acmqueue

Realtime Computer Vision with OpenCV

Mobile computer-vision technology will soon become as ubiquitous as touch interfaces.

Kari Pulli, NVIDIA Research
Anatoly Baksheev, Itseez
Kirill Kornyakov, Itseez
Victor Eruhimov, Itseez

Computer vision is a rapidly growing field devoted to analyzing, modifying, and high-level understanding of images. Its objective is to determine what is happening in front of a camera and use that understanding to control a computer or robotic system, or to provide people with new images that are more informative or esthetically pleasing than the original camera images. Application areas for computer-vision technology include video surveillance, biometrics, automotive, photography, movie production, Web search, medicine, augmented reality gaming, new user interfaces, and many more.

Modern cameras are able to focus automatically on people's faces and trigger the shutter when they smile. Optical text-recognition systems help transform scanned documents into text that can be analyzed or read aloud by a voice synthesizer. Cars may include automated driver-assistance systems that help users park or warn them about potentially dangerous situations. Intelligent video surveillance plays an increasingly important role in monitoring the security of public areas.

As mobile devices such as smart phones and tablets come equipped with cameras and more computing power, the demand for computer-vision applications is increasing. These devices have become smart enough to merge several photographs into a high-resolution panorama, or to read a QR code, recognize it, and retrieve information about a product from the Internet. It won't be long before mobile computer-vision technology becomes as ubiquitous as touch interfaces.

Computer vision is computationally expensive, however. Even an algorithm dedicated to solving a very specific problem, such as panorama stitching or face and smile detection, requires a lot of power. Many computer-vision scenarios must be executed in realtime, which implies that the processing of a single frame should be completed within 30-40 milliseconds. This is a very challenging requirement, especially for mobile and embedded computing architectures. Often, it is possible to trade off quality for speed. For example, the panorama-stitching algorithm can find more matches in source images and synthesize an image of higher quality, given more computation time. To meet the constraints of time and the computational budget, developers either compromise on quality or invest more time in optimizing the code for specific hardware architectures.

VISION AND HETEROGENEOUS PARALLEL COMPUTING

In the past, an easy way to increase the performance of a computing device was to wait for the semiconductor processes to improve, which resulted in an increase in the clock speed of the device. When the clock speed increased, all applications got faster without the programmer modifying them or the libraries that they relied on. Unfortunately, those days are over.

As transistors get denser, they also leak more current, and hence are less energy efficient. Improving energy efficiency has become an important priority. The process improvements now allow for more transistors per area, and there are two primary ways to put them to good use. The first is via parallelization: creating more identical processing units instead of making the single unit faster and more powerful. The second is via specialization: building domain-specific hardware accelerators that can perform a particular class of functions more efficiently. The concept of combining these two ideas—that is, running a CPU or CPUs together with various accelerators—is called heterogeneous parallel computing.

High-level computer-vision tasks often contain subtasks that can be run faster on special-purpose hardware architectures than on the CPU, while other subtasks are computed on the CPU. The GPU (graphics processing unit), for example, is an accelerator that is now available on every desktop computer, as well as on mobile devices such as smart phones and tablets.

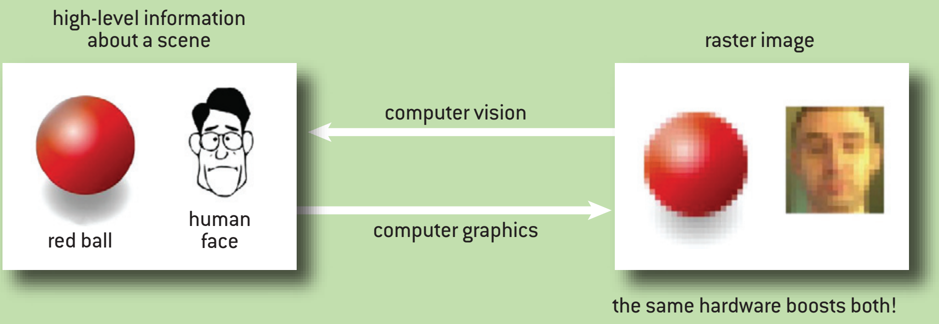
GRAPHICS HARDWARE

The first GPUs were fixed-function pipelines specialized for accelerated drawing of shapes on a computer display, as illustrated in figure 1. As GPUs gained the capability of using color images as input for texture mapping, and their results could be shared with the CPU rather than just being sent to the display, it became possible to use GPUs for simple image-processing tasks.

Making the fixed-function GPUs partially programmable by adding shaders was a big step forward. This enabled programmers to write special programs that were run by the GPU on every three-dimensional point of the surface and at every pixel rendered onto the output canvas. This vastly expanded the GPU's processing capability, and clever programmers began to try general-purpose computing on a GPU (GPGPU), harnessing the graphics accelerator for tasks for which it was not originally designed. The GPU became a useful tool for image processing and computer-vision tasks.

Graphics shaders, however, did not provide access to many useful hardware capabilities such as synchronization and atomic memory operations. Modern GPU computation languages such as

FIGURE 1 **Computer Vision and GPU**


high-level information about a scene
red ball
human face
raster image
computer vision
computer graphics
the same hardware boosts both!

CUDA, OpenCL, and DirectCompute are explicitly designed to support general-purpose computing on graphics hardware. GPUs are still not quite as flexible as CPUs, but they perform parallel stream processing much more efficiently, and an increasing number of non-graphics applications are being rewritten using the GPU compute languages.

Computer vision is one of the tasks that often naturally map to GPUs. This is not a coincidence, as computer vision solves the inverse of the computer graphics problem. While graphics transforms a scene or object description to pixels, vision transforms pixels to higher-level information. GPUs contain lots of similar processing units and are very efficient in executing simple, similar subtasks such as rendering or filtering pixels. Such tasks are often known as "embarrassingly parallel," because they are so easy to parallelize efficiently on a GPU.

Many tasks, however, do not parallelize easily, as they contain serial segments where the results of the later stages depend on the results of earlier stages. These serial algorithms do not run efficiently on GPUs and are much easier to program and often run faster on CPUs. Many iterative numerical optimization algorithms and stack-based tree-search algorithms belong to that class.

Since many high-level tasks consist of both parallel and serial subtasks, the entire task can be accelerated by running some of its components on the CPU and others on the GPU. Unfortunately, this introduces two sources of inefficiency. One is synchronization: when one subtask depends on the results of another, the later stage needs to wait until the previous stage is done. The other inefficiency is the overhead of moving the data back and forth between the GPU and CPU memories—and since computer-vision tasks need to process lots of pixels, it can mean moving massive data chunks back and forth. These are the key challenges in accelerating computer-vision tasks on a system with both a CPU and GPU.

