САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Чирков Иван Викторович

# Исследование и оптимизация алгоритмов поиска соответствий в изображениях на массово-параллельных платформах

**Дипломная работа**

Допущен к защите

Зав. кафедрой:

д.ф.-м.н., профессор **Терехов А.Н.**

Научный руководитель:

**Пименов А. А.**

Рецензент:

**Плискин М. М.**

**Санкт-Петербург**
**2010**

# Оглавление

# 1. Введение

В современном мире большую часть хранимой и передаваемой информации составляет видео. Оно содержится на жёстких дисках компьютеров, DVD-дисках, портативных устройствах; транслируется в онлайне, распространяется через торрент и аналогичные сети. В связи с этим актуальна задача его обработки – сжатия, качественного увеличения разрешения, избавления от чересстрочности, удаления шума, увеличения плавности за счёт увеличения числа кадров в секунду. Все эти задачи можно решать с применением поиска соответствия в соседних кадрах.

На данный момент существует два концептуальных подхода в поиске соответствий в изображениях – «прямой» и «косвенный». «Прямой» подход основывается на поиске соответствий между отдельными пикселями или блоками пикселей, а «косвенный» - на соответствии характерных деталей, например, углов, рёбер, пятен. Среди «косвенных» методов стоит упомянуть метод Лукаса-Канады [1], основанный на градиентах пятен, метод Хорна-Шанка [2], основанный на условии константности яркости пятен и гладкости получаемого поля векторов движения, и метод Бакстона-Бакстона [3], основанный на модели движения рёбер в видеопоследовательности. Среди «прямых» методов наиболее популярным является метод сопоставления блоков. Характерной чертой этого подхода является то, что соответствия для блоков текущего кадра ищутся независимо друг от друга, что говорит о том, что этот метод должен эффективно реализовываться на массово-параллельных платформах.

Описанные выше задачи обработки видео крайне актуальны для обычных пользователей – конечных потребителей этого видео. Поэтому очень важно использовать массово-параллельную платформу доступную широким массам. Вычисления общего назначения на видеокартах – наиболее

распространённая на сегодняшний день группа массово-параллельных платформ доступных пользователям. Современные видеокарты содержат множество простых ядер, работающих параллельно на частотах свыше 1 Ггц, что говорит об их высокой эффективности в качестве аппаратной части массово-параллельной платформы. На данный момент актуальными платформами, использующими вычисления общего назначения на видеокартах, являются пиксельные программы OpenGL, OpenCL и CUDA. Наиболее зрелая и производительная из них – CUDA, созданная компанией Nvidia. В её основе лежит язык C, однако существуют решения, позволяющие встраивать код, написанный с использованием CUDA, в программы на .NET и JAVA, что позволяет без особого труда ускорять существующие продукты, переписывая трудоёмкие части с использованием CUDA.

Реализация поиска соответствий в изображениях с помощью алгоритма сопоставления блоков на массово-параллельной платформе CUDA позволит создавать высокоэффективные приложения обработки видео для обычных пользователей.

Целями данной работы являются: анализ алгоритма сопоставления блоков для переноса на массово-параллельную платформу, реализация алгоритма на платформе CUDA, исследование и реализация оптимизаций алгоритма на данной платформе, оценка оптимизаций.

# 2. Обзор существующих решений

Как уже упоминалось выше, на данный момент существует два концептуальных подхода в поиске соответствий в изображениях – «прямой» и «косвенный». «Прямой» подход основывается на поиске соответствий между отдельными пикселями или блоками пикселей, а «косвенный» - на соответствии характерных деталей. Говоря о «косвенных» методах, актуальными являются методы Лукаса-Канады [1], основанный на градиентах пятен, и метод Хорна-Шанка [2], основанный на условии константности яркости пятен и гладкости получаемого поля векторов движения, а также метод Бакстона-Бакстона [3], основанный на модели движения рёбер в видеопоследовательности. Однако существенными препятствиями для переноса данных алгоритмов на массово-параллельную архитектуру служит наличие большого числа зависимостей вычислений.

Среди «прямых» методов наиболее популярным является метод сопоставления блоков. Метод сопоставления блоков заключается в поиске наиболее подходящего блока последующего кадра для каждого блока текущего кадра. Критерием того, насколько один блок соответствует другому, может выступать или сумма квадратов разниц, или сумма абсолютных разниц, или взаимнокорреляционная функция. Очевидно, что полный перебор сопоставления всех блоков текущего кадра со всеми блоками последующего кадра слишком трудоёмок, поэтому применяются различные техники, позволяющие сузить набор кандидатов. В работе [4] рассмотрены методы поиска в три шага, двумерного логарифмического поиска, бинарного поиска, ортогонального поиска, иерархического поиска и других. Представленные оценки показывают, что наилучшие результаты демонстрируют поиск в три шага и иерархический поиск. Иерархический поиск строит несколько уровней изображения и на каждом этапе проводит

поиск полным перебором. Характерной чертой этих двух поисков является то, что соответствия для блоков текущего кадра ищутся независимо друг от друга, что говорит о том, что эти методы должны эффективно реализовываться на массово-параллельных платформах.

# 3. Обзор платформы

Для введения в архитектуру графических процессоров и программную среду CUDA будет сделан обзор платформы и соответствующих аппаратных решений. Перед этим в разделе 3.1 будет приведена краткая историческая справка для введения в вычисления общего назначения на графических процессорах. Программная среда CUDA используется для программирования графического процессора для вычислений общего назначения. Эта среда состоит из потоковой модели, описанной в разделе 3.2, и инструментов для компиляции. Для понимания подхода и особенностей программирования в этой среде в разделе 3.3 проведен обзор архитектуры графического процессора. В нем рассматриваются различные исполнительные блоки и типы памяти графического процессора. В разделе 3.4 проводится связь между программной средой CUDA и рассмотренной аппаратной архитектурой.

## 3.1 Графический процессор, как процессор для вычислений общего назначения

В последние годы демонстрируется интерес в программировании задач общего назначения на графических процессорах. С появлением специализированных решений от производителей графических процессоров область стала расти экспоненциально. Для введения в программирование графических процессоров будет сделан небольшой обзор истории этой области в секции 3.1.1 и обзор основных понятий в секции 3.1.2.

