GPU-Accelerated Vision for Robots with OpenCV and CUDA

OpenCV is an open source computer vision and machine learning library for C/C++/Python available for Windows, Linux, macOS and Android platforms. It contains low-level image processing functions as well as high-level algorithms such as object identification, face recognition, action classification in videos, etc. OpenCV has become very popular with more than 47,000 people in their user community and 18 million downloads (https://opencv.org/about/). Under a BSD license, it can be used in academic and commercial applications.

Significant part of computer vision is image processing, with massive parallel computations. Modern GPUs (Graphical Processing Units) are highly parallel multi-core systems, powerful enough for performing general purpose computations in large blocks of data. So it is challenging yet potentially very rewarding to accelerate OpenCV on graphics processors.

CUDA (Computing Unified Device Architecture) is a parallel computing architecture created by Nvidia that makes it possible to use the many computing cores in a GPU to perform general-purpose mathematical calculations [1]. However, it only works on Nvidia cards.

OpenCV and CUDA have been available since more than ten years [2], and their use has increased significantly; however, their combined application is not so widespread. Considering that a GPU/CUDA module for OpenCV is available since 2010, the number of works published in IEEE Xplore using both libraries is relatively small and grows very slowly: Figure 1 depicts the number of references in IEEE Xplore citing CUDA, OpenCV, or both (in 2018 only 7 references are found, compared with 240 for CUDA and 180 for OpenCV).

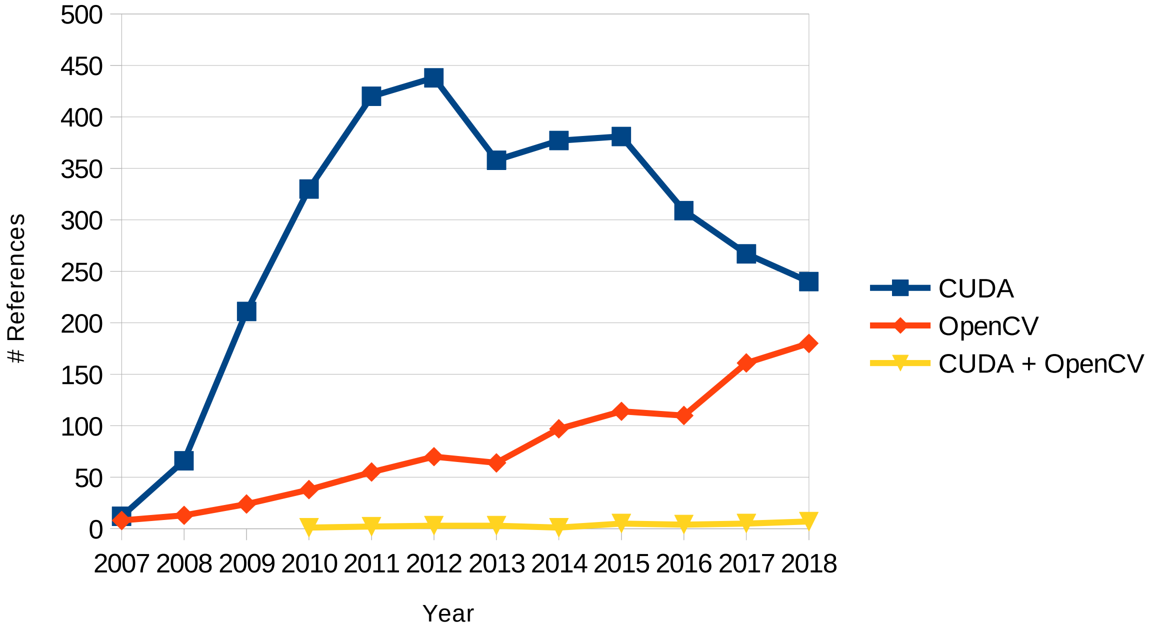


Figure 1: Number of references in IEEE Xplore for CUDA and OpenCV.

The aim of this paper is to describe how the CUDA module for OpenCV works, with some examples of well-known vision problems documented with source code, in order to encourage more robotics researchers to migrate their applications towards GPU computation.

The usefulness of CUDA in robotics and vision has been successfully demonstrated with significant speed-ups in many applications [3-6]. However, it introduces an overhead due to the need of transferring data between the CPU and GPU spaces, because most GPU processors work in a dedicated memory, independent from the system memory of the CPU. Consequently, image data needs to be moved back and forth between the different types of memory for the processing in the GPU. The processing flow consists of the following steps:

1. Upload data from main memory to GPU memory
2. Initiate the GPU computing kernel
3. Parallel computation in the GPU's cores
4. Download the resulting data from GPU memory to main memory

In the following, we will assume that the reader is familiar with OpenCV and C++ programming (for novices, an introduction is provided in [7]). Unless otherwise stated, the code snippets are based on OpenCV 3.4.0, but they can be easily adapted to earlier (2.4) or later (4.x) versions.

In the OpenCV library, all the classes and functions are defined in the name space `cv`. The main object is the class `cv::Mat`, which is essentially a matrix holding pixel values of an image. GPU modules in OpenCV define a class `cv::cuda::GpuMat` which is a container for an image data kept in GPU memory, with a very similar interface to its CPU counterpart.

