



Systematic Optimization of Image Processing Pipelines Using GPUs

Peter Roch^(✉), Bijan Shahbaz Nejad, Marcus Handte, and Pedro José Marrón

University of Duisburg-Essen, Essen, Germany
`peter.roch@uni-due.de`

Abstract. Real-time computer vision systems require fast and efficient image processing pipelines. Experiments have shown that GPUs are highly suited for image processing operations, since many tasks can be processed in parallel. However, calling GPU-accelerated functions requires uploading the input parameters to the GPU's memory, calling the function itself, and downloading the result afterwards. In addition, since not all functions benefit from an increase in parallelism, many pipelines cannot be implemented exclusively using GPU functions. As a result, the optimization of pipelines requires a careful analysis of the achievable function speedup and the cost of copying data. In this paper, we first define a mathematical model to estimate the performance of an image processing pipeline. Thereafter, we present a number of micro-benchmarks gathered using OpenCV which we use to validate the model and which quantify the cost and benefits for different classes of functions. Our experiments show that comparing the function speedup without considering the time for copying can overestimate the achievable performance gain of GPU acceleration by a factor of two. Finally, we present a tool that analyzes the possible combinations of CPU and GPU function implementations for a given pipeline and computes the most efficient composition. By using the tool on their target hardware, developers can easily apply our model to optimize their application performance systematically.

Keywords: Image processing · Performance evaluation · OpenCV · Computer vision · CUDA · Parallel processing

1 Introduction

Computer vision systems have been applied successfully in a broad spectrum of application areas, for example traffic surveillance [2, 3], automatic attendance management [19] or tracking of human interactions [11]. These and other similar systems often have real-time constraints and require the image processing to run at high frame rates. As a result, their implementation must be fast and efficient.

In most applications, the image processing logic is structured as a pipeline of image processing functions that are executed sequentially. First, the image needs to be captured from a camera. Thereafter, the image runs through a sequence

of pre-processing functions, which often include filtering noise, resizing or gray-scaling. The main task is then to extract the desired information by means of edge detection, feature-matching or stereo-correspondence matching, to name a few. Some applications may also require additional post-processing steps, e.g., to filter results or to overlay the extracted information onto the original image.

Since GPUs are capable of executing highly parallel code and since many image processing functions can significantly benefit from a parallel implementation, they can be a powerful tool to reduce the processing time. As a result, widely used libraries such as OpenCV [12] often provide alternative function implementations that can leverage a dedicated graphics card.

However, calling such a GPU-accelerated function requires uploading the input parameters to the GPU's memory, calling the function itself, and downloading the result afterwards. In addition, since not all functions benefit from an increase in parallelism, many pipelines cannot be implemented exclusively using GPU functions. As a result, the optimization of pipelines can become a non-trivial task which requires careful and thorough analysis of the achievable function speedup and the cost of copying data.

To support the optimization of image processing pipelines using GPUs in a systematic manner, this paper makes the following contributions:

1. First, the paper defines a mathematical model to compute the execution time of arbitrary pipelines that combine CPU and GPU functions.
2. Second, the paper studies the costs of GPU-acceleration and validates the analytical model using a number of benchmarks gathered with OpenCV.
3. Third, the paper presents a framework, which computes the optimal combination of CPU and GPU functions for a given image processing pipeline.

2 Analytical Model

NVIDIA CUDA [10] is a general purpose parallel computing platform and programming model to make use of NVIDIA GPUs. A GPU is organized into multi-dimensional *Grids* of *Thread-Blocks*, each containing multiple *Threads*. Every *Thread* has access to its own local memory as well as to the shared memory of its *Thread-Block*. Every *Grid*, and therefore every *Thread-Block* inside also has access to the GPU's global memory, often referred to as device-memory. This architecture is illustrated in Fig. 1. A *Kernel* is defined as a function, which is executed on one or more *Thread-Blocks*, hence each *Thread* inside the same *Thread-Block* will execute the same *Kernel*.

Since the GPU uses its own memory, data exchanged between the GPU and the host system needs to be copied between the device memory and the host memory. The operation of copying data to the device memory is commonly referred to as *uploading*, whereas copying data from device memory to host memory is called *downloading*. Since CUDA 6, it is possible to allocate data inside *unified memory*, where the CUDA software or hardware will manage uploading and downloading. Usually, host memory is pageable, meaning that the OS can

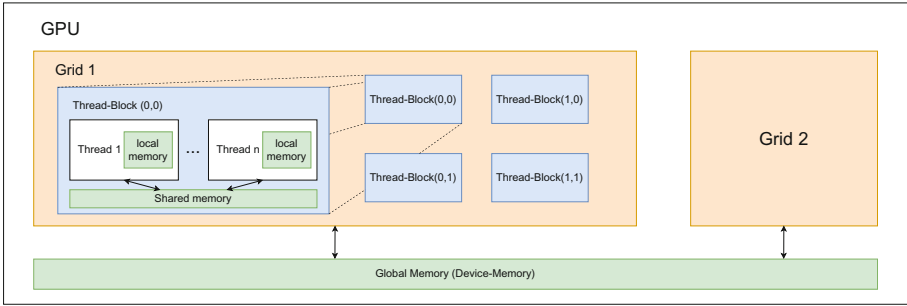


Fig. 1. GPU architecture

allocate memory not only inside the physical RAM, but also inside a swap-file located on the hard-drive. In contrast, page-locked memory cannot be moved to a swap-file and has to be kept always inside the physical RAM.

