APPLICATION NOTE

How to Use OpenCV and CUDA in pylon Applications

Document Number: AW001763
Version: 01 Language: 000 (English)
Release Date: 02 December 2022

BASLER
the power of sight

Contacting Basler Support Worldwide

Europe, Middle East, Africa
Tel. +49 4102 463 515
support.europe@baslerweb.com

The Americas
Tel. +1 610 280 0171
support.usa@baslerweb.com

Asia-Pacific
Tel. +65 6367 1355
support.asia@baslerweb.com

Singapore
Tel. +65 6367 1355
support.asia@baslerweb.com

Taiwan
Tel. +88 3 558 3955
support.asia@baslerweb.com

China
Tel. +86 10 6295 2828
support.asia@baslerweb.com

Korea
Tel. +86 31 714 3114
support.asia@baslerweb.com

Japan
Tel. +81 3 6672 2333
support.asia@baslerweb.com

www.baslerweb.com/support

Table of Contents

1 Introduction

OpenCV is a highly optimized computer vision library with focus on real-time applications. It is open source and released under the BSD 3-Clause License (free for commercial use). The library is available for various programming languages, such as C++, Python, and Java, and various platforms, such as Linux, macOS, Windows, iOS, and Android.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface that allows software to use Nvidia graphics processing units (GPUs) for general purpose processing, an approach called general-purpose computing on GPUs (GPGPU).

For certain image (pre-)processing tasks, using the computer's CPU is not ideal, as the CPU processes tasks sequentially. A GPU, however, provides many processing cores, which can be used to work highly parallelized.

A good example is demosaicing a Bayer image. When using an even halfway decent demosaicing algorithm to generate good RGB images with low demosaicing artifacts, the CPU easily reaches its limits. This is particularly true for embedded processors with a relatively weak CPU. A GPU with its many kernels is much better suited for such a task.

This application note describes how you can integrate OpenCV functions into your pylon application and how you can offload high quality demosaicing to the GPU by using CUDA. It was tested on an Nvidia Jetson Nano board.

As many other embedded devices, the Jetson Nano uses shared memory for both the CPU and GPU. By default, the Basler pylon SDK grabs images into the memory domain of the CPU. However, by using the pylon buffer factory, you can override this behavior and let pylon grab images directly into the GPU memory without requiring any further memory copy operation.

2 Preparations

To prepare a Jetson Nano DevKit for running the code sample in chapter 3:

1. Create an SD card for your Jetson Nano DevKit as explained here: https://developer.nvidia.com/embedded/learn/get-started-jetson-nano-devkit
2. After booting the Jetson Nano DevKit, install OpenCV with OpenGL support.
   OpenGL support is needed to display the demosaiced images from the GPU memory without any intermediate memory copy operation. Unfortunately, the pre-built OpenCV Debian package comes without OpenGL support, so you need to build OpenCV from source.
   The following article describes how to do this: https://yunusmuhammad007.medium.com/build-and-install-opencv-4-5-3-on-jetson-nano-with-cuda-opencl-opengl-and-gstreamer-enable-6dc7141be272
3. Install the most recent Camera Enablement Package for Nvidia Jetson Boards from the Basler website: https://www.baslerweb.com/en/downloads/software-downloads/#type=embedded_software;language=all;version=all;os=linuxarm
4. Install a suitable code editor or IDE for C++ (e.g. Qt Creator or VisualStudio Code).

3 Code Sample

The following code sample demonstrates developing a buffer factory class that uses OpenCV (namespace “cv”) to grab images directly into the CUDA GPU memory.

The pylon SDK provides a good generic example of the buffer factory class: https://docs.baslerweb.com/pylonapi/cpp/sample_code#grab_usingbufferfactory.

To avoid any unnecessary copy operation from the user memory to the GPU memory, you can use the buffer factory class to tell pylon where to put the image data.

First, create a buffer factory by deriving it from the pylon **IBufferFactory** class. Then, override the following virtual member functions:

- **AllocateBuffer()**: Override this function to create the buffer and provide pylon with a pointer to the allocated buffer. The function is called from pylon after **CInstantCamera::StartGrabbing** for each buffer you want to put into the buffer queue.
  For example, if you have set **CInstantCamera::MaxNumBuffer** to 5, it will be called 5 times to create 5 buffers to be put into the buffer queue.
- **FreeBuffer()**: Override this function to free all allocated buffers after **CInstantCamera::StopGrabbing.**

```
// A user-provided buffer factory for grabbing into CUDA GPU memory.
class CudaBufferFactory : public IBufferFactory
{
public:
    CudaBufferFactory(int width, int height, int type, cv::cuda::HostMem::AllocType alloc_type):
          m_width(width),           // Width of image
          m_height(height),         // Height of image
          m_element_type(type),     // OpenCV pixel format of the raw Bayer image, e.g., CV_8UC1 (8 bit
per pixel, unsigned, one color channel)
          m_alloc_type(alloc_type), // CUDA allocation type, cuda::HostMem::SHARED -> shared GPU memory
          m_dummy(1,1,type)         // dummy OpenCV Mat used to check whether the m_element_type, width,
and height fit to the buffer size.
    {
    }

    virtual ~CudaBufferFactory()
    {
    }

    // Will be called when the Instant Camera object needs to allocate a buffer.
    // Returns the buffer and context data in the output parameters.
    // Warning: This method can be called by different threads.
    virtual void AllocateBuffer( size_t bufferSize, void** pCreatedBuffer, intptr_t& bufferContext)
    {
        try
        {
        if(m_dummy.elemSize() * m_width * m_height < bufferSize)
                {
                cout << "Buffer Factory configuration doesn't fit the requested buffer. No buffer
created." << endl;
                return;
        }

        // Here, the CUDA shared GPU memory is allocated:
                cv::cuda::HostMem* m = new cv::cuda::HostMem(m_height, m_width, m_element_type,
        m_alloc_type);
        if(!m->isContinuous())
                {
                cout << "HostMem is not continuous. This will damage the image." << endl;
        }
```

```
                // Create buffer context.
                 m->step = m_width;
        cv::Mat cpuMat = m->createMatHeader();
                 *pCreatedBuffer = cpuMat.data;
                bufferContext = (intptr_t)(void*)m;
        }
        catch (const std::exception&) {
            throw;
        }
    }


    // Frees a previously allocated buffer.
    // Warning: This method can be called by different threads.
    virtual void FreeBuffer( void* pCreatedBuffer, intptr_t bufferContext)
    {
        cv::cuda::HostMem* m = (cv::cuda::HostMem*)bufferContext;
        delete m;
    }


    // Destroys the buffer factory.
    // This will be used when you pass the ownership of the buffer factory instance to pylon
    // by defining Cleanup_Delete. pylon will call this function to destroy the instance
    // of the buffer factory. If you don't pass the ownership to pylon (Cleanup_None),
    // this method will be ignored.
    virtual void DestroyBufferFactory()
    {
        delete this;
    }


protected:

    cv::cuda::HostMem::AllocType m_alloc_type;
    int m_width;
    int m_height;
    int m_element_type;
    cv::Mat m_dummy;

}; // End of CudaBufferFactory class
```

In your pylon code, you can now create the buffer factory and tell pylon to use it before start grabbing images:

```
CudaBufferFactory* factory = new CudaBufferFactory(width, height, CV_8UC1, cuda::HostMem::SHARED);
camera.SetBufferFactory(factory, Cleanup_Delete);
// Start grabbing images
camera.StartGrabbing(GrabStrategy_LatestImages);
```

You can now use OpenCV methods that call highly optimized CUDA functions under the hood, performing demosaicing directly in the GPU without copying data back to the user memory:

```
// Creates a GUI window for displaying images with OpenGL support.
// It is important that you build OpenCV with OpenG support to directly blit the image date
// from the GPU to the screen, otherwise you'll need to create
// namedWindow("Zerocopy and Demosaicing in GPU", WINDOW_NORMAL)
// and an additional copy operation from GPU to the user memory
// before displaying the image with imshow.
namedWindow(WINDOW_TITLE, WINDOW_OPENGL);

// OpenCV CUDA matrices for the raw Bayer and for the converted color BGR image.
cuda::GpuMat mat_color;
cuda::GpuMat mat_raw;
```

```
while ( camera.IsGrabbing())
{
        // Wait for an image and then retrieve it. A timeout of 5000 ms is used.
        camera.RetrieveResult(5000, ptrGrabResult, TimeoutHandling_ThrowException);

        // Image grabbed successfully?
        if (ptrGrabResult->GrabSucceeded())
        {
                // Image has been grabbed directly into the GPU memory (provided by the buffer factory).
                cuda::HostMem* cmem = (cuda::HostMem*)ptrGrabResult->GetBufferContext();
                mat_raw = cmem->createGpuMatHeader();
                cuda::demosaicing (mat_raw, mat_color, COLOR_BayerGB2BGR);
        }
        else
        {
                // Error handling …
        }

        imshow("Zerocopy and Demosaicing in GPU", mat_color);

        // More code …
}
```

The code sample is available as a complete project. You can use #define options to build the project with or without OpenGL support and with or without CUDA support. You'll notice significant differences in terms of CPU load. Enjoy!

Revision History

Document Number	Date	Changes
AW00176301000	02 December 2022	Initial release version of this document.