### 3.1.1 Краткий обзор истории архитектур графических процессоров

Современные трехмерные графические процессоры, в том виде, в котором они существуют сейчас, эволюционировали из графического конвейера с фиксированными функциями в программируемые параллельные процессоры. Программируемость графических процессоров начала развиваться около 2001 года с появлением программируемых вершинных процессоров [5]. Позднейшее введение программируемых пиксельных процессоров ещё увеличило их программируемость. В связи с растущей необходимостью в вычислительной мощности для трёхмерных игр производители добавляли вершинные и пиксельные процессоры в графические ускорители. Однако несбалансированность вершинных и пиксельных операций в трёхмерных сценах вынудила двигаться в сторону программируемого графического ускорителя. Таким образом, был создан унифицированный процессор, способный выполнять как вершинные, так и пиксельные операции. И хотя специализированные пиксельные и вершинные процессоры демонстрировали высокую скорость и эффективность сами по себе, графические ускорители с унифицированными процессорами демонстрировали большую производительность за счёт возможности динамически менять количество блоков, отведенных для вершинных и пиксельных операций в соответствии с требованиями приложения.

### 3.1.2 Введение в программирование общего назначения на графических процессорах

Введение графических ускорителей, использующих унифицированный процессор, ознаменовало начало программирования общего назначения на графических процессорах. Так как новая архитектура стала обладать высокой степенью программируемости, появилась возможность решать широкий круг

задач, названных задачами общего назначения, не связанных с построением трёхмерных сцен. В эти задачи входят задачи компьютерного зрения, обработки сигналов, финансовые задачи, физические и задачи потоковых медиа [6].

Хотя попытки решать задачи общего назначения на графических процессорах предпринимались и до унификации архитектуры, они не были успешны. Из-за отсутствия специализированного языка программирования разработчики были вынуждены использовать OpenGL. И хотя OpenGL даёт доступ к графическому процессору, от разработчика требуется очень много усилий и знаний для эффективного программирования графического процессора [7]. Для решения этой проблемы компания NVIDIA выпустила программную платформу CUDA (Compute Unified Device Architecture) для гибкого и удобного программирования на своих графических процессорах.

В рамках платформы был создан специализированный язык программирования. Этот язык является упрощённым языком программирования C с поточной моделью и специальными аппаратными функциями.

Компания ATI, прямой конкурент компании Nvidia, создала собственную платформу для вычислений общего назначения на графических процессорах - Close To Metal, которая позже была заменена StreamSDK. После создания обеими компаниями своих собственных платформ группа Khronos, владелец спецификации OpenGL, начала работу над открытым языком для вычислений общего назначения на графических процессорах, доступным для графических процессоров всех производителей. В начале 2009 года группа Khronos выпустила спецификацию OpenCL (Open Computing Language) [8]. На данный момент OpenCL имеет очень слабую программную поддержку, но постепенно ситуация улучшается.

В данной работе CUDA выбрана в качестве платформы вычислений общего назначения на графических процессорах, поскольку она является наиболее распространенной и лучше всего документированной.

## 3.2 Потоковая модель CUDA

Потоковая модель платформы CUDA, созданная для доступа к аппаратной многопоточности, предоставляет программисту возможность для распараллеливания последовательного кода. В рамках этой модели вводятся следующие понятия:

**ядро(kernel) –** небольшая программа, выполняемая множество раз на разных данных, SPMD (Single Program, Multiple Data) [5]. Ядро выполняется на графическом процессоре и в любой момент может исполняться только одно ядро.

**поток(thread) –** экземпляр ядра. Количество одновременно исполняемых потоков может достигать десятков тысяч.

**блок потоков(threadblock)** – каждый поток принадлежит некоторому блоку потоков. Внутри одного блока потоков возможна синхронизация потоков. Также все потоки внутри одного блока имеют доступ к общей разделяемой памяти, используемой в качестве кэша или для коммуникаций между потоками [5].

**сетка(grid)** – каждый блок потоков принадлежит некоторой сетке. Все блоки потоков внутри сетки представляют полную модель исполнения ядра. Внутри сетки блоки потоков не имеют возможностей для синхронизации или коммуникации [5].

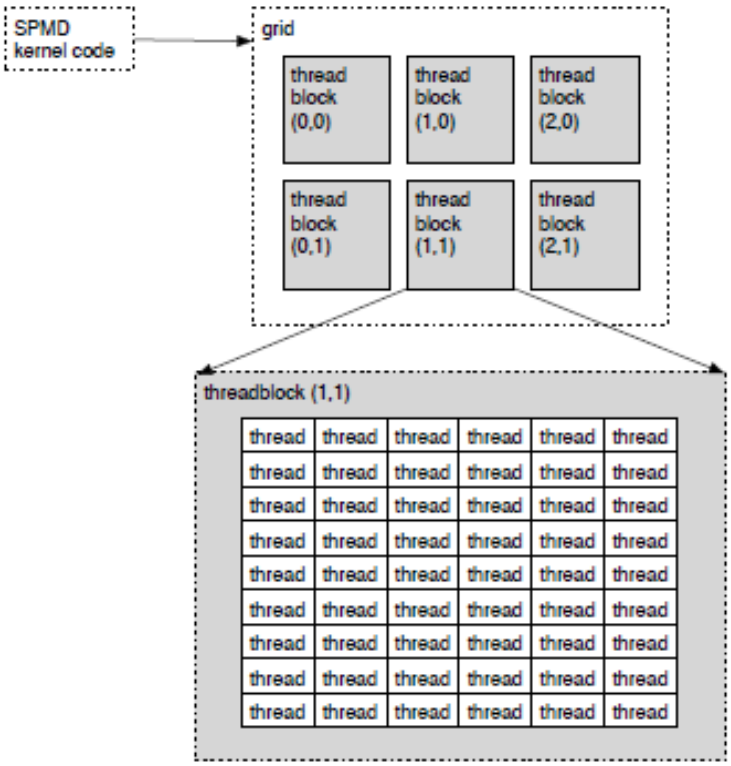


Рисунок 1: Потоковая модель CUDA

На рисунке 1 изображен пример организации потоков. В примере потоки и блоки потоков объединены в двумерные массивы, в общем же случае потоки внутри блока потоков могут быть объединены в одно-, двух- или трехмерный массив, а блоки потоков внутри сетки в одно- или двухмерный массив. При вызове ядра в коде программы программист явно указывает конфигурацию потоков для этого вызова.

