Министерство образования республики беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

**ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ**

**Кафедра дискретной математики и алгоритмики**

ДЕМЧЕНКО Андрей Аркадьевич

**РАЗРАБОТКА ПАРАЛЛЕЛЬНЫХ АЛГОРИТМОВ ФЛОЙДА-УОРШЕЛЛА ДЛЯ РЕАЛИЗАЦИИ НА GPU**

Магистерская диссертация

специальность 1-31 81 09 «Алгоритмы и системы обработки больших объёмов информации»

|  |  |
| --- | --- |
|  | Научный руководитель  Николай Александрович Лиходед  доктор физико-математических наук,  профессор |
| Допущена к защите:  «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2017 г.  Зав. кафедрой дискретной математики и алгоритмики  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ В. М. Котов  доктор физико-матетматических наук, профессор | |

Минск, 2017

ОГЛАВЛЕНИЕ

[ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ 4](#_Toc484640780)

[ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ 5](#_Toc484640781)

[ВВЕДЕНИЕ 8](#_Toc484640784)

[Глава 1. Анализ существующих решений и постановка задачи 9](#_Toc484640785)

[1.1 Анализ существующих решений 9](#_Toc484640786)

[1.2 Постановка задачи 9](#_Toc484640787)

[1.3 Подход к решению 10](#_Toc484640788)

[1.4 Выводы 10](#_Toc484640789)

[Глава 2. Параллельные вычисления с использованием NVIDIA CUDA 11](#_Toc484640790)

[2.1 Базовые сведения о GPU 11](#_Toc484640791)

[2.2 Вычислительная архитектура CUDA 12](#_Toc484640792)

[2.3 Ветвления в коде ядра 15](#_Toc484640793)

[2.4 Межпроцессное взаимодействие 17](#_Toc484640794)

[2.5 Организация памяти 17](#_Toc484640795)

[2.6 Выводы 21](#_Toc484640796)

[Глава 3. Алгоритм Флойда-Уоршелла 23](#_Toc484640797)

[3.1 Постановка задачи нахождения кратчайших путей 23](#_Toc484640798)

[3.2 Стандартный алгоритм Флойда-Уоршелла 24](#_Toc484640799)

[3.3 Блочный алгоритм Флойда-Уоршелла 25](#_Toc484640800)

[3.4 Параллельные алгоритмы Флойда-Уоршелла 27](#_Toc484640801)

[3.4.1 Блочный алгоритм 28](#_Toc484640802)

[3.4.2 Многостадийный блочный алгоритм 29](#_Toc484640803)

[3.5 Выводы 31](#_Toc484640804)

[Глава 4. Теоретические аспекты эффективного использования регистров, разделяемой памяти и кэшей GPU 32](#_Toc484640805)

[4.1 Приватизация потоками элементов массивов 32](#_Toc484640806)

[4.1.1 Блоки и потоки вычислений 32](#_Toc484640807)

[4.1.2 Условия приватизации элементов массива 34](#_Toc484640808)

[4.2 Определение максимально возможных блоков вычислений для GPU с заданными характеристиками объема памятей 35](#_Toc484640809)

[4.3 Выводы 37](#_Toc484640810)

[Глава 5. Новая модификация многостадийного блочного параллельного алгоритма Флойда-Уоршелла 38](#_Toc484640811)

[5.1 Описание алгоритма 38](#_Toc484640812)

[5.2 Выводы 39](#_Toc484640813)

[Глава 6. Реализация алгоритма и вычислительные эксперименты 40](#_Toc484640814)

[6.1 Используемые инструменты и технологии 40](#_Toc484640815)

[6.2 Сравнение различных реализаций алгоритма Флойда-Уоршелла 41](#_Toc484640816)

[6.3 Возможные направления дальнейших исследований 45](#_Toc484640817)

[6.4 Выводы 46](#_Toc484640818)

[ЗАКЛЮЧЕНИЕ 47](#_Toc484640819)

[БИБЛИОГРАФИЧЕСКИЙ СПИСОК 48](#_Toc484640820)

[ПРИЛОЖЕНИЯ 49](#_Toc484640821)

[ПРИЛОЖЕНИЕ А. Модификация многостадийного блочного алгоритма с использованием константной памяти 49](#_Toc484640822)

[ПРИЛОЖЕНИЕ Б. Модификация многостадийного блочного алгоритма с использованием текстурной памяти 55](#_Toc484640824)

# ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ

**GPU** (*Graphics Processing Unit*) - специализированная плата, предназначенная для быстрых манипуляций с памятью с целью ускорения подготовки изображений в кадровом буфере для последующего отображения на экране.

**CUDA** (*Compute Unified Device Architecture*) - программно-аппаратная архитектура параллельных вычислений, которая позволяет существенно увеличить вычислительную производительность благодаря использованию графических процессоров фирмы Nvidia.

# ОБЩАЯ ХАРАКТЕРИСТИКА РАБОТЫ

Магистерская диссертация, 60 стр., 8 рис., 2 прил., 9 источников.

ПАРАЛЛЕЛЬНЫЕ АЛГОРИТМЫ, ПОИСК КРАТЧАЙШИХ ПУТЕЙ В ГРАФЕ, АЛГОРИТМ ФЛОЙДА-УОРШЕЛЛА, БЛОЧНЫЙ АЛГОРИТМ, CUDA, GPU.

*Объект исследования*: алгоритмы поиска кратчайших путей в графе, алгоритм Флойда-Уоршелла.

*Цель работы*: исследование алгоритма Флойда-Уоршелла нахождения кратчайших путей между всеми парами вершин в графе, разработка и реализация новых параллельных версий алгоритма с использованием графического процессора.

*Предмет исследования*: параллельные алгоритмы Флойда-Уоршелла для реализации на GPU.

*Результат*: разработаны и реализованы на языке C++ с использованием технологии CUDA новые модификации параллельных алгоритмов Флойда-Уоршелла, предназначенные для работы на графическом процессоре. Проведены сравнительные эксперименты новых и существующих алгоритмов, выявлены основные недостатки предложенных модификаций, рассмотрены возможные направления дальнейших исследований.

*Область применения*: задачи маршрутизации, логистики в системах навигации, робототехники, экономики, компьютерных играх.

# АГУЛЬНАЯ ХАРАКТЭРЫСТЫКА РАБОТЫ

Магістарская дысертацыя, 60 ст., 8 мал., 2 прыл., 9 крынiц.

ПАРАЛЕЛЬНЫЕ АЛГАРЫТМЫ, ПОШУК НАЙКАРОТКІХ ШЛЯХОУ У ГРАФЕ, АЛГАРЫТМ ФЛОЙДА-УОРШАЛА, БЛОЧНЫ АЛГАРЫТМ, CUDA, GPU.

*Аб’ект даследавання*: алгарытмы пошуку найкартокіх шляхоў у графе, алгарытм Флойда-Уоршала.

*Мэта работы*: даследаванне алгарытма Флойда-Уоршала пошуку найкароткіх шляхоў паміж усімі парамі вяршын у графе, распрацоўка і рэалізацыя новых паралельных версій алгарытму з выкарыстаннем графічнага працэсара.

*Прадмет даследавання*: паралельныя алгарытмы Флойда-Уоршала для рэалізацыі на GPU.

*Результат*: распрацаваны і рэалізаваны на мове С++ з выкарыстаннем тэхналогіі CUDA новыя мадыфикацыі паралельных алгарытмаў Флойда-Уоршала, прызначаныя для выканання на графічным працэсары. Праведзены параўнальные эксперыменты новых і існуючых алгарытмаў, выяўлены асноўныя недахопы прапанаваных мадыфікацый, разгледжаны магчымыя напрамкі далейшых даследаванняў.

*Вобласть ужывання*: задачы маршрутызацыі, лагістыкі ў сістэмах навігацыі, робататэхнікі, эканомікі, камп'ютэрных гульнях.

# GENERAL THESIS DESCRIPTION

Master’s thesis, 60 p., 8 images, 2 app., 9 sources.

PARALLEL ALGORITHMS, ALL PAIRS SHORTEST PATH IN GRAPH, FLOYD-WARSHALL ALGORITHM, BLOCK ALGORITHM, CUDA, GPU.

*Object of research*: algorithms for finding all-pairs shortest paths in graph, Floyd-Warshall algorithm.

*Purpose of work*: research of the Floyd-Warshall algorithm for finding all-pairs shortest paths in graph, design and implementation of new parallel versions of the algorithm for graphics processing unit.

*Subject of research*: parallel Floyd-Warshall algorithms for implementation on the GPU.

*Result of work*: new modifications of parallel Floyd-Warshall algorithm for running on graphics processing unit have been developed and implemented in C++ language using CUDA technology. Comparative experiments of new and existing algorithms have been carried out, basic flaws of the proposed modifications have been identified, and possible directions for further research have been considered.

*Application Area*: tasks of routing, logistics in navigation systems, robotics, economics, computer games.

# ВВЕДЕНИЕ

Проблема поиска кратчайших путей хорошо исследована в теории графов. Полученные алгоритмы имеют широкое применение в задачах маршрутизации и задачах логистики. Все алгоритмы решения задачи поиска кратчайших путей между всеми парами вершин графа имеют нелинейную трудоемкость, поэтому одним из путей решения задачи большого размера является разработка параллельных версий алгоритма.

Для решения задачи поиска кратчайших путей между всеми парами вершин в графе существует достаточно много различных алгоритмов. Алгоритмы отличаются своей скоростью работы, предрасположенностью к распараллеливанию и другими свойствами.

Среди множества этих алгоритмов самыми известными являются следующие (указана также вычислительная сложность алгоритма):

* Алгоритм Симбела, основанный на умножении матриц – .
* Алгоритм Флойда-Уоршелла – .
* Алгоритм Джонсона –

Одним из наиболее используемых на практике алгоритмов для поиска кратчайших путей между всеми парами вершин во взвешенных графах является алгоритм Флойда-Уоршелл. Именно этот алгоритм будет далее рассматриваться в работе, т.к. он обладает хорошими задатками к распараллеливанию.

В настоящее время большой популярностью среди архитектур, позволяющих параллельное исполнение алгоритмов, пользуются графические процессоры . Это связано с их внутренним строением – графические процессоры позволяют запускать тысячи потоков, каждый из которых будет работать на своем собственном вычислительном ядре.

В данной работе рассматривается возможность применения константной и текстурной памяти при реализации на графических процессорах известного многостадийного блочного алгоритма Флойда-Уоршелла.

