ОПТИМИЗАЦИЯ ПРОЦЕССА ОБНАРУЖЕНИЯ ФИДУЦИАРНЫХ МАРКЕРОВ С ПОМОЩЬЮ ВЫЧИСЛЕНИЙ НА GPU

OPTIMIZATION OF FIDUCIAL MARKER DETECTION WITH GPU COMPUTATIONS



УДК 004.93'12

DOI: 10.24411/2658-4964-2021-103681

Бугрышев Роман Игоревич, студент магистратуры 1 курс, факультет «Программная инженерия и компьютерная техника», Университет ИТМО, Россия, г. Санкт-Петербург

Григорьева Анна Дмитриевна, студент магистратуры 1 курс, факультет «Программная инженерия и компьютерная техника», Университет ИТМО, Россия, г. Санкт-Петербург

Bugryshev Roman Igorevich, koteotakepe@gmail.com

Grigorieva Anna Dmitrievna

Аннотация

Данная статья посвящена вопросу оптимизации производительности обнаружения фидуциарных маркеров ArUco, достигаемой с помощью переноса части вычислений на GPU (Graphics Processing Unit). В ходе исследования был проанализирован алгоритм определения позиционных меток ArUco и его реализация в пакете OpenCV, выявлены фрагменты логики алгоритма, производительность которых от выполнения на GPU потенциально возрастает; рассмотрены возможности, предоставляемые стандартным модулем OpenCV gpu, для ускорения обработки изображений методом GPGPU. Результатом исследования является набор предложений по улучшению производительности

функционала модуля aruco, достигаемым при использовании графического процессора в части вычислений.

S u m m a r y

The article aims to study different ways to improve ArUco marker detection performance by moving some of the algorithm calculations to the GPU (Graphics Processing Unit). During the research an ArUco fiducial marker detection algorithm was thoroughly examined, as well as its implementation included in OpenCV computer vision library. This research also studied which parts of ArUco algorithm implementation could possibly benefit from utilizing GPU calculations and various methods provided by OpenCV to use GPGPU calculations in library modules. The product of this research is a set of proposals to enhance OpenCV aruco module performance by partially moving marker detection logic to the GPU.

Ключевые слова: компьютерное зрение, OpenCV, фидуциарные маркеры, ArUco, GPU, GPGPU, оптимизация.

Keywords: computer vision, OpenCV, fiducial markers, ArUco, GPU, GPGPU, optimization.

Введение

В настоящее время, технология фидуциарных маркеров применяется во многих сферах деятельности, подразумевающих взаимодействие пользователя с виртуальным трехмерным пространством: маркеры применяются в медицине, архитектуре, трехмерном моделировании, визуализации данных, автоматизации производства [1]. Большинство из существующих применений данной технологии опираются на работу в реальном времени, в связи с чем производительность и точность обнаружения фидуциарных меток являются одними из ключевых характеристик конкретной реализации алгоритма определения позиции маркеров.

Распространенным способом реализации функций компьютерного зрения (в том числе и поиска фидуциарных меток) в современных приложениях является использование функционала библиотеки OpenCV [2]. В качестве

реализации алгоритма поиска фидуциарных маркеров OpenCV использует метод, описанный в работе [3], далее называемый маркерами ArUco или метками ArUco. Данный метод нацелен на решение ряда существующих в других реализациях проблем (поддержка пользовательских наборов меток, отображение виртуальных объектов с учетом перекрытия их реальными), кроме того, реализация обнаружения маркеров ArUco более производительна, чем существующие реализации поиска иных квадратных фидуциарных меток (AprilTag, ARToolKitPlus, ChiliTags) [4].

Существует множество способов оптимизации производительности алгоритмов компьютерного зрения, одним из которых является частичный перенос вычислений, традиционно выполняемых компьютером на центральном процессоре (CPU, Central Processing Unit) на графический процессор (GPU, Graphics Processing Unit). Особенности архитектуры графических ускорителей, главным образом выражающиеся наличием большего количества ядер и векторизацией вычислений, позволяют эффективно производить с помощью GPU большое количество параллельных вычислений, что позволяет заметно ускорить обработку входного потока изображений, если шаги обработки входного изображения могут быть подвергнуты параллелизации [5].