Let's see a quick example with a color image, which is converted into gray, and binarized with a fixed threshold. In the CPU version, the source image `src` is first converted to an intermediate gray image `src_gray`, which is then thresholded into the resulting image `dst`. We need to define the variables (line 1) and call the OpenCV functions `cv::cvtColor` and `cv::threshold` (lines 2-3) for executing the task:

```
1 cv::Mat src, src_gray, dst;
2 cv::cvtColor( src, src_gray, cv::COLOR_BGR2GRAY );
3 cv::threshold( src_gray, dst, 128, 255, cv::THRESH_BINARY );
```

This processing flow is depicted in Figure 2: all the data is stored in the CPU memory, and all the operations are performed by the CPU.

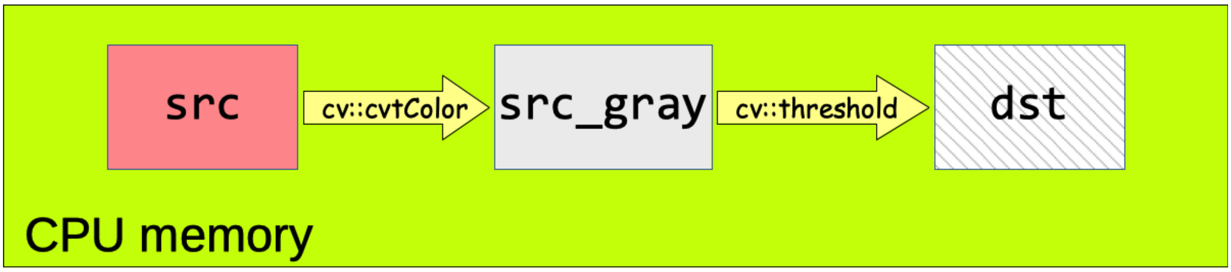


Figure 2: Image processing flow with CPU only.

In the GPU version, in addition to the variables for the initial and destination images (line 1), we need some new variables for processing the data in the GPU memory (line 2); the intermediate image `src_gray` is also stored in the GPU memory for minimizing data transfers:

```
1 cv::Mat src, dst;
```

```
2 cv::cuda::Mat gpu_src, gpu_dst, src_gray;
3 gpu_src.upload( src );
4 cv::cuda::cvtColor( gpu_src, src_gray, cv::COLOR_BGR2GRAY );
5 cv::cuda::threshold( src_gray, gpu_dst, 128, 255, cv::THRESH_BINARY );
6 gpu_dst.download( dst );
```

The processing task is performed by the equivalent functions of the OpenCV CUDA module `cv::cuda::cvtColor` and `cv::cuda::threshold`. First, the image is transferred from CPU to GPU memory (line 3); then, the processing steps are executed (lines 4-5), and finally, the resulting image is transferred from GPU back to CPU memory (line 6).

The processing flow is depicted in Figure 3, where the CPU and GPU memory spaces and the different processing steps are represented.

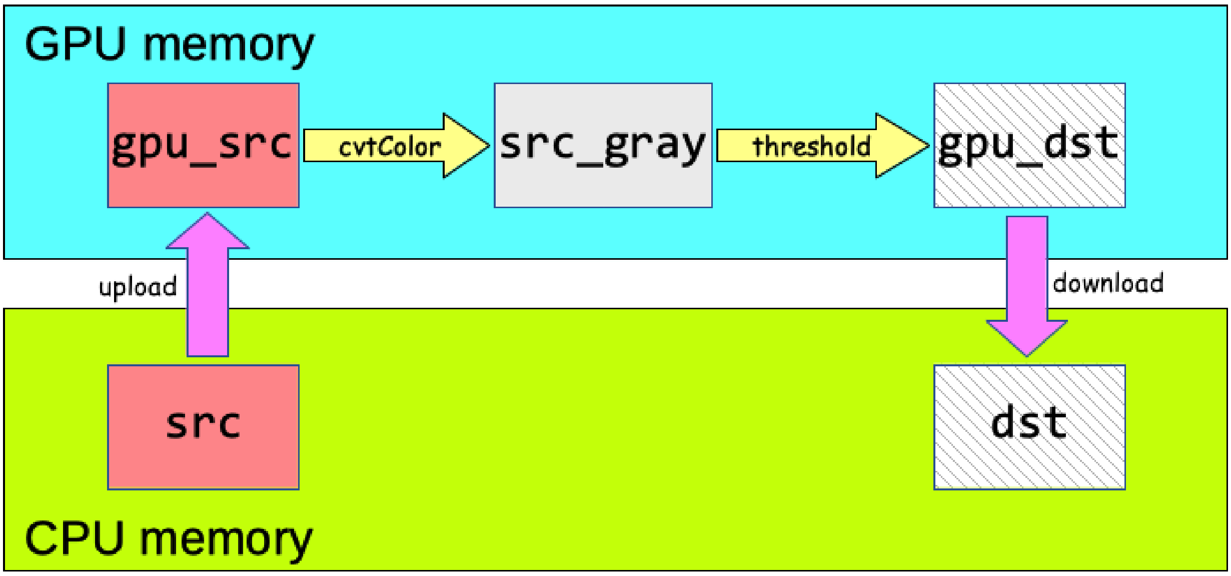


Figure 3: Image processing flow with CPU and GPU.

