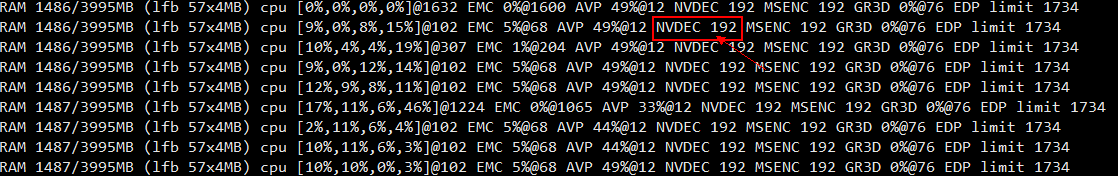
**基于NVIDIA TEGRA系列板卡的硬件解码及视频推流**

以NVIDIA TX1为例硬解码就是利用硬件芯片来解码的，TX1有单独的解码模块，NVDEC.

软解码是用软件程序来解码，比较占用CPU资源

查看cpu gpu 以及编解码模块的使用：

sudo ./tegrastats



**1.gstreamer概述**

Gstreamer是一个libraries和plugins的集合，用于帮助实现各种类型的多媒体应用程序，比如播放器，转码工具，多媒体服务器等。

利用Gstreamer编写多媒体应用程序，就是利用elements构建一个pipeline。element是一个对多媒体流进行处理的object，比如如下的处理：

\*\*\*读取文件。

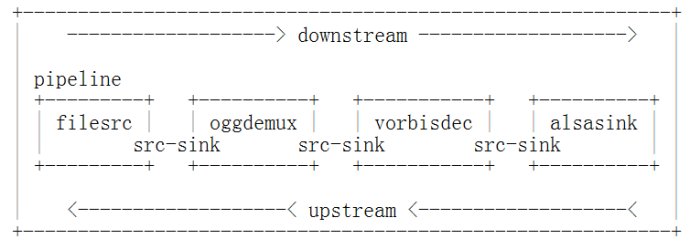
\*\*\*不同格式的编解码。

\*\*\*从硬件采集设备上采集数据。

\*\*\*在硬件设备上播放多媒体。

\*\*\*多个流的复用。

elements的输入叫做sink pads，输出叫做source pads。应用程序通过pad把element连接起来构成pipeline，如下图所示，其中顺着流的方向为downstream，相反方向是upstream。



应用程序会收到来自pipeline的消息和通知，比如EOS等。

**总体设计**  
  
Gstreamer的设计目标如下：

快速处理大规模数据。

对多线程处理的完全支持。

能处理各种格式的流媒体。

不同数据流的同步。

处理多种设备的能力。

基于Gstreamer的应用程序能够具备的处理能力依赖于系统中安装的不同种类功能的elements的数量。  
  
 Gstreamer核心不具备处理具体的media的功能，但是element处理media时需要具备的特性很多是由Gstreamer的核心提供的。  
  
**elements**  
  
 element是pipeline的最小组成部分。element提供了多个pads，或者为sink，或者为source。一个element有四种可能的状态，分别是NULL，READY，PAUSED，PLAYING。NULL和READY状态下，element不对数据做任何处理，PLAYING状态对数据进行处理，PAUSE状态介于两者之间，对数据进行preroll。应用程序通过函数调用控制pipeline在不同状态之间进行转换。  
  
element的状态变换不能跳过中间状态，比如不能从READY状态直接变换到PLAYING状态，必须经过中间的PAUSE状态。  
  
element的状态转换成PAUSE会激活element的pad。首先是source pad被激活，然后是sink pad。pad被激活后会调用activate函数，有一些pad会启动一个Task。  
  
PAUSE状态下，pipeline会进行数据的preroll，目的是为后续的PLAYING状态准备好数据，使得PLAYING启动的速度更快。一些element需接收到足够的数据才能完成向PAUSE状态的转变，sink pad只有在接收到第一个数据才能实现向PAUSE的状态转变。  
   
