СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 7](#_Toc262398106)

[**1** ОБЗОР ЛИТЕРАТУРЫ 9](#_Toc262398107)

[**2** ОПИСАНИЕ ХААРОВСКОГО КЛАССИФИКАТОРА 14](#_Toc262398108)

[**3** ОБЗОР АРХИТЕКТУРЫ CUDA 18](#_Toc262398109)

[**4** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 22](#_Toc262398110)

[**5** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 26](#_Toc262398111)

[**5.1** Описание конфигурации запуска ядер для GPU 26](#_Toc262398112)

[**5.2** Описание структуры и локальных переменных классификатора 26](#_Toc262398113)

[**5.3** Модуль инициализации данных и CUDA 29](#_Toc262398114)

[**5.4** Модуль обработки изображения на GPU 30](#_Toc262398115)

[**5.5** Модуль генератора кода 35](#_Toc262398116)

[**5.6** Модуль расчета оставшихся каскадов на CPU 35](#_Toc262398117)

[**6** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 37](#_Toc262398118)

[**6.1** Алгоритм работы классификатора 37](#_Toc262398119)

[**6.2** Алгоритм работы планировщика 37](#_Toc262398120)

[**6.3** Алгоритм работы обработчика каскада 39](#_Toc262398121)

[**6.4** Алгоритм работы преобразователя структуры классификатора 41](#_Toc262398122)

[**6.5** Алгоритм работы функции восстановления маски изображения 41](#_Toc262398123)

[**6.6** Инициализация CUDA 42](#_Toc262398124)

[**7** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ 43](#_Toc262398125)

[**8** РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 46](#_Toc262398126)

[**8.1** Требования к аппаратному и программному обеспечению 46](#_Toc262398127)

[**8.2** Руководство по работе с программой распознавания лиц 48](#_Toc262398128)

[**9** ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ПРОГРАММНОГО МОДУЛЯ ДЛЯ РЕАЛИЗАЦИИ ХААРОВСКОГО КЛАССИФИКАТОРА НА ПАРАЛЛЕЛЬНОЙ АРХИТЕКТУРЕ GPU 49](#_Toc262398129)

[**9.1** Характеристика проекта 49](#_Toc262398130)

[**9.2** Экономический эффект у разработчика 49](#_Toc262398131)

[**9.3** Оценка экономического эффекта у пользователя 57](#_Toc262398132)

[**10** ОХРАНА ТРУДА, ЭКОЛОГИЧЕСКАЯ БЕЗОПАСНОСТЬ, ЭНЕРГОСБЕРЕЖЕНИЕ 58](#_Toc262398133)

[**10.1** Энергосбережение при разработке и эксплуатации программного модуля для реализации Хааровского классификатора на параллельной архитектуре GPU 58](#_Toc262398134)

[ЗАКЛЮЧЕНИЕ 64](#_Toc262398135)

[СПИСОК ЛИТЕРАТУРЫ 65](#_Toc262398136)

[ПРИЛОЖЕНИЕ А. Исходный текст функции scan 67](#_Toc262398137)

[ПРИЛОЖЕНИЕ Б. Исходный текст функции DataToQueueB 69](#_Toc262398138)

[ПРИЛОЖЕНИЕ В. Исходный текст функции haar\_first\_stage 71](#_Toc262398139)

[ПРИЛОЖЕНИЕ Г. Исходный текст функции haar\_next\_stage 74](#_Toc262398140)

[ПРИЛОЖЕНИЕ Д. Исходный текст функции haar\_next\_stage\_4threads 79](#_Toc262398141)

[ПРИЛОЖЕНИЕ Е. Спецификация проекта 83](#_Toc262398142)

[ПРИЛОЖЕНИЕ Ж. Ведомость документов 84](#_Toc262398143)

# ВВЕДЕНИЕ

Впервые алгоритм Хааровского классификатора был описан Виола и Джонсом в своей статье [1]. После публикации он получил широкий отклик в мировой общественности и вызвал целый ряд исследований в этом направлении. В результате этого появились модификации и расширения данного алгоритма.

Уменьшение стоимости хранения одного гигабайта данных при значительном росте производительности системы в целом привело к массовому распространению видеоматериалов, записанных в высоком разрешении (High-Definition Video, HD). На сегодняшний день приняты два стандартных разрешения картинки, которые соответствуют HD, – 720p (1280х720 пикселей) и 1080p или 1080i (1920x1080 пикселей). Также на рынке появилось относительно дешевое оборудование, которое позволяет как воспроизводить видео поток в формате HD, так и записывать его. При этом пользователь получает значительно более качественную картинку. Все это способствует росту популярности и снижению стоимости видеокамер, позволяющих записывать в высоком разрешении.

Но существует сферы деятельности, в которых требуется не просто произвести запись, но также необходимо произвести обработку полученной картинки. Например, в таких местах как банки и аэропорта желательно не просто записать видео поток, но и осуществлять поиск лиц в нем. Найденные лица могут быть автоматически проанализированы с помощью баз данных и, например, система может предупредить службу безопасности о том, что данный человек, возможно, находится в розыске. Такая автоматизация позволяет значительно облегчить работу службы безопасности и повысить уровень безопасности.

Однако мощности современных центральных процессоров не достаточно, чтобы обеспечить даже обработку видео потока в высоком разрешении в реальном масштабе времени. Тем не менее, в современных персональных компьютерах остается незадействованным устройство, которое обладает значительной пиковой производительностью – графический процессор.

Целью данного дипломного проекта является разработка параллельного алгоритма Хааровского классификатора, который требуется для поиска лиц на цифровом изображении, с применением графического процессора. При этом будет предпринята попытка достигнуть времени обработки одного изображения, которое позволит применять алгоритм в системах обработки видео потока формата HD в реальном масштабе времени.

За основу была взята классическая реализация алгоритма Хааровского классификатора, которая входит в состав открытой библиотеки OpenCV[2]. Несмотря на использование высокоуровневых оптимизаций и поддержку современных многоядерных процессоров, существующая реализация из OpenCV не может справиться с поставленной задачей.

Графические процессоры (Graphics Processing Unit, GPU) за последние 5-7 лет превратились из специализированных устройств, предназначенных для построения трехмерных сцен, в универсальные вычислительные сопроцессоры. Архитектура современных GPU относится к классу SIMT[3] (Single Instruction Multiple Data) и обладает пиковой производительностью более 1 TFlops[4], то есть более операций с плавающей точкой в секунду. Класс SIMT означает, что все вычислительные блоки в рамках одного мультипроцессора исполняют одну и ту же инструкцию, но при этом обрабатываются разные блоки данных.

На сегодняшний день существует несколько программный архитектур, позволяющих использовать графический процессор для расчета задач, непосредственно не связанных с обработкой графики. Наиболее популярными являются CUDA и OpenCL. Основное отличие между ними заключается в том, что OpenCL предоставляет возможность использовать в вычислениях любые устройства, которые ее поддерживают, в то же время архитектуру CUDA поддерживает только компания NVIDIA. Однако полноценную поддержку со стороны сред разработки OpenCL получила лишь недавно, поэтому для реализации алгоритма была выбрана программная архитектура CUDA (Compute Unified Device Architecture)[4]. Данная архитектура появилась на рынке в 2005 году и на сегодняшний день является одной из самых популярных в данной области.

Найти разработки по реализации Хааровского классификатора с применением графического процессора не удалось.

# **1** ОБЗОР ЛИТЕРАТУРЫ

В процессе поиска информации по теме дипломного проекта было найдено относительно небольшое число подходящей литературы. В частности, по архитектуре CUDA были найдены только три книги, две из которых являются официальной документацией от компании-разработчика. Тем не менее, в Интернете доступно достаточно большое число материалов с различных конференций, а также спецкурсов, которые читают в университетах. Такая ситуация связана с тем, что данное направление является относительно молодым.

Основными материалами по Хааровскому классификатору являются [1, 5]. Обе статьи были представлены на конференциях IEEE. Статья [1] является первым доступным описанием алгоритма классификатора, который удалось найти. В ней доказывается практическая возможность нахождения лица на цифровом изображении и дается базовое представление о том, как это делается.

Достаточно подробное математическое описание алгоритма приводится в [5]. Реализация алгоритма в соответствии с этим описанием приводится в проекте [2]. Так как данный проект является открытой разработкой, он был выбран в качестве основы для разработки параллельного алгоритма Хааровского классификатора. В сочетании с презентацией [6], дающей наглядное представление того, как алгоритм классификатора работает, материалов по данной тематике достаточно, чтобы понять, каким образом работает модуль нахождения лиц в [2].

На сегодняшний день существует несколько программных архитектур, которые позволяют задействовать графические процессоры – OpenCL и CUDA. OpenCL (Open Computer Language) является открытой разработкой и поддерживается консорциумом Khronos Group, в которую входит достаточно большое число крупных компаний, – Apple, AMD, Intel, NVIDIA и другие. Обзор технологии можно найти на сайте [7]. Однако на практике поддержка данной технологии со стороны драйверов по-прежнему остается не самой эффективной, что делает ее не самым лучшим вариантом для применения. Такая ситуация видимо связана с относительной молодостью технологии.

В отличие от OpenCL архитектура CUDA обладает значительно более качественной поддержкой со стороны драйверов, быстротой исполнения программ. Поэтому, несмотря на то, что поддержка технологии осуществляется только компанией NVIDIA, данная программная архитектура была выбрана в качестве основы для разработки алгоритма классификатора. Также в дальнейшем можно осуществить достаточно быстрый переход на технологию OpenCL, так как данные архитектуры являются очень похожими с точки зрения программной реализации.

Видеокарты компании NVIDIA, которые поддерживают архитектуру CUDA, представлены на сайте [8]. На данном сайте также можно найти достаточно подробную спецификацию по каждой из представленных видеокарт. Также здесь собраны модели графических процессоров, которые поддерживают CUDA и устанавливаются в ноутбуки и системы на базе NVIDIA ION. Основным недостатком данного сайта является то, что нельзя выбрать несколько карт и сравнить их между собой. Исправить данный недостаток можно с помощью сайта [9]. На нем пользователю предлагается выбрать до 10 различных моделей графических архитектур и сравнить их в виде таблицы.

Сравнить эффективность использования архитектуры CUDA по сравнению с применением классического центрального процессора архитектуры x86 можно на сайте [10]. Авторы статьи специально выбрали не самые новые комплектующие, чтобы максимально приблизиться к конфигурации системы, которая на сегодняшний день является наиболее распространенной с их точки зрения. Результаты тестирования по материалам сайта однозначно сводятся к использованию CUDA для вычислений, так как выигрыш в производительности является достаточно существенным.

История развития применения графического процессора для вычислений, не связанных непосредственно с выводом графики, представлена на сайте [11]. Анализируются области, в которых применение такого подхода является наиболее оптимальным, рассматривается разница между CPU и GPU в параллельных расчетах. Также очень коротко рассматривается архитектура CUDA. Особенностью данной статьи является то, что авторы рассматривают процесс компиляции программы. В частности, рассматривается процесс преобразования из кода, написанного на одном из поддерживаемых языков программирования, в промежуточный ассемблерный код PTX (Parallel Thread Execution). Полученный код с помощью драйвера, окончательно обрабатывается, преобразуется в код для данного графического процессора и передается на исполнение графическому процессору. При этом промежуточный код PTX можно получить при компиляции программы. Это может пригодиться, если требуется провести низкоуровневую оптимизацию программы.

Сравнительный обзор архитектуры классических центральных процессоров с архитектурой видеокарты дается в [3]. Данная статья объясняет, почему графический процессор при решении определенных задач позволяет значительно ускорить процесс исполнения.

Архитектура Fermi является одной из последних разработок компании NVIDIA, поддерживающей CUDA. Статья [4] дает представление о пиковой производительности графического процессора, а также описывает возможности, которые появляются у разработчика, при использовании данной архитектуры. Но из-за того, что программа, разработанная для Fermi, не будет работать в системах на основе предыдущего поколения, разработка алгоритма будет осуществляться для графических процессоров предыдущего поколения. При этом разработанный алгоритм будет работать на видеокартах с архитектурой Fermi, так как данная преемственность сохраняется.

Достаточно интересным русскоязычным ресурсом по архитектуре CUDA является сайт [12]. На нем собраны материалы по всем современным разработкам компании NVIDIA, предлагается огромное количество видеоматериалов с презентацией различных технологий, которые были сделаны на конференциях и специализированных выставках. В частности, на сайте детально рассматриваются особенности архитектуры Fermi, которая является передовой технологией у компании NVIDIA, с точки зрения программиста, использующего архитектуру CUDA в своих разработках. Также автор рассказывает о среде разработки Microsoft Visual Studio и специально разработанной компанией NVIDIA надстройки для данной среды, которая значительно облегчает процесс разработки и отладки кода для архитектуры CUDA. Данное средство представляет собой достаточно мощный отладчик кода, который позволяет работать непосредственно с GPU, а не с центральным процессором, как это было раньше. Однако на текущий момент данное программное средство доступно только для очень узкого круга людей, так как оно находится на стадии Beta-тестирования. Тем не менее, последующее его использование может значительно облегчить процесс отладки программного кода.

Официальное руководство по разработке программ с использованием архитектуры CUDA является книга [13]. Это одна из лучших книг, в которой в максимально краткой и лаконичной форме изложен весь материал, который необходим, чтобы начать писать программы с использованием графического процессора.

Книга дает представление о программной модели CUDA, о таких базовых понятиях как ядро (функция, которая исполняется на видеокарте) и треды. Отдельная глава посвящена программному интерфейсу. Здесь описываются доступные типы памяти, рассказывается о функциях, с помощью которых можно обратиться к ним, о синхронизации тредов и других особенностях.

Отдельно описываются аппаратные особенности реализации архитектуры CUDA. Данные знания необходимы, чтобы программист мог представить то, каким образом код будет исполняться на видеокарте. Это позволяет писать значительно более качественный код.

Дополняет и значительно расширяет методы оптимизации использования GPU в программах книга [14]. Отдельная глава посвящена методике тестирования приложений, использующих для расчетов графические процессоры. В частности описываются особенности использования таймера, который предоставляется библиотекой CUDA. А также приводится методика расчета теоретической и эффективной пропускной способности памяти графического процессора. Указывается на различия эффективной пропускной способности и пропускной способности, полученной с помощью профилировщика CUDA Profiler. Таким образом, данная глава позволяет всесторонне оценить производительность, найти узкие места в программе.

Отдельный раздел книги посвящен оптимизации работы с памятью. Этот раздел является очень важным, так как неправильная работа с ней может значительно ухудшить время исполнения программы. В частности, приводится способ ускорения работы с глобальной памятью, которая чаще всего используется для передачи данных.

Даются рекомендации, как добиться максимальной производительности от среды исполнения. Приводится методика расчета числа регистров, которые может использовать один тред. Попытка превысить данное число приведет к тому, что будет использоваться локальная память. Это значительно ухудшит время исполнения программы.

Отдельный раздел посвящен оптимизации инструкций. Библиотека CUDA предоставляет достаточно большое количество арифметических функций, которыми можно заменить стандартные. Данное действие позволяет ускорить время исполнения, но это достигается в ущерб точности расчетов. Даются рекомендации по работе с циклами и условным ветвлениям.

Единственный учебник по CUDA, который удалось найти, является [15]. Авторы книги на примере перемножения двух матриц описывают методику разработки параллельного алгоритма, рассказывают об особенностях архитектуры. Основным достоинством данной книги является то, что на примере алгоритма перемножения матриц последовательно приводятся функции-ядра, проводится анализ эффективности такого подхода и даются рекомендации по устранению найденных недостатков. Такой последовательный подход значительно упрощает процесс разработки массово-параллельных программ.

Доступная в Интернете документация [16] предоставляет доступ к подробному описанию всех доступных функций и структур данных в библиотеке CUDA. Даются подробные описания входных переменных и возвращаемое функцией значение.

Достаточно большое число примеров программ, а также необходимые для разработки компоненты можно взять на сайте [17]. На этом же сайте собрана документация, которая позволяет лучше понять особенности работы видеокарт.

Одним из лучших русскоязычных ресурсов по программированию с использованием архитектуры CUDA является [18]. Данный курс разработан при поддержке компании NVIDIA. Также на данном сайте работает форум, на котором можно попробовать найти ответы на вопросы, которые чаще всего возникают у начинающих разработчиков, применяющих архитектуру CUDA.

Достаточно интересным ресурсом по разработке с использованием GPU является сайт [19]. На данном сайте собираются все новости, которые связаны с вычислениями на GPU, даются рекомендации по оптимизации программ, проводятся сравнительные обзоры архитектур видеокарт.