There is an inherent overhead in the GPU processing flow due to the transfer of the images between the CPU and GPU memories. Such overhead can be minimized if all the processing operations are performed in the GPU, and only the initial and final images are transferred:

$$T_{overhead} = T_{upload} + T_{download}$$

Let's define T_{CPU} and T_{GPU} as the computation times of the image processing operations (`cvtColor`, `threshold`) at the CPU and GPU respectively. A speed gain will be obtained if and only if:

$$T_{CPU} > T_{overhead} + T_{GPU}$$

Those computation times depend mainly on two factors:

- Hardware technology of the respective boards: OpenCV is highly optimized for CPUs with multiple cores and vector instructions.
- Degree of parallelization of the processing algorithms: some vision operations may benefit more than others from the use of multiple cores in the GPU.

In the following, we elaborate four examples of image processing applications (edge detection, feature extraction, optical flow, and object detection with deep neural networks) that use OpenCV with CPU and GPU in different hardware configurations.

The first example is a simple edge detection application with the well-known Canny algorithm [8]. The CPU version of the application is:

```
1 cv::Mat src, dst;
2 const int lowThreshold = 20;
3 const int ratio = 3;
4 const int kernel_size = 3;
5 cv::Mat src_gray, blurred, edges;

6 cv::cvtColor( src, src_gray, cv::COLOR_BGR2GRAY );
7 cv::blur( src_gray, blurred, cv::Size(3,3) );
8 cv::Canny( blurred, edges, lowThreshold, lowThreshold*ratio, kernel_size );
9 src.copyTo( dst, edges );
```

Besides the initial and final images defined in line 1, three more variables are created in line 5 for storing the intermediate images. The algorithm parameters are defined in lines 2-4, and the processing steps are executed in lines 6-8: converting to gray, blurring, and computing the edges. Finally, the edges are used as a pixel mask for copying the original image to the destination image in line 9.

The CUDA version is very similar, yet there some changes in the API of the processing functions:

```
 1 cv::Mat src, dst;
 2 const int lowThreshold = 20;
 3 const int ratio = 3;
 4 const int kernel_size = 3;
 5 cv::cuda::GpuMat gpu_src, gpu_dst;
 6 cv::cuda::GpuMat src_gray, blurred, edges;

 7 gpu_src.upload( src );
 8 cv::Ptr<cv::cuda::Filter> blur =
     cv::cuda::createBoxFilter( CV_8UC1, CV_8UC1, cv::Size(3,3) );
 9 cv::Ptr<cv::cuda::CannyEdgeDetector> canny =
     cv::cuda::createCannyEdgeDetector( lowThreshold,
     lowThreshold*ratio, kernel_size );

10 cv::cuda::cvtColor( gpu_src, src_gray, cv::COLOR_BGR2GRAY );
11 blur->apply( src_gray, blurred );
12 canny->detect( blurred, edges );
13 gpu_src.copyTo( gpu_dst, edges );
14 gpu_dst.download( dst );
```

Now we need to define the original and destinations images both as `Mat` and `GpuMat` variables (lines 1 and 5). The parameters are defined in the same ways as in the previous version (lines 2-4), and the intermediate images are defined as `GpuMat` (line 6).

The original image is uploaded to the GPU memory in line 7. Then, two new objects have to be defined for applying the blur filter and the Canny detector respectively in lines 8 and 9.

Image processing is executed in lines 10-12, and the original image is masked with the detected edges and copied to the destination (line 13). Finally, in line 14 the result is downloaded to the CPU memory. Figure 4 depicts the output for a frame of a video recorded during a car navigation task.

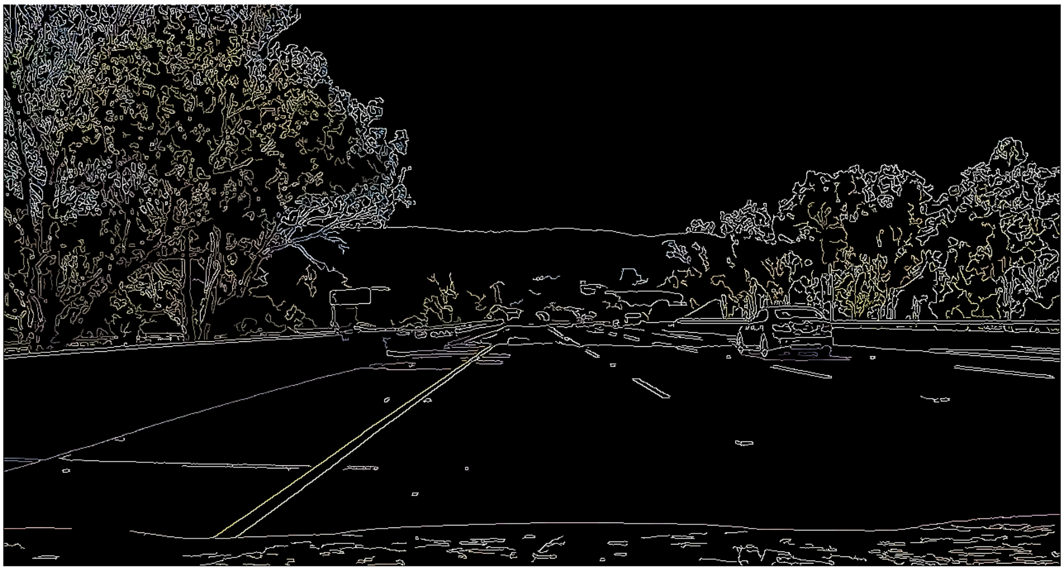


Figure 4: Output image of the edge detection example.

