САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-Механический Факультет

Кафедра Системного Программирования

“Оптимизация алгоритма SGM для архитектур на базе GPGPU”

Дипломная работа студента 545 группы

Шевченко Андрея Игоревича

Научный руководитель	_ _ _ _ _ _ _ _ /подпись/	Пименов А.А.
Рецензент	_ _ _ _ _ _ _ _ /подпись/	ст. преп. Кривошеин Б.Н.
“Допустить к защите” заведующий кафедрой,	_ _ _ _ _ _ _ _ /подпись/	д. ф.-м. н., проф. Терехов А.Н.

Санкт-Петербург

2010

Оглавление

1. Введение

Задача восстановления глубины по нескольким изображениям, сделанным одновременно, является крайне важной и необходимой для многих приложений, например для тех, задачей которых является восстановление 3D-модели объекта по нескольким его фотографиям, снятым с разных ракурсов. Эта фундаментальная проблема была и остается предметом многолетних исследований не только в компьютерной графике, но и в когнитологии и психофизиологии. Для решения этой задачи существует целый ряд алгоритмов, которые можно разделить на 2 категории: локальные и глобальные. Локальные подходы, как правило, отличаются меньшей вычислительной сложностью, однако результат их работы может содержать очень грубые ошибки, что является совершенно неприемлемым для некоторых приложений. Глобальные подходы дают очень хорошие результаты, но использовать их в реальном времени, на сегодняшний день, практически невозможно. Алгоритм Semi-Global Matching дает практически такие же конечные результаты как глобальные алгоритмы, но при этом работает гораздо быстрее. Тем не менее, скорость его работы, при реализации на CPU, по прежнему недостаточна для приложений реального времени, а также для работы с большим количеством изображений или с изображениями крайне высокого разрешения.

GPGPU (General-purpose computing on graphics processing units) – техника использования графического процессора видеокарты для общих вычислений, которые обычно производит центральный процессор. Современные видеокарты, начиная от GeForce 8800GTX, обладают крайне производительным многоядерным графическим процессором с очень мощными вычислительными способностями и высокой пропускной способностью памяти. Для сравнения, видеокарта GeForce 8800 GTX обладает производительностью в 500 гигафлоп и пропускной способностью около 90 гб\сек, в то время как процессор Intel Core i7-975 XE 3,33 ГГц имеет лишь 70 гигафлоп и около 15 гб\сек пропускной способности. Более того, та же видеокарта предоставляет 128 потоковых процессоров на уровне железа, что вкупе с ее вычислительной мощностью позволяет достигнуть очень большой эффективности.

Одной из самых лучших реализаций GPGPU является технология CUDA от компании NVIDIA. CUDA даёт разработчику возможность по своему усмотрению организовывать доступ к набору инструкций

графического ускорителя и управлять его памятью, организовывать на нём сложные параллельные вычисления. В основе CUDA лежит язык C, но также существуют реализации для Java и .Net. Код, написанный с использованием CUDA, можно встраивать в уже существующие проекты без особого труда, что позволяет значительно ускорять работу сложных алгоритмов, используемых в приложении, переписывая их отдельные части (или даже целиком) на CUDA, не затрачивая при этом усилий на последующую интеграцию.

Реализация алгоритма SGM с использованием технологии CUDA позволит значительно увеличить скорость его работы, по сравнению с реализациями на CPU, приблизив ее к реальному времени. Это значит, что реализация должная задействовать всю мощь современных GPU устройств, и, в частности, их возможности в области параллельных вычислений.

Целями данной работы являются: анализ алгоритма SGM для переноса его на массово-параллельные архитектуры, реализация модифицированного алгоритма на наиболее известном представителе GPGPU – CUDA, исследование и внесение предложений по оптимизации базового алгоритма, количественная и качественная оценка оптимизаций. Кроме того, перенос данного алгоритма на GPGPU поможет при его реализации на любой другой архитектуре SIMD и даже на FPGA архитектуре.

2. Подготовка и предобработка данных. Формальное задание проблемы

Рассмотрим основные определения и понятия, возникающие при решении задачи восстановления глубины. Мы рассмотрим, вкратце, весь процесс: от, собственно, непосредственной видео-фото съемки, до получения итогового результата.

Для простоты мы будем считать, что у нас есть две камеры (C_1 и C_2), которые снимают одну и ту же сцену одновременно, но из различных точек обзора. Камеры делают два снимка, и мы получаем, соответственно, две фотографии одной и той же сцены.

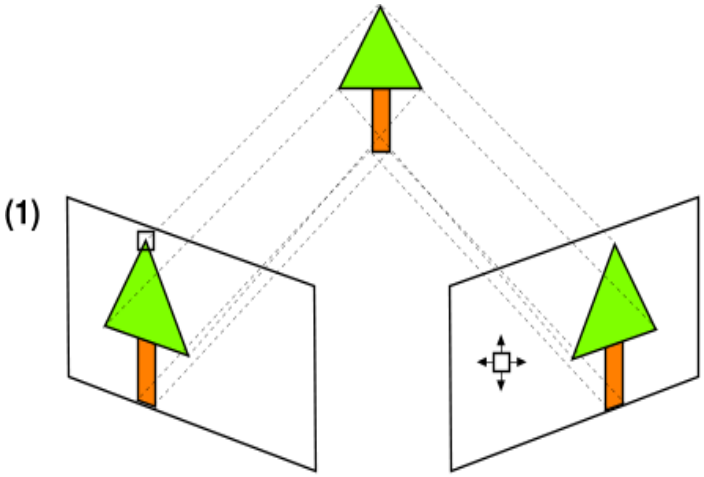

Рисунок 1. Так могут выглядеть фотографии, полученные после видео-фото съемки.

Следующим шагом необходимо найти каждой точке левого изображения соответствующую ей точку на правом изображении. Эта проблема получила название проблема соответствия. На рисунке 1 изображено пространство поиска подходящей точки – это все правое изображение, то есть мы вынуждены производить наш поиск в двух измерениях. Если бы мы могли неким специальным образом выровнять наши изображения так, чтобы сузить пространство поиска до одного измерения, решение проблемы соответствия можно было бы найти с гораздо меньшими производственными затратами. Процесс коррекции нескольких изображений, после которого они имеют общую поверхность изображения (image surface), называется ректификацией. После ректификации, изображения на рисунке 1 станут выглядеть так

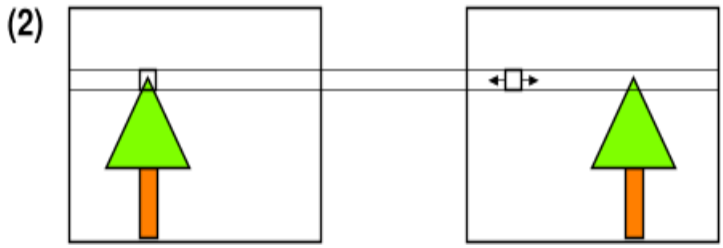

Рисунок 2. Изображения после процедуры ректификации. Пространство поиска сужено до одной строчки.

Ректификация позволяет сузить пространство поиска решений для проблемы соответствия до одного измерения. Более конкретно, после проведения ректификации, все «подозрительные» пиксели, то есть все пиксели правого изображения, которые могут соответствовать данной точке левой картинки, будут находиться в одной строчке. Именно пара ректифицированных изображений является входными параметрами к задаче восстановления глубины.

Следующим глобальным этапом является решение проблемы соответствия, а именно, построение карты диспаратности (disparity map) по двум ректифицированным изображениям. Для того чтобы понять что такое карта диспаратности рассмотрим следующий пример

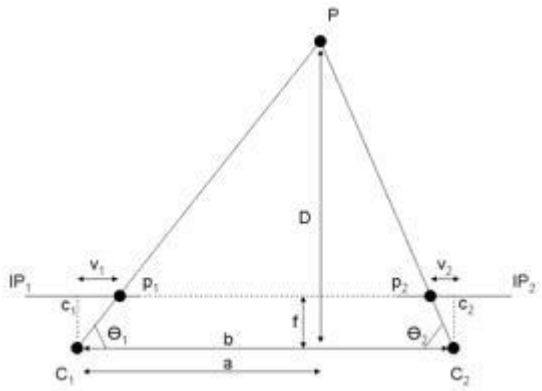

Рисунок 3. Геометрическое представление наблюдаемой сцены.

C_1 и C_2 – это центры камер. IP_1 и IP_2 – это плоскости изображения, то есть по сути фотографии. P – наблюдаемая точка, а P_1 и P_2 – это перспективные проекции на соответствующую плоскость изображения. Расстояние между камерами – b. Наша задача – найти D, то есть расстояние до точки в реальном мире. Для этого мы посчитаем диспаратность для точки P, которая равна разнице между V_1 и V_2, где $V_{1(2)}$ - это смещение точки $P_{1(2)}$ относительно центра первой (второй) камеры, то есть точки $c_{1(2)}$. Диспаратность можно понимать по-другому, как смещение в пикселях точки

P_2 относительно точки P_1. После того, как в каждой точке посчитана диспаратность, нахождение реальной глубины не является сложной задачей. Пусть f – фокусное расстояние камеры, тогда $D = \frac{b \cdot f}{d}$. Причем, поскольку фокусное расстояние и расстояние между камерами остаются постоянными для всех точек на фотографии, то диспаратность можно использовать как относительную глубину точек. Зная диспаратность в каждой точке, мы можем построить карту диспаратности, например

Рисунок 4. Левое и среднее изображения - фотографии Пентагона. Правое - карта диспаратности.

Построение карты диспаратности – сложная вычислительная задача. Кроме того, проблема соответствия была и остается нерешенной полностью проблемой. Это связано со следующими причинами, в соответствии с [13]

- *Шум*. Если мы говорим о съемке в реальных, а не лабораторных условиях, то на получаемых фотографиях практически всегда будут присутствовать такие неизбежные факторы: световые блики, размытие изображения и просто шум, полученный из-за несовершенных сенсоров. Из этого следует, что алгоритмы, решающие проблему соответствия, должны быть устойчивыми.
- *Однотонные области*. Эта проблема также получила название проблема апертуры. Информация, получаемая с крайне текстурированных областей, должна быть распространена на однотонные области
- *Разрывность глубины*. Многие алгоритмы требуют для своей работы того, чтобы на некоторой области глубина была непрерывной. Это создает проблемы, когда в нужную область попадают границы объектов, то есть места, где глубина терпит разрыв.

- *Заслоненные области*. Те пиксели, которые являются заслоненными на одном изображении, не должны быть поставлены в соответствие пикселям из другого изображения.

Несмотря на вышеперечисленные проблемы, существует множество различных подходов со своими плюсами и минусами. Обзору существующих решений посвящена следующая часть. Прежде чем перейти к следующей части, сформулируем проблему соответствия более строго. Будем считать, что у нас на вход дается два изображения: базовое и парное к нему. Определим $I_b = \{I(p) \mid p \in$ базовому изображению$\}$ и $I_m = \{I(p) \mid p \in$ парному изображению$\}$, где функция $I(p)$ – функция сопоставления пикселю изображения его интенсивности. Наша задача: построить карту диспаратности $f = \{f_p \mid p \in$ базовому изображению$\}$, где каждое f_p представляет соответствие между пикселем p базового изображения и пикселем p - f_p парного изображения. Поскольку мы полагаем, что изображения были предварительно ректифицированы, то f_p можно рассматривать как скаляр, а не как двумерный вектор, и прибавление осуществляется к x-координате пикселя.

3. Обзор существующих решений

Принято различать два достаточно широких класса алгоритмов для решения проблемы соответствия: локальные и глобальные алгоритмы. В сумме эти два подхода объединяют большинство из существующих решений. В соответствии с [11]: большинство решений проблемы соответствия выполняют, частично или полностью, следующие шаги

1. Вычисление стоимостей соответствий.
2. Суммирование стоимостей.
3. Вычисление диспаратностей на основе вычисленных стоимостей.
4. Улучшение карты диспаратности.

Реальная последовательность шагов зависит от конкретного алгоритма.

3.1. Локальные алгоритмы.

