2. Стронгин Р.Г. Поиск глобального оптимума. – М.: Знание, 1990.

3. Стронгин Р.Г. Параллельная многоэкстремальная оптимизация с использованием множества разверток // Вычисл. матем. и матем. физ. – 1991. – Т.31. № 8. – С. 1173–1185.

4. Стронгин Р.Г., Баркалов К.А. О сходимости индексного алгоритма в задачах условной оптимизации с ε-резервированными решениями // Математические вопросы кибернетики. – М.: Наука, 1999. – С. 273–288.

5. Strongin R.G., Sergeyev Ya.D. Global optimization with non-convex constraints. Sequential and parallel algorithms. Kluwer Academic Publishers, Dordrecht, 2000.

6. Баркалов К.А. Ускорение сходимости в задачах условной глобальной оптимизации. – Н. Новгород: Изд-во Нижегород. гос. ун-та, 2005.

7. Баркалов К.А., Сидоров С.В., Рябов В.В. Параллельные вычисления в задачах многоэкстремальной оптимизации // Вестн. ННГУ. Математическое моделирование и оптимальное управление. – Н. Новгород: Изд-во Нижегород. гос. ун-та. – № 6(1), 2009. – С. 171–177.

**В.А. Сапрыкин**

Нижегородский государственный университет им. Н.И. Лобачевского

## РЕАЛИЗАЦИИ АДАПТИВНОГО АЛГОРИТМА ВЫЧИТАНИЯ ФОНА: ПОСЛЕДОВАТЕЛЬНАЯ, ПАРАЛЛЕЛЬНАЯ, OpenCL-РЕАЛИЗАЦИЯ ДЛЯ GPU

С задачи отделения фона от переднего плана в видеопотоке начинаются многие алгоритмы машинного зрения [3]. Адаптивный алгоритм вычитания фона [1, 2], использующий смесь гауссианов, способен достаточно эффективно сегментировать изображение в условиях изменчивого заднего плана. Однако его производительность на современных настольных компьютерах зачастую не позволяет работать в реальном времени. Цель на-

стоящей работы – исследовать три направления повышения производительности: за счет оптимизации последовательной версии алгоритма, распараллеливания и реализации на графическом процессоре. При этом рассматриваются только такие способы реализации, которые не требуют внесения значительных изменений в существующий программный код, реализующий последовательную версию алгоритма. Такой подход упрощает поддержку нескольких реализаций одного алгоритма для различных вычислительных конфигураций. Кроме того, он позволят нам распространить полученные результаты на ряд других алгоритмов обработки изображений в виде методики повышения производительности готовой реализации.

**Постановка задачи.** Пусть есть исходная последовательная реализация некоторого алгоритма, опирающаяся на базовые функции библиотеки OpenCV [3]. Пусть такая реализация удовлетворяет следующим ограничениям:

1. Входная $\{A_1, A_2, ..., A_p\}$ и выходная $\{B_1, B_2, ..., B_q\}$ информация представляет собой множества двумерных изображений.

2. Все обрабатываемые изображения (входные, выходные, промежуточные) имеют одинаковый размер.

3. Каждый пиксель обрабатывается единообразно и независимо от остальных $(\forall i = 1, ..., q)(\forall x, y)\ B_i^{x,y} = g_i\left(A_1^{x,y}, A_2^{x,y}, ..., A_p^{x,y}\right)$.

Требуется разработать методы повышения производительности такого алгоритма и опробовать их на примере адаптивного вычитания фона. В частности, требуется разработать инструмент, позволяющий на основе последовательности вызовов функций OpenCV автоматически генерировать и запускать программы для GPU.

**Схема данных алгоритма вычитания фона.** Каждый пиксель изображения моделируется *n*-мерной (по числу цветовых каналов) случайной величиной, распределение которой является смесью *k* (обычно от 3 до 5) нормальных распределений [1, 2].

$$f(x) = \sum_{i=0}^{k-1} w_i N(x, \mu_i, Q_i) = \sum_{i=0}^{k-1} w_i \frac{1}{\sqrt{(2\pi)^n |Q_i|}} e^{-\frac{1}{2}(x-\mu_i)^T Q_i^{-1}(x-\mu_i)}, \sum_{i=0}^{k-1} w_i = 1, \quad (1)$$

где $w_i$ – вес, $\mu_i$ – вектор математического ожидания, $Q_i$ – матрица ковариации $i$-го $n$-мерного распределения.