For measuring the average computing times of the algorithm, we have processed the frames of a benchmarking video on three different hardware configurations of CPU and GPU:

- **Desktop PC**: with a CPU Intel Core i7-6700 at 3.4 GHz, and a GPU GeForce GTX 1080.
- **Laptop PC**: with a CPU Intel Core i7-8550U at 3.3 GHz, and a GPU GeForce GTX 1050.
- **Embedded PC**: NVIDIA Jetson Nano, with an ARM-A57 processor, and an integrated GPU.

The video consists of a 50-second footage from a car in a highway available at Udacity's Advanced Line Finding Project (https://github.com/udacity/CarND-Advanced-Lane-Lines), recorded at 25 Hz with a resolution of 1280 x 720 RGB 24-bit pixels. The main specifications of the GPUs for the three systems are presented in Table 1. The desktop PC features the most powerful CPU, both in terms of processing cores and transfer speed, but it also requires more energy power, compared to the laptop and embedded PCs, which are more adequate for mounting on a small robotic platform.

Table 1: Technical specifications of the systems used in the experiments.

GPU features	Desktop PC	Laptop PC	Embedded PC
CUDA Cores	2560	640	128
Memory	8 GB	4 GB	4 GB
Memory Interface	GDDR5	GDDR5	LPDDR4
Memory Interface Width	256-bit	128-bit	64-bit
Memory Bandwidth	320 GB/sec	112 GB/sec	25.6 GB/sec
Power consumption	180 W	40-50 W	10 W

The source code with instructions for compilation and execution is publicly available in https://github.com/RobInLabUJI/opencv-cuda. For the sake of reproducibility, we use docker (https://www.docker.com), a Linux container technology that offers some advantages for an easy replication of code: encapsulation, isolation, portability, and control. In addition, containers have less overhead than virtual machines, and they can access the GPU transparently (usually the impact should be in the order of less than 1% and hardly noticeable.) As a downside, the GPU-enabled version of docker (nvidia-docker) does not support Windows nor macOS yet.

The code can also be compiled and executed natively in a Linux computer (as long as all the requirements are previously installed, basically OpenCV and CUDA) with the typical building commands:

```
mkdir -p build
cd build
cmake ..
make
```

The results are shown in Table 2: they measure the mean and standard deviation of the execution time for 1200 frames in the video (the initial 60 frames are skipped for avoiding initialization delays). The execution time is measured starting from the first call to processing functions, until the final result is returned; the result is averaged with a moving window of 30 frames. For the GPU cases, the measured time includes the uploading of the initial image to the GPU memory, and the downloading of the result image back to CPU memory. Visual information (edges, ORB keypoints, optical flow) is included in the measured code for clarity and debugging purposes, though in a real setup it could be removed in order to increase the throughput.

Table 2: Computation times (in milliseconds) for the edge detection algorithm (lower is better).

	Desktop PC	Laptop PC	Embedded PC
CPU	**3.514 ± 0.272**	**4.562 ± 0.331**	**12.985 ± 1.197**
GPU	4.422 ± 0.150	7.469 ± 0.106	54.064 ± 1.993

It is worth noting that for this application the CPUs are faster than the GPUs in all the three systems: edge detection is a relatively simple computation, and the execution time is small in comparison with the overhead of transferring the images into the GPU memory.

An example of the benchmarking code for measuring the execution time is presented below:

```
double ticks = (double)cv::getTickCount();
if (use_gpu) {
    gpu_processing(frame, dst);
} else {
    cpu_processing(frame, dst);
}
ticks = ((double)cv::getTickCount() - ticks)/cv::getTickFrequency()*1000;
```

The OpenCV functions `cv::getTickCount()` and `cv::getTickFrequency` are used for getting the number of ticks before and after the processing work, translated into seconds. A boolean variable indicates whether to use the CPU or GPU; the value of this variable can be toggled through a keyboard press.

In a second example, ORB features are detected and extracted from the image. Such features are very important in robotics applications, e.g. for visual SLAM [9]. The source code for the CPU version is:

```
cv::Mat src, dst;
cv::Mat src_gray, descriptors;
std::vector<cv::KeyPoint> keypoints;

cv::Ptr<cv::ORB> detector = cv::ORB::create();
```

```
5 cv::cvtColor( src, src_gray, cv::COLOR_BGR2GRAY );
6 detector->detect( src_gray, keypoints );
7 detector->compute( src_gray, keypoints, descriptors );
8 cv::drawKeypoints( src, keypoints, dst,
    cv::Scalar::all(-1), cv::DrawMatchesFlags::DEFAULT );
```

Firstly, we define the necessary variables for storing the original and final images, the intermediate gray image, and the structures for storing the keypoints and descriptors of the ORB features (lines 1-4).

Secondly, the feature detector is initialized with default parameters in line 4.