OPENCV LIBRARY

OpenCV, the open source computer vision library, began as a research project at Intel in 1998.[5] It has been available since 2000 under the BSD open source license. OpenCV is aimed at providing the tools needed to solve computer-vision problems. It contains a mix of low-level image-processing functions and high-level algorithms such as face detection, pedestrian detection, feature matching, and tracking. The library has been downloaded more than 3 million times.

In 2010 a new module that provides GPU acceleration was added to OpenCV. The GPU module covers a significant part of the library's functionality and is still in active development. It is implemented using CUDA and therefore benefits from the CUDA ecosystem, including libraries such as NPP (NVIDIA Performance Primitives).

The GPU module allows users to benefit from GPU acceleration without requiring training in GPU programming. The module is consistent with the CPU version of OpenCV, which makes adoption easy. There are differences, however, the most important of which is the memory model. OpenCV implements a container for images called `cv::Mat` that exposes access to image raw data. In the GPU module the container `cv::gpu::GpuMat` stores the image data in the GPU memory and does not provide direct access to the data. If users want to modify the pixel data in the main program running on the GPU, they first need to copy the data from `GpuMat` to `Mat`.

```
#include <opencv2/opencv.hpp> #include <opencv2/gpu/gpu.hpp>
using namespace cv;
...
Mat image = imread(“file.png”);
gpu::GpuMat image_gpu;
image_gpu .upload(image);
gpu::GpuMat result;
gpu::threshold(image_gpu, result, 128, CV_THRESH_BINARY);
result.download(image);
imshow(“WindowName”, image);
waitKey ();
```

In this example, an image is read from a file and then uploaded to GPU memory. The image is thresholded there, and the result is downloaded to CPU memory and displayed. In this simple example only one operation is performed on the image, but several others could be executed on the GPU without transferring images back and forth. The usage of the GPU module is straightforward for someone who is already familiar with OpenCV.

This design provides the user with explicit control over how data is moved between CPU and GPU memory. Although the user has to write some additional code to start using the GPU, this approach is both flexible and allows more efficient computations. In general, it is a good idea to research, develop, and debug a computer-vision application using the CPU part of OpenCV, and then accelerate it with the GPU module. Developers should try different combinations of CPU and GPU processing, measure their timing, and then choose the combination that performs the best.

Another piece of advice for developers is to use the asynchronous mechanisms provided by CUDA and the GPU module. This allows simultaneous execution of data transfer, GPU processing, and CPU computations. For example, while one frame from the camera is processed by the GPU, the next frame is uploaded to it, minimizing data-transfer overheads and increasing overall performance.

PERFORMANCE OF THE OPENCV GPU MODULE

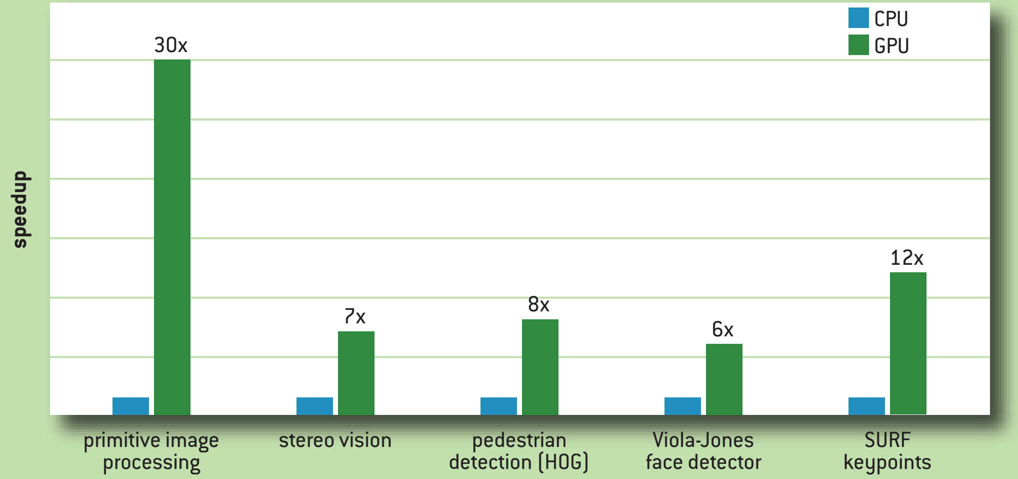
OpenCV’s GPU module includes a large number of functions, and many of them have been implemented in different versions, such as the image types (char, short, float), number of channels, and border extrapolation modes. This makes it challenging to report exact performance numbers. An added source of difficulty in distilling the performance numbers down is the overhead of synchronizing and transferring data. This means that best performance is obtained for large images where a lot of processing can be done while the data resides on the GPU.

To help the developer figure out the trade-offs, OpenCV includes a performance benchmarking suite that runs GPU functions with different parameters and on different data sets. This provides a detailed benchmark of how much different data sets are accelerated on the user’s hardware.

Figure 2 is a benchmark demonstrating the advantage of the GPU module. The speedup is measured against the baseline of a heavily optimized CPU implementation of OpenCV. OpenCV was compiled with Intel’s SSE (Streaming SIMD Extensions) and TBB (Threading Building Blocks) for multicore support, but not all algorithms use them. The primitive image-processing speedups have been averaged across roughly 30 functions. Speedups are also reported for several high-level algorithms.

FIGURE 2

CPU Versus GPU Performance Comparison



It is quite normal for a GPU to show a speedup of 30 times for low-level functions and up to 10 times for high-level functions, which include more overhead and many steps that are not easy to parallelize with a GPU. For example, the granularity for color conversion is per-pixel, making it easy to parallelize. Pedestrian detection, on the other hand, is performed in parallel for each possible pedestrian location, and parallelizing the processing of each window position is limited by the amount of on-chip GPU memory.

As an example, we accelerated two packages from ROS (Robot Operation System)[8]—stereo visual odometry and textured object detection—that were originally developed for the CPU. They contain many functional blocks and a class hierarchy.

Wherever it made sense, we offloaded the computations to the GPU. For example, OpenCV GPU implementations performed SURF (Speeded-up Robust Feature) key point detection, matching, and search of stereo correspondences (block matching) for stereo visual odometry. The accelerated packages were a mix of CPU/GPU implementations. As a result, the visual odometry pipeline was accelerated 2.7 times, and textured object detection was accelerated from 1.5 to 4 times, as illustrated in figure 3. Data-transfer overhead was not a significant part of the total algorithm runtime. This example shows that replacing only a few lines of code results in a considerable speedup of a high-level vision application.