通常情况下，element的状态转变需要协调一致。  
  
可对element进行如下分类：

source，只提供数据源。

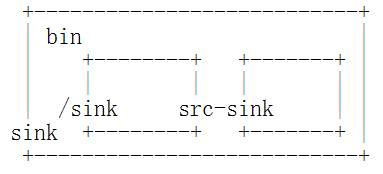
sink，比如播放设备。

transform

demuxer

muxer

**Bin**  
  
bin是由多个element构成的特殊的element，用图来说明：



**Pipeline**

pipeline是具备如下特性的特殊的bin：

选择并管理一个全局的时钟。

基于选定的时钟管理running\_time。running\_time用于同步，指的是pipeline在 PLAYING状态下花费的时间。

管理pipeline的延迟。

通过GstBus提供element与应用程序间的通讯方式。

管理elements的全局状态，比如EOS，Error等。

**Dataflow and buffers**  
  
Gstreamer支持两种类型的数据流，分别是push模式和pull模式。在push模式下，upstream的element通过调用downstream的sink pads的函数实现数据的传送。在pull模式下，downstream的element通过调用upstream的source pads的函数实现对数据的请求。  
  
push模式是常用的模式，pull模式一般用于demuxer或者低延迟的音频应用等。

在pads之间传送的数据封装在Buffer里，Buffer中有一个指向实际数据的指针以及一些metadata。metadata的内容包括：

* Timestamp
* Offset
* Duration
* media type
* 其它

在push模式下，element通过调用gst\_pad\_push（）函数把buffer传送给对应的pad。在pull模式下，element通过调用gst\_pad\_pull\_range（）函数把pull过来。  
  
element在push buffer之前需要确认对应的element具备处理buffer中的数据类型的能力。在传说红之前首先查询对应的element能够处理的格式的种类，并从中选择合适的格式，通过gst\_buffer\_set\_caps（）函数对buffer进行设置，然后才传送数据。  
  
收到一个buffer后，element要首先对buffer进行检查以确认是否能够处理。  
  
可以调用gst\_buffer\_new（）函数创建一个新的buffer，也可以调用gst\_pad\_alloc\_buffer（）函数申请一个可用的buffer。采用第二种方法接收数据的buffer可以设定接收其它类型的数据，这是通过对buffer的caps进行设定来实现的。  
  
选择媒体类型并对buffer进行设定的处理过程叫做caps negotianation。  
  
**Caps**  
  
Caps，也就是媒体类型，采用key/value对的列表来描述。key是一个字符串类型，value的类型可能是int/float/string类型的single/list/range。  
  
**Data flow and events**  
  
除了数据流，还有events流。与数据流不同，events的传送方向既有downstream的，也有upstream的。  
  
events用于传递EOS，flushing，seeking等消息。  
  
有的events必须和data flow一起进行serialized。serialized的events比如TAG，非serialized的events比如FLUSH。  
  
**Pipeline construction**  
  
gst\_pipeline\_create（）函数用于创建一个pipeline，gst\_bin\_add（）函数用于向pipeline中添加element，gst\_bin\_remove（）函数用于从pipeline中移除element。gst\_element\_get\_pad（）函数用于检索pipeline中的element。gst\_pad\_link（）函数用于把pads连接在一起。  
  
有的element会在数据流开始传送的时候创建新的pads，通过调用函数g\_signal\_connect（）函数，能在新的pads被创建的时候接收到消息。  
  
由于处理的数据互相不兼容，有的elements是不能被连接到一起的。gst\_pad\_get\_caps（）函数查询element能够处理的数据类型。  
  
**Pipeline clock**  
  
Pipeline的一个重要功能是为pipeline中的所有elements选择一个全局时钟。  
  
时钟的作用是提供一个每秒为GST\_SECOND的单调递增的时钟，单位是纳秒。element利用这个时钟时间来播放数据。  
  
在pipeline被设为PLAYING之前，pipeline查询每一个element是否能提供clock，并按照如下次序来选择clock：

* 应用程序选择了一个clock。
* 如果source element提供了clock。
* 其它任何提供了clock的element。
* 选择一个默认的系统clock。

也有特殊的情况，比如存在音频sink提供了clock，那么就选择其提供的clock。  
  
**Pipeline states**  
  
完成了pads的链接和signals的链接，就可以设定pipeline为PAUSED状态启动数据流的处理。当bin（这里指的是pipeline）进行状态转换的时候要转换所有的children的状态，转换的次序是从sink element开始到source element结束，这样做的目的是为了确保upstream element提供数据的时候，downstream element已经准备好。  
  