Этот класс алгоритмов подсчитывает диспаратность каждого пикселя в отдельности, используя окна фиксированного или адаптируемого размера для корреляции. Выбор формы окна и его размеров является непростой задачей. Это связано с тем, что, с одной стороны, корреляция предполагает, что глубина всех пикселей внутри окна не терпит разрывов. Это значит, что увеличение размеров окна ведет к нарушению этого условия и, как следствие, к некорректной работе алгоритма. С другой стороны, уменьшение размеров окна ведет к увеличению влияния шума, что приводит к уменьшению корректных совпадений. Для локальных алгоритмов стоимость соответствия (т.е. первый шаг работы) определяется как схожесть между двумя областями, одна из которых находится в базовом изображении, а другая в парном. Форма этих областей зависит от конкретного алгоритма. Стандартные подходы, появившиеся на заре исследований проблемы соответствия, используют прямоугольное окно фиксированного размера. Размеры этого окна определяются опытным путем. Объем вычислений, в этом случае, сильно сокращается по сравнению с современными методами (о них позже), но существует ряд проблем, избежать которых, используя фиксированный размер окна, практически невозможно. Для того чтобы понять: почему фиксированный размер окна порождает проблемы, мы рассмотрим, как, в общем, работают локальные алгоритмы.

Рассмотрим произвольный пиксель базового изображения, вокруг него строится наше окно. Для данного пикселя базового изображения существует ряд подозрительных на соответствие пикселей парного изображения. Вокруг

каждого такого пикселя строится такое же окно. Подобная ситуация показана на рисунке 5

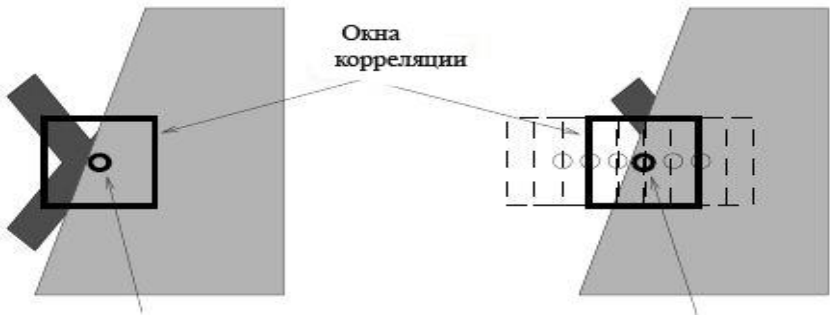

Рисунок 5. Рассматриваемый пиксель. Соответствующий пиксель.

Возможные позиции пикселей в парном изображении определены минимально допустимым расстоянием между камерами и объектом, что дает нам максимальную диспаратность. Для каждой возможной пары окон подсчитывается стоимость соответствия. Наиболее распространенными стоимостями являются квадрат разности интенсивностей (SD) и абсолютная разность интенсивностей (AD). Пара окон с наибольшей или наименьшей, в зависимости от алгоритма вычисления, стоимостью и определяют пиксель соответствующий данному. Тем не менее, если в окне нарушается условие на непрерывность глубины, то его часть будет влиять на результирующую стоимость неопределенным образом, в соответствии с [6]. Вернемся к рисунку 5. Левая часть корреляционного окна, как в базовом изображении, так и в парном, содержит фон. Но этот фон разный для обоих окон. Более того, стоит отметить, что левая часть окна в левом изображении, по-крайней мере частично, заслонена на правом битмапе. Размеры заслоненной части зависят от разности диспаратностей и размеров окна. Все это приводит нас к тому, что та часть окна, которая содержит фон, вносит ошибки в подсчет стоимостей. Возможным решением проблемы будет уменьшение размеров окна, но, как я уже говорил, это приведет к возрастанию влияния шума при подсчете стоимости.

Как было показано выше, фиксированный размер окна не дает необходимой точности при решении проблемы соответствия. Kanade в [7] предложил решение этой проблемы путем изменения размеров и формы окна в зависимости от локальных характеристик диспаратности. Этот алгоритм позволил уменьшить число ошибок, возникающих на границах объектов. Тем не менее, данный алгоритм является слишком медленным для выполнения в реальном времени на неспециализированном оборудовании, в соответствии с [11].

Адаптацией подхода, предложенного Kanade в [7], является многооконный алгоритм, разработанный Fusiello в [3]. Стандартная конфигурация состоит из 9 окон одного размера, но расположенных в разных местах относительно рассматриваемого пикселя. Корреляция производится для каждого из 9 окон в отдельности, но только результат, показанный «лучшим» окном, используется. Данный алгоритм показывает лучшие результаты в сравнении со стандартной корреляцией и его эффективность достаточна для использования в реальном времени. Тем не менее, по сравнению с глобальными подходами, данное решение дает неточные результаты.

Boykov в [1] разработал алгоритм, работающий следующим образом: для каждого пикселя метод выбирает окно произвольной формы. Эта форма меняется от пикселя к пикселю. Несмотря на возможность использовать метод в реальном времени, он дает довольно большой процент ошибок, что было отмечено самими авторами алгоритма.

Объединенный стерео-алгоритм, предложенный Kanade и Zitnick в [11] демонстрирует, что количество ошибок может быть уменьшено значительно, если время исполнения не является важным фактором. Данный алгоритм является слишком медленным для реализации его в реальном времени.

Hirschmuller в [6] создал метод коррекции ошибок на границе объектов. Данная коррекция применяется на этапе постпроцесса и позволяет уменьшить количество ошибок на границах на 50%. Идея алгоритма состоит в использовании нескольких (а именно 5) окон для уменьшения количества ошибок, а также в предоставлении функции, которая определяет неверные совпадения.

Наряду с исследованиями, связанными с формой и размерами окна, существуют различными подходы к измерению стоимости соответствия. Уже были упомянуты два стандартных подхода SD (разность интенсивностей в квадрате)

$$SDC(\mathrm{p},\mathrm{d}) = \sum_{\mathrm{p}\in\mathrm{W}} \left|\mathrm{I_{bp}} - \mathrm{I_{mp}}\right|^2, m = p - d.$$

И AD (абсолютная разность интенсивностей)

$$ADC(p,d) = \sum_{p\in W} \left|I_{bp} - I_{mq}\right|, q = p - d.$$

В обоих случаях, подсчет стоимости можно сделать очень эффективным. Однако, в случае выбора этих формул для вычисления, неявно делается предположение, что для данного пикселя базового изображения, соответствующий пиксель парного имеет примерно такую же интенсивность. Это значит, что если у двух камер была выставлена разная экспозиция или на фотографии, сделанной одной из камер, появился блик, то эти формулы будут давать неверный результат. Также, оба этих подхода довольно сильно подвержены влиянию шума и случайных выбросов в данных.

Zabih в [14] представил измерение основанное не на значениях интенсивностей, а на их числовом порядке.

$$\xi(I,P,P') = \begin{cases} 1 & \textit{если } I(P) < I(P') \\ 0 & \textit{в} \text{ другом случае} \end{cases},$$

$$Err(P,P') = \xi(I_b,P,P') - \xi(I_m,P,P'),$$

$$C(W) = \sum_{P \in W_1} \sum_{P' \in W_2} \left| Err(P,P') \right|,$$

$$R(W) = \sum_{P \in W_1} \left(\sum_{P \in W_2} Err(P,P') \right)^2.$$

Здесь C(W) – это трансформация данных корреляции, а R(W) - упорядочивание данных корреляции. На их основе вычисляется итоговая стоимость соответствия. Такой подход позволяет уменьшить воздействие шума и случайных выбросов данных по сравнению со стандартным подходом. Тем не менее, различия в настройке камер по-прежнему могут сильно влиять на итоговый результат.

Градиентное измерение, представленное Scharstein в [10] является нечувствительным к различиям между настройками в цветопередаче камер.

После того, как для каждого пикселя p в базовом изображении и для каждой возможной диспаратности d, от 0 до максимальной диспаратности, подсчитана стоимость соответствия, локальный алгоритм, обычно, переходит к шагу суммирования стоимостей. Стандартный подход состоит в суммировании или усреднении стоимостей соответствия в некоторой области. Эта область может быть или двумерной, в этом случае мы считаем, что диспаратность фиксирована, или трехмерной, когда диспаратность изменяется наравне с координатами пикселей. Усреднение может

производиться с помощью свертки с каким-нибудь ядром, чаще всего Гауссовым.

Наконец, финальным шагом является вычисление диспаратности. На основе просуммированных или усредненных стоимостей нахождение соответствия становится тривиальной задачей. Диспаратность для данного пикселя p базового изображения – это такое d, на котором достигается минимальное значение стоимости. Такой подход получил название «победитель получает все» (WTA). Проблема, возникающая при таком подходе, каждой точке выбирается уникальное соответствие в парном изображении. На самом же деле, одной точке может соответствовать сразу несколько. К сожалению, эта проблема возникает не только у локальных алгоритмов, но и у подавляющего большинства методов, целью которых является построение карты диспаратности.

Просуммировав все вышесказанное про локальные алгоритмы, можно сказать, что существует огромное число локальных методов, решающих проблему соответствия. Тем не менее, если методу удается решить задачу быстро, то это означает большое количество ошибок в вычислениях, что, конечно, не всегда приемлемо. И наоборот, локальные методы решающие проблему с малым числом ошибок, такие как метод Zitnick в [16], не представляется возможным реализовать в реальном времени на современном оборудовании.

3.2. Глобальные алгоритмы.

В отличие от локальных алгоритмов, где нахождение диспаратности происходит для каждого пикселя отдельно, целью глобального подхода является поиск наилучшей карты диспаратности для всего изображения сразу. Глобальные методы практически всю работу выполняют на шаге вычисления диспаратностей, часто пропуская шаг суммирования стоимостей соответствия. Чаще всего глобальные алгоритмы решают проблему соответствия путем минимизации функционала энергии.

Допустим, мы хотим найти наилучшую конфигурацию f. Подходящая конфигурация будет минимизировать функционал E – функционал глобальной энергии

$$E(f) = E_{data}(f) + \lambda \cdot E_{smooth}(f)$$

Первое слагаемое показывает: насколько хорошо данная конфигурация согласуется с парой входных изображений. Для того чтобы подсчитать первое слагаемое, используются попиксельные стоимости

$$E_{data}(f) = \sum C(x, y, f(x, y))$$

Второе слагаемое отвечает за штраф, налагаемый на данную конфигурацию, в случае, когда она нарушает непрерывность диспаратностей. Чаще всего рассматриваются соседние пиксели.

$$E_{smooth}(f) = \sum_{(x,y)} \rho(f(x, y) - f(x+1, y)) + \rho(f(x, y) - f(x, y+1))$$

Здесь, в качестве функции ρ выступает некоторая монотонно-возрастающая функция штрафа. В [4] был представлен другой подход в вычислении E_{smooth}

$$E_{smooth}(f) = \sum_{(x,y)} \rho_f(f(x, y) - f(x+1, y)) \cdot \rho_I(\|I(x, y) - I(x+1, y)\|) +$$
$$\sum_{(x,y)} \rho_f(f(x, y) - f(x, y+1)) \cdot \rho_I(\|I(x, y) - I(x, y+1)\|)$$

В данном случае, ρ_I - это некоторая монотонно-убывающая функция, которая снижает стоимость в случае областей, где интенсивность меняется значительно от пикселя к пикселю.

После того, как функционал энергии определен, существует целый ряд алгоритмов, позволяющих найти его минимум. Классическими подходами являются: Марковские сети, максимальный поток, graph-cuts. К сожалению, поиск минимума это очень сложная вычислительная задача. Одним из методов, призванных решить эту проблему, является динамическое программирование. Вместо поиска минимума для всего изображения сразу, поиск ведется построчно, причем строки обрабатываются независимо друг от друга. Такой подход применяется, например, в [2]. Нахождение минимума для одной строки возможно, в этом случае, за полиномиальное время. Проблемы данного подхода очевидны, при вычислении карты диспаратностей мы совершенно не учитываем связей между пикселями в разных строках, кроме того, этот подход требует, чтобы относительный

порядок пикселей в строке был одинаковым для левого и правого изображения, что невозможно выполнить в том случае, когда на сцене находится узкий объект на переднем плане.

Глобальные алгоритмы дают отличные результаты в нахождении карты диспаратностей. Количество ошибок, допускаемых ими, относительно невелико. Тем не менее, в настоящее время, ни один из предложенных алгоритмов не может быть реализован в реальном времени на современном оборудовании, в соответствии с [11].