STEREO CORRESPONDENCE WITH THE GPU MODULE

Stereo correspondence search in a high-resolution video is a demanding application that demonstrates how CPU and GPU computations can be overlapped. OpenCV's GPU module includes an implementation that can process full HD resolution stereo pair in real time (24 frames per second) on the NVIDIA GTX580.

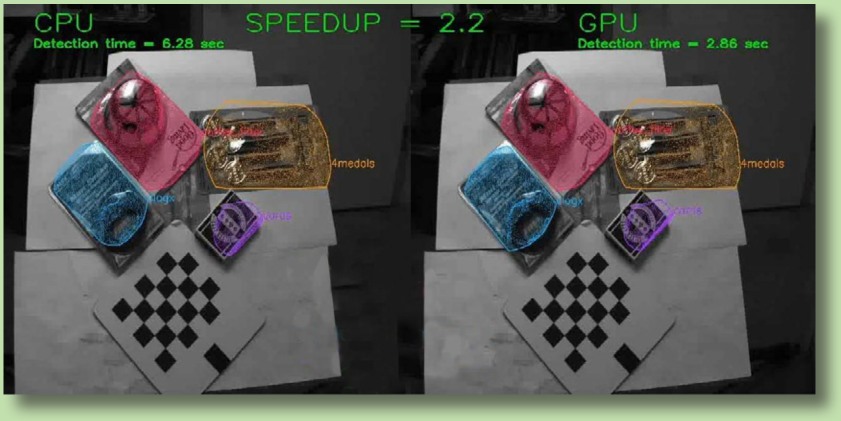
In a stereo system, two cameras are mounted facing in the same direction. While faraway objects

FIGURE 3

Texture Object Detection Application: CPU and GPU

Objects

- 🟥 coffee filter
- 🟨 medals
- 🟦 clog remover
- 🟪 playing cards



project to the same image locations on each camera, nearby objects project to different locations. This is called *disparity*. By locating for each pixel on the left camera image where the same surface point projects to on the right image, you can use the disparity to compute the distance to that surface point. Finding these correspondences between pixels in the stereo image pairs is the key challenge in stereovision.

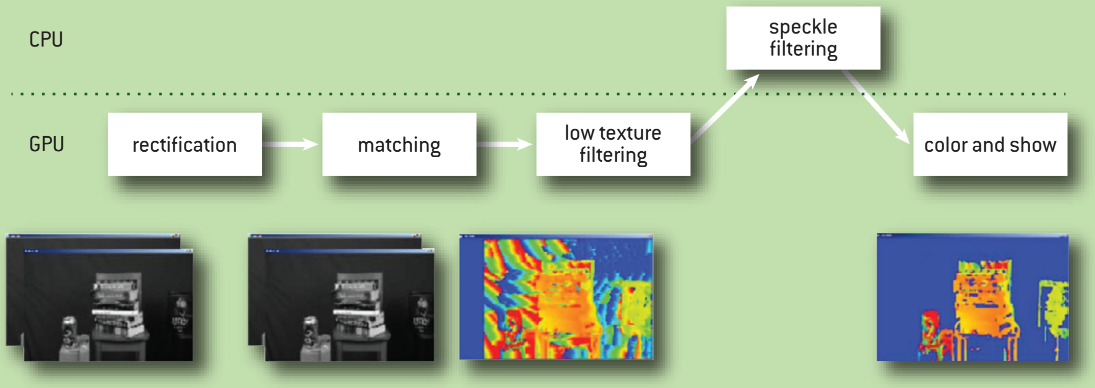
This task is made easier by rectifying the images. Rectification warps the images to an ideal stereo pair where each scene surface point projects to a matching image row. This way, only points on the same scan line need to be searched. The quality of the match is evaluated by comparing the similarity of a small window of pixels with the candidate matching pixel. Then the pixel in the right image whose window best matches the window of the pixel on the left image is selected as the corresponding match.

The computational requirements obviously increase as the image size increases, because there are more pixels to process. In a larger image the range of disparities measured in pixels also increases, which requires a larger search radius. For small-resolution images the CPU may be sufficient to calculate the disparities; with full HD resolution images, however, only the GPU can provide enough processing power.

Figure 4 presents a block-matching pipeline that produces a disparity image d(x,y) such that LeftImage(x,y) corresponds to RightImage(x - d(x,y),y). The pipeline first rectifies the images and then finds the best matches, as previously described. In areas where there is little texture—for example, a blank wall—the calculated matches are unreliable, so all such areas are marked to be ignored in later processing. As the disparity values are expected to change a lot only near object borders, the speckle-filtering stage eliminates speckle noise within large continuous regions of disparity image. Unfortunately, the speckle-filtering algorithm requires a stack-based depth-first search that is difficult to parallelize, so it is run on the CPU. Finally, the results are visualized using a false-color image.

All the steps except speckle filtering are implemented on the GPU. The most compute-intensive step is block matching. NVIDIA GTX580 has accelerated it seven times faster than a CPU

FIGURE 4 **Stereo Block Matching Pipeline**



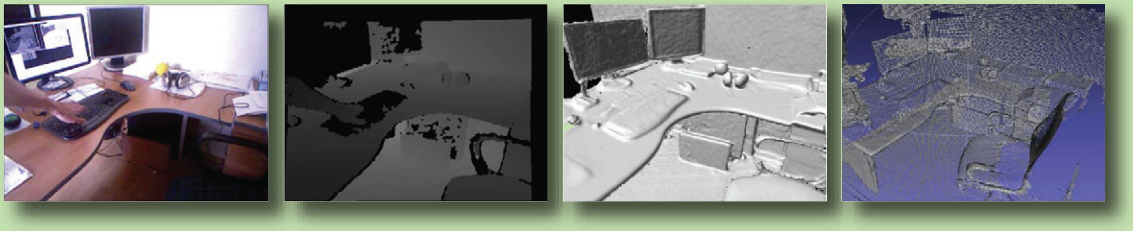
implementation on a quad core Intel i5-760 2.8-GHz processor with SSE and TBB optimizations. After this speedup the speckle filtering becomes the bottleneck, consuming 50 percent of the frame-processing time.