Компиляция программ, использующих графический процессор, осуществляется с помощью программы nvcc. Так как данная программа является консольной, компиляция исходного текста требует знаний параметров управления данной программой. Достаточно качественным руководством по поддерживаемым командам является справочник [20]. В ней можно найти описания всех директив и макросов, которые поддерживаются компилятором в тексте программы, описываются команды, с помощью которых можно получить как финальный исполняемый файл, так и промежуточные файлы на различных этапах компиляции (например, такие файлы, как PTX-код). Также описывается, как скомпилировать программы в режиме эмуляции. Такой подход преобразует код, предназначенный для исполнения на графическом процессоре, в код, который будет исполняться на центральном процессоре. На текущий момент это один из самых простых и эффективных способов отладки программы, так как для этого можно использовать стандартные средства. Отдельно описываются команды оптимизации с помощью компилятора.

# **2** ОПИСАНИЕ ХААРОВСКОГО КЛАССИФИКАТОРА

Впервые Харовский классификатор для распознавания образов был применен Виола и Джонсом и описан в статье [1], представленной на конференции IEEE в 2001 году. Для распознавания они применили специальное окно размером пикселей. В рамках окна выделяется дополнительная область меньшего размера, которую можно описать формулой:

|  |  |
| --- | --- |
| , | (2.1) |

где - выделенная область; - координаты области; - ширина и высота области соответственно; - угол, на который повернута область.

При этом на параметры накладываются следующие ограничения:

(2.2)

Примеры областей представлены на рисунке 2.1.



Рисунок 2.1 – Примеры вертикальной и повернутой на области

Полный набор значений признаков, которые можно описать с помощью окна и выделенной области можно описать следующей формулой:

|  |  |
| --- | --- |
| , | (2.3) |

где – вес; – область ; – сумма пикселей в области ; N – число рассматриваемых областей.

Но так как полный набор признаков является бесконечно большим, то, как правило, накладывают следующие ограничения:

* рассматриваются только взвешенные комбинации сумм пикселей двух прямоугольных областей (то есть N = 2);
* веса используются противоположных знаков и применяются для компенсации различий в размере областей между двумя прямоугольниками.

Если учесть данные ограничения, то остается 14 допустимых прототипов признаков, представленных на рисунке 2.2.

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Рисунок 2.2 – Прототипы признаков для хааровского классификатора

Для расчета суммы пикселей в вертикальной области используется метод расчета суммарной площади таблицы (Summed Area Table, SAT). Причем значения SAT рассчитываются для каждой точки с координатами () по формуле:

|  |  |
| --- | --- |
| . | (2.4) |

Либо, если выразить через значения SAT на предыдущих этапах:

|  |  |
| --- | --- |
| . | (2.5) |

Причем следующие значения заранее известны и является постоянной:

|  |  |
| --- | --- |
| . | (2.6) |

На основе полученных значений значение , необходимое для расчета значения признака , для произвольной области можно найти по формуле:

|  |  |
| --- | --- |
| . | (2.7) |

Правильность данного метода для расчета суммы пикселей в области опишем с помощью рисунка 2.3.



Рисунок 2.3 – Пример расчета суммы пикселей в области D

Пусть требуется найти сумму пикселей в области D. При этом сумма пикселей в точке 1 соответствует области A, в точке 2 – сумме пикселей областей A+B, в точке 3 – области A+C, в точке 4 – области A+B+C+D. Также точка 1 соответствует , точка 2 – , точка 3 – , точка 4 – в формуле 2.7. Отсюда легко получить, что сумма пикселей области D действительно находится по формуле 2.8:

|  |  |
| --- | --- |
| . | (2.8) |

Для области, повернутой на угол , по аналогии применяется метод расчета суммарной площади повернутой таблицы (Rotated Summed Area Table, RSAT):

|  |  |
| --- | --- |
| . | (2.9) |

Либо, если выразить через значения RSAT на предыдущих этапах:

|  |  |
| --- | --- |
| . | (2.10) |

Причем

|  |  |
| --- | --- |
| . | (2.11) |
| . | (2.12) |

На основе полученных значений значение для произвольной области можно найти по формуле:

|  |  |
| --- | --- |
| . | (2.13) |

В рамках дипломной работы использовалась вертикальная область, то есть угол .

Прототип признака последовательно накладывается на каждый пиксель изображения и для выделенной области вычисляется значение суммы пикселей. Вычисленная сумма используется для расчета значение признака по формуле 2.3, которое сравнивается с эталонным значением. На основе результата сравнения классификатор выдает результат +1, если выделенная область подходит, и -1, в противном случае.

Каскады объединяются в каскады классификаторов (cascade of classifiers). Процесс анализа изображения сводится к тому, что цифровое изображение последовательно анализируется на каждом каскаде классификатора. При этом на каждом этапе происходит отсеивание некоторого количества пикселей, которые на последующих стадиях обрабатываться уже не будут. После того, как изображение пройдет через все каскады классификатора, выдается результат.

Однако из-за того, что прототип признака имеет фиксированный размер, можно найти только лица, которые по размеру соответствуют анализируемой области. Для поиска лица произвольного размера применяется два подхода: либо масштабирование классификатора, либо масштабирование самого изображения.

# **3** ОБЗОР АРХИТЕКТУРЫ CUDA

Архитектура CUDA относится к типу архитектур с массовым параллелизмом. Обработка данных выполняется множеством тредов. При этом все треды одновременно исполняют одну и туже функцию, которая называется ядром. Множество тредов группируется в блоки, которые формируют решетку (grid) блоков. Перед тем, как передать управлению ядру, разработчик должен произвести настройку конфигурации запуска, в которой указывается число блоков и тредов. Также разработчик должен обеспечивать, чтобы каждый блок тредов, входящий в решетку тредов, обрабатывал свой локальный участок данных.

Архитектура CUDA накладывает определенные ограничения на конфигурацию запуска. Так, например, число тредов в одном блоке не может превышать 512, количество блоков в решетке тредов – 65535 по каждой из размерностей блока (размерность блока может быть 1-, 2-, 3-мерной). Динамическое изменение конфигурации не допускается.



Рисунок 3.1 – Схема графического процессора

Вычисления в графическом процессоре осуществляются массивом мультипроцессоров. Их число может достигать 30. Каждый из мультипроцессоров подключен по общей шине к контроллеру ОЗУ, обеспечивающего связь с глобальной памятью. Общая схема графического процессора представлена на рисунке 3.1.



Рисунок 3.2 – Устройство мультипроцессора

Каждый из мультипроцессоров состоит из 8 вычислительных блоков (ВБ), блока регистров, блока разделяемой памяти и устройства управления (УУ). Данная схема приведена на рисунке 3.2.

Мультипроцессор использует парадигму SIMT (Single Instruction Multiple Thread). Особенностью данной парадигмы является то, что процессор одновременно выполняет большое количество тредов, которые синхронно исполняют одну и туже инструкцию. При этом каждый тред обрабатывает один элемент данных.

Варп (warp) – это группа тредов, для которых поддерживается синхронное исполнение. Для современных устройств варп содержит 32 треда. Блок тредов обрабатываются одним мультипроцессором, для исполнения он разбивается на варпы. Между тредами, которые принадлежат различным варпам, автоматическая синхронизация не поддерживается. Но мультипроцессор содержит специальную инструкцию для выполнения барьерной синхронизации всех тредов в блоке.

Графический процессор предоставляет доступ к различным типам памяти. Наиболее важными являются следующие типы – глобальная, разделяемая, регистровая и константная.

Глобальный тип памяти является самым большим по объему, но обладает высокой латентностью. Физически чип памяти расположен вне кристалла графического процессора и не поддерживает кэширование данных. Архитектура CUDA предоставляет способ, при котором процесс обмена данными будет осуществляться значительно быстрее, – механизм транзакций.

Транзакция позволяет объединить 16 обращений от соседних тредов в одно обращение, которое обрабатывается эффективнее. Однако для этого требуется выполнение ряда условий:

* обращения всех 16 тредов должны попасть в один 64- или 128-байтный сегмент, который выровнен в памяти;
* тип, используемый при обращении, должен занимать 32, 64 или 128 бит;
* k-ый поток должен читать соответствующее ему k-ое слово из сегмента данных в памяти.

Правильное обращение к памяти с формированием транзакций проиллюстрировано на рисунке 3.3. В качестве примера используется тип данных float.

Как видно из рисунка 3.3, допускается пропуск тредом операции чтения соответствующего адреса. Любые другие варианты не приведут к формированию транзакций.

Разделяемая память физически расположена в мультипроцессоре. Соответственно, ее можно использовать для обмена данными между тредами одного блока, так как они исполняются одним и тем же мультипроцессором. Это один из самых быстрых типов памяти, которые доступны разработчику, но он обладает незначительным объемом. На массово распространенных видеокартах объем составляет 16 Кб (для видеокарт на архитектуре Fermi это значение может быть увеличено за счет константной памяти). Для ускорения работы память разбивается на 16 блоков (ячеек одинакового размера), обращение к которым может осуществляться независимо. Для достижения пиковой производительности каждый тред в пределах варпа должен обращаться к блокам так, чтобы 2 и более треда не читали один и тот же блок. Единственное исключение – все треды читают один и тот же блок.



Рисунок 3.3 – Примеры формирования транзакций

Регистровая память обладает самой низкой латентностью. На ее основе формируется регистровый файл для каждого исполняемого треда. Разработчику этот тип памяти напрямую не доступен.

Константная память доступна только для чтения, поэтому эффективна только при размещении констант. Физически она располагается в глобальной памяти, но обладает собственным кэшем. За счет этого, в случае, если данные ранее были прочитаны в кэш и они там сохранились, процесс чтения осуществляется значительно быстрее.

Для достижения пиковой производительности должны соблюдаться правила работы с памятью, то есть выполняться механизм транзакций при чтении или записи в глобальную память, отсутствовать пересечения тредов при обращении к блокам разделяемой памяти. Также требуется, чтобы конфигурация запуска ядра максимально задействовала имеющиеся вычислительные ресурсы.

# **4** СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Весь разрабатываемый проект разбивается на отдельные модули. Это необходимо, чтобы обеспечить гибкую структуру программы. В частности, при таком подходе допускается модернизация любого из выбранных модулей без изменения остальных. Для разрабатываемой программы были выделены модули, которые представлены в структурной схеме (см. чертеж ГУИР.400201.062 С1).

Работа программы начинается с инициализации всех входных параметров. На данном этапе выбирается эталонный классификатор, с которым будет осуществляться дальнейшая работа, задается входное изображение, которое будет анализироваться.

После этого управление передается модулю подготовки изображения. Его основная задача заключается в том, чтобы осуществить масштабирование изображения. Как отмечалось в разделе 2, процесс масштабирования необходим, чтобы была возможность найти лица на картинке, которые по размерам больше, чем выбранный размер окна классификатора. Из двух возможных вариантов (масштабирование классификатора или самого изображения) был выбран вариант с масштабированием изображения. Это связано с тем, что такой подход позволяет значительно эффективнее распределить нагрузку на графическом процессоре. Также на этом этапе рассчитывается интегральная сумма изображения и инициализируется структура для каскадов классификатора, которые будут использоваться при обработке изображения на центральном процессоре. Интегральная сумма необходима, чтобы обеспечить оптимизированный вариант расчета значения признака по формуле 2.3 для изображения и, в частности, значения переменной . В конце работы модуля создается маска изображения, элементами которой являются пиксели исходного изображения. В дальнейшем она будет использоваться для обработки на каскадах классификатора. После обработки всего классификатора по маске будут определяться области, на которых предположительно найдены лица.

Последующая работа программы допускается в одном из двух режимов работы:

* расчет каскадов классификатора полностью осуществляется на центральном процессоре;
* расчет каскадов классификатора осуществляется с применением мощности графического процессора.

Такое разделение необходимо, чтобы обеспечить максимальную производительность. Применение графического процессора становится эффективным, если есть возможность задействовать все мультипроцессоры графического процессора. Однако применение масштабирования приводит к тому, что, начиная с определенного этапа, полностью задействовать ресурсы GPU становится невозможным. Поэтому начиная с этого момента, становится эффективным расчет с применением классического центрального процессора.

Модуль обработки изображения на центральном процессоре последовательно обрабатывает каждый пиксель изображения на каждом каскаде классификатора. Если пиксель «прошел» все каскады, то в маске изображения он отмечается единицей, в противном случае – нулем. После того, как будут обработаны все пиксели картинки, получаем маску изображения, которая будет использоваться для анализа областей, на которых нашли лицо. Так как результаты обработки пикселей не зависят друг от друга, их обработку можно осуществлять независимо друг от друга с использованием многоядерной архитектуры современного микропроцессора. Это позволит дополнительно ускорить вычисления.

Однако такой подход не приемлем при расчетах с использованием графического процессора. Это связано в первую очередь с ограниченностью ресурсов самого GPU, а также с особенностями его работы. Среди основных проблем можно выделить следующие:

* кроме интегральной суммы изображения требуется хранить в памяти все необходимые данные для каскадов классификатора, а разместить все наиболее оптимальным образом является достаточно сложной задачей;
* каждый пиксель изображения пройдет различное число каскадов классификатора, что приведет к неравномерной нагрузке вычислительных блоков мультипроцессора.

Указанные проблемы могут привести к тому, что эффективность от использования графического процессора будет отсутствовать. В связи с этим был придуман иной способ обработки при использовании GPU.

Чтобы была возможность использовать графический процессор для расчета, требуется осуществить инициализацию CUDA, а также выделить память для данных, которые будут использоваться на GPU. В связи с тем, что программа не допускает динамическое изменение каскадов классификатора, также на этом этапе осуществляется инициализация структуры классификатора для GPU. При этом данные действия могут осуществляться только один раз перед первым применением функций-ядер.

Основная идея разработанного алгоритма заключается в том, что ядро обрабатывает не все каскады сразу, а по одному для каждого масштаба изображения. Единственное исключение – обработка первых двух каскадов классификатора осуществляется одним ядром. Такой подход требует значительно меньшего объема памяти для хранения необходимых данных.

Дополнительно уменьшить число элементов маски изображения, которые требуется обрабатывать в дальнейшем, можно за счет использования фильтрации. Основная задача данного модуля заключается в том, чтобы найти по определенному правилу точки, обработка которых на дальнейших каскадах классификатора не принесет значительной погрешности в результатах работы программы (как правило, погрешность, составляющая несколько пикселей).

Однако остается еще одна проблема. После обработки изображения на первых двух каскадах классификатора отсеивается около 75% пикселей. Каждый последующий каскад также отсеивает значительное число пикселей. В результате этого элементы маски изображения, которые требуется обрабатывать, становятся разреженными. Это приводит к тому, что последовательное чтение данных из глобальной памяти является малоэффективным, так как значительная часть тредов в этом случае будет простаивать. Выборочное чтение данных из памяти также неэффективно, так как это не приводит к формированию транзакций. Для того чтобы оптимизировать нагрузку на графическом процессоре вводится модуль формирования плана. Его основная задача заключается в том, чтобы создать массив, элементами которого являются координаты пикселей, которые требуется обрабатывать на последующих каскадах.

После того, как будет сформирован очередной план, система возвращается к вызову функции-ядра исполнения каскада. При этом данные функции должны поддерживать работу с планом, а не только с исходным интегральным изображением.

Предложенный алгоритм является универсальным. Это значит, что можно взять абсолютно любой классификатор и использовать его в качестве эталона при распознавании. Однако такой подход требует значительного числа лишних операций (например, чтение из памяти параметров классификатора и другие). Существует альтернативный подход – использование генератора кода и функции-ядра, работающие на его основе.

Основная задача генератора кода заключается в том, чтобы по шаблону составить код функции для заданного каскада, который в точности повторит действия универсального ядра для этого же каскада. Однако при этом подставит в исходный текст программы все переменные выбранного каскада. В результате этого полученный код теряет универсальность. Чтобы была возможность использовать этот классификатор, требуется предварительно использовать генератор кода, который создаст требуемый набор функций-ядер, подключить сгенерированные ядра к проекту, скомпилировать проект и только после этого их можно использовать. Однако, несмотря на сложность использования такого подхода, это позволяет значительно ускорить процесс вычисления на графическом процессоре. Это связано с уменьшением числа инструкций в коде (например, операций чтения параметров классификатора).