Для нашей задачи глобальные алгоритмы в чистом виде не подходят. Предложенный алгоритм SGM представляет собой гибридный метод, при котором сочетается качество глобальных алгоритмов со скоростью локальных подходов.

4. Обзор CUDA

Для того чтобы лучше понять, какие проблемы возникают при разработке и оптимизации приложений на CUDA, необходимо знание того, из чего, собственно, CUDA состоит, какие возможности она предоставляет и где, вероятней всего, будут узкие места. В рамках этого диплома, я дам лишь краткое описание этой технологии, для более подробной информации можно обратиться к [9].

Из-за высокого рыночного спроса на высококлассную графику реального времени, видеокарты развились до многопоточных, многоядерных процессоров с потрясающей вычислительной мощностью и очень высокой пропускной способностью памяти. Это, в том числе, означает, что графический ускоритель дает широкие возможности для параллелизации. Рассмотрим рисунок 6, где сравниваются вычислительные возможности центрального процессора и процессора видеокарты, а также рисунок 7, где сравнивается пропускная способность памяти.

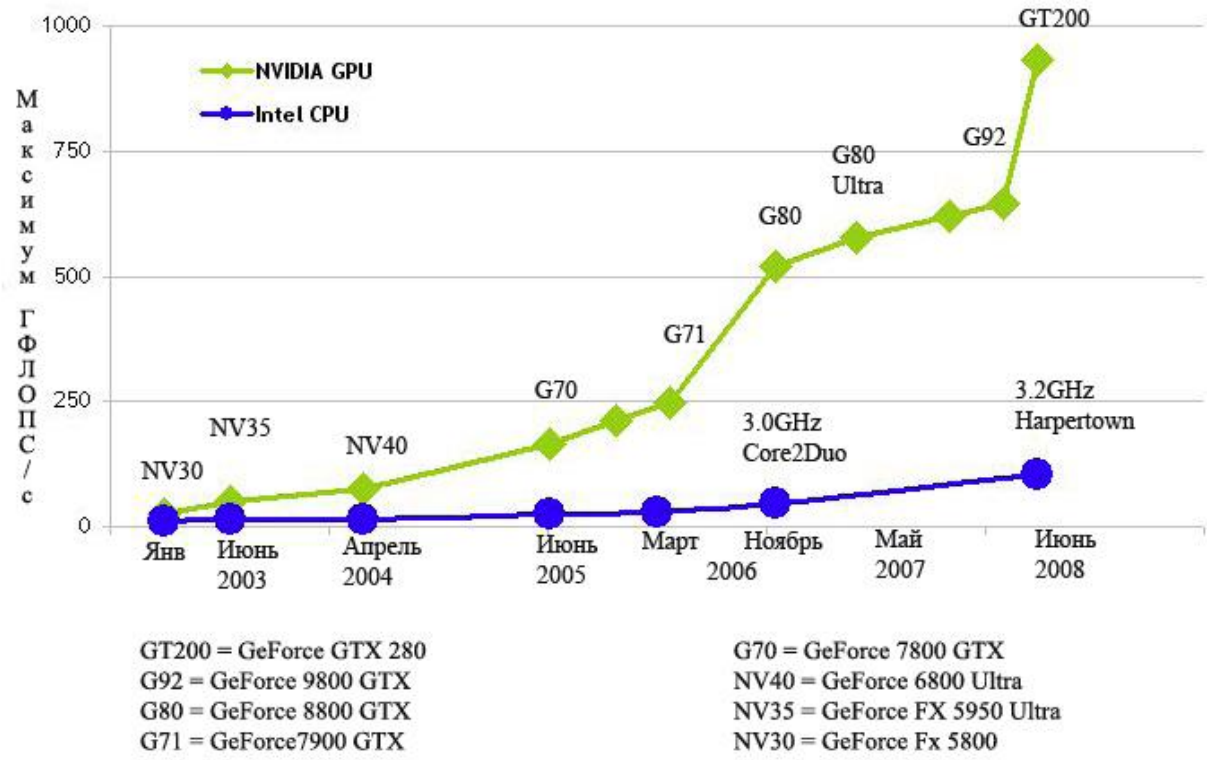

Рисунок 6. Количество операций с плавающей точкой для CPU и GPU.

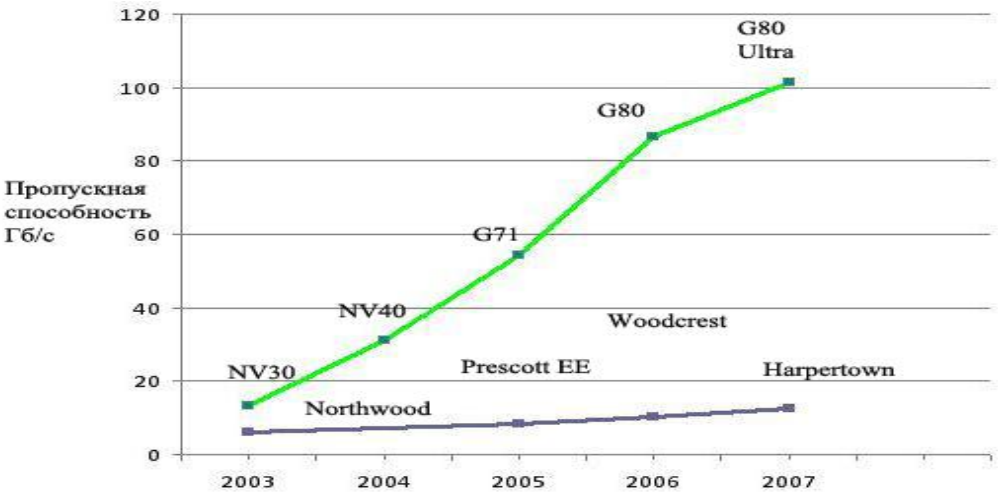

Рисунок 7. Пропускная способность памяти для GPU и CPU.

Причины такого различия в количестве операций с плавающей точкой между CPU и GPU кроются в том, что графические ускорители создавались для выполнения специфичных задач, а именно рендеринга. Специфика здесь в том, что одна и та же программа выполняется параллельно для многих данных, причем количество арифметических операций, по сравнению с операциями с памятью, велико. Отсюда и вытекают две ключевые особенности GPU: отсутствие необходимости в сложном управлении исполнения программы (то есть предсказание переходов и т.д.) и отсутствие большого кеша данных, так как задержка в доступе к памяти может быть скрыта вычислениями.

В ноябре 2006 года, компания NVIDIA представила новую технологию для GPGPU, получившую название CUDA. Уже на тот момент, процессор GPU представлял собой высокопараллельное устройство, причем степень параллелизма продолжала возрастать по закону Мура. Это значит, что необходимо было обеспечить такой уровень абстракции, чтобы переход от одной видеокарты к другой, с большей степенью параллелизма, во-первых, не влиял на исходный код задачи, а во-вторых, исполнение данной задачи на новой видеокарте задействовало бы все возможные ресурсы. Для решения этой проблемы, CUDA предоставляет три ключевых абстракции: иерархия групп потоков, разделенная память и барьерная синхронизация. Расскажем о них подробнее.

Функция, полностью выполняемая на видеокарте, называется ядро. Когда мы вызываем ядро из нашей программы, то оно исполняется N раз на N параллельных CUDA потоках, в противоположность стандартному вызову функции. Величина N зависит от конфигурации ядра. Для того чтобы лучше

понять, что такое конфигурация ядра, мы рассмотрим следующий пример: допустим, что мы хотим сложить две матрицы размерами N×N. Эту задачу выполняет следующий код:

```
// это наше ядро
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
      int i = threadIdx.x + blockId.x*blockDim.x;
      int j = threadIdx.y + blockId.y*blockDim.y;
      C[i][j] = A[i][j] + B[i][j];
};

int main()
{
      // определение размеров сетки и вызов ядра
      dim3 dimBlock(16, 16);
      dim3 dimGrid(N/16, N/16);
      MatAdd<<<dimGrid, dimBlock>>>(A, B, C);
}
```

Каждый поток, выполняющий данное ядро, обладает уникальным идентификатором, доступ к которому можно получить с помощью встроенной переменной threadIdx. Эта переменная является трехкомпонентным вектором, т.е. она определена как переменная типа dim3, это сделано для того, чтобы потоки могли быть идентифицированы по одномерному, двумерному или трехмерному индексу. Несколько потоков объединяются в один блок, размеры которого также задаются при вызове ядра. В данном примере размеры блока задаются переменной dimBlock. Число 16 было выбрано, в некотором смысле, произвольно, единственное ограничение на размер – это количество потоков в блоке, которое не может превышать некоторого числа, заданного конкретной версией CUDA. Также необходимо отметить, что два различных блока потоков должны быть способны выполнятся параллельно и без всякой синхронизации. Наконец, из блоков потоков формируется сетка, размеры которой зависят как от размеров отдельного блока, так и от задачи, которая стоит перед программистом. В данном случае, размеры сетки заданы переменной dimGrid. Сетка всегда двумерная, в то время как блок потоков может быть трехмерный. Такое разбиение задачи дает нам целый ряд преимуществ. Во-первых, разные блоки могут выполняться одновременно на разных мультипроцессорах GPU, а это означает, что их исполнение может быть осуществлено на любом количестве мультипроцессоров в любом порядке, параллельно или последовательно, что позволяет программисту писать код, который автоматически масштабируется

с увеличением количества мультипроцессров GPU. Один блок потоков выполняется на некотором конкретном мультипроцессоре. Мультипроцессор видеокарты, на данный момент, состоит из 8 скалярных процессоров, многопоточного модуля для инструкций, 2 блоков для подсчета операций вида квадратный корень, синус и т.д., блока разделенной памяти, находящейся непосредственно на чипе. Каждый блок, в свою очередь, разбивается на несколько варпов, каждый из которых состоит из 32 потоков. Варп – это единица, с которой работает один мультипроцессор одновременно, реализуя архитектуру, получившую название SIMT (single-instruction multiple-threads, одна инструкция – много потоков). Каждый раз, когда мультипроцессор получает на исполнение один блок потоков, этот блок бьется на варпы, из которых формируется очередь на исполнение. Как только модуль инструкций выбирает следующую команду, из очереди варпов берется следующий на исполнение блок из 32 потоков и инструкция выполняется одновременно сначала для первой половины варпа, а потом для второй половины. Также, в случае если по каким-то причинам исполнение текущего блока приостановлено, мультипроцессор может, если хватает ресурсов, взять еще один или несколько блоков потоков, и выполнять их в моменты, когда исходный блок находится в ожидании. Остается сказать, что на современных видеокартах есть не меньше 8 мультипроцессоров (тестирование производилось на видеокартах с 14 мультипроцессорами) и в будущем это количество будет только расти. Независимость же исходного кода, использующего CUDA, от этой внутренней архитектуры означает, что он будет выполняться все быстрее с выходом новых видеокарт без всякой модификации.

Такое разбиение задачи позволяет добиться очень высокой степени параллелизма. В примере с матрицами, мы создаем поток на каждый элемент. Этот подход получил название иерархия групп потоков и является одной из ключевых особенностей CUDA.

Рассмотрим модели памяти, которые предоставляет CUDA. Потоки CUDA во время исполнения могут получить доступ к разным областям памяти. Во-первых, каждый поток имеет свою собственную локальную память и свои регистры. Во-вторых, каждый блок потоков имеет общую память. Наконец, все потоки сетки имеют доступ к глобальной памяти видеокарты. Кроме того существует еще два пространства памяти, доступ к которым осуществляется только на чтение: текстурная память и глобальная константная память. Эти области памяти, в первую очередь, отличаются

пропускной способностью. И, так как, именно пропускная способность часто становится узким местом в программе, ей стоит уделить особое внимание.

Глобальная память является наиболее медленной из всех вышеперечисленных областей. Задержка при чтении из глобальной памяти составляет от 400 до 600 тактов процессора, а это значит, что доступ к глобальной памяти надо сокращать до минимума и стараться скрывать задержку интенсивными арифметическими вычислениями. Также следует отметить, что данные из глобальной памяти читаются блоками по 4, 8 или 16 байт. Если вспомнить, что одна инструкция для чтения выполняется одновременно для половины одного варпа, то становится понятным, что наибольшая эффективность глобальной памяти достигается тогда, когда данные, которые читает данная половина варпа, находятся в последовательной области памяти размером 32, 64 или 128 байт.