An elegant parallel-processing solution is to run speckle filtering on the CPU in parallel with the GPU processing. While the GPU processes the next frame, the CPU performs speckle filtering for the current frame. This can be done using asynchronous OpenCV GPU and CUDA capabilities. The heterogeneous CPU/GPU system now provides a sevenfold speedup for the high-resolution stereo correspondence problem, allowing realtime (24 fps) performance at full HD resolution.

KINECTFUSION

Microsoft's KinectFusion[4] is an example of an application that previously required slow batch processing but now, when powered by GPUs, can be run at interactive speeds. Kinect is a camera that produces color and depth images. Just by aiming the Kinect device around, one can digitize the 3D geometry of indoor scenes at an amazing fidelity, as illustrated in figure 5. An open source implementation of such a scanning application is based on the Point Cloud Library,[6] a companion

FIGURE 5 **RGB Frame, Depth Frame, Ray-Casted Frame, and Point Cloud**



library to OpenCV that uses 3D points and voxels instead of 2D pixels as basic primitives.

Implementing KinectFusion is not a simple task. Kinect does not return range measurements for all the pixels, and it works reliably only on continuous smooth matte surfaces. The range measurements that it returns are noisy, and depending on the surface shapes and reflectance properties, the noise can be significant. The noise also increases with the distance to the measured surface. Kinect generates a new depth frame 30 times in a second. If the user moves the Kinect device too fast, the algorithm gets confused and cannot track the motion using the range data. With a clever combination of good algorithms and using the processing power provided by GPUs, however, KinectFusion works robustly.

There are three key concepts that make a robust interactive implementation feasible. First, the tracking algorithm is able to process the new scan data so fast that the camera has time to move very little between the consecutive frames. This makes it feasible to track the camera position and orientation using just the range data.

Second, fusion of depth data is done using a volumetric surface representation. The presentation is a large voxel grid that makes it easier to merge the data from different scans in comparison with surface-based representations. To obtain high model quality, the grid resolution is chosen to be as dense as possible (512x512x512), so it has to be processed by the GPU for realtime rates.

Finally, the manner in which the new data is merged with the old reduces the noise and uncertainty as more data is gathered, and the accuracy of the model keeps improving. As the model gets better, tracking gets easier as well. Parallel ray casting through the volume is done on the GPU to get depth information, which is used for camera tracking on the next frame. So frame-to-frame movement estimation is performed only between the first and second frames. All other movements are computed on model-to-frame data, which makes camera tracking very robust.

All of these steps are computationally intensive. Volumetric integration requires the wide memory bandwidth that only the GPU can deliver at a price low enough to be affordable by normal consumers. Without GPUs this system would simply not be feasible.

Not every step of the computation is easy to do on a GPU, however. For example, the camera position is tracked on a CPU. Though the linear equation matrix required for camera position estimation is fully computed on the GPU, computing the final solution does not parallelize well, so it is done on the CPU, which results in some download and API call overhead. Another problem is that the bottom-level image in the hierarchical image processing approach is only 160x120, which is not large enough to fully load a GPU. All the other parts are ideal for GPU but limited by the amount of available GPU memory and computing resources.

Further development requires even more GPU power. At the moment, the size of the scene is limited by the volumetric representation. Using the same number of voxels but making them bigger would allow us to capture a larger scene but at a coarser resolution. Retaining the same resolution while scanning larger scenes would require more voxels, but the number of voxels is limited by the amount of memory available on the GPU and by its computational power.

ACCELERATING COMPUTER VISION ON MOBILE DEVICES

While PCs are often built with a CPU and a GPU on separate chips, mobile devices such as smart phones and tablets put all the computing elements on a single chip. Such an SoC (system on chip) contains one or more CPUs and a GPU, as well as several signal processors for audio and video processing and data communication. All modern smart phones and some tablets also contain one or more cameras, and OpenCV is available on both Android and iOS operating systems. With all these components, it is possible to create mobile vision applications. The following sections look at the mobile hardware in more detail, using NVIDIA's Tegra 2 and Tegra 3 SoCs as examples, and then introduce several useful multimedia APIs. Finally, two mobile vision applications are presented: panorama creation and video stabilization.

TOOLS FOR MOBILE COMPUTER VISION

At the core of any general-purpose computer is the CPU. While Intel's x86 instruction set rules on desktop computers, almost all mobile phones and tablets are powered by CPUs from ARM. ARM processors follow the RISC (reduced instruction set computing) approach, as can be deduced from ARM's original name, Advanced Risc Machines. While x86 processors were traditionally designed for high computing power, ARM processors were designed primarily for low-power usage, which is a clear benefit for battery-powered devices. As Intel is reducing power usage in its Atom family for mobile devices, and recent ARM designs are getting increasingly powerful, they may in the future reach a similar design point, at least on the high end of mobile computing devices. Both Tegra 2 and Tegra 3 use ARM Cortex-A9 CPUs.

Mobile phones used to have only one CPU, but modern mobile SoCs are beginning to sport several, providing symmetric multiprocessing. The reason is the potential for energy savings. One can reach roughly a similar level of performance using two cores running at 1 GHz each than with one core running at 2 GHz. Since the power consumption increases super-linearly with the clock speed, these two slower cores together consume less power than the single faster core. Tegra 2 provides two ARM cores, while Tegra 3 provides four. Tegra 3 actually contains five (four plus one) cores, out of which one, two, three, or four cores can be active at the same time. One of the cores, known as the shadow or companion core, is designed to use particularly little energy but can run only at relatively slow speeds. That mode is sufficient for standby, listening to music, voice calls, and other applications that rely on dedicated hardware such as the audio codec and require only a few CPU cycles. When more processing power is needed (e.g., reading e-mail), the slower core is replaced by one of the faster cores, and for increased performance (browsing, gaming) additional cores kick in.

SIMD (single instruction, multiple data) processing is particularly useful for pixel data, as the same instruction can be used on multiple pixels simultaneously. SSE is Intel's SIMD technology, which exists on all modern x86 chips. ARM has a similar technology called NEON, which is an optional coprocessor in the Cortex A9. The NEON can process up to eight and sometimes even 16 pixels at the same time, while the CPU can process only one element at a time. This is very attractive for computer-vision developers, as it is often easy to obtain three to four times performance speedup—and with careful optimization even more than six times. Tegra 2 did not include the NEON extension, but each of Tegra 3's ARM cores has a NEON coprocessor.