Finally, the processing steps are performed in lines 5-7: the original color image is converted into a gray image, the keypoints are detected, and their descriptors are computed. In line 8 the keypoints are drawn into the destination image for visualization.

The CUDA version of this example is straightforward:

```
 1 cv::Mat src, dst;
 2 cv::cuda::GpuMat gpu_src, gpu_dst;
 3 cv::cuda::GpuMat src_gray, descriptors;
 4 std::vector<cv::KeyPoint> keypoints;

 5 gpu_src.upload(src);
 6 cv::Ptr<cv::cuda::ORB> detector = cv::cuda::ORB::create();

 7 cv::cuda::cvtColor( gpu_src, src_gray, cv::COLOR_BGR2GRAY );
 8 detector->detect(  src_gray, keypoints );
 9 detector->compute( src_gray, keypoints, descriptors );
10 cv::drawKeypoints( src, keypoints, dst,
    cv::Scalar::all(-1), cv::DrawMatchesFlags::DEFAULT );
```

As in the previous example, we need to define two `GpuMat` variables for the original and destination images (line 2). The intermediate image is also stored in GPU memory, as well as the descriptors (line 3), but the keypoints are stored in CPU memory (line 4). After uploading the image in line 5, the feature detector is created and the processing steps are executed.

One should note that the processing code is basically similar to the previous version: lines 4-7 of the CPU code and lines 6-9 of the GPU code only differ in the use of the namespace `cv::cuda` instead of `cv` for the class `ORB` (line 4/6) and the functions `ORB::create` and `cvtColor` (lines 4/6 and 5/7).

Finally, drawing the keypoints is done in exactly the same way (line 10 of the GPU code is the same as line 8 of the CPU code). The output of the ORB detector is shown in Figure 5.

For debugging purposes, the code examples include visualization, and the corresponding function calls have been included in the benchmarking. Since the visualization process is using the same function call in both CPU and GPU versions, it should not affect the difference in the performance between them.

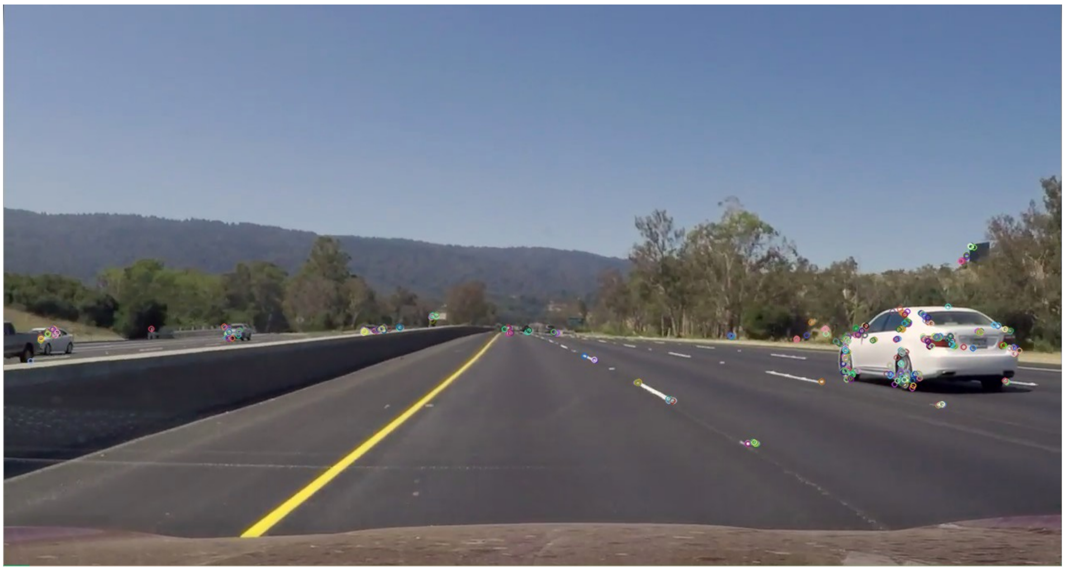


Figure 5: Output image of the ORB feature detector example.

The results are shown in Table 3. In this case the GPUs are faster than the CPUs, due to the increased computational workload demanded by the ORB algorithm.

For simplicity, this example has not computed the matching of ORB features, but it is possible to use either the CPU or the GPU for that purpose with the classes `cv::DescriptorMatcher` and `cv::cuda::DescriptorMatcher`, respectively.

Table 3: Computation times (in milliseconds) for the ORB feature extraction algorithm (lower is better).

	Desktop PC	Laptop PC	Embedded PC
CPU	15.323 ± 0.788	19.281 ± 0.883	101.586 ± 4.112
GPU	**10.777 ± 1.246**	**11.588 ± 0.397**	**66.697 ± 2.936**

In the third example, we compute the dense optical flow with the Farneback algorithm [10]. The source code for the CPU version is:

```
1 cv::Mat src, dst;
2 cv::Mat prev, cv::Mat next;

3 cv::Mat flow(prev.size(), CV_32FC2);

4 cv::cvtColor(src, next, cv::COLOR_BGR2GRAY);
6 cv::calcOpticalFlowFarneback(prev, next, flow, 0.5, 3, 15, 3, 5, 1.2, 0);
```

Since optical flow is computed with the difference between the current and previous frames, we need to define some more variables in line 2 for storing the frames. We also define a matrix of float numbers `flow` for the result (line 3 – `CV_32FC2` means a 2-channel [complex] floating-point array): this flow matrix contains the gradient of the movement between 2 frames; for each pixel location in the original frame, the channels contain `dx` and `dy`, so that `prev_x` + `dx` = `next_x`, and `prev_y` + `dy` = `next_y`.