**Pipeline status**  
  
Pipeline会通过bus向应用程序通报发生的events。bus是由pipeline提供的一个object，可以通过gst\_pipeline\_get\_bus（）函数取得。  
  
bus分布到加入pipeline的每一个element。element利用bus来发布messages。有各种不同类型的messages，比如ERRORS，WARNINGS，EOS，STATE\_CHANGED等。  
  
pipeline以特殊的方式处理接收到的EOS message，只有当所有的sink element发送了EOS message的时候，pipeline才会把EOS发送给应用程序。  
  
也可以通过gst\_element\_query（）函数获取pipeline status，比如获取当前的位置或者播放的时间。  
 **Pipeline EOS**  
  
当source filter遇上了流结束，会沿着downstream的方向向下一个element发送一个EOS的event，这个event依次传送给每一个element，接收到EOS event的element不再接收数据。  
  
启动了线程的element发送了EOS event后就不再发送数据。  
  
EOS event最终会到达sink element。sink element会发送一个EOS消息，通告流结束。pipeline在接收到EOS消息以后，把消息发送给应用程序。只有在PLAYING状态下会把EOS的消息传送给应用程序。  
  
发送了EOS以后，pipeline保持PLAYING状态，等待应用程序把pipeline的状态置为PAUSE或者READY。应用程序也可以进行seek操作。

**2, Gstreamer解码**

1>调用Gstreamer解码多路rtsp

/\*

\*Author:mxj

\*/

#ifndef \_\_GSTREAMER\_CAMERA\_H\_\_

#define \_\_GSTREAMER\_CAMERA\_H\_\_

#include <gst/gst.h>

#include <string>

*struct* \_GstAppSink;//声明结构体和类

*class* QWaitCondition;

*class* QMutex;

/\*\*\* gstreamer CSI camera using nvcamerasrc (or optionally v4l2src)

\* @ingroup util

\*/

*class* gstCamera

{

public:

    // 创建camera类

    static gstCamera\* Create( *int* v4l2\_device=-1 ); // use onboard camera by default (>=0 for V4L2)

    static gstCamera\* Create( *uint32\_t* width, *uint32\_t* height, *int* v4l2\_device=-1 );

    // 析构函数

    ~gstCamera();

    // 开始和停止流

*bool* Open();

*void* Close();

    // 采集YUV(NV12格式)

*bool* Capture( *void*\*\* cpu, *void*\*\* cuda, *unsigned* *long* timeout=ULONG\_MAX );

    // 抓取YUV-NV12 CUDA image, 转换成 float4 RGBA (像素范围在 0-255)

    // 转换如果在CPU上进行，设置zeroCopy=true,默认只在CUDA上.

*bool* ConvertRGBA( *void*\* input, *void*\*\* output, *bool* zeroCopy=false );

    // 图像大小信息 inline(内联函数，适合简单的函数)

    inline *uint32\_t* GetWidth() const     { return mWidth; }

    inline *uint32\_t* GetHeight() const    { return mHeight; }

    inline *uint32\_t* GetPixelDepth() const { return mDepth; }

    inline *uint32\_t* GetSize() const      { return mSize; }

    // 默认图像大小，可以在create时改变

    static const *uint32\_t* DefaultWidth = 1280;

    static const *uint32\_t* DefaultHeight = 720;

private:

    static *void* onEOS(\_GstAppSink\* sink, *void*\* user\_data);

    static GstFlowReturn onPreroll(\_GstAppSink\* sink, *void*\* user\_data);//GstFlowReturn 传递流

    static GstFlowReturn onBuffer(\_GstAppSink\* sink, *void*\* user\_data);

    gstCamera();

*bool* init();

*bool* buildLaunchStr();

*void* checkMsgBus();

*void* checkBuffer();

    //GstBus

    \_GstBus\* mBus;//GstBus 异步同步消息

    \_GstAppSink\* mAppSink;

    \_GstElement\* mPipeline;

    std::string mLaunchStr="rtspsrc location=rtsp://admin:buaa123456@192.168.1.106:554/h264/ch1/main/av\_stream latency=0 ! queue ! rtph264depay ! h264parse ! queue ! omxh264dec ! appsink name=mysink";