Современные библиотеки компьютерного зрения активно используют технику GPGPU (General-purpose computing on graphics processing units, неспециализированные вычисления на графических процессорах), позволяющую переносить часть этапов анализа исходных изображений на GPU для повышения производительности обработки данных. Так, в OpenCV существует программный модуль gpu, содержащий набор высокоуровневых функций, позволяющих производить операции над стандартными типами данных библиотеки с использованием графических процессоров фирмы Nvidia [2]. Это позволяет быстро модифицировать новые и существующие алгоритмы компьютерного зрения для использования GPGPU, что в большинстве случаев дает заметный прирост к производительности работы приложения.

Так, в результате исследования [6] авторами был выявлен значительный прирост производительности (снижение времени обработки входных данных от 1.8 до 5 раз) при выполнении операций изменения размера, определения пороговых значений пикселей, поиска границ объектов изображения с использованием мощностей GPU. Также, авторы исследования обнаружили увеличение прироста производительности от переноса вычислений на GPU с ростом разрешения исходного изображения (рисунок 1).

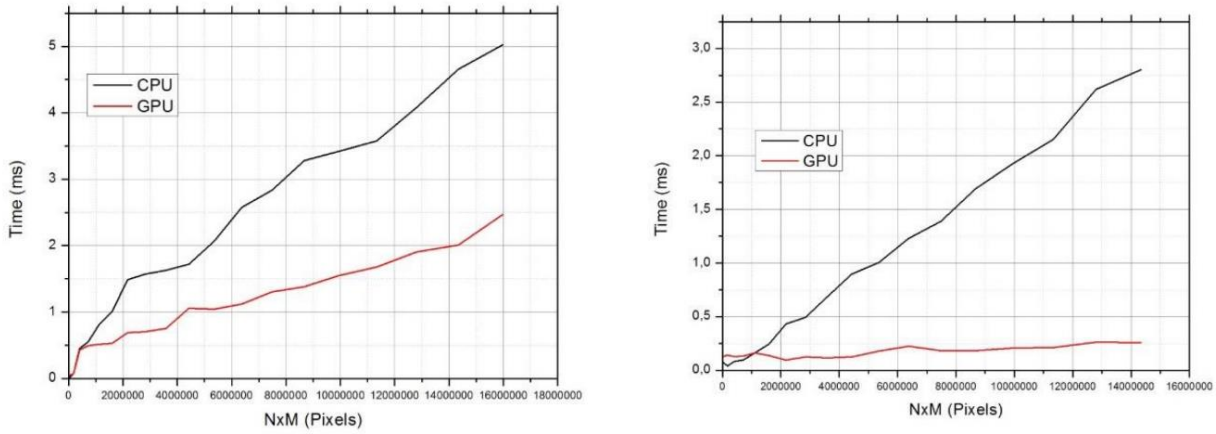


Рисунок 1. Время выполнения операций изменения размера изображения и бинаризации в зависимости от размера входных данных.

Официальный сайт OpenCV также предоставляет аналитические данные, указывающие на значительный рост быстродействия определенного набора алгоритмов при переносе части программной логики на GPU. На рисунке 2 представлено сравнение производительности CPU Intel Core i5-760 с GPU Nvidia Tesla C2050 в ряде задач, включая базовую обработку изображений, распознавание лиц и людей [7].

