AscendCL最佳实践

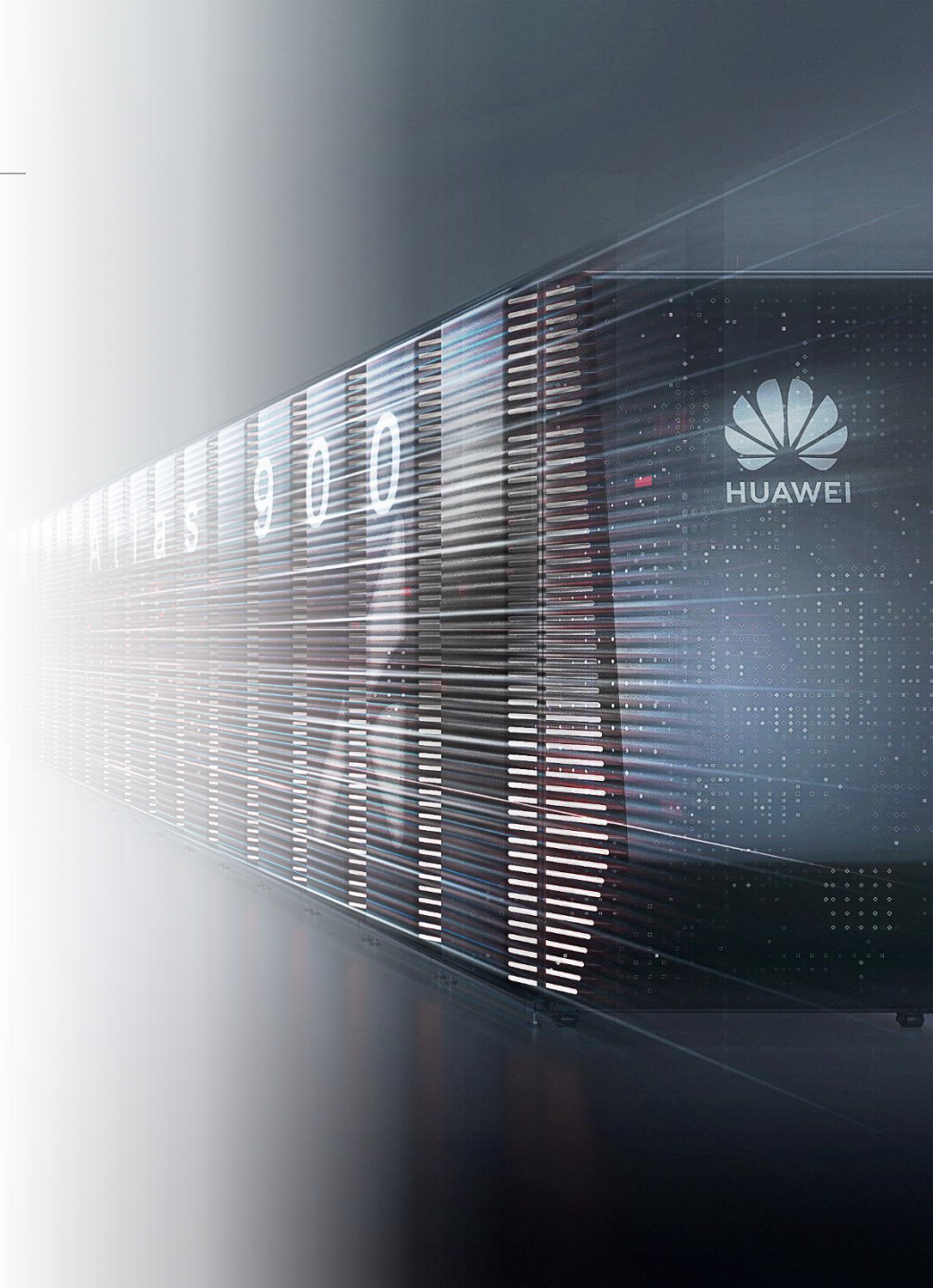
HUAWEI

课程目标

- 学完本课程，您应该能够：
  - >掌握AscendCL应用程序的性能优化方法，在性能不达标时有清晰的分析和优化方向

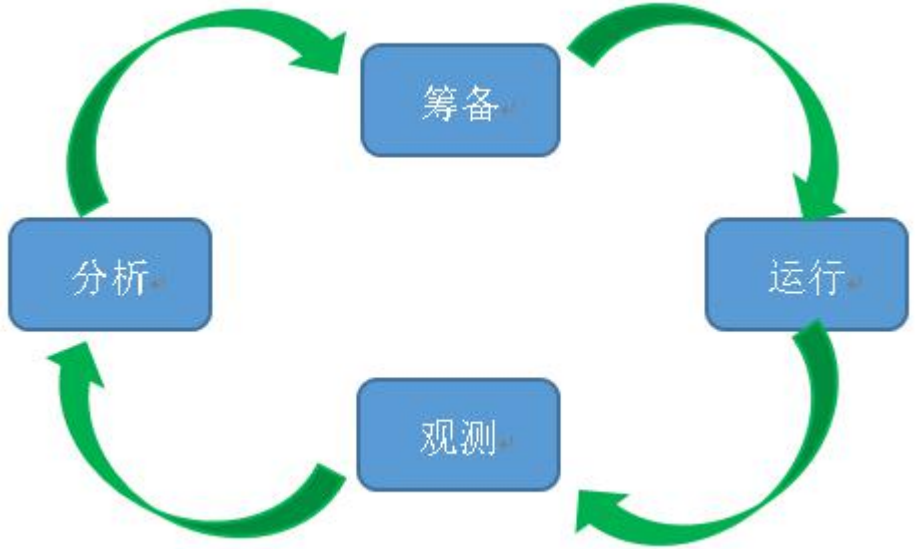
- 为了达成上述目标，您应该具备如下知识：
  - >熟练的C/C++语言编程能力
  - >ATC模型转换工具用法
  - >AscendCL基础功能
  - >基础Linux命令
  - >Linux自带性能分析工具top、free等的用法

目录 TABLE OF CONTENT



1 基本思路：筹备、运行、观测、分析

AscendCL性能调优的一般思路是先组装一条流水线，然后观测性能趋势和基本的占用量，然后再将多种不同业务作为流水线放入昇腾**AI**处理器中，从而达成单个昇腾**AI**处理器的最大化利用；在此过程中可能需要多次调整和部署才能达到设备资源的最大化利用；单个昇腾**AI**处理器利用率达到最大以后，再考虑以阵列的形式将多个昇腾**AI**处理器部署优化起来，套路也是一样，整体思路如下：



筹备：该过程需要对整体流程进行规划和定义，此处回答应该跑几路解码、几个模型、每个模型用多少内存等等。

运行：运行阶段主要是部署任务，将筹备过程中将压力稳定放到昇腾**AI**处理器上，而不能是间歇性的压力。

观测：此时需要采集在稳定条件下系统给出的响应时间，尤其要和筹备阶段时规划的内容进行匹配，主要关注跑起来时系统的响应是否符合预期。

分析：拿到加压后的数据可以方便地评估系统的输出能力、资源用量等等，从而决定下一轮调整的策略。是增加流水线？还是减少内存用量？是增加模型数量？还是减少线程数？整个过程全由上一阶段的观测数据决定。

最终调优的退出条件通常是资源接近用完，或者是几个子系统都接近性能边界。

1 常用指标 – 概念

在整网推理开始之前需要了解推理功能中各个模块基本性能情况，再进一步给出策略。先确保各个模块能够独立运行，

之后再一起加载到昇腾**AI**处理器上，观测各自的性能并且记录如下数据：

1、**Device**上的**top**信息和**free**信息：将整个网络在单条流下加压，观测设备上的内存利用量**\Ctrl CPU\AICPU** 的利用率，以及内存带宽的利用情况。

2、主机上的**top**信息和**free**信息：加压过程中一般情况下主机是压力的发送端，要确保**CPU**的性能指标和内存指标都未达到瓶颈，否则应该先解决上位机的性能瓶颈。

3、主机上的**IO**性能：通过**sar**命令和**iostat**命令观测压力的来源，防止主机的**io**能力到达瓶颈而无法进一步提高性能。在使用主机的硬盘或者通过网络读取测试数据的时候容易遇到这种情况。

4、主机上和设备上的**/proc/meminfo**文件记录了内存更加详细的内容，重点关注**CommitLimit\Committed_AS\MemTotal\MemFree\HugePages_Total\HugePages_Free**，观测是否有用量不足的问题。

1 常用指标 – 案例resnet50的改造1

先将**${toolkit}/acllib/sample/acl_execute_model/acl_resnet50**中的**model_process.cpp**文件稍作处理，使用**for**循环将推理步骤包起来，让推理成为整个应用的主要耗时点以达到稳定运行的目的：

```
306
307 for(uint64_t i=0;i<=0xFFF;i++)
308 {
309     aclError ret = aclmdlExecute(modelId_, input_, output_);
310     if (ret != ACL_ERROR_NONE) {
311         ERROR_LOG("execute model failed, modelId is %u", modelId_);
312         return FAILED;
313     }
314 }
315     INFO_LOG("model execute success");
316     return SUCCESS;
317 }
```

采集**device**上**top**命令和**free –m** 命令的情况，加压以后**top**中**AICPU**为**4~7**、**Ctrl CPU**为**0~3**，推理时这些**CPU**利用量都会有变化，不同网络压力不同。

```
Mem: 2621276K used, 5383300K free, 213948K shrd, 0K buff, 213948K cached
CPU0:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU1:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU2:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU3:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU4:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU5:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU6:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU7:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 6.79 6.80 6.78 2/171 2784
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 1496     1 HwHiAiUs S<    352m  4.4   3  0.0 /var/dmp_daemon -I -U 8087
   51     2 root     IW       0  0.0   0  0.0 [kworker/0:1-eve]
    1     0 root     S     4156  0.0   0  0.0 init
 1305     1 HwHiAiUs S<    160m  2.0   2  0.0 /var/ascend_monitor
```

实测后发现设备端的**CPU**用量很少，内存用量也很少。
但是主机的**CPU**用的却很多，主机的内存用了**16G*5.8%**如下：（命令**top -p 7771 -H**）

```
top - 22:41:53 up  2:46,  6 users,  load average: 0.02, 0.06, 0.05
Threads:   5 total,   0 running,   5 sleeping,   0 stopped,   0 zombie
%Cpu0  :   3.4/2.7     6[||||||                                                  ]
%Cpu1  :   0.7/0.3     1[|                                                       ]
%Cpu2  :   3.9/2.0     6[||||||                                                  ]
%Cpu3  :   2.7/2.7     5[||||||                                                  ]
%Cpu4  :   0.0/0.0     0[                                                        ]
%Cpu5  :   0.0/0.0     0[                                                        ]
%Cpu6  :   0.0/0.0     0[                                                        ]
%Cpu7  :   0.0/0.0     0[                                                        ]
KiB Mem : 16141740 total, 13771196 free,  1471768 used,   898776 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14334304 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 7771 root      20   0 8193.4g 944264   9464 S  7.3  5.8   0:22.28 main
 7774 root      20   0 8193.4g 944264   9464 S  7.0  5.8   0:18.61 RT_SEND
 7775 root      20   0 8193.4g 944264   9464 S  6.3  5.8   0:18.67 RT_RECV
 7772 root      20   0 8193.4g 944264   9464 S  0.0  5.8   0:00.00 LevelNotifyWatc
 7773 root      20   0 8193.4g 944264   9464 S  0.0  5.8   0:00.00 LevelNotifyWatc
```

1 常用指标 –案例resnet50的改造2

主机内存启动前：

```
[root@localhost out]# free -m
              total        used        free      shared  buff/cache   available
Mem:          15763         426       14459           9         877       15009
Swap:          8000           0        8000
[root@localhost out]#
```