The GPU cannot access pageable host memory directly. If data has to be copied to the device memory, the CUDA driver will allocate a temporary buffer inside page-locked memory. This additional copy operation can be avoided if host memory is allocated as page-locked memory beforehand. However, this approach has to be used carefully, because physical RAM is limited. If an application allocates too much page-locked memory, the operating system is forced to use a swap-file for other processes, which can slow down the whole system.

Since most non-trivial applications will require the sequential execution of multiple image processing functions, we can mathematically describe the execution time of such a pipeline by summing up the processing times of all functions and adding time required for uploading and downloading the function parameters for each transition between CPU and GPU as follows:

Let P be an arbitrary pipeline containing N tasks, where $T_i(p_{i1}, p_{i2}, \dots, p_{in})$, noted as $T_i(p_{ij})_{j=1}^n$, is the i -th task in P accepting n parameters, with p_{ij} being the j -th parameter of Task i . Further, we define $u(p_{ij})$ and $d(p_{ij})$ as the time needed to upload or download the parameter p_{ij} and $t_{\uparrow}(p_{ij})$ and $t_{\downarrow}(p_{ij})$ as the time needed to make parameter p_{ij} available for GPU or CPU execution:

$$\begin{aligned}
 t_{\uparrow}(p_{ij}) &= \begin{cases} 0, & \text{if parameter } p_{ij} \text{ is stored in device memory} \\ u(p_{ij}), & \text{otherwise} \end{cases} \\
 t_{\downarrow}(p_{ij}) &= \begin{cases} 0, & \text{if parameter } p_{ij} \text{ is stored in host memory} \\ d(p_{ij}), & \text{otherwise} \end{cases}
 \end{aligned} \tag{1}$$

Moreover, we define $CPU(T_i(p_{ij})_{j=1}^n)$ and $GPU(T_i(p_{ij})_{j=1}^n)$ as the time needed to execute the OpenCV CPU or GPU function corresponding to task $T_i(p_{ij})_{j=1}^n$. The processing time $PT(T_i(p_{ij})_{j=1}^n)$ of task $T_i(p_{ij})_{j=1}^n$ is calculated as shown in Eq. 2:

$$PT \left(T_i (p_{ij})_{j=1}^n \right) = \begin{cases} \sum_{j=1}^n t_{\downarrow}(p_{ij}) + CPU \left(T_i (p_{ij})_{j=1}^n \right), & \text{CPU execution} \\ \sum_{j=1}^n t_{\uparrow}(p_{ij}) + GPU \left(T_i (p_{ij})_{j=1}^n \right), & \text{GPU execution} \end{cases} \quad (2)$$

The execution time $ET(P)$ for the complete pipeline can then be deduced as follows:

$$ET(P) = \sum_{i=1}^N PT \left(T_i (p_{ij})_{j=1}^n \right) \quad (3)$$

Note that this model does not introduce restrictions on the sequence of CPU and GPU functions. However, in most cases, we expect that the first function (image acquisition) and the last function (usage of the results) cannot run on the GPU. Thus, even “fully” GPU-accelerated pipelines will usually require at least one parameter upload for the image and one download to use the final result.

3 Benchmarks and Validation

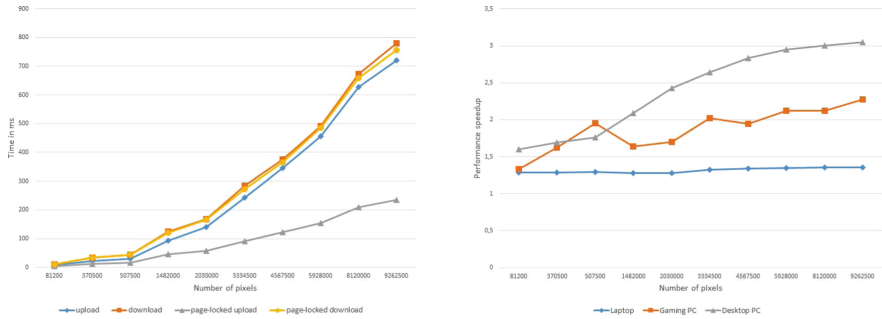
To estimate the performance of an image processing pipeline using the model defined in Sect. 2, it is necessary to determine the time required for executing each function on the CPU and GPU. In addition, it is also necessary to determine the time required for uploading and downloading of their respective parameters given a particular pipeline composition. Thus, to validate the model, we first measure the timings for representative functions using micro-benchmarks and then compare the model with actual measurements of two different pipelines.