Тем не менее, обработка всех каскадов классификатора на GPU также является не эффективной. Это связано с тем, что данных для обработки становится слишком мало приблизительно на 10-15 каскаде, чтобы обеспечить равномерную нагрузку на графический процессор. Поэтому начиная с этого момента, последующая обработка каскадов классификатора осуществляется с применением центрального процессора. Данный процесс аналогичен тому, как осуществляется обработка всего изображения на CPU, только обработка осуществляется не для всех каскадов, а только тех, которые еще не были обработаны. Момент перевода с GPU на CPU определяется отношением их вычислительных мощностей и для разных систем различен.

После того как пройдены все каскады классификатора, анализируется результирующая маска изображения. Найденные координаты предполагаемых лиц сохраняются для последующей финальной обработки. После этого рассчитывается следующий коэффициент масштабирования и, если предел масштаба изображения не достигнут, процесс поиска лиц повторяется, начиная с этапа масштабирования изображения.

После того, как будут проанализированы все возможные масштабы изображения, осуществляется финальная обработка найденных областей. Основной задачей этого этапа является поиск областей, которые входят в более крупные области, и удаление их из списка рассматриваемых. После этого результаты накладываются на анализируемое изображение и полученная картинка сохраняется.

# **5** ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

## **5.1** Описание конфигурации запуска ядер для GPU

Для разрабатываемой системы можно выделить две основные конфигурации запуска. Такое разбиение связано с тем, что первых два каскада классификатора обрабатываются одним и тем же ядром, а это, в свою очередь, для корректной работы требует значительно больший объем разделяемой памяти.

Для равномерной нагрузки графического процессора и экономного использования имеющейся памяти изображение разбивается на блоки по 16х16 пикселей, если обрабатываются первых два каскада, либо по 32х32 пикселя для всех остальных. Распределение пикселей изображения между тредами задается статически и изменению не подлежит. При этом количество пикселей, приходящихся на 1 тред, одинаково. Размер накладываемого признака – 20х20 пикселей. Следовательно, чтобы была возможность обрабатывать весь блок шириной 16х16 либо 32х32 пикселя, требуется хранить в разделяемой памяти фрагмент изображения общим размером 48х48 либо 52х52 пикселя соответственно.

Также при запуске ядра требуется указать число тредов и блоков, которые должны отработать. Для обработчиков каскадов используется следующая конфигурация запуска: число тредов в одном блоке – 16х16 элементов, число блоков в решетке блоков рассчитывается исходя из размеров самого изображения. Так как обрабатываются участки размером 32х32, изображение предварительно выравнивается с учетом данного размера. После этого определяется, какое число блоков должно участвовать в обработке, чтобы обработать все изображение. Для первых двух каскадов размерность определяется путем деления размера изображения по обеим координатам на 16, для остальных ядер – делением на 32. Размерность решетки блоков сохраняется для всех ядер.

## **5.2** Описание структуры и локальных переменных классификатора

Структуру классификатора, который используется в программе для поиска лица, графически можно изобразить как на рисунке 5.1.

Как видно из данного рисунка, классификатор состоит из множества каскадов, в каждый из которых входит набор признаков. Каждый из признаков определяет 2 или 3 окна (так как третье окно может отсутствовать, оно отмечено штрихом). Однако такая структура не является оптимальной, потому что она требует лишние операции чтения при работе. Поэтому была разработана структура, которая минимизирует число операций чтений памяти. Она изображена на рисунке 5.2. Более подробно алгоритм работы классификатора представлен в подразделе 6.1.



Рисунок 5.1 – Структура классификатора после чтения его из файла



Рисунок 5.2 – Структура классификатора для GPU

Для описания классификатора на рисунке 5.2 были разработаны следующие структуры:

struct gpu\_cascade

{

unsigned int node2\_count;

unsigned int node2\_first;

unsigned int node3\_count;

unsigned int node3\_first;

unsigned int node\_position;

float threashold;

};

struct gpu\_rect

{

unsigned short p1;

unsigned short p2;

unsigned short p3;

unsigned short p4;

};

struct gpu\_node

{

float a;

float b;

float threashold;

};

Структура gpu\_cascade соответствует описанию каскадов. Так как все признаки и окна располагаются линейно без разделения на каскады, структура должна содержать индексы начала блоков по 2 окна (node2\_first), по 3 окна (node3\_first) и начало блока признаков (node\_position). Также, чтобы упростить вычисления, введено число окон по 2 (node2\_count) и по 3 (node3\_count), число признаков для данного каскада определяется как сумма блоков по 2 и 3 окна (то есть node2\_count + node3\_count). Переменная threashold используется для принятия окончательного решения – подходит или нет данная область.

Все окна описывает структура gpu\_rect. Ее элементы описывают смещение относительно пикселя, который рассматривается в текущий момент данным вычислительным блоком.

Множеству признаков соответствует структура gpu\_node. Данная структура используется для наращивания порогового значения, на основе которого будет приниматься окончательное решение, на фиксированную величину (a или b) в зависимости от результата сравнения признака, полученного после обработки 2 (3) окон, с эталонным значением данного признака threashold.

С учетом данных структур введены следующие переменные:

\_\_device\_\_ \_\_constant\_\_ gpu\_rect haar\_rects2[RECTS2];

\_\_device\_\_ \_\_constant\_\_ float haar\_rect\_weights2[RECTS2];

\_\_device\_\_ \_\_constant\_\_ gpu\_rect haar\_rects3[RECTS3];

\_\_device\_\_ \_\_constant\_\_ float haar\_rect\_weights3[RECTS3];

\_\_device\_\_ \_\_constant\_\_ gpu\_node haar\_nodes[NODES];

\_\_device\_\_ \_\_constant\_\_ gpu\_cascade haar\_cascade[CASCADES];

В них переменные RECTS2 и RECTS3 описывают максимальное число по 2 и 3 окна соответственно, NODES – число признаков на все каскады, CASCADES – число каскадов, которые можно обработать с применением графического процессора. Переменные haar\_rect\_weights2 и haar\_rect\_weights3 используются для хранения значения весов, которые используются для расчета значения признака (см. формулу 2.3).

## **5.3** Модуль инициализации данных и CUDA

Перед тем, как использовать ядра, требуется произвести инициализацию среды CUDA. За это отвечает функция InitCUDA.

На данном этапе рассчитываются размеры массивов, которые необходимы для работы программы, осуществляется выделение соответствующей памяти. Выделение памяти может осуществляться только один раз, так как применение масштабирования изображения приводит к уменьшению размеров массивов, а не к их увеличению. Соответственно, выделять память для нового размера не требуется, достаточно только рассчитать новые размеры массива.

Основные массивы, которые при этом используются, делятся на два типа – выделяемые в классическом ОЗУ и в памяти видеокарты. Для видеокарты используются следующие массивы – маска изображения (unsigned char \*res\_gpu), два массива, содержащих старое и новое значение плана (unsigned short \*plans\_gpu, \*\_plans\_gpu), интегральное изображение, используемое для обработки, (unsigned int \*integral\_img\_gpu), ряд других массивов, используемых для временного хранения данных.

Из массивов, хранящих данные в классическом ОЗУ, можно выделить следующие – массив, использующийся для восстановления маски изображения после обработки на графическом процессоре (unsigned char\* res\_cpu), а также план, который используется для тех же целей, (unsigned short\* plans\_cpu).

Также в задачу данного модуля входит инициализация константной памяти для хранения значений всех каскадов классификатора, которые будут исполняться на графическом процессоре. При этом осуществляется преобразование структур, используемых в классической реализации, в структуры, которые эффективно использовать на GPU (см. описание в подразделе 5.2). За преобразование структур отвечает функция SetCascadeToGPU.

Единственная функция, которая должна исполняться каждый раз перед первым применением функций-ядер для текущего масштаба изображения, связана с расчетом коэффициентов нормализации \_calcVarainceNormfactor. В связи с тем, что значения пикселей изменяются при использовании очередного коэффициента масштабирования, пороговые значения для 2 либо 3 признаков должны изменяться с учетом изменения масштаба. Это обеспечивает данная функция.

## **5.4** Модуль обработки изображения на GPU

В программе реализовано 2 подхода к обработке очередного каскада классификатора – на основе универсального исполнения и на основе генератора кода. Такие варианты сделаны для того, чтобы у пользователя был выбор – использовать универсальный вариант, но тратить больше времени при работе программы, или использовать модули, полученные на основе генератора кода, получив при этом максимальную производительность, но заплатив при этом сложностью использования такого подхода и его неуниверсальностью. Режим работы задается с помощью специального флага mode.

Универсальный режим работы.

Для обработки первых двух каскадов используется функция haar\_first\_stage. Эта функция обрабатывает изображение блоками 16х16, что связано с нехваткой быстрой памяти для хранения всех требуемых данных. Выходным значением данной функции является массив res\_gpu, который представляет собой сформированную маску изображения (см. рисунок 5.3.а). Дополнительно для данной функции может применяться ядро filter. В его задачу входит модифицирование маски таким образом, чтобы из нее был удален каждый второй подряд идущий элемент, а также преобразование блока обработки из 16х16 пикселей в блок 32х32 пикселей, так как последующие ядра работают только с блоком размером 32х32. Пример работы фильтра представлен на рисунке 5.3. Результат обработки маски представлен на рисунке 5.3.б (измененные значения выделены жирным).

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *а)* | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 |  | *б)* | 1 | **0** | 1 | **0** | 1 | 0 | 1 | 0 |
|  | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |  |  | 0 | 1 | 0 | 1 | 0 | 1 | **0** | 0 |
|  | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |  |  | 1 | **0** | 0 | 1 | **0** | 1 | **0** | 0 |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |  |  | 0 | 0 | 0 | 1 | **0** | 1 | 0 | 0 |
|  | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |  |  | 1 | **0** | 1 | 0 | 0 | 0 | 1 | **0** |
|  | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |  |  | 0 | 0 | 0 | 1 | **0** | 1 | **0** | 0 |
|  | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |  |  | 1 | 0 | 0 | 0 | 1 | 0 | 1 | **0** |
|  | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |  |  | 1 | **0** | 0 | 0 | 0 | 1 | **0** | 1 |

Рисунок 5.3 – Фрагмент маски изображения: *а* – до применения фильтра; *б* –после этого

Применение такой фильтрации оправдано только при применении после haar\_first\_stage. Это связано с тем, что суммарное время работы фильтра и исполнения очередного каскада превышает время обработки каскада без предварительной фильтрации.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *а)* | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |  | *б)* | 1 | 3 | 5 | 7 | 10 | 12 | 14 | 17 |
|  | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |  |  | 20 | 22 | 28 | 30 | 33 | 35 | 39 | 44 |
|  | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |  |  | 46 | 49 | 53 | 55 | 57 | 61 | 63 | 2048 |
|  | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |  |  | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |
|  | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |  |  | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |
|  | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |  |  | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |
|  | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |  |  | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |
|  | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |  |  | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |

Рисунок 5.4 – Пример: *а* – фрагмент маски изображения; *б* – план для данной маски

После того как маска сформирована, управление передается модулю формирования плана. В случае если план готовится впервые (осуществляется подготовка к обработке третьего каскада классификатора), вызывается функция DataToQueueB1\_32. Данное ядро формирует план. Элемент плана представляет собой координаты единичных элементов в маске в рамках блока в 32x32 пикселя. Элементы плана, которые ничего не обрабатывают, отмечены числом 2048. Пример маски изображения и плана для данной маски представлен на рисунке 5.4 (для примера блок равен 8х8 пикселей).

Как видно из рисунка 5.4.б, элементы плана расположены в массиве компактно. Последовательное чтение данного массива приведет к формированию транзакций и позволит всем тредам работать с максимальной эффективностью.

После этого осуществляется вызов функции-ядра для обработки очередного каскада классификатора. При этом данные ядра уже должны поддерживать работу с планом, а не с исходной маской изображения. При этом реализуется три варианта ядер – haar\_next\_stage, haar\_next\_stage\_4threads и haar\_next\_stage\_8threads. Основное отличие между ними заключается в том, какое число тредов будут обрабатывать один и тот же элемент маски изображения. Для функции haar\_next\_stage – 2 треда, для функции haar\_next\_stage\_4threads – 4 треда, для функции haar\_next\_stage\_8threads – 8 тредов на 1 элемент маски.

Такое разделение на 3 типа ядер связано с тем, что маска изображения становится слишком разреженной и, несмотря на использование плана, максимальной производительности добиться не удается, так как для значительного числа тредов данных для обработки нет. В случае, когда один элемент плана обрабатывается двумя, четырьмя либо восемью тредами, все множество признаков, которые требуется обработать для данного каскада, разделяются между данным числом тредов равномерно, что позволяет оптимизировать нагрузку между вычислительными блоками мультипроцессора и добиться значительного ускорения.

После того, как будет обработан очередной каскад классификатора, управление передается функции DataToQueueB. Она является аналогом функции DataToQueueB1\_32, но учитывает уже имеющийся план. На основе уже имеющегося плана и модифицированной маски изображения она составляет новый план. После этого снова вызывается обработчик ядер и цикл повторяется. Число каскадов, которые будут обрабатываться с применением графического процессора, задаются во время компиляции программы и должны зависеть от конфигурации системы, на которой планируется ее использование.

Этапы перехода между функциями-обработчиками каскада haar\_next\_stage, haar\_next\_stage\_4threads и haar\_next\_stage\_8threads определяются экспериментальным путем и, в целом, являются одинаково оптимальными для любого типа графического процессора.

Режим обработки каскадов классификатора на основе генератора кода представляет собой аналогичный процесс, только вместо универсальных функций подставляются специализированные, созданные с помощью генератора кода.

В процессе анализа производительности было обнаружено, что простое применение плана, которое описано выше, является не эффективным. Это связано с тем, что в процессе чтения плана ядром возникают постоянно конфликты по доступу к разделяемой памяти (обращение к одним и тем же банкам памяти). Чтобы минимизировать число конфликтов, был найден дополнительный способ оптимизации ­– элементы внутри плана сортируются по остатку от деления на 16 по формуле 5.1:

, (5.1)

где , – координаты пикселя внутри блока изображения; число 52 связано с размером блока изображения, которое хранится в разделяемой памяти; число 16 связано с числом тредов, которые исполняются на мультипроцессоре одновременно.

В результате сортировки 16 тредов обращаются к разным банкам разделяемой памяти, что значительно уменьшает число конфликтов. Для оставшихся элементов плана, которые указанным способом разместить нельзя, рассчитывается число варпов, в которые укладывается обработка оставшихся элементов. Формируется массив, показывающий, какое число элементов осталось с определенным остатком от деления, находится максимальное значение среди них. На основе полученного максимума определяется число элементов плана, значения которых будут последовательно размещены, начиная с текущей позиции, в новом плане. Находятся элементы плана, которые дают выбранный остаток от деления, и копируются в результирующий план. Операция поиска максимального значения и размещения выполняется для всех элементов варпа, а также для каждого из оставшихся варпов. Такая перестановка элементов плана обеспечивает минимальное число конфликтов по доступу к разделяемой памяти.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 2 | 4 | 6 | 8 | 11 | 23 | 32 | 34 | 36 | 38 | 40 | 42 | 44 | 55 | 66 |
| 70 | 74 | 96 | 129 | 131 | 133 | 135 | 137 | 141 | 197 | 199 | 201 | 203 | 205 | 209 | 215 |
| 704 | 763 | 767 | 775 | 788 | 792 | 803 | 823 | 835 | 844 | 848 | 860 | 868 | 876 | 880 | 882 |
| 899 | 903 | 907 | 909 | 911 | 913 | 915 | 929 | 931 | 935 | 937 | 939 | 941 | 945 | 947 | 960 |
| 964 | 966 | 968 | 972 | 976 | 978 | 980 | 982 | 984 | 990 | 994 | 1006 | 1008 | 1012 | 1018 | 1023 |

Рисунок 5.5 – План до сортировки

Пусть, например, функция возвратила план, показанный на рисунке 5.5. Первый этап сортировки переставляет элементы плана по возрастанию по формуле 5.1. В результирующем плане данный этап соответствует первым 2 строчкам на рисунке 5.6. Остатки от деления на 16 для плана на рисунке 5.5 представлены на рисунке 5.7.

Остальные элементы разместить указанным способом нельзя. Поэтому работает второй этап обработки. Число элементов, которые требуется разместить, – 48. Размер варпа – 16 тредов. Отсюда получается, что требуется обработать 3 блока варпов. В таблице 5.1 указаны остатки от деления и число элементов, которые соответствуют данным остаткам.