Локальная память является в точности такой же медленной, как и глобальная. Данная область памяти не кэшируется, и задержка составляет те же 400-600 тактов процессора. Программист не может сам управлять чтением или записью в локальную память потока, так как компилятор сам решает, какие из автоматических переменных необходимо помещать в локальную память. Чаще всего там располагаются те структуры, размеры которых таковы, что они потребуют слишком много регистров.

Константная область памяти кэшируется. Это означает, что чтение из этой области стоит, как одно чтение из глобальной памяти, в случае промаха кэша, или одно чтение из кэша.

Текстурная память кэшируется. Кэш текстурной памяти оптимизирован под двумерные данные, следовательно, если потоки в пределах одного варпа читают данные из текстурной памяти по соседним адресам, то будет достигнута оптимальная производительность. Чтение из текстурной памяти имеет ряд преимуществ по сравнению с константной и глобальной памятью. Во-первых, задержка доступа здесь скрыта лучше, а значит, что можно ожидать увеличения производительности в случае произвольного доступа к памяти. Во-вторых, кэш оптимизирован под двухмерные данные. В-третьих, 8-битные и 16-битные данные могут быть конвертированы в 32-битные числа с плавающей точкой в диапазоне [0.0, 1.0], что может быть очень полезно в ряде приложений.

Разделенная память является одной из ключевых особенностей CUDA. Эта область памяти доступна в рамках одного блока потоков, и скорость

чтения и записи является такой же, как аналогичные операции с регистрами, до тех пор, пока не возникает конфликтов банков памяти. Такие конфликты возникают из-за того, что в целях повышения пропускной способности, разделенная память разбита на 16 банков, доступ к которым может быть параллельным. Это значит, что если текущая половина варпа читает данные из разделенной памяти, и при этом каждый варп читает из своего банка, то конфликта не возникает, и скорость доступа равна скорости доступа к регистрам. В случае же обращения к одному банку, скорость замедляется линейно, в зависимости от количества конфликтов. Тем не менее, она будет оставаться существенно выше, чем доступ к глобальной памяти. Следовательно, разделенную память следует использовать везде, где только имеется подобная возможность. Для синхронизации потоков в рамках одного блока используется функция __syncthreads(), предоставляющая барьерную синхронизацию. Издержками на выполнение этой функции можно пренебречь. Легкая барьерная синхронизация является третьей ключевой особенностью CUDA.

Доступ к регистрам, в общем случае, составляет 0 дополнительных тактов, если в каждом блоке находится не меньше 192 потоков.

Завершая описание моделей памяти CUDA, следует сказать, что на размер каждой из областей существует ограничение, зависящее от версии CUDA. На рисунке 8 показана иерархия памяти CUDA

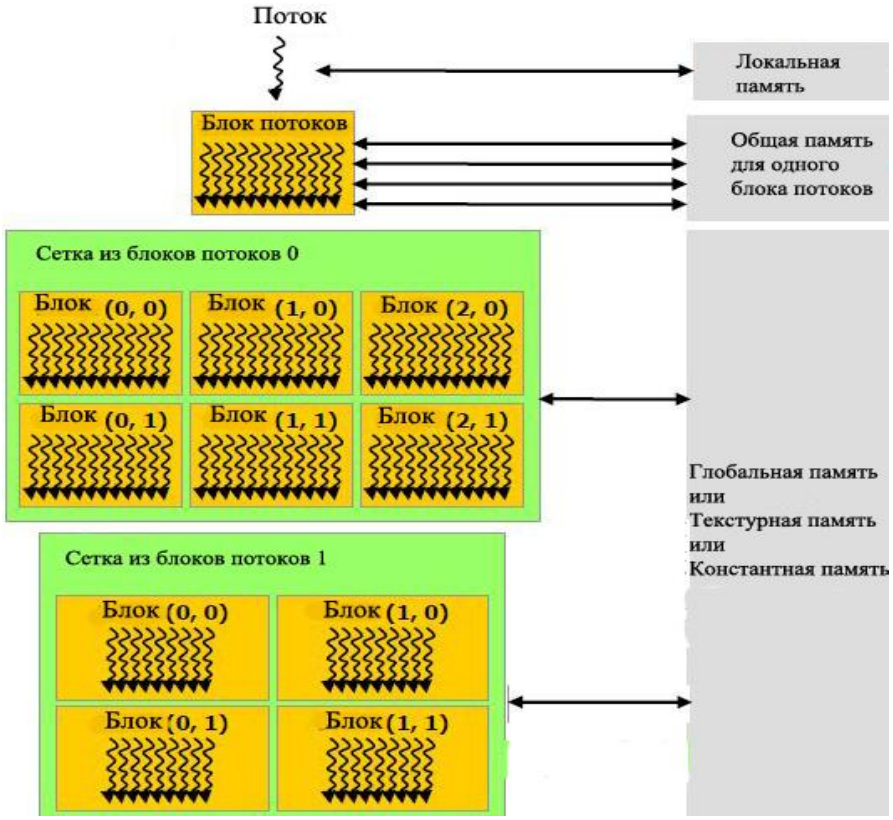

Рисунок 8. Иерархия памяти CUDA.

Выше были кратко описаны ключевые особенности модели CUDA. Несмотря на то, что большую работу на себя берут компилятор и CUDA-драйвер, существует ряд принципиальных трудностей при переносе алгоритмов на CUDA-платформу. В первую очередь, это необходимость разбиения задачи на независимые блоки, что не всегда возможно. Во-вторых, необходимость серьезной оптимизации доступа к памяти и активное использование разделенной памяти предполагает существенную адаптацию алгоритмов. В-третьих, в отличие от стандартного программирования на CPU, мы сильно ограничены по памяти, например, в версии CUDA 1.1 мы не можем выделить больше, чем 512 Мб, что стало существенным ограничением при работе над алгоритмом SGM. Программирование с использованием CUDA, не является «серебряной пулей», но дает, при правильном подходе, широкие возможности, которые, в рамках персональных компьютеров, получить нигде больше не удастся.

5. Основные результаты

5.1. Описание алгоритма

Алгоритм SGM (Semi-Global Matching) был предложен Heiko Hirschmuller в 2005 году в статье [5]. Преимущества этого подхода, по сравнению с другими решениями, состоит в том, что, с одной стороны, он дает очень неплохие результаты с точки зрения качества, так как не является уже локальным алгоритмом. С другой стороны, скорость его работы существенно выше, чем у глобальных подходов. Все это позволяет предположить, что перенос этого алгоритма на платформу CUDA сделает возможным решение задачи соответствия в реальном времени с приемлемым качеством.

В качестве тестовых изображений использовалось 3 набора пар стереокартинок, которые были предварительно ректифицированы. Первый набор состоит из пар размерами 1024×1024, второй - 512×512 и третий - 384×288. В качестве максимальной диспаратности было выбрано 100, что является вполне разумным предположением в рамках реального применения алгоритма.

SGM состоит из 3 шагов, плюс выделяется один опциональный шаг, не входящий формально в состав SGM, но необходимый для его корректной работы:

0. Вычисление интенсивностей пикселей.
1. Вычисление попиксельной стоимости.
2. Суммирование попиксельных стоимостей.
3. Вычисление карт диспаратности.

Шаг 0 выделяется отдельно, так как он не является частью алгоритма. Также этот шаг является опциональным, в зависимости от начальных данных. В том случае, когда на вход приходит изображение в формате RGB, интенсивность можно вычислить в соответствии со стандартом Rec. 709

$$Y_{709} = 0.2125 \cdot R + 0.7154 \cdot G + 0.0721 \cdot B.$$

В дальнейшем я буду считать, что шаг 0 выполнен, если его выполнение необходимо. То есть все дальнейшие вычисления я буду производить с интенсивностями пикселей.

Каждый из последующих шагов будет подробно описан в дальнейшем. Следует обратить внимание на первый шаг, так как при его разработке было принято решение, существенно повлиявшее на дальнейшие шаги и на реализацию алгоритма в целом.

Вычисление попиксельной стоимости означает присваивание некоего числа с плавающей точкой каждой паре пикселей (p, p - d), где p – это произвольный пиксель базового изображения, а d – это число в диапазоне [0, max_disparity]. Это значит, что размер этого массива составит, в случае, например, первого тестового набора (1024×1024), 100 Мб × (размер одной стоимости). К сожалению, в CUDA реализована нативная поддержка чисел типа float и double (с версии CUDA 1.3), но не реализована поддержка типа half-float. Это означает, что размер массива, хранящего попиксельную стоимость, возрастает до 400 Мб. Это является абсолютно неприемлемым по разным причинам, но главным образом потому, что ограничение на размеры используемой приложением глобальной памяти составляет 512 Мб. В том случае, когда один только массив попиксельных стоимостей занимает 400 Мб, это ограничение соблюсти не удается. Первоначально, в качестве решения этой проблемы, предлагалось реализовать свою версию half-float типа. Это позволило бы уменьшить накладные расходы на память в два раза, так как размеры half-float, по стандарту IEEE 754, составляют 2 байта.

5.2. Решение проблем с памятью с использованием типа half-float

Тип half-float состоит из одного знакового бита, экспоненты размером 5 бит и смещением равным 15, а также 10-битной мантиссы. Интерпретация такого представления следующая:

- Если экспонента находится в диапазоне [1..30]:

$$value = (-1)^{s} \cdot 2^{eeeee-15} \cdot 1.mmmmmmmmmm.$$

  В этом случае мы получаем нормализованный half-float.
- Если экспонента равна нулю, а мантисса нулю не равна:

$$value = (-1)^{s} \cdot 2^{14} \cdot 0.mmmmmmmmmm.$$

  В этом случае мы получаем денормализованный half-float.

- Если экспонента равна нулю и мантисса равна нулю:

$$value = \pm 0.0.$$

- Если экспонента равна 31 и мантисса равна нулю:

$$value = \pm \infty.$$

- Наконец, если экспонента равна 31, а мантисса нет:

$$value = \pm NaN$$ (Not a Number).

Наша задача состоит в том, чтобы быстро конвертировать из half-float во float и наоборот. Float тип, в соответствии со стандартом IEEE 754, состоит из знакового бита, 8-битной экспоненты со смещением 128 и 23-битной мантиссы. Существует очень быстрый вариант конвертации в одну и в другую сторону, описанный в статье [15]. Этот алгоритм учитывает все возможные специальные случаи: ноль, денормализованные half-float, бесконечность и NaN. Конвертация из half-float во float основывается на том наблюдении, что, во-первых, за исключением, когда экспонента half-float равна 0, его мантисса просто расширяется нулями. Во-вторых, если экспонента равна 31, то результирующий float будет иметь экспоненту 255. И, наконец, когда экспонента 0, а мантисса не 0, то это денормализованный half-float, который можно преобразовать в обычный нормализованный с помощью сдвига мантиссы и экспоненты. Для конвертации используются особым образом сгенерированные таблицы, размер которых равен 8576 байтам. Таблицы генерируются один раз и хранятся в памяти видеокарты. Окончательная же конвертация выглядит следующим образом:

f=mantissa[offsettable[h >> 10] + (h & 0x3ff)] + exponenttable[h >> 10].

Обратная конвертация также учитывает всевозможные специальные случаи и основана на вычисленных заранее таблицах. Размер этих таблиц составляет 1536 байт. Финальный результат выглядит так

h=basetable[(f >> 23) & 0x1ff]+((f & 0x007fffff) >> shifttable[(f >> 23) & 0x1ff]).

Таким образом, используя дополнительно лишь 10112 байт памяти, мы получили оптимально быструю конвертацию в обе стороны, способную обрабатывать все особые случаи. Тестирование полученного алгоритма дало следующие результаты.

Количество конвертаций	Из Half-Float во Float	Из Float в Half-Float
384×288 = 110592	5.2 ms	3.538 ms
512×512 = 262144	12.5 ms	8.39 ms
1024×1024 = 1048576	50.3 ms	33.5 ms

Таблица 1. Средняя скорость конвертации из Half-Float во Float и обратно.