All modern smart phones include a GPU. The first generation of mobile GPUs implemented the fixed-functionality graphics pipeline of OpenGL ES 1.0 and 1.1. Even though the GPUs were designed for 3D graphics, they could be used for a limited class of image-processing operations such as warping and blending. The current mobile GPUs are much more flexible and support GLSL (OpenGL shading language) programming with the OpenGL ES 2.0 API, allowing programmers to run fairly complicated shaders at each pixel. Thus, many old-school GPGPU tricks developed for desktop GPUs about 10 years ago can now be reused on mobile devices. The more flexible GPU computing languages such as CUDA and OpenCL will replace those tricks in the coming years but are not available yet.

Consumption and creation of audio and video content is an important use case on modern mobile devices. To support them, smart phones contain dedicated hardware encoders and decoders both for audio and video. Additionally, many devices have a special ISP (image signal processor) that processes the pixels streaming out from the camera. These media accelerators are not as easily accessible and useful for computer-vision processing, but the OpenMAX standard helps.[1] OpenMAX defines three different layers: AL (application), IL (integration), and DL (development). The lowest, DL, specifies a set of primitive functions from five domains: audio, video, and image coding, and image andsignal processing. Some of them are of potential interest for computer-vision developers, especially video coding and image processing, because they provide a number of simple filters, color space conversions, and arithmetic operations. IL is meant for system programmers for implementing the multimedia framework and provides tools for purposes such as camera control. AL is meant for application developers and provides high-level abstractions and objects such as Camera, Media Player, and Media Recorder. The OpenMAX APIs are useful for passing image data efficiently between the various accelerators and other APIs such as OpenGL ES.

Sensors provide another interesting opportunity for computer-vision developers. Many devices contain sensors such as accelerometers, gyroscopes, compasses, and GPSes. They cannot perform calculations, but can be useful if the application needs to reconstruct the camera orientation or 3D trajectory. The problem of extracting the camera motion from a set of frames is challenging, in terms of both performance and accuracy. SLAM (simultaneous localization and mapping), SfM (structure from motion), and other approaches can compute the camera position and even the shapes of the objects the camera sees, but these methods are not easy to implement, calibrate, and optimize, and they require a lot of processing power. The sensors can nonetheless deliver a fairly accurate estimate of the device orientation at a fraction of the cost of relying only on visual processing. For accurate results the sensor input should be used only as a starting point, to be refined using computer-vision techniques.

OPENCV ON TEGRA

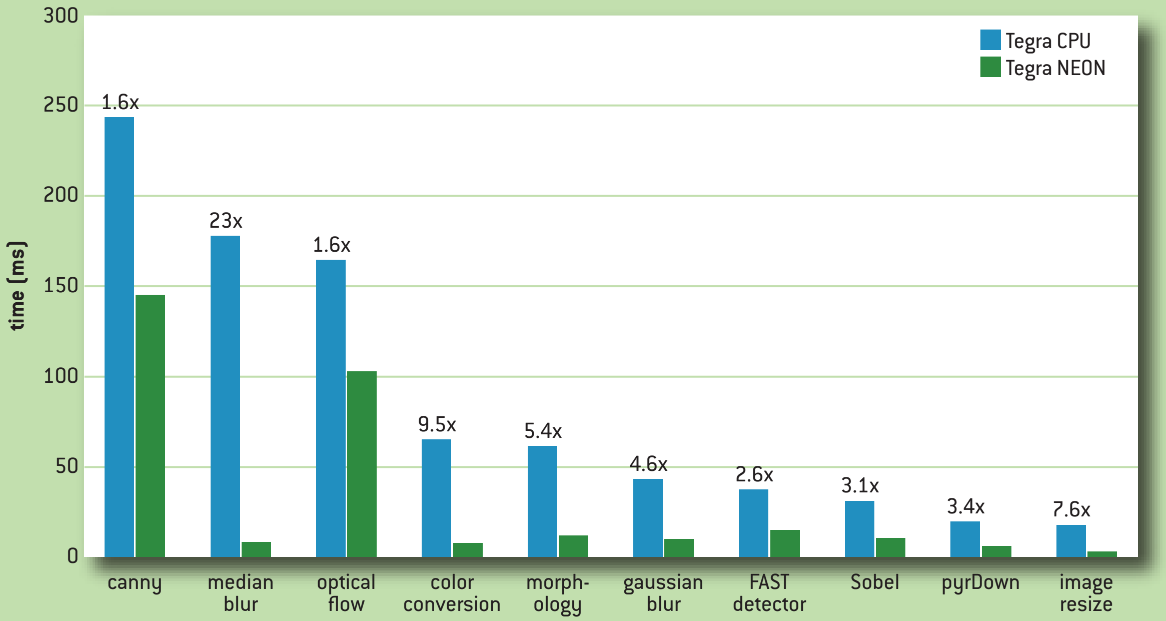
One of the major design and implementation goals for OpenCV has always been high performance. Porting both OpenCV and applications using it to mobile devices requires care, however, to retain a sufficient level of performance. OpenCV has been available on Android starting from the Google Summer of Code 2010 when it was first built and run on Google Nexus One. Several demo applications demonstrated almost realtime behavior, but it was obvious that OpenCV needed optimization and fine-tuning for mobile hardware.

That is why NVIDIA and Itseez decided to create a Tegra-optimized version of OpenCV. This work benefited from three major optimization opportunities: code vectorization with NEON, multithreading with the Intel TBB (Threading Building Blocks) library, and GPGPU with GLSL.

Taking advantage of the NEON instruction set was the most attractive of the three choices. Figure 6 compares the performance of original and NEON-optimized versions of OpenCV. In general, NEON requires basic arithmetic operations using simple and regular memory-access patterns. Those requirements are often satisfied by image-processing primitives, which are almost ideal for