Потоки могут быть представлены в виде программ, которые используют одни и те же инструкции, но следуют разными путями исполнения в зависимости от идентификаторов, хранящихся в специальных регистрах. Эти идентификаторы различны для разных потоков, а их значения соответствуют координатам потока внутри блока потоков и координатам блока потоков внутри сетки. Формирование потоков и их отображение на аппаратную часть рассматривается в разделе 3.4

## 3.3 Архитектура графического процессора

Для ознакомления с аппаратной архитектурой, в этом разделе приводится краткий обзор исполнительных блоков и типов памяти графического процессора. В качестве примера рассматривается структура архитектуры G80, первой поддерживающей платформу CUDA. Чипы с этой архитектурой используются на видеокартах GeForce 8, а также на некоторых платах Quadro и Tesla [6]. Более поздние архитектуры, поддерживающие CUDA, имеют схожую структуру, но могут отличаться в количестве исполнительных блоков, размерах памяти и т.п.

### 3.3.1 Иерархия исполнительных блоков

Рассмотрим устройство с точки зрения исполнительных блоков. Оно состоит из некоторого количества кластеров обработки текстур (Texture Processing Cluster - TPC), от одного до восьми для архитектуры G80 и от двух до десяти для архитектуры GT200. В архитектуре G80 каждый кластер обработки текстур содержит два потоковых мультипроцессора (Streaming Multiprocessors - SM), каждый их которых состоит из восьми исполнительных элементов (Processing Elements - PE) и двух блоков

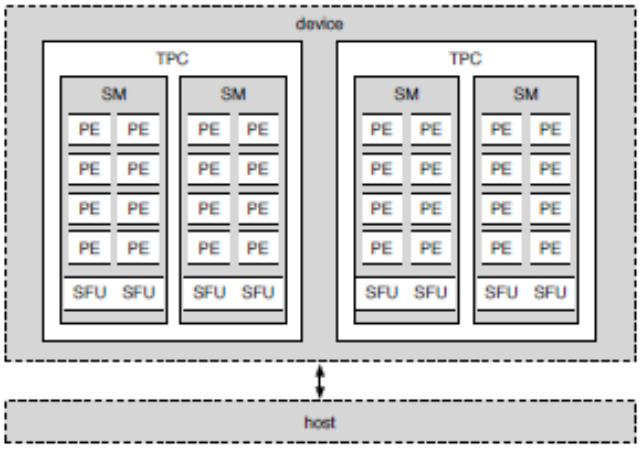


Рисунок 2: иерархия исполнительных блоков

специальных функций (Special Function Unit - SFU) [5].

Потоковый мультипроцессор может быть представлен как SIMD процессор, содержащий исполнительные элементы и блоки специальных функций – основные вычислительные блоки CUDA. Исполнительные элементы состоят из скалярных блоков умножения-сложения, а блоки специальных функций используются для трансцендентных функций, но могут выступать и в качестве умножителей чисел с плавающей запятой [5]. Кроме элементов, указанных на рисунке 2, архитектура G80 содержит части графического конвейера. Они не используются в CUDA, и потому исключены из рассмотрения.

### 3.3.2 Иерархия памяти

С точки зрения памяти каждый кластер обработки текстур содержит текстурный кэш (Texture cache). Каждый потоковый мультипроцессор внутри кластера обработки текстур содержит регистры (Register file), разделяемую память (Shared memory) и константный кэш (Constant cache).

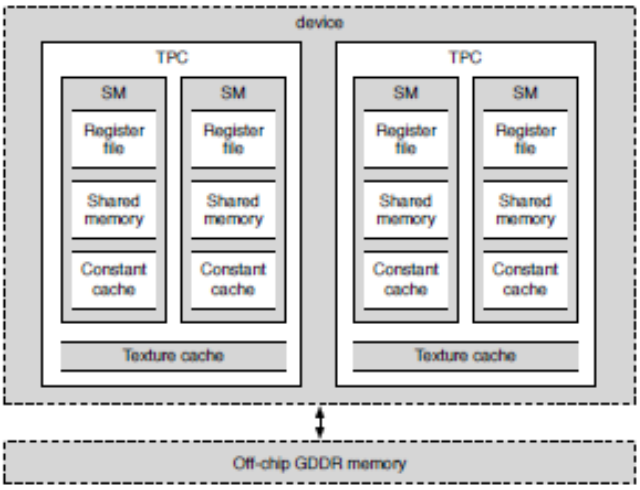


Рисунок 3: иерархия памяти

В платформе CUDA память делится на несколько типов, каждый со своим названием. Память вне мультипроцессоров содержит глобальную память, текстурную память и константную память. Для потоков глобальная

память доступна на чтение и запись, а текстурная и константная – только на чтение. Глобальная память не кэшируется, а текстурная и константная кэшируются в соответствующие кэши. Кэширование прозрачно с точки зрения программиста. Разделяемая память и регистры доступны для чтения и записи для потоков. Для части приложения, исполняемой на обычном процессоре, доступны только глобальная, текстурная и константная память на чтение и запись.

В таблице 1 приводятся зоны видимости, размеры и типичные задержки доступа для различных типов памяти.

| Тип памяти | Область видимости | Расположение | Размер | Задержка доступа |
|---|---|---|---|---|
| Регистры | Поток | SM | 8К/16К* значений | ~0 циклов |
| Разделяемая память | Блок потоков | SM | 16 Кб | ~0 циклов |
| Константный кэш | Программа | SM | 8 Кб | ~0 циклов |
| Текстурный кэш | Программа | TPC | 16 Кб | ~0 циклов |
| Глобальная память | Программа | Вне чипа | 0-1 Гб | 200-600 циклов |
| Текстурная память | Программа | Вне чипа | 0-1 Гб | 200-600 циклов |
| Константная память | Программа | Вне чипа | 64 Кб | 200-600 циклов |

* - 8192 в архитектуре G80, 16384 в архитектуре GT200

Таблица 1: типы и характеристики памяти

## 3.4 Отображение потоков на мультипроцессоры

После введения потоковой модели и рассмотрения аппаратной архитектуры необходимо понять, как они связаны, т.е. каким образом потоки и блоки потоков отображаются на исполнительные элементы графического процессора. Для этого вводится понятие варпа (warp).