Таблица 5.1 – Число элементов в соответствии с остатком от деления

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Остаток | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| Количество | 4 | 2 | 0 | 2 | 4 | 2 | 2 | 6 | 6 | 0 | 2 | 6 | 4 | 2 | 4 | 2 |

Максимально количество элементов с остатками 7, 8, 11 (их по 6 штук). Размещаем по 2 элемента с данными остатками (первые 6 элементов третьей строки на рисунке 5.6). Соответственно элементов с данным остатком остается по 4 штуки. Теперь максимальное количество элементов с остатками 0, 4, 12 и 14 (их по 4 штуки), при этом рассматриваемые на предыдущем шаге элементы уже не учитываются. Размещаем элементы с данными остатками. Аналогично размещаем 2 элемента с остатком 3. Третья строка плана сформирована.

Аналогично осуществляется перестановки для оставшихся 32 элементов.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 129 | 2 | 131 | 4 | 133 | 6 | 23 | 8 | 137 | 38 | 11 | 40 | 141 | 42 | 199 |
| 44 | 201 | 74 | 203 | 32 | 205 | 34 | 135 | 36 | 209 | 66 | 55 | 96 | 197 | 70 | 215 |
| 704 | 792 | 763 | 775 | 767 | 823 | 788 | 844 | 882 | 966 | 868 | 968 | 880 | 964 | 899 | 915 |
| 848 | 876 | 803 | 903 | 835 | 907 | 860 | 972 | 982 | 994 | 984 | 1012 | 913 | 941 | 929 | 945 |
| 960 | 976 | 990 | 1018 | 931 | 947 | 978 | 1006 | 935 | 1023 | 980 | 1008 | 909 | 937 | 911 | 939 |

Рисунок 5.6 – План после сортировки

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 8 | 8 | 7 | 7 | 11 | 11 | 4 | 4 | 14 | 14 | 0 | 0 | 12 | 12 | 3 | 3 |
| 8 | 8 | 7 | 7 | 11 | 11 | 4 | 4 | 14 | 14 | 0 | 0 | 1 | 1 | 5 | 5 |
| 8 | 8 | 6 | 6 | 7 | 7 | 10 | 10 | 11 | 11 | 12 | 12 | 13 | 13 | 15 | 15 |

Рисунок 5.7 – Остаток от деления на 16 для плана после сортировки

После того, как будут обработаны все каскады классификатора, которые обрабатываются с применением графического процессора, вызывается функция RestoreMask. Ее основная задача сводится к тому, чтобы преобразовать маску изображения, полученную после обработки на GPU, к виду, пригодному к обработке на центральном процессоре. В частности, графический процессор обрабатывает маску блоками 32х32 пикселя, а центральный процессор обрабатывает последовательно весь массив, не разбивая его на блоки. Данным преобразованием и занимается эта функция.

## **5.5** Модуль генератора кода

Данный модуль осуществляет генерацию функций-ядер для обработки каскадов классификатора по заданному шаблону, который представляет собой описание на основе универсальный функций. Основная управляющая функция – GenerateCascade. Ее алгоритм построен таким образом, что пользователю требуется только указать, какой набор ядер (на 2, 4, 8 потоков) требуется сгенерировать и для какого числа каскадов. После этого осуществляется формирование структуры классификатора, аналогичные тем, что описаны в подразделе 5.2. Далее поэтапно формируются выходные заголовочные файлы, содержащие код функции для очередного каскада. Разработчику надо лишь прописать данные заголовочные файлы и перекомпилировать проект.

## **5.6** Модуль расчета оставшихся каскадов на CPU

Применение всех указанных методов оптимизации все равно приводит к тому, что, начиная с определенного каскада, применение графического процессора становится не оправданным. Это связано с тем, что данных для обработки слишком небольшое число, поэтому наиболее эффективным становится применение классического центрального процессора. Уменьшение данных для обработки связано с коэффициентом масштабирования, так как чем он больше, тем меньше данных для обработки. График изменения множества данных (ось Y), которые требуется обработать, в зависимости от масштаба (ось X) изображен на рисунке 5.8.

При реализации алгоритма на центральном процессоре организуются циклы по осям x и y, проверяется элемент маски изображения, находящийся по заданной координате и, если этот элемент надо обрабатывать, осуществляется проверка уже всех оставшихся каскадов классификатора. Если данная координата прошла все каскады, то она отмечается для последующего анализа. Циклы по осям легко распараллелить, поэтому с этой целью применяется технология OpenMP. Данная технология позволяет автоматически распараллеливать указанный цикл на число потоков, который будет указан. Для реализации расчета оставшихся каскадов по данному алгоритму используется функция cvRunHaarClassifierCascade.

Основная сложность данного модуля заключается в том, что число каскадов, которые эффективно рассчитывать с применением графического процессора, не является постоянной величиной. Она очень сильно зависит от конфигурации системы и, в частности, от используемого типа центрального процессора и типа видеокарты.

Рисунок 5.8 – Зависимость данных для обработки от коэффициента масштабирования

В рамках дипломного проекта число каскадов, которые будут рассчитываться с применением графического процессора, задаются во время компиляции программы.

Диаграмма последовательности, описывающая последовательность вызовов функций, представлена на чертеже ГУИР.400201.062 PP. Диаграмма соответствует обработке каскадов классификатора для начального масштаба.

Объект CPU содержит функции, которые будут исполняться с применением центрального процессора, кроме 2 функций, связанных с инициализацией архитектуры CUDA и относящиеся к объекту cudaInitialize. Объект gpuKernels содержит ядра, исполняемые на GPU.

# **6** РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ

## **6.1** Алгоритм работы классификатора

В начале программы организуется цикл перебора пикселей изображения. Так как размеры обрабатываемой картинки изменяются, после организации цикла перебора всех пикселей изображения вычисляется коэффициент масштаба для данного изображения. После этого организуется перебор каскадов классификатора с обнуление порога. Порог используется для принятия решения, прошел пиксель данный каскад или нет. Так как каждый каскад содержит некоторое число признаков, осуществляется цикл перебора по данным признакам.

Из-за применения масштабирования вычисляется эталонный порог для классификатора. Для этого применяется коэффициент масштабирования, вычисленный ранее.

Каждый признак содержит характеристики для 2 или 3 окон. Для данного числа окон вычисляются признаки, после чего производится их суммирование. Если полученная в результате сумма меньше эталонного значения для каскада, то к порогу добавляется значение a, хранящееся в описании классификатора, иначе добавляется значение b.

После того, как будет посчитано пороговое значения для всех признаков каскада, сравнивается полученное значение порога с эталонным. Если порог меньше, это означает, что данный пиксель не прошел все признаки и не является левым верхним пикселем лица. Поэтому цикл обработки каскадов прерывается и осуществляется переход к обработке следующего пикселя изображения.

Алгоритм работы реализации Хааровского классификатора представлен на чертеже ГУИР.400201.062 ПД.1 .

## **6.2** Алгоритм работы планировщика

Работа планировщика состоит из трех этапов обработки.

Первый этап связан с функцией scan. Это основной этап работы планировщика. В его задачу входит подготовить массив индексов результирующего плана, по которым требуется записать координаты обрабатываемого в текущий момент элемента маски изображения. На практике данная функция является параллельной реализацией алгоритма последовательного суммирования элементов массива. Функция состоит из 2 действий: сканирование массива по возрастающей и сканирование массива по убывающей.

Схема работы сканирования по возрастающей изображена на рисунке 6.1 (в качестве примера выбрано d = 3). Для алгоритма, используемого в программе, число d равно 8, так как 2 в 8 степени – 256 (число тредов в конфигурации запуска ядра). Стрелочками указано, какой элемент с каким складывается и в какую ячейку записывается результат.

После того, как данный этап будет выполнен, осуществляется обнуление последнего элемента и организуется сканирование по убывающей. Схема процесса изображена на рисунке 6.2.

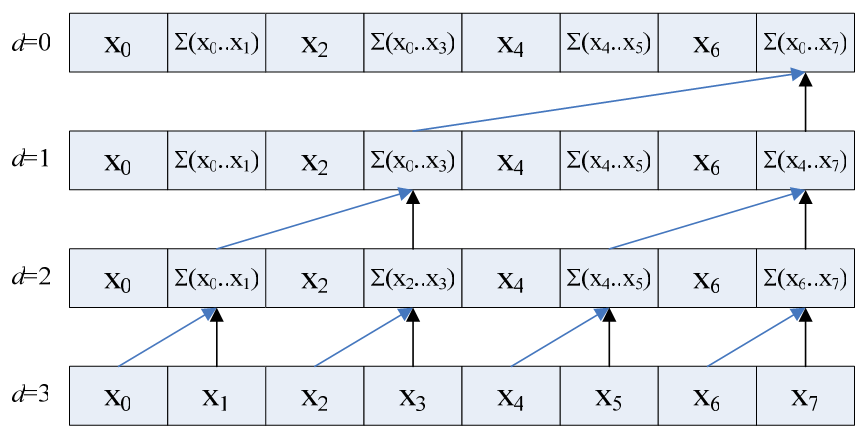


Рисунок 6.1 – Сканирования массива по возрастающей

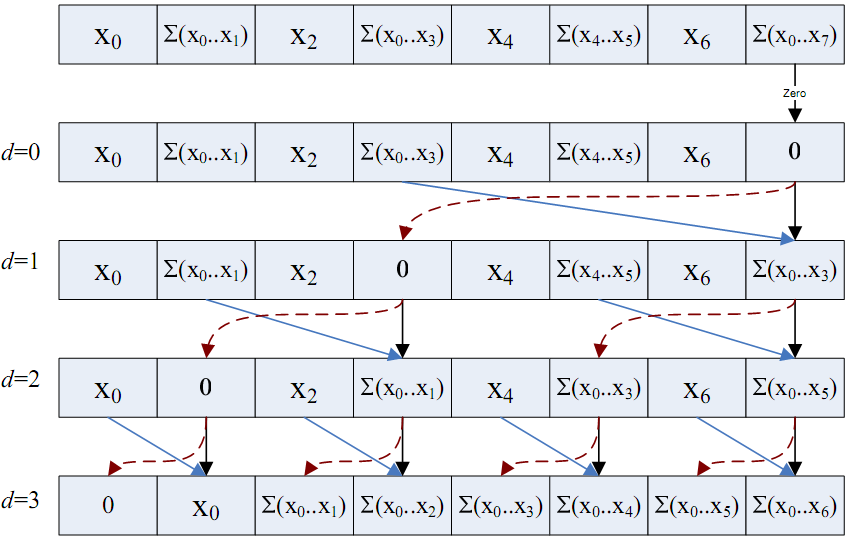


Рисунок 6.2 – Сканирование массива по убывающей

Стрелочками на рисунке 6.2 указывается направление суммирования, как это было сделано для рисунка 6.1. Стрелка со штрихом указывает, в какую позицию в массиве переносится значение соответствующей ячейки.

Как видно на рисунка 6.2, в результате исполнения данного алгоритма получается, что каждый последующий элемент равен сумме все предыдущих элементов массива. Так как такой обработке подвергается маска изображения, элементы которой либо 0, либо 1, то в результате элементы массива – индексы результирующего плана. Исходный текст программы приведен в приложении А.

Второй этап обработки связан непосредственно с формированием плана. Каждый тред проверяет соответствующий ему элемент маски изображения. Если значение элемента равно 1, то данный пиксель требуется дальше обрабатывать. В этом случае тред обращается к соответствующему элементу массива сумм и получает индекс плана. По данному индексу записывается координата элемента маски изображения. Данные действия выполняет функция DataToQueueB1\_32.

Так как, начиная с третьего каскада, также требуется учитывать существующий план, для данного каскада и всех последующих, которые будут обрабатываться с использованием графического процессора, применяется другая функция – DataToQueueB. Она полностью повторяет, функцию DataToQueueB1\_32, но дополнительно вводит проверку нескольких условий, с помощью которых учитывается существующий план. Исходный текст данной функции представлен в приложении Б.

Третий этап обработки связан с сортировкой плана. Данный процесс подробно описан в разделе 5.4, поэтому здесь описываться не будет.

## **6.3** Алгоритм работы обработчика каскада

Как отмечалось в разделе 5, разрабатывалось 4 функции-ядра для обработки очередного каскада. По структуре эти функции являются одинаковыми и различаются только индексация элементов, которые будут обрабатываться текущим ядром.

Первый этап работы функции изображен на рисунке 6.3. Изображение разбивается на множество блоков размером 32х32. Треды копируют соответствующий им элемент в выделенном блоке, а также элементы дополнительного участка, отмеченный штриховкой на рисунке. Это необходимо, чтобы была возможность обработать крайние элементы основного блока (выделен жирным). Тред копирует по 4 элемента, чтобы обеспечить 4-байтный тип данных и формировались транзакции.

Второй этап связан с обработкой выбранного каскада. Так как элементов для обработки больше, чем число тредов, организуется дополнительный цикл. Соответственно, каждый тред обрабатывает по 4 пикселя для 2-поточной версии, по 8 пикселей для 4-поточной версии и по 16 пикселей для 8-поточной версии ядер.

Учитывая структуру данных, описанную разделе 5.2, производится инициализация индексов элементов для обработки. Также, учитывая, что данные для 2 и 3 окон хранятся независимо друг от друга, то последовательно организуются циклы, в которых перебираются все признаки с данными окнами.



Рисунок 6.3 – Разбиение изображения на блоки

Число тредов, обрабатывающие один элемент маски изображения, влияют на данные циклы. Например, для 2-поточной версии haar\_next\_stage четные треды обрабатывают четные признаки, нечетные соответственно нечетные признаки. После того, как данный этап будет пройден, производится синхронизация. Складываются значения признака, полученного нулевым и первым, вторым и третьим тредами и так далее. В результате этого нечетные треды будут хранить значения признака для соответствующего элемента маски изображения. После того, как получен суммарный признак осуществляется его проверка с эталонным значением и принимается решение, прошел данный элемент или нет. Версии на 4- и 8- потоков работают аналогичным образом. Только все множество признаков разделяется между 4 и 8 тредами соответственно.

На третьем этапе осуществляется сохранение модифицированной маски изображения в глобальной памяти.

Исходные тексты функции haar\_first\_stage, обрабатывающей первые два каскада классификатора, haar\_next\_stage, обрабатывающей каскады в режиме 2 потока на один элемент маски изображения, и haar\_next\_stage\_4threads, обрабатывающей каскады в режиме 2 потока на один элемент маски, приведены в приложениях В, Г и Д соответственно.

## **6.4** Алгоритм работы преобразователя структуры классификатора

Как отмечалось в разделе 5.2, структура классификатора, предлагаемая библиотекой OpenCV не эффективна при применении GPU. Поэтому была разработана структура, которая минимизирует число обращений к памяти, ее описание приводилось в разделе 5.2. Функция преобразования SetCascadeToGPU исполняется один раз.

Вначале разработанная функция сканирует каскады классификатора из структуры OpenCV. При этом проверяется доступность всех данных, рассчитывается объем памяти, необходимый для хранения данных, требуемых GPU. В случае если разместить требуемый объем каскадов не удается, программа сообщает об ошибке и прекращает свою работу.

Если предыдущий этап выполнен успешно, каскады классификатора сканируются сначала и заполняется структура для графического процессора. При этом производятся оптимизационные расчеты (расчет ряда константных значений) для уменьшения требуемой памяти и упрощения алгоритма функции обработчика каскада. После того, как будет инициализирована вся структура, она копируется в константную память видеокарты.

## **6.5** Алгоритм работы функции восстановления маски изображения

Необходимость функции RestoreMask связано с тем, что некоторое число каскадов требуется обрабатывать с применением центрального процессора. Маска на графическом процессоре обрабатывается блоками в 1024 элемента, которые требуется перенести по 32 элемента на соответствующие позиции исходного изображения, также маска изображения для графического процессора выровнена до границы в 32, что также надо учитывать.

Данная задача решается посредством применения 4 циклов: 2 цикла перебирают блоки изображения по координате y и x соответственно, 2 других цикла обеспечивают перебор элементов внутри обрабатываемого блока. Так как ширина изображения известна, данных циклов достаточно, чтобы определить значение в маске изображения для GPU и соответствующее значение в маске для центрального процессора.

## **6.6** Инициализация CUDA

Перед тем, как использовать CUDA в вычислениях, требуется определить устройство, которое поддерживает данную архитектуру, и назначить его в качестве исполнителя программы. За эти действия отвечает функция InitCUDA. Вначале она получает число устройств, которые поддерживают требуемую архитектуру. После этого организуется перебор полученного числа устройств, для каждого из них получаются параметры. Сравнивается версия архитектуры CUDA для каждого из устройств с минимально допустимой версией для разрабатываемой программы. В случае, если найдена подходящая видеокарта, цикл перебора устройств прерывается и устанавливается флаг, что устройство найдено.