Как мы видим из этих результатов, накладные расходы на конвертацию примерно одного миллиона элементов, то есть в случае изображения размером 1024×1024, составляют в сумме 83.8 ms. И это без учета третьего

измерения, т.е. диспаратности, вообще. Хотя одна конвертация из half-float во float занимает в среднем 4.2 наносекунды, а обратная в среднем 3.2 наносекунды, количество конвертаций, которое нам придется производить, таково, что суммарное время выходит за рамки требований, предъявляемых приложениям, работающим в реальном времени.

От предложенного решения, из-за полученных результатов, пришлось отказаться. Тем не менее, его потенциал достаточно велик, так как с увеличением производительности, будет увеличиваться и скорость конвертации. Кроме того, возможно в будущих версиях CUDA будет встроена нативная поддержка half-float, что позволит уменьшить объемы потребляемой памяти вдвое, вообще без дополнительных затрат на конвертацию. Тем не менее, полностью решить проблему памяти, используя half-float тип, не удастся еще и из-за того, что в случае изменений требований к входным данным, например увеличение размера до 2048×2048, размер массива попиксельных стоимостей составит 800 Мб, что также выходит за рамки ограничений, возникающих при работе с CUDA.

5.3. Окончательное решение проблем с памятью с использованием тайлов

Для того чтобы окончательно решить проблему нехватки памяти для больших входных изображений, был разработан алгоритм разбиения входного изображения на более маленькие – тайлы. Каждый тайл обрабатывается SGM, как отдельное изображение. Отсюда: затраты на память уменьшаются пропорционально количеству тайлов. Так как финальным результатом должна стать карта диспаратности для изображения целиком, тайлы пересекаются друг с другом по нескольким пикселям. Существует возможность тонкой настройки этого разбиения, включающей в себя количество тайлов в одном ряду и в одном столбце, а также размер области перекрытия двух соседних тайлов как по вертикали, так и по горизонтали. Для того чтобы избежать ненужного, то есть лишнего, копирования данных с памяти видеокарты в оперативную память и обратно, входная пара изображений хранится целиком, тайл же является просто прямоугольным «окном» в этих входных массивах интенсивностей. Накладные расходы на память при таком хранении не велики, и составляют всего 2 Мб в случае изображений размером 1024×1024. В свою очередь время, которое мы экономим на копировании, довольно значительно, так как перенос данных с оперативной памяти в память видеокарты существенно более медленный, чем перенос данных в пределах видеокарты. При таком

подходе нам необходимо различать, где мы должны использовать ширину и высоту исходного изображения, а где тайла. Я буду использовать full_image_with и full_image_height для обозначения, соответственно, ширины и высоты исходного изображения, а tile_width и tile_height – для ширины и высоты одного тайла.

Предложенный подход полностью решает проблему нехватки памяти. С увеличением размера изображений достаточно просто увеличить количество тайлов, тем самым сохранив потребление памяти на прежнем уровне. Такой гибкости не добиться, если просто менять тип данных для стоимости или уменьшать размеры соответствующего массива на некоторую константную величину. Тем не менее, подход не лишен недостатков. Во-первых, разбиение изображения на тайлы несколько увеличивает суммарное время работы, так как некоторые пиксели приходится обрабатывать дважды для разных тайлов. Во-вторых, в случае вычисления попиксельной стоимости на основе не только соответствующей пары пикселей, но всего изображения, результат будет отличаться, в зависимости от разбиения, и, в любом случае, будет учитывать только часть пикселей исходного изображения. В-третьих, наконец, в том случае, когда количество тайлов в строке больше одного, пиксели, являющиеся первыми в каждой строке в тайле, но лежащие в середине исходного изображения, могут получить некорректные результаты, несмотря на существование подходящих пикселей из парного изображения. Это может произойти в том случае, когда эти подходящие пиксели находятся в другом тайле, а, следовательно, не будут обработаны. Данную проблему можно решить, если разбивать изображение так, чтобы tile_width было равно full_image_width. Несмотря на вышеперечисленные недостатки, предложенный подход зарекомендовал себя, показав отличные результаты в тестах.

В рамках работы над дипломом было реализовано два различных подсчета попиксельной стоимости. Первый из них – подсчет стоимости на основе взаимной информации (подход, примененный в классическом SGM), второй – модифицированный алгоритм на основе модуля разности интенсивностей. Для начала подробно рассмотрим алгоритм, основанный на понятии взаимной информации.

5.4. Подсчет попиксельной стоимости на основе взаимной информации. Математический аппарат

Понятие взаимной информации было введено в статье [12]. Математически, взаимная информация основывается на энтропии вероятностных распределений, лежащих в основе нашей пары изображений. Основным достоинством взаимной информации является то, что ее использование позволяет превзойти многие ограничения, которые появляются при решении задачи соответствия. Так, например, если у нас есть базовое изображение и негатив парного изображения, то с помощью ВИ мы также легко получим соответствие, как и при сравнении базового и парного самих по себе. Так же ВИ может справиться и с нефункциональными зависимостями, возникающими между базовым и парным изображением, например, когда один и тот же объект был снят в разных спектрах. Самым же большим недостатком подсчета попиксельных стоимостей на основе взаимной информации является скорость работы, которая может быть значительно ниже, чем скорость работы некоторых других алгоритмов. Попытка обойти этот недостаток была проведена в рамках этого диплома.

Взаимная информация зависит от одиночной энтропии и совместной энтропии двух случайных величин. В нашем случае, случайные величины – пиксели изображений. Давайте предположим, что значение пикселя X, есть некоторая случайная величина с дискретной плотностью распределения P, тогда можно определить энтропию H:

$$H(X) \triangleq -E_x[\log(P(X))].$$

Теперь можно определить, что такое взаимная информация. Пусть X и Y – случайные величины, P(X, Y) – совместное распределение. Тогда:

$$MI(X,Y) \triangleq E_{X,Y}[\log(\frac{P(X,Y)}{P(X)\cdot P(Y)})] \triangleq H(X)+H(Y)-H(X,Y).$$

Теперь, когда мы определились с понятием взаимной информации, нам необходимо понять, как ее можно эффективно считать. Как уже говорилось при обзоре предыдущих работ, в случае глобального подхода мы минимизируем функционал энергии. Этот минимум достигается на самой лучшей карте диспаратности для данной пары изображений. Стандартный функционал выглядит так

$$E(f)=E_{smooth}(f)+E_{data}(f)$$

Первое слагаемое выражает штраф для конфигураций, нарушающих ограничение на непрерывность глубины. Второе слагаемое дает нам штраф для конфигураций, несогласованных с нашими исходными изображениями. Стандартная форма выглядит так

$$E_{data}(f)=\sum_{p} D_p(f_p)$$

Здесь D_p как раз и является попиксельной стоимостью между $I_b(p)$ и $I_m(p - f_p)$. Будем использовать ВИ, как второе слагаемое, в соответствии с [8].

$$E_{data}^{MI}(f)=-MI(I_b, I_m, f)$$

Знак минус стоит потому, что взаимная информация максимизируется, в то время как функционал энергии минимизируется. Здесь, $MI(I_b, I_m, f)$ представляет собой взаимную информацию между нашими исходными изображениями при данной карте диспаратности. Задача состоит в том, чтобы разложить взаимную информацию в сумму попиксельных стоимостей. Для этого воспользуемся методом Парцена, выбрав в качестве ядра Гауссово распределение.

$$MI(I_b, I_m, f)=H(I_b)+H(I_m, f)-H(I_b, I_m)=$$
$$-\int_0^1 P_{I_b}(i)\log P_{I_b}(i)di-\int_0^1 P_{I_m}(i)\log P_{I_m}(i)di+\int_0^1\int_0^1 P_{I_b,I_m,f}(i_1,i_2)\log P_{I_b,I_m,f}(i_1,i_2)di_1di_2.$$

Здесь мы определяем

$$P_{I_b}(i)=\frac{1}{N}\sum_{p} g_\psi(i-I_b(p)),$$
$$P_{I_m,f}(i)=\frac{1}{N}\sum_{p} g_\psi(i-I_m(p-f_p)),$$
$$P_{I_b,I_m,f}(i_1,i_2)=\frac{1}{N}\sum_{p} g_\psi((i_1,i_2)-(I_b(p),I_m(p-f_p))).$$

Здесь $g_\psi(x-\mu)$ это двумерное Гауссово распределение. N – число пикселей в базовом изображении. Далее можно воспользоваться разложением в ряд Тейлора функции $F(x)$

$$F(x) = x \cdot \log x = F(x^0)(x - x^0) + O((x - x^0)^2) = x^0 \log x^0 + (1 + \log x^0)(x - x^0) + O((x - x^0)^2)$$
$$= -x^0 + (1 + \log x^0)x + O((x - x^0)^2)$$

Применяя полученное преобразование к $E_{data}^{MI}(f)$ и преобразовывая, получаем следующий результат:

$$D_p^{MI}(f_p) = -\frac{1}{N}\iint(\log(P_{I_b, I_m, f^0}(i_1, i_2)) \cdot g_\psi((i_1, i_2) - (I_p(p), I_m(p - f_p))))di_1 di_2$$
$$+\frac{1}{N}\int(\log(P_{I_b}(i)) \cdot g_\psi(i - I_b(p)))di + \frac{1}{N}\int(\log(P_{I_m, f}(i)) \cdot g_\psi(i - I_m(p - f_p)))di$$
$$E_{data}^{MI} \cong \sum_p D_p^{MI}(f_p).$$

Заметим, что попиксельная стоимость зависит от f^0 - текущей карты диспаратности. Это значит, что мы не можем подсчитать попиксельную стоимость на основе взаимной информации без знания о карте диспаратности. Эту проблему мы будем решать итеративно, начиная с карты диспаратности, заполненной нулями. В соответствии с Kim в [8]: для хорошего приближения хватает всего лишь трех итераций, на каждой из которых мы будем последовательно уточнять карту диспаратности. Стоит отметить, что эта карта не является финальным результатом и используется только для подсчета попиксельных стоимостей. Теперь, просуммировав все вышесказанное, получаем финальный алгоритм для подсчета попиксельной стоимости на основе взаимной информации.

- Задаем f^0 - первое приближение карты диспаратности. Заполняем ее нулями. Считаем что текущее приближение карты диспаратности $f_D = f^0$
- Вычисляем совместное распределение по формуле

$$P_{I_b, I_m}(i,k) = \frac{1}{N}\sum_p T[(i,k) == (I_b(p), I_m(p - f_D(p)))],$$
$$T[boolean] = \begin{cases} 0, boolean == false \\ 1, boolean == true \end{cases}.$$

- Вычисляем распределения для базового и парного изображения, используя для этого совместное распределение:

$$P_{I_b} = \sum_k P_{I_b,I_m}(i,k),$$
$$P_{I_m} = \sum_i P_{I_b,I_m}(i,k).$$

- В соответствие с полученными формулами вычисляем термы данных для двух одиночных энтропий и для совместной энтропии

$$h_{I_b}(i) = -\frac{1}{N}(\log(P_{I_b}(i) \otimes g(i)) \otimes g(i)),$$
$$h_{I_m}(i) = -\frac{1}{N}(\log(P_{I_m}(i) \otimes g(i)) \otimes g(i)),$$
$$h_{I_b,I_m}(i,k) = -\frac{1}{N}(\log(P_{I_b,I_m}(i,k) \otimes g(i,k)) \otimes g(i,k)).$$

- Вычисляем попиксельную стоимость по формуле

$$C_{MI}(p,d) = h_{I_b,I_m}(I_b(p), I_m(p-d)) - h_{I_b}(I_b(p)) - h_{I_m}(I_m(p)).$$

- Если это итерация 3, то выходим. Иначе для каждого пикселя вычисляем диспаратность на которой достигается наименьшая попиксельная стоимость. Записываем соответствующее значение в f_D и идем на шаг 2.

После того, как мы определились с математической стороной вопроса и построили алгоритм вычисления попиксельных стоимостей на основе ВИ, рассмотрим, как этот алгоритм «ложится» на платформу CUDA, какие трудности возникали, и какие оптимизации были применены. Для удобства я буду описывать каждый шаг данного алгоритма в отдельности.