主机内存启动后：

```
[root@localhost src]# free -m
              total        used        free      shared  buff/cache   available
Mem:          15763        1435       13449           9         877       13999
Swap:          8000           0        8000
```

因为当前**demo**样例的数据来源只读取一次，所以应用启动初期磁盘有压力，一段时间不读就没有压力了，效果如下：
命令**iostat -dmx 2**主要关注**%util**字段，代表当前磁盘的使用情况。

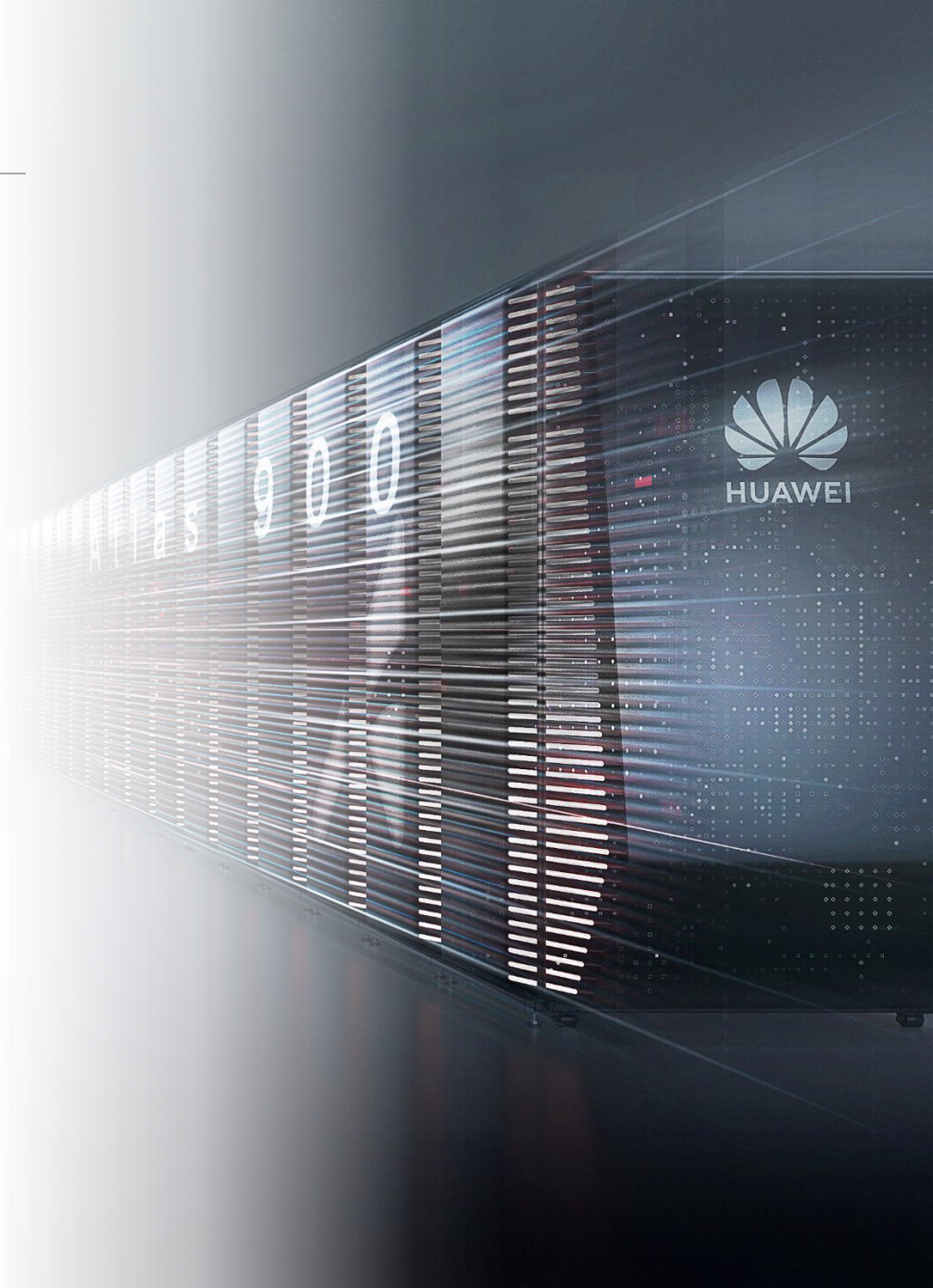
```
Linux 3.10.0-957.el7.x86_64 (localhost.localdomain)     09/09/2020      _x86_64_        (8 CPU)

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.06    1.24    0.43     0.06     0.02    92.99     0.98    1.69    0.19    6.08 588.24  98.20

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.50    0.00    3.50     0.00     0.02    14.14     0.89    2.71    0.00    2.71 252.14  88.25

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.00    0.00    1.50     0.00     0.01     8.00     0.64    5.00    0.00    5.00 424.33  63.65

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00

Device:         rrqm/s   wrqm/s     r/s     w/s    rMB/s    wMB/s avgrq-sz avgqu-sz   await r_await w_await  svctm  %util
nvme0n1           0.00     0.00    0.00    0.00     0.00     0.00     0.00     0.00    0.00    0.00    0.00   0.00   0.00
```

pmap定位占用得到如下截图：

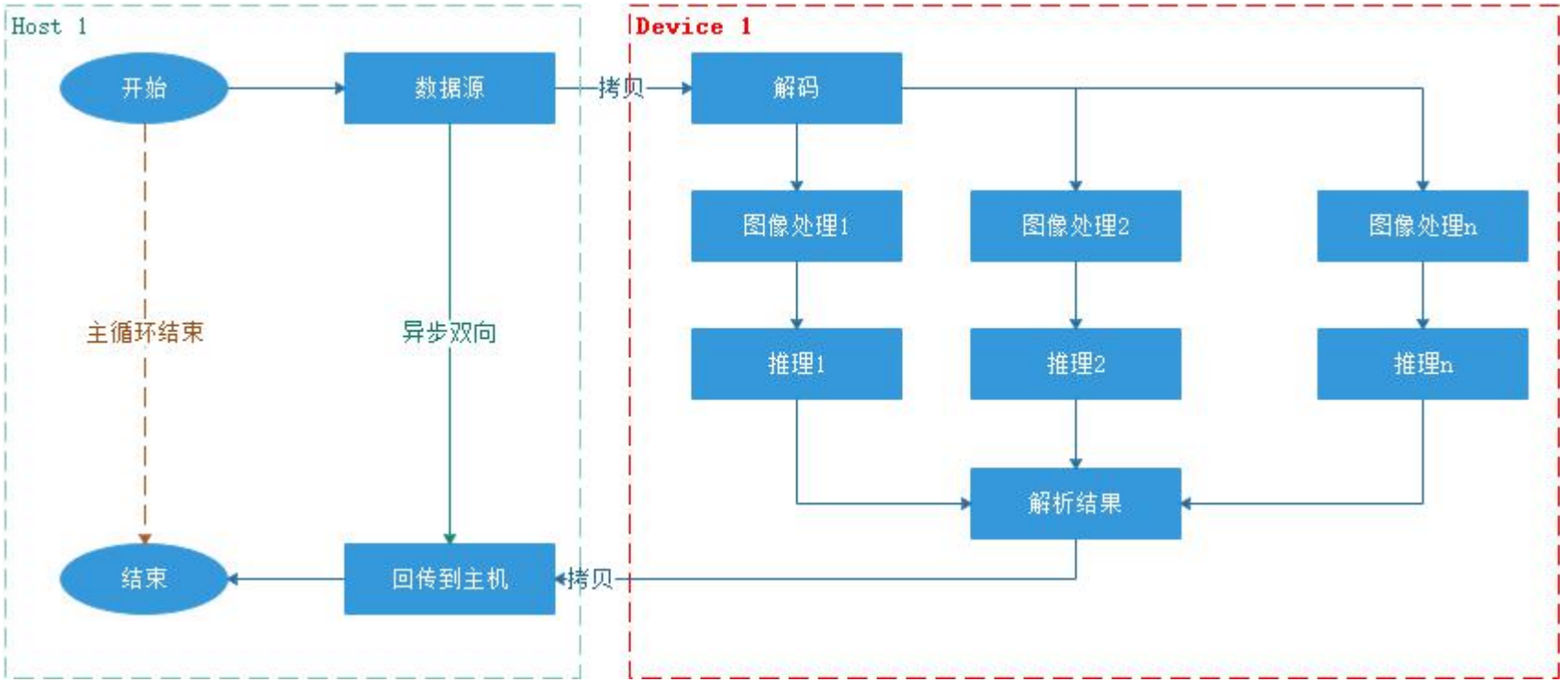
```
[root@localhost tools]# pmap 7170
7170:   ./main
0000000000400000     32K r-x-- main
0000000000607000      4K r---- main
0000000000608000      4K rw--- main
0000000000d55000   4600K rw---   [ anon ]
000000ffc0000000 229440K rw-s- davinci0
0000100000000000 8589934592K rw-s- devmm_svm
00007fd5b0000000    132K rw---   [ anon ]
00007fd5b0021000  65404K -----   [ anon ]
00007fd5b8000000    132K rw---   [ anon ]
00007fd5b8021000  65404K -----   [ anon ]
00007fd5bf270000 262148K rw---   [ anon ]
00007fd5cf271000      4K -----   [ anon ]
00007fd5cf272000   8192K rw---   [ anon ]
00007fd5cfa72000      4K -----   [ anon ]
00007fd5cfa73000   8192K rw---   [ anon ]
00007fd5d0273000      4K -----   [ anon ]
00007fd5d0274000   8192K rw---   [ anon ]
00007fd5d0a74000      4K -----   [ anon ]
00007fd5d0a75000   8192K rw---   [ anon ]
00007fd5d7243000    248K rw---   [ anon ]
00007fd5dd498000  34208K rw---   [ anon ]
00007fd5df600000      4K -----   [ anon ]
00007fd5df601000    128K rw---   [ anon ]
00007fd5df621000      4K -----   [ anon ]
00007fd5df622000 395268K rw---   [ anon ]
00007fd5f7823000      4K -----   [ anon ]
00007fd5f7824000   8192K rw---   [ anon ]
00007fd5f8024000   2408K r-x--  libtsdclient.so
00007fd5f827e000   2044K -----  libtsdclient.so
00007fd5f847d000     32K r----  libtsdclient.so
00007fd5f8485000      4K rw---  libtsdclient.so
00007fd5f8486000 262160K rw---   [ anon ]
00007fd60848a000      4K -----   [ anon ]
00007fd60848b000   8192K rw---   [ anon ]
00007fd608c8b000    412K r-x--  libascend_hal.so
```

结论：由于**resnet50 demo**是同步推理，所以主机主要的压力都用来发送和接收**device**上的调用，而**device**资源占用不大，主机的能耗高些。主机的首次加载模型和读取输入数据会对主机磁盘产生一定压力，但是压力是瞬时的。