# Глава 1. Анализ существующих решений и постановка задачи

## 1.1 Анализ существующих решений

Одной из основных проблем использования алгоритма Флойда-Уоршелла для решения задачи поиска кратчайших путей в графе является трудоемкость алгоритма, равная (подробнее алгоритм будет рассмотрен в главе 2).

Существует также блочная модификация этого алгоритма, обладающая такой же асимптотикой трудоемкости, которая и используется в качестве основы для параллельных алгоритмов.

Так как последовательному алгоритму Флойда-Уоршелла на новой итерации необходимо использовать матрицу расстояний, полученную на предыдущей итерации, операции внутри итерации можно выполнять независимо, то есть каждый поток вычислений при реализации на графических процессорах может заниматься обновлением одного элемента. Такой подход был предложен в [1] и у него есть большой минус – постоянные обращения к глобальной памяти могут быть довольно медленными.

Как следствие в 2008 году в [2] была предложена реализация блочного алгоритма Флойда-Уоршелла на GPU. Основная идея данного подхода заключается в использовании разделяемой памяти блоков вместо глобальной памяти устройства. Рассмотренная в [2] техника позволяет сократить количество данных, посылаемых через шину глобальной памяти, в 32 раза по сравнению с базовым алгоритмом для GPU.

Дальнейшей оптимизацией алгоритма путем уменьшения объема используемой разделяемой памяти в блочном алгоритме Флойда-Уоршелла занялись B. Lund и J. W. Smith в 2010 году [3]. Главная идея их подхода – многостадийное чтение одиночно-зависимых тайлов при вычислении дважды-зависимых.

Дальнейшей целью работы ставится усовершенствование подхода описанного в [3], уменьшив при этом количество данных пересылаемых через шину глобальной памяти.

## 1.2 Постановка задачи

В рамках работы над магистерской диссертацией ставится задача изучить известные параллельные алгоритмы Флойда-Уоршелла, исследовать некоторые теоретические аспекты эффективного использования регистров, разделяемой памяти и кэшей GPU, определить максимально возможных блоков вычислений для GPU с заданными характеристиками объема памятей, разработать и программно реализовать модификации параллельного алгоритма Флойда-Уоршелла для реализации на графических процессорах.

## 1.3 Подход к решению

При решении будет использована теория алгоритмов на графах, а также теория параллельных вычислений.

## 1.4 Выводы

В первой главе получены следующие результаты:

* выполнен анализ литературы по теме магистерской диссертации и выделены основные проблемы;
* сформулирована задача;
* описан подход к решению, который будет рассмотрен в следующих главах.

Перед тем как переходить к разработке алгоритмов, подробнее рассмотрим архитектуру вычислительного устройства, с которым предстоит работать.

# Глава 2. Параллельные вычисления с использованием NVIDIA CUDA

## 2.1 Базовые сведения о GPU

Исторически, для обработки изображений и отрисовки различных данных на экране, использовался центральный процессор компьютера. Со временем инженеры пришли к выводу, что есть смысл в том, чтобы вынести вычисления, связанные с подготовкой и отрисовкой графической информации, с основного процессора на какое-то другое устройство – графическую карту.

Первые версии графических карт были очень узко специализированы и подходили только для визуализации двумерных данных. Однако, с развитием игровой индустрии и появлением “псевдотрехмерных” сцен в играх, остро стал вопрос об обработке и 3D визуализации.

В ходе своего развития, графические карты все пополнялись новыми возможностями и росли по мощности, однако в течении длительного времени для них был реализован только ограниченный набор операций над входными данными. У программистов не было никакой возможности визуализации алгоритмов, и для повышения гибкости появились небольшие программы, выполняющиеся видеокартой каждого пикселя - шейдеры (shaders). В их задачи входили преобразования над пикселями и затенение — расчет освещения в точке.

В настоящее время шейдеры все еще очень распространены, но стоит помнить, что они изначально разрабатывались для покрытия очень узкого класса задач.

В то же время, на возникших мощностях (графические карты уже давно обогнали по суммарной производительности средние CPU) хотелось производить вычисления общего назначения. Так и появилась идея создать унифицированный подход для программирования процессора графической карты – GPU.

Итак, графический процессор (GPU, Graphics Processing Unit) – специализированная плата, предназначенная для быстрых манипуляций с памятью с целью ускорения подготовки изображений в кадровом буфере для последующего отображения на экране.

В настоящее время графические процессоры очень распространены – используются в персональных компьютерах, мобильных телефонах, встраиваемых системах и игровых приставках.

Современные GPU позволяют очень эффективно работать с компьютерной графикой и производить обработку изображений. Благодаря высоко-параллельной структуре современных графических карт их использование для параллельных вычислений на больших блоках данных эффективнее, чем использование обычных CPU, даже и с большим количеством ядер.

В современных персональных компьютерах GPU может быть встроена как в видео карту, так и прямо в материнскую плату. В некоторых моделях графический процессор встраивается прямо в ту же плату, что и главный процессор.

## 2.2 Вычислительная архитектура CUDA

Архитектура CUDA по характеру работы с данными относится к *SIMD* – одна команда применяется к множеству данных. Одним из основных понятий является понятие мультипроцессора.

Мультипроцессор (SM, streaming multiprocessor) – многоядерный SIMD процессор, который позволяет выполнять в каждый определенный момент времени на всех ядрах только одну инструкцию. При этом каждое из ядер (SP, simple/streaming processor) – скалярное, т.е. не поддерживает векторных операций.

В документации по архитектуре CUDA, под *устройством (device)* понимается видеоадаптер, работающий с драйвером CUDA, или же другое специализированное устройство, предоставляющее интерфейс для программирования, сходный с интерфейсом CUDA. Для реализации алгоритмов и программ под архитектуру CUDA нет необходимости знать особенности конкретного устройства. Достаточно реализовать всю логику, опираясь на программный интерфейс CUDA.

Под *хостом (host)* в документации CUDA называют программу, находящуюся в оперативной памяти компьютера, выполняющуюся на CPU и вызывающую управляющие функции для контроля работы устройства.

Устройство логически представляется как набор мультипроцессоров (рисунок 2.1), управляемых через драйвер CUDA.

|  |
| --- |
|  |
| **Рисунок 2.1 – Структура аппаратного обеспечения GPU** |

Программа, которая выполняется на CUDA-совместимом устройстве, называется ядром (Kernel).

Особенностью параллелизма на архитектуре CUDA является блочно-сетевая организация множества потоков выполнения. Такой тип организации является достаточно необычным для многопоточных приложений. Облегчает его использование то, что драйвер CUDA распределяет ресурсы устройства между потоками выполнения автоматически.

Как видно на рисунке 2.2 , все потоки, выполняющие ядро **Kernel 1**, объединяются в *блоки* **(Block)**, которые, в свою очередь, объединяются в *сетку* **(Grid)**. Заметим так же, что потоки в этой нотации индексируются двумя индексами.

Такая организация потоков позволяет удобно работать с трехмерными, двухмерными и одномерными индексами (в зависимости от того, какой из способов удобнее программисту).

В общем случае индексы являются трехмерными векторами. Для каждого потока будут известны:

* индекс потока в блоке (threadIdx);
* индекс блока в сетке (blockIdx).

Фактически, через эти индексы и осуществляется управление логикой обработки данных. Программа составляется таким образом, что каждый поток определяет свою область ответственности в обрабатываемых данных именно по этим индексам.

Нам, как разработчикам кода под архитектуру CUDA, гарантируется, что один блок будет исполняться на одном мультипроцессоре устройства. Но на один мультипроцессор может приходиться выполнение нескольких различных блоков.

Блок потоков выполняется на мультипроцессоре частями (*пулами*), которые в документации называют **warp**. Размер **warp** на большинстве современных GPU c поддержкой CUDA равен 32 потокам. Программа внутри пула **warp** исполняется в SIMD стиле. Это означает, что во всех потоках внутри **warp** одновременно может выполняться лишь одна инструкция.

Стоит сделать оговорку, что размер **warp** не гарантируется спецификацией графического процессора. К тому же, во многих архитектурах, поддерживающих CUDA, количество процессоров внутри одного мультипроцессора равно 8, а не 32. Таким образом, процессы внутри одного **warp** не могут выполняться одновременно и будут разбиты на блоки, выполняемые последовательно (т.к. потоковые процессоры внутри мультипроцессора являются скалярными).

|  |
| --- |
|  |
| **Рисунок 2.2 – Организация потоков в CUDA** |

## 2.3 Ветвления в коде ядра

После описания того, как выполняются инструкции внутри потоков одного пула, может возникнуть закономерный вопрос: если все потоки в один момент времени выполняют одну и ту же инструкцию, то как быть с ветвлениями?

Все верно – если в коде программы есть условный переход, то и последовательность инструкций может быть различной для разных потоков.

Выход в сложившейся ситуации – стандартный для SIMD-систем. Рассмотрим схему на рисунке 2.3.

|  |
| --- |
|  |
| **Рисунок 2.3 – Организация ветвления в SIMD** |

В этом примере мы имеем 10 потоков выполнения. 9 из них должны выполнить операторы **A**, **B**, **D** (т.е. условие **cond** будет выполнено), а один – **A**, **C**, **D** (т.к. условие для него будет ложно).

Исходя из того, что все потоки выполнения должны исполнять единовременно одну и ту же инструкцию, не получится запустить 9 потоков на выполнение операции **B**, а оставшийся поток – операции **C**.

Для решения этой проблемы, предлагается запустить обработку инструкции **B** на всех потоках, при этом “заблокировав” работу одного потока, чтобы он не мог модифицировать никаких данных. Затем блокируем работу всех потоков кроме того, которому нужно выполнить операцию **C** и выполняем ее. Операцию **D** все потоки выполняют одновременно.

Как можно заметить, ветвления при этом подходе работаю значительно хуже, чем на обычных параллельных процессорах.

Во-первых, процессор каждого “заблокированного” потока выполняет излишние операции.

Во-вторых, все процессоры выполнят все ветки условного оператора.

Данный подход к выполнению условных конструкций реализуется на уровне драйвера CUDA и программисту делать блокировку потоков явно не надо. Этим вычисления с использованием CUDA выгодно отличаются от использования SSE команд в современных CPU для параллелизации – для их применения требуется вручную объединять данные по блокам фиксированного размера (обычно это 4), выравнивать специальным образом данные и использовать очень низкоуровневые примитивы.

Рассмотрев принцип работы ветвлений на архитектуре CUDA можно сделать вывод, что сами ветвления являются причиной падения производительности. Замедление вызывают только те ветвления, которые разбивают потоки одного пула на группы. При этом если потоки расходятся внутри одного блока, но в разных пулах или вообще в разных блоках, это не окажет никакого замедляющего эффекта.

