notif_register_t;
```

Example: ... ...

8、异步FastRPC:

FastRPC 是一个远程过程调用框架，供 CPU 客户端有效地将任务卸载到 DSP。 FastRPC 默认是同步的,
异步支持实例：
$HEXAGON_SDK_ROOT/addons/compute/docs/docs/examples/benchmark/README.md,

IDL异步支持：IDL 文档 ../4.5.0.3/docs/reference/idl.html#async-fastrpc-support

异步描述符：FastRPC 异步描述符由通知类型、作业 ID 和回调组成；

通知声明：

1、回调通知：用户需要将回调函数放在描述符中

```
struct fastrpc_async_descriptor desc;
desc.type = FASTRPC_ASYNC_CALLBACK;
desc.cb.fn= fn;
desc.cb.context = ptr;
```

2、作业完成的用户投票：用户需要使用 FASTRPC_ASYNC_POLL 作为 notify_type

```
struct fastrpc_async_descriptor desc;
desc.type = FASTRPC_ASYNC_POLL;
```

3、不需要通知：用户需要为 notify_type 使用 FASTRPC_ASYNC_NO_SYNC

```
struct fastrpc_async_descriptor desc;
desc.type = FASTRPC_ASYNC_NO_SYNC;
```

释放异步 JobID：

客户端在收到用户轮询或回调的呼叫完成通知后应释放jobid。

9、远程文件系统：

DSP没有自己的文件系统，所以它使用CPU远程文件系统来读取共享对象文件。

远程文件系统上的搜索路径：

使用DSP_LIBRARY_PATH：用于指定远程文件系统上的搜索路径的定界变量。使用以下说明设置，使用示例，不同设置的不同搜索顺序共享对象。