目录 TABLE OF CONTENT



2 规划数据的流转



通过左图可以观测到：

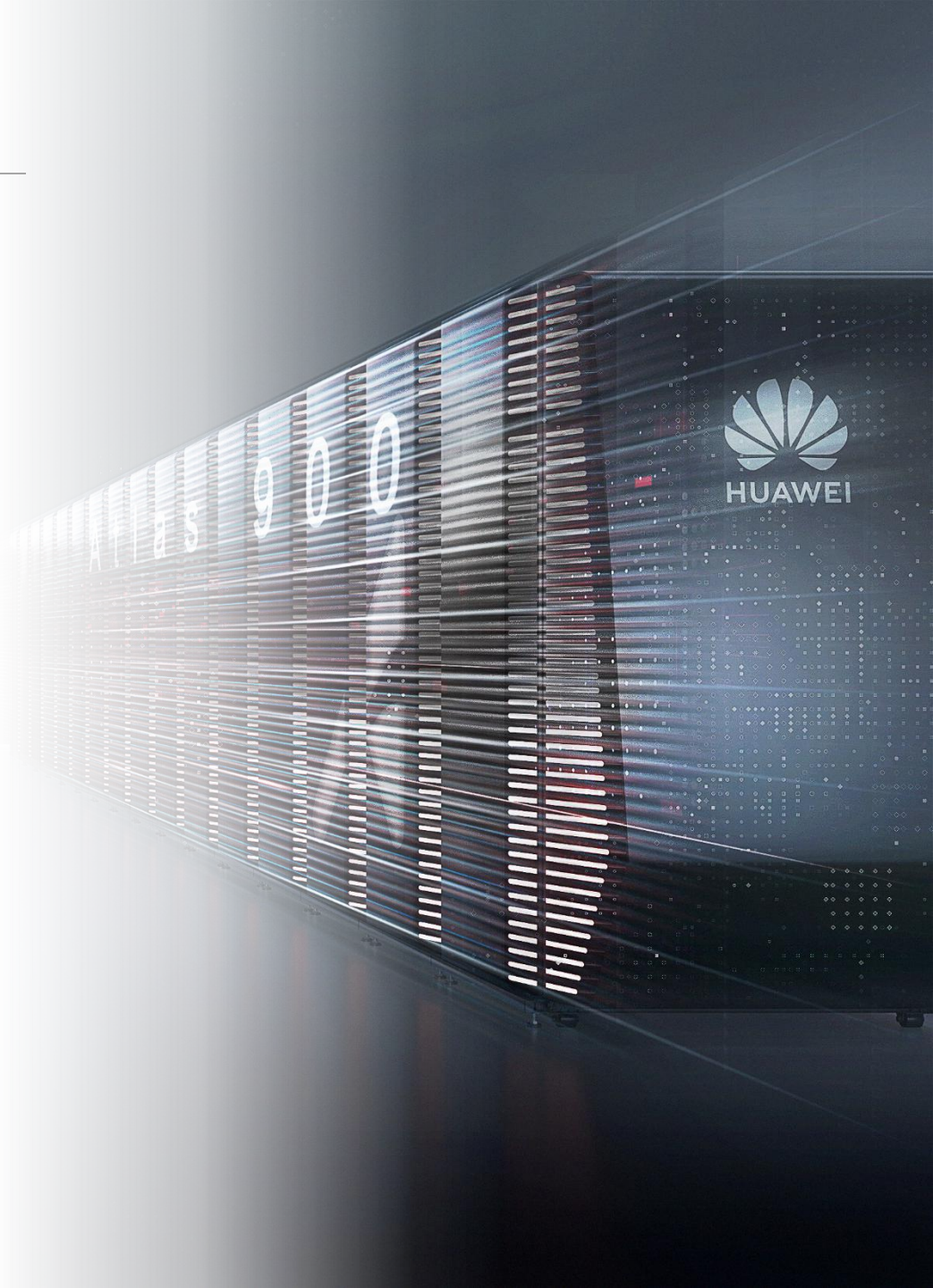
被处理的数据总是从主机一侧，流动到设备上，在设备上处理，然后再回传到主机上，所以目前一般的数据流，需要要多次拷贝完成推理，一方面组织数据提高吞吐，一方面协调各种数据之间的速度，以达到比较好的利用率。

为了组织数据，一般有下面**2**种情况：

1、流转型数据：每笔要处理的源数据，都通过新申请的方式传入被操作的**API**内，传入时指定输出，配合智能指针完成释放，通过模块功能实现拷贝，流转仅发送指针。

2、复用内存：内存在外部拷贝到池子里，通过标志位表示池子中的数据是否处于空闲，配合异步**API**实现数据在不搬运时，完成处理，待全部处理完以后，统一搬运一次。

目录 TABLE OF CONTENT



3 确认常用的指标 – 筹备

在组装流水线的过程中，会看到某些规格先达到瓶颈，所以我们需要确认如下几个关键性能规格，以保证单条流水线的设计速度和多条流水线设计输出符合预期：

1、各个解码器的性能规格，传入数据的速度不能高于该规格。以**1080P**图像的规格为例**DVPP**的规格如下：

名称	帧率（单位：FPS）	备注
Jpegd	**256**	**2**路
Jpege	**64**	
Vdec	**480**	**16**路、每路**30fps**
Venc	**30**	
Vpc	**1440**	**3**线程**YUV420SP**

流入**jpeg**的编解码速度，**video**的编解码速度，**vpc**的缩放速度。
解码时产生的带宽用量可以粗略的估算如下：
等效带宽 **=** 帧率规格*单帧像素个数*通道数（字节**/s** ）。
（例如：**1080P 16**路**vdec**解码等效带宽**= 16**路*每路**30**帧**/s* yuv**格式图像（**1080*1920*3/2**）**=1536MB/s**）

2、由于主机到设备的搬运速度有限，使用的是**PCIE 3.0x4** 的带宽，所以最大的请求数量和单笔业务的请求带宽也有限，各个主机性能不同所以这个值也会不同，建议分别使用单线程和多线程，发送不同大小包的数据到**device**上，验证主机端**1**个核最大的发送能力。

3、单个模型推理使用的内存带宽评估：可以对单个模型加压并测试，关注带宽利用率。
方法参考**1.4**

注：发送数据的测试一般建议使用**aclrtMemcpyAsync**接口和**aclrtSynchronizeStream**配合测试得到

3 确认常用的指标 – 总结

经过上面两轮的基本信息摸排应该能够评估出单条流水线的基本情况了

以单个昇腾**AI**处理器为例：

内存容量和带宽：

8G内存，带宽48GB/s

Device上的处理核：

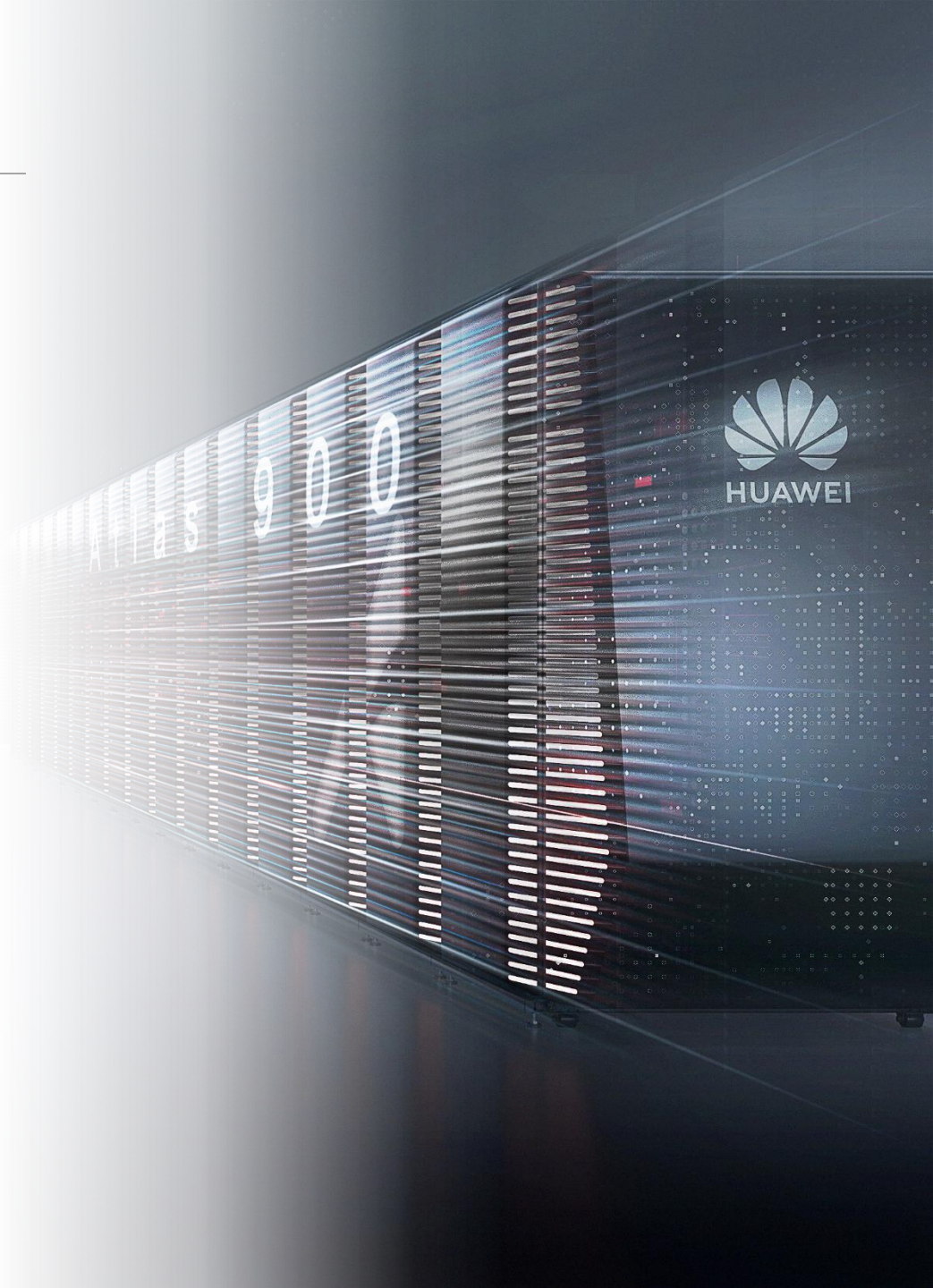
CPU核有8个，分为ctrl CPU和AI CPU，比例可配置；
CTRL CPU有4个，用于PCIE中断和数据搬运等等；
AI CPU核有4个，负责DVPP的中断负载，AICPU的算子压力等等。

主机的依赖的资源：

主机的**CPU**，承担发送接收数据**DMA**的功能。
主机的内存，承担外来数据流转的缓冲和读写，同时承载**DMA**到设备的带宽开销。
主机的**IO**资源，承担外来数据的接收和预处理，这些数据可能是**socket**传入，也可能来自**iscsi**磁盘或者本地磁盘。

注：如果上位机的发送能力有限，那么上位机会先到瓶颈，导致设备无法发挥能力。上述任意一个资源先到达瓶颈，都有可能影响整个系统的输出能力。

目 录 TABLE OF CONTENT



4 部署业务 – 常规的3种方案

根据业务流量的不同、业务相互依赖关系的不同，我们一般建议以下几种部署方式。

方案1 一个进程对应一个昇腾AI处理器：

使用单个进程在主机上进行隔离，进程中的线程数不多，业务聚焦，**1**个昇腾**AI**处理器的资源恰好用完，主机的大致消耗**5~8**个核以内（**2.5G**的**x86CPU**为例），主机上资源可以用**cgroup**（注）进行分割不影响主机其他业务。

方案2 多个进程对应1个昇腾AI处理器：

单条流一个进程，一个进程内创建**1**个**context**，**context**中存放各类内存和**stream**资源，部署简单同类业务只要启动多个进程**+**配置文件就可以实现阵列。例如：进程提供**web socket** 服务直接导出不同的端口区别进程，就可以实现相同或者类同的业务部署在**1**个昇腾**AI**处理器上。这种场景一般情况时**1**个昇腾**AI**处理器的算力无法被**1**个业务填满，需要几个业务连同动态使用**1**个昇腾**AI**处理器的能力。

方案3 1个进程对应多个昇腾AI处理器：

1个进程管理**1**个标卡，或者**2**个以上标卡，此种用法适用于业务流量非常大、可以多个昇腾**AI**处理器联动、要求数据在几个昇腾**AI**处理器上流通的场景，这种场景流水线设计往往非常复杂，需要多个**context**进行隔离，每个**context**有几个线程负责数据的搬运和信令的配发。

下图给出方案**1**的基本流程（也是本胶片后面时间的案例）

注：**Cgruop** 的具体配置方法 https://access.redhat.com/documentation/zh-cn/red_hat_enterprise_linux/7/html/resource_management_guide/index