3.1 Test Setup

For both, the micro-benchmarks and the validating measurements, we are using the widely used image processing library OpenCV. While OpenCV itself is written in C and C++, it offers bindings for Java, Python and MATLAB. Since Java is less error-prone and has a higher productivity compared to C++ [15], we are using OpenCV’s Java interface for all measurements. The overhead of calling native methods from Java is provenly in the order of nanoseconds [8], so the experimental results detailed in this paper should be comparable to other interfaces as well. Since the official Java interface does not support GPU functions, we used the JavaCV [17] wrapper to invoke them.

As input data for the measurements, we use a skyscraper image of [5] and the left image of a stereo image pair of [18], showing a motorcycle, and resize them to different resolutions. To minimize timer effects during bench-marking, each measurement represents the total time taken by 100 executions of the respective function. Since we cannot control the OS scheduling, we repeat each measurement 100 times and we report the average processing time \bar{time} in milliseconds as well as the standard deviation σ . To study the effects of different hardware, we repeat all measurements on three different computer system configurations:

- *L*: A high-end laptop with an Intel[®] Core[™] i7-9750H CPU, an NVIDIA GeForce RTX 2070 with Max-Q Design GPU and 16 GB RAM



(a) Desktop PC Upload & Download Times (b) Paged & Page-locked Upload Speedup

Fig. 2. Comparison of upload and download functions on different hardware

- *GPC*: A high-end gaming PC with an Intel[®] Core™ i7-7800X CPU, an NVIDIA GeForce GTX 1080 Ti GPU and 32 GB RAM
- *DPC*: A desktop PC with an Intel[®] Core™ i5-7500 CPU, an NVIDIA GeForce GTX 1050 Ti GPU and 8 GB RAM.

3.2 Micro-benchmarks for Uploading and Downloading

In OpenCV, larger parameter values, including images, are represented as instances of the class `cv::Mat` (i.e., matrix). The complementary GPU representation of this class is `cv::cuda::GpuMat`. OpenCV’s implementation of `GpuMat` does not use the *unified memory* feature introduced in CUDA 6, so the programmer has to take care of uploading and downloading the data manually. This is done by the functions `GpuMat.upload(Mat)` and `GpuMat.download(Mat)`, respectively.

Figure 2 shows the results of micro-benchmarks for uploading and downloading `Mat` parameters representing images of varying size using paged and page-locked memory. Figure 2a shows the timings of executing 100 operations per measurement on the desktop PC. The x-axis shows the number of pixels, whereas the y-axis denotes the processing time in ms. The average standard deviation σ across these measurements is 5% of the average processing time. The maximum is 20%.

As depicted in Fig. 2a, page-locked memory solely affects the upload operation and the remaining operations exhibit a similar overhead. Since this result is identical for the laptop and the gaming PC, we omit their figures for the sake of brevity. Instead, we focus on the difference between paged and page-locked uploads for all hardware configurations in Fig. 2b.

Interestingly, the laptop’s i7-9750H processor shows only little benefit of page-locked memory, while the i5-7500 processor has the highest benefit of page-locked memory. This is probably due to internal memory handling of the different processor types. Furthermore, both i7 processors seem to have an approximately

constant acceleration, whereas the i5’s acceleration factor increases with increasing parameter sizes.

3.3 Micro-benchmarks of OpenCV Functions

Since OpenCV is implementing more than 2500 functions, it is not practical to report on every single one. Instead, we group them into different classes depending on their structure and measure a small number of representative functions for each group. The intuition is that the functions belonging to the same class are likely to behave similarly. Table 1 shows the test results with input images of different resolutions: 350×232 , 1482×1000 and 3705×2500 . The faster implementation of each CPU-GPU pair is highlighted.

Table 1. Micro-benchmarks for OpenCV Functions (100 Operations in ms)