```
adb shell
// To set up a single directory for DSP_LIBRARY_PATH
export DSP_LIBRARY_PATH = “foo”
// To set up multiple directories for DSP_LIBRARY_PATH
export DSP_LIBRARY_PATH = “foo; bar”
//To set up multiple directories for DSP_LIBRARY_PATH, including the current
process directory:
export DSP_LIBRARY_PATH = “; foo; bar”
```

10、动态加载：

Hexagon SDK 提供了通过动态共享对象在 DSP 上创建和执行自定义代码的工具和服务。

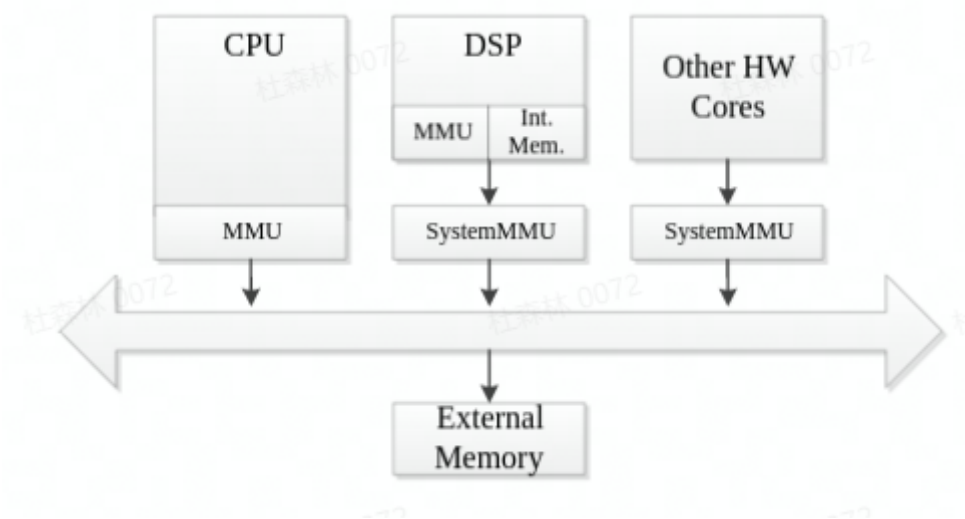
动态共享对象类似于 Linux SO 和 Windows DLL 文件。它们以 ELF 文件的形式实现，在 CPU 文件系统中，它们以文件的形式存在，由 DSP 通过处理器间通信机制加载。加载后，共享对象公开导出的所有符号都可以被引用或调用。

支持动态加载如下：在FastRPC调用中； 外部FastRPC调用：dlopen(), 音频和传感器静态代码。

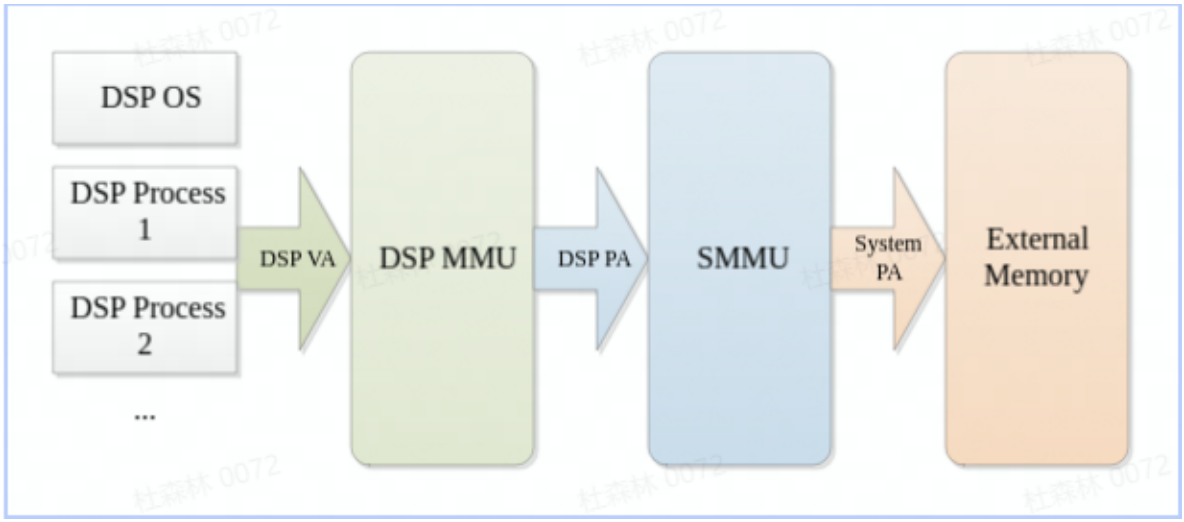
11、内存管理：

讨论基于 FastRPC 的应用程序的 Hexagon DSP 内存管理模型：可用的内存类型、内存管理硬件以及您必须了解的内存性能影响。

硬件概述：



MMU和地址空间：



DSP 上的每个进程都在其自己的虚拟地址空间 (DSP VA) 中运行；

如果进程试图访问未映射到 DSP MMU 的内存，则 DSP 端进程会因页面错误而被终止；

从 Hexagon DSP 到系统的内存访问位于 DSP 物理地址空间 (DSP PA)；

DSP PA 被用作 SMMU 的输入地址，SMMU 将它们转换为系统物理地址。SMMU 用于两个目的：控制 DSP 可以访问哪些内存，并为 DSP 提供物理上不连续缓冲区的连续视图；SMMU 由主应用程序 CPU 管理。如果 DSP 尝试访问未映射到 SMMU 的内存，则访问会导致 CPU 处理 SMMU 页面错误，这通常会导致系统崩溃；

最后，从 SMMU 到系统总线和外部存储器的内存访问位于系统物理地址空间（System PA）中。该地址空间在整个 SoC 中是全局的。

FastRPC 内存管理:

每个 FastRPC 客户端进程在单独的动态用户 PD 中运行其 DSP 代码。每个这样的 PD 都是一个独立的进程，具有自己的虚拟地址空间，与其他 DSP 进程及其 CPU 端对应物分开。用户 PD 从一些可用的本地内存开始，它可以通过两种方式访问更多内存：

(1) 堆分配：DSP 上的标准 C/C++ 堆分配操作（例如 malloc() 和 operator new）使用自定义堆管理器，根据需要向 CPU 请求更多内存。此内存自动映射到进程的地址空间，但 CPU 端客户端无法访问。

(2) 共享缓冲区：客户端应用程序可以在 RPC 调用期间临时分配共享内存缓冲区并将它们映射到 DSP，或者通过显式的 map/unmap 调用持久地映射它们。使用共享缓冲区可以让 CPU 端客户端应用程序及其 DSP 对应方有效地共享数据，而无需在处理器之间进行复制。有关详细信息，请参阅分配内存部分。

FastRPC 框架根据需要将内存映射到 DSP MMU 和 SMMU。默认情况下，FastRPC 将函数调用中传递的所有参数从 CPU 可访问内存复制到 DSP 可访问内存作为调用的一部分。对于少量数据，这不是问题，但对于大型输入/输出缓冲区，例如相机图片或视频帧，复制可能需要大量时间。**为避免复制，FastRPC 客户端应为所有大型输入/输出数据缓冲区使用共享 ION 缓冲区。**

为共享缓冲区分配内存：在 Android 和其他受支持的 Linux 平台上，必须使用 ION 分配器分配共享内存缓冲区。最简单的方法是使用 RPCMEM 库；它会自动使用正确的 ION API 并为 FastRPC 注册缓冲区。如果无法使用 RPCMEM，例如，当缓冲区由不同的框架分配时，客户端可以使用定义为远程 API 的一部分的 remote_register_buf() 函数：