HUAWEI

4 部署业务 – 规划一个流水线的几个关键变量

以常用的方案**1**为例规划一个有单条流水线的进程，关键数据结构如下：

◆ 全局创建1个context：用于存放各种资源。
◆ 创建1个线程：线程的数量用于隔离某一种业务，一个业务一个线程。
◆ 每个线程创建1个stream：利用stream的保序功能，顺序递交请求，保序处理各个流程的工作内容。
◆ 加载1个模型：因为对同一个model Id的模型，不能调用aclmdlExecuteAsync接口执行多Stream并发场景下的模型推理，所以我们需要每个线程加载一个id。
◆ 申请5块内存空间：通过上图的描述7个工步串联，那么至少需要device上4个buffer用于存放各自不同的数值。
上面几个步骤构成**1**条照片流，这样多个**stream**，每个线程独立使用**1**个**stream**下发压力就可以实现多条流水线了。

经过多条流水线的加压以后参考第**1**节的介绍就可以进一步调优资源用量了，从而我们也可以得到以下预想的代码，**stream**函数作为线程工作的逻辑。

```
450  void *steam(void *arg)
451  {
452      printf("run steam start line %d \n",__LINE__);
453      aclError ret;
454      uint64_t thread_number = *(uint64_t *)arg;
455      printf("thread_number %d\n",thread_number);
456      ret = aclrtSetCurrentContext(context);if(ret!=ACL_ERROR_NONE){printf("copy_to_device set current context fail! ret = %ld \n",ret);return nullptr;}
457      // 主循环 复用1个steam 不考虑放入保序的问题
458      while(run_flag){
459          ret = copy_to_device(thread_number);if(ret!=ACL_ERROR_NONE){printf("copy_to_device fail! thread_number = %lld \n",thread_number);return nullptr;}
460          ret = jpeg(thread_number);if(ret!=ACL_ERROR_NONE){printf("jpeg fail! thread_number = %lld \n",thread_number);return nullptr;}
461          ret = vpc(thread_number);if(ret!=ACL_ERROR_NONE){printf("vpc fail! thread_number = %lld \n",thread_number);return nullptr;}
462          ret = infer(thread_number);if(ret!=ACL_ERROR_NONE){printf("infer fail! thread_number = %lld \n",thread_number);return nullptr;}
463          ret = aclrtSynchronizeStream(stream[thread_number]);
464          if(ret!=ACL_ERROR_NONE){printf("SynchronizeStream fail! thread_number = %lld ret = %ld \n",thread_number,ret);return nullptr;}
465      }
466      printf("run steam end line %d \n",__LINE__);
467      return nullptr;
468  }
```

其他方法：客户自己使用队列来管理请求，模拟**stream**的功能也可以达到相同效果，这样用户可以更灵活的控制队列的长度和缓存量。

4 部署业务 – 一般的建议

①调优时尽量将瓶颈放到**AICORE**的推理边界上：应优先获得满载条件下的推理速度作为基线，让**AICORE**填满以完整利用昇腾**AI**处理器的能力。

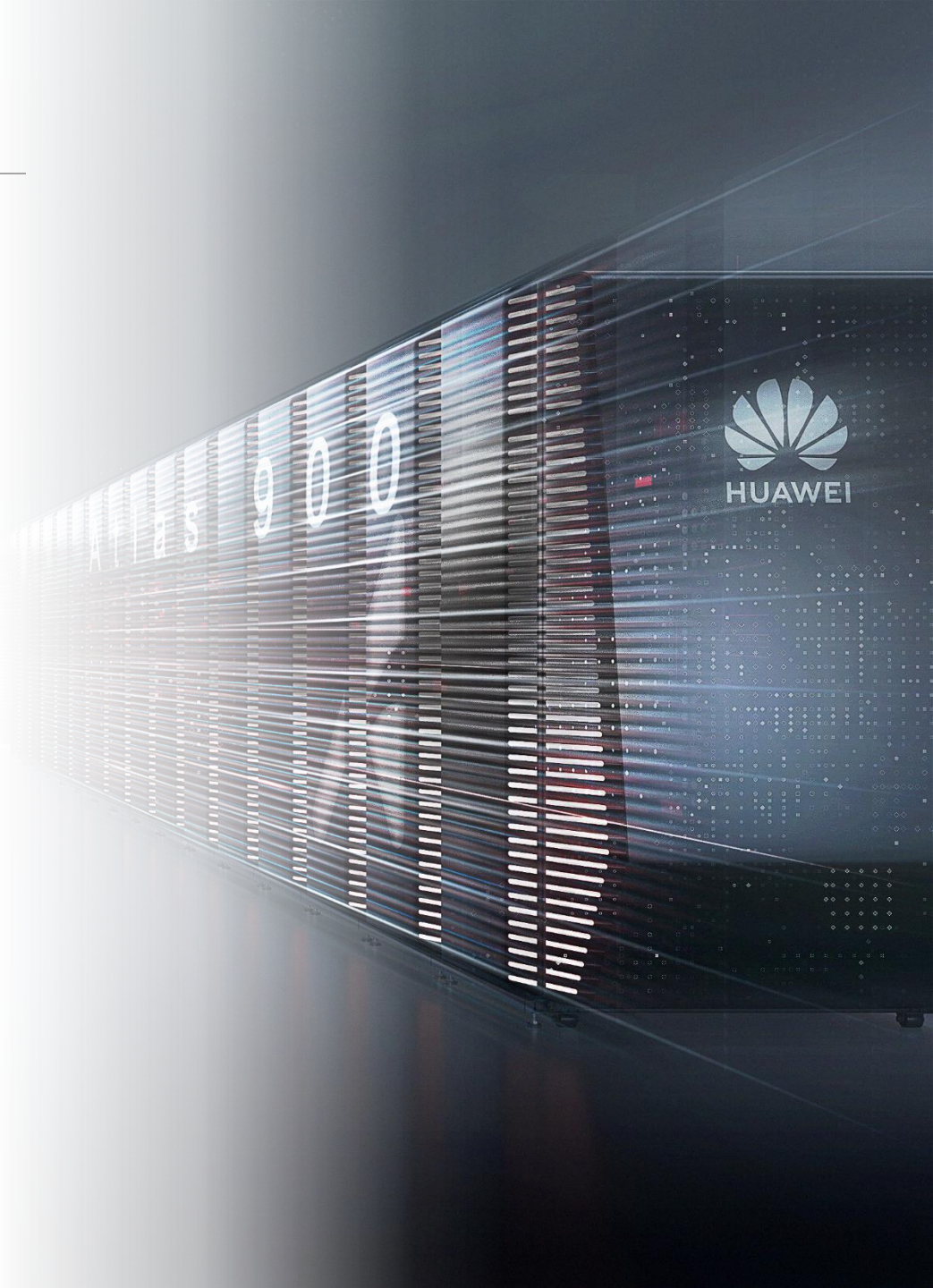
②尽量避免内存带宽成为瓶颈：减少拷贝内存，尽量将带宽能力用于推理、解码、**DMA**跨侧搬运数据。

③业务设计时尽量**1**个进程对应**1**个昇腾**AI**处理器，管理和资源分配比较容易，避免多个**context**在一个进程中出现的场景，会导致资源切分、下放变得困难，问题不好定位。

④主机的资源预留要做好计算，发送和接收的最高速度都要能满足昇腾**AI**处理器的需求。

⑤先从简单的流水线入手，摸底单个工步完成需要的资源情况，以及最大的线程数，基线找到以后再增加线程，多次调参以实现最大性能。

目录 TABLE OF CONTENT



5 案例实践 - 单独做同步推理最快能到多少?

下面以**resnet50**推理为例把**1**个昇腾**AI**处理器的能力用到极限，这个过程需要用到：搬运数据、解码、推理**3**个主要环节。调优这个业务最终的目的就是让这个业务能在期望的设备上跑到最佳，一方面要物尽其用，另一方面要减少重复和冲突的工作量，减小内耗。

在开始测试之前我们需要一份基线代码，然后编译一下，并且通过配置关闭**info**和**event**日志，只开启**error**级别。代码和命令如下：

代码调整：

acl_resnet50.cpp

日志调整：

adc --host xx.xx.xx.xx：22118 --log 'SetLogLevel(0)[error]' --device 0

adc --host xx.xx.xx.xx：22118 --log 'SetLogLevel(2)[disable]' --device 0

编译：

g++ ./acl_resnet50.cpp -o ./acl_resnet50 -I/home/HwHiAiUser/Ascend/ascend-toolkit/latest/acllib/include -I/usr/include/ -std=c++11 -fPIE -fstack-protector-all -Wl,-z,relro,-z,now,-z,noexecstack -pie -lrt -ldl -L/usr/lib -L/usr/local/lib -L/home/HwHiAiUser/Ascend/ascend-toolkit/latest/acllib/lib64 -L/usr/local/Ascend/driver/lib64 -L/home/HwHiAiUser/Ascend/ascend-toolkit/latest/atc/lib64 -lacl_dvpp -lascendcl

5 案例实践 - 单独做同步推理最快能到多少?

使用上面的基线代码，将推理的速度调到最大，将**aclmdlExecute**包装起来看看**800**次需要花费多长时间。结果是**3.143s**

```
for(uint64_t i=0;i<=0x320;i++){
        ret = aclmdlExecute(model_id, input_dataset, output_dataSet);if(ret!=ACL_ERROR_NONE){printf("add output DatasetBuffer fail! \n");return -1;}
}
```

```
[root@localhost fast_test]# time ./acl_resnet50
acl init ok
mem_size = 4723712 model_size = 51180544
aclmdlExecute success

real    0m3.143s
user    0m0.395s
sys     0m0.719s
```

主机侧分析：
通过监控可以发现主机的内存和**CPU**用量不多，由于是同步接口，所以主机侧的**cpu**要参与下发请求有一定的用量。截图如下：

```
top - 18:59:48 up 23:04,  8 users,  load average: 0.10, 0.04, 0.05
Tasks: 207 total,   1 running, 206 sleeping,   0 stopped,   0 zombie
%Cpu0  :   3.7/3.3     7[|||||||                                                  ]
%Cpu1  :   2.6/2.0     5[|||||                                                    ]
%Cpu2  :   1.3/1.0     2[|||                                                      ]
%Cpu3  :   2.3/2.0     4[||||                                                     ]
%Cpu4  :   0.0/0.3     0[                                                         ]
%Cpu5  :   0.0/0.0     0[                                                         ]
%Cpu6  :   0.0/0.0     0[                                                         ]
%Cpu7  :   0.0/0.0     0[                                                         ]
KiB Mem : 16141740 total, 13519052 free,  1587828 used,  1034860 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14189476 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
10410 root      20   0 8193.4g 993408   9448 S  20.5  6.2   0:04.39 acl_resnet50
 5630 HwHiAiU+  30  10  604060   9564   4360 S   0.7  0.1   1:30.37 ada
10432 root      20   0  162256   2396   1576 R   0.7  0.0   0:00.10 top
 7028 root      20   0  161304   6488   4808 S   0.3  0.0   0:00.78 sshd
    1 root      20   0  193880   6932   4152 S   0.0  0.0   0:02.26 systemd
```

Device侧分析：
Device上的**CPU**几乎没有压力，内存余量也充足。

```
Mem: 2621784K used, 5382792K free, 214532K shrd, 0K buff, 214532K cached
CPU0:  0.0% usr  0.0% sys  0.0% nic 95.4% idle  0.0% io  0.0% irq  4.5% sirq
CPU1:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU2:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU3:  0.0% usr  4.5% sys  0.0% nic 95.4% idle  0.0% io  0.0% irq  0.0% sirq
CPU4:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU5:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU6:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
CPU7:  0.0% usr  0.0% sys  0.0% nic  100% idle  0.0% io  0.0% irq  0.0% sirq
Load average: 6.71 6.76 6.75 2/171 8130
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 2776  2465 root     R     4156  0.0   3  0.0 top
 1496     1 HwHiAiUs S<    352m  4.4   0  0.0 /var/dmp_daemon -I -U 8087
 2375  2361 HwHiAiUs S    18868  0.2   0  0.0 sshd: HwHiAiUser@pts/0
 3838     2 root     IW        0  0.0   0  0.0 [kworker/0:0-eve]
 1305     1 HwHiAiUs S<    160m  2.0   2  0.0 /var/ascend_monitor
 1462     1 HwHiAiUs S     420m  5.3   2  0.0 /var/adda
```

```
[root@(none) /]# free -m
              total        used        free      shared  buff/cache   available
Mem:           7816        2350        5257         209         209        5172
Swap:             0           0           0
[root@(none) /]#
```

结论：

这个实验直观的给出了模型的串行推理速度**aclmdlExecute**的最大速度，设备能力还有很多余量，我们下一步尝试异步**API** **aclmdlExecuteAsync**的速度，**让框架和设备自动完成并行**

5 案例实践 - 单独做异步推理能多快?