Task	PC	350×232				1482×1000				3705×2500			
		CPU		GPU		CPU		GPU		CPU		GPU	
		time	σ	time	σ	time	σ	time	σ	time	σ	time	σ
T	L	< 0.1	<0.1	4.9	2.2	2.1	0.4	5.3	0.9	54.8	3.1	9.7	1.3
	GPC	< 0.1	<0.1	3.7	0.5	3.0	0.6	4.5	0.5	40.7	1.2	9.1	0.4
	DPC	< 0.1	<0.1	4.2	0.5	2.1	0.3	8.0	1.2	153.7	2.5	27.8	1.7
G	L	3.0	<0.1	4.1	0.4	12.9	1.1	6.9	1.1	135.8	2.9	24.0	1.5
	GPC	3.0	<0.1	3.3	0.4	11.8	1.9	7.9	0.4	98.3	2.1	17.2	0.5
	DPC	3.0	<0.1	4.8	1.2	16.7	0.9	14.4	1.4	286.4	1.5	66.2	1.0
MF	L	259.6	4.8	155.4	11.0	4342.6	15.8	2086.7	3.0	26211.4	26.0	12170.8	13.7
	GPC	255.4	2.5	210.8	4.7	4349.1	18.8	2913.5	13.6	26281.9	63.7	17366.4	2.7
	DPC	274.4	1.0	390.2	4.0	4663.4	21.1	4075.8	9.1	28074.4	42.6	22189.4	2.6
GF	L	9.5	1.2	8.5	0.5	59.2	2.5	21.2	1.0	325.9	10.3	98.8	0.8
	GPC	8.9	2.4	11.9	0.8	67.8	3.2	101.2	1.5	294.3	8.7	524.5	1.0
	DPC	11.7	0.8	13.5	2.0	87.7	1.3	106.3	2.0	474.9	3.8	577.8	0.9
DIL	L	9.1	0.3	25.2	3.8	52.0	1.7	171.1	1.2	310.5	1.6	934.5	1.7
	GPC	9.2	1.6	80.6	2.3	51.3	0.7	849.9	1.8	353.6	6.6	4775.9	2.0
	DPC	10.0	<0.1	105.2	3.6	54.4	0.9	1112.7	2.6	351.4	1.9	6401.5	0.9
ERO	L	9.1	0.3	24.0	0.1	51.0	1.6	171.8	1.2	310.9	2.1	929.8	0.9
	GPC	9.0	0.1	80.2	1.0	51.7	2.5	859.0	1.4	352.4	6.2	4778.0	1.9
	DPC	10.0	<0.1	105.4	2.3	53.8	0.6	1119.3	3.5	351.3	3.5	6421.9	0.8
CAN	L	70.6	0.8	46.4	9.7	472.1	2.5	85.5	5.9	1717.1	16.0	328.2	2.8
	GPC	50.6	4.3	54.4	1.6	369.6	3.7	85.9	3.3	1693.5	16.3	297.5	3.3
	DPC	57.9	0.3	54.1	4.8	706.8	2.3	245.6	7.7	3050.3	10.1	1108.6	3.0
SKY	L	245.1	6.9	666.2	77.9	2341.0	6.6	1191.1	15.5	6436.7	9.3	1982.1	62.4
	GPC	234.6	3.1	661.2	95.6	3505.6	105.1	1611.7	22.6	8629.4	51.6	3423.1	30.3
	DPC	248.4	1.0	727.0	73.2	2463.6	9.6	1523.4	13.3	7366.0	10.7	3840.0	24.9
MOT	L	296.2	4.2	723.2	74.4	1975.5	5.0	1187.3	3.8	6712.7	16.4	1987.3	55.4
	GPC	287.4	2.0	895.5	18.1	2791.8	88.8	1611.4	11.4	8962.9	67.7	3562.7	50.1
	DPC	305.2	1.2	791.3	34.0	2092.2	3.6	1519.9	11.9	7643.2	10.4	3943.4	99.5
STR	L	173.8	2.8	121.0	11.3	4248.0	25.8	895.2	6.5	25060.6	213.9	5710.3	12.7
	GPC	168.4	4.4	241.1	12.5	4222.8	28.1	711.0	5.4	23213.5	207.6	3889.5	50.5
	DPC	183.0	1.3	264.5	7.3	5649.1	24.5	2619.5	8.7	40224.3	198.7	18217.0	1.6

Pixel-Wise Operations. These operations need to perform some function on every single pixel in the image. As a result, the image content does not affect the processing time for the task, instead the time is primarily determined by the image size. As representative functions for this category, we select thresholding (“T”) and gray-scaling (“G”). Thresholding low resolution images is faster when executed on the CPU, with the GPU being faster only with higher resolution images as input. The same holds for gray-scaling. With small input images, these functions cannot be parallelized sufficiently to be accelerated by the GPU.

Kernel Operations. Kernel operations are functions which need to manipulate each pixel of the image by taking into account the pixel’s neighborhood. The kernel can have any shape or size and is scanned over the whole image. The new value of the kernels anchor point is determined by a function applied to every pixel inside the kernels area. Kernel operations can be further categorized into two sub-categories: filtering and morphological operations.

To evaluate filtering operations, we use median filter (“MF”) and gaussian filter (“GF”). The median filter is fundamentally slower than the gaussian filter. However, it can be accelerated by its GPU implementation on all tested hardware. The gaussian filter, while speeded up on the laptops graphics card, is slower on both other GPUs. We believe that this is caused by hardware differences.

Both morphological operations, dilation (“DIL”) and erosion (“ERO”), need approximately the same amount of time. Interestingly, their GPU implementation is considerably slower. The CUDA implementation of OpenCV’s morphological operations uses the NVIDIA Performance Primitives (NPP), which has a time complexity of $O(npq)$, with n image pixels and a rectangular kernel of size $p * q$, according to [21]. This makes the NPP implementation comparably slow. The authors describe a parallel implementation of the van Herk/Gil-Werman (vHGW) algorithm [4, 7] using CUDA, which is significantly more efficient.