## 2.4 Межпроцессное взаимодействие

Программный интерфейс, предоставляемый программисту под архитектуру CUDA, позволяет организовывать взаимодействие (синхронизацию) между потоками только в рамках одного блока.

Для синхронизации выполнения используется функция \_\_synchtreads. Она является примитивом для реализации барьерной синхронизации потоков.

Обмен данными возможен через разделяемую память, т.к. она общая для всех задач внутри блока.

## 2.5 Организация памяти

В архитектуре CUDA выделяют шесть видов памяти (рисунок 2.4):

* регистры;
* локальная память;
* глобальная память;
* разделяемая память;
* константная память;
* текстурная память.

|  |
| --- |
|  |
| **Рисунок 2.4 – Виды памяти в CUDA** |

Такое большое разнообразие видов памяти обусловлено первичным предназначением видеокарты. Разработчики GPU стараются разрабатывать не универсальные вычислители, а устройства для более быстрого обсчета и отрисовки графических задач. Соответственно, в каких-то местах приходится жертвовать универсальностью за скорость при неизменной цене.

Ниже будут рассмотрены подробнее каждый из видов памяти.

**Регистры**

Как и обычные CPU, процессоры в графических картах имеют рядом с собой небольшое количество сверхбыстрой памяти – регистров. По сути, процессор не может производить операции над операндами в оперативной памяти или другой нерегистровой памяти – нужно предварительно эти данные скопировать в регистры.

При компиляции программы в машинный код, компилятор старается размещать все локальные переменные функций в регистрах. Это позволяет наиболее оперативно с ними работать.

Количество доступных регистров в одном мультипроцессоре зависит от конкретного устройства, но обычно это 8192 32-разрядных регистра. Если брать количество потоков в блоке равным 64, то получаем всего 128 регистров на поток. Компилятор не выделит все 128 регистров под один поток – скорее всего он выделит порядка 40, а остальные переменные будут находиться в локальной памяти.

Так происходит из-за того, что на одном мультипроцессоре может обрабатываться одновременно несколько блоков, а компилятор, с целью увеличить эффективность, старается максимизировать число параллельно работающих блоков.

**Локальная память**

Как было замечено выше, поместить все необходимые переменные в регистровую память не всегда возможно. В таких случаях компилятор помещает часть переменных в локальную память. Физически эта память является аналогом глобальной памяти и работает с той же скоростью. Контролировать использование этой памяти напрямую нельзя, так что в целях оптимизации скорости вычислений обычно рекомендуют размещать переменные в регистровой памяти, если возможно.

**Глобальная память**

При работе с GPU не через драйвер CUDA, могут возникать проблемы, которые обычно не возникают при программировании в системах с общей UMA памятью – не вся память одинаково доступна для кода, выполняющегося на GPU. Плюсом же архитектуры CUDA является возможность произвольной адресации глобальной памяти – можно читать из любой ячейки памяти и писать в произвольную ячейку.

Однако, глобальная память обладает и своим недостатком – она не кэшируется. Из-за этого она работает очень медленно и количество обращений к ней стоит минимизировать.

Обычно, глобальную память используют для сохранения результатов работы для отправки их на хост.

Распределением глобальной памяти для выполнения ядра занимается хостовая программа.

Можно создавать переменные в глобальной памяти – для этого используется служебное слово \_\_global\_\_.

Для выделения динамических блоков в глобальной памяти используется функция cudaMalloc – полный аналог функции malloc.

Для копирования хостовых данных с и на устройство можно использовать функцию cudaMemcpy – полный аналог функции memcpy, но принимающий последним аргументом параметр “направление копирования”. Эта функция так же должна выполняться с хоста.

Для более эффективной работы с глобальной памятью рекомендуют в потоках ядра CUDA обращаться к последовательным ячейкам памяти, причем с учетом выравнивания (4, 8 или 16 байт).

**Разделяемая память**

Разделяемая память является так же некэшируемой, но обладает более высокой скоростью доступа. Обычно ее используют как управляемый кэш.

В современных GPU на один мультипроцессор обычно доступно всего 16 килобайт разделяемой памяти.

На рисунке 2.4 видно, что разделяемая память одна на весь блок. Соответственно ее можно использовать для обмена данными только между потоками одного блока.

С точки зрения сохранности и доступности данных в разделяемой памяти, спецификация CUDA гарантирует, что во время исполнения блока на мультипроцессоре содержимое разделяемой памяти будет сохраняться. Однако после того как на мультипроцессоре сменился блок, не гарантируется, что содержимое старого блока сохранилось.

Чтобы поместить переменную в разделяемой памяти, нужно объявить ее с квалификатором \_\_shared\_\_.

**Константная память**

Как видно на рисунке 2.1, константная память кэшируется, так что доступ к ней довольно быстрый. Кэш общий для всех задач внутри блока.

Как следует из названия, константная память с устройства доступна только на чтение. На хосте же в константную память можно предварительно записать данные, вызвав функцию cudaMemcpyToSymbol.

Константная память очень удобна в использовании. Можно размещать в ней данные любого типа и читать их при помощи простого присваивания.

Переменные, которые должны быть отображены в константную память нужно помечать квалификатором \_\_constant\_\_.

Единственный недостаток константной памяти – ее размер. В современных GPU он составляет порядка 64 килобайт на все устройство. Соответственно, в ней можно хранить только небольшие куски часто используемых данных.

**Текстурная память**

По аналогии с константной памятью, текстурная память кэшируется и является общей для всех процессов одного блока.

Название данного вида памяти происходит от понятия “текстура” или “текстурирование” – процесс наложения текстуры (картинки) на полигон в процессе растеризации.

Текстурная память оптимизирована под выборку 2D данных и имеет следующие возможности:

быстрая выборка значений фиксированного размера (байт, слово, двойное или учетверенное слово) из одномерного или двухмерного массива

нормализованная адресация числами типа float в интервале [0,1)

аппаратная линейная или билинейная (в случае 2D) интерполяция соседних значений в случае нормализованной адресации

аппаратная обработка выхода за границу массива с использованием двух режимов: **clamp** и **wrap**.

Текстурная память является просто логическим представлением части глобальной памяти. Выделив функцией cudaMalloc некий участок глобальной памяти, можно затем указать с помощью функции cudaBindTexture, что данный участок теперь будет рассматриваться как текстурная память.

Плюсы текстурной памяти заключаются в том, что текстурной памяти можно использовать достаточно много (обычно сотни мегабайт).

Но минус в том, что набор операций и типов, совместимых с текстурной памятью, очень узкий. Из текстурной памяти можно читать данные только встроенных в компилятор типов, имеющих размер 1, 2, 4, 8 или 16 байт, и только с помощью специальных функций — tex1D, tex2D или tex1Dfetch, tex2Dfetch. Так что нельзя сделать указатель на область текстурной памяти и работать с ним как со структурой произвольного размера. В случае с константной памятью, так делать было можно. Для того чтобы воспользоваться объектом или структурой в текстурной памяти, его придется сначала оттуда вычитать в регистровую или разделяемую память (используя одну из специальных функций, упомянутых выше) и после этого только использовать.

## 2.6 Выводы

Во второй главе получены следующие результаты:

* приведены базовые сведения о GPU;
* рассмотрены основные особенности программирования под архитектуру CUDA.

Далее, перед разработкой новых модификаций, определим основные понятия, касающиеся алгоритма Флойда-Уоршелла, а также рассмотрим известные параллельные алгоритмы.

# Глава 3. Алгоритм Флойда-Уоршелла

## 3.1 Постановка задачи нахождения кратчайших путей

Пусть – ориентированный взвешенный граф, у которого – множество вершин, а – множество упорядоченных пар или дуг. Для графа *G* задана весовая функция , которая каждой дуге ставит в соответствие вес – действительное число .

Суммарная длина пути равна сумме весов всех входящих в него дуг:

|  |  |
| --- | --- |
|  | (3.1) |

Под длиной кратчайшего пути (весом пути), соединяющего вершины , будем понимать величину , которая определяется по формуле:

|  |  |
| --- | --- |
|  | (3.2) |

где *P* – множество всех путей, соединяющих вершины *u* и *v*. Кратчайшим путем из *u* в *v* будем считать любой путь *p* из вершины *u* в *v,* длина которого равна:

***Задача о кратчайшем пути между всеми парами вершин:*** Для каждой вершины требуется найти кратчайший путь в каждую вершину .

***Входные данные*:** Пронумеруем вершины множества натуральными числами от 1 до *n*, где . По графу *G* построим матрицу смежности *W* размером , у которой элемент равен весу ориентированного ребра , если оно существует в графе, и иначе.

|  |  |
| --- | --- |
|  | (3.3) |

Веса ребер – действительные числа, необязательно положительные. Однако будем предполагать, что в графе отсутствуют циклы отрицательного веса.

***Выходные данные:*** матрица кратчайших расстояний *D*, в которой элемент равен длине кратчайшего пути из вершины в , если такой существует, либо равен +\infty, если вершина не достижима из .

## 3.2 Стандартный алгоритм Флойда-Уоршелла

Дан граф . Алгоритм Флойда-Уоршелла допускает наличие ребер с отрицательным весом в графе. Работает корректно, если в графе нет циклов отрицательного веса, а в случае, когда такой цикл есть, позволяет найти хотя бы один такой цикл.

В основе алгоритма Флойда-Уоршелла лежит итерационный поиск кратчайшего пути. На каждой итерации последовательно рассматриваются промежуточные вершины для кратчайшего пути. Под промежуточной вершиной для простого пути понимается любая вершина , отличная от начальной и конечной вершин.

Пусть вершины графа пронумерованы натуральными числами от 1 до *n*. По матрице смежности графа *G* построим матрицу , элементы которой определяются следующим образом:

|  |  |
| --- | --- |
|  | (3.4) |

На *k*-й итерации () для каждой пары вершин *i, j* рассмотрим все пути из вершины *i* в вершину *j*, промежуточные вершины которых могут быть выбраны только из множества , и допустим, что среди них *p* – это кратчайший путь с минимальным весом. Обозначим вес пути *p* как . Тогда для вычисления длины пути *p* необязательно перебирать все подходящие пути – можно просто воспользоваться вычислениями, полученными на предыдущей итерации. Для этого рассмотрим два варианта:

***k не является промежуточной вершиной для пути p.*** В таком случае все промежуточные вершины пути *p* принадлежат множеству . Получаем, что кратчайший путь из вершины *i* в вершину *j* со всеми промежуточными вершинами из множества одновременно является кратчайшим путем из вершины *i* в вершину *j* со всеми промежуточными вершинами из множества ,поэтому длина пути *p*: .