这个例子里我们把同步**API**换成异步**API**看看能有多快，增加**stream** 的创建和**aclrtSynchronizeStream**的调用，给出代码如下：


```
70      for(uint64_t i=0;i<=0x320;i++){
71      //ret = aclmdlExecute(model_id, input_dataset, output_dataSet);if(ret!=ACL_ERROR_NONE){printf("add output DatasetBuffer fail! \n");return -1;}
72      ret = aclmdlExecuteAsync(model_id, input_dataset, output_dataSet,resnet_stream);if(ret!=ACL_ERROR_NONE){printf("add output DatasetBuffer fail! \n");return -1;}
73      }
74
75      ret=aclrtSynchronizeStream(resnet_stream);if(ret!=ACL_ERROR_NONE){printf("Synchronize fail! \n");return -1;}
76      printf("aclmdlExecuteAsync + aclrtSynchronizeStream  success\n");
```

关键代码就是把原来串行的**API**异步下发到**device**侧主机端在**sync**接口上等待设备的返回。
stream的保序执行，会让每次递交都按序依次执完，而主机不需要再关注太多的中断下发。

```
[root@localhost fast_test]# time ./acl_resnet50_async
acl init ok
mem_size = 4723712 model_size = 51180544
aclmdlExecuteAsync + aclrtSynchronizeStream  success

real    0m2.865s
user    0m0.812s
sys     0m1.125s
[root@localhost fast_test]# time ./acl_resnet50
acl init ok
mem_size = 4723712 model_size = 51180544
aclmdlExecute success

real    0m3.171s
user    0m0.284s
sys     0m0.537s
[root@localhost fast_test]#
```

经过修改以后在**device**上的内存带宽用量**39~40%**
AI CORE 97~ 98% 大致已经到达了瓶颈。
命令行工具参考 附录**2**：

```
mem_util = 32, ai_core_util = 97, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 98, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 32, ai_core_util = 97, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 97, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 39
```

在推理的上下加时间计算得到如下结果**2.006**秒/**800**帧：

```
[root@localhost fast_test]# ./acl_resnet50_async
acl init ok
mem_size = 4723712 model_size = 51180544
Completed in 2.006 secs
aclmdlExecuteAsync + aclrtSynchronizeStream  success
[root@localhost fast_test]#
```

结论：

经过上面两轮调整**resnet50**这个网络填满**AI core**需要**40%**左右的内存带宽，性能大致在**400**帧/**s**左右。
2个线程应该可以满载。

HUAWEI

5 案例实践 - 单个线程搬运数据能跑多快？ （请求合并版）

一笔推理业务运行中涉及到**2**次搬运，一次搬运数据进去，另外一次数据搬运出来。所以要知道搬运数据从里面到外面单个线程能跑多快。本次使用的样例打开一个**jpeg**的图片，把图片多次拼凑形成一个大的负载来验证带宽的能力和单个核的中断受理能力，观测主机的**CPU**和设备的**CPU**是否可以承载这个单线程流量，如果测试中发现其中**1**端的设备**CPU**用满了，那就代表无法完成这个搬运。

把这个图片拷贝1024次，每次75k左右，所以 75kX1024 = 75M左右，再拷贝800次（800次拷贝请求）

总容量 75M * 800 = 58.6GB，关键代码如下：

acl_memcpy_async.cpp

运行结果如下：

相当于**58.6GB/19.8s ≈ 3GB/s** 带宽。**1910**单个昇腾**AI**处理器的**PCIE**总线能力是

PCIE 3.0 x4 根据右图可见，带宽基本见底了。

```
[root@localhost fast_test]# ./acl_memcpy_async
acl init ok
Completed in 19.583 secs
[root@localhost fast_test]# 
```

设备监控如下：

1个线程的发送能力应该可以满足推理的需求，而且不会吃掉太多的内存带宽。使用相同的办法可以测试推理结束回传数据的速度，这里不再展开讲解。

```
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 5
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 6
```

Speeds and Feeds

PCIe Version	1.0	2.0	3.0	4.0
Line Code	8b/10b	8b/10b	128b/130b	128b/130b
Transfer Rate	2.5 GT/s	5 GT/s	8 GT/s	16 GT/s
x1 Bandwidth	250 MB/s	500 MB/s	984.6 MB/s	1.969 GB/s
x4	1 GB/s	2 GB/s	3.938 GB/s	7.877 GB/s
x8	2 GB/s	4 GB/s	7.877 GB/s	15.754 GB/s
x16	4 GB/s	8 GB/s	15.754 GB/s	31.508 GB/s

结论：

对于**jpeg**这样的压缩图片同步到**device**上单个线程基本上可以跑到带宽能力**78%**，如果需要可以使用**2**个**stream**以上达到最大的效果，**但是本例已经不需要了。**

当前图片的发送速度**=819200**帧**/19.58s ≈ 41838帧/s** （推理速度约**400**帧**/s**）

Device上内存带宽的消耗大致在**3GB/s**，总带宽约**48GB/s** 所以占用约**5%~6%**左右，

正好与设备侧的监控数据吻合。

HUAWEI

5 案例实践 - 单个线程搬运数据能跑多快？（请求不合并版）

上一个案例展示了中断合并后的效果（**1024**合并到**1**）这种方式比较适合大量碎块文件统一发送的场景，对于延时要求苛刻的情况就不再满足之列了，所以我们测试一次不合并单独发送可以跑到多少。

把**file_count** 改成**1**就实现这个用法，结果如下：

```
[root@localhost fast_test]# ./acl_memcpy_async
acl init ok
Completed in 0.279 secs
Completed in 0.287 secs
Completed in 0.291 secs
Completed in 0.289 secs
Completed in 0.307 secs
Completed in 0.293 secs
Completed in 0.296 secs
```

约**0.29**秒完成**800**次的请求，所以单个线程的收发能力为 **800/0.29s ≈ 2758 IOPS**。 主机侧的**cpu**利用量约**15%~16%**。见下图

```
top - 01:31:34 up 1 day,  5:36,  8 users,  load average: 0.39, 0.16, 0.08
Tasks: 209 total,   1 running, 208 sleeping,   0 stopped,   0 zombie
%Cpu0  :   1.3/1.7     3[|||                                                  ]
%Cpu1  :   0.0/0.0     0[                                                    ]
%Cpu2  :   0.0/0.0     0[                                                    ]
%Cpu3  :   4.7/1.7     6[|||||||                                             ]
%Cpu4  :   0.0/0.3     0[                                                    ]
%Cpu5  :   0.0/0.0     0[                                                    ]
%Cpu6  :   2.7/3.7     6[|||||||                                             ]
%Cpu7  :   0.0/0.0     0[                                                    ]
KiB Mem : 16141740 total, 13304300 free,  1545232 used,  1292208 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14225576 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
13462 root      20   0 8193.3g 939192   8372 S  15.9  5.8   0:06.66 acl_memcpy_asyn
 5323 HwHiAiU+   0 -20  526364   4356   1936 S   0.3  0.0   2:46.36 ascend_monitor
11700 root      20   0  161304   6460   4808 S   0.3  0.0   0:00.67 sshd
13339 root      20   0  162256   2396   1576 R   0.3  0.0   0:10.69 top
    1 root      20   0  193880   6932   4152 S   0.0  0.0   0:02.89 systemd
```

结论：

多数情况下我们不需要将帧数合并到**1024**，也不是每帧发一个，而是在**2**者之间取得一个合理值，让延时在可接受的范围内。

合并后的帧率 **= 819200帧/19.58s ≈ 41838帧/s**　　合并前的帧率 **= 800帧/0.29s ≈ 2758 帧/s**（推理速度约**400**帧/**s**）

显然推理速度低于搬运速度，我们不需要合并全部请求**单个线程发送就能让推理到达瓶颈了。**

5 案例实践 - 单个线程解码能跑多快？能抵得上推理的速度吗？

通过第3章，我们知道**jpeg**的解码速度是**256帧/s**，而**resnet50**网络的输入是**224*224*3**所以解码后需要缩放，这个环节要求有解码和缩放**2**个步骤，而**vpc**的性能非常高**1440帧/s**，所以**jpegd**的解码可能成为瓶颈。
下面我们来具体看下解码的能力在哪如何压出极限。关键代码如下：

acl_jpegd_Async.c

```
    ret = acldvppMalloc(&output_buffer, decode_out_buffer_size);if(ret!=ACL_ERROR_NONE){printf("output_buffer aclrtMalloc fail! \n");return -1;}

    decode_output_desc = acldvppCreatePicDesc();if(decode_output_desc == nullptr){printf("channel_desc create fail! \n");return -1;}
    acldvppSetPicDescData(decode_output_desc, output_buffer);
    acldvppSetPicDescFormat(decode_output_desc, PIXEL_FORMAT_YUV_SEMIPLANAR_420);
    acldvppSetPicDescWidth(decode_output_desc, input_width);
    acldvppSetPicDescHeight(decode_output_desc, input_height);
    acldvppSetPicDescWidthStride(decode_output_desc, decode_out_width_stride);
    acldvppSetPicDescHeightStride(decode_output_desc, decode_out_height_stride);
    acldvppSetPicDescSize(decode_output_desc, decode_out_buffer_size);

//for(uint64_t i=0;i<=0x320;i++){
    // 异步拷贝到device上
    double t = tick();
    for(uint64_t i=0;i<=0x320;i++){
        ret = acldvppJpegDecodeAsync(channel_desc, input_buffer, total_len,decode_output_desc, file_stream);
    }
    ret=aclrtSynchronizeStream(file_stream);if(ret!=ACL_ERROR_NONE){printf("Synchronize fail! \n");return -1;}
    t = tick() - t;printf("Completed in %.3f secs \n", t);

//}
```

```
[root@localhost fast_test]# ./acl_jpegd_Async
acl init success
aclrtSetDevice success
aclrtCreateContext success
aclrtCreateStream success
aclrtGetRunMode success
open file success file = ./cat.jpg
get jpg file size = 75825
total_len = 75825 file = 0x23c8480
input_buffer aclrtMalloc success
copy file to input buff success
Completed in 2.223 secs
Completed in 2.224 secs
Completed in 2.223 secs
Completed in 2.224 secs
Completed in 2.223 secs
Completed in 2.223 secs
Completed in 2.223 secs
```

经过测试后发现**800**帧图片大致需要**2.223**秒，也就是约**360帧/s**的速度，当前图片的大小是**1024x1536** ，参考**1080p**的解码速度是**256帧/s**，像素等效比**=1920*1080 / 1536*1024 = 1.318**，速度等效比**= 360 / 256=1.406** 两者接近，**小帧的解码效率还是略高些。**