Варп определяется как 32 потока, которые исполняются вместе [9]. При исполнении ядра потоки разбиваются на группы по 32 потока, которые

объединяются в варп и исполняются одним мультипроцессором. Так как варп состоит из 32 потоков, а мультипроцессор содержит 8 исполнительных элементов, исполнение одной простой инструкции для каждого потока внутри варпа занимает 4 цикла [10]. Далее выполняется эта же инструкция вторым варпом. Когда все варпы выполнят эту инструкцию, начинается выполнение следующей инструкции первым варпом. Этот процесс продолжается до тех пор, пока не будет встречена инструкция загрузки, которая должна ждать ответа от памяти вне чипа. В этот момент мультипроцессор откладывает исполнение этого варпа. Когда варп получает данные от памяти, он продолжает исполняться мультипроцессором как обычно. В случае ожидания данных от памяти всеми варпами, мультипроцессор простаивает.

Другой причиной простаивания мультипроцессора или его отдельных исполнительных элементов могут являться условные конструкции такие, как if, for, while и другие. Особенностью этих конструкций является то, что они предполагают наличие нескольких ветвей исполнения, выбор которых осуществляется на основе вычисления результатов условия. Как уже говорилось, потоки внутри варпа исполняются совместно, однако может возникнуть ситуация, когда потокам внутри варпа придётся идти по разным ветвям исполнения, поэтому такие ситуации мультипроцессор обрабатывает специальным образом. Когда исполнение доходит до условной конструкции, условие вычисляется для каждого потока и определяется, какую ветвь нужно исполнять для каждого потока. После этого ветви обрабатываются последовательно – сначала идёт исполнение первой ветви для соответствующих потоков, потом второй и так далее. Во время исполнения некоторой ветви мультипроцессором потоки, для которых требуется выполнить другую ветвь, простаивают и, соответственно, могут простаивать некоторые исполнительные элементы мультипроцессора.

Потоковая модель также накладывает ограничения на конфигурацию исполнения потоков. Как говорилось в разделе 3.2, потоки внутри блока потоков могут синхронизироваться и использовать разделяемую память. Поэтому блок потоков целиком исполняется одним мультипроцессором. Число блоков потоков одновременно исполняемых одним мультипроцессором определяется исходя из следующих ограничений:

- Во-первых, регистры мультипроцессора делятся между всеми потоками всех блоков потоков, одновременно исполняемых этим мультипроцессором. Количество регистров необходимых одному потоку определяется на этапе компиляции, а количество регистров мультипроцессора зависит от архитектуры (см. Таблица 1).
- Во-вторых, разделяемая память распределяется между одновременно исполняемыми блоками потоков. Объём разделяемой памяти необходимый одному блоку потоков определяется на этапе компиляции или при вызове ядра.
- Наконец, существует ограничение на максимальное количество одновременно исполняемых потоков и блоков потоков одним мультипроцессором. Мультипроцессор может одновременно исполнять не более 8 блоков потоков и не более 768 потоков для архитектуры G80. Для архитектуры GT200 максимальное число потоков увеличено до 1024.

Группа варпов, использующих одну и ту же разделяемую память и синхронизацию с помощью барьеров, называется согласованным массивом потоков (Concurrent Thread Array - CTA). Варпы внутри согласованного массива потоков исполняются параллельно за исключением барьеров синхронизации, устанавливаемых программистом [11]. Можно заметить, что

согласованный массив потоков имеет те же свойства, что и блок потоков – они имеют одинаковые ограничения на синхронизацию и разделяемую память – и поэтому могут считаться эквивалентными. Концепция блока потоков является взглядом с программисткой точки зрения, а согласованного массива потоков – с аппаратной.

Используя все описанные модели и концепции, важно осознать влияние числа потоков в блоке потоков и блоков потоков в сетке. Во-первых, число потоков в блоке потоков должно быть не менее 32 для формирования полноценного варпа и использования всех исполнительных элементов. Во-вторых, желательно чтобы число потоков в блоке потоков было кратно 32, чтобы не было незаполненных варпов. В-третьих, очень важно минимизировать число условных конструкций, чтобы избежать простаивания исполнительных элементов и последовательного исполнения разных ветвей. Но важнее всего максимизировать число потоков. Так как архитектура и окружение CUDA спроектированы таким образом, что исполнение варпа приостанавливается на время ожидания памяти, задержки памяти могут быть частично или полностью скрыты за счёт большого числа варпов. Интенсивность операций работы с памятью вместе с интенсивностью вычислительных операций позволяют определить достаточное число варпов, которое может быть вычислено аналитически [10], необходимое для полного сокрытия задержек доступа к памяти. Однако, увеличение числа варпов, а, следовательно, и потоков, поможет скрыть задержки доступа к памяти в независимости от структуры программы.

Графические процессоры в основном являются обладателями многопоточной архитектуры. Хотя архитектура G80 включает текстурный кэш, он слишком мал для большинства приложений, чтобы скрыть задержки доступа к памяти [12]. Вместо этого используется множество параллельно исполняемых потоков. Кроме многопоточной архитектуры существует

многоядерная архитектура. Многоядерная архитектура использует большой кэш для сокращения задержек обращения к памяти. Примером такой архитектуры является чип Intel Larrabee [13].