Предполагая независимость одномерных случайных величин [8] для каждого цветового канала, имеем

$$Q_i = diag\left(\sigma_{i,1}^2, \sigma_{i,2}^2, ..., \sigma_{i,n}^2\right). \quad (2)$$

Таким образом, для хранения текущих оценок параметров распределения (1) нам потребуется $3k$ изображений: $M_0, ..., M_{k-1}$ – для хранения оценок матожиданий, $D_0, ..., D_{k-1}$ – для хранения оценок дисперсий, $W_0, ..., W_{k-1}$ – для хранения весов (рис. 1).

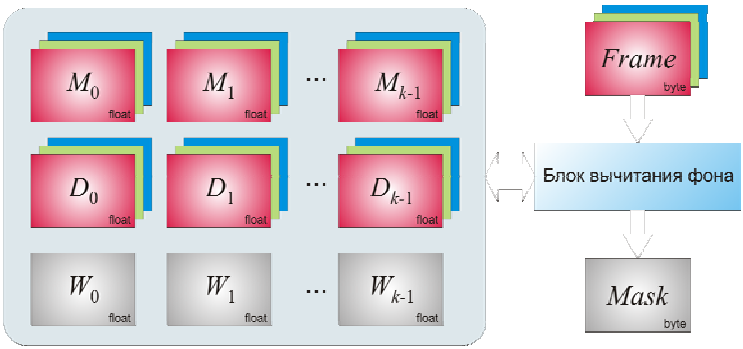


Рис. 1. Схема данных алгоритма вычитания фона

Время обработки алгоритмом каждого пикселя очередного кадра размера $a \times b$ постоянно. Оценим объем памяти, необходимый для работы алгоритма. Основная часть изображений хранится в формате с плавающей точкой, одинарной точной, по одному или по три цветовых канала. Маски хранятся в одноканальных изображениях с глубиной цвета 8 бит на пиксель. Последовательная реализация требует следующего объема памяти (байт) с учетом исходного, выходного и промежуточных изображений:

$$M = (29k + 60)ab. \quad (3)$$

**Автоматизация разрезания изображения на полосы.** В результате проведенных экспериментов в вычислительной системе на базе процессора Intel Core 2 Duo E8200 (2,6 ГГц, 6 Мб L2) было выявлено, что исходная последовательная реализация алгоритма, использующая смесь трех гауссианов, способна обрабатывать цветное изображение размером 640×480, поступающее с видеокамеры, со скоростью примерно 10,5 кадров/с.

Кэш-память процессора зачастую не в состоянии вместить целиком все основные и вспомогательные изображения, участвующие в обработке текущего кадра. Это дает возможность получения большей производительности, если обрабатывать изображение не целиком, а по частям.

Для автоматизации этого процесса был создан набор макросов, позволяющий легко превратить код, работающий с целыми изображениями, в код, работающий с полосой. Так, макрос CutImage (листинг 1) создает экземпляр структуры IplImage и подменяет указатель на целое изображение указателем на полосу этого изображения:

```
struct CStrip
{
  int fStartLine;
  int fEndLine;
};

#define CutImage(s, pImage, pSrtip) \
  IplImage Var_##pSrtip = *pImage;  \
  IplImage *pSrtip = &Var_##pSrtip; \
  Var_##pSrtip.imageData +=
s.fStartLine*Var_##pSrtip.widthStep; \
       Var_##pSrtip.height = s.fEndLine - s.fStartLine;
```

Есть макрос, делающий то же самое для массива изображений. Эти макросы можно использовать совместно с заключением нужного участка кода в составной оператор (блок), с тем чтобы получить возможность локально перекрывать имена исходных переменных. В итоге код последовательной реализации остается без изменений.

Минимальное количество полос, на которое следует разрезать изображение, можно попробовать оценить теоретически, воспользовавшись значением объема кэша процессора и формулой оценки требуемой памяти (3). Например, для описанного выше случая имеем $M = (29 \cdot 3 + 60)640 \cdot 480 \approx 50{,}9$Мб, что при объеме кэша в 6Мб говорит о необходимости разрезать изображение как минимум на 8–9 полос. Экспериментальные данные зависимости скорости обработки (кадров в секунду) от числа горизонтальных полос, на которые разбивается каждое изображение, подтверждают это значение (рис. 2). Кроме того, они позволяют оценить максимальное число полос, после превышения которого из-за возросших накладных расходов наступает некоторое снижение производительности. Таким образом, из графика легко видеть, что разбиение изображения на 10–30 горизонтальных полос позволяет достичь производительности 14,5 кадров в секунду.