*uint32\_t* mWidth;

*uint32\_t* mHeight;

*uint32\_t* mDepth;

*uint32\_t* mSize;

    static const *uint32\_t* NUM\_RINGBUFFERS = 16;//环形队列来解决数据阻塞问题

*void*\* mRingbufferCPU[NUM\_RINGBUFFERS];

*void*\* mRingbufferGPU[NUM\_RINGBUFFERS];

    QWaitCondition\* mWaitEvent;

    //mutex.lock() //锁住互斥量（mutex）。如果互斥量是解锁的，那么当前线程就立即占用并锁定它。否则，当前线程就会被阻塞，知道掌握这个互斥量的线程对它解锁为止。

    //mutex.unlock()//解锁

    //mutex.tryLock()//尝试解锁，如果该互斥量已经锁住，它就会立即返回

    QMutex\* mWaitMutex;

    QMutex\* mRingMutex;

*uint32\_t* mLatestRGBA;

*uint32\_t* mLatestRingbuffer;

*bool* mLatestRetrieved;

*void*\* mRGBA[NUM\_RINGBUFFERS];

*int* mV4L2Device;  // -1 for onboard, >=0 for V4L2 device

    inline *bool* onboardCamera() const       { return (mV4L2Device < 0); }

};

#endif

*bool* gstCamera::Capture( *void*\*\* cpu, *void*\*\* cuda, *unsigned* *long* timeout )

{

    /\*wait() 函数必须传入一个已上锁的 mutex 对象，在 wait() 执行过程中，

    mutex一直保持上锁状态，直到调用操作系统的wait\_block 在阻塞的一瞬间把 mutex 解锁

    （严格说来应该是原子操作，即系统能保证在真正执行阻塞等待指令时才解锁）。

    另一线程唤醒后，wait() 函数将在第一时间重新给 mutex 上锁（这种操作也是原子的）

    ，直到显示调用 mutex.unlock() 解锁。\*/

    mWaitMutex->lock();

const *bool* wait\_result = mWaitEvent->wait(mWaitMutex, timeout);

mWaitMutex->unlock();

    if( !wait\_result )

        return false;

    mRingMutex->lock();

    const *uint32\_t* latest = mLatestRingbuffer;

    const *bool* retrieved = mLatestRetrieved;

    mLatestRetrieved = true;

    mRingMutex->unlock();

    // skip if it was already retrieved

    if( retrieved )

        return false;

    if( cpu != NULL )

        \*cpu = mRingbufferCPU[latest];

    if( cuda != NULL )

        \*cuda = mRingbufferGPU[latest];

    return true;

}

#define release\_return { gst\_sample\_unref(gstSample); return; }

// checkBuffer

*void* gstCamera::checkBuffer()

{

*bool* write\_flags=true;//默认写数据

    if( !mAppSink )

        return;

    // block waiting for the buffer 函数被唤醒until A sample or EOS 可用 或者appsink 被设置成 ready/null state

    GstSample\* gstSample = gst\_app\_sink\_pull\_sample(mAppSink);

    if( !gstSample )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_app\_sink\_pull\_sample() returned NULL...\n");

        return;

    }

    //get buffer from gstSample

    GstBuffer\* gstBuffer = gst\_sample\_get\_buffer(gstSample);

    if( !gstBuffer )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_sample\_get\_buffer() returned NULL...\n");

        return;

    }

    // retrieve

    GstMapInfo map;

    if( !gst\_buffer\_map(gstBuffer, &map, GST\_MAP\_READ) )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_buffer\_map() failed...\n");

        return;

    }

    //gst\_util\_dump\_mem(map.data, map.size);

*void*\* gstData = map.data; //GST\_BUFFER\_DATA(gstBuffer);

    const *uint32\_t* gstSize = map.size; //GST\_BUFFER\_SIZE(gstBuffer);

    if( !gstData )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_buffer had NULL data pointer...\n");

        release\_return;

    }

    // 取出caps

    GstCaps\* gstCaps = gst\_sample\_get\_caps(gstSample);

    if( !gstCaps )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_buffer had NULL caps...\n");

        release\_return;