The computation steps are quite simple: the color image is converted into a gray image (line 4), and the optical flow algorithm is executed (line 5). For the sake of simplicity, we have omitted additional instructions for displaying the result, and storing the frames.

The GPU version is not very different:

```
1 cv::Mat src, dst, flow;
2 cv::cuda::GpuMat gpu_src, gpu_flow;
3 cv::cuda::GpuMat prev, next;

4 gpu_src.upload(src);

5 cv::Ptr<cv::cuda::FarnebackOpticalFlow> fof =
    cv::cuda::FarnebackOpticalFlow::create();

6 cv::cuda::cvtColor( gpu_src, next, cv::COLOR_BGR2GRAY );
7 fof->calc( prev, next, gpu_flow );
8 gpu_flow.download( flow );
```

Besides defining all the intermediate matrices in GPU memory (lines 2-3), the main difference is in the interface to the optical flow algorithm. In this version, the algorithm object is first defined in line 5, then applied to the frames in line 7. Finally, the result is downloaded to CPU memory for visualization.

The output of the optical flow algorithm is displayed in Figure 6. The hue of each pixel block represents the orientation of the optical flow vector at that point, and the intensity is proportional to the magnitude of the flow.

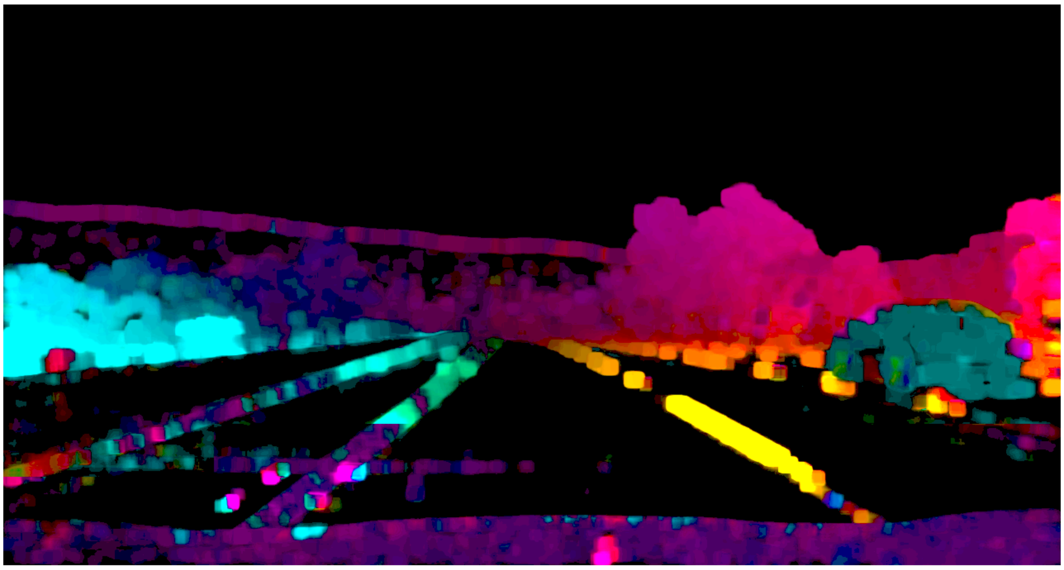


Figure 6: Output image of the dense optical flow algorithm; the hue represents the flow angle, and the intensity is proportional to the flow magnitude.

The results are shown in Table 4. As in the previous example, the execution times for the GPUs are lower than for the CPUs, since the computation of dense optical flow is a demanding operation.

Table 4: computation times (in milliseconds) for the dense optical flow algorithm (lower is better).

	Desktop PC	Laptop PC	Embedded PC
CPU	196.382 ± 0.889	228.838 ± 6.736	983.943 ± 12.776
GPU	**33.970 ± 0.231**	**78.561 ± 0.498**	**686.960 ± 9.729**

Finally, we test the Deep Neural Networks (DNN) module for OpenCV. Since version 3.1 there is a DNN module in the library that implements forward pass (inferencing) with networks pre-trained using some popular deep learning frameworks such as Caffe [11] or TensorFlow [12]. A backend for

CUDA was added in OpenCV 4.2.0. In this example we use the YOLO v3 network [13], a state-of-the-art, real-time object detection system.

While the details of the OpenCV DNN module are out of the scope of this paper, its design is based on a unique interface that runs on different backends and computation devices (CPU, OpenCL, CUDA). Consequently, the source code is exactly the same, no matter if the CPU or GPU is used, except for the parameters that select the appropriate backend and computation target. The values for using the CPU are:

```
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV);
net.setPreferableTarget (cv.dnn.DNN_TARGET_CPU);
```

And the GPU can be selected with:

```
net.setPreferableBackend(cv.dnn.DNN_BACKEND_CUDA);
net.setPreferableTarget (cv.dnn.DNN_TARGET_CUDA);
```

A typical output image from the DNN module is shown in Figure 7, where a number of cars are correctly identified in the input image. The frame rate for CPU and GPU versions running on the three types of computers used in the tests is shown in Table 5.