图像

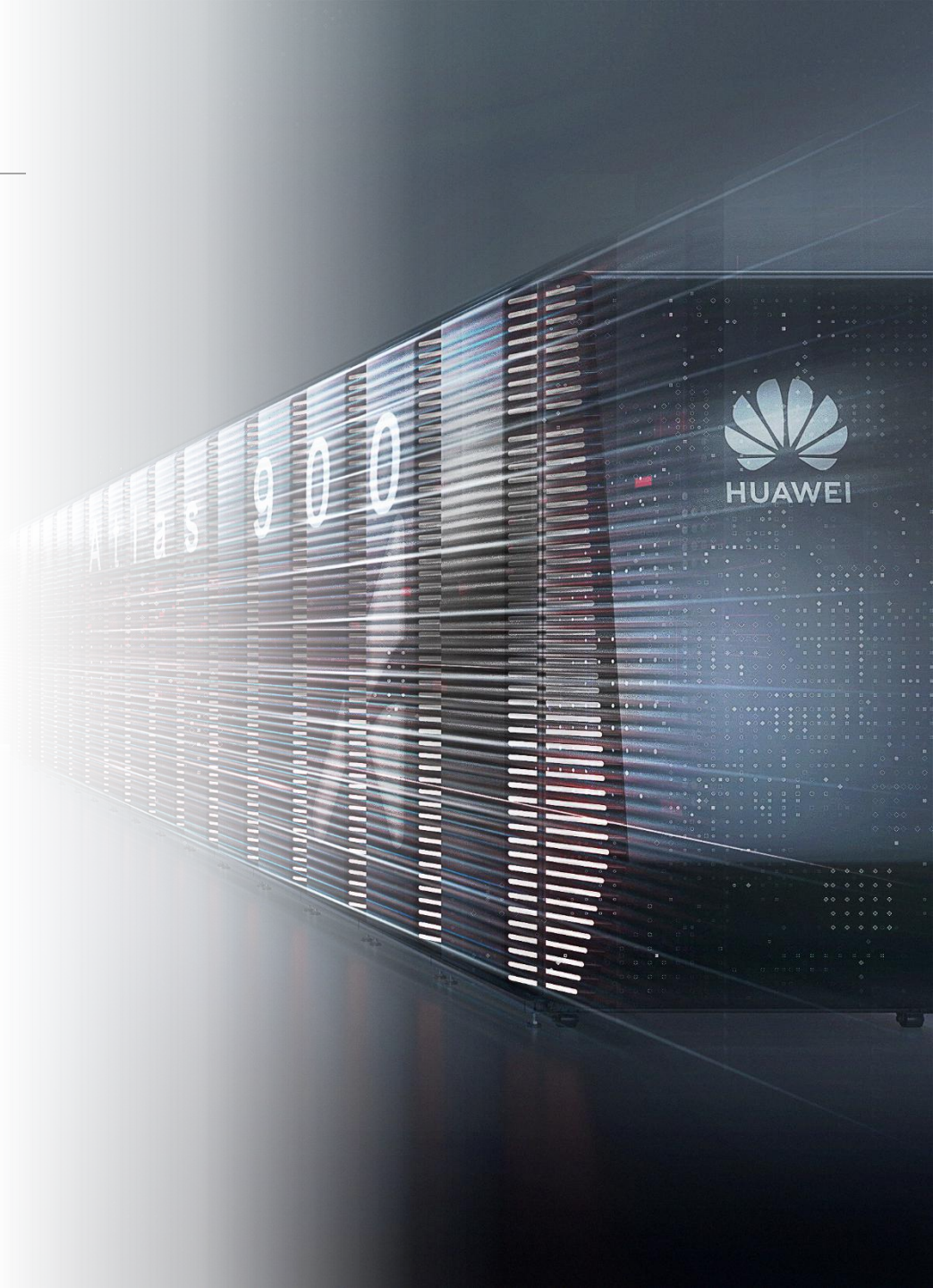
图像 ID	
分辨率	1024 × 1536
宽度	1024 像素
高度	1536 像素
水平分辨率	96 dpi
垂直分辨率	96 dpi
位深度	24

```
[root@localhost tool]# ./mem_bandwidth
mem_util = 32, ai_core_util = 0, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 0, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 0, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 0, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 0, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 0, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 1
```

结论：

异步下放**800**帧图片同时解码基本上会将**jpegd**的解码器压满，而且设备侧的压力情况并不高，**还有很多资源。AICPU**没有出现占满证明**device**的中断能力还有余量。

目录 TABLE OF CONTENT



6 案例实践 – 组装流水线

在开始组装流水线之前我们需要明确几个指标整理如下表：

名称	Host to device	jpeg解码	vpc缩放	Resnet50推理	Device to host	结果解析
延时	0.36ms	2.8ms	-	2.5ms	-	-
吞吐	3GB/s	360 QPS	-	400QPS	-	-

单线程：

通过上面的表明确得出结论，最大的解码速度还是没有推理快，其他的几个模块性能较高，暂时不会是瓶颈，所以我们下面组装流水线与验证时**jpeg**的解码能力可能是瓶颈。

多线程：

之前没有实测过需要测试一轮基线测试，再测并发量。

6 案例实践 - 先将拷贝和jpeg解码组合起来

通过多线程填充**2**个请求的间隙，然后通过各自线程的**stream**进行下放，关键代码如下：

```
void *steam(void *arg)
{
    printf("run steam start line %d \n",__LINE__);
    aclError ret;
    uint64_t thread_number = *(uint64_t *)arg;
    printf("thread_number %d\n",thread_number);
    ret = aclrtSetCurrentContext(context);if(ret!=ACL_ERROR_NONE){printf("copy_to_device set current context fail! ret = %ld \n",ret);return nullptr;}
    // 主循环 复用1个steam 不考虑放入保序的问题
    while(run_flag){
        // 异步拷贝
        ret = copy_to_device(thread_number);if(ret!=ACL_ERROR_NONE){printf("copy_to_device fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = jpeg(thread_number);if(ret!=ACL_ERROR_NONE){printf("jpeg fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = aclrtSynchronizeStream(stream[thread_number]);if(ret!=ACL_ERROR_NONE){printf("jpeg fail! thread_number = %lld ret = %ld \n",thread_number,ret);return nullptr;}
    }
    printf("run steam end line %d \n",__LINE__);
    return nullptr;
}
```

acl_stream_onew
ay.cpp

核心思想：

1、**1**个线程**1**个**stream**

2、线程索引当成数组索引，数组索引管理资源。

3、每个线程触发结束然后**sync**一次等待。

4、多个线程下发后主机侧直接等待

调整线程数量和下放请求的数量得到右侧表格：

线程数量	请求数	QPS	Host CPU(%)	Device 内存用量(%)	Ai cpu 利用率(%)	Ctrl cpu 利用率(%)	Device 内存带宽(%)
1	16	320	14	32	1~2	0	1
1	32	352	15.3	32	2~3	0	1
1	64	320	13.6	32	2~3	0	1~2
2	64	588	19.6~20.3	32	5~6	0	3
3	64	729	21.9~22.8	32	5~7	0	5
4	64	768	24.6~28.8	32	6~7	0	5~6
8	64	759	28.6~28.9	36	7	0	5

6 案例实践 - 先将拷贝和jpeg解码组合起来

三个线程以后出现了比较明显拐点，**768 QPS**，随着线程的增加后面帧率明显再次增加。

多数实际的工程各个**jpeg**照片流的大小不同，**并不一定全部fix到1080P，**所以多数情况下需要通过实测性能，确认最大吞吐。

下图给出了**3**个线程条件下输出的**QPS**实测值，有一定的抖动，但是可以看出平稳在**710~720 QPS**之间。其他的线程测试同理，就不一一截图了。

```
watch_frame_out loop_count = 0 copy-count = 0  jpeg-count = 0 QPS = 0.000
watch_frame_out loop_count = 1 copy-count = 3520  jpeg-count = 3520 QPS = 704.000
watch_frame_out loop_count = 2 copy-count = 3456  jpeg-count = 3456 QPS = 691.000
watch_frame_out loop_count = 3 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 4 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 5 copy-count = 3520  jpeg-count = 3520 QPS = 704.000
watch_frame_out loop_count = 6 copy-count = 3584  jpeg-count = 3584 QPS = 716.000
watch_frame_out loop_count = 7 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 8 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 9 copy-count = 3456  jpeg-count = 3456 QPS = 691.000
watch_frame_out loop_count = 10 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 11 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 12 copy-count = 3584  jpeg-count = 3584 QPS = 716.000
watch_frame_out loop_count = 13 copy-count = 3520  jpeg-count = 3520 QPS = 704.000
watch_frame_out loop_count = 14 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 15 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 16 copy-count = 3456  jpeg-count = 3456 QPS = 691.000
watch_frame_out loop_count = 17 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 18 copy-count = 3648  jpeg-count = 3648 QPS = 729.000
watch_frame_out loop_count = 19 copy-count = 3584  jpeg-count = 3584 QPS = 716.000
watch_frame_out end
```

结论：

只有拷贝数据和jpeg解码的条件下，单线程解码速度 为352QPS

在8线程时饱和值在760QPS左右，偶尔会出现780QPS。

8线程时内存的利用量超过了32%，AI CORE达到了100%，主机侧CPU利用率基本上平滑到28%左右，也就是不到0.3个核。

6~8线程增加过程中没观测到明显的CPU用量增长，也就是流水线出现了瓶颈，增加线程不会再增加吞吐了。

6 案例实践 - 把推理放进流水线（单线程）

推理解码后的图片还是原始分辨率，**resnet50是224*224*3的shape**所以我们需要把图片调整到对应的大小，串联缩放**vpc**功能，使用**vpc**的批量缩放和抠图功能。
关键代码如下：

acl_stream_onew
ay_resnet50.cpp

```
void *steam(void *arg)
{
    printf("run steam start line %d \n",__LINE__);
    aclError ret;
    uint64_t thread_number = *(uint64_t *)arg;
    printf("thread_number %d\n",thread_number);
    ret = aclrtSetCurrentContext(context);if(ret!=ACL_ERROR_NONE){printf("copy_to_device set current context fail! ret = %ld \n",ret);return nullptr;}
    // 主循环 复用1个steam 不考虑放入保序的问题
    while(run_flag){
        // 异步拷贝
        ret = copy_to_device(thread_number);if(ret!=ACL_ERROR_NONE){printf("copy_to_device fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = jpeg(thread_number);if(ret!=ACL_ERROR_NONE){printf("jpeg fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = vpc(thread_number);if(ret!=ACL_ERROR_NONE){printf("vpc fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = infer(thread_number);if(ret!=ACL_ERROR_NONE){printf("infer fail! thread_number = %lld \n",thread_number);return nullptr;}
        ret = aclrtSynchronizeStream(stream[thread_number]);if(ret!=ACL_ERROR_NONE){printf("SynchronizeStream fail! thread_number = %lld ret = %ld \n",thr
    }
    printf("run steam end line %d \n",__LINE__);
    return nullptr;
}
```

6 案例实践 - 把推理放进流水线（单线程）

经过测试后发现单线程的推理能力大致为**160~180QPS**，相比**resnet50**的单独推理速度慢了很多，监控显示**AI CORE**的利用率**4x%** 还有很多余量。

```
vpc_channel_desc create success 0
vpc channel create success 0
vpc_batch_pic_desc create success 0
copy_to_device_stream create success 0
----------- init_fromwork line 217 dvpp stuf end jpeg out size = 2359296 vpc out size = 75264
----------- init_fromwork line 219 mem buff init
----------- init_fromwork line 246 mem buff end
----------- init_fromwork line 252 end
main init_fromwork success
----------- line 447 main
i = 0 ,tid[i] = -1042102528
run steam start line 413
watch_frame_out loop_count = 0 copy-count = 0  jpeg-count = 0 vpc-count = 0 QPS = 0.000
thread_number 0
watch_frame_out loop_count = 1 copy-count = 896  jpeg-count = 896 vpc-count = 896 QPS = 179.000
watch_frame_out loop_count = 2 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 3 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 4 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 5 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 6 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 7 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
watch_frame_out loop_count = 8 copy-count = 832  jpeg-count = 832 vpc-count = 832 QPS = 166.000
```