В конце работы функции проверятся флаг, указывающий на то, что устройство было найдено. Если флаг установлен в значение «истина», то устройство назначается в качестве исполнителя кода, иначе программа сообщает об ошибке и выходит.

Блок-схема алгоритма представлена на чертеже ГУИР.400201.062 ПД.2.

# **7** ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

Входные параметры программы проверяются на наличие ошибок. В частности, проверяется правильность путей к файлу для последующей обработки, путь к описанию каскада классификатора (рисунок 7.1). В случае обнаружения ошибки, выдается соответствующее сообщение и прекращается обработка фотографии.

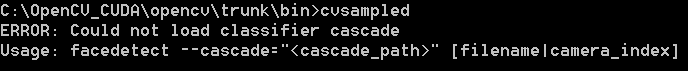


Рисунок 7.1 – Пример ошибки, когда не указан классификатор

Так как основной задачей при модернизации программы было уменьшение времени обработки, то дополнительно в процессе разработки проводились замеры времени и в дальнейшем использовались те варианты функций, которые обладали минимальным временем исполнения.

В качестве эталонного времени исполнения было выбрано время обработки изображения высокого разрешения (1280х720 пикселей) на центральном процессоре Intel Core i7-920. При этом время обработки составило 220 мс, что составляет около 4,5 кадров в секунду. Также для тестов использовалась видеокарта NVIDIA GTX 285.

При реализации планировщика были разработаны несколько вариантов алгоритма. За основу был взят разработанный базовый вариант. Эта реализация простое составление плана без дальнейших перестановок. Результат времени обработки представлен в таблице 7.1. Однако число конфликтов по доступу к банкам памяти (см. раздел 3) для данного случая указывает на то, что требуется оптимизация алгоритма, чтобы уменьшить число конфликтов и тем самым ускорить время обработки. Вначале была предпринята попытка сортировки внутри обрабатываемых блоков по 32х32 пикселя. В итоге удалось добиться некоторого прироста в производительности (см. базовый вариант с простой сортировкой в таблице 7.1). Однако в дальнейшем был найден способ, обеспечивающий максимальную производительность (см. базовый вариант с сортировкой и оптимизацией по банкам в таблице 7.1). В итоге время работы планировщика составило 1270 мкс. Именно этот алгоритм описан в разделе 6.2.

Также дополнительно проводились сравнения производительности универсальной реализации и при применении генератора кода. Результаты измерения времени обработки второго каскада классификатора сведены в таблицу 7.2. При этом реализация на основе генератора кода ускоряет процесс обработки каскада приблизительно в два раза.

Таблица 7.1 – Время работы планировщика в различных режимах

|  |  |  |
| --- | --- | --- |
| Режим работы | Время работы, мкс | Число конфликтов |
| 1. Базовый вариант | 3000 | 153335 |
| 2. Базовый вариант с простой сортировкой | 1592 | 103487 |
| 3. Базовый вариант с сортировкой и оптимизацией по банкам | 1270 | 72823 |

Таблица 7.2 – Время обработки второго каскада классификатора

|  |  |
| --- | --- |
| Режим работы | Время работы, мкс |
| 1. Универсальный вариант | 1435 |
| 2. Применяется генератор кода | 776 |

Как отмечалось в разделе 5, для обработки каскада классификатора используется 4 типа функций, различающихся между собой числом тредов, которые будут обрабатывать один пиксель изображения. Необходимость этого связана с резким уменьшением числа пикселей, которые требуется обрабатывать. График изменения числа пикселей в зависимости от каскада классификатора приведен на рисунке 7.2. В графике используется логарифмическая шкала с основанием 2.

Рисунок 7.2 – График изменения числа пикселей в зависимости от каскада классификатора

Как видно из графика на рисунке 7.2, вначале требуется обработать около 180 тысяч пикселей, ко второму каскаду требуется обработать около 130 тысяч пикселей. Однако уже к третьему каскаду требуется обработать всего около 30 тысяч и это число продолжает резко уменьшаться. Поэтому эффективно обеспечить нагрузку на все мультипроцессоры графического процессора одним и тем же ядром невозможно. Поэтому и применяются различные версии ядер.

Таблица 7.3 – Сравнительная таблица режимов работы программы

|  |  |
| --- | --- |
| Режим работы | Время работы, мс |
| 1. Реализация OpenCV | 220 |
| 2. Универсальный вариант | 82 |
| 3. Применяется генератор кода | 60 |

Сравнительная таблица работы программы до оптимизации и 2 режима работы после оптимизации сведены в таблицу 7.3. В результате всех оптимизаций было достигнуто время обработки одного каскада около 82 миллисекунд на изображении высокого разрешения. Реализация на основе сгенерированного кода обеспечивает еще более высокую производительность и составляет приблизительно 60 миллисекунд.

# **8** РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

## **8.1** Требования к аппаратному и программному обеспечению

Минимальные системные требования для нормальной работы программы:

* процессор Intel Pentium 4 или старше;
* видеокарта с графическим ядром nVidia GeForce 8-ой серии или новее;
* 512 мегабайт оперативного запоминающего устройства (ОЗУ).

Однако из-за поддержки программой многопоточности рекомендуется центральный процессор минимум с двумя ядрами или поддержкой технологии Intel Hyper-Threading.

Рекомендуется операционная система Windows XP (32- или 64-разрадная) или новее. Причем версия драйверов, установленных в системе, должна быть не младше 190.хх. Для проверки установленной версии драйверов требуется вызвать Панель управления NVIDIA, доступную, если щелкнуть правой кнопкой по рабочему столу. После этого перейти по вкладке Информация о системе (см. рисунок 8.1). В появившемся окне можно узнать точное название установленной видеокарты, ее параметры, а также версию установленных драйверов.

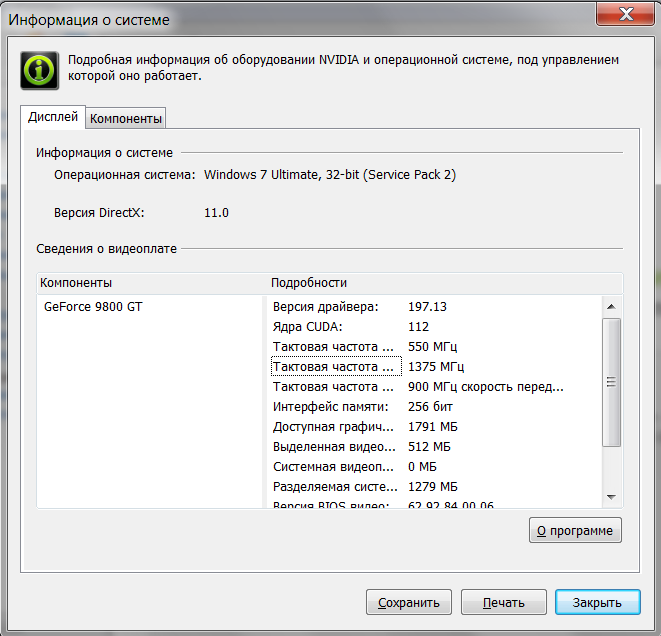


Рисунок 8.1 – Вкладка «Информация о системе»

Также должны быть установлены приложения для CUDA. В случае, если требуется модификация кода, то рекомендуется использовать среду разработки Microsoft Visual Studio 2008 с надстройкой CUDA vs Wizard 2.0. Данная надстройка позволяет упростить написание кода и компиляцию программ, использующих архитектуру CUDA.

В пакет приложений для CUDA входят следующие программы – CUDA Toolkit и CUDA SDK. Первой устанавливается CUDA Toolkit. По-умолчанию она устанавливается в папку C:\CUDA. При этом создаются переменные окружения (для папки по-умолчанию):

CUDA\_BIN\_PATH = C:\CUDA\bin

CUDA\_INC\_PATH = C:\CUDA\include

CUDA\_LIB\_PATH = C:\CUDA\lib

После этого устанавливается CUDA SDK. После установки данных программа можно проверить правильность установки приложения. Для этого последовательно нажимаем на клавиши WIN + R, последовательно вводим команды:

cmd

cd “%CDROM%\SystemAbout\Release\”

SystemAbout.exe

Вместо %CDROM% должна быть указана буква привода CD-ROM (например, E:).

Скриншот работы программы изображен на рисунке 8.2.

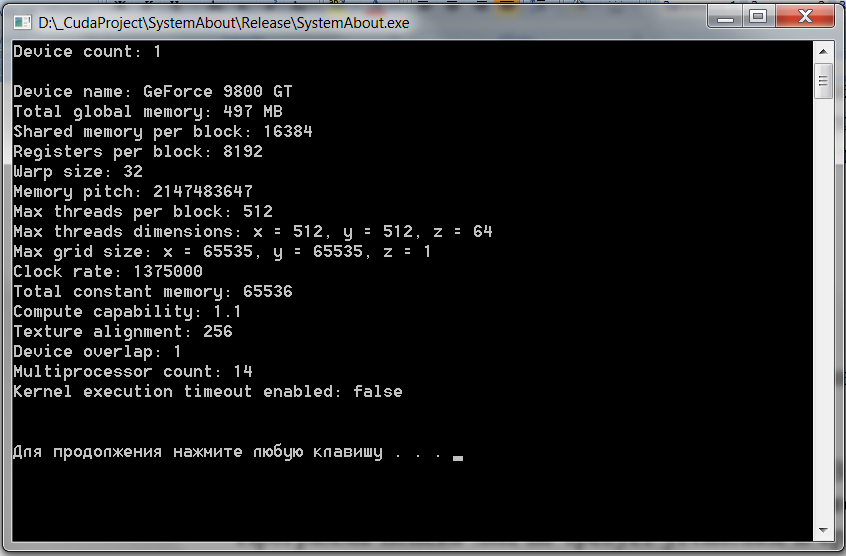


Рисунок 8.2 – Результат работы программы SystemAbout

В случае если поле Device count будет равно 0, но видеокарта на базе GeForce 8-ой серии, то либо драйвера видеокарты, либо приложения, необходимые для CUDA, установлены некорректно.

## **8.2** Руководство по работе с программой распознавания лиц

Программа поиска лиц не требует установки и сразу готова к работе. Она является консольным приложение, что позволяет автоматизировать процесс использования с помощью командных файлов (\*.bat, \*.cmd). Формат запуска программы следующий:

cvsample.exe –cascade=Cascade.xml Image.jpg

Флаг –cascade – указывает программе, что после символа равно следует путь к XML–описанию каскадов классификатора, а также название соответствующего файла. Image.jpg указывается путь к файлу­–изображению, включая само изображение

После запуска программы выводится информация об успешной инициализации CUDA, после чего начинается процесс обработки. Когда изображение будет обработано, выводится время обработки данного изображения и координаты найденных лиц, результат сохраняется в результирующий файл, имеющий название, аналогичное входному файлу, но к нему будет добавлено расширение «.GPU.jpg».

Все необходимые программы представлены на компакт-диске «Диск №1».

# **9** ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ПРОГРАММНОГО МОДУЛЯ ДЛЯ РЕАЛИЗАЦИИ ХААРОВСКОГО КЛАССИФИКАТОРА НА ПАРАЛЛЕЛЬНОЙ АРХИТЕКТУРЕ GPU

## **9.1** Характеристика проекта

В рамках разрабатываемого проекта осуществляется модернизация существующей библиотеки по распознаванию лиц с применением Хааровского классификатора. При этом конечной целью является значительное ускорение процесса поиска заданного объекта на изображении. Необходимость модернизации алгоритма связана с тем, что мощности современного центрального процессора недостаточно для обработки видеопотока высокого разрешения.

Проект разрабатывается в среде Microsoft Visual C++ с надстройкой CUDA. Данная надстройка позволяет задействовать вычислительную мощь графического процессора, который, как правило, в расчетах не применяется и, соответственно, во время работы программы простаивает. Однако процесс разработки алгоритмов для данного типа устройств связан с большими трудностями, так как GPU являются массово параллельными устройствами. Это приводит к тому, что применение классических алгоритмов является не эффективным. Поэтому дополнительно разрабатываются алгоритмы, которые можно исполнять одновременно большим числом потоков.

За основу была взята библиотека OpenCV, которая свободно распространяется в сети Интернет. Данная библиотека распространяется по лицензии BSD (Berkeley Software Distribution). Данный тип лицензии означает, что весь исходный код – собственность BSD, но все правки – собственность автора. Соответственно, разработки на основе данной библиотеки можно использовать как в свободно распространяемых продуктах, так и в коммерческих проектах.

В рамках дипломного проекта будет получено полностью готовое программное средство, которое допускается использовать в работе.

Смета затрат на разработку составлена по состоянию на 10 апреля 2010 года. Проект разрабатывается под заказ. Основным заказчиком является государственное учреждение БГУИР.

## **9.2** Экономический эффект у разработчика

Данный программный модуль относится к 1 категории сложности, так как обладает значительным распараллеливанием вычислений. При этом дополнительный коэффициент сложности равняется 1,12 (так как в программе присутствует значительное распараллеливание вычислений и оптимизационные расчеты). По степени новизны ПС относится к категории Б с коэффициентом новизны , так как ПО является развитием предыдущей версии, но в работе используется новый тип оборудования.

При разработке модуля используются существующие технологии и средства разработки, которые охватывают около 20 – 30% реализуемых функций, поэтому коэффициент использования стандартных модулей принимается равным 0,8.

**9.2.1** Определение объема и трудоемкости ПО

Для того, чтобы рассчитать плановую смету затрат на разработку ПО, требуется определить общий объем ПС (). В качестве единицы измерения возьмем количество строк исходного кода (Lines of Code, LOC). Объем разрабатываемых функций определяется по каталогу функций (см. таблицу 9.1).

Таблица 9.1 – Каталог функций ПО

|  |  |  |
| --- | --- | --- |
| Код функции | Наименование (содержание) функции | Объем функций (строк исходного текста) |
| 101 | Организация ввода информации | 150 |
| 102 | Контроль, предварительная обработка и ввод информации | 450 |
| 309 | Формирование файла | 1020 |
| 401 | Генерация рабочих программ | 3360 |
| 503 | Управление внешними устройствами и объектами | 5900 |
| 505 | Управление внешней памятью | 200 |
| 507 | Обеспечение интерфейса между компонентами | 970 |
|  | Итого | 12050 |

На основе общего объема ПС рассчитывается нормативная трудоемкость ПО (). Так как общий объем ПС = 12050, категория сложности ПО – 1, нормативная трудоемкость = 376. Тогда общая трудоемкость разработки определяется по формуле:

(9.1)

где – дополнительный коэффициент сложности; – коэффициент, учитывающий использование типовых программ и модулей; – коэффициент новизны.

Подставляя значения в 9.1, получим:

(человеко-дней)

На основе общей трудоемкости определяется плановое число разработчиков по формуле:

, (9.2)

где – численность исполнителей проекта; – общая трудоемкость разработки проекта (чел./дн.); – срок разработки проекта (лет); – эффективный фонд времени работы одного работника в течение года (дн).

Срок разработки проекта составляет один год ( = 0,5 год). Эффективный фонд времени определяется по формуле:

, (9.3)

где – количество дней в году; – количество праздничных дней в году; – количество выходных дней в году; – количество дней отпуска.

Подставляя значения в формулу 9.3, получим:

(дней).

При решении сложных задач с длительным периодом разработки ПО трудоемкость определяется по стадиям разработки: техническое задание (ТЗ), эскизный проект (ЭП), технический проект (ТП), рабочий проект (РП), внедрение (ВН). При этом трудоемкость разработки ПО отличается в зависимости от стадий. Общий вид формулы, по которой рассчитывает трудоемкость изготовления ПС:

, (9.4)

где – трудоемкость изготовления ПО на данной стадии; – нормативная трудоемкость; – коэффициент, учитывающий сложность изготовления ПО на выбранной стадии; – удельный вес трудоемкости выбранной стадии разработки ПО в общей трудоемкости разработки ПО; – коэффициент, учитывающий степень новизны ПО на выбранной стадии.

При этом для стадии «Рабочий проект» полученное значение трудоемкости изготовления ПО требуется умножить на . Общая трудоемкость определяется как сумма трудоемкости изготовления ПО на каждой из стадий разработки. Результаты расчетов трудоемкости по стадиям сведены в таблицу 9.2.