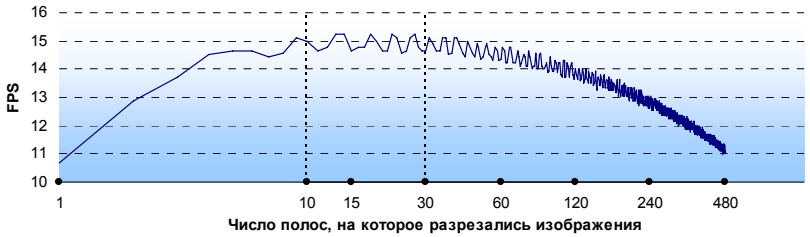


Рис. 2. Зависимость скорости обработки от числа полос

**Распараллеливание алгоритма.** Рассматриваемый алгоритм вычитания фона легко распараллеливается по данным. Наиболее удобно воспользоваться уже имеющимся у нас инструментом – разрезать очередной кадр на полосы и обрабатывать их параллельно на различных вычислительных ядрах. Из библиотеки Intel TBB [4] для этой цели удобно воспользоваться алгоритмом parallel_for, разбивая пространство номеров строк изображения. Существенно, что функтор, требуемый алгоритмом parallel_for, записывается довольно компактно вне зависимости от сложности алгоритма.

Размер порции данных (grain size) для parallel_for достаточно сильно влияет на итоговую производительность. Слишком большое значение скажется в худшую сторону на масштабируемости, а слишком маленькое – приведет к высоким накладным расходам [7]. Оптимальное значение можно выявить экспериментально. На следующем графике (рис. 3) в логарифмическом масштабе показана зависимость производительности (кадров в секунду) от величины grain size при распараллеливании вычислений на два потока. Таким образом, значение grain size из диапазона 10–60 дает возможность достичь производительности порядка 25 кадров в секунду.

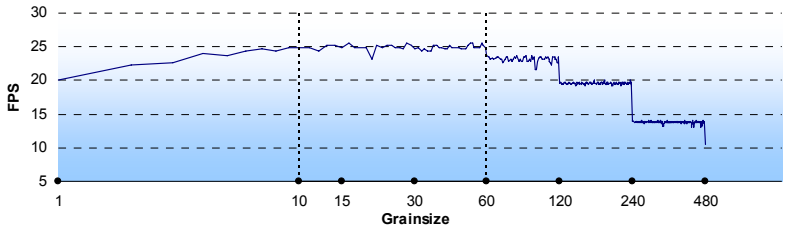


Рис. 3. Зависимость скорости обработки от величины grain size

Следует отметить, что библиотека TBB позволяет установить число создаваемых потоков в единицу. Тогда, указывая в качестве grain size полную высоту изображения, мы получим обыкновенную последовательную реализацию, а указывая меньшее значение – более производительную последовательную реализацию с разрезанием на полосы.

**Автоматизация реализации на GPU.** Для облегчения реализации алгоритма на графическом процессоре был создан следующий набор классов: CGpuContext, CVarList, CProgram. Класс CGpuContext является простой оберткой над функциями OpenCL, служащими для создания контекста и получения описателей устройств.

Класс CVarList позволяет сформировать список используемых в вычислении изображений. Для каждого изображения указывается тип памяти: частная/глобальная, при этом во втором случае в рамках контекста создаются соответствующие

объекты памяти (cl_mem). Все входные, выходные изображения и изображения, которые требуется хранить в памяти GPU между запусками kernel-а OpenCL, должны быть глобальными. Формирование списка происходит на основе указателей на структуры IplImage, описывающие каждое изображение. Причем обязательное выделение памяти под хранение данных требуется лишь для входных и выходных изображений.