Device的性能指标如下：

```
mem_util = 32, ai_core_util = 42, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 40, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 22
mem_util = 32, ai_core_util = 41, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 40, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 42, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 28
mem_util = 32, ai_core_util = 39, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 42, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 15
mem_util = 32, ai_core_util = 39, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 40, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 9
mem_util = 32, ai_core_util = 40, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 12
mem_util = 32, ai_core_util = 42, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 42, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 12
mem_util = 32, ai_core_util = 40, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 7
mem_util = 32, ai_core_util = 42, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 31
mem_util = 32, ai_core_util = 42, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 18
mem_util = 32, ai_core_util = 41, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 1
mem_util = 32, ai_core_util = 40, ai_cpu_util = 1, ctrl_cpu_util = 0, mem_band_width = 26
```

主机**CPU**的参与度也不高：

```
top - 04:11:21 up 27 min,  4 users,  load average: 0.05, 0.04, 0.04
Threads:   7 total,   1 running,   6 sleeping,   0 stopped,   0 zombie
%Cpu0  :   0.7/2.0     3[|||                                                  ]
%Cpu1  :   0.0/0.0     0[                                                     ]
%Cpu2  :   0.0/0.0     0[                                                     ]
%Cpu3  :   1.0/1.3     2[||                                                   ]
%Cpu4  :   0.0/0.0     0[                                                     ]
%Cpu5  :   1.3/0.3     2[|                                                    ]
%Cpu6  :   0.0/0.0     0[                                                     ]
%Cpu7  :   0.0/0.0     0[                                                     ]
KiB Mem : 16141748 total, 13994140 free,  1680020 used,   467588 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14131784 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 6645 root      20   0 8193.6g   1.1g   9880 R  4.0  6.9   0:04.82 RT_RECV
 6662 root      20   0 8193.6g   1.1g   9880 S  2.7  6.9   0:03.24 acl_stream_onew
 6644 root      20   0 8193.6g   1.1g   9880 S  1.7  6.9   0:02.46 RT_SEND
 6641 root      20   0 8193.6g   1.1g   9880 S  0.0  6.9   0:00.41 acl_stream_onew
 6642 root      20   0 8193.6g   1.1g   9880 S  0.0  6.9   0:00.00 LevelNotifyWatc
 6643 root      20   0 8193.6g   1.1g   9880 S  0.0  6.9   0:00.00 LevelNotifyWatc
 6663 root      20   0 8193.6g   1.1g   9880 S  0.0  6.9   0:00.00 acl_stream_onew
```

结论：

1、单条线程在发送和接收数据时有很多间隙，流水线上的各个模块没有发挥到最佳，**A**模块在工作时**B**模块在睡觉。通过上面主机的**CPU**用量可以得出结论，时间消耗在了发送和返回**API**中断上了，而且都在等待，控制回传的线程都在睡觉没有干活。

2、从设备内存带宽的波峰波谷效果分析，对比只有异步单**batch**推理的监控可得出结论，波谷的带宽处基本上在准备数据，此时**AI CORE**内外带宽压力并不高。

6 案例实践 - 把推理放进流水线（多线程）

推理的单线程速度不高**AI CORE**的用量在 **4x%**，那么匹配多少条线程合适？我们不妨从**2**线程开始尝试看看效果：

Device 2线程：

```
mem_util = 32, ai_core_util = 71, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 32, ai_core_util = 76, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 24
mem_util = 32, ai_core_util = 77, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 78, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 24
mem_util = 32, ai_core_util = 78, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 32
mem_util = 32, ai_core_util = 78, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 27
mem_util = 32, ai_core_util = 78, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 78, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 24
mem_util = 32, ai_core_util = 77, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 32, ai_core_util = 77, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 77, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 30
mem_util = 32, ai_core_util = 78, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 32, ai_core_util = 77, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 32, ai_core_util = 77, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 77, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
```

主机 **2**线程：

```
top - 04:48:20 up  1:04,  4 users,  load average: 0.04, 0.05, 0.05
Tasks: 207 total,   1 running, 206 sleeping,   0 stopped,   0 zombie
%Cpu0  :   0.7/1.7     2[|||                                                  ]
%Cpu1  :   0.3/0.7     1[|                                                    ]
%Cpu2  :   0.3/0.0     0[                                                     ]
%Cpu3  :   0.7/1.7     2[|||                                                  ]
%Cpu4  :   0.0/0.0     0[                                                     ]
%Cpu5  :   2.0/0.0     2[||                                                   ]
%Cpu6  :   0.0/0.3     0[                                                     ]
%Cpu7  :   0.0/0.0     0[                                                     ]
KiB Mem : 16141748 total, 13836856 free,  1754704 used,   550188 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14029652 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 7073 root      20   0 8193.6g   1.1g   9868 S  10.3  7.0   0:33.77 acl_stream_onew
 2109 root      20   0       0      0      0 S   0.3  0.0   0:01.82 xfsaild/nvme0n1
```

应用 输出 **2**线程：

```
watch_frame_out loop_count = 12 copy-count = 1600  jpeg-count = 1600 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 13 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 14 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 15 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 16 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 17 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 18 copy-count = 1600  jpeg-count = 1600 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 19 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 20 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 21 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 22 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 23 copy-count = 1600  jpeg-count = 1564 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 24 copy-count = 1536  jpeg-count = 1572 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 25 copy-count = 1600  jpeg-count = 1600 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 26 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
```

结论：

1、帧率从**160 QPS**提升到了**300QPS**，获取了接近**2**倍的提升。
2、主机的**CPU**利用率 从**4%~** 升到了**10%~**
3、内存带宽的波峰波谷相对平滑了，内存的用量和带宽稳步上升
4、**AICORE**的用量也从原来的**40%**涨到了接近**80%** 代表双线程有了接近**2**倍的收益。

流水线的条数可以进一步提高到**4**尝试下。

6 案例实践 - 把推理放进流水线（多线程）

Device 4线程：

```
mem_util = 32, ai_core_util = 88, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 41
mem_util = 32, ai_core_util = 88, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 33
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 88, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 88, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 88, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 88, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 33
mem_util = 32, ai_core_util = 88, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 35
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 32, ai_core_util = 88, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 35
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 33
```

Device 6线程：

```
mem_util = 32, ai_core_util = 1, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 34, ai_core_util = 31, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 41
mem_util = 34, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 34, ai_core_util = 92, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 92, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 5, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 34, ai_core_util = 93, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 39
```

应用 **4**线程：

```
i = 3 ,tid[i] = 1139201792
watch_frame_out loop_count = 1 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 2 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 3 copy-count = 1728  jpeg-count = 1728 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 4 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 5 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 6 copy-count = 1728  jpeg-count = 1683 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 7 copy-count = 1664  jpeg-count = 1709 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 8 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 9 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
```

应用 **6**线程：

```
watch_frame_out loop_count = 11 copy-count = 1856  jpeg-count = 1856 vpc-count = 1856 QPS = 371.000
watch_frame_out loop_count = 12 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 13 copy-count = 1856  jpeg-count = 1856 vpc-count = 1856 QPS = 371.000
watch_frame_out loop_count = 14 copy-count = 1920  jpeg-count = 1920 vpc-count = 1920 QPS = 384.000
watch_frame_out loop_count = 15 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 16 copy-count = 1856  jpeg-count = 1856 vpc-count = 1856 QPS = 371.000
watch_frame_out loop_count = 17 copy-count = 1856  jpeg-count = 1856 vpc-count = 1856 QPS = 371.000
watch_frame_out loop_count = 18 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 19 copy-count = 1920  jpeg-count = 1920 vpc-count = 1920 QPS = 384.000
```

6 案例实践 - 把推理放进流水线（多线程）

Device 8线程：

```
mem_util = 42, ai_core_util = 17, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 32
mem_util = 42, ai_core_util = 86, ai_cpu_util = 5, ctrl_cpu_util = 0, mem_band_width = 31
mem_util = 42, ai_core_util = 86, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 42, ai_core_util = 86, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 42, ai_core_util = 86, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
```

应用 **8**线程：

```
i = 7 ,tid[i] = 659605248
watch_frame_out loop_count = 0 copy-count = 13  jpeg-count = 0 vpc-count = 0 QPS = 0.000
watch_frame_out loop_count = 1 copy-count = 1779  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 2 copy-count = 1728  jpeg-count = 1728 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 3 copy-count = 1600  jpeg-count = 1600 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 4 copy-count = 1600  jpeg-count = 1600 vpc-count = 1600 QPS = 320.000
watch_frame_out loop_count = 5 copy-count = 1728  jpeg-count = 1728 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 6 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 7 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
```

结论：

1、**8**线程时每次**sync**递交**64**个请求的情况下，出现了拐点，拐点发生在**6~8**之间。此时再增加线程已经无意义，**AICORE**的利用量已经下降，代表递交请求的切换过于频繁，**TS CORE** 基本已经到达瓶颈了，**AICORE**的利用率开始下降。（递交过于频繁导致）

2、**jpeg**的**8**线程时可以到**760 QPS**，**QPS**是输出的**2**倍多，也代表着**jpeg**在处理完请求后进入了睡觉状态，等待推理同步完成。

6 案例实践 - 增加或减少队里中的缓存长度

之前的尝试都是调整了线程的数量，下面尝试将线程固定到**8**，调整每次递交的业务数量进行摸高，观测**QPS**是否还会继续上涨，进一步验证**TS CORE**下发**task**的能力，是否真的到瓶颈了。

关键代码如下：

```
int32_t device_id = 0;                    // 压力所在的deviceid
#define COUNT_TIME 32                     // 单笔异步递交的frame 帧数
#define THREAD_COUNT 8                    // 并发工作的线程数
//uint32_t COUNT_TIME = 16;
```

降到**32**

```
watch_frame_out loop_count = 1 copy-count = 1693  jpeg-count = 1696 vpc-count = 1696 QPS = 339.000
watch_frame_out loop_count = 2 copy-count = 1728  jpeg-count = 1728 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 3 copy-count = 1632  jpeg-count = 1632 vpc-count = 1632 QPS = 326.000
watch_frame_out loop_count = 4 copy-count = 1728  jpeg-count = 1728 vpc-count = 1728 QPS = 345.000
watch_frame_out loop_count = 5 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 6 copy-count = 1696  jpeg-count = 1696 vpc-count = 1696 QPS = 339.000
watch_frame_out loop_count = 7 copy-count = 1696  jpeg-count = 1696 vpc-count = 1696 QPS = 339.000
```

加到**128**

```
copy_to_device_stream create success 7
----------- init_fromwork line 213 dvpp stuf end jpeg out size = 2359296 vpc out size = 75264
----------- init_fromwork line 215 mem buff init
aclrtMalloc async_jpeg_out 7 60 fail! ret = 500000
main init_fromwork failed !
[root@localhost fast_test]#
```

```
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 33
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 34
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 87, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 87, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 32, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
```

```
, result=6, bytesize=2359328.
[ERROR] DEVMM(7778,acl_stream_oneway_resnet50):2020-09-18-05:02:50.083.077 [hardware/npu_inc/../dev_platform/devmm/devmm/devmm_svm.c:153][devmm_alloc_managed 153] <curpid:7778,0x1e62> <errno:2> new_heap_alloc out of memory, result=6,
bytesize=2359328.
[ERROR] RUNTIME(7778,acl_stream_oneway_resnet50):2020-09-18-05:02:50.083.087 [runtime/feature/src/npu_driver.cc:800]7778 DevDvppMemAlloc:halMemAlloc failed: errorcode=6, device_id=0, size=2359328
[ERROR] ASCENDCL(7778,acl_stream_oneway_resnet50):2020-09-18-05:02:50.083.098 [acl/types/dvpp.cpp:54]7778 acldvppMalloc: "alloc device memory for dvpp failed, result = 117571600"
[root@localhost slog]#
```