```
remote_register_buf(buffer, size, fd); // register
remote_register_buf(buffer, size, -1); // unregister
```

Transient shared buffers(瞬态共享缓冲区):单元

使用dmahandle 对象的持久共享缓冲区：在 FastRPC 调用期间自动映射的共享缓冲区仅在调用期间进行映射。需要跨函数调用持久映射缓冲区的客户端可以将缓冲区作为 dmahandle 对象传递给 DSP，并手动管理它们的映射生命周期。这些基于 dmahandle 的持久共享缓冲区必须从 ION 分配，类似于常规瞬态共享缓冲区，但它们通过单独的函数调用传递给 DSP，

```
interface example {
    AEEResult map_buffer(in dmahandle buffer);
    AEEResult unmap_buffer();
};
//...
buffer = rpcmem_alloc(RPCMEM_HEAP_ID_SYSTEM, RPCMEM_DEFAULT_FLAGS, size);
fd = rpcmem_to_fd(buffer);
example_map_buffer(fd, 0, size);
// ... Use buffer ...
example_unmap_buffer();
```

在 DSP 端，实现可以使用 HAP_mmap() 将缓冲区映射到进程，并在完成后使用 HAP_munmap() 取消映射缓冲区：

```
AEEResult example_map_buffer(int bufferfd, uint32 bufferoffset,
                             uint32 bufferlen) {
    //...
    buffer = HAP_mmap(NULL, bufferlen, HAP_PROT_READ|HAP_PROT_WRITE, 0,
bufferfd, 0);
}

AEEResult example_unmap_buffer(void) {
    //...
HAP_munmap(buffer, size);
}
```

如果缓冲区用于在 CPU 和 DSP 之间在 map/unmap 操作之间共享数据，应用程序还必须手动执行缓存维护操作：有关如何使用 dmahandle 对象将缓冲区持久映射到 DSP 以及如何在 DSP 上执行缓存维护操作的示例，请参见 **$HEXAGON_SDK_ROOT/examples/asyncdspq_sample/ 中的 fcvqueue 库。**

使用 fastrpc_mmap 的持久共享缓冲区：fastrpc_mmap()

在 CPU 端，应用程序分配 ION 内存并将其映射到 DSP 上的远程进程，如下所示:

```
#include "remote.h"

buffer = rpcmem_alloc(RPCMEM_HEAP_ID_SYSTEM, RPCMEM_DEFAULT_FLAGS, size);
fd = rpcmem_to_fd(buffer);
nErr = fastrpc_mmap(domain, fd, buffer, 0, size, FASTRPC_MAP_FD);
// ... share fd with DSP for accessing the same buffer on DSP ...
nErr = fastrpc_munmap(domain, fd, buffer, size);
```

在 DSP 端，实现可以使用 HAP_mmap_get() 传递缓冲区文件描述符，以增加对缓冲区映射的引用并获取缓冲区的虚拟地址，并在完成后使用 HAP_mmap_put() 释放缓冲区上的引用:

```
//...
nErr = HAP_mmap_get(fd, &buffer, NULL);
//... use buffer and release reference once done ...
nErr = HAP_mmap_put(fd);
```

如果缓冲区用于在 CPU 和 DSP 之间在 map/unmap 操作之间共享数据，应用程序还必须手动执行缓存维护操作: DSP端：qurt_mem_cache_clean()； CPU端：...

性能影响：TLB压力；**共享缓冲区与复制数据；**（默认情况下，FastRPC 将函数调用中传递的所有参数从 CPU 可访问内存复制到 DSP 可访问内存作为函数调用的一部分。对于大型缓冲区，这可能会花费大量时间。**为避免复制，客户端应用程序应为所有大型输入/输出缓冲区使用共享 ION 缓冲区，如上文为共享缓冲区和瞬态共享缓冲区分配内存中所述。**）

12、FastRPC调试：

13、FastRPC性能：