За непосредственную генерацию текста OpenCL-программы отвечает класс CProgram. Он содержит ряд методов, сигнатуры которых совпадают с базовыми функциями OpenCV. В эти методы следует передавать те же указатели на IplImage, что использовались при формировании списка переменных. Ниже показан пример использования полученного набора классов для автоматической генерации OpenCL-кода, выполняющего взвешенное сложение изображений:

```
CGpuContext context(init);
CVarList vars(context);
vars.AddVar(A, atGlobalRead);
vars.AddVar(B, atGlobalRead);
vars.AddVar(C, atGlobalWrite);
vars.AddVar(Temp1, atPrivate);
vars.AddVar(Temp2, atPrivate);
vars.AddVar(Mask,
atGlobalRead);

CProgram prog1(vars);
prog1.BeginProgram("MyProg1");
prog1.cvScale(A,          Temp1,
alpha);
prog1.cvScale(B,   Temp2,   1.0-
alpha);
prog1.cvAdd(Temp1,   Temp2,   C,
Mask);
prog1.EndProgram();
```

```
__kernel void
MyProg1(
  __read_only
__global float4
*V0,
  __read_only
__global float4
*V1,
  __write_only
__global float4
*V2,
  __read_only
__global uchar
*V5)
{
  int id =
get_global_id(0);
  float4 V3;
  float4 V4;

  V3 = V0[id]*0.2;
//cvConvertScale
  V4 = V1[id]*0.8;
//cvConvertScale
  if (V5[id])
    V2[id] = V3+V4;
//cvAdd
}
```

В результате проделанной работы была создана методика повышения производительности адаптивного алгоритма вычитания фона, которая работает без внесения существенных изменений в исходный код реализации. На GPU удалось получить производительность в среднем 112 FPS (рис. 4) для цветного изображения 640×480 (Intel Core 2 Duo E8200 (2,6 ГГц, 6Мб L2), ОЗУ 2Гб, NVidia GeForce GTS 250). На рис. 4 также показаны частоты кадров, достигнутые исходной последовательной программой (1), последовательной программой с разрезанием изображения на полосы (1+) и параллельной программой, запущенной на двух потоках (2).

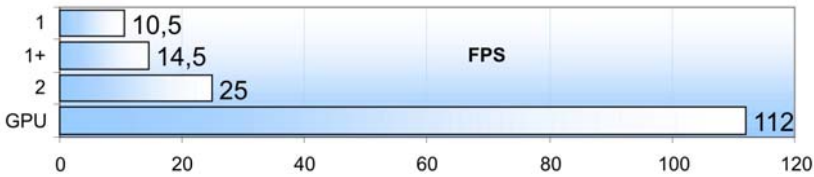


Рис. 4. Производительность при различных методах обработки изображений

В будущем планируется расширить сферу применимости созданных инструментов на более широкий класс алгоритмов. Для этого, в частности, потребуется научиться автоматически генерировать эффективный OpenCL-код на основе рекомендаций изложенных в статье [5].

Работа выполнена при финансовой поддержке ФЦП «Научные и научно-педагогические кадры инновационной России», госконтракт № 02.740.11.0839.

## Список литературы

1. Stauffer C., Grimson W. Learning Patterns of Activity Using Real-Time Tracking // IEEE Transactions on Pattern Analysis and Machine Intelligence. 2000. – Vol. 22, No. 8.

2. Power P.W., Schoonees J.A. Understanding background mixture models for foreground segmentation //Proceedings Image and Vision Computing New Zealand, 2002.

3. Bradski Gary, Kaehler Adrian Learning OpenCV. – O'Reilly, 2008.

4. Intel® Threading Building Blocks. Tutorial. Revision: 1.13. – Intel Corporation, 2007.

5. OpenCL Best Practices Guide. Version 1.0. – NVIDIA Corporation, 2009.

6. The OpenCL specification. Version: 1.0. – Khronos OpenCL Working Group, 2009.

7. Мееров И.Б., Сиднев А.А., Сысоев А.В. Библиотека Intel Threading Building Blocks – краткое описание. – Н. Новгород, 2007.

8. Федоткин М.А. Основы прикладной теории вероятностей и статистики. – М.: Высшая школа, 2006.

9. Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Вильямс, 2003.

**Б.Г. Севрюков, А.А. Лукьяница**

Московский государственный университет им. М.В. Ломоносова

## ИСПОЛЬЗОВАНИЕ GPU В ЗАДАЧАХ ТРЕХМЕРНОЙ РЕКОНСТРУКЦИИ СЦЕН

Задачи определения пространственной структуры реальных объектов или сцен по двумерным изображениям ставятся в разнообразных областях, таких как: компьютерное моделирование, навигация роботов по заранее неизвестной местности, кино и телевидение, медицина, архитектура. За последнее десятилетие методы решения этого класса задач получили существенное развитие, однако значительный объем вычислений, необходимый для получения качественной реконструкции, не позволяет использовать их в системах реального времени.

С другой стороны, вычислительная мощь современных видеокарт, невысокая цена и общая доступность делают их привлекательными для решения указанных задач. Поэтому нами было проведено исследование возможности переноса вычислений на GPU. Устройство, предназначенное для синтеза двумерных изображений, мы будем использовать для решения обрат-