# 4. Основные результаты

В качестве алгоритма поиска соответствий в изображениях для переноса на массово-параллельную платформу CUDA был выбран алгоритм сопоставления блоков. Как было сказано выше, этот алгоритм хорошо подходит для массово-параллельной платформы за счёт множества независимых вычислений и отсутствия управляющих конструкций. Из рассмотренных модификаций хорошие результаты по сравнению с полным перебором кандидатов демонстрирует только иерархический поиск. Остальные же помимо худших результатов вносят много условных конструкций, что крайне нежелательно с точки зрения выбранной платформы. Иерархический поиск, в свою очередь, на каждом своем этапе использует полный перебор внутри некоторого окна для выбора лучшего кандидата на этом этапе. Поэтому в данной работе была рассмотрена реализация сопоставления блоков полным перебором, так как она может быть использована как сама по себе, так и на этапах иерархического поиска.

## 4.1 Описание алгоритма сопоставления блоков полным перебором

В алгоритме сопоставления блоков два изображения делятся на маленькие блоки. Далее для каждого блока в первом изображении необходимо поставить в соответствие некоторый блок второго кадра. Полученный вектор сдвига является результатом поиска.

Для каждого блока в первом кадре - назовём его исходным блоком - можно выделить три различных шага:

1. Во-первых, нужно сравнить исходный блок из первого кадра с некоторым множеством кандидатов из второго кадра. Обычно множество кандидатов ограничивается прямоугольным окном

вокруг позиции исходного блока. Например, если ограничить длину сдвига блока по горизонтали и вертикали тремя блоками, то получится окно 7 на 7 блоков, т.е. 49 кандидатов. Сравнивать блоки можно либо с помощью суммы абсолютных разниц, либо суммы квадратов разниц. При этом для лучших результатов сопоставления сравниваются не сами блоки, а выбирается некоторое окно, содержащее блок с некоторой окрестностью вокруг него.

2. Результаты сопоставления блоков говорят о похожести на исходный блок. Для выбора соответствующего блока для исходного необходимо выбрать блок с наилучшим результатом сопоставления. В случае суммы абсолютных разниц и суммы квадратов разниц необходимо выбрать блок с наименьшим результатом.
3. Смещение найденного блока относительного исходного является искомым вектором сдвига. Его необходимо записать для дальнейшего использования.

Вычисление результата сопоставления между исходным блоком и кандидатом происходит следующим образом: для каждого пикселя окна вокруг исходного блока значение его яркости вычитается из значения яркости соответствующего пикселя окна вокруг блока-кандидата. В зависимости от выбранной функции соответствия вычисляется либо абсолютное значение, либо квадрат этой разницы. Далее полученные значения суммируются для всех пикселей окна, формируя либо значение суммы абсолютных разниц, либо значение суммы квадратов разниц для исходного блока.

## 4.2 Перенос на платформу CUDA

Для реализации алгоритма на платформе CUDA необходимо произвести разбиение всех вычислений в алгоритме на независимые части, которые можно выполнять параллельно отдельными потоками. При этом важно учитывать рекомендации, полученные в разделе 3.4. То есть необходимо максимизировать число потоков, а число потоков в блоке потоков делать кратным 32. Поэтому был отвергнут вариант, когда один поток целиком обрабатывает исходный блок, вычисляя результаты сопоставления для всех кандидатов и выбирая из них минимальный, так как при этом блоки исходного изображения нужно объединять в группы кратные 32, что влечёт за собой использование небольшого количества блоков потоков и, соответственно, блоков. Другой крайний вариант - когда вычисление сопоставления для одного кандидата разбивается на несколько потоков, - тоже не очень хорош, так как придётся синхронизировать много потоков и суммировать результаты отдельных потоков для получения результата сопоставления для одного кандидата. Поэтому был выбран вариант являющийся средним между этими двумя – один блок потоков обрабатывает один блок исходного изображения, а один поток этого блока потоков вычисляет результат сопоставления с одним кандидатом. Подробнее этот способ описан ниже.

Вначале изображения разбиваются на отдельные блоки, для примера 2 на 2 пикселя. Создаётся сетка с числом блоков потоков равным числу полученных блоков в изображении. Так как блок изображения соответствует блоку потоков, действия для блока потоков можно описать следующим образом, в соответствии с шагами из раздела 4.1:

1. Во-первых, нужно сравнить исходный блок из первого кадра с каждым кандидатом из второго кадра. Для этого в каждом блоке

потоков создаётся число потоков равное числу кандидатов. Каждый поток сравнивает исходный блок с кандидатом, то есть вычисляет сумму абсолютных разниц или сумму квадратов разниц.

2. Далее необходимо найти кандидата с наименьшим результатом сопоставления. Для этого каждый поток сохраняет вычисленное им значение в разделяемую память. После этого необходимо выбрать из этого массива минимальное значение. Это можно эффективно сделать с помощью параллельной редукции [14].
3. После нахождения минимального значения среди результатов сопоставления кандидатов с исходным блоком необходимо вычислить вектор сдвига и сохранить его. Делать это может только один поток, поэтому остальные в этот момент будут простаивать.

## 4.3 Наивная реализация алгоритма

Для начала было решено перенести алгоритм сопоставления блоков полным перебором с организацией потоков из предыдущего раздела самым простым способом. То есть без применения оптимизаций, связанных с использованием разделяемой памяти, текстурной памяти и других.

В качестве входных данных были использованы два последовательных кадра из видео разрешения 640 на 480 пикселей. Максимальный сдвиг в каждом направлении был ограничен тремя блоками, что даёт 49 кандидатов. Ширина окна сравнения для блока была выбрана, как 1/20 ширины кадра, что дало окно сравнения 32 на 16 пикселей. Все тесты проводились на видеокарте GeForce GTX 275, которая содержит 240 ядер (30 мультипроцессоров), работающих на частоте 1404 МГц.