***k является промежуточной вершиной для пути p.*** Тогда путь p можно разбить на два участка: и , где – кратчайший путь из вершины *i* в *k*, промежуточные вершины которого принадлежат множеству ,а – кратчайший путь из вершины *k* в *j* с промежуточными вершинами из множества *.* Тогда .

Исходя из вышесказанного, общую формулу для вычисления длины кратчайшего пути из *i* в *j*, промежуточные вершины которого принадлежат множеству , можно представить в виде:

|  |  |
| --- | --- |
|  | (3.5) |

Очевидно, что матрица представляет собой матрицу кратчайших расстояний *D* между всеми парами вершин графа.

Отметим, что нет необходимости на *k*-й итерации создавать временную матрицу : все изменения можно делать сразу в матрице D. На самом деле, уменьшая величину в матрице расстояний, нельзя ухудшить итоговый вес кратчайшего пути между двумя другими вершинами, обработанными позднее.

Псевдокод базового алгоритма Флойда:

for (size\_t k = 0, k < n; ++k) {  
 for (size\_t i = 0; i < n; ++i) {  
 for (size\_t j = 0; j < n; ++j) {  
 d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

}

}

}

Время работы алгоритма Флойда-Уоршелла определяется тремя вложенными друг в друга циклами *for*. Причем, операция *min*, находящаяся под циклами, обладает константным временем работы, поэтому общая трудоемкость алгоритма . Дополнительной памяти необходимо .

## 3.3 Блочный алгоритм Флойда-Уоршелла

Опишем блочную модификацию алгоритма Флойда-Уоршелла. Пусть дан граф и по данному графу построена матрица смежности W. Для простоты изложения будем считать, что существует такой размер блока (обозначим его b), что n делится на b. Тогда разобьем матрицу смежности W на блоки размером и всего таких блоков , где . Блок , где , можно представить в виде:

|  |  |
| --- | --- |
|  | (3.6) |

Будем выполнять алгоритм блочными итерациями по . На-й блочной итерации существуют следующие виды блоков:

* – ведущий блок;
* – блоки ведущей строки;
* – блоки ведущего столбца;
* – остальные блоки.

Будем говорить, что матрица расстояний находится в *m*-м состоянии, если для всех пар вершин графа уже подсчитаны длины кратчайших путей, промежуточные вершины которых принадлежат множеству .

По завершению -й блочной итерации, текущая матрица находится в -м состоянии. Задачей -й итерации является обновить состояние данной матрицы до -го состояния. При этом во время вычисления текущей итерации для разных типов блоков существуют следующие зависимости:

Для вычисления элементов ведущего блока нужны только его элементы (его будем вычислять первым).

Для вычисления блоков ведущей строки и ведущего столбца необходимы их собственные элементы и уже подсчитанные элементы ведущего блока. Между собой эти блоки никак не конкурируют, поэтому последовательность их вычисления может быть произвольной.

Для обычных блоков нужны их собственные элементы, а также элементы соответствующих блоков с ведущей строки и столбца. Эти блоки вычисляются в последнюю очередь в произвольном порядке.

|  |  |  |
| --- | --- | --- |
| k1.jpg | k2.jpg | k3.jpg |
| **Рисунок 3.1** – Демонстрация итераций блочного алгоритма с размером блока b=2 для графа из 6 вершин. | | |

Опишем подробнее -ю блочную итерацию:

* Вычисляем главный блок. Постепенно повышаем состояние элементов матрицы, соответствующих главному блоку, начиная с состояния до состояния. По сути, запускаем обычный алгоритм Флойда-Уоршела на ведущем блоке. В итоге сохраняется только последняя версия ведущего блока (в состоянии).
* Для каждого блока с ведущей строки и каждого столбца запускаем процесс повышения состояния. В случае, когда возникает необходимость обращаться к элементам главного блока – обращаемся к его последней версии.
* Итерационно повышаем состояние оставшихся блоков. Если необходимо прочитать значение из главного блока, то берем последнюю версию данного значения.

Псевдокод блочной модификации алгоритма Флойда-Уоршелла:

for (size\_t = 0; < n/b; ++) {

UpdateLeadBlock();

UpdateLeadRowBlocks();

UpdateLeadColumnBlocks();

UpdateAllBlocks()

}

Легко заметить, что по завершению последней *N*-й итерации, блочный алгоритм Флойда-Уоршелла приведет к корректному результату. То есть, полученная в итоге матрица будет представлять собой матрицу кратчайших расстояний между всеми парами вершин.

Как можно было заметить из описания алгоритма, он не делает никаких лишних по сравнению с базовым алгоритмом действий. Поэтому время работы аналогично базовой версии – . Так как на каждой итерации нам нет необходимости хранить промежуточные слои матрицы расстояний (достаточно только последнего слоя), то асимптотика по памяти будет , как и у базового алгоритма.

## 3.4 Параллельные алгоритмы Флойда-Уоршелла

Последовательный алгоритм Флойда-Уоршелла, описанный в разделе 3.2, обладает естественным параллелизмом в рамках одной итерации. Для вычисления одного элемента на *k*-й итерации нам могут понадобиться только элементы с -й итерации, что позволяет вычислять все элементы матрицы *D* на очередной итерации параллельно.

В виду внутреннего устройства GPU и особенностей CUDA мы можем запускать по одному потоку для каждого элемента матрицы. Данные потоки, обладая знаниями о своем положении в матрице потоков, будут вычислять положение элемента, за который они отвечают, вычитывать из памяти нужные значения с предыдущей итерации и обновлять значение вверенного им элемента в памяти.

Поскольку каждому потоку может понадобиться доступ к любому из элементов матрицы, единственным местом, где может храниться эта матрица, является глобальная память устройства. Необходимость пересылки данных между глобальной памятью и мультипроцессорами значительно увеличивает время работы этого алгоритма.

Впервые данную реализацию предложили Harish и Narayanan в 2007 году. Данный подход можно описать следующим псевдокодом:

CopyMatrixToGlobalMemory();

for (size\_t k = 0; k < n; ++k) {

CalculateElementInAllThreads(k);

WaitAllThreads();

}

### 3.4.1 Блочный алгоритм

Реализацию блочного алгоритма Флойда-Уоршелла предложили Katz и Kider в 2008 году [2]. Основная идея данного подхода заключается в использовании разделяемой памяти блоков вместо глобальной памяти устройства. В блочном алгоритме вся матрица расстояний разбивается на тайлы размера *b×b*, и во время очередной итерации алгоритма каждый элемент матрицы обновляется на *b* слоев. Пусть , и . Тогда тайл матрицы расстояний определяется по формуле:

|  |  |
| --- | --- |
|  | (3.7) |

Как было описано в главе разделе 3.3, блочная итерация требует последовательного выполнения 3 шагов:

1. Вычисление элементов главного тайла .
2. Вычисление элементов тайлов из -го столбца и -й строки.
3. Вычисление элементов оставшихся тайлов .

Рассмотрим подробнее, как можно выполнить параллельно каждый из приведенных выше шагов.

В первую очередь, для всех элементов тайла запустим по одному потоку, каждый из которых будет постепенно поднимать свой элемент на *b* слоев вверх. Используя *b*=32, мы можем запустить все необходимые потоки в одном блоке (в 1 блоке CUDA доступно 1024 потока). Кроме того, как было отмечено в пункте 2.2, для обновления своего элемента потоку могут понадобиться только элементы текущего слоя этого же тайла. Поэтому, сохранив слой главного тайла в начале шага в разделяемую память используемого блока и обновляя его там, мы можем уменьшить количество операций с глобальной памятью, которая значительно медленнее разделяемой. Таким образом, каждый из потоков должен скопировать один элемент из глобальной памяти в разделяемую, обновить его на *b* слоев, и вернуть новое значение назад в глобальную память.

Теперь опишем процесс обновления тайлов -й строки и -го столбца. Для каждого такого тайла мы будем использовать по одному блоку GPU, запуская в нем по *b×b* потоков – один поток для одного элемента тайла. Данные тайлы – одиночно-зависимые (зависят только от главного тайла и от самого себя). Поэтому для каждого из запускаемых блоков нам придется хранить в разделяемой памяти две матрицы размером *b*×*b*: -й слой тайла , и текущий слой своего тайла. В этом заключается единственное отличие от предыдущего шага – каждый поток должен скопировать из глобальной памяти не только свой элемент, но и один элемент главного блока.

Похожим образом будем поступать с оставшимися дважды-зависимыми тайлами: для каждого из тайлов запустим по одному блоку, в которых создадим *b*×*b* потоков – один поток будет отвечать за один элемент тайла. В разделяемой памяти -го блока будем хранить 3 матрицы размером *b×b*: -е слои тайлов и , и текущий слой тайла . Аналогично предыдущим шагам, каждый поток будет копировать по 3 элемента из глобальной памяти в разделяемую.

Заметим, что на третьем шаге в разделяемой памяти блоков достаточно хранить только по две матрицы: -е слои тайлов и . Объясняется это тем, что потокам не понадобится обращаться к элементам тайла, за которые отвечают другие потоки – нужен будет только свой элемент и элементы из одиночно-зависимых тайлов. Поэтому вместо того, чтобы хранить текущий слой тайла в разделяемой памяти блока, каждый поток может хранить элемент, за который он отвечает, в своей регистровой памяти. Назовем данную оптимизацию регистровой.

Нетрудно заметить, что данный алгоритм позволяет сократить количество данных, посылаемых через шину глобальной памяти, в 32 раза по сравнению с базовым алгоритмом для GPU.

### 3.4.2 Многостадийный блочный алгоритм

Дальнейшей оптимизацией алгоритма путем уменьшения объема используемой разделяемой памяти в блочном алгоритме Флойда-Уоршелла занялись B. Lund и J. W. Smith в 2010 году [3]. Главная идея их подхода – многостадийное чтение одиночно-зависимых тайлов при вычислении дважды-зависимых.

Все вычисления, производимые при обновлении элементов дважды-зависимого тайла от слоя *l* до слоя *m*, будут зависеть только от определенных подмножеств элементов в одиночно-зависимых тайлах. Так при обновлении дважды-зависимого тайла от слоя *l* до слоя *m*, все вычисления будут зависеть от элементов, находящихся в строках с *l* по *m* одиночно-зависимого тайла , и от элементов, находящихся в столбцах с *l* по *m* одиночно-зависимого тайла . Разобьем процесс исполнения итерации на стадии размером STAGE\_SIZE, разделяемые точкой синхронизации. Перед началом очередной стадии часть потоков блока будет загружать в разделяемую память новую порцию данных, необходимую на этой стадии. Затем, после синхронизации, все потоки смогут выполнить очередные STAGE\_SIZE задач для своих элементов.