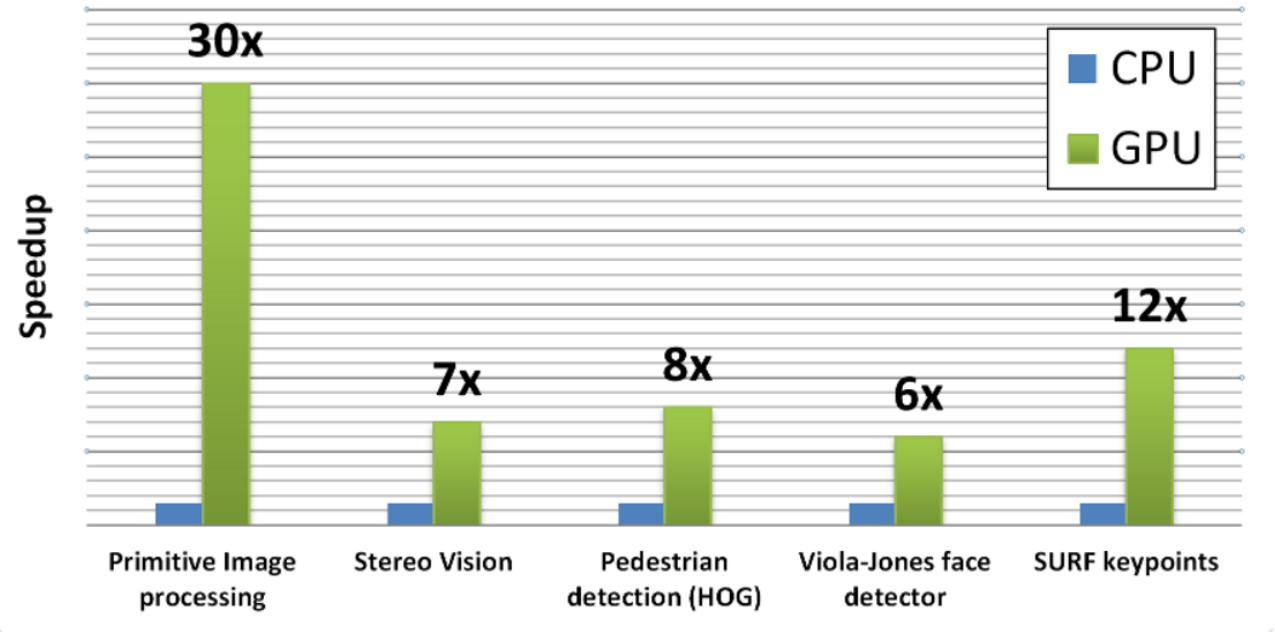


Рисунок 2. Сравнение производительности выполнения ряда операций на Intel Core i5-760 и Nvidia Tesla C2050.

Однако, несмотря на кажущееся превосходство GPU над CPU в задачах обработки изображений, в связи с различиями архитектур центральных и графических процессоров метод GPGPU может быть применен лишь для ограниченного набора операций над изображениями. Кроме того, при сравнительно небольшом объеме исходной информации, её обработка на центральном процессоре может оказаться более производительной из-за накладных расходов на пересылку данных в или из памяти графического процессора.

Таким образом, основными задачами данного исследования являются анализ этапов реализации алгоритма определения позиции фидуциарных маркеров ArUco в OpenCV, а также определение возможности и целесообразности переноса части этих этапов на GPU средствами, предоставляемыми в поставке OpenCV.

Анализ алгоритма определения позиции фидуциарных маркеров ArUco

В данной части работы будет произведен анализ версии алгоритма поиска и определения позиций маркеров ArUco, реализованной в библиотеки OpenCV, для выявления участков кода, где использование GPGPU может принести прирост производительности, а также определения используемых в работе стандартных функций OpenCV.

Как было сказано ранее, особенностью реализации фидуциарных маркеров ArUco является возможность создания пользовательских наборов меток, что повышает применимость технологии в различных практических приложениях. Однако, генерация набора меток, как правило, производится лишь один раз, на этапе конфигурации производимого программного продукта, в связи с чем данный процесс не будет рассматриваться как кандидат на оптимизацию с помощью подхода GPGPU [4].

Первым этапом обнаружения позиций маркеров ArUco во входящей видеопоследовательности является перевод исходного изображения в градации серого и применение к результату операции определения пороговых значений для каждой из его точек (thresholding). Для перевода изображения в градации серого используется стандартная функция OpenCV cv::cvtColor; для выполнения операции вычисления граничных значений алгоритм использует функцию cv::adaptiveThreshold, вычисляющей пороговое значение точки с учетом оригинальных значений цвета окружающих точек [3].

Далее, в полученном изображении необходимо обнаружить контуры маркеров-кандидатов (кандидатом для маркера считается любой замкнутый четырехугольный контур). Сначала производится обнаружение контуров всех объектов, присутствующих на изображении с использованием метода, описанного в исследовании [8] с помощью встроенной функции cv::findContours. Затем, из множества полученных контуров необходимо сформировать подмножество замкнутых контуров прямоугольной формы. Для этого используется полигональная аппроксимация полученных на предыдущем этапе контуров изображения (с помощью cv::approxPolyDB), после чего становится

возможным проверка количества углов контура, замкнутости контура и расстояния между углами контуров.