В итоге реализации описанной модели с приведенными параметрами без оптимизаций был получен следующий код:

```
__device__ void cudaSubSearch_first(
        unsigned char* pYIn0, int pYIn0offset,
        unsigned char* pYIn1, int pYIn1offset,
        int iWidth,
        int iHeight,
        int iStride,
        int iWindowWidth,
        int iWindowHeight,
        int *pMVX,
        int *pMVY,
        int *pSAD
        ) {
    __shared__ int lSAD[49];
    int iSAD = 0;
    int x, y, index, row;

    index = threadIdx.y*blockDim.x + threadIdx.x;

    int offset;
    offset = (-(iWindowHeight>>1))*iStride - (iWindowWidth>>1);

    y = threadIdx.y - 3;
    x = threadIdx.x - 3;

    int offset0 = pYIn0offset + offset;
    int offset1 = pYIn1offset + offset + y*iStride + x;
    for (; iWindowHeight > 0; iWindowHeight--) {
        for (int i = 0; i < iWindowWidth; i++) {
            iSAD += abs(pYIn0[offset0+i] - pYIn1[offset1+i]);
        }
        offset0 += iStride;
        offset1 += iStride;
    }

    lSAD[index] = iSAD;
```

```
        __syncthreads();

        __shared__ int minI[7];
        if (index < blockDim.x) {
            int st = 1 + index*blockDim.x;
            int fn = st+blockDim.x-1;
            minI[index] = st-1;

            for (int i = st; i < fn; i++) {
                if (lSAD[i] < lSAD[minI[index]]) {
                    minI[index] = i;
                }
            }

            if (index == 0) {
                for (int i = 1; i < blockDim.x; i++) {
                    if (lSAD[minI[i]] < lSAD[minI[0]]) {
                        minI[0] = minI[i];
                    }
                }
                *pMVX += (minI[0]%blockDim.x) - 3;
                *pMVY += (minI[0]/blockDim.x) - 3;
                *pSAD = lSAD[minI[0]];
            }
        }
    }
```

Прокомментируем этот код. `pYIn0` – указатель на первое изображение, `pYIn0offset` – смещение обрабатываемого пикселя относительно начала изображения. Далее аналогичные параметры для второго изображения. Далее идут параметры изображения, где `iStride` – ширина расширенного изображения. Работа велась с расширенным изображением для упрощения обработки краевых точек изображения. `PMVX`, `pMVY` и `pSAD` – указатели на вектора и результат сравнения для обрабатываемого пикселя. `threadIdx` – уникальный идентификатор потока, обозначающий его номер внутри блока

потоков. Каждый поток вычисляет значение SAD для своего смещения и сохраняет его в разделяемую память. После этого выбирается смещение с наименьшим значением SAD и вектор для данного пикселя сохраняется.

Данное ядро запускалось с числом блоков потоков в сетке равным числу пикселей в изображении и числом потоков в блоке потоков равным числу кандидатов, то есть 49. Время работы данной реализации приведено в Таблице 2.

| Реализация | Время работы, мс |
|---|---|
| Без оптимизаций, 7x7 потоков в блоке | 1350 |

Таблица 2: время работы реализации

Однако, в данной реализации число потоков в блоке потоков не кратно 32, что приводит к неравномерному распределению потоков блока потоков по варпам, что, в свою очередь, неэффективно загружает исполнительные блоки. Поэтому было решено использовать 64 потока в блоке потоков. Для потоков, отвечающих за лишние смещения, вычисления не производились, а SAD присваивалось заведомо большое значение. Время работы такой реализации приведено в Таблице 3.

| Реализация | Время работы, мс |
|---|---|
| Без оптимизаций, 8x8 потоков в блоке | 1025 |

Таблица 3: время работы реализации

Теперь, когда число элементов в промежуточном массиве со значениями результатов сравнения исходного блока со всеми кандидатами стало степенью 2, появилась возможность применить параллельную

редукцию для поиска минимального. Она была реализована следующим образом:

```
__shared__ int minI[64];
 minI[index] = index;
 if (index < 32) {
     index = index+index;
     if (lSAD[index+1] < lSAD[index])
         minI[index] = index+1;
 }
 if (index < 32) {
     index = index + index;
     if (lSAD[minI[index+2]] < lSAD[minI[index]])
         minI[index] = minI[index+2];
 }
 if (index < 32) {
     index = index + index;
     if (lSAD[minI[index+4]] < lSAD[minI[index]])
         minI[index] = minI[index+4];
 }
 if (index < 32) {
     index = index + index;
     if (lSAD[minI[index+8]] < lSAD[minI[index]])
         minI[index] = minI[index+8];
 }
 if (index < 32) {
     index = index + index;
     if (lSAD[minI[index+16]] < lSAD[minI[index]])
         minI[index] = minI[index+16];
 }
 if (index == 0) {
     if (lSAD[minI[32]] < lSAD[minI[0]]) minI[0] = minI[32];
     *pMVX += (minI[0]%8) - 4;
     *pMVY += (minI[0]/8) - 4;
     *pSAD = lSAD[minI[0]];
 }
```

Данная оптимизация не сильно уменьшает количество сравнений для поиска минимума, но позволяет ещё уменьшить время работы.

| Реализация | Время работы, мс |
|---|---|
| Без оптимизаций, 8x8 потоков в блоке, параллельная редукция | 1000 |

Таблица 4: время работы реализации

Далее было решено использовать оптимизации для сокращения обращений к глобальной памяти. Все потоки внутри блока потоков обращаются к одному и тому же исходному блоку, а блоки кандидаты имеют большие области пересечения.

## 4.4 Использование текстурной памяти

В разделе 3.3.2 при описании типов памяти платформы CUDA было отмечено, что программе для передачи данных на видеокарту, доступна и текстурная память. Её отличие от глобальной памяти заключается в том, что она доступна только на чтение, но при этом чтение из этой памяти кэшируется. Как уже было замечено выше, многие потоки в одном блоке потоков по множеству раз обращаются к одним и тем же ячейкам глобальной памяти. Также в разделе 3.3.1 при обзоре архитектуры чипа было отмечено, что в одном кластере обработки текстур содержится несколько мультипроцессоров, что тоже должно оказать положительное эффект, так как данные, используемые разными блоками потоков, также имеют пересечения.

Для использования текстурной памяти необходимо создать переменные специального типа и связать их с уже существующими областями глобальной памяти. После этого обращения к ячейке текстуры `texYIn` с индексом `index`

происходят с помощью функции `tex1Dfetch() - tex1Dfetch(texYIn, index)`. Таким образом, текстурная память является оберткой глобальной памяти.