Таблица 9.2 – Расчет общей трудоемкости разработки ПО и численности исполнителей с учетом стадий

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Показатели | Стадии | | | | | Итого |
| ТЗ | ЭП | ТП | РП | ВН |
| 1. Коэффициенты удельных весов трудоемкости стадии разработки ПО () | 0,10 | 0,08 | 0,09 | 0,58 | 0,15 | 1,00 |
| 2. Коэффициент сложности ПО () | 1,12 | 1,12 | 1,12 | 1,12 | 1,12 |  |
| 3. Коэффициент, учитывающий использование стандартных модулей |  |  |  | 0,80 |  |  |
| 4. Коэффициент, учитывающий новизну ПО () | 0,90 | 0,90 | 0,90 | 0,90 | 0,90 |  |
| 5. Общая трудоемкость ПО (), чел./дн. | 37,90 | 30,32 | 34,11 | 175,86 | 56,85 | 335,04 |

На основе уточненной трудоемкости разработки ПО с использованием формулы 9.2 найдем общую численность разработчиков, которые требуются, чтобы вложиться в заданные сроки:

(человека).

**9.2.2** Расчет сметы затрат и цены заказного ПО

Основная статья расходов на создание ПО является заработная плата разработчиков проекта, то есть людей, непосредственно занимающихся разработкой.

Месячная тарифная ставка каждого специалиста () определяется по формуле:

, (9.5)

где – месячная тарифная ставка первого разряда (тыс.руб.); – тарифный коэффициент, соответствующий установленному тарифному разряду.

Часовая тарифная ставка :

, (9.6)

где – часовая тарифная ставка (тыс.руб); – среднемесячная норма рабочего времени в часах (составляет 170 часов).

Расчет месячных и часовых тарифных ставок сведен в таблицу 9.3.

Таблица 9.3 – Расчет месячных и почасовых тарифных ставок

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Должность | Количество ставок | Тарифный разряд | Тарифный коэффициент | Месячная тарифная ставка (руб.) | Часовая тарифная ставка (руб.) |
| Ведущий инженер-программист | 1 | 14 | 3,25 | 263 250 | 1548,53 |
| Инженер-программист 1-ой категории | 2 | 12 | 2,84 | 230 040 | 1353,18 |

Основная заработная плата исполнителей рассчитывается по формуле:

, (9.7)

где n – количество исполнителей; – часовая тарифная ставка -го исполнителя (ден.ед.); – количество часов работы в день, ч; – коэффициент премирования; – плановый фонд рабочего времени -го исполнителя (дн.).

(руб).

Дополнительная заработная плата () включает в себя оплаты отпусков и другие выплаты, предусмотренные законодательством. Определяется по формуле:

, (9.8)

где – норматив дополнительной заработной платы (10-20%).

(руб).

Отчисления в фонд социальной защиты определяется по формуле:

, (9.9)

где – норматив отчислений в фонд социальной защиты наделения.

Отчисления в фонд социальной защиты – 34%, отчисления в фонд социального страхования – 1%. Исходя из этого, получаем:

(руб).

Расходы по статье «Материалы» отражают расходы на магнитные носители, бумагу, тонер и прочие вещи, необходимые для разработки ПО. Нормы расхода материалов в суммарном выражении () определяется в расчете на 100 строк исходного кода. Сумма затрат на расходные материалы определяется по формуле:

, (9.10)

где – норма расхода материалов в расчете на 100 строк исходного кода ПО (руб); – общий объем ПО (строк исходного кода).

(руб).

Расходы по статье «Машинное время» включает оплату машинного времени, необходимого для разработки и отладки ПО. Норматив на 100 строк исходного кода () зависит от характера решаемых задачи и типа приложений. Расходы по этой статье определяются по формуле:

, (9.11)

где – цена одного машино-часа (руб); – общее время работы над проектом (часов).

(руб).

Расходы по статье «Научные командировки» () определяются по формуле:

, (9.12)

где – норматив расходов на командировки в целом по организации (%).

(руб).

Расходы по статье «Прочие затраты» () включают затраты на приобретение и подготовку специальной научно-технической информации и специальной литературы. Определяется по формуле:

, (9.13)

где – норматив прочих затрат в целом по организации.

(руб).

Затраты по статье «Накладные расходы» () связаны с необходимостью содержания аппарата управления, вспомогательных хозяйств и опытных производств. Определяются по формуле:

, (9.14)

где – норматив накладных расходов в целом по организации.

(руб).

Общая сумма расходов по смете ():

(руб).

Затраты на сопровождение и адаптацию ПО ():

, (9.15)

где – норматив расходов на сопровождение (%).

(руб).

Затраты на производство и продвижение программы определяется по формуле:

(9.16)

Подставляя значения в формулу 9.16, получим:

(руб).

Прибыль от создаваемого ПО определяется по формуле:

, (9.17)

где – уровень рентабельности ПО (%).

Подставляя значения в формулу 9.17, получим:

(руб).

Прогнозируемая цена ПО без налогов () определяется по формуле:

(9.18)

Подставляя значения в формулу 9.18, получим:

(руб).

Налог на добавленную стоимость ():

, (9.19)

где – норматив НДС (%).

(руб).

Прогнозируемая отпускная цена ():

(9.20)

Подставляя значения в формулу 9.20, получим:

(руб).

Прибыль за вычетом налога на прибыль составит рублей. Данная сумма остается при реализации собственнику разработки и представляет собой экономический эффект у собственника.

Таким образом, в результате оценки экономического эффекта у разработчика были получены следующие результаты:

* себестоимость проекта составила 14,4 миллиона рублей;
* прогнозируемая отпускная цена – 21,3 миллиона рублей;
* прибыль за вычетом налога составит 1,76 миллиона рублей.

## **9.3** Оценка экономического эффекта у пользователя

Для определения экономического эффекта от использования нового ПО у потребителя необходимо сравнить расходы по всем основным статьям сметы затрат на эксплуатацию нового ПО (расходы на заработную плату с начислениями, затраты на расходные материалы расходы на машинное время) с расходами по соответствующим статьям базового варианта. При этом за базовый вариант следует принять аналогичное программное средство. Создание нового ПО окажется экономически целесообразным лишь в том случае, если все капитальные затраты окупятся за счет получаемой экономии в ближайшее время (до 2 лет).

Однако из-за того, что отсутствует актуальная база для сравнения, расчет экономического эффекта у пользователя невозможен.

# **10** ОХРАНА ТРУДА, ЭКОЛОГИЧЕСКАЯ БЕЗОПАСНОСТЬ, ЭНЕРГОСБЕРЕЖЕНИЕ

## **10.1** Энергосбережение при разработке и эксплуатации программного модуля для реализации Хааровского классификатора на параллельной архитектуре GPU

Республика Беларусь не может полностью обеспечить себя требуемым количеством электроэнергии, поэтому вынуждена закупать энергоресурсы у стран-соседей. Ситуация усложняется тем, что цены на них постоянно растут, что делает страну зависимой от поставщиков. Для минимизации зависимости энергосбережение было объявлено как приоритетное направление государственной политики Республики Беларусь[21]. Однако белорусских нормативов и стандартов, которые можно было бы применить к компьютерной технике, найти не удалось. Поэтому за основу были взяты международный стандарт Energy Star[22] и аналитический обзор, посвященный эффективности энергопотребления современных комплектующих компьютера[23].

Несмотря на наличие международных стандартов, на такие показатели как энергопотребление и эффективность энергопотребления в сфере высоких технологий стали обращать внимание сравнительно недавно. Энергопотребление характеризует количество энергии, которое потребляет элемент компьютерной системы либо вся система целиком. Эффективность энергопотребления характеризует производительность, приходящуюся на 1 ватт[23], и определяется по формуле 10.1.

(10.1)

где – эффективность энергопотребления; – производительность системы, определяемая одним из способов; – потребляемая мощность, Вт.

Так как программа анализирует изображение с целью поиска лиц, то в качестве показателя производительности используется число фотографий, которые можно обработать за 1 секунду, и определяется по формуле 10.2.

(10.2)

где – время обработки одной фотографии, с.

Формула 10.2 получается, если выразить из пропорции 10.3, которая, в свою очередь, получается, если учесть, что время обработки одной фотографии постоянно и составляет секунд, а для обработки фотографий требуется 1 секунда.

. (10.3)

Подставив формулу 10.2 в 10.1, получим итоговую формулу определения эффективности энергопотребления:

*.* (10.4)

В зависимости от сферы применения важным становится либо значение энергопотребления, либо ее эффективность. Например, если компании требуется обрабатывать данные с максимальной скоростью, придется применять сверхсовременное оборудование, что, как правило, требует высокого энергопотребления, а ее эффективность уменьшается. Также следует учитывать, что развитие технологий быстро приводит к тому, что на рынке появляются комплектующие, которые обеспечивают туже производительность, но при этом показатель эффективности энергопотребления в этом случае оказываются значительно выше, чем в первом варианте.

Учитывая данные особенности, конечный пользователь должен внимательно подходить к выбору компьютерной системы. Использование комплектующих с высоким энергопотреблением становится выгодно только в достаточно узкой области применения, например, в научно-исследовательских целях, при обработке больших массивов аудио- и видеоинформации. При этом пользователю требуется учитывать не только высокое энергопотребление, но и обдумать схему отвода тепла, которое выделяют в окружающую среду устройства. Так как температура может достигать высоких значений, пассивные системы охлаждения применять нельзя, поэтому используются достаточно мощные вентиляторы, которые создают постоянный уровень шума и дополнительную вибрацию, что отрицательно сказывается на состоянии здоровья пользователя.

Существует два основных пути оптимизации эффективности энергопотребления – оптимизация алгоритма работы программы (в том числе за счет применения оборудования, которое раньше не применялось, но входило в состав компьютера) и замена комплектующих на аналогичные модели с более высоким показателем эффективности энергопотребления.

Конфигурация системы, на которой работает программа поиска лиц с применением Хааровского классификатора, представлена в таблице 10.1. В ней приведены основные элементы-потребители энергии. Также для них приводится приблизительная оценка энергопотребления, полученная на основе официальных сайтов компаний-разработчиков при максимальной нагрузке [24, 25]. Следует учитывать, что под максимальной потребляемой мощностью понимается количество энергии, которое потребляет устройство при максимальной нагрузке (например, при стопроцентной нагрузке центрального процессора Intel Core i7-920 энергопотребление составит максимум 130 ватт). При этом многие ведущие компании-разработчики предлагают технологии, которые позволяют в случае, если устройство не используется, переключать его в режим пониженного энергопотребления. Однако данные технологии поддерживаются только последними версиями операционных систем, поэтому не всегда используются на практике.

Таблица 10.1 – Основные элементы компьютерной системы и их потребляемая мощность

|  |  |
| --- | --- |
| Наименование | Максимальная потребляемая мощность, Вт |
| 1. Intel Core i7-920 | 130 |
| 2. nVidia GeForce GTX 285 | 205 |
| 3. Итого: | 335 |

Как видно из таблицы 10.1 два основных компонента системы при максимальной нагрузке уже потребляют около 335 ватт. При условии, что данная система работает 8-часовой рабочий день, получается, что в сутки центральный процессор и видеокарта потребляет при постоянной высокой нагрузке около кВт.

Использование данной конфигурации системы связано с тем, что центральный процессор обладает достаточно высокой производительностью, а процесс обработки изображения с помощью Хааровского классификатора достаточно сложен. В результате время обработки одной фотографии высокого разрешения (1280х720 пикселей) на данном компьютере при применении классического алгоритма Хааровского классификатора для распознавания лиц составило 220 мс, что является достаточно большим значением и не позволяет применять данный алгоритм для поиска лиц в реальном масштабе времени в видеопотоке высокого разрешения. Подставив полученные значения в формулу 10.4, эффективность энергопотребления составила:

.

Существенным недостатком данного алгоритма является то, что видеокарта GeForce GTX 285 остается незадействованной, но при этом потребляет достаточно большое количество энергии.

Целью дипломной работы была модернизация алгоритма Хааровского классификатора за счет использования мощности графического процессора. Использование данного типа устройств является достаточно эффективным с точки зрения энергопотребления, так как, несмотря на то, что видеокарты потребляют достаточно большое количество энергии, производительность у них значительно выше, чем у центрального процессора. В результате эффективность энергопотребления в зависимости от сравниваемых моделей может быть выше в 4-20 раз по сравнению с центральным процессором.

В результате модернизации алгоритма время обработки одного изображения высокого разрешения было уменьшено с 220 до 82 миллисекунд. Эффективность энергопотребления в этом случае составила (формула 10.4):

.

Оптимизация алгоритма позволила повысить эффективность энергопотребления в 2,68 раза. Таким образом, если пользователя устраивает время обработки до оптимизации (220 мс), то применение усовершенствованного алгоритма позволит использовать для расчетов систему со значительно более низким энергопотреблением.

Многие производители, кроме стандартного модельного ряда комплектующих, выпускают специальные версии устройств, относящиеся к категории Green Edition. При этом пользователю предлагается модель аналогичная стандартной версии, но за счет небольшого снижения скорости работы достигается экономия электроэнергии. Например, для видеокарт GeForce 9800 GT и GeForce 9800 GT Green Edition разница в производительности составляет приблизительно 1,25 раза в пользу стандартной версии, но при этом максимальное значение энергопотребления составляет 110 и 75 ватт соответственно (разница 1,47 раза). Пример системы с пониженным энергопотреблением представлен в таблице 10.2.

Таблица 10.2 – Основные элементы компьютерной системы и их потребляемая мощность

|  |  |
| --- | --- |
| Наименование | Максимальная потребляемая мощность, Вт |
| 1. Intel Core2Duo E8400 | 65 |
| 2. nVidia GeForce 9800 GT Green Edition | 75 |
| 3. Итого: | 140 |

Время обработки на компьютере с указанными в таблице 10.2 комплектующими составляет приблизительно 190 мс. При этом эффективность энергопотребления составила (формула 10.4):

.

Как видно из результатов измерения времени время обработки существенно не улучшилось по сравнению с классической реализацией алгоритма Хааровского классификатора (разница составила 30 мс). Эффективность энергопотребления немного превысила эффективность, полученную на первоначальной системе после оптимизации. Но при этом энергопотребление центрального процессора и видеокарты существенно сократилось (разница составляет приблизительно 195 ватт). В результате за сутки данные комплектующие при максимальной нагрузке потребляют около кВт. Это приблизительно в 2,39 раза меньше, чем у применяемой компьютерной системы до оптимизации. К сожалению, температура элементов компьютера по-прежнему достигает высоких значений, хотя и меньших, чем при использовании самых производительных компонентов. Поэтому применение вентиляторов все равно требуется, но допускается использование вентиляторов с пониженной мощностью, что положительно влияет на уровень шума и вибрацию.

Видеокарта и центральный процессор является основными потребителями энергии, однако, кроме них значительную часть энергии потребляет материнская плата, блок питания, жесткий диск и монитор. Разрабатываемая программа не может оказывать влияние на данные устройства, но, если пользователь решит пойти по второму пути оптимизации энергопотребления (замена комплектующих), то за счет обдуманного выбора компонентов можно дополнительно сэкономить электроэнергию.

Основными потребителями энергии на материнской плате являются чипсет и стабилизатор напряжения для центрального процессора. Некоторые компании, производящие материнские платы, разработали механизмы динамической регулировки фаз стабилизатора напряжения (например, у компании Gigabyte данная технология называется Dynamic Energy Saver, у компании ASUS – Energy Processor Unit[26]). В зависимости от нагрузки центральному процессору требуется разный уровень напряжения. Данные технологии отслеживают нагрузку на процессор и, как только она становится максимальной, включают все фазы стабилизатора напряжения, но, как только нагрузка уменьшается, число работающих фаз стабилизатора уменьшается, тем самым уменьшая потребляемую энергию.

Основным показателем эффективности энергопотребления для блока питания является коэффициент полезного действия (КПД). В зависимости от производителя КПД блока питания может изменяться в большом диапазоне значений, но блоки питания, отвечающие современные требованиям, имеют маркировку «80+». Это гарантирует, что КПД блока питания при стандартной нагрузке составляет не менее 80%.

Жесткий диск обладает не значительным энергопотреблением. Тем не менее, некоторые производители предлагают специальные версии, относящиеся к категории Green Edition и обладающие пониженным энергопотреблением. Экономия энергии достигается с помощью понижения скорости вращения пластин (как правило, 5400 оборотов в минуту вместо стандартных 7200). При этом скорости чтения и записи в дисках данного типа по-прежнему достаточно высокие. Дополнительно следует обращать внимание на число пластин в диске, так как их уменьшение приводит к уменьшению энергопотребления, так как не требуется тратить энергию на то, чтобы раскрутить пластины и поддерживать их постоянную скорость вращения.