FIGURE 6

Performance Improvement with NEON on Tegra 3



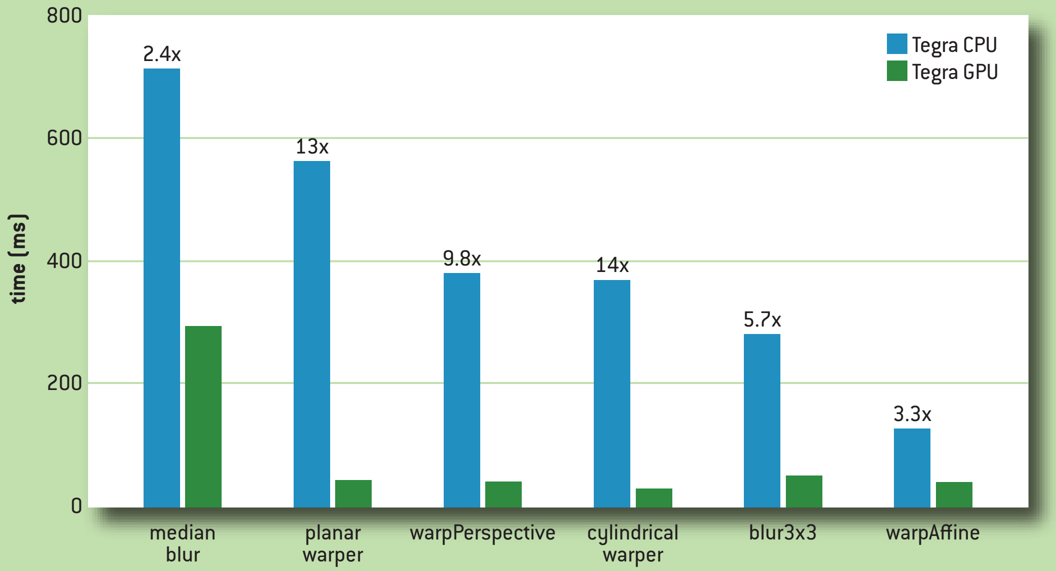
acceleration by NEON vector operations. As those primitives are often in the critical path of high-level computer vision workflows, NEON instructions can significantly accelerate OpenCV routines.

Multithreading on up to four symmetric CPUs can help at a higher level. TBB and other threading technologies enable application developers to get the parallel-processing advantage of multiple CPU cores. At the application level, independent activities can be distributed among different cores, and the operating system will take care of load balancing. This approach is consistent with the general OpenCV strategy for multithreading—to parallelize the whole algorithmic pipeline—while on a mobile platform we often also have to speed up primitive functions.

One approach is to split low-level functions into several smaller subtasks, which produces faster results. A popular technique is to split an input image into several horizontal stripes and process them simultaneously. An alternative approach is to create a background thread and get the result later while the main program works on other parts of the problem. For example, in the video stabilization application a special class returns an asynchronously calculated result from the previous iteration. Multithreading limits the speedup factor by the number of cores, which on the most advanced current mobile platforms is four, while NEON supports vector operations on 16 elements. Of course, these two technologies can be combined. If the algorithm is constrained by the speed of memory access, however, multithreading may not provide the expected performance improvement. For example, the NEON version of `cv::resize` does not gain from adding new threads, because a single thread already fully consumes the memory-bus capacity.

The final method applied during the optimization of the OpenCV library for the Tegra platform

FIGURE 7 Performance Improvement with GLSL on Tegra 3



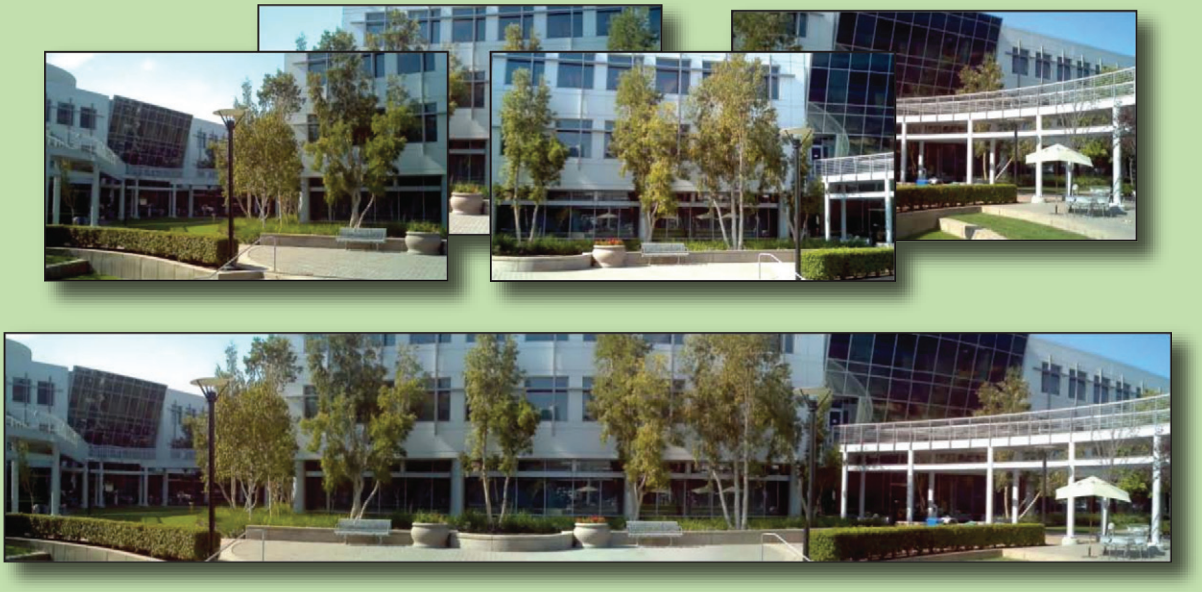
is GPGPU with GLSL shaders. Though the mobile GPU has limitations as discussed previously, on certain classes of algorithms the GPU is able to show an impressive performance boost while consuming very little energy. On mobile SoCs it is possible to share memory between CPU and GPU, which allows interleaving C++ and GLSL processing of the same image buffer. Figure 7 shows example speedups of some filters and geometric transformations from the OpenCV library.

An additional benefit of using the GPU is that at full speed it runs at a lower average power than the CPU. On mobile devices this is especially important, since one of the main usability factors for consumers is how long the battery lasts on a charge. We measured the average power and time elapsed to perform 10,000 iterations of some optimized C++ functions, compared with the same functions written in GLSL. Since these functions are both faster on the GPU, and the GPU runs at lower peak power, the result is significant energy savings (see table 1). We measured energy savings of 3-15 times when porting these functions to GPU.

TABLE 1: Energy savings with GLSL on Tegra 3.

OpenCV Function (10,000 iterations)	Energy Savings
Median Blur	3.43
Planar Warper	6.25
warpPerspective	6.45
Cylindrical Warper	3.89
blur3x3	3.6
warpAffine	15.38

FIGURE 8 Input Images and the Resulting Panorama



APPLICATIONS

We have developed two mobile vision applications using OpenCV: one that stitches together a panoramic image from several normal photographs, and another that stabilizes streaming video. The performance requirements are challenging. Our goal is realtime performance, where each frame should be processed within about 30 milliseconds, of which basic operations such as simply copying a 1280x720-pixel frame may take 8 milliseconds. Consequently, to a large extent the final design of an application and its underlying algorithm is determined by this constraint.