Для обоих изображений были созданы текстуры, связаны с областями памяти, в которых лежат эти изображения, а все обращения к глобальной памяти изображений были заменены на описанные конструкции. Результаты этих изменений можно видеть в Таблице 5.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, 8x8 потоков в блоке, параллельная редукция | 354 |

Таблица 5: время работы реализации

Использование текстурной памяти без особых затрат позволило ускорить программу почти в 3 раза. Но, к сожалению, механизм кэширования текстурной памяти неподвластен программисту, поэтому нельзя с уверенностью утверждать, что данные кэшируется оптимальным образом. Чтобы получить больший контроль над переиспользованием данных, было решено воспользоваться для этих целей разделяемой памятью.

## 4.5 Использование разделяемой памяти

Разделяемая память находится внутри мультипроцессора. Это делает её доступной для всех потоков внутри блока потоков, а время доступа к ней сравнимо с временем доступа к регистрам. Данные преимущества делают её идеальным кандидатом для использования в качестве управляемого кэша глобальной памяти.

Для указания, что некоторая переменная будет храниться в разделяемой памяти, нужно перед её именем написать ключевое слово `__shared__`. После этого с ней можно работать как с обычной переменной, но

необходимо помнить, что доступ к ней будут иметь все потоки. Поэтому, чтобы не происходило ситуаций, когда какой-то поток пытается читать из разделяемой памяти, до того как другой поток записал туда необходимые данные, в CUDA существуют средства барьерной синхронизации потоков внутри блока потоков. Когда в коде встречается инструкция `__syncthreads()`, исполнение потоков приостанавливается, пока все потоки данного блока потоков не достигнут этой инструкции.

Для начала было решено загрузить окно исходного блока целиком в разделяемую память и заменить обращения к текстуре на обращения к разделяемой памяти. Для этого был создан массив в разделяемой памяти размером с окно исходного блока, а перед циклом вычисления SAD была вставлена загрузка из текстуры в этот массив. Результаты применения данной оптимизации приведены в Таблице 6.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, разделяемая память для окна исходного блока, 8x8 потоков в блоке, параллельная редукция | 268 |

Таблица 6: время работы реализации

Данный результат говорит о том, что предположение о неоптимальном кэшировании данных в текстурном кэше было верным. Это значит, что окна блоков кандидатов также стоит попытаться сохранить в разделяемой памяти.

В данной реализации рассматривается 8 смещений по горизонтали и 8 смещений по вертикали, а окно вокруг блока имеет размер 32 на 16 пикселей. Поэтому объединение окон блоков кандидатов лежит внутри прямоугольной области размером 40 на 24 пикселя. Как и для окна исходного блока, для данной области был создан массив в разделяемой памяти. Так как ширина данного массива 40, а доступно 64 потока, было опробовано два способа загрузки. В первом случае для загрузки проверялось, что номер потока менее

40, и загрузка производилась только этими потоками. Во втором случае каждый поток загружал ячейку с номером равным номеру потока по модулю 40. Результаты обоих способов приведены в Таблице 7.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, разделяемая память для окна исходного блока и для окон блоков кандидатов (с модулем), 8x8 потоков в блоке, параллельная редукция | 405 |
| Текстурная память, разделяемая память для окна исходного блока и для окон блоков кандидатов (с условием), 8x8 потоков в блоке, параллельная редукция | 420 |

Таблица 7: время работы реализации

К сожалению, в таком виде загрузка данных для окон блоков кандидатов ускорения не принесла. Это связано с тем, что все потоки загружают достаточно много данных и скрыть задержки доступа к памяти за счёт вычислений не удаётся. Но можно отметить, что загрузка несколькими потоками одинаковых данных предпочтительнее использования условной конструкции.

Идея использовать разделяемую память для окон блоков кандидатов не была отвергнута. Можно заметить, что при вычислении каждой строки SAD требуется всего одна строка окна исходного блока и всего 8 строк области окон блоков кандидатов. При этом при переходе к следующей строке в области окон блоков кандидатов 7 строк сохраняются, а изменяется всего одна. Это наблюдение позволяет, во-первых, уменьшить объём используемой памяти, а во-вторых, существенно уменьшить число подряд идущих загрузок, что позволит эффективнее скрывать задержки обращения к памяти. Таким образом, были проделаны следующие изменения: предварительно в массив загружалось всего 7 строк области окон блоков кандидатов и далее, перед

вычислением очередной строки SAD, подгружалась одна строка окна исходного блока и одна строка области окон блоков кандидатов. Размеры же соответствующих массивов были уменьшены до 32 и 320 элементов. Результаты данных изменений приведены в Таблице 8.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, разделяемая память для окна исходного блока и для окон блоков кандидатов с постепенной загрузкой, 8x8 потоков в блоке, параллельная редукция | 430 |

Таблица 8: время работы реализации

К сожалению, время работы всё равно не уменьшилось. Но здесь нужно отметить одну особенность разделяемой памяти. Она организована в виде 32 банков по 4 байта каждый. При этом независимо обрабатываются только обращения к разным банкам, а несколько обращений к одному банку выполняются последовательно [9]. Соседним потокам требуются соседние элементы области окон блоков кандидатов, а они хранятся как байты, что вызывает конфликт обращения к разделяемой памяти. После введения последовательного чтения массивов в разделяемой памяти их размеры значительно уменьшились, что позволяет перейти к использованию 4х байтового типа `int` для хранения значений. Это позволит устранить конфликт обращения к разделяемой памяти. Результаты этих изменений приведены в Таблице 9.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, разделяемая память типа int для окна исходного блока и для окон блоков кандидатов с постепенной загрузкой, 8x8 потоков в блоке, параллельная редукция | 268 |

Таблица 9: время работы реализации

Полученные результаты совпадают с результатами, полученными при использовании текстурного кэша для окон блоков кандидатов и разделяемой памяти для окна исходного блока, приведенными в Таблице 6. Из этого можно сделать вывод, что задержки обращения к памяти уже достаточно хорошо скрыты и стоит попытаться ускорить программу за счёт вычислительных оптимизаций.

## 4.6 Использование аппаратных функций и избавление от циклов

Как было отмечено выше, обращения к памяти уже достаточно хорошо оптимизированы, и следует переходить к оптимизации вычислений. Основная арифметическая нагрузка происходит при вычислении значения SAD, поэтому оптимизации нужно проводить именно там.