    }

    GstStructure\* gstCapsStruct = gst\_caps\_get\_structure(gstCaps, 0);

    if( !gstCapsStruct )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_caps had NULL structure...\n");

        release\_return;

    }

    // get width & height of the buffer

*int* width = 0;

*int* height = 0;

    if( !gst\_structure\_get\_int(gstCapsStruct, "width", &width) ||

        !gst\_structure\_get\_int(gstCapsStruct, "height", &height) )

    {

        printf(LOG\_GSTREAMER "gstreamer camera -- gst\_caps missing width/height...\n");

        release\_return;

    }

    if( width < 1 || height < 1 )

        release\_return;

    mWidth = width;

    mHeight = height;

    mDepth = (gstSize \* 8) / (width \* height);

    mSize = gstSize;

    //printf(LOG\_GSTREAMER "gstreamer camera recieved %ix%i frame (%u bytes, %u bpp)\n", width, height, gstSize, mDepth);

    // make sure ringbuffer is allocated

    if( !mRingbufferCPU[0] )

    {

        for( *uint32\_t* n=0; n < NUM\_RINGBUFFERS; n++ )

        {

            if( !cudaAllocMapped(&mRingbufferCPU[n], &mRingbufferGPU[n], gstSize) )

                printf(LOG\_CUDA "gstreamer camera -- failed to allocate ringbuffer %u (size=%u)\n", n, gstSize);

        }

        printf(LOG\_CUDA "gstreamer camera -- allocated %u ringbuffers, %u bytes each\n", NUM\_RINGBUFFERS, gstSize);

    }

    // copy to next ringbuffer

    const *uint32\_t* nextRingbuffer = (mLatestRingbuffer + 1) % NUM\_RINGBUFFERS;

    //printf(LOG\_GSTREAMER "gstreamer camera -- using ringbuffer #%u for next frame\n", nextRingbuffer);

    memcpy(mRingbufferCPU[nextRingbuffer], gstData, gstSize);

    // FILE \*fp=fopen("out.yuv","w+");

    // fwrite(gstData,gstSize,1,fp);

    // fclose(fp);

    //test h264 write

    //void \*writedata=map.data;

    // while(write\_flags==true)

    // {

    //  FILE \*fp=fopen("out.264","a+");

    //  fwrite(writedata,gstSize,1,fp);

    //  write\_flags=false;

    //  fclose(fp);

    // }

    gst\_buffer\_unmap(gstBuffer, &map);

    //gst\_buffer\_unref(gstBuffer);

    gst\_sample\_unref(gstSample);

    // update and signal sleeping threads

    mRingMutex->lock();

    mLatestRingbuffer = nextRingbuffer;

    mLatestRetrieved = false;

    mRingMutex->unlock();

    mWaitEvent->wakeAll();

}

可以在带显示的时候解码4路1080rtsp流.

不加显示可以做到6路1080p解码

2>opencv中使用Gstreamer解码海康rtsp摄像头

A.安装gstreamer依赖



B.重新编译安装opencv

cmake -D CMAKE\_BUILD\_TYPE=RELEASE -D CMAKE\_INSTALL\_PREFIX=/usr/local -D CUDA\_GENERATION=Kepler ..

C.确认opencv cmake后的提示中Gstreamer的五个选项都为on

D.编译opencv之后的使用方法：

VideoCapture(“rtspsrc location=\"rtsp://admin:admin666@192.168.1.106/h264/h264/main/av\_stream\" latency=10 ! rtph264depay ! h264parse ! omxh264dec ! videoconvert ! appsink sync=false”)

3>调用tegra\_multimedia解码

这是nvidia自带的编解码框架 目前用官方的解码1080p的h264可以做到解码速度达到150fps

接入rtsp的相机流

解决方法：ffmpeg/live555解析之后做个数据拷贝

陈昱霖目前在做rtsp解析成h264的部分

我在看nvidia官方的demo，了解清楚数据结构之后就可以对接解析好的h264视频流

4>视频推流

需求：将处理后的opencv数据进行推流到网页

解决方案：

A> 用tegra\_multimedia编码数据为h264, live555推流为rtsp

B> 直接用gstreamer推流（gstreamer自带rtsp的部分）

目前正在尝试用B方案解决