Работу, выполняемую потоком во время одной итерации, можно представить следующим образом:

for (size\_t stage = 0; stage < TILE\_SIZE/STAGE\_SIZE; ++stage) {

if (threadIdx.y / STAGE\_SIZE == stage) {

LoadElementsFromSingleDependentToSM(threadIdx.x,

threadIdx.y);

}

WaitAllThreads ()

for (size\_t locIter = 0; locIter < STAGE\_SIZE; ++locIter) {

UpdateElement(stage \* STAGE\_SIZE + locIter);

}

WaitAllThreads();

}

При использовании такого подхода, только 2\*b\*STAGE\_SIZE элементов должны быть загружены в разделяемую память в конкретный момент времени. На рисунке 3.2 представлено использование разделяемой памяти блока во время одной стадии.

|  |
| --- |
| staged.png |
| **Рисунок 3.2 – Использование разделяемой памяти при многостадийном вычислении дважды-зависимых тайлов. Красные зоны – данные, хранящиеся в разделяемой памяти; синие – в регистровой памяти; серые – в глобальной** |

Данный подход не изменяет итоговое количество информации, пересылаемой по шине. Тем не менее, уменьшение размеров разделяемой памяти, необходимой блоку в каждый момент времени, позволяет получить заметный выигрыш в производительности. Причиной этому является тот факт, что разделяемая память делится мультипроцессором поровну между всеми блоками, которые на нем выполняются. Таким образом, уменьшая размер разделяемой памяти, используемой в блоке, мы позволяем мультипроцессору запустить большее количество блоков одновременно.

## 3.5 Выводы

В третьей главе получены следующие результаты:

* сформулирована формальная постановка задачи поиска кратчайших путей в графе;
* описан последовательный алгоритм Флойда-Уоршелла её решения;
* описаны известные параллельные алгоритмы.

На основе рассмотренных в главе 3 алгоритмов, в главе 5 будет новый алгоритм, учитывающий особенности архитектуры, рассмотренной в главе 2, однако перед этим необходимо рассмотреть некоторые теоретические аспекты эффективного использования ресурсов GPU.

# Глава 4. Теоретические аспекты эффективного использования регистров, разделяемой памяти и кэшей GPU

## 4.1 Приватизация потоками элементов массивов

В работе [6] были предложены необходимые условия приватизации потоками элементов массивов. Рассмотрим их подробнее на примере алгоритма Флойда-Уоршелла.

### 4.1.1 Блоки и потоки вычислений

Будем считать, что алгоритм задан последовательной программой линейного класса [5] (в литературе используются также термины «аффинные гнезда циклов», «алгоритмы с аффинными зависимостями».). Основную вычислительную часть такой программы составляют циклические конструкции; границы изменения параметров циклов задаются неоднородными формами, линейными по совокупности параметров циклов и внешних переменных. Разбить множество операций алгоритма на блоки и потоки можно путем преобразования алгоритма для получения макроопераций [6].

Пусть в гнезде циклов имеется Θ наборов выполняемых операторов – совокупностей операторов, окруженных одним и тем же множеством циклов. Обозначим , , – области изменения параметров циклов, окружающих наборы операторов,  – размерность области , число циклов, окружающих -й набор операторов. Обозначим ν*l* – размерности массивов *al*. Обозначим размеры блоков вычислений натуральными числами ;  – число значений параметра *j*ζ, приходящихся на один блок -го набора операторов;  – число блоков -го набора операторов по координате с номером ζ; нумеровать блоки вычислений будем по каждой координате в пределах от 0 до *–*1, . Далее для простоты записи будем использовать обозначения без индекса .

Зададим в блоках вычислений потоки вычислений посредством выделения блоков второго уровня. Зададим размеры потоков натуральными числами *r*1,2, *r*2,2, …, *rnβ*,2; *r*ζ,2 – число значений параметра *j*ζ, приходящихся на один поток; *Q*1,2, *Q*2,2, …, *Qnβ*,2 – число таких потоков; *Q*ζ,2*=*

Вхождением (*l*,β,*q*) будем называть *q*-е вхождение массива *al* в оператор *S*β. Индексы элементов *l* - го массива, связанных с вхождением (*l*,β,*q*), выражаются функцией вида 

Обозначим *=*rank*Fl,*β,*q*, *=*rank, где  – вектор-строка размера *n*β, у которой координата с номером ζ равна 1, а остальные координаты нулевые.

**Пример**. Рассмотрим основную часть алгоритма Флойда-Уоршалла поиска кратчайших путей между всеми парами вершин графа.

do *k =* 1*, n*

do *i =* 1*, n*

do *j =* 1*, n*

*a*(*i*,*j*) *=* min(*a*(*i,j*)*, a*(*i,k*) *+ a*(*k,j*))

enddo

enddo

enddo

В гнезде циклов имеется один выполняемый оператор *S*1 (один набор операторов), и используется один массив *a* размерности 2. Область изменения параметров циклов (область итераций) *V*1*=V*1*=*{(*k,i,j*)Z3| 1≤*k*≤*n*, 1≤*i*≤*n*, 1≤*j*≤*n*} для оператора *S*1 имеет размерность 3. Для матриц *Fl,*β,*q* на вхождениях (*l,*β,*q*) имеем:

*F*1,1,1*=F*1,1,2*=*, *F*1,1,3*=*, *F*1,1,4*=*.

Блочный алгоритм с 3D блоками размера *r×r×r* получен в работе [7]. Блоки имеют следующий вид (*kgl*, *igl*, *jgl* – номера частей, на которые при формировании блоков разбиваются области значений параметров *k, i, j* циклов):

do *k =* 1 + *kglr*, min((*kgl* + 1)*r*, *n*)

do *i =* 1 + *iglr*, min((*igl* + 1)*r*, *n*)

do *j =* 1 + *jglr*, min((*jgl* + 1)*r*, *n*)

*a*(*i*,*j*) *=* min(*a*(*i,j*)*, a*(*i,k*) *+ a*(*k,j*))

enddo

enddo

enddo

Выделим потоки (блоки второго уровня):

do *kgl,2  =* 0*, Q*1,2–1

do *igl,2 =* 0*, Q*2,2–1

do *jgl,2 =* 0*, Q*3,2–1

do *k=*1+*kglr*+*kgl,*2*r*1*,*2, min(*kglr*+(*kgl,2*+1)*r*1*,*2, *n*) // Начало блока второго уровня

do *i =* 1+*iglr*+*igl,*2*r*2*,*2, min(*iglr*+(*igl,2*+1)*r*2*,*2, *n*)

do *j =* 1+*jglr*+*jgl,*2*r*3*,*2, min(*jglr*+(*jgl,2*+1)*r3,*2, *n*)

*a*(*i*,*j*)*=* min(*a*(*i,j*)*, a*(*i,k*)*+a*(*k,j*))

enddo

enddo

enddo // Конец блока второго уровня

enddo

enddo

enddo

### 4.1.2 Условия приватизации элементов массива

Результаты, представленные в этом разделе, определяют условия осуществления приватизации элементов массива, накладываемые на матрицу *Fl,*β,*q*. Рассмотрим сначала случай, когда потоки вычислений помечаются как узлы одномерной сетки*.*

**Теорема 1** [6]**.** Пусть зафиксирован блок вычислений, потоки вычислений помечены как узлы одномерной сеткиThr(), и для вхождения(*l,*β,*q*)массива *al* в оператор *S*βвыполнено условие . Тогда каждый элемент массива *al,* связанный с вхождением(*l*,β,*q*)*,* используется только одним потоком. Если выполнено условие **, то каждый элемент массива используется несколькими потоками.

Действительно, из условия теоремы следует, что при различных значениях параметра *j*ζ используются различные элементы массива *al*, поэтому каждый такой элемент используется только одним потоком. Если **, то на каждой итерации цикла с параметром *j*ζ совокупности используемых элементов массива *al* совпадают, поэтому каждый элемент массива *al* используется более чем одним потоком.

Аналогичное обоснование в общем случае для параллельных компьютеров с распределенной памятью (использование элемента массива только одним процессором) приведено в работе [8].

Рассмотрим теперь наиболее частый для практики случай, когда потоки вычислений помечаются как узлы двумерной сетки.

**Теорема 2** [6]**.** Пусть зафиксирован блок вычислений, потоки вычислений помечены как узлы двумерной сетки Thr(,), , и для вхождения(*l,*β,*q*)массива *al* в оператор *S*βвыполнено условие . Тогда каждый элемент массива *al,* связанный с вхождением(*l*,β,*q*)*,* используется только одним потоком*.* Если выполнено условие ** или **, то каждый элемент массива используется несколькими потоками.

Обоснование теоремы 2 проводится аналогично обоснованию теоремы 1, следует только рассмотреть циклы с параметрами ** и **.

**Пример** (продолжение). Пусть потоки вычислений помечаются как узлы двумерной сеткиThr(*i,j*). На вхождениях (1,1,1) и (1,1,2) одновременно используется один тот же элемент массива. Для вхождения (1,1,1) имеем *=* rank*F*1,1,1*=* rank*=*2, *=*rank*=*2, *=*rank*=*2, выполняется условие  теоремы 2. Для вхождения (1,1,2) точно так же . Таким образом, условия теоремы 2 выполняются, каждый элемент массива *al*, связанный с вхождениями (1,1,1) и (1,1,2), используется только одним потоком. Каждый элемент массива *al*, связанный с вхождениями (1,1,3) или (1,1,4), используется несколькими потоками, так как , , , , , . Аналогичным образом результаты приватизации элементов массива потоками вычислений для потоков, помеченных как узлы двумерных сеток Thr(*i,k*) и Thr(*k,j*), представлены в таблице 1. Второй и третий столбцы указывают вхождения, на которых каждый элемент массива *al* используется только одним потоком и несколькими потоками соответственно (таблица 4.1).

|  |  |  |
| --- | --- | --- |
| Сетка потоков | Используется только одним потоком | Используется несколькими потоками |
| Thr(*i,j*) | *a(i,j)* | *a(i,k)*, *a(k,j)* |
| Thr(*i,k*) | *a(i,k)* | *a(i,j)*, *a(k,j)* |
| Thr(*k,j*) | *a(k,j)* | *a(i,j)*, *a(i,k)* |

**Таблица 4.1 - Приватизация элементов массива потоками вычислений**

## 4.2 Определение максимально возможных блоков вычислений для GPU с заданными характеристиками объема памятей

При разработке для GPU необходимо учитывать физические ограничения графического процессора, на котором будет выполняться программа. Одним из основных параметров, по которым можно судить об эффективности использования ресурсов GPU является количество одновременно выполняемых warp-ов, причем удобнее рассматривать именно отношение реального значения к максимально возможному. Обозначим это отношение как *Occupancy*.

|  |  |
| --- | --- |
| , | (4.1) |

где *activeWarps* – количество одновременно выполняемых warp-ов на одном мультипроцессоре,

*maximumActiveWarps* – максимальное количество одновременно выполняемых warp-ов на одном мультипроцессоре.