Geometric Information Retrieval Operations. Functions of this category extract geometric information out of the image. Common algorithms falling into this category are Canny edge detection, Hough line transform or Hough circle transform. To evaluate geometric information retrieval operations, we use the Canny edge detection function (“CAN”), since it has a broad range of applications. The laptop and gaming PC perform similar, while the desktop PC is notably slower. The GPU implementation is faster on all three systems.

Feature-Based Operations. Feature-based operations are functions which are used to compare two images. A feature is defined as some point in an image with information describing that point. A feature extractor can find points distinguishable from other points in the image and compute descriptors describing these points. A feature matcher can compare descriptors of different features and match corresponding features found in different images. A special case of feature detection and matching is stereo matching, where a stereo matcher has to find

corresponding points in two images in horizontal lines in order to compute a disparity value, which in turn can be used to compute the physical distance to the observed objects.

From this category, we measure feature detection, feature matching and stereo matching. To test feature detection, we resize the skyscraper image (“SKY”) and the motorcycle image (“MOT”) to have the same resolutions. Despite the fact that the motorcycle image is more suitable for feature detection, the results show similar processing times on both images. Again, the GPU accelerates the feature detection process depending on the images resolution. For stereo matching (“STR”) two horizontally aligned images from the same scene are needed. For this, we use a second image of the motorcycle, taken from a different perspective. Similar to feature detection, stereo matching is also accelerated by the GPU at higher resolutions.

Feature matching does not operate on images, but instead uses the computed descriptors to identify matches. Hence, the time required to match two images depends on the number of features rather than their resolution. Thus, instead of varying the image size, we vary the number of features when comparing feature matching implementations. Figure 3 shows the resulting GPU speedup.

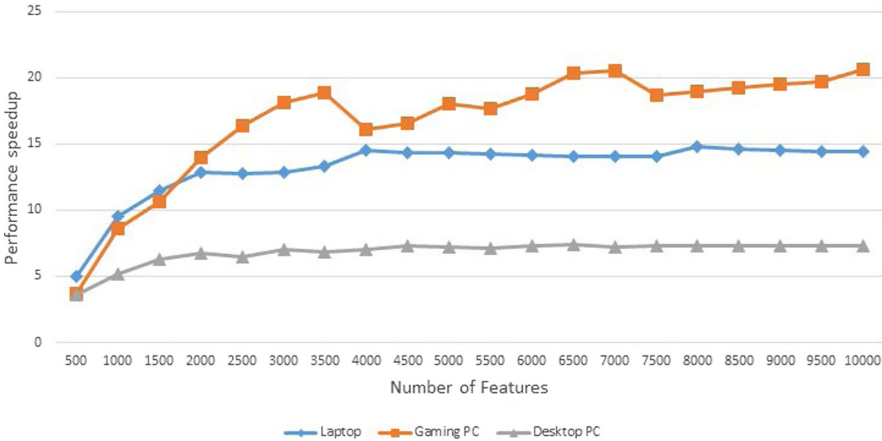


Fig. 3. Speedup of GPU-based feature matching

Structural Analysis Operations. Operations of this category are used to analyze structures in images. This covers different functions such as finding contours, finding the convex hull of an object or polygon approximation. Such operations are often unsuited for parallel programming, because parts of the image need to be analyzed in connection with other parts.

Typical OpenCV functions of this category like `findContours`, `convexHull` or `approxPolyDP` are only implemented using the CPU. Therefore, we don't

discuss their timings. However, for a systematic optimization, the presence of such functions must be considered, since pipelines that include them will have to switch from GPU to CPU (and back), which can change the optimal composition.

3.4 Experimental Validation of the Analytical Model

To validate the model defined in Sect. 2, we discuss experiments with two image processing pipelines. The first pipeline performs edge detection and involves gray-scaling, noise reduction using a Gaussian filter and finally a Canny edge detector. The second pipeline computes a disparity map and it involves gray-scaling, a Gaussian filter as well as stereo-matching. For each pipeline, we create a CPU- and a GPU-based implementation and measure their execution time for different image sizes. Then, we use the analytical model described in Sect. 2 and the results of the micro-benchmarks to compute an estimate for the execution time, with and without considering the overhead for data transfer among GPU and CPU. Figure 4a shows the results for the lowest resolution and Fig. 4b shows the results for the highest resolution.