5.5. Подсчет попиксельной стоимости на основе взаимной информации. Реализация с использованием CUDA

Первый шаг алгоритма довольно примитивен, размер карты диспаратности совпадает с размером одного тайла и для его обнуления используется стандартная функция cudaMemset. Здесь и далее будут описаны результаты, полученные при тестировании на следующих конфигурациях: первый набор стереопар (размер 1024×1024) был разбит на 4 тайла с областью пересечения равной 64, второй набор стереопар (размер 512×512)

на тайлы не разбивался, третий набор (размер 384×288), также как и первый был разбит на тайлы с той же конфигурацией. Среднее время работы первого шага алгоритма составляет:

Размеры карты диспаратности	Nvidia GeForce 8800 GT
208×160	0,0166 мс
512×512	0,0717 мс
528×528	0,0771 мс

Таблица 2. Средняя скорость работы обнуления карты диспаратности для разных конфигураций.

Следующим шагом работы алгоритма является вычисление совместного распределения. При реализации этого шага на GPU возникает ряд неочевидных сложностей. В случае параллельной обработки каждого пикселя базового изображения в отдельности, может возникнуть ситуация, когда два различных потока захотят записать значение в одну и ту же ячейку памяти. Это значит, что при реализации на GPU нужно либо неким образом синхронизировать доступ к каждой ячейке памяти, в которую может произойти запись, либо превратить операцию записи в одну и ту же ячейку в операцию чтения, путем разбиения работы на несколько шагов, то есть на несколько последовательных ядер. В процессе разработки были протестированы оба способа.

Синхронизация доступа к ячейке памяти доступна, начиная с CUDA версии 1.1. В этой версии были представлены атомарные операции с памятью, включая сложение, которое в данном случае нам необходимо. Существуют лишь две видеокарты, которые не поддерживают CUDA версии 1.1, а значит, что с точки зрения системных требований, поддержка данной версии не является серьезным ограничением. Тем не менее, накладные расходы на атомарные операции слишком велики для работы в реальном времени. При тестировании производительности был получен следующий результат:

Размер одного тайла	Nvidia GeForce 8800 GT
208×160	2,5719 мс
512×512	15,9147 мс
528×528	16,9476 мс

Таблица3. Средняя скорость вычисления совместного распределения на основе атомарных операций.

Преобразование операции записи в операцию чтения происходит в три этапа. На первом этапе, для каждого пикселя базового изображения и соответствующей ему диспаратности, из текущего приближения карты

диспаратности, вычисляется смещение в массиве размерами 256×256, который представляет собой данные совместного распределения. На втором шаге эти смещения сортируются. На последнем этапе, для каждой ячейки массива совместного распределения, осуществляется бинарный поиск в отсортированных данных. В процессе поиска мы узнаем, сколько раз мы должны были записать данные в эту ячейку. То есть необходимую нам информацию. Реализация подобного алгоритма упирается в скорость сортировки. Была реализована одна из самых быстрых GPU-сортировок: bitonic merge sort. Сортировка с ее помощью 512×512 = 262144 элементов занимает 32,7 ms. Учитывая, что это всего лишь один тайл для изображения размерами 1024×1024, а также то, что для каждого тайла такая сортировка должна быть произведена 3 раза, мы получаем суммарное время работы одной лишь сортировки для изображения из первого набора стереопар равное 392,4 ms. Для работы в реальном времени это слишком большой результат, ограничивающий скорость работы 2.5 кадрами в секунду. Поэтому от этого способа решено было отказаться.

Поскольку скорость работы второго шага все еще слишком большая, было решено реализовать второй шаг полностью на CPU. За счет того, что код перестал быть параллельным, и, следовательно, отпала необходимость в любых средствах синхронизации, скорость работы второго шага увеличилась значительно. Следует отметить, что для вычисления на CPU нам необходимо совершать трансфер данных с памяти видеокарты в оперативную память и обратно. С учетом копирования данных, среднее время работы дано в таблице 4.

Размер одного тайла	Core2Quad 2.67 GHz
208×160	0,5483 мс
512×512	1,8375 мс
528×528	2,0784 мс

Таблица 4. Скорость вычисления совместного распределения на CPU.

Второй шаг алгоритма вычисления попиксельных стоимостей на основе взаимной информации – единственный шаг, выполняющийся полностью на CPU. Тем не менее, пользователю алгоритма предоставлена возможность выбора между реализацией на центральном процессоре и атомарными операциями на графическом. Причинами такой большой разницы в скорости являются, во-первых, отсутствие сложных алгебраических вычислений, а во-вторых, необходимость синхронизации доступа к памяти при параллельной реализации. Независимо от способа

вычислений, в том случае, когда для данного пикселя p базового изображения, диспаратность такая, что соответствующий пиксель парного лежит за пределами тайла, в качестве соответствующего выбирается пиксель парного изображения с x-координатой равной 0, а y-координатой совпадающей с y-координатой базового. За счет этого достигается более устойчивый результат.

Следующий шаг алгоритма снова довольно простой. Нам необходимо просуммировать все строки и столбцы матрицы совместного распределения. Вычисления выполняются полностью на GPU. Необходимо обратить внимание, что скорость работы не зависит от входных данных. Это связано с тем, что вычисления производятся над массивом фиксированного размера 256×256. Скорость работы приведена в таблице 5

Nvidia 8800 GT	Nvidia GTS 240
0,277632 мс	0,249153 мс

Таблица 5. Скорость вычисления одиночных распределений для базового и парного изображения.

Шаг номер 4 можно условно разделить на 2 части. Первая – вычисление одиночных энтропий. Вторая – вычисление совместной энтропии. Рассмотрим первую часть. Основная операция здесь – это свертка с ядром Гаусса. В качестве ширины свертки было выбрано 7. В этом случае свертка производится со следующим вектором, полученным при помощи функции Гаусса с математическим ожиданием 0 и дисперсией равной 1:

$$\text{gaussian\_ker} = (0.006,\ 0.061,\ 0.242,\ 0.383,\ 0.242,\ 0.061,\ 0.006)$$

Свертка считается одновременно для базового и парного одиночного распределения. Здесь, как и в случае с шагом 2, если необходимые для свертки пиксели выходят за границу тайла, то вместо них выбираются соответствующие граничные пиксели тайла. Вычисление логарифма и нормализация полученных данных путем умножения их на $\frac{1}{N}$ происходят в отдельных ядрах. Причем вычисления также выполняются одновременно для базового и парного одиночного распределения.

Следующая часть четвертого шага – вычисление совместной энтропии. При вычислении можно воспользоваться тем, что свертка с двумерным Гауссовым ядром представима в виде композиции сверток с одномерными одинаковыми ядрами Гаусса. Это верно из-за того, что Гауссова свертка является сепарабельной и симметричной. Поэтому мы разбиваем двумерную

свертку на два шага. Сначала делается горизонтальный проход: свертка происходит построчно, а потом – вертикальный проход: в котором свертка происходит по столбцам. И горизонтальный и вертикальный проходы производят свертки с тем же вектором, что и в случае вычисления одиночной энтропии.

В результате, среднее время работы приведено в таблице 6.

	Nvidia GeForce 8800 GT	Nvidia GTS 240
Вычисление одиночной энтропии	0,03372 мс	0,02528 мс
Вычисление совместной энтропии	0,94336 мс	0,8965 мс

Таблица 6. Скорость вычисления одиночной и совместной энтропий на GPU.

Можно говорить, что предыдущие шаги являются лишь подготовкой к шагу номер 5. На этом шаге будет подсчитана, собственно, попиксельная стоимость. Реализация этого шага на платформе CUDA представляет собой гораздо более сложную задачу, чем кажется на первый взгляд. Наивная реализация, когда один поток обрабатывает один пиксель базового изображения, обращаясь для каждой d из промежутка [0..max_disparity] к массивам в глобальной памяти и хранящим энтропии и интенсивности, работает очень долго. Средняя скорость работы приведена в таблице 7

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
18,346 мс	141,643 мс	150,306 мс

Таблица 7. Время вычисления попиксельной стоимости на основе ВИ без оптимизаций.

Такое большое время работы связано с тем, что обращение к глобальной памяти в CUDA обладает задержкой от 400 до 600 тактов процессора. Более того, как уже было упомянуто в параграфе про CUDA, обращения к глобальной памяти не кэшируются и происходят блоками по 32 байта. В нашем случае, когда мы читаем из массива интенсивностей парного изображения, мы читаем по одному байту, но гораздо более важно то, что мы читает одно и то же много раз. Это становится понятным, если рассмотреть два соседних по ряду пикселя. Пусть они имеет координаты (x, y) и $(x+1, y)$. Поток, обрабатывающий первый из них, прочтет из массива интенсивностей парного изображения все пиксели от $(x - max\_disparity, y)$ до (x, y). В свою очередь, поток, обрабатывающий второй пиксель прочтет из того же массива

пиксели в диапазоне от $(x - max\_disparity + 1, y)$ до $(x + 1, y)$. Тем самым, эти два потока делают на $max\_disparity - 1$ обращений к глобальной памяти больше, чем нужно. Эту проблему можно решить, использую разделенную память.

Пусть мы и дальше будем ставить в соответствие одному пикселю базового изображения один поток. Предположим, что ширина и высота одного блока потоков равны block_width и block_height соответственно. Заметим, что в пределах одного блока все потоки обращаются только к определенной части массива интенсивностей. Более конкретно, это прямоугольный блок, высота которого составляет block_height, а ширина составляет block_width + max_disparity. Смещение этого блок относительно начала массива интенсивностей можно получить по следующей формуле:

$$global\_block\_offset = thread\_block\_number\_in\_row \cdot block\_width - max\_disparity.$$

Суть первой оптимизации состоит в следующем: вместо того, чтобы каждый поток в блоке обращался к глобальной памяти, сначала загрузим тот блок глобальной памяти, к которому происходят обращения, в разделенную память, а дальше все потоки будут читать из разделенной памяти, которая работает существенно быстрее. Сделав таким образом, мы избавимся от большей части повторных обращений к массиву интенсивностей, лежащему в глобальной памяти, ускорив тем самым работу алгоритма. Для загрузки одной строки таблицы используются первые 16 потоков в каждой строке блока потоков. Число 16 выбрано не случайно, это половина одного варпа. За счет специфики работы мультипроцессоров CUDA такой выбор позволит уменьшить время выполнения. Конечно, при определенных условиях на конфигурацию ядра, мы будем читать какие-то интенсивности из глобальной памяти дважды или более. Это будет происходить, если из-за недостаточного размера блоков, два потока из разных блоков будут читать один и тот же элемент массива интенсивностей. Тем не менее, эти затраты можно считать несущественными при достаточно большой ширине блока относительно максимальной диспаратности. Вторая оптимизация состоит в том, что массивы одиночных энтропий помещаются в разделенную память также. Они занимают 512×(размер типа float), что составляет 2048 байт. Прирост в скорости достигается, опять же, за счет уменьшения повторных обращений к массивам одиночных энтропий. Делается 512 обращений к глобальной памяти. До этой оптимизации делалось гораздо больше обращений, а именно

$number\_threads\_in\_block \times max\_disparity + 1$. Результаты работы получившегося алгоритма приведены в таблице 8.

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
6,1868 мс	47,2753 мс	50,3464 мс

Таблица 8. Скорость работы 5 шага после уменьшения количества обращений к глобальной памяти.

Несмотря на существенное уменьшение времени работы 5 шага, оно, по-прежнему, остается достаточно большим. Это связано, главным образом, с тем, что каждый поток читает $max\_disparity + 1$ раз из массива, где хранится совместная энтропия. Эти обращения происходят в ячейки, номер которых зависит от пары интенсивностей: базовой и парной к ней. То есть, можно считать, что обращения не то что нелинейные, но вообще случайные. Поскольку нам необходимо прочесть из памяти всего лишь 4 байта каждый раз, а реально из нее читается 32 байта, случайные обращения, наряду с отсутствием кэша, существенно замедляют скорость работы алгоритма. К счастью, в CUDA есть текстурная память. По словам самих разработчиков CUDA, в соответствии с [9], текстурная память лучше подходит для тех случаев, когда мы обращаемся к памяти нелинейно. Скорость работы падает только в случае непопадания в кэш, и это вполне приемлемо. Текстурная память недоступна на запись напрямую, поэтому нам придется копировать из глобальной памяти в текстурную. Поскольку мы копируем из памяти видеокарты в память видеокарты, то это не занимает существенного времени, т.е. выигрыш в скорости доступа к памяти перекрывает время копирования. Таким образом, поместим в текстурную память массив для совместной энтропии. Кроме того, поместим в текстурную память массив интенсивностей парного изображения. Время работы ускоренного алгоритма составляет:

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
1,28026 мс	7,95776 мс	8,68781 мс

Таблица 9. Скорость работы шага 5, после переноса части данных в текстурную память.