Значение *Occupancy* характеризует то, насколько эффективно используются доступные ресурсы графического процессора, при решении той или иной задачи. В теории, значение *Occupancy* порядка 65% уже является достаточным и дальнейшие улучшения необходимы в исключительных случаях. Основными возможными причинами ухудшения *Occupancy* могут быть:

* количество используемых блоком регистров;
* размер используемой блоком разделяемой памяти;
* размер блока (количество потоков);

Пусть некоторый алгоритм использует *ThreadCount* потоков на блок, *RegCount* регистров на поток, *SharedMemory* разделяемой памяти в байтах на блок. В таком случае при оценке *Occupancy* необходимо сравнивать следующие значения.

Для начала введем вспомогательную величину, характеризующую количество warp-ов, необходимых для выполнения *ThreadCount* потоков (*warpsPerBlock*):

|  |  |
| --- | --- |
| , | (4.2) |

где *limitThreadsPerWarp* – максимальное количество потоков в warp-е.

Максимальное количество одновременно обрабатываемых блоков в силу физических ограничений на количество потоков (*limitBlocksDueToWarps*) можно оценить по следующей формуле:

|  |  |
| --- | --- |
| , | (4.3) |

где *limitBlocksPerSM* – максимальное количество блоков, исполняемых на мультипроцессоре;

*limitWarpsPerSM* – максимальное количество warp-ов, обрабатываемых мультипроцессором.

Максимальное количество одновременно обрабатываемых блоков в силу физических ограничений на количество используемых регистров (*limitBlocksDueToRegs*), при условии, что количество используемых регистров одним потоков не превышает максимальное значение для данной GPU, можно оценить по следующей формуле:

|  |  |
| --- | --- |
| , | (4.4) |

где *limitTotalRegisters* – максимальное количество регистров, доступных для использования мультипроцессору.

Максимальное количество одновременно обрабатываемых блоков в силу физических ограничений на количество используемых разделяемой памяти (*limitBlocksDueToSMem*), при условии что размер используемой блоком разделяемой памяти не превышает максимальное значение для данной GPU, можно оценить по следующей формуле:

|  |  |
| --- | --- |
| ), | (4.5) |

где *limitTotalSharedMemory* – максимальное количество разделяемой памяти, доступной мультипроцессору.

В таком случае количество активных блоков на мультипроцессоре (*activeThreadBlocksPerSM*) будет следующим:

|  |  |
| --- | --- |
|  | (4.6) |

А количество активных warp-ов можно рассчитать по следующей формуле:

Таким образом для заданного набора физических ограничений GPU, и в условиях использования определенного количества регистров и разделямой памяти можно подобрать оптимальные размеры блоков вычислений для достижения максимального значения *Occupancy*.

## 4.3 Выводы

В четвертой главе были получены следующие результаты:

* рассмотрены условия приватизации потоками элементов массивов;
* проведена оценка максимально возможных блоков вычислений для GPU с заданными характеристиками объема памятей;

Данные результаты будут использованы при разработке нового параллельного алгоритма Флойда-Уоршелла, который будет рассмотрен в главе 5.

# Глава 5. Новая модификация многостадийного блочного параллельного алгоритма Флойда-Уоршелла

## 5.1 Описание алгоритма

Главной задачей описанного в разделе 3.4.2 многостадийного блочного алгоритма является поиск баланса между количеством разделяемой памяти, используемой каждым потоком и количеством блоков, потоки которых могут одновременно выполняться на мультипроцессоре. Достигается данный баланс путем разбиения процесса обновления дважды-зависимого тайла на стадии, как это было описано ранее.

Однако, данную задачу можно решать иначе. Предлагается, использовать разделяемую память для хранения только одного из одиночно-зависимых тайлов, а второй поместить либо в константную, либо в текстурную память. Каждая из этих типов памяти обладает быстрым read-only кэшем, а значит обращение к ней должно быть достаточно быстрым (более подробное описание каждого из приведенных типов памяти можно найти в разделе 2.5).

Загрузка данных в выбранную память (константную или текстурную) будет отнимать дополнительное время, поэтому предлагается загрузив одиночно-зависимый тайл ведущей строки (столбца), выполнить запуск ядра, за который будут рассчитаны все дважды-зависимые тайлы столбца (строки), которые от него зависят. В таком случае обработка всех дважды-зависимых тайлов на очередной блочной итерации будет происходить не за один запуск ядра, а за запусков.

В таком случае псевдокод блочной модификации алгоритма Флойда-Уоршелла примет следующий вид (в примере в константную память загружается тайл ведущего столбца):

for (size\_t = 0; < n/b; ++) {

UpdateLeadBlock();

UpdateLeadRowBlocks();

UpdateLeadColumnBlocks();

LoadKLeadColumnBlockToConstantMemory();

for (size\_t rowIter = 0; rowIter < n/b; ++rowIter) {

UpdateRowBlocks(, rowIter);

}

}

При этом потоки блока должны будут загружать только один одиночно-зависимый тайл в разделяемую памяти и при пересчет элементов за данным о ведущей строке обращаться в разделяемую память, а за данными о ведущем столбцы в константную или текстурную.

Применяя описанный выше подход, мы с одной стороны в два раза сокращаем использование разделяемой памяти, что позволит запустить большее количество одновременно исполняемых потоков, но с другой появляются накладные расходы на многократный запуск ядра, а также копирование данных в константную (текстурную) память.

## 5.2 Выводы

В пятой главе сформулированы новые модификации многостадийного блочного алгоритма Флойда-Уоршелла, описаны основные преимущества и возможные недостатки. Детали реализации и оценка перспективности приведенных модификаций будут приведены в главе 6.

# Глава 6. Реализация алгоритма и вычислительные эксперименты

## 6.1 Используемые инструменты и технологии

Разработка проиводилась на платформе Windows. В качестве основной среды разработки было решено использовать **Microsoft Visual Studio Ultimate 2013**.

**Microsoft Visual Studio** - линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Данные продукты позволяют разрабатывать как консольные приложения, так и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox, Windows Phone .NET Compact Framework и Silverlight.

**Visual Studio** включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

В качестве платформы для разработки с использованием технологии CUDA компания Nvidia предлагает пакет продуктов под названием **Nvidia Nsight**. Данный пакет предоставляет мощные инструменты для отладки и профилирования, которые позволяют добиться максимальной производительности разрабатываемого приложения.

При реализации алгоритмов в рамках данной магистерской работы, Nvidia Nsight использовался в виде расширения для **Visual Studio**. Данный дистрибутив распространяется под названием **NVIDIA Nsight Visual Studio Edition**.

## 6.2 Сравнение различных реализаций алгоритма Флойда-Уоршелла

Для проведения экспериментов были реализованы:

* последовательный алгоритм Флойда-Уоршелла;
* многостадийный блочный алгоритм Флойда-Уоршелла, описанный в разделе 3.4.2;
* новые модификации многостадийного блочного алгоритма, рассмотренные в главе 5.

В качестве экспериментальных данных использовались случайные графы, заданные матрицами смежности. Для простоты реализации некоторых из описанных ранее параллельных версий использовались только графы с количеством вершин кратным 32.

Все эксперименты проводились на компьютере с видеокартой NVIDIA GeForce 960M. Данная видеокарта обладает графическим процессором со следующими характеристиками:

* CUDA Capatibility Version: 5.0;
* Количество мультипроцессоров: 5;
* Количество ядер в мультипроцессоре: 128;
* Общая память устройства: 4 ГБ;
* Общая константная память: 64 Кб;
* Максимальная размерность текстурной памяти (x, y, z):

1D – (65536), 2D – (65536, 65536), 3D – (4096, 4096, 4096);

* Разделяемая память блока: 48 Кб;
* Количество 32-битных регистров в блоке: 65536;
* Размер **warp**-а: 32 потока;
* Максимальное количество потоков в блоке: 1024;
* Максимальное количество потоков в мультипроцессоре: 2048.

В таблице 6.1 приведены значения времен работы в секундах последовательного алгоритма, многостадийного блочного алгоритма, а также двух модификаций, рассмотренных в главе 5, использующих константную и текстурную память для хранения одного из одиночно-зависимых тайлов. Результаты представленные в таблице 6.1 проиллюстрированы на рисунке 6.1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ***N*** | Последовательный алгоритм  (, сек.) | Многостадийный блочный алгоритм (, сек.) | Многостадийный блочный алгоритм (константная) (, сек.) | Многостадийный блочный алгоритм (текстурная) (, сек.) |
| 512 | 0.1875 | 0.0014 | 0.0781 | 0.0625 |
| 1024 | 1.4683 | 0.0404 | 0.2813 | 0.2969 |
| 2048 | 11.8649 | 0.1936 | 1.2173 | 1.2601 |
| 4096 | 89.7875 | 1.2024 | 5.5524 | 6.1771 |
| 8192 | … | 9.7435 | 26.4995 | 31.4351 |

**Таблица 6.1 – Сравнение времен работы реализованных алгоритмов**

**Рисунок 6.1 – Сравнение времен работы реализованных алгоритмов:**

График (1) – последовательная версия, (2) – многостадийный блочный алгоритм, (3) – модификация с константной памятью, (4) – модификация с текстурной памятью

Видно, что разработанные модификации хоть и работают быстрее по сравнению с последовательным алгоритмом, однако проигрывают многостадийному блочному алгоритму.

После подробного исследования реализации было выявлено три возможные причины такого ухудшения:

1. На каждой блочной итерации происходит запусков ядра, что влечет за собой дополнительные накладные расходы по сравнению с многостадийной блочной версией;
2. Каждому из запусков ядра предшествует загрузка одиночно-зависимого тайла из глобальной памяти в константную или текстурную;
3. Операции чтения из константной и текстурной памяти занимают намного дольше времени чем из разделяемой, несмотря на наличие быстрого кэша.