При использовании монитора на базе электронно-лучевой (ЭЛТ) трубки, рекомендуется его замена на современный жидкокристаллический (ЖК). Стандарт TCO’06, регулирующий качество современных мониторов, накладывает жесткие требования на энергопотребление. В результате потребление ЖК мониторов в 2-4 раза меньше, чем ЭЛТ монитора с такой же диагональю.

# ЗАКЛЮЧЕНИЕ

Развитие технологий является непрерывным процессом. Постепенно становятся массовыми телевизоры и видеокамеры, поддерживающие высокое разрешение. Однако требования к оборудованию, которое будет обрабатывать видео высокого разрешения, предъявляются достаточно высокие с точки зрения производительности.

В результате модернизации программы и использования графического процессора удалось повысить производительность системы в 2,7 раза (с 220 до 82 миллисекунд). При этом можно обрабатывать видеопоток высокого разрешения со скоростью 12 кадров в секунду.

Однако этого недостаточно для полноценной обработки, так как для этого требуется обрабатывать как минимум 25 кадров в секунду, что составляет приблизительно 40 миллисекунд на одну фотографию. Это является основным недостатком программы.

Существуют дополнительные пути оптимизации программы. Например, в текущей реализации остаются незадействованными вторая видеокарта при ее наличии, а также нагрузка на центральный процессор значительно уменьшается, пока данные обрабатывает графический процессор. В дальнейших версиях программы можно устранить данные недостатки.

# СПИСОК ЛИТЕРАТУРЫ

[1] Viola, P. Rapid Object Detection using a Boosted Cascade of Simple Features. / P. Viola, M. Jones // Proc. of IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’01) – Los Alamitos, USA, 2001. – P. 511–520.

[2] Open Computer Vision Library [Электронный ресурс]. – Электронные данные. – Режим доступа: http://sourceforge.net/projects/opencvlibrary.

[3] Owens, J. GPU architecture overview / J. Owens // International Conference on Computer Graphics and Interactive Techniques (SIGRAPH’07) – San Diego, USA, 2007.

[4] Whitepaper NVIDIA’s Next Generation CUDA Compute Architecture: Fermi [Электронный ресурс] / NVIDIA. – Электронные данные. – Режим доступа: www.nvidia.com/content/PDF/fermi\_white\_papers/NVIDIA\_Fermi\_ Compute\_Architecture\_Whitepaper.pdf.

[5] Lienhart, R. An Extended Set of Haar-Like Features for Rapid Object Detection. // R. Lienhart, J. Maydt // Proc of IEEE International Conference on Image Processing (ICIP’02) – Rochester, New York, USA, 2002. – P.900–903.

[6] Robust Real-time Object Detection by Paul Viola and Michael Jones [Электронный ресурс]. – Электронные данные. – Режим доступа: <http://www.cs.tau.ac.il/research/chen.goldberg/Chen_Goldberg_on_Viola&Jones.ppt>

[7]. OpenCL. Подробности технологии [Электронный ресурс]. – Электронные данные. – Режим доступа: http://m.habrahabr.ru/post/72650/

[8] CUDA GPUs [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.nvidia.com/object/cuda\_gpus.html

[9] Характеристики видеокарт [Электронный ресурс]. – Электронные данные. – Режим доступа: http://nvworld.ru/kb/

[10] nVidia CUDA: тесты приложений на GPU для массового рынка [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.thg.ru/graphic/nvidia\_cuda\_test/print.html

[11]. NVIDIA CUDA — неграфические вычисления на графических процессорах [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.ixbt.com/video3/cuda-1.shtml

[12] Знакомство с NVIDIA CUDA, параллельные вычисления с помощью GPU в CG [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.render.ru/books/show\_book.php?book\_id=840&com\_start =20

[13] NVIDIA CUDA Programming Guide [Электронный ресурс] / NVIDIA. – Электронные данные. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/2\_3/toolkit/docs/NVIDIA\_ CUDA\_ProgrammingGuide\_2.3.pdf

[14] NVIDIA CUDA C. Programming Best Practices Guide. CUDA Toolkit 2.3 [Электронный ресурс] / NVIDIA. – /лектронные данные. – Режим доступа: http://developer.download.nvidia.com/compute/cuda/2\_3/toolkit/docs/NVIDIA\_ CUDA\_BestPracticesGuide\_2.3.pdf

[15] Kirk, D. Programming Massively Parallel Processors: A Hands-on Approach / D. Kirk, Wen-mei Hwu. – Riverport Ln: Elsevier Inc, 2010. – 256 p.

[16] NVIDIA CUDA Library Documentation [Электронный ресурс]. – Электронные данные. – Режим доступа: http://developer.download.nvidia.com/ compute/cuda/2\_3/toolkit/docs/online/index.html

[17] CUDA Zone [Электронный ресурс]. – Электронные данные. – Режим доступа: http://www.nvidia.ru/object/cuda\_home\_new\_ru.html.

[18] Группа Московского государственного университета им. М.В. Ломоносова по изучению CUDA [Электронный ресурс]. – Электронные данные. – Режим доступа: http://groups.google.ru/group/cudacsmsusu? lnk=srg&hl=ru.

[19] General-Purpose Computation on Graphics Hardware [Электронный ресурс]. – Электронные данные. – Режим доступа: http://gpgpu.org/.

[20]. The CUDA compiler driver: NVCC [Электронный ресурс]. – Электронные данные. – Режим доступа: http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc\_2.0.pdf

[21] Закон Республики Беларусь №190-3 от 15 июля 1998 г. «Об энергосбережении» [Электронный ресурс]. – Электронные данные. – Режим доступа: http://pravo.by/WEBNPA/text.asp?RN=H19800190

[22] Energy Star Computers [Электронный ресурс] / Energy Star. – Электронные данные. – Режим доступа: http://www.energystar.gov/index.cfm? fuseaction=find\_a\_product.showProductGroup&pgw\_code=CO

[23] Оптимизация энергопотребления ПК [Электронный ресурс] / Tom’s Hardware Guide. – Электронные данные. – Режим доступа: http://www.thg.ru/howto/energy\_efficient\_pc/print.html

[24] ARK [Электронный ресурс] / INTEL. – Электронные данные. – Режим доступа: http://ark.intel.com/

[25] Графические процессоры GeForce8 и более новые [Электронный ресурс] / Мир NVIDIA. – Электронные данные. – Режим доступа: http://nvworld.ru/kb/compare/?form=postgf8gpus&item=gfgtx285&item=gf9800gt&item=gf9800gtgreen&compare=Сравнить

[26] ASUS EPU против Gigabyte DES [Электронный ресурс]. – Электронные данные. – Режим доступа: http://ko.com.ua/node/35298

Палицын В. А. Технико-экономическое обоснование дипломных проектов: методическое пособие для студентов всех специальностей БГУИР, в 4 т. / В. А. Палицын. – Мн.: БГУИР, 2006. – Т.4. – 76 с.

# ПРИЛОЖЕНИЕ А

(обязательное)

Исходный текст функции scan

#define condition\_1(value) { \

\_\_syncthreads(); \

if (thid < value) \

{ \

getOffset(); \

sum[bi] += sum[ai]; \

} \

offset = offset << 1; \

} \

#define condition\_2(value) { \

offset = offset>>1; \

\_\_syncthreads(); \

if (thid < value) \

{ \

getOffset(); \

unsigned short t = sum[ai]; \

sum[ai] = sum[bi]; \

sum[bi] += t; \

} \

} \

\_\_device\_\_ unsigned short scan(unsigned short \*sum, unsigned short \*len )

{

int offset = 1;

int thid = threadIdx.x;

// build the sum in place up the tree

int ai = offset \* ( ( thid << 1 ) + 1 ) - 1;

int bi = offset \* ( ( thid << 1 ) + 2 ) - 1;

ai += CONFLICT\_FREE\_OFFSET(ai);

bi += CONFLICT\_FREE\_OFFSET(bi);

sum[bi] += sum[ai];

offset = offset << 1;

condition\_1( 128 );

condition\_1( 64 );

condition\_1( 32 );

condition\_1( 16 );

condition\_1( 8 );

condition\_1( 4 );

condition\_1( 2 );

condition\_1( 1 );

// scan back down the tree

// clear the last element

if (thid == 0)

{

int index = 2\*THREADS - 1;

index += CONFLICT\_FREE\_OFFSET(index);

\*len = sum[index];

sum[index] = 0;

}

// traverse down the tree building the scan in place

condition\_2( 1 );

condition\_2( 2 );

condition\_2( 4 );

condition\_2( 8 );

condition\_2( 16 );

condition\_2( 32 );

condition\_2( 64 );

condition\_2( 128 );

offset = offset>>1;

\_\_syncthreads();

getOffset();

unsigned short t = sum[ai];

sum[ai] = sum[bi];

sum[bi] += t;

\_\_syncthreads();

return 0;

}

# ПРИЛОЖЕНИЕ Б

(обязательное)

Исходный текст функции DataToQueueB

#define PROC\_BLOCK\_SIZE 1024

#define PROC\_BLOCK\_SIZE\_LOG 10

#define PROC\_BLOCK\_SIZE4 256

#define PROC\_BLOCK\_SIZE4\_LOG 8

#define THREADS 256

#define LOG\_NUM\_BANKS\_SHORT 5

#define CONFLICT\_FREE\_OFFSET(index) (((index) >> LOG\_NUM\_BANKS\_SHORT)<<1)

\_\_global\_\_ static void DataToQueueB(unsigned int \* mask , unsigned int \*old\_plan, unsigned int \*new\_plan )

{

\_\_shared\_\_ unsigned char src[PROC\_BLOCK\_SIZE];

\_\_shared\_\_ unsigned short res[PROC\_BLOCK\_SIZE];

\_\_shared\_\_ unsigned short \_old\_plan[PROC\_BLOCK\_SIZE];

\_\_shared\_\_ unsigned short sum[THREADS\*2+32];

\_\_shared\_\_ unsigned short len;

unsigned int \*ps = (unsigned int \*)src;

int base\_offset = \_\_umul24(gridDim.x, blockIdx.y) + blockIdx.x;

unsigned int mask\_pos = (base\_offset<<8) + (threadIdx.y<<4) + threadIdx.x;

unsigned int tmp = mask[mask\_pos];

ps[threadIdx.x] = ((tmp&0x80808080)>>7);

unsigned int old\_plan\_pos = (base\_offset<<9) + (threadIdx.y<<4) + threadIdx.x;

unsigned int \*\_old\_planI = (unsigned int \*)\_old\_plan;

\_old\_planI[threadIdx.x] = old\_plan[old\_plan\_pos];

\_old\_planI[threadIdx.x+256] = old\_plan[old\_plan\_pos+256];

\_\_syncthreads();

int i2 = threadIdx.x<<1;

int i4 = threadIdx.x<<2;

int bankOffset = CONFLICT\_FREE\_OFFSET(i2);

sum[i2 + bankOffset] = src[i4] + src[i4 + 1];

sum[i2 + bankOffset + 1] = src[i4 + 2] + src[i4 + 3];

res[i4] = 2048;

res[i4+1] = 2048;

res[i4+2] = 2048;

res[i4+3] = 2048;

\_\_syncthreads();

scan(sum, &len);

unsigned int bp = sum[i2 + bankOffset];

unsigned int c = src[i4];

if (src[i4])

res[bp] = i4;

if (src[i4 + 1])

res[bp + c] = i4 + 1;

bp = sum[i2 + bankOffset + 1];

c = src[i4 + 2];

if (src[i4 + 2])

res[bp] = i4 + 2;

if (src[i4 + 3])

res[bp + c] = i4 + 3;

\_\_syncthreads();

unsigned int device\_res\_pos = (base\_offset<<9) + threadIdx.x;

tmp = 0;

unsigned int res\_pos = threadIdx.x<<1;

if (res[res\_pos]<=1024)

tmp = \_old\_plan[res[res\_pos]];

else

tmp = 2048;

res\_pos += 1;

if (res[res\_pos]<=1024)

tmp += (\_old\_plan[res[res\_pos]]<<16);

else

tmp += (2048<<16);

new\_plan[device\_res\_pos] = tmp;

tmp = 0;

res\_pos = (threadIdx.x<<1) + 512;

if (res[res\_pos]<=1024)

tmp = \_old\_plan[res[res\_pos]];

else

tmp = 2048;

res\_pos += 1;

if (res[res\_pos]<=1024)

tmp += (\_old\_plan[res[res\_pos]]<<16);

else

tmp += (2048<<16);

new\_plan[device\_res\_pos + 256] = tmp;

}

# ПРИЛОЖЕНИЕ В

(обязательное)

Исходный текст функции haar\_first\_stage

\_\_global\_\_ static void haar\_first\_stage(unsigned int \*img, float\* sqsum, unsigned int \*res, unsigned int img\_width, unsigned int img\_height, unsigned int img\_buffer\_width, unsigned int sqsum\_buffer\_width, unsigned int cascade\_count)

{

\_\_shared\_\_ unsigned int \_img[36][36]; //(20+16) \* (20+16)

\_\_shared\_\_ float \_sqsum[16][16];

\_\_shared\_\_ unsigned char \_res[256];

int \_imgX = threadIdx.x;

int \_imgY = threadIdx.y;

unsigned int PosX = (blockIdx.x<<4) + threadIdx.x;

unsigned int PosY = (blockIdx.y<<4) + threadIdx.y;

int img\_pos = \_\_umul24(PosY, img\_buffer\_width) + PosX;

\_img[\_imgY][\_imgX] = img[img\_pos];

\_img[\_imgY][\_imgX+16] = img[img\_pos+16];

\_img[\_imgY+16][\_imgX] = img[img\_pos+(img\_buffer\_width<<4)];

\_img[\_imgY+16][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<4) + 16];

if (threadIdx.x<4 && PosX+32<img\_buffer\_width )

{

\_img[\_imgY][\_imgX+32] = img[img\_pos + 32];

\_img[\_imgY+16][\_imgX+32] = img[img\_pos+32 + (img\_buffer\_width<<4)];

}

if (threadIdx.y<4 && PosY+32<img\_height)

{

\_img[\_imgY+32][\_imgX] = img[img\_pos+(img\_buffer\_width<<5)];

\_img[\_imgY+32][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<5)+16];

}

if (threadIdx.x<4 && threadIdx.y<4 && PosX+32<img\_buffer\_width && PosY+32<img\_height )

{

\_img[\_imgY+32][\_imgX+32] = img[img\_pos+(img\_buffer\_width<<5)+32];

}

int sqsum\_pos = \_\_umul24(PosY, sqsum\_buffer\_width) + PosX;

\_sqsum[\_imgY][\_imgX] = sqsum[sqsum\_pos];

int res\_pos = (threadIdx.y<<4) + threadIdx.x;

\_res[res\_pos] = 255;

\_\_syncthreads();

if ((PosX + window\_STEPX + 3<img\_width) && (PosY + window\_STEPY + 3<img\_height))

{

unsigned char\* cur\_img = (unsigned char\*)&\_img[threadIdx.y][threadIdx.x];

float variance\_norm\_factor = \_sqsum[\_imgY][\_imgX];

//gpu\_cascade\* curr\_cascade = haar\_cascade;

gpu\_node\* curr\_node = haar\_nodes;

for(int k=0;k<cascade\_count && \_res[res\_pos] == 255 ;k++)

{

float stage\_sum = 0.0f;

int j = haar\_cascade[k].node2\_first;

gpu\_rect \*curr\_rec = &haar\_rects2[j];

for (int i=0; i<haar\_cascade[k].node2\_count;i++ )

{

float t = curr\_node->threashold \* variance\_norm\_factor;

int sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4);

float r = (sum\*haar\_rect\_weights2[j]);

curr\_rec++;

j++;

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4);

r += (sum\*haar\_rect\_weights2[j]);

stage\_sum += r < t ? curr\_node->a : curr\_node->b;

curr\_rec++;

j++;

curr\_node++;

}

j = haar\_cascade[k].node3\_first;

curr\_rec = &haar\_rects3[j];

for (int i=0; i<haar\_cascade[k].node3\_count;i++ )

{

float t = curr\_node->threashold \* variance\_norm\_factor;

int sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4);

float r = (sum\*haar\_rect\_weights3[j]);

curr\_rec++;

j++;

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4);

r += (sum\*haar\_rect\_weights3[j]);

curr\_rec++;

j++;

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4);

r += (sum\*haar\_rect\_weights3[j]);

stage\_sum += r < t ? curr\_node->a : curr\_node->b;

curr\_rec++;

j++;

curr\_node++;