In both cases we were able to satisfy the time limits by using the GPU to optimize the applications' bottlenecks. Several geometric transformation functions such as image resizing and various types of image warping were ported to the GPU, doubling the application's performance. The results were not nearly as good when performing the same tasks using NEON and multithreading. One reason was that both applications deal with high-resolution four-channel images. As a result, the memory bus was overloaded and the CPU cores competed for the cache memory. At the same time we needed to program bilinear interpolation manually; ths process is implemented in GPU hardware. We learned that the CPU does not work as well for full-frame geometric transformations, and the help of the GPU was invaluable. Let's consider both applications in more detail.

PANORAMA STITCHING

In the panorama-stitching application our goal was to combine several ordinary images into a single panorama that has a much larger FOV (field of view) than the input images.[7] Figure 8 demonstrates the stitching of several detailed shots into a single high-resolution image of the whole image.

FIGURE 9

Panorama Stitching Pipeline

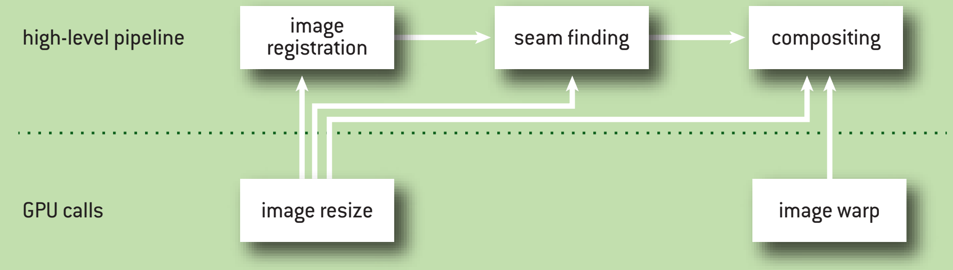


Figure 9 shows the processing pipeline for the OpenCV panorama-stitching application. The process of porting to Tegra started from some algorithmic improvements, followed by NEON and multithreading optimization; yet after all these efforts, the application still wasn't responsive enough and could not stitch and preview the resulting panorama at interactive speeds. Among the top bottlenecks were image resizing and warping. Resizing is required because different algorithmic steps are performed at different resolutions, and each input frame is resized about three times, depending on the algorithmic parameters. The type of warping needed depends on the desired projection mode (spherical, cylindrical, etc.), and images are warped before the final panorama blending.

With the GPU version of `cv::resize` we were able to decrease scaling time from 41 to 26 milliseconds for each input frame, which is equal to 1.6 times local speedup. Because of the GPU implementation of image warping, we could achieve even better local improvements—a boost of 8-14 times in performance, depending on the projection type. As a result, total application speedup was 1.5-2.0 times, meeting performance requirements.

VIDEO STABILIZATION

One of the negative consequences of recording video without a tripod is camera shake, which significantly degrades the viewing experience. To achieve visually pleasant results, all the movements should be smooth, and the high-frequency variations in camera orientation and translation need to be filtered. Numerous approaches have been developed, some of which have become open source or commercially available tools. There exist offline computationally intensive approaches that take a considerable amount of time, while the lightweight online algorithms are more suitable for mobile devices. High-end approaches often reconstruct the 3D movement of the camera and apply sophisticated nonrigid image warping to stabilize the video.[2] On mobile devices more lightweight approaches using translation, affine warping, or planar perspective transformations may make more sense.[3]

We experimented with translation and affine models, and in both cases the GPU was able to eliminate the major hotspot, which was the application of the compensating transformation to an input frame. Translation to compensate for the motion simply means shifting the input frame along the X and Y axes and cutting off some of the boundary areas for which some of the frames now do not contain color information (see figure 10).

FIGURE 10

Video Stabilization Input Sequence

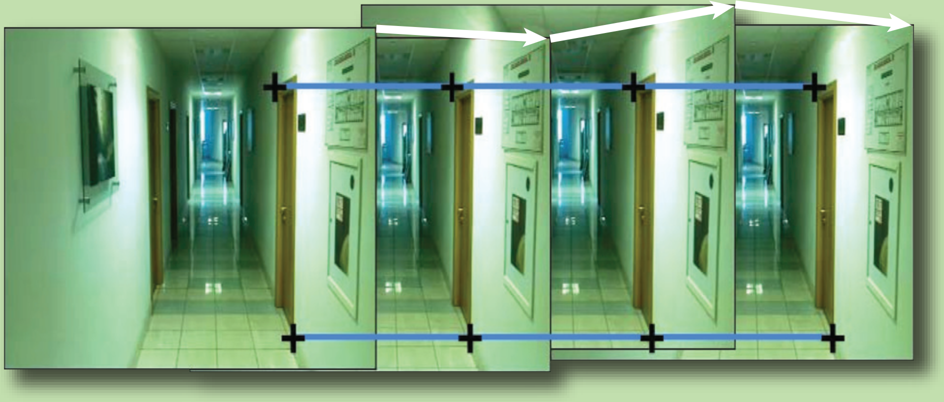
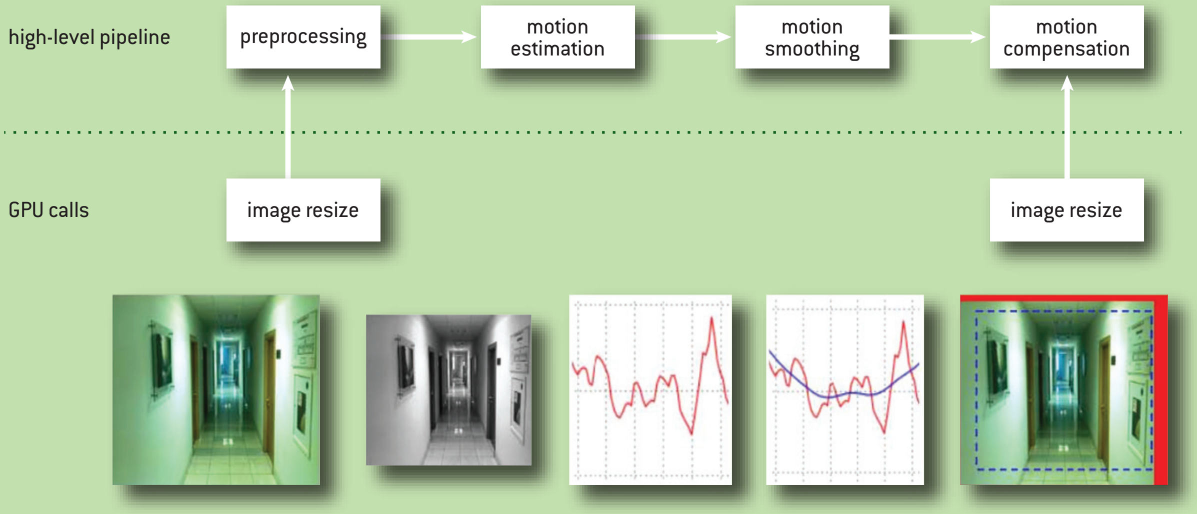


FIGURE 11

Video Stabilization Pipeline



In terms of programming, one should choose a properly located submatrix and then resize it into a new image at the same resolution as the original video stream, as suggested in figure 11. Surprisingly, this simple step consumed more than 140 milliseconds. Our GPU GLSL implementation was five to six times faster than C++ and took about 25 milliseconds.