Для проверки первой причины реализация многостадийного блочного алгоритма была преобразована так, чтобы за один вызов ядра рассчитывалась только одна строка дважды-зависимых тайлов, как это происходит в предложенной модификации, однако с использованием разделяемой памяти для обоих одиночно-зависимых тайлов. Результаты экспериментов приведены в таблице 6.2.

|  |  |  |
| --- | --- | --- |
| **N** | Многостадийный блочный алгоритм  (, сек.) | Многостадийный блочный алгоритм  (n/b запусков) (, сек.) |
| 512 | 0.0014 | 0.0156 |
| 1024 | 0.0404 | 0.0625 |
| 2048 | 0.1936 | 0.3281 |
| 4096 | 1.2024 | 1.7372 |
| 8192 | 9.7435 | 12.1324 |

**Таблица 6.2 – Сравнение времен работы многостайдиного блочного алгоритма**

Из таблицы 6.2 видно, что многократный перезапуск ядра CUDA действительно влечет за собой некоторые накладные расходы. Однако, сравнив таблицы 6.1 и 6.2 видно, что это не является основной проблемой полученного алгоритма, поэтому целесообразно провести дальнейшие исследования.

Для проверки второй из приведенных выше возможных проблем ухудшения результатов отдельно был произведен замер времени, которое тратится непосредственно на загрузку данных в константную или текстурную память. Результаты приведены в таблицах 6.3, 6.4.

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | Многостадийный блочный алгоритм (константная) (, сек.) | Загрузка данных в константную память (, сек.) |  |
| 512 | 0.0781 | 0.0625 | 0.0156 |
| 1024 | 0.2813 | 0.1407 | 0.1408 |
| 2048 | 1.2173 | 0.6882 | 0.5297 |
| 4096 | 5.5524 | 2.5155 | 3.0369 |
| 8192 | 26.4995 | 10.374 | 16.1255 |

**Таблица 6.3 – Оценка времени потраченного на загрузку данных в константную память**

|  |  |  |  |
| --- | --- | --- | --- |
| **N** | Многостадийный блочный алгоритм (текстурная) (, сек.) | Загрузка данных в текстурную память (, сек.) |  |
| 512 | 0.0625 | 0.0469 | 0.0156 |
| 1024 | 0.2969 | 0.2031 | 0.0938 |
| 2048 | 1.2601 | 0.6940 | 0.5661 |
| 4096 | 6.1771 | 2.7435 | 3.4336 |
| 8192 | 31.4351 | 9.8112 | 21.6239 |

**Таблица 6.4 – Оценка времени потраченного на загрузку данных в текстурную память**

Исходя из проведенных замеров видно, что значительная часть времени работы уходит именно на загрузку данных в выбранную память (константную или текстурную). Объяснить это можно следующим образом: граф хранится в глобальной памяти в виде матрицы смежности, соответственно для того чтобы переписать один тайл выполняется b обращений к памяти, где b – размер тайла, то есть на каждой блочной итерации происходит n обращений к глобальной памяти графического процессора, что и приводит к описанным выше ухудшениям.

Практическая проверка третьей гипотезы невозможна, однако исходя из проведенных ранее экспериментов, можно говорить, что она отчасти верна. На графике, изображенном на рисунке 6.2, представлено сравнение времен работы алгоритмов, в котором для предложенных модификаций из общего времени работы (таблица 6.1, и соответственно) вычтено время, затраченное на многократный запуск ядра (таблица 6.2, ) и время, затраченное на загрузку данных в константную (текстурную) память (таблица 6.3, и таблица 6.4, ). Видно, что даже без учета времени, затраченного на эти дополнительные операции, предложенные модификации работают несколько медленнее.

**Рисунок 6.2 – Сравнение времен работы реализованных алгоритмов:**

График (1) – последовательная версия, (2) – многостадийный блочный алгоритм, (3) – модификация с константной памятью без учета доп. расходов, (4) – модификация с текстурной памятью без учета доп. расходов

Чтобы добиться ускорения алгоритма необходимо учитывать особенности использования константной и текстурной памяти: констатная будет давать наилучший результат, если все потоки будут обращаться к одному и тому же элементу используемого массива, в то время как текстурная может быть более эффективно использована в случае хорошей локальности данных. В следующем пункте будут приведены усовершенствования алгоритма, которые могли бы учесть описанные особенности.

## 6.3 Возможные направления дальнейших исследований

Исходя из обнаруженных выше недостатков текущей реализации алгоритма, можно предложить несколько направлений дальнейших исследований.

Так как потенциально текстурная память может иметь довольно большой размер, то вместо загрузки одного одиночно-зависимого тайла, в текстурную память можно загружать целую строку одиночно-зависимых тайлов текущей блочной итерации. После чего всё также выполнять запусков ядра. Таким образом мы с одной стороны, избавимся от значительной части накладных расходов на загрузку данных в память (так как это будет непрерывная часть данныз глобальной памяти), а с другой - всё также позволим потокам использовать меньше разделяемой памяти по сравнению со стандартным многостадийным блочным алгоритмом.

Другим направлением исследований может быть использование нестандартного формата хранения матрицы смежности в памяти. Например, в статье [3] авторы предлагают хранить матрицы в row-major стиле – так, что любой тайл b x b находится в матрице последовательно, что позволяет считывать его быстрее, чем в разработанном в данной работе алгоритме. Кроме того, путём некоторых модификаций, более подробного рассмотренных в этой же статье, можно добиться преимуществ, необходимых для эффективного использования константной и текстурной памяти.

## 6.4 Выводы

В главе 6 были достигнуты следующие результаты:

* описаны использованные при разработке инструменты и технологии;
* проведено сравнение различных реализованных алгоритмов, выявлены сильные и слабые стороны предложенной модификации параллельного алгоритма Флойда-Уоршелла;
* описаны возможные пути дальнейших исследований.

# ЗАКЛЮЧЕНИЕ

В процессе выполнения дипломной работы, поставленные задачи были решены, цель достигнута.

В первой главе был проведен анализ литературы, поставлена задача, сформулирован подход к решению.

Во второй главе была рассмотрена вычислительная архитектура CUDA: особенности реализации ветвлений в коде ядра, межпроцессного взаимодействия, организации памяти.

В третьей главе был описан последовательный алгоритм Флойда-Уоршелла, его блочная модификация, а также известные параллельные алгоритмы: блочный и многостадийный блочный.

В четвертой главе были рассмотрены теоретические аспекты эффективного использования регистрой, разделяемой памяти и кэшей GPU, приведен алгоритм определения максимально возможных блоков вычислений для GPU с заданными характеристиками объема памятей.

В пятой главе на основе изученных параллельных алгоритмов Флойда-Уоршелла была разработана новая модификация многостадийного блочного параллельного алгоритма Флойда-Уоршелла.

В шестой главе приведены результаты вычислительных экспериментов для разработанных алгоритмов, проведено сравнение результатов, описаны возможные направления дальнейших исследований.

Таким образом, основными результатами данной работы стали разработка новых модификаций параллельного алгоритма Флойда-Уоршелла, основанных на использовании константной (текстурной) памяти вместо раделяемой, их реализация и сравнение с существующими параллельными алгоритмами.

# БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. P. Harish, Accelerating large graph algorithms on the GPU uing CUDA / P. Harish, P. Narayanan // Lecture Notes in Computer Science. – 2007 – p. 197.
2. G. J. Katz, All-pairs shortest-paths for large graphs on the GPU / G. J. Katz, J. T. Kider Jr. // Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware – 2008 – p. 47–55.
3. B. Lund, J. W. Smith, A Multi-Stage CUDA Kernel for Floyd-Warshall - 2010.
4. Programming Guide :: CUDA Toolkit Documentation [Электрон. ресурс] / Programming Guide : CUDA Toolkit Documentation. – Режим доступа: https://docs.nvidia.com/cuda/cuda-c-programming-guide/ – Дата доступа: 05.11.2016.
5. В. В. Воеводин, Параллельные вычисления / В.В. Воеводин. СПб. БХВ-Петербург – 2002 – 600 с.
6. Полещук М.А., Лиходед Н.А. Приватизация элементов массивов потоками вычислений // Международный конгресс по информатике: информационные системы и технологии CSIST'2016, 24–27 октября 2016 г., Минск, Беларусь. Бел. гос. ун-т. – Минск, 2016. С. 883–888.
7. В. Фролов, Введение в технологию CUDA / Владимир Фролов // Сетевой журнал "Компьютерная графика и мультимедиа". Выпуск №6(1) / 2008 [Электронный ресурс]. – Режим доступа: http://cgm.computergraphics.ru/issues/issue16/cuda – Дата доступа: 01.02.2017
8. А. А. Букатов, В. Н. Дацюк, А. И. Жегуло, Программирование многопроцессорных вычислительных систем – Ростов на Дону: ООО “ЦВВР” – 2003 – 208 с.
9. C. Severance, High Performance Computing (RISC Architectures, Optimization & Benchmarks) // Charles Severance, Kevin Dowd – O’Reilly Media – 1998 – 466 p.