Визуализация описанных выше шагов обработки изображения представлена на рисунке 3.

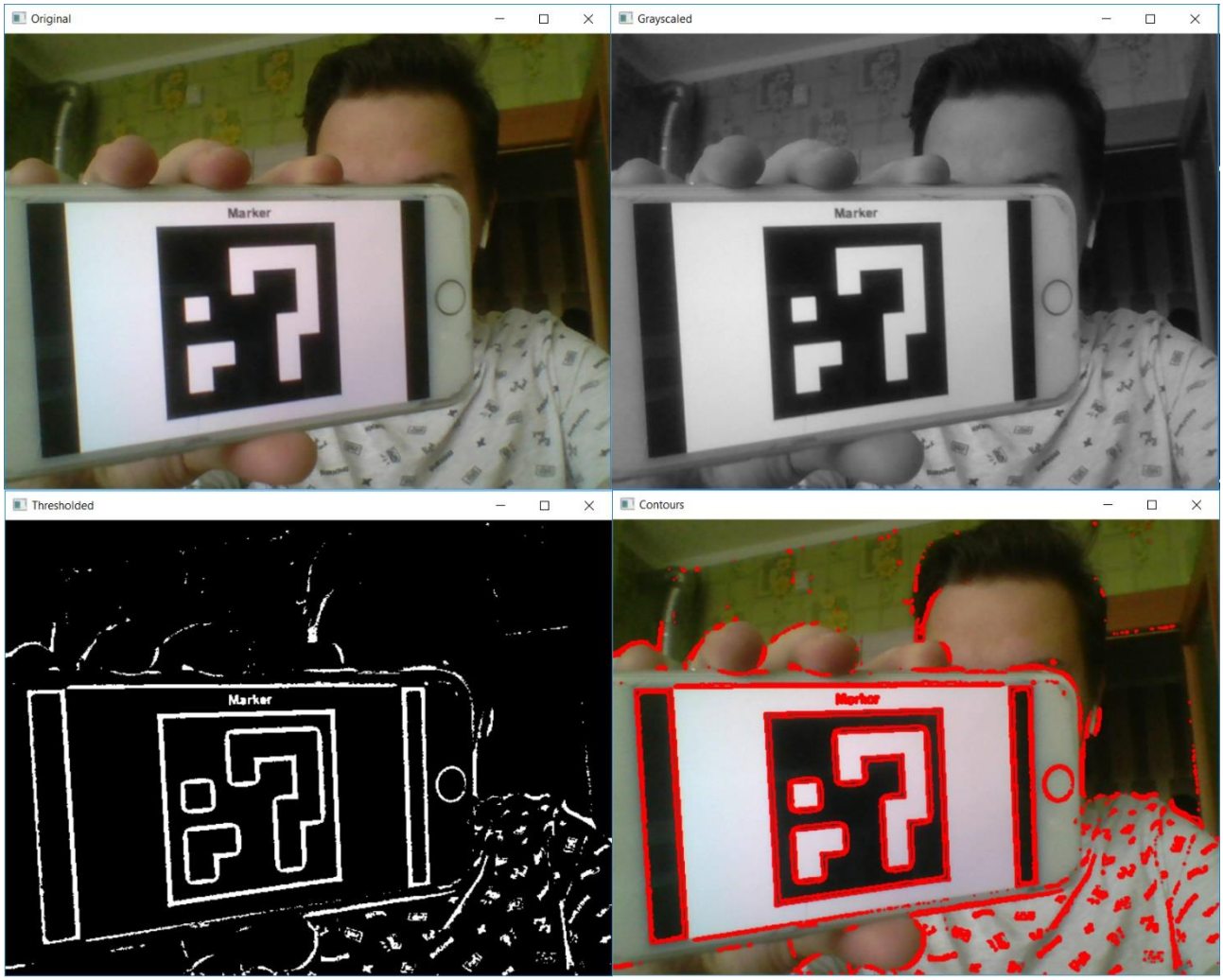


Рисунок 3. Визуализация определения контуров изображения с использованием функционала библиотеки OpenCV.