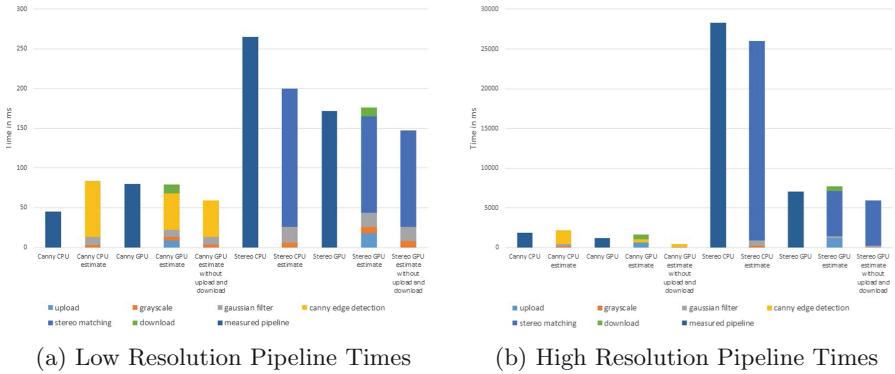


Fig. 4. Comparison of model estimates and measurements for different pipelines

When comparing the measurements with the estimate gathered from applying the analytical model, we find that, in general, the model is able provide an estimate that lies within 10–20% of the measured time. Given that the micro-benchmarks used to compute the estimate are not perfect and given the fact that they cannot completely capture all effects such as caching, for example, we would argue that this result is sufficiently accurate. When simply comparing the time required to execute the CPU and GPU functions, as done when ignoring the time for uploading and downloading, the computed estimations get significantly worse. In fact, in all cases, this results in an overestimation of the speedup achievable by a GPU implementation and depending on the pipeline, the resulting error can become fairly large. For example, when looking at the edge detector pipeline with a resolution of 3705×2500 , the actual time required

by the GPU implementation exceeds the estimate by a factor of more than 2. This clearly indicates that the model presented in Sect. 2 is a more realistic and thus, better basis for performance optimizations.

4 Optimization Framework

Although the model presented in Sect. 2 is fairly simple, applying it in practice can be cumbersome. Since many functions can be executed on the GPU or CPU interchangeably, the number of possible pipeline compositions usually grows exponentially with its length. In addition, since the upload and download effort depends on the location of parameters, it is necessary to track the parameter location for each possible pipeline composition. Last but not least, as shown in Fig. 2 and Table 1, the effort for the individual components of the model can vary significantly based on the hardware configuration.

To mitigate these problems and to simplify the practical use of the model, we have developed an optimization framework that takes care of finding the optimal pipeline composition for a given hardware configuration. To use the framework, a developer specifies the processing pipeline together with its input parameters. The framework then measures and compares the processing time of the specified CPU functions and their GPU counterparts. Additionally, the framework tracks necessary uploads and downloads of parameters and evaluates them by running benchmarks. This information is used to build a graph, where every node represents a single test result. The optimal combination of CPU and GPU functions is determined by finding the shortest path through the graph.

Code 1.1. Example Code for Defining a Pipeline of OpenCV Functions

```

1 TestUtils<Mat> utils = new TestUtils<>();
2 utils.setCommandPipeline(mat -> {
3     Mat image = mat.get();
4     Mat edges = new Mat();
5     Imgproc.cvtColor(image, image,
6         Imgproc.COLOR_BGR2GRAY);
7     Imgproc.medianBlur(image, image, 15);
8     Imgproc.Canny(image, edges, 15, 45);
9 });
10 utils.setInput(new MatInput(theImage));
11 utils.initializeCommands(true);
12 utils.warmup(20);
13 utils.runTests(25);
14 utils.evaluate();

```

Code 1.1 shows exemplary usage of our framework. First, a `TestUtils` object has to be instantiated. The type argument specifies the type of the input parameter. The method `setCommandPipeline(Consumer<Input<T>>)` defines the pipeline which shall be evaluated. Inside the lambda expression, normal OpenCV function calls are made, except that the import statement should be changed. We implemented custom classes and methods matching the signatures

of common OpenCV functions, but instead of computing something, they track invocations.

The remaining interaction is simple. The method `setInput(Input<T>)` sets the input parameter used for the command pipeline. Useful implementations for a single `Mat`, a 2-tuple of `Mats`, or an array of `Mats` are available, while custom implementations can be added. The method `initializeCommands(boolean)` initializes the pipeline and tracks method invocations. The boolean flag indicates whether all possible combinations of functions or only an efficient subset of them should be evaluated. It is likely that GPU functions are faster than their CPU counterparts. If the flag is set to false, the framework only compares paths with as few uploads and downloads as possible, which accelerates the testing time. Methods `warmup(int)` and `runTests(int)` are used to set the number of iterations, which are used to warm up the system and test every function, respectively. The method `evaluate()` starts the actual evaluation.

For this specific pipeline, 8 paths have to be evaluated. There are 3 different functions, each of them can be executed as a CPU or GPU function, which results in 8 different combinations. If the flag in `initializeCommands(boolean)` is set to `false`, only 2 paths (all functions only run on CPU or GPU) will be evaluated, which can significantly speed up the evaluation. The resulting graph is shown in Fig. 5. Rounded rectangles represent a single step in the pipeline, either as a CPU function or a GPU function. Circles represent the `Mat` parameters. The transition between CPU functions can be executed immediately. Paths switching between CPU and GPU functions need to upload or download the respective parameters, which increases the processing time. Uploading and downloading transitions are indicated by red or blue arrows, respectively. Based on previous experiments the fastest path would be either CPU-only execution or GPU-only execution, depending on the resolution of the input image.