}

if( stage\_sum < haar\_cascade[k].threashold )

{

\_res[res\_pos] = 0;

break;

}

else

\_res[res\_pos] = 255;

}

}

\_\_syncthreads();

unsigned int \*\_resI = (unsigned int \*)\_res;

if (threadIdx.y<4)

{

int resPos = ( (\_\_umul24(gridDim.x, blockIdx.y) + blockIdx.x)<<6 ) + res\_pos;

res[resPos] = \_resI[res\_pos];

}

}

# ПРИЛОЖЕНИЕ Г

(обязательное)

Исходный текст функции haar\_next\_stage

#define NODE2\_CYCLE( i\_add, curr\_rec\_add, j\_add, curr\_node\_add ) { \

for (int i = arrayPos; i<curr\_cascade->node2\_count; i+=i\_add ) \

{ \

float t = curr\_node->threashold \* variance\_norm\_factor; \

int sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4); \

float r = sum\*haar\_rect\_weights2[j]; \

curr\_rec++; \

j++; \

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4); \

r += (sum\*haar\_rect\_weights2[j]); \

stage\_sum += r < t ? curr\_node->a : curr\_node->b; \

curr\_rec += curr\_rec\_add; \

j += j\_add; \

curr\_node+=curr\_node\_add; \

} \

} \

#define NODE3\_CYCLE( i\_add, curr\_rec\_add, j\_add, curr\_node\_add ) { \

for (int i=arrayPos; i<curr\_cascade->node3\_count;i+=i\_add ) \

{ \

float t = curr\_node->threashold \* variance\_norm\_factor; \

int sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4); \

float r = sum\*haar\_rect\_weights3[j]; \

curr\_rec++; \

j++; \

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4); \

r += (sum\*haar\_rect\_weights3[j]); \

curr\_rec++; \

j++; \

sum = \*(unsigned int\*)(cur\_img + curr\_rec->p1) - \*(unsigned int\*)(cur\_img + curr\_rec->p2) - \*(unsigned int\*)(cur\_img + curr\_rec->p3) + \*(unsigned int\*)(cur\_img + curr\_rec->p4); \

r += (sum\*haar\_rect\_weights3[j]); \

stage\_sum += r < t ? curr\_node->a : curr\_node->b; \

curr\_rec+=curr\_rec\_add; \

j+=j\_add; \

curr\_node+=curr\_node\_add; \

} \

} \

#define PLAN\_ITEM(cycleInc, node2Inc, node3Inc, savePos) { \

if (plan\_item <= 1024) \

{ \

unsigned char\* cur\_img = (unsigned char\*)&\_img[(plan\_item>>5)][plan\_item&0x1F]; \

float variance\_norm\_factor = \_sqsum[plan\_item]; \

gpu\_node\* curr\_node = &haar\_nodes[haar\_start\_position];\

int j = curr\_cascade->node2\_first + (arrayPos<<1); \

gpu\_rect \*curr\_rec = &haar\_rects2[j]; \

NODE2\_CYCLE( cycleInc, node2Inc, node2Inc, cycleInc ); \

\

curr\_node = &haar\_nodes[curr\_cascade->node2\_count + haar\_start\_position]; \

j = curr\_cascade[0].node3\_first + \_\_umul24(arrayPos, 3); \

curr\_rec = &haar\_rects3[j]; \

NODE3\_CYCLE( cycleInc, node3Inc, node3Inc, cycleInc ); \

\

if ( ( arrayPos & 1 ) == 1) \

\_stage\_sum [savePos] = stage\_sum; \

} \

\_\_syncthreads(); \

} \

#define PLAN\_ITEM\_BRANCH\_2threads(base\_offset) { \

PLAN\_ITEM(2, 3, 4, plan\_pos); \

if (arrayPos == 0 && plan\_item <= 1024 ) \

{ \

stage\_sum = \_stage\_sum[plan\_pos] + stage\_sum; \

if( stage\_sum > curr\_cascade->threashold ) \

\_res[base\_offset + plan\_pos] = 255; \

} \

} \

\_\_global\_\_ static void haar\_next\_stage(unsigned int \*img ,float\* sqsum, unsigned int \*res, unsigned int img\_width , unsigned int img\_height, unsigned int img\_buffer\_width, unsigned int sqsum\_buffer\_width, unsigned int cascade\_count, unsigned int \*plan, unsigned short \*elementIndex)

{

\_\_shared\_\_ unsigned int \_img[52][52]; //(20+16) \* (20+16)

\_\_shared\_\_ float \_sqsum[32\*32];

\_\_shared\_\_ unsigned char \_res[256];

\_\_shared\_\_ unsigned short \_plan[256];

\_\_shared\_\_ float \_stage\_sum[128];

\_\_shared\_\_ unsigned short elementsIndex/\*[32]\*/;

int \_imgX = threadIdx.x;

int \_imgY = threadIdx.y;

unsigned int PosX = (blockIdx.x<<5) + threadIdx.x;

unsigned int PosY = (blockIdx.y<<5) + threadIdx.y;

int img\_pos = \_\_umul24(PosY, img\_buffer\_width) + PosX;

\_img[\_imgY][\_imgX] = img[img\_pos];

\_img[\_imgY][\_imgX+16] = img[img\_pos+16];

\_img[\_imgY+16][\_imgX] = img[img\_pos+(img\_buffer\_width<<4)];

\_img[\_imgY+16][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<4) + 16];

\_img[\_imgY][\_imgX+32] = img[img\_pos+32];

\_img[\_imgY+16][\_imgX+32] = img[img\_pos+32+(img\_buffer\_width<<4)];

\_img[\_imgY+32][\_imgX] = img[img\_pos+(img\_buffer\_width<<5)];

\_img[\_imgY+32][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<5) + 16];

\_img[\_imgY+32][\_imgX+32] = img[img\_pos+(img\_buffer\_width<<5) + 32];

if (threadIdx.x<4 && PosX+48<img\_buffer\_width )

{

\_img[\_imgY][\_imgX+48] = img[img\_pos + 48];

\_img[\_imgY+16][\_imgX+48] = img[img\_pos + 48 + (img\_buffer\_width<<4)];

\_img[\_imgY+32][\_imgX+48] = img[img\_pos + 48 + (img\_buffer\_width<<5)];

}

if (threadIdx.y<4 && PosY+48<img\_height)

{

\_img[\_imgY+48][\_imgX] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)];

\_img[\_imgY+48][\_imgX+16] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+16];

\_img[\_imgY+48][\_imgX+32] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+32];

}

if (threadIdx.x<4 && threadIdx.y<4 && PosX+48<img\_buffer\_width && PosY+48<img\_height )

{

\_img[\_imgY+48][\_imgX+48] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+48];

}

int sqsum\_pos = \_\_umul24(PosY, sqsum\_buffer\_width) + PosX;

\_sqsum[(\_imgY<<5) + \_imgX] = sqsum[sqsum\_pos];

\_sqsum[(\_imgY<<5) + \_imgX+16] = sqsum[sqsum\_pos + 16];

\_sqsum[((\_imgY+16)<<5) + \_imgX] = sqsum[sqsum\_pos + (sqsum\_buffer\_width<<4)];

\_sqsum[((\_imgY+16)<<5) + \_imgX+16] = sqsum[sqsum\_pos + (sqsum\_buffer\_width<<4) + 16];

int lpos = (threadIdx.y<<4) + threadIdx.x;

int \_base\_offset = \_\_umul24(blockIdx.y, gridDim.x)+blockIdx.x;

int basic\_plan\_pos = \_base\_offset<<9;

unsigned int plan\_pos = (lpos & 31) + ( ( (lpos & 192) >> 6 ) << 5);

unsigned int arrayPos = ( lpos & 32 ) >> 5;

gpu\_cascade\* curr\_cascade = &haar\_cascade[cascade\_count];

int haar\_start\_position = curr\_cascade->node\_position + arrayPos;

if (lpos == 0 )

elementsIndex = elementIndex[\_base\_offset];

\_\_syncthreads();

unsigned short elemCount = elementsIndex;

unsigned int saveOffset = (\_base\_offset<<8) + (threadIdx.y<<4) + threadIdx.x;

int baseOffset = 0;

int flag = 0;

for (int basic\_plan\_offset = 0; basic\_plan\_offset<elemCount; basic\_plan\_offset+=64)

{

if ( ( basic\_plan\_offset & 64 ) == 0 )

\_res[lpos] = 0;

if (lpos<128 && ( basic\_plan\_offset & 64 ) == 0)

{

unsigned int \*\_planI = (unsigned int\*)\_plan;

\_planI[lpos] = plan[basic\_plan\_pos + basic\_plan\_offset + lpos];

baseOffset = basic\_plan\_offset;

flag = 1;

}

\_\_syncthreads();

int offset = ( basic\_plan\_offset & 64 ) << 1;

unsigned short plan\_item = \_plan[plan\_pos + offset];

float stage\_sum = 0.0f;

PLAN\_ITEM\_BRANCH\_2threads( offset );

\_\_syncthreads();

unsigned int \*\_resI = (unsigned int \*)\_res;

if (threadIdx.y < 4 && ( basic\_plan\_offset & 64 ) > 0)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = \_resI[lpos];

flag = 0;

}

\_\_syncthreads();

}

unsigned int \*\_resI = (unsigned int \*)\_res;

if (threadIdx.y < 4 && flag == 1)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = \_resI[lpos];

//flag = 0;

}

baseOffset += 128;

//\_\_syncthreads();

// дополнительно требуется обнуление массива до конца

for ( ; baseOffset < 512; baseOffset += 128)

{

if (lpos < 128)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = 0;

}

}

}

# ПРИЛОЖЕНИЕ Д

(обязательное)

Исходный текст функции haar\_next\_stage\_4threads

#define PLAN\_ITEM\_BRANCH\_8threads(base\_offset) { \

PLAN\_ITEM(8, 15, 22, writePos); \

if ( (arrayPos & 1) == 0 && plan\_item <= 1024) \

{ \

int pos = (arrayPos & 7) >> 1; \

stage\_sum += \_stage\_sum [writePos]; \

\

if ( pos > 0) \

\_stage\_sum [ ( (pos - 1) << 5 ) + plan\_pos] = stage\_sum; \

} \

\_\_syncthreads(); \

\

if ( arrayPos == 0 && plan\_item <= 1024) \

{ \

stage\_sum = \_stage\_sum[plan\_pos] + \_stage\_sum[32 + plan\_pos] + \_stage\_sum[64 + plan\_pos] + stage\_sum; \

if( stage\_sum > curr\_cascade->threashold ) \

\_res[base\_offset + plan\_pos] = 255; \

} \

} \

\_\_global\_\_ static void haar\_next\_stage\_4threads(unsigned int \*img ,float\* sqsum, unsigned int \*res, unsigned int img\_width, unsigned int img\_height, unsigned int img\_buffer\_width, unsigned int sqsum\_buffer\_width, unsigned int cascade\_count, unsigned int \*plan, unsigned short \*elementIndex)

{

\_\_shared\_\_ unsigned int \_img[52][52]; //(20+16) \* (20+16)

\_\_shared\_\_ float \_sqsum[32\*32];

\_\_shared\_\_ unsigned char \_res[256];

\_\_shared\_\_ unsigned short \_plan[256];

\_\_shared\_\_ float \_stage\_sum[128];

\_\_shared\_\_ unsigned short elementsIndex;

int \_imgX = threadIdx.x;

int \_imgY = threadIdx.y;

unsigned int PosX = (blockIdx.x<<5) + threadIdx.x;

unsigned int PosY = (blockIdx.y<<5) + threadIdx.y;

int img\_pos = \_\_umul24(PosY, img\_buffer\_width) + PosX;

\_img[\_imgY][\_imgX] = img[img\_pos];

\_img[\_imgY][\_imgX+16] = img[img\_pos+16];

\_img[\_imgY+16][\_imgX] = img[img\_pos+(img\_buffer\_width<<4)];

\_img[\_imgY+16][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<4) + 16];

\_img[\_imgY][\_imgX+32] = img[img\_pos+32];

\_img[\_imgY+16][\_imgX+32] = img[img\_pos+32+(img\_buffer\_width<<4)];

\_img[\_imgY+32][\_imgX] = img[img\_pos+(img\_buffer\_width<<5)];

\_img[\_imgY+32][\_imgX+16] = img[img\_pos+(img\_buffer\_width<<5) + 16];

\_img[\_imgY+32][\_imgX+32] = img[img\_pos+(img\_buffer\_width<<5) + 32];

if (threadIdx.x<4 && PosX+48<img\_buffer\_width )

{

\_img[\_imgY][\_imgX+48] = img[img\_pos + 48];

\_img[\_imgY+16][\_imgX+48] = img[img\_pos + 48 + (img\_buffer\_width<<4)];

\_img[\_imgY+32][\_imgX+48] = img[img\_pos + 48 + (img\_buffer\_width<<5)];

}

if (threadIdx.y<4 && PosY+48<img\_height)

{

\_img[\_imgY+48][\_imgX] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)];

\_img[\_imgY+48][\_imgX+16] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+16];

\_img[\_imgY+48][\_imgX+32] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+32];

}

if (threadIdx.x<4 && threadIdx.y<4 && PosX+48<img\_buffer\_width && PosY+48<img\_height )

{

\_img[\_imgY+48][\_imgX+48] = img[img\_pos+\_\_umul24(img\_buffer\_width, 48)+48];

}

//\_\_syncthreads();

int sqsum\_pos = \_\_umul24(PosY, sqsum\_buffer\_width) + PosX;

\_sqsum[(\_imgY<<5) + \_imgX] = sqsum[sqsum\_pos];

\_sqsum[(\_imgY<<5) + \_imgX+16] = sqsum[sqsum\_pos + 16];

\_sqsum[((\_imgY+16)<<5) + \_imgX] = sqsum[sqsum\_pos + (sqsum\_buffer\_width<<4)];

\_sqsum[((\_imgY+16)<<5) + \_imgX+16] = sqsum[sqsum\_pos + (sqsum\_buffer\_width<<4) + 16];

int lpos = (threadIdx.y<<4) + threadIdx.x;

int \_base\_offset = \_\_umul24(blockIdx.y, gridDim.x)+blockIdx.x;

int basic\_plan\_pos = \_base\_offset<<9;

unsigned int plan\_pos = (lpos & 31) + (((lpos & 128) >> 7)<<5);

unsigned int arrayPos = (lpos & 96) >> 5;

int writePos = ( ( (arrayPos & 2) >> 1 ) <<6 ) + plan\_pos;

gpu\_cascade\* curr\_cascade = &haar\_cascade[cascade\_count];

int haar\_start\_position = curr\_cascade->node\_position + arrayPos;

if (lpos == 0 )

elementsIndex = elementIndex[\_base\_offset];

\_\_syncthreads();

unsigned short elemCount = elementsIndex;

unsigned int saveOffset = (\_base\_offset<<8) + (threadIdx.y<<4) + threadIdx.x;

int baseOffset = 0;

int flag = 0;

for (int basic\_plan\_offset = 0; basic\_plan\_offset<elemCount; basic\_plan\_offset+=32)

{

if ( ( basic\_plan\_offset & 127 ) == 0 )

\_res[lpos] = 0;

if (lpos<128 && ( basic\_plan\_offset & 127 ) == 0)

{

unsigned int \*\_planI = (unsigned int\*)\_plan;

\_planI[lpos] = plan[basic\_plan\_pos + basic\_plan\_offset + lpos];

baseOffset = basic\_plan\_offset;

flag = 1;

}

\_\_syncthreads();

int offset = ( basic\_plan\_offset & 0x60 ) << 1;

unsigned short plan\_item = \_plan[plan\_pos + offset];

float stage\_sum = 0.0f;

PLAN\_ITEM\_BRANCH\_4threads(offset);

\_\_syncthreads();

unsigned int \*\_resI = (unsigned int \*)\_res;

if (threadIdx.y < 4 && offset == 192)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = \_resI[lpos];

flag = 0;

}

\_\_syncthreads();

}

unsigned int \*\_resI = (unsigned int \*)\_res;

if (threadIdx.y < 4 && flag == 1)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = \_resI[lpos];

}

baseOffset += 128;

// дополнительно требуется обнуление массива до конца

for ( ; baseOffset < 512; baseOffset += 128)

{

if (lpos < 128)

{

int resPos = saveOffset + (baseOffset>>1);

res[resPos] = 0;

}

}

}

# ПРИЛОЖЕНИЕ Е

# ПРИЛОЖЕНИЕ Ж