Следующий этап обработки предполагает выделение из полученных маркеров-кандидатов внутреннего графического кода, позволяющего определить принадлежность кандидата к конкретному словарю маркеров ArUco. Для этого, фрагменты исходного изображения, ограниченные контурами, преобразуются к “виду сверху” с помощью вычисления матрицы гомографии (cv::getPerspectiveTransform) и применения встроенной функции cv::warpPerspective. Затем, над преобразованным изображением производится

бинаризация изображения методом Оцу (cv::threshold), при условии, что область внутри границ маркера-кандидата не обладает однородным цветом. Далее, бинаризированное изображение делится на заранее определенное используемым набором меток количество секторов, по цветовым значениям которых определяется уникальный идентификатор обнаруженной метки [3].

Последним этапом обработки является определение пространственной позиции маркеров (pose estimation) с предварительным уточнением координат краев ранее идентифицированных меток. Для уточнения координат используется функция cv::cornerSubPix, реализующая метод, описанный в исследовании [9]. Для определения позиции обнаруженных маркеров по их двухмерным координатам в кадре используется cv::solvePnP с методом SOLVEPNP_ITERATIVE [3].

Возможные варианты оптимизации алгоритма ArUco с применением вычислений на GPU

Рассмотрев принципы реализации алгоритма поиска фидуциарных маркеров ArUco, можно определить два основных направления оптимизации разработки с использованием методов GPGPU: адаптация фрагментов кода, написанных авторами модуля ArUco для исполнения на графическом процессоре, и замена используемых в коде модуля функций библиотеки OpenCV на их реализации, исполняющиеся на GPU.

Для выбора приоритетного направления в первую очередь должны быть учтены особенности архитектуры современных графических процессоров. Несмотря на крайне высокую производительность при выполнении векторизированных операций, присутствуют значительные накладные расходы на пересылку исходных данных в память видеокарты и загрузку результата операции обратно в основную оперативную память. Таким образом, затраты на пересылку данных в памяти могут нивелировать прирост производительности от ускоренной параллельной обработки данных на GPU [5].

При анализе исходного кода модуля aruco было выявлено, что наиболее вычислительно сложными и хорошо распараллеливаемыми операциями

являются трансформации цвета и перспективы исходного изображения, выполняемые с помощью стандартных реализаций функций библиотеки OpenCV. Также, большинство “тяжелых” вычислений, выполняемых кодом модуля aruco, обладают большим количеством ветвлений, что значительно понизит производительность их параллельного выполнения на GPU без предварительной переработки исходного кода.

Из чего следует, что наиболее целесообразным решением на первом этапе оптимизации алгоритма поиска фидуциарных маркеров ArUco будет внедрение существующих GPGPU-реализаций стандартных функций OpenCV в базовую реализацию алгоритма поиска меток. Таблица 1 содержит сводку об использовании функционала библиотеки OpenCV в модуле aruco, роли используемых функций и краткое текстовое описание входных данных, передаваемых на обработку в конкретную функцию.

Таблица 1. Использование встроенных функций OpenCV модулем aruco.

Стандартная функция OpenCV	**Использование в алгоритме ArUco**	**GPGPU-вариант**	**Входные данные**
cv::cvtColor	Перевод в градации серого исходного изображения	cv::cuda::cvtColor	Исходный кадр
cv::adaptiveThreshold	Бинаризация исходного изображения	cv::cuda::threshold	Исходный кадр
cv::findContours	Поиск границ объектов в исходном кадре и выделение полигональных кривых	-	Исходный кадр
cv::approxPolyDB		-	Множество контуров, полученное на пред. шаге
cv::getPerspectiveTransform		-	Фрагмент исходного кадра
cv::warpPerspective		cv::cuda::warpPerspective	Фрагмент исходного кадра
cv::threshold	Бинаризация метки для извлечения ее кода	cv::cuda::threshold	Фрагмент исходного кадра
cv::cornerSubPix	Уточнение координат границ метки	-	Фрагмент исходного кадра
cv::solvePnP	Определение пространственной позиции маркера.	cv::cuda::solvePnPRansac	Фрагмент исходного кадра

Необходимо предоставить некоторые пояснения касаемо содержания таблицы 1.

Курсивом в колонке “GPGPU-вариант” обозначены функции из пакета gpu, не являющиеся прямыми аналогами к оригинальным функциям OpenCV, но выполняющие ту же задачу с использованием альтернативных методов. Соответственно, замена таких функций на GPGPU-варианты может привести к

разнице в результатах, получаемых при обработке исходных данных оригинальным алгоритмом и его ускоренным вариантом.