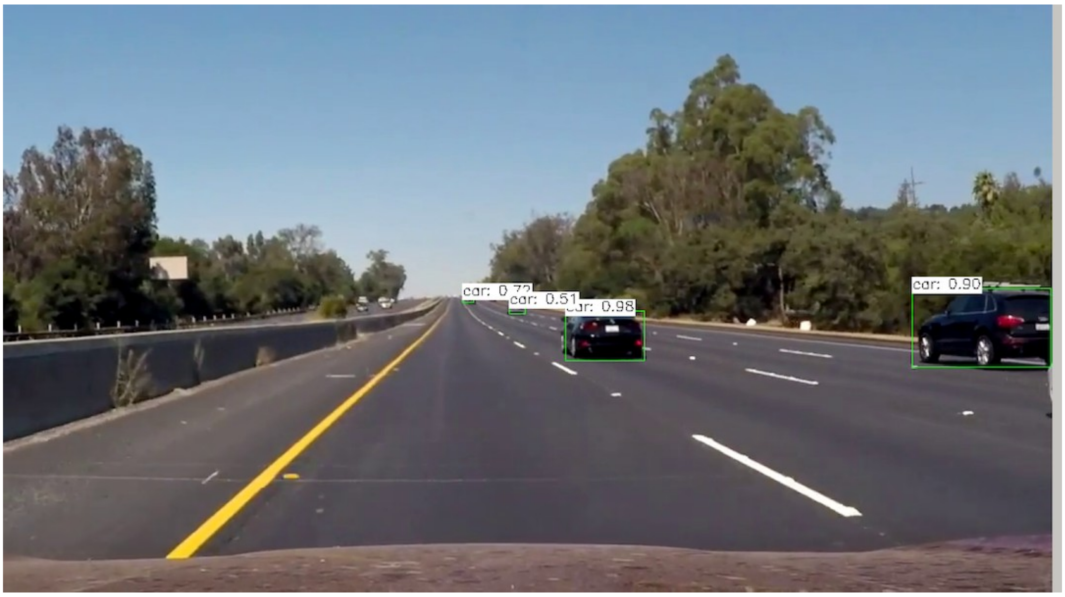
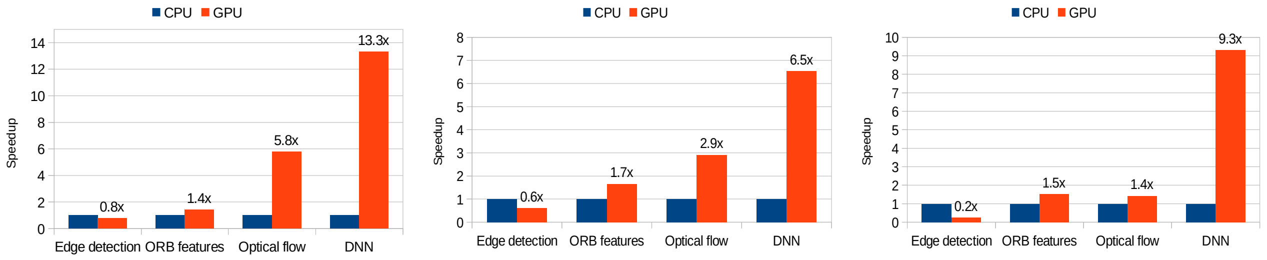


Figure 7: Output image of the DNN module with YOLO v3.

Table 5: frame rates (in Hz) for the YOLO v3 network in the DNN module (higher is better).

	Desktop PC	Laptop PC	Embedded PC
CPU	3.51	2.63	0.23
GPU	**46.6**	**17.2**	**2.14**

Besides absolute timings, it is illustrative to calculate the speed-up of the GPU with respect to the CPU, i.e. how much the GPU is faster than the CPU for a given application. The speed-up results are shown in Figure 8, displaying the values for each application (edge detection, ORB features, optical flow, deep neural network) on each of the platforms (desktop, laptop, and embedded PC).



a) Desktop PC
b) Laptop PC
c) Embedded PC

Figure 8: Performance comparisons of CPU versus GPU.

The overhead penalty can be noticed for the edge detection application on every platform. On the other hand, the speed-up is higher in the other applications, sky-rocketing in the last example (DNN for object recognition). This result is not surprising, since CUDA is used intensively by the deep learning community. But the benefits of using the GPU with other OpenCV functions cannot be overseen, obtaining speed-ups of 580% and 290% for the computation of the optical flow in the desktop and laptop PCs, respectively.

OpenCV is a widely-used library in robotics projects, and consequently there is a precompiled module for ROS [14] (http://wiki.ros.org/opencv3), which unfortunately does not include CUDA support. However, GPU acceleration can still be used by replacing the standard module with a CUDA-enabled version of the OpenCV library. This can be done by installing the library and setting the appropriate path in the file `CMakeLists.txt` of any ROS module using OpenCV:

```
find_package(OpenCV REQUIRED
  PATHS /usr/local
  NO_DEFAULT_PATH
)
```

In addition, all other ROS packages which are dependent upon OpenCV (`cv_bridge`, `image_pipeline`, `image_transport`, etc.) must be rebuild, i.e. their source code must be downloaded into a ROS workspace and compiled with `catkin_make`. A complete example of a simple subscriber is presented in the “ros” branch of the source code repository of this paper at https://github.com/RobInLabUJI/opencv-cuda/tree/ros.