Мы ускорили работу алгоритма почти в 18 раз, тем не менее, ее можно ускорить сильнее, если избавится от конфликтов, возникающих между потоками одной половины варпа, при доступе к разделенной памяти. Разделенная память разбита на 16 банков по 1 Кб каждый. При этом, если мы

рассмотрим массив, например, float чисел, то нулевой элемент массива будет лежать в банке 1, первый элемент в банке 2, второй в банке 3, и так далее. Это значит, что если, например, потоки в пределах одной половины варпа читают данные из этого массива, обращаясь к элементу под номером thread_id + const каждый, то конфликта не возникает, так как каждый из них обращается к своему банку памяти. В то же время, если бы в этом массиве хранились не числа с плавающей точкой, а просто байты, то мы бы получили ситуацию, в которой 4 разных потока пытаются обратиться к одному банку памяти. В этом случае происходит сериализация запросов, а значит скорость работы падает пропорционально количеству конфликтов. Точно такие же проблемы возникают, когда потоки пишут в разделенную память.

Для того чтобы избавиться от конфликтов памяти будем хранить в нашей табличке интенсивностей для блока не числа типа unsigned char, а числа типа unsigned int. Заполнение этой таблицы, как уже говорилось, осуществляется построчно, причем одну строку читает одна половина варпа. Чтение происходит следующим образом: сначала каждый из 16 потоков заполняет по одному элементу из первых 16, потом происходит смещение на 16, и происходит заполнение следующих 16 элементов, и так далее, пока вся строка не будет заполнена. Конфликтов доступа к памяти на запись здесь не происходит, а значит варпы не сериализуются. Также исчезают конфликты при вычислении уже непосредственно стоимостей. Так как в пределах одной половины варпа потоки обращаются к массиву интенсивностей по индексу thread_id + const, что как было выяснено, конфликтов не дает. Эта оптимизация по скорости увеличивает количество разделенной памяти, которую потребляет один блок. Тем не менее, используется меньше трети доступной памяти при любой доступной конфигурации ядра и максимальной диспаратности не превышающей 100. Это означает, что даже если увеличить максимальную диспаратность, то разделенной памяти будет хватать. Время работы алгоритма после применения этой оптимизации дано в таблице 10.

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
1,1577 мс	6,59229 мс	6,90371 мс

Таблица 10. Скорость работы пятого шага, после разрешения конфликтов доступа к разделенной памяти.

Этот шаг показал, что наивный подход к реализации алгоритмов с использованием CUDA может давать результаты более чем в 20 раз отличающиеся от оптимальных результатов.

При реализации шага номер 5 использовались всевозможные оптимизации, начиная от оптимизаций доступа к глобальной памяти и заканчивая разрешением конфликтов доступа к банкам разделенной памяти. Данный пример показывает, что узкие места при реализации алгоритмов на платформе CUDA отличаются от узких мест в обычных приложениях. Это значит, что без глубокого понимания внутренних механизмов CUDA программисту не обойтись, если он хочет добиться приемлемого результата.

Последний шаг в вычислении попиксельных стоимостей на основе ВИ состоит в уточнении карты диспаратности. На самом деле, этот шаг был объединен с предыдущим с целью сокращения количества обращений к глобальной памяти. Время работы пятого шага, представленное в таблицах 7-10, уже включает в себя реализацию шага номер 6.

В таблице 11 представлено суммарное время работы вычисления попиксельных стоимостей на основе ВИ.

	Время работы до любых оптимизаций	Время работы после переноса данных в текстурную память	Финальное время работы (после всех оптимизаций)
Тайл размером 208×160	60,085 мс	11,1804 мс	9,6429 мс
Тайл размером 512×512	520,64246 мс	34,769 мс	30,841 мс
Тайл размером 528×528	575,1386 мс	36,5389 мс	32,647 мс
Изображение 384×288	278,08005 мс	44,9462 мс	40,1464 мс
Изображение 512×512	520,64246 мс	34,769 мс	30,841 мс
Изображение 1024×1024	2309,3642 мс	169,19505 мс	154,49054 мс

Таблица 11. Сводная таблица времени работы вычисления попиксельной стоимости на основе ВИ.

Вторая и четвертая строчка в таблице совпадают из-за того, что второй набор стереопар мы не разбиваем на тайлы.

5.6. Подсчет попиксельной стоимости на основе модифицированного алгоритма AD

Вычисление стоимости на основе взаимной информации позволяет получить качественный результат для итоговой карты диспаратностей, кроме того, ВИ позволяет обрабатывать пары изображений, снятых камерами, настроенными абсолютно по-разному. Но, даже несмотря на достигнутые результаты по скорости, время выполнения остается достаточно большим, в соответствии с таблицей 11, что может быть неприемлемо для ряда задач. Поэтому был реализован альтернативный вариант подсчета попиксельных стоимостей на основе модифицированного алгоритма абсолютных разностей. Следует напомнить, что формула для подсчета стоимости выглядит так:

$$ADC(p,d) = \sum_{p \in W} \left| I_{bp} - I_{mq} \right|, q = p - d.$$

В классическом алгоритме подразумевается, что если подозрительный пиксель парного изображения находится за пределами парного изображения, то есть, если выполнено $p - d < 0$, то соответствующая стоимость называется неопределенной, что соответствует записи в нужную ячейку некоторого заранее оговоренного значения. Модификация алгоритма состоит в следующем: в случае, если $p - d < 0$, в качестве подозрительного пикселя парного изображения выбирается такой пиксель, чтобы $p - d = 0$. Модифицированный алгоритм дает в результате карту диспаратности с меньшим числом ошибок относительно классического подхода.

При реализации модифицированного алгоритма на CUDA были применены подходы, описанные в 5.5., где они применялись для оптимизации вычислений стоимостей на основе взаимной информации. Полученные результаты оптимизаций приведены в таблице 12.

	Время работы до любых оптимизаций	Время работы после переноса данных в текстурную память	Время работы при использовании особой таблицы данных, хранящейся в общей памяти
Тайл размером 208×160	3,07491 мс	0,7366 мс	0,7710 мс
Тайл размером 512×512	33,9606 мс	4,5815 мс	4,7366 мс

Тайл размером 528×528	42,6928 мс	5,5049 мс	6,0667 мс

Таблица 12. Сводная таблица времени работы вычисления попиксельной стоимости на основе модифицированного алгоритма AD для одного тайла до и после оптимизаций.

Здесь следует обратить внимания на следующую деталь: использование общей памяти не ускорило работу алгоритма. Это связано, в первую очередь, с тем, что накладные расходы на использование общей памяти превышают выгоду, полученную за счет быстрого чтения данных. По результатам исследований: от использования общей памяти было решено отказаться. Таким образом, суммарное время работы модифицированного алгоритма AD приведено в таблице 13.

	Время работы до любых оптимизаций	Время работы после переноса данных в текстурную память
Изображение 384×288	12,328 мс	2,9664 мс
Изображение 512×512	33,9606 мс	4,5815 мс
Изображение 1024×1024	170,7977 мс	22,1031 мс

Таблица 13. Сводное время работы модифицированного алгоритма AD для всего изображения до и после оптимизаций.

Как видно из таблицы 11 и таблицы 13: скорость подсчета попиксельной стоимости на основе модифицированного алгоритма AD превышает скорость подсчета на основе ВИ примерно в 7 раз. Тем не менее, область применения данного алгоритма ограничена, а количество ошибок в итоговой карте диспаратности больше, чем при использовании ВИ.

5.7. Суммирование попиксельных стоимостей. Вычисление итоговой карты диспаратности

После того как вычислены попиксельные стоимости, необходимо совершить последний шаг работы алгоритма SGM, а именно, суммирование стоимостей. Как уже упоминалось в 3.2, стандартным подходом для глобальных алгоритмов является нахождение такой конфигурации, которая минимизирует некий функционал энергии вида:

$$E(f) = E_{data}(f) + \lambda \cdot E_{smooth}(f).$$

Такая задача в большинстве случаев является NP-полной. Классическим оптимизационным подходом является динамическое программирование. В этом случае суммирование происходит вдоль одного направления, а значит связи между пикселями в разных рядах практически, а часто и вообще, не учитываются. В [5] был предложен новый подход. Суть его заключается в том, что суммирование будет происходить одновременно по всем направлениям. Рассмотрим этот метод более подробно.

Пусть мы хотим просуммировать стоимости в направлении r для данного пикселя p и диспаратности d. Суммирование происходит рекурсивно:

$$L_r(p,d) = C(p,d) + \min(L_r(p-r,d), L_r(p-r,d-1) + P_1,$$
$$L_r(p-r,d+1) + P_1, \min_i(p-r,i) + P_2).$$

Здесь первое слагаемое – это попиксельная стоимость, а второе – это минимум из четырех чисел, которые зависят от суммированной стоимости для предыдущего пикселя в данном направлении. P_1 - это константный штраф, который налагается в том случае, если диспаратности у двух соседних пикселей отличаются на один. P_2 - это константный штраф, который налагается в том случае, когда диспаратности у двух соседних пикселей отличаются больше, чем на один.

В [5] предлагается производить подсчеты по 16 разным направлениям. В предложенной реализации алгоритма подсчет производится только по восьми направлениям, результат такой оптимизации не сильно ухудшает итоговую карту диспаратностей, скорость же увеличивается в два раза. Так же существует подход, при котором подсчет производится только по пяти направлениям, который также был реализован в рамках работы над данным дипломом.

Реализация данного алгоритма на CUDA является непростой задачей. Это связано с тем, что суммирование должно происходить для каждого направления в отдельности, так как в CUDA не существует средств для синхронизации и обмена данными на уровне ядра между двумя различными блоками потоков. Такие средства становятся необходимы в том случае, когда мы хотим объединить вычисления для, например, трех направлений сразу: левое диагональное сверху-вниз, правое диагональное сверху-вниз,

вертикальное сверху-вниз, так как информация, полученная в результате работы одного блока потоков, должна быть распространена в другие блоки, причем, начиная с некоторого места, блоки должны выполняться в определенном порядке, что противоречит архитектуре CUDA-приложения.

Простейшая реализация суммирования стоимостей в одном направлении, когда каждому потоку соответствует вся цепочка пикселей по этому направлению, дает следующий результат

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
21,1515 мс	73,4793 мс	75,6535 мс

Таблица 14. Время работы «наивного» алгоритма суммирования стоимостей в одном направлении.

Время работы «наивного» подхода достаточно велико. Это связано с тем, что происходит много обращений к глобальной памяти, что, как уже было не раз сказано, является узким местом при программировании с использованием CUDA. При работе над данной реализацией было применено несколько подходов, с целью оптимизировать алгоритм и уменьшить время его работы. Полученные результаты можно разделить на три группы, отличающиеся областью применения.

Первый подход – использование текстурной памяти для хранения массива стоимостей. Данный подход был успешно применен на этапе подсчета попиксельных стоимостей. Однако при суммировании стоимостей, подобный метод дает наихудшие результаты по скорости. Даже простейшая реализация оказалась быстрее. Время работы приведено в таблице 15.

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
22,1416 мс	96,1518 мс	101,1452 мс

Таблица 15. Время работы алгоритма суммирования стоимостей в одном направлении при использовании текстурной памяти для массива стоимостей.

В таблице 15, к тому же, не учитывается время копирования данных в текстурную память. Копирование трехмерного массива данных размером 528×528×100 занимает около 25 мс. Такое низкое время работы получается из-за того, что на данный момент CUDA работает с трехмерными текстурами гораздо хуже, чем с двухмерными. Скорее всего ситуация в скором времени изменится в лучшую сторону, так как быстрая работа с трехмерными текстурами, в компьютерной графике реального времени, очень важна. Тем не менее, сейчас такой подход применять не имеет смысла.