Очевидно, что наиболее перспективными функциями для оптимизации являются функции, обрабатывающие наибольшее количество исходной информации. Такими являются первые 3 кандидата: cv::cvtColor, cv::adaptiveThreshold и cv::findContours; для первых двух функций из списка в стандартном пакете существуют GPGPU-реализации.

Функция cv::cvtColor имеет прямую альтернативу cv::cuda::cvtColor, которая может быть использована в коде модуля aruco без значительных изменений кодовой базы. В то же время, функция cv::adaptiveThreshold не имеет прямого аналога в модуле gpu на данный момент. Обратившись к исследованию [3], можно заметить, что ранние версии алгоритма обнаружения меток ArUco использовали для поиска контура метод, описанный в работе [10] и реализованный в стандартной библиотеке OpenCV функцией cv::Canny, имеющей GPGPU-аналог в виде функционала класса cv::cuda::CannyEdgeDetector. Однако, авторы библиотеки отказались от данного решения из-за недостаточно высокой производительности метода, негативно сказывавшейся на обнаружение маркеров в реальном времени. В связи с чем, перед исследованием открывается возможность замены существующего метода поиска границ объектов в исходном изображении на ускоренную версию детектора границ Кэнни, с последующим анализом изменения общей производительности алгоритма.

Стоит отметить, что несмотря на наличие GPGPU-аналогов у многих из остальных используемых в aruco функций, от их использования не ожидается положительных изменений производительности алгоритма в связи с малым объемом обрабатываемых данных, и, соответственно, высоким влиянием накладных расходов на пересылку данных в памяти на быстродействие обработки информации.

При внедрении в код модуля aruco GPGPU-вариантов описанных выше функций будет целесообразно реализовать выбор между CPU и GPU –

вариантами функционала OpenCV в зависимости от разрешения исходных кадров изображения для решения ранее описанной проблемы, возникающей при обработке на графическом процессоре малого количества информации.

В последующих этапах исследования, рассмотренные выше оптимизации будут внедрены в модуль aruco, после чего будет проведено профилирование и выявлено влияние предлагаемых оптимизаций на производительность алгоритма при работе с различными наборами входных данных. При практическом подтверждении эффективности примененных мер модификации алгоритма будут внесены в исходный код OpenCV в рамках развития модуля aruco

Литература

1. Fiala M. Designing Highly Reliable Fiducial Markers // IEEE Transactions on Pattern Analysis and Machine Intelligence. 2010. № 7 (32). С. 1317–1324.
2. Pulli K. [и др.]. Real-time computer vision with OpenCV // Communications of the ACM. 2012. № 6 (55). С. 61–69.
3. Garrido-Jurado S. [и др.]. Automatic generation and detection of highly reliable fiducial markers under occlusion // Pattern Recognition. 2014. № 6 (47). С. 2280–2292.
4. Romero-Ramirez F. J., Muñoz-Salinas R., Medina-Carnicer R. Speeded up detection of squared fiducial markers // Image and Vision Computing. 2018. (76). С. 38–47.
5. Misic M., Durdevic D., Tomasevic M. Evolution and trends in GPU computing // Proceedings of the 35th International Convention MIPRO. 2012. С. 289-294.
6. Hangun B., Eyecioglu O. Performance Comparison Between OpenCV Built in CPU and GPU Functions on Image Processing Operations // International Journal of Engineering Science and Application. 2017. № 2 (1). С 34-41.
7. OpenCV CUDA Platform. [Электронный ресурс]. URL: https://opencv.org/platforms/cuda/ (дата обращения: 08.07.2021).
8. Suzuki S., Abe K. Topological structural analysis of digitized binary images by border following // Computer Vision, Graphics, and Image Processing. 1985. № 1 (30). С. 32–46.

9. Forstner, W., Gulch E. A Fast Operator for Detection and Precise Location of Distinct Points, Corners and Centers of Circular Features. Proceedings of the ISPRS Intercommission Conference on Fast Processing of Phonogrammic Data. 1987. № 1. С. 281-305.
10. Canny J. A Computational Approach to Edge Detection // IEEE Transactions on Pattern Analysis and Machine Intelligence. 1986. № 6 (PAMI-8). С. 679–698.