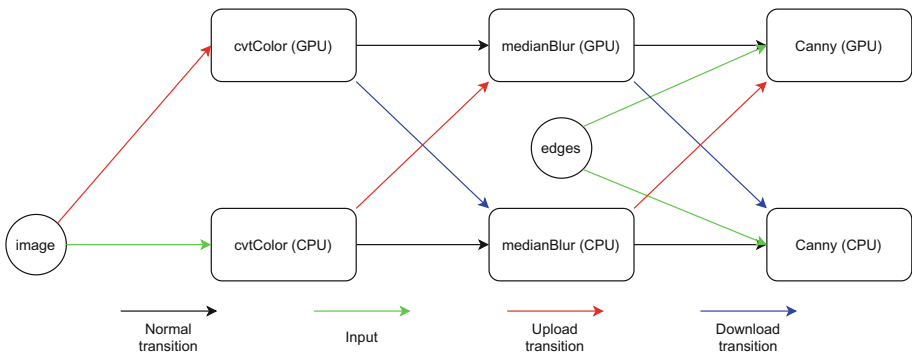


Fig. 5. The graph corresponding to the pipeline shown in Code 1.1

Internally, functions register implementations of the interface `Task` in a shared list. Every task can be converted to a `CpuTask` or a `GpuTask`, which will run the

corresponding OpenCV function on the CPU or GPU, accordingly. A description is used to describe the task in the final output. The list of `Tasks` can be traversed to evaluate every combination of CPU and GPU functions. Necessary uploads and downloads of `Mats` are tracked and connected with the functions where they occurred.

The framework includes a range of implementations of the `Task` interface for a representative subset of the OpenCV functions, including thresholding, color conversion, resizing, morphology operations and some image filters. Custom implementations can be used to test any other function. If a GPU implementation is not available for a given function, the framework only tests the CPU implementation and automatically detects which `Mats` have to be downloaded.

Given the available task implementations, the extensible nature of the framework, and the algorithms to compute the optimal pipeline composition, we are convinced that the framework can serve as a solid basis to support the systematic optimization of image processing pipelines.

5 Related Work

There have been many different attempts to accelerate sophisticated algorithms by using hardware suited for parallel programming, such as GPUs or FPGAs. For instance, the performance of different random number generation algorithms [20] or image processing algorithms [1] was compared on such hardware. Two other algorithms, the push-relabel algorithm [13] and the “Vector Coherence Mapping” algorithm [14] were implemented using CUDA. Compared to equivalent CPU implementations, both are substantially faster, the former 15, the latter 22 times.

A micro-benchmark suite for OpenCL is presented in [22]. OpenCL is a vendor independent framework for computing on heterogeneous platforms. The authors compare the performance of different GPUs and CPUs in regards to the presented micro-benchmarks.

OpenCV includes modules supporting the usage of general-purpose computing on graphics processing units (GPGPUs), which are already used by previous research. The authors of [9] explain the theoretical background of many tasks related to computer vision. They also give an introduction into OpenCV’s GPU module and its performance. In addition to that, the authors of [16] evaluated different functions of OpenCV and compared their processing time if run on the CPU or on the GPU. Another performance comparison between OpenCV’s CPU and GPU functions is presented in [6]. The authors compare the processing time of different common OpenCV functions with varying image sizes.

While both papers give an insight on the performance gain when using the GPU module, they don’t provide much information on uploading and downloading the data. The authors of [16] mention that programmers need to copy data between CPU and GPU and also explain some design considerations, but don’t quantify the time needed to do so.

This paper not only quantifies the time needed to upload and download images, but also presents a mathematical model to estimate the execution time

of any image processing pipeline. As shown, it is not sufficient to compare the processing time of functions themselves. Instead, it is necessary to take uploading and downloading of the data into consideration. Additionally, image processing functions are seldom executed in isolation. As more functions are added to the pipeline, the underlying model gains complexity, handled by the optimization framework. Developers can use it to compute the optimal combination of CPU and GPU functions.

6 Conclusion and Future Work

Computer vision systems require fast image processing pipelines. One way to reduce the execution time is to leverage the parallelism of modern GPUs to speed up individual image processing functions. However, since not all functions can benefit from a parallel execution and due to the fact that transitions between GPU and CPU code introduce overhead for uploading and downloading, optimizing the performance of image processing pipelines requires a careful analysis.