Второй подход – уменьшение используемой глобальной памяти за счет использования локальной или общей памяти в самом ядре. Здесь следует отметить то, что даже если в ядре используется локальная память, доступ к которой по скорости равен доступу к глобальной, суммарный объем памяти по сравнению с простейшей реализацией сильно меньше. Это позволило получить существенный прирост в скорости.

	Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
Локальная память	19,8614 мс	61,3216 мс	66,458 мс
Общая память	9,6714 мс	59,8937 мс	63,2039 мс

Таблица 16. Время работы алгоритма при использовании второго подхода.

Как видно из таблицы, существенного прироста удалось добиться для тайлов маленького размера. К сожалению, из-за отсутствия качественных средств профайлинга приложений, использующих CUDA, точных данных, почему большого прироста удалось добиться только для маленьких тайлов, но не для больших - нет. Тем не менее, можно с уверенностью предполагать, что у менеджера памяти CUDA, в случае тайлов маленького размера, существует больше возможностей для оптимизации, которыми ему удается успешно воспользоваться.

Для больших тайлов можно добиться еще большего увеличения скорости, если отказаться от предварительного подсчета и хранения попиксельных стоимостей. Вместо этого, стоимости можно вычислять тогда, когда это необходимо алгоритму суммирования. В этом и заключается третий подход. В рамках этого подхода также существует две различных реализации – с использованием локальной памяти и с использованием общей памяти. Скорость работы приведена в таблице 17.

	Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
Локальная память	16,7908 мс	54,6402 мс	56,2931 мс
Общая память	12,1549 мс	77,6766 мс	80,1651 мс

Таблица 17. Время работы алгоритма при использовании третьего подхода.

Использование третьего подхода невозможно в том случае, когда попиксельные стоимости вычисляются на основе ВИ. Это связано с тем, что алгоритм подсчета состоит из нескольких однотипных итераций, постепенно уточняющих стоимости, причем уточнение происходит на основе данных, получаемых со всего изображения. Следовательно, не существует возможности вычислять стоимости «на ходу» во время работы алгоритма

суммирования. Тем не менее, если в качестве механизма вычисления стоимостей выбран модифицированный алгоритм AD, данный подход дает наилучшие по скорости результаты для тайлов большого размера.

Таким образом, общее время работы алгоритма суммирования стоимостей по всем направлениям показано в таблице 18, для модифицированного алгоритма AD, и в таблице 19, для ВИ.

	Неоптимизированный подход	Попиксельные стоимости считаются заранее	Попиксельные стоимости считаются в процессе работы
Изображение 384×288	105,7575 мс	48,357 мс	60,7745 мс
Изображение 512×512	367,3965 мс	299,4685 мс	273,201 мс
Изображение 1024×1024	1513,07 мс	1264,078 мс	1125,862 мс

Таблица 18. Общее время работы алгоритма суммирования стоимостей по пяти направлениям для модифицированного алгоритма AD.

	Неоптимизированный подход	Попиксельные стоимости считаются заранее	Попиксельные стоимости считаются в процессе работы
Изображение 384×288	105,7575 мс	48,357 мс	-
Изображение 512×512	367,3965 мс	299,4685 мс	-
Изображение 1024×1024	1513,07 мс	1264,078 мс	-

Таблица 19. Общее время работы алгоритма суммирования стоимостей по пяти направлениям для ВИ.

В таблицах подчеркнуто лучшее время для данного изображения, красным цветом выделяется время, полученное при использовании общей памяти, синим – время работы, полученное при использовании локальной памяти. В таблице 19 в последнем столбце стоит прочерк, так как использование подхода невозможно при вычислении стоимостей на основе взаимной информации.

После того, как подсчитаны суммарные стоимости для каждого из направлений, вычисляется общая сумма всех стоимостей для каждого из пикселей.

$$S(p,d) = \sum_r L_r(p,d).$$

Карта диспаратности строится теперь очень просто. Для каждого пикселя базового изображения p, соответствующая ему диспаратность находится как минимум по всем диспаратностям в общем массиве стоимостей.

$$f(p) = \min_d S(p,d).$$

Поиск минимума в этом массиве осуществляется за следующее время

Тайл размером 208×160	Тайл размером 512×512	Тайл размером 528×528
0,615136 мс	4,07818 мс	4,3519 мс

Таблица 20. Поиск минимума в массиве стоимостей.

5.8. Сравнение с существующими алгоритмами. Результирующее время работы

В таблице 21 приведено суммарное время работы алгоритма для всех тестовых наборов

	Модифицированный AD + суммирование стоимостей по 5 направлениям	Взаимная информация + суммирование стоимостей по 5 направлениям
Изображение 384×288	55,2896 мс	92,4696 мс
Изображение 512×512	280,23208 мс	337,34058 мс
Изображение 1024×1024	1155,7796 мс	1448,48614 мс

Таблица 21. Время работы алгоритма SGM с учетом всех оптимизаций.

На следующем рисунке приведены результаты работы полученного алгоритма по сравнению с другими алгоритмами.

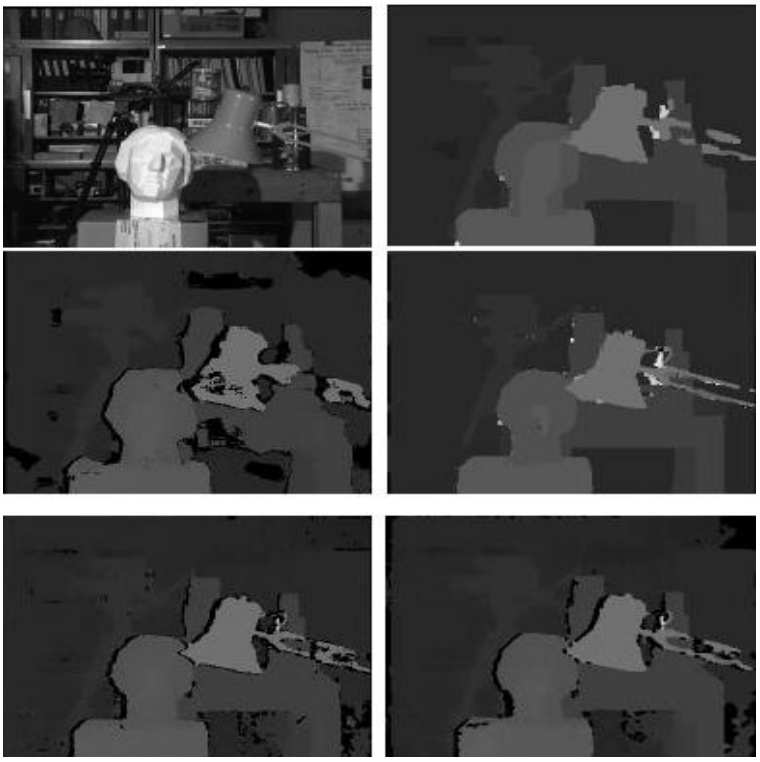

Рисунок 9. Результаты работы алгоритма SGM по сравнению с другими алгоритмами.

На рисунке 9 приведены результаты работы следующих алгоритмов: левый верхний рисунок – исходное базовое изображение. Правый верхний – результат работы алгоритма Belief Propagation из работы [13]. Среднее левое изображение – результат работы локального алгоритма, представленного в работе [6]. Средний правый рисунок – карта диспаратности, полученная путем применения алгоритма Graph Cuts. Левый нижний рисунок – результат работы предложенного в дипломе алгоритма с использованием модифицированного AD-алгоритма для вычисления попиксельных стоимостей. Правый нижний – результат работы предложенного в дипломе алгоритма с использованием взаимной информации. Процент ошибок при работе предложенного алгоритма с использованием взаимной информации составляет примерно 4% и около 5% при использовании модифицированного AD алгоритма. Это сравнимо с глобальными алгоритмами, такими как Belief Propagation, где процент составляет 2-3%.

Для изображения размерами не больше 512×512 применение алгоритма возможно в реальном времени. К сожалению, изображения размерами 1024×1024 невозможно обрабатывать данным алгоритмом чаще, чем 0.86 раз в секунду, что является недостаточным, если работа происходит с видеопотоком. Однако с развитием видеокарт следует ожидать того, что скорость работы алгоритма будет повышаться. Это связано со спецификой платформы CUDA. Исполняемый код, написанный программистом, не

зависит от внутреннего строения видеокарты, то есть увеличение количества мультипроцессоров, пропускной способности памяти или тактовой частоты видеокарты будет способствовать уменьшению времени работы.

6. Заключение

Работа над дипломом велась в рамках исследования, проводимого для компании ЗАО «Ланит-Терком», а в частности для разработки DeepView. По итогам работы удалось достичь следующих результатов:

- Был разработан модифицированный алгоритм на основе SGM, предназначенный для массово-параллельных архитектур.
- Полученный алгоритм был реализован на платформе CUDA. Итоговое приложение имеет гибкие настройки, позволяющие пользователю управлять ходом работы алгоритма.
- Реализована возможность обработки изображений неограниченного размера без увеличения затрат по памяти.
- Предложены и внесены различные оптимизации, позволяющие обрабатывать изображения размерами до 512×512 в реальном времени на скорости не ниже 3.8 кадров в секунду.
- Проведены количественные и качественные сравнения скорости работы алгоритма и его результатов с существующими решениями.

Для изображений размерами 1024×1024 не удалось достичь скорости работы приемлемой для их обработки в реальном времени на современном поколении видеокарт. Тем не менее, время работы алгоритма будет уменьшаться с выходом новых, более современных видеоадаптеров.

Таким образом, все, поставленные в рамках работы, задачи были выполнены.

7. Список литературы

[1] Y.Boykov, O.Veksler, R.Zabih. A variable window approach to early vision. IEEE Transaction on Pattern Analysis and Machine Intelligence, 1998

[2] J.J.Cox, S.L.Hingorani, S.B.Rao, B.M.Maggs. A maximum likelihood stereo algorithm. CVIU, страницы 542-567, 1996

[3] A.Fusiello, V.Roberto, E.Trucco. Efficient stereo with multiple windowing. Proceedings of the Conference on Computer Vision and Pattern Recognition, страницы 858-863, 1997

[4] E.Gamble, T.Poggio. Visual integration and detection of discontinuities: the key role of intensity edges. A. I. Memo 970, MIT, 1987

[5] H.Hirschmuller. Accurate and efficient stereo processing by semi-global matching and mutual information. IEEE Conference on Computer Vision and Pattern Recognition, 2006

[6] H.Hirschmuller, P.R.Innocent, J.M.Garibaldi. Real-time correlation-based vision with reduced border errors. International Journal of Computer Vision, страницы 229-246, 2002

[7] T.Kanade, M.Okutomi. A stereo matching algorithm with an adaptive window: theory and experiment. IEEE Transaction on Pattern Analysis and Machine Intelligence, страница 920, 1994

[8] J.Kim, V.Kolmogorov, R.Zabih. Computing visual correspondence using energy minimization and mutual information. International Conference on Computer Vision, 2003

[9] NVIDIA Company. Cuda Best Programming Guide. http://developer.nvidia.com/object/cuda_3_0_downloads.html, 2009

[10] D.Scharstein. Matching images by comparing their gradient fields. B ICPR, страницы 572-575, 1994

[11] D.Scharstein, R.Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. International Journal of Computer Vision, страницы 7-42, 2002

[12] C.Shannon. A mathematical theory of communication. Bell Systems Technical Journal, страницы 379-423, 1948

[13] J.Sun, N.Zheng, H.Shum. Stereo Matching Using Belief Propagation. IEEE Pattern Analysis and Machine Intelligence, страницы 787–800, 2003

[14] R.Zabih, J.Woodfill. Non-parametric local transforms for computing visual correspondence. Proceedings of the Conference on Computer Vision, страницы 151-158, 1994

[15] J.Zijp. Fast half-float conversions. http://www.fox-toolkit.org/ftp/fasthalffloatconversion.pdf, 2008

[16] C.Zitnick, T.Kanade. A cooperative algorithm for stereo matching and occlusion detection. Tech.Rep. CMU-RI-TR-99-35, Carnegie Mellon University, 1999