6 案例实践 - 增加或减少队里中的缓存长度（7线程的补充）

加到**128**时内存不足，尝试降低到**7**个线程尝试：

```
watch_frame_out loop_count = 181 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 182 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 183 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 184 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 185 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 186 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 187 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 188 copy-count = 1408  jpeg-count = 1408 vpc-count = 1408 QPS = 281.000
watch_frame_out loop_count = 189 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 190 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 191 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 192 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
watch_frame_out loop_count = 193 copy-count = 1536  jpeg-count = 1536 vpc-count = 1536 QPS = 307.000
watch_frame_out loop_count = 194 copy-count = 1664  jpeg-count = 1664 vpc-count = 1664 QPS = 332.000
watch_frame_out loop_count = 195 copy-count = 1792  jpeg-count = 1792 vpc-count = 1792 QPS = 358.000
```

```
mem_util = 62, ai_core_util = 84, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 84, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 21
mem_util = 62, ai_core_util = 83, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 25
mem_util = 62, ai_core_util = 84, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 62, ai_core_util = 84, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 82, ai_cpu_util = 5, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 62, ai_core_util = 84, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 84, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 35
mem_util = 62, ai_core_util = 84, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 33
mem_util = 62, ai_core_util = 85, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 62, ai_core_util = 84, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 85, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 25
mem_util = 62, ai_core_util = 82, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 84, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 84, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 62, ai_core_util = 83, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 62, ai_core_util = 85, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
```

结论：

1、**8**线程的 **32**请求的没有改善还是**330+QPS** ，与**64**结论一致，进一步验证了 **TS CORE**成为了瓶颈的事实。

2、**128**请求的直接无法启动，内存不足降低一个线程再试。

3、**7**个线程以后发现没有**QPS**任何改善。

6线程**64**帧**sync**基本上达到了最高速度**370QPS**接近**400QPS**的极限速度（**370/400≈92.7%**），也就是说单**batch**的能力基本到上限，下一步可以增加**batch**数了。

6 案例实践 – 多batch和单batch

经过上面的调优发现，**resnet50**网络带**AIPP**，在**224*224*3**的条件下，可以达到**370QPS**了，那么输出的能力如果进一步提高，就需要将下发推理的请求聚合起来，减少中断次数，加大搬运吞吐，减少**AICORE**内外的数据同步，从而节省递交频繁的开销。

我们先看看**64Batch**条件下极限推理能力在哪，然后锚定目标，再做尝试。

64 batch 递交**800**个请求的场景：

```
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 64
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 66
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 66
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 68
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 66
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 64
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 67
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 69
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 67
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 72
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 66
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 73
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 66
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 72
mem_util = 32, ai_core_util = 100, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 67
```

```
[root@localhost fast_test]# ./acl_resnet50_async_64b
acl init ok
get model input  index = 0 get size = 4816896
get model output index = 0 get size = 256000
mem_size = 82083328 model_size = 51409920

Completed in 101.475 secs
aclmdlExecuteAsync + aclrtSynchronizeStream  success
```

出帧率衰减到 **800/101.475≈7.88QPS ，接近8帧/s，这个是很异常的，是值得分析的。**

AICORE的利用率很高但是出帧率却降低了2个数量级，这种情况是不可接受的。

6 案例实践 – 多batch和单batch （解决一个瓶颈）

直接观测主机的**CPU**利用率：

```
top - 18:46:32 up 15:03,  4 users,  load average: 0.16, 0.18, 0.14
Threads:   7 total,   1 running,   6 sleeping,   0 stopped,   0 zombie
%Cpu0  :   0.0/0.0     0[                                                                                                    ]
%Cpu1  :   0.0/0.0     0[                                                                                                    ]
%Cpu2  :   0.0/0.0     0[                                                                                                    ]
%Cpu3  :   0.0/0.0     0[                                                                                                    ]
%Cpu4  :   0.0/0.0     0[                                                                                                    ]
%Cpu5  :   0.0/0.0     0[                                                                                                    ]
%Cpu6  : 100.0/0.0   100[||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
%Cpu7  :   0.0/0.0     0[                                                                                                    ]
KiB Mem : 16141748 total, 13523432 free,  1717136 used,   901180 buff/cache
KiB Swap:  8193020 total,  8193020 free,        0 used. 14044660 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
14400 root      20   0 8193.6g   1.0g   9704 R 99.9  6.8   0:13.60 RT_SEND
14397 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.30 acl_resnet50_as
14398 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.00 LevelNotifyWatc
14399 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.00 LevelNotifyWatc
14401 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.00 RT_RECV
14434 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.02 acl_resnet50_as
14435 root      20   0 8193.6g   1.0g   9704 S  0.0  6.8   0:00.02 acl_resnet50_as
```

结论：

规避主机性能瓶颈以后，得到的等效帧速 **128*64 = 8192/16.216s ≈ 505 QPS** 比单个帧推理提高高了约**130**帧

```
[root@localhost fast_test]# ./acl_resnet50_async_64b
acl init ok
get model input  index = 0 get size = 4816896
get model output index = 0 get size = 256000
run steam start line 43
thread_number 0
Completed in 16.216 secs
aclmdlExecuteAsync + aclrtSynchronizeStream  success
aclmdlUnload success
[root@localhost fast_test]#
```

分析后主机的**CPU**成为性能瓶颈，递交的次数太频繁导致主机**CPU**的利用率单核直接达到了极限。而且是框架的工作线程达到了瓶颈。**降低异步推理的次数再试。**从**800**降低到**128**防止出现主机端的瓶颈。关键代码如下：

```
//for(uint64_t j=0;j<6;j++){
    for(uint64_t i=0;i<800;i++){
        //ret = aclmdlExecute(model_id, input_dataset, output_dataset);if(ret!=ACL_ERROR_NONE){printf("add output DatasetBuffer fail! \n");return -1;}
        ret = aclmdlExecuteAsync(model_id[thread_number], input_dataset[thread_number], output_dataset[thread_number],stream[thread_number]);
        if(ret!=ACL_ERROR_NONE){printf("add output DatasetBuffer fail! \n");return nullptr;}
    }
        ret=aclrtSynchronizeStream(stream[thread_number]);if(ret!=ACL_ERROR_NONE){printf("Synchronize fail! \n");return nullptr;}
//}
    t = tick() - t;printf("Completed in %.3f secs \n", t);
    return nullptr;
```

acl_resnet50_asyn
c_64b.cpp

6 案例实践 – 多batch和单batch （进一步探索）

经过上面的分析将**batch**数增加以后确实可以带来一定收益，但是增加多少才可以最优？各个模块不会成为瓶颈。
所以我们需要一张表描述**batch**数、单帧延时、系统资源利用率还有**QPS**的关系。

Batch数	128 轮帧耗时	AI CORE利用率%	Device内存带宽%	QPS
8	16.231	100	68	63.08915
14	3.425	100	62	523.2117
16	2.788	100	49	734.5768
18	4.350	100	63	529.6552
20	4.83	100	63	530.0207
24	6.000	100	64	512
28	7.009	100	65	511.3426
32	6.577	100	58	622.7763
64	16.216	100	72	505.1801

结论：

1、经过上面的测试发现各种**batch**在推理速度上是不同的，同一个模型并非**batch**越大越好，需要根据模型的实际情况进行分析。
2、**16 batch**的性能最高我们摸到了**734QPS**，接近单**batch**的**2**倍了。

6 案例实践 – 全流程打通

流水线的改进版本，直接将推理拉到**16 batch**上，修正代码以后适配多**batch**中间增加了**1**步拷贝，挤掉空余的间隙内存，拷贝的过程中要注意**aclrtMemsetAsync API**要求源和目的地址都要**64**对齐。

真实场景**1**：

计算长度可能不是**64**对齐，需要**stride**到**64**像素或者**128**像素（**VPC**地址**128**对齐时性能最高），**本例的输出VPC长度=224*224*3/2=75264/64=1176 恰好是64的倍数**，所以自动形成了对齐。

真实场景**2**：

对齐的话最好调整输出边宽高的像素到**128**的倍数，**然后使用AIPP裁边，实现0拷贝。**

真实场景**3**：

对于多种输入的网络，可能依赖于几个输入，而几个输入又依赖其他的计算过程，这就导致了时间不对齐，此时就需要拷贝数据，中间增加**buff**实现，本例就模拟了这种可能最坏的情况。

acl_stream_onew
ay_resnet50_16b.c

线程数	AI CORE利用率%	Device内存带宽%	QPS
1	40	波动	212
2	65	50	322
4	100	49~50	608
8	100	49~50	627

结论：

经过上面的优化可以把整网端到端的速度锁定在**630QPS**左右，极限出帧为**730QPS**，所以相当于达到了**86.3%**的极限速度。

6 案例实践 – 后记

1、流水线的各个步骤都是用一个**stream**串联起来的，此种方法简单但是无法再提升**QPS**数，可以通过**event**的方式直接串联**stream**但是需要支持**query**功能，目前还无法实现。

2、没有搬运数据回主机解析的步骤，如果需要可以再增加异步操作。
3、通过**event**实现**2**个**stream**互等的机制，这样一来可以保证每条流在工作时穿插进行，工步与工步之间实现多个**buff**轮询，最大化发挥流水线的能力。
aclrtCreateEvent-->aclrtRecordEvent-->aclrtStreamWaitEvent-->aclrtResetEvent

其他参考：

1、开启**ssh**

F:\code\fast_test\tool\set_ssh.c

2、获取**device**资源占用的小工具

F:\code\fast_test\tool\mem_bandwidth.c

3、加压的过程中模型数量较多时应该给出一定预热时间，因为首帧的速度较慢过一段时间稳定后再观测效果。

```
mem_util = 32, ai_core_util = 1, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 32, ai_core_util = 0, ai_cpu_util = 0, ctrl_cpu_util = 0, mem_band_width = 0
mem_util = 34, ai_core_util = 31, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 41
mem_util = 34, ai_core_util = 87, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 37
mem_util = 34, ai_core_util = 92, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 92, ai_cpu_util = 2, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 5, ctrl_cpu_util = 0, mem_band_width = 36
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 38
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 40
mem_util = 34, ai_core_util = 93, ai_cpu_util = 4, ctrl_cpu_util = 0, mem_band_width = 39
mem_util = 34, ai_core_util = 93, ai_cpu_util = 3, ctrl_cpu_util = 0, mem_band_width = 39
```

7 FAQ

1、单进程，多模型，多芯片场景下 如何平衡整体业务负载到不同**device**上?
答：一般情况是将要计算的负载进行划分，尽量将**device**的算力和内存平衡一下，在用满算力的条件下将内存也尽量填满，然后如果模型的分类比较复杂可以考虑把流水线架在几个**device**上进行流转。

2、**aclrtProcessReport** 这个接口用法应该是**1**个**channel**绑一个，还是多个**channel**绑**1**个?
答：一个线程绑定一个。

Thank you.

昇腾开发者社区

http：//ascend.huawei.com

把数字世界带入每个人、每个家庭、
每个组织，构建万物互联的智能世界。

**Bring digital to every person， home，and
organization for a fully connected，intelligent world。**

**The information in this document may contain predictive statements including， without limitation， statements regarding the future financial and operating results， future product portfolio， new technology， etc。 There are a number of factors that
could cause actual results and developments to differ materially from those expressed or implied in the predictive statements。 Therefore， such information is provided for reference purpose only and constitutes neither an offer nor an acceptance。 Huawei may change the information at any time without notice。**

HUAWEI