In this paper, we introduced a mathematical model that captures the relevant relationships as basis for the systematic optimization of image processing pipelines. Using micro-benchmarks collected with OpenCV, we analyzed different classes of image processing functions. The measurements show that not all of them will benefit equally. In addition, for simple filtering functions, functions with a sub-optimal implementation or applications operating on low resolution images, moving the computation from CPU to GPU can even increase the total execution time. As indicated by our validation, it is essential to account for the upload and download time when estimating the time required to execute a particular pipeline composition, since negligence can easily yield an estimation error that exceeds a factor of two. We hope that the model, measurements and optimization framework presented in this paper will help developers to find the optimal configuration for their application.

At the present time, we are currently analyzing the effects of asynchronous GPU calls which are supported by the class `cv::cuda::Stream`. Asynchronous calls can potentially increase the parallelism. However, when using streams for asynchronous calls it is necessary to allocate matrices in page-locked memory. In addition, some operations cannot be parallelized in all cases.

References

1. Asano, S., Maruyama, T., Yamaguchi, Y.: Performance comparison of FPGA, GPU and CPU in image processing. In: 2009 FPL, pp. 126–131, August 2009
2. Beymer, D., McLauchlan, P., Coifman, B., Malik, J.: A real-time computer vision system for measuring traffic parameters. In: Proceedings of IEEE CVPR, pp. 495–501, June 1997
3. Coifman, B., Beymer, D., McLauchlan, P., Malik, J.: A real-time computer vision system for vehicle tracking and traffic surveillance. *Transp. Res. Part C Emerg. Technol.* **6**(4), 271–288 (1998)

4. Gil, J., Werman, M.: Computing 2-d min, median, and max filters. *IEEE PAMI* **15**(5), 504–507 (1993)
5. Griffin, G., Holub, A., Perona, P.: Caltech-256 object category dataset (2007, unpublished). <https://resolver.caltech.edu/CaltechAUTHORS:CNS-TR-2007-001>
6. Hangün, B., Eyecioglu, Ö.: Performance comparison between OpenCV built in CPU and GPU functions on image processing operations. *IJESA* **1**, 34–41 (2017)
7. van Herk, M.: A fast algorithm for local minimum and maximum filters on rectangular and octagonal kernels. *Pattern Recognit. Lett.* **13**(7), 517–521 (1992)
8. Kurzyniec, D., Sunderam, V.: Efficient cooperation between Java and native codes - JNI performance benchmark. In: 2001 PDPTA (2001)
9. Marengoni, M., Stringhini, D.: High level computer vision using OpenCV. In: 2011 24th SIBGRAPI Conference on Graphics, Patterns, and Images Tutorials, pp. 11–24, August 2011
10. NVIDIA Corporation: CUDA C++ programming guide (2020). <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed 22 Sept 2020
11. Oliver, N.M., Rosario, B., Pentland, A.P.: A Bayesian computer vision system for modeling human interactions. *IEEE PAMI* **22**(8), 831–843 (2000)
12. OpenCV team: OpenCV (2020). <https://opencv.org/>. Accessed 28 Feb 2020
13. Park, S.I., Ponce, S.P., Huang, J., Cao, Y., Quek, F.: Low-cost, high-speed computer vision using NVIDIA's CUDA architecture. In: 2008 37th IEEE AIPR Workshop, pp. 1–7, October 2008
14. Park, S.I., Ponce, S.P., Huang, J., Cao, Y., Quek, F.: Low-cost, high-speed computer vision using NVIDIA's CUDA architecture. In: 2008 37th IEEE AIPR Workshop, pp. 1–7, October 2008
15. Phipps, G.: Comparing observed bug and productivity rates for Java and C++. *Softw. Pract. Exper.* **29**(4), 345–358 (1999)
16. Pulli, K., Baksheev, A., Korniyakov, K., Eruhimov, V.: Real-time computer vision with OpenCV. *Commun. ACM* **55**(6), 61–69 (2012)
17. Samuel Audet: Java interface to OpenCV, FFmpeg, and more (2020). <https://github.com/bytedeco/javacv>. Accessed 28 Feb 2020
18. Scharstein, D., et al.: High-resolution stereo datasets with subpixel-accurate ground truth. In: Jiang, X., Hornegger, J., Koch, R. (eds.) *GCPR 2014*. LNCS, vol. 8753, pp. 31–42. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11752-2_3
19. Shehu, V., Dika, A.: Using real time computer vision algorithms in automatic attendance management systems. *Proc. ITI* **2010**, 397–402 (2010)
20. Thomas, D.B., Howes, L., Luk, W.: A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In: *Proceedings of the ACM/SIGDA FPGA, FPGA 2009*, pp. 63–72. Association for Computing Machinery, New York (2009)
21. Thurley, M.J., Danell, V.: Fast morphological image processing open-source extensions for GPU processing with CUDA. *IEEE JSTSP* **6**(7), 849–855 (2012)
22. Yan, X., Shi, X., Wang, L., Yang, H.: An OpenCL micro-benchmark suite for GPUs and CPUs. *J. Supercomput.* **69**(2), 693–713 (2014)