Во-первых, для вычисления суммы абсолютной разницы на платформе CUDA реализована специальная функция `__sad()`, которая принимает 3 параметра и возвращает сумму третьего параметра и абсолютной разницы первых двух. Во-вторых, тело внутреннего цикла вычисления SAD содержит только загрузку двух значений и вычисление их абсолютной разницы. При этом на каждой итерации происходит проверка условия, которая может давать заметное замедление, если значения для вычислений загружаются из разделяемой памяти или текстурного кэша, так как задержки загрузки могут оказаться недостаточными для сокрытия проверки условия. Поэтому было решено раскрыть цикл, заменив его необходимым количеством одинаковых строк.

Так как результаты использования текстурной памяти для области окон блоков кандидатов совпали с результатами при использовании разделяемой

памяти, вычислительные оптимизации были проведены для обоих вариантов. Их результаты приведены в Таблице 10.

| Реализация | Время работы, мс |
|---|---|
| Текстурная память, разделяемая память типа int для окна исходного блока и для окон блоков кандидатов с постепенной загрузкой, 8x8 потоков в блоке, параллельная редукция, раскрутка цикла, функция __sad() | 111 |
| Текстурная память, разделяемая память для окна исходного блока, 8x8 потоков в блоке, параллельная редукция, раскрутка цикла, функция __sad() | 202 |

Таблица 10: время работы реализаций

После уменьшения вычислительной нагрузки стало понятно, что разделяемая память всё-таки лучше скрывает задержки обращения к памяти. Также была проведена попытка применить данные вычислительные оптимизации к реализациями, использующим только текстурную память или только глобальную. Как ожидалось, никакого заметного эффекта они не оказали, так как в соответствующих реализациях время работы определяется задержками обращения к памяти.

## 4.7 Сводные результаты. Сравнение с реализацией на центральном процессоре

Для оценки полученных результатов тестирования была сделана реализация данного алгоритма на центральном процессоре. Было проведено такое же раскрытие цикла, как и в предыдущем разделе. Также были экстраполированы результаты работы [15]. В данной работе рассматривается

оптимизация алгоритма сопоставления блоков с помощью применения инструкций SSE2. Результаты этой работы показывают ускорение до 4 раз.

| Платформа | Реализация | Время работы, мсек | Ускорение |
|---|---|---|---|
| Core2 Duo E8200, 2667 МГц | без оптимизаций | 36000 | 1.00 |
| | раскрутка циклов | 31000 | 1.16 |
| | раскрутка циклов, SSE2* | 7750 | 4.65 |
| GeForce GTX 275, 240 ядер, 1404 МГц | без оптимизаций | 1350 | 26.67 |
| | 8x8 блок потоков | 1000 | 36.00 |
| | вычислительные оптимизации, разделяемая память | 111 | 324.32 |

Таблица 11: сравнение реализаций

Размеры исходных изображений 640 на 480 пикселей. Максимальный сдвиг в каждую сторону не более трёх. Размер окна вокруг блока 32 на 16 пикселей.

# 5. Заключение

По итогам данной работы были достигнуты следующие результаты:

- Алгоритм сопоставления блоков был проанализирован для переноса на массово-параллельную платформу.
- Данный алгоритм был реализован на платформе CUDA. Итоговое приложение было создано с возможностью выбора параметров поиска.
- Для конкретных параметров были приведены шаги оптимизации, позволяющие достичь значительного ускорения.

Таким образом был создан инструмент для эффективного поиска соответствия в изображениях, который был встроен в продукты компании Avarex.

Задачи, поставленные в рамках работы, были выполнены.

# Список использованной литературы

[1] Lucas B D and Kanade T, «An iterative image registration technique with an application to stereo vision», Proceedings of Imaging understanding workshop, pp 121—130, 1981

[2] B.K.P. Horn and B.G. Schunck, «Determining optical flow», Artificial Intelligence, vol 17, pp 185 — 203, 1981

[3] Buxton, B.F., Buxton, H., «Monocular Depth Perception from Optical Flow by Space Time Signal Processing», RoyalP (B-218), pp. 27 — 47, 1983

[4] Deepak Turaga, Mohamed Alkanhal, «Search Algorithms for Block-Matching in Motion Estimation», http://www.ece.cmu.edu/~ee899/project/deepak_mid.htm, 1998

[5] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym. «NVIDIA Tesla: A unified graphics and computing architecture.» IEEE Micro, 28(2): 39–55, 2008.

[6] NVIDIA. CUDA zone. http://www.nvidia.com/cuda.

[7] David Luebke, Mark Harris, Jens Kr¨uger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. «GPGPU: general purpose computation on graphics hardware.» In SIGGRAPH ’04: ACM SIGGRAPH 2004 Course Notes, page 33, New York, NY, USA, 2004. ACM.

[8] Aaftab Munshi. OpenCL: Parallel computing on the GPU and CPU. 2008.

[9] NVIDIA. NVIDIA CUDA Programming Guide, 2.3 edition, 2009.

[10] Sunpyo Hong and Hyesoon Kim. «An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness», SIGARCH Comput. Archit. News, 37(3): 152–163, 2009.

[11] G. Diamos, A. Kerr, and M. Kesavan. «Translation GPU binaries to tiered SIMD architectures with Ocelot» Technical report, 2009.

[12] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U.C. Weiser. «Many-core vs. many-thread machines: Stay away from the valley» IEEE Computer Architecture Letter, 2008.

[13] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. «Larrabee: a many-core x86 architecture for visual computing.» In SIGGRAPH '08: ACM SIGGRAPH 2008 papers, pp. 1–15, New York, NY, USA, 2008. ACM.

[14] Mark Harris. «Mapping computational concepts to GPUs.» In SIGGRAPH'05: ACM SIGGRAPH 2005 Courses, page 50, New York, NY, USA, 2005. ACM.

[15] Trung Hieu Tran, Hyo-Moon Cho, and Sang-Bock Cho, «Performance Enhancement of Motion Estimation Using SSE2 Technology», World Academy of Science, Engineering and Technology (40), pp. 168 – 171, 2008