Converting a ROS topic image to a CUDA image is straightforward; the topic message is converted to an OpenCV image, and this image is uploaded to the GPU:

```
cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
gpuInImage.upload(cv_ptr->image);
```

Once the image is uploaded, the processing can be done as usual, and the result can be downloaded and converted into a ROS message.

CUDA for OpenCV is an easy solution for accelerating vision applications in robotics, for systems equipped with a CUDA-enabled GPU. The migration of the code from CPU-based to GPU-based is simple and relatively straightforward, even trivial in some cases. The speed-up that can be achieved is system- and problem-dependent: for simple vision algorithms, modern CPUs can be faster; for complex problems involving a sequence of operations on the image, the parallelization in the GPU leads to better performance; and for deep learning applications, the improvement is significant.

We have provided some examples with well-known algorithms that are widely used by the robotics community, with the aim of encouraging the researchers to improve the throughput of their systems by squeezing all the computing power out of their hardware. CUDA and other computing frameworks (DirectCompute [15], OpenCL [16]) have become programming standards for parallel computing, and their inclusion in popular libraries like OpenCV is an opportunity for developers to benefit from parallelization without a significant investment in learning some specific parallel programming techniques.

An advantage of an open framework such as OpenCL over CUDA is that it is supported by both AMD and Nvidia cards. The interested reader can refer to [17] for details about using OpenCL in OpenCV.

References

[1] D. Luebke, «CUDA: Scalable parallel programming for high-performance scientific computing,» in 2008 5th IEEE International Symposium on Biomedical Imaging: from Nano to Macro, pp. 836-838, DOI: 10.1109/ISBI.2008.4541126.

[2] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov «Real-Time Computer Vision with OpenCV,» *ACM Queue* vol. 55, no. 6, pp. 61-69, June 2012, DOI:10.1145/2184319.2184337.

[3] P. Michel, J. Chestnutt, S. Kagami, K. Nishiwaki, J. Kuffner, and T. Kanade, «GPU-accelerated real-time 3D tracking for humanoid locomotion and stair climbing,» in 2007 *IEEE/RSJ International Conference on Intelligent Robots and Systems,* pp. 463-469, DOI: 10.1109/IROS.2007.4399104.

[4] T. Xu, T. Pototschnig, K. Kuhnlenz and M. Buss, «A high-speed multi-GPU implementation of bottom-up attention using CUDA," in 2009 *IEEE International Conference on Robotics and Automation*, pp. 41-47, DOI: 10.1109/ROBOT.2009.5152357.

[5] J. Kim, E. Park, X. Cui, H. Kim and W. A. Gruver, «A fast feature extraction in object recognition using parallel processing on CPU and GPU,» in 2009 *IEEE International Conference on Systems, Man and Cybernetics*, pp. 3842-3847, DOI: 10.1109/ICSMC.2009.5346612.

[6] N. Dalmedico, M. A. S. Teixeira, H. S. Barbosa, A. S. de Oliveira, L. V. R. de Arruda, and F. Neves Jr, «GPU and ROS: the Use of General Parallel Processing Architecture for Robot Perception,» in *Robot Operating System (ROS),* pp. 407-448, Springer, Cham, 2018.

[7] I. Culjak, D. Abram, T. Pribanic, H. Dzapo and M. Cifrek, «A brief introduction to OpenCV,» in 2012 *Proceedings of the 35th International Convention MIPRO,* pp. 1725-1730.

[8] J. Canny, «A Computational Approach To Edge Detection, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 8, no. 6. pp. 679-698, 1986.

[9] R. Mur-Artal and J. D. Tardós, «ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras,» in *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255-1262, Oct. 2017, DOI: 10.1109/TRO.2017.2705103.

[10] G. Farneback, «Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field,» in *2011 IEEE International Conference on Computer Vision,* pp. 171-177, DOI: 10.1109/ICCV.2001.937514.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama and T. Darrell, «Caffe: Convolutional architecture for fast feature embedding,» in *Proceedings of the*

22nd ACM international conference on Multimedia, pp. 675-678, 2014, DOI: 0.1145/2647868.2654889.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard and M. Kudlur, «Tensorflow: A system for large-scale machine learning,» in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 265-283, 2016.

[13] J. Redmon and A. Farhadi, «Yolov3: An incremental improvement,» *arXiv preprint arXiv*:1804.02767.

[14] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. «ROS: an open-source Robot Operating System.» In *ICRA workshop on open source software*, vol. 3, no. 3.2, p. 5. 2009.

[15] T. Ni, «Direct Compute: Bring GPU computing to the mainstream,» in *GPU Technology Conference*, p. 23, 2009.

[16] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, «From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming,» *Parallel Computing*, 38(8), 391-407, 2012.

[17] H. Gasparakis, «Heterogeneous compute in computer vision: OpenCL in OpenCV,» *Visual Information Processing and Communication V*. Vol. 9029. International Society for Optics and Photonics, 2014.