Nevertheless, 25 milliseconds is still too long for a realtime algorithm, which is why we next tried to obtain more speed from asynchronous calls. A special class was created for stabilizing frames on the GPU. This class immediately returns a result from the previous iteration stored in its image-buffer field and creates a `TBB::task` for processing the next frame. As a result, GPU processing is performed in the background, and the apparent cost and delay for the caller is equal to just copying a full frame. This trick was also applied to an expensive color-conversion procedure, and with further optimizations of the memory-access patterns, we achieved realtime processing performance.

FUTURE DIRECTIONS

GPUs were originally developed to accelerate the conversion of 3D scene descriptions into 2D images at interactive rates, but as they have become more programmable and flexible, they have also been used for the inverse task of processing and analyzing 2D images and image streams to create a 3D description, to control some applications so they can react to the user or events in the environment, or simply to create higher-quality images or videos. As computer-vision applications become more commonplace, it will be interesting to see whether a different type of computer-vision processor that would be even more suitable for image processing is created to work with a GPU, or whether the GPU remains suitable even for this task. The current mobile GPUs are not yet as flexible as those on larger computers, but this will change soon enough.

OpenCV (and other related APIs such as Point Cloud Library) have made it easier for application developers to use computer vision. They are well-documented and vibrant open source projects that keep growing, and they are being adapted to new computing technologies. Examples of this evolution are the transition from a C to a C++ API in OpenCV and the appearance of the OpenCV GPU module. The basic OpenCV architecture, however, was designed mostly with CPUs in mind. Maybe it is time to design a new API that explicitly takes heterogeneous multiprocessing into account, where the main program may run on a CPU or several CPUs, while major parts of the vision API run on different types of hardware: a GPU, a DSP (digital signal processor), or even a dedicated vision processor. In fact, Khronos has recently started working on such an API, which could work as an abstraction layer that allows innovation independently on the hardware side and allows for high-level APIs such as OpenCV to be developed on top of this layer while being somewhat insulated from the changes in the underlying hardware architecture.

ACKNOWLEDGMENTS

We would like to thank our colleagues for help with numerical experiments and constructive discussions; Colin Tracey and Marina Kolpakova for help with power analysis; Andrey Pavlenko and Andrey Kamaev for GLSL and NEON code; and Shalini Gupta, Shervin Emami, and Michael Stewart for additional comments. NVIDIA provided support, such as all the hardware used in the experiments.

REFERENCES

1. Khronos OpenMAX standard; http://www.khronos.org/openmax.
2. Liu, F., Gleicher, M., Wang, J., Jin, H., Agarwala, A. 2011. Subspace video stabilization. *ACM Transactions on Graphics* 30(1):4:1–4:10.
3. Matsushita, Y., Ofek, E., Ge, W., Tang, X., Shum, H.-Y. 2006. Full-frame video stabilization with motion inpainting. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(7):1150–1163.
4. Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., Kohli, P., Shotton, J., Hodges, S., Fitzgibbon, A. W. 2011. Kinectfusion: Real-time dense surface mapping and tracking. IEEE International Symposium on Mixed and Augmented Reality (ISMAR): 127–136.

5. OpenCV library; http://code.opencv.org.
6. Point Cloud Library; http://pointclouds.org.
7. Szeliski, R. 2006. Image alignment and stitching: a tutorial. *Foundations and Trends in Computer Graphics and Vision* 2(1):1–104.
8. Willow Garage. Robot Operating System; http://www.ros.org/wiki/.

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

KARI PULLI is a senior director at NVIDIA Research, where he heads the Mobile Visual Computing Research team and works on topics related to cameras, imaging, and vision on mobile devices. Previously he was at Nokia, where he was the sixth Nokia Fellow and a member of the CEO technology council. He has worked on standardizing mobile media APIs at Khronos and JCP and was the technical lead of the Digital Michelangelo Project at Stanford University. He has a B.Sc. from the University of Minnesota, M.Sc. and Lic. Tech. from the University of Oulu (Finland), and Ph.D. from the University of Washington (Seattle), all in computer science/engineering; and an MBA from the University of Oulu. He is the author of *Mobile 3D Graphics* (Morgan Kaufmann, 2007).

ANATOLY BAKSHEEV is a project manager at Itseez. He started his career there in 2006 and was the principal developer of the multi-projector system Argus Planetarium. In 2008 he worked as a senior engineer on the OpenCV team. Since 2010 he has been the leader of the OpenCV GPU project that brings computationally intensive vision algorithms to GPU. He acts as both the architect and the principal developer of the OpenCV GPU module and in 2011 started working on the GPU acceleration module for Point Cloud Library. Anatoly graduated from Nizhny Novgorod State University, Russia, with an M.Sc. in computer science.

KIRILL KORNYAKOV is a project manager at Itseez, where he leads the development of the OpenCV library for mobile devices. He manages activities on mobile operating system support and computer-vision applications development, including performance optimization for NVIDIA Tegra platform. He has also worked on realtime computer-vision systems for open source and commercial products, among them stereovision on GPU and face detection in complex environments. Kirill has B.Sc. and M.Sc. degrees from Nizhny Novgorod State University, Russia.

VICTOR ERUHIMOV is CTO of Itseez. Prior to co-founding the company, he worked as a project manager and senior research scientist at Intel, where he applied computer-vision and machine-learning methods to automate Intel fabs and revolutionize data processing in semiconductor manufacturing. Before joining the manufacturing group he was developing technologies for human-motion capture, image retrieval, super-resolution, and face analysis. He is the author of more than 20 computer-vision and machine-learning papers, and holds several U.S. and international patents.