# ПРИЛОЖЕНИЯ

## ПРИЛОЖЕНИЕ А

### Модификация многостадийного блочного алгоритма с использованием константной памяти

#include <chrono>

#include <cuda.h>

#include <fstream>

#include <iostream>

#include <algorithm>

#include <iostream>

#include <stdint.h>

#include <stdio.h>

#include <cuda\_runtime.h>

#include <device\_functions.h>

#include <device\_launch\_parameters.h>

#define TILE\_SIZE 32

#define BASE\_STAGE\_SIZE\_STEP2 8

#define STAGE\_SIZE\_STEP2 BASE\_STAGE\_SIZE\_STEP2

#define STAGE\_COUNT\_STEP2 TILE\_SIZE / STAGE\_SIZE\_STEP2

#define BASE\_STAGE\_SIZE\_STEP3 32

#define STAGE\_SIZE\_STEP3 BASE\_STAGE\_SIZE\_STEP3

#define STAGE\_COUNT\_STEP3 TILE\_SIZE / STAGE\_SIZE\_STEP3

#define HANDLE\_ERROR(status) \

{ \

if (status != cudaSuccess) \

{ \

printf("%s failed at line %d \nError message: %s \n", \

\_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(status)); \

exit(EXIT\_FAILURE); \

} \

}

\_\_global\_\_ void WakeGpuKernel(int reps)

{

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx >= reps) return;

}

\_\_global\_\_ void CalculateLeadBlock(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter)

{

const int locI = threadIdx.y;

const int locJ = threadIdx.x;

const int glI = TILE\_SIZE \* blockedIter + locI;

const int glJ = TILE\_SIZE \* blockedIter + locJ;

if (glI >= n || glJ >= n ||

glI >= TILE\_SIZE \* (blockedIter + 1) || glI < TILE\_SIZE \* blockedIter ||

glJ >= TILE\_SIZE \* (blockedIter + 1) || glJ < TILE\_SIZE \* blockedIter)

{

return;

}

\_\_shared\_\_ uint32\_t leadBlock[TILE\_SIZE \* TILE\_SIZE];

leadBlock[locI \* TILE\_SIZE + locJ] = graph[glI \* n + glJ];

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < TILE\_SIZE; ++locIter) {

uint32\_t newPathLen = leadBlock[locI \* TILE\_SIZE + locIter]

+ leadBlock[locIter \* TILE\_SIZE + locJ];

if (newPathLen < leadBlock[locI \* TILE\_SIZE + locJ]) {

leadBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

graph[glI \* n + glJ] = leadBlock[locI \* TILE\_SIZE + locJ];

}

\_\_global\_\_ void CalculateLeadRowAndColumn(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter)

{

if (threadIdx.y \* TILE\_SIZE + threadIdx.x > TILE\_SIZE \* TILE\_SIZE

|| blockIdx.x == blockedIter)

{

return;

}

int blockPosI, blockPosJ;

if (blockIdx.y == 0) {

// This is lead row

blockPosI = blockedIter \* TILE\_SIZE;

blockPosJ = blockIdx.x \* TILE\_SIZE;

}

else {

// This is lead column

blockPosI = blockIdx.x \* TILE\_SIZE;

blockPosJ = blockedIter \* TILE\_SIZE;

}

int locI = threadIdx.y;

int locJ = threadIdx.x;

int glI = blockPosI + threadIdx.y;

int glJ = blockPosJ + threadIdx.x;

\_\_shared\_\_ uint32\_t leadBlock[TILE\_SIZE \* STAGE\_SIZE\_STEP2];

\_\_shared\_\_ uint32\_t curBlock[TILE\_SIZE \* TILE\_SIZE];

curBlock[locI \* TILE\_SIZE + locJ] = graph[glI \* n + glJ];

size\_t leadBlockOffset = blockedIter \* TILE\_SIZE;

if (blockIdx.y == 0) {

// This is lead row

#pragma unroll

for (size\_t stage = 0; stage < TILE\_SIZE / STAGE\_SIZE\_STEP2; ++stage) {

if (locI / STAGE\_SIZE\_STEP2 == stage) {

leadBlock[locJ \* STAGE\_SIZE\_STEP2 + (locI % STAGE\_SIZE\_STEP2)] =

graph[(leadBlockOffset + locJ) \* n + leadBlockOffset + locI];

}

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < STAGE\_SIZE\_STEP2; ++locIter) {

uint32\_t newPathLen = curBlock[(stage \* STAGE\_SIZE\_STEP2 + locIter) \* TILE\_SIZE + locJ]

+ leadBlock[locI \* STAGE\_SIZE\_STEP2 + locIter];

if (newPathLen < curBlock[locI \* TILE\_SIZE + locJ]) {

curBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

}

}

else {

// This is lead column

#pragma unroll

for (size\_t stage = 0; stage < TILE\_SIZE / STAGE\_SIZE\_STEP2; ++stage) {

if (locI / STAGE\_SIZE\_STEP2 == stage) {

leadBlock[(locI % STAGE\_SIZE\_STEP2) \* TILE\_SIZE + locJ] =

graph[(leadBlockOffset + locI) \* n + leadBlockOffset + locJ];

}

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < STAGE\_SIZE\_STEP2; ++locIter) {

uint32\_t newPathLen = curBlock[locI \* TILE\_SIZE + stage \* STAGE\_SIZE\_STEP2 + locIter]

+ leadBlock[locIter \* TILE\_SIZE + locJ];

if (newPathLen < curBlock[locI \* TILE\_SIZE + locJ]) {

curBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

}

}

graph[glI \* n + glJ] = curBlock[locI \* TILE\_SIZE + locJ];

}

\_\_constant\_\_ uint32\_t c\_leadCol[TILE\_SIZE \* TILE\_SIZE];

\_\_global\_\_ void CalculateRestBlocks(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter, uint32\_t rowIter)

{

\_\_shared\_\_ uint32\_t leadRow[TILE\_SIZE \* STAGE\_SIZE\_STEP3];

uint32\_t curBlockElem;

if (blockIdx.x == blockedIter)

{

return;

}

int blockPosY = rowIter \* TILE\_SIZE;

int blockPosX = blockIdx.x \* TILE\_SIZE;

#pragma unroll

for (int stage = 0; stage < STAGE\_COUNT\_STEP3; ++stage) {

size\_t leadBlocksOffset = blockedIter \* TILE\_SIZE;

int locY = threadIdx.y;

int locX = threadIdx.x;

if (threadIdx.y / STAGE\_SIZE\_STEP3 == stage) {

leadRow[(threadIdx.y % STAGE\_SIZE\_STEP3) \* TILE\_SIZE + locX] =

graph[(leadBlocksOffset + threadIdx.y) \* n + (blockPosX + locX)];

}

\_\_syncthreads();

int gl = (blockPosY + locY) \* n + blockPosX + locX;

curBlockElem = graph[gl];

#pragma unroll

for (int locIter = 0; locIter < STAGE\_SIZE\_STEP3; ++locIter) {

uint32\_t newPathLen = c\_leadCol[locY \* TILE\_SIZE + stage \* STAGE\_SIZE\_STEP3 + locIter]

+ leadRow[locIter \* TILE\_SIZE + locX];

if (newPathLen < curBlockElem) {

curBlockElem = newPathLen;

}

}

graph[gl] = curBlockElem;

\_\_syncthreads();

}

}

\_\_global\_\_ void PrintGraph(uint32\_t \*graph, uint32\_t n) {

printf("Graph\n");

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

printf("%d ", graph[i \* n + j]);

}

printf("\n");

}

}

\_\_global\_\_ void PrintConstant() {

printf("Constant\n");

for (int i = 0; i < TILE\_SIZE; ++i) {

for (int j = 0; j < TILE\_SIZE; ++j) {

printf("%d ", c\_leadCol[i \* TILE\_SIZE + j]);

}

printf("\n");

}

}

\_\_host\_\_ void FloydBlocked(uint32\_t \*h\_graph,

uint32\_t \*h\_floydResult,

uint32\_t n)

{

// Copy graph to device global memory

std::chrono::steady\_clock::time\_point start = std::chrono::steady\_clock::now();

std::chrono::duration<double> totalTimeSpentOnCopy;

cudaError\_t cudaStatus;

uint32\_t \*d\_graph;

cudaMalloc(&d\_graph, sizeof(uint32\_t)\* n \* n);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaMemcpy(d\_graph, h\_graph, sizeof(uint32\_t)\* n \* n, cudaMemcpyHostToDevice);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

dim3 firstStepGridSize(1, 1, 1);

dim3 firstStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // no EXTRACOUNTX

dim3 secondStepGridSize((n - 1) / TILE\_SIZE + 1, 2, 1);

dim3 secondStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // no EXTRACOUNTX, EXTRACOUNTY

dim3 thirdStepGridSize((n - 1) / TILE\_SIZE + 1, 1, 1);

dim3 thirdStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // TILE\_SIZE \* TILE\_SIZE threads

cudaEvent\_t stepFinishedEvent;

cudaEventCreate(&stepFinishedEvent);

for (uint32\_t blockedIteration = 0; blockedIteration < n / TILE\_SIZE; ++blockedIteration) {

CalculateLeadBlock << <firstStepGridSize, firstStepBlockSize >> >

(d\_graph, n, blockedIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

CalculateLeadRowAndColumn << <secondStepGridSize, secondStepBlockSize >> >

(d\_graph, n, blockedIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

for (uint32\_t rowIteration = 0; rowIteration < n / TILE\_SIZE; ++rowIteration) {

if (rowIteration == blockedIteration)

continue;

std::chrono::steady\_clock::time\_point startCopy = std::chrono::steady\_clock::now();

for (int threadRowIteration = 0; threadRowIteration < TILE\_SIZE; ++threadRowIteration) {

cudaMemcpyToSymbol(

c\_leadCol,

d\_graph + (rowIteration \* n \* TILE\_SIZE + threadRowIteration \* n + blockedIteration \* TILE\_SIZE),

sizeof(uint32\_t)\* TILE\_SIZE,

sizeof(uint32\_t)\* TILE\_SIZE \* threadRowIteration,

cudaMemcpyDeviceToDevice);

}

totalTimeSpentOnCopy += std::chrono::steady\_clock::now() - startCopy;

CalculateRestBlocks << <thirdStepGridSize, thirdStepBlockSize >> >

(d\_graph, n, blockedIteration, rowIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

}

}

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

// Copy results to host

cudaMemcpy(h\_floydResult, d\_graph, sizeof(int)\* n \* n, cudaMemcpyDeviceToHost);

// Calculate all time used by cuda, and print it to console

/\*std::chrono::steady\_clock::duration duration = std::chrono::duration\_cast<std::chrono::milliseconds>

(std::chrono::steady\_clock::now() - start);\*/

std::chrono::duration<double> duration = std::chrono::steady\_clock::now() - start;

std::cout << n << ";" << duration.count() << ";" << totalTimeSpentOnCopy.count() << std::endl;

cudaFree(d\_graph);

}

\_\_host\_\_ int main(int argc, char \*\*argv) {

if (argc < 3) {

std::cout << "usage: " << argv[0] << " graph\_path results\_path" << std::endl;

return 1;

}

// Read vertex count and all graph

uint32\_t n;

//std::fstream graph\_reader(argv[1], std::fstream::in | std::fstream::binary);

FILE\* fin = fopen(argv[1], "r");

fscanf(fin, "%d", &n);

//graph\_reader.read((char\*)&n, 4);

if (n % TILE\_SIZE != 0) {

std::cout << "Number of vertex shoud be divided by tile size (just for easier implementation). "

<< "Tile size: " << TILE\_SIZE << ". Vertex's count: " << n << "."

<< std::endl;

//graph\_reader.close();

return 1;

}

uint32\_t \*h\_graph = new uint32\_t[n \* n];

uint32\_t \*h\_floydResult = new uint32\_t[n \* n];

for (size\_t i = 0; i < n \* n; ++i) {

uint32\_t current\_elem = 0;

fscanf(fin, "%d", &current\_elem);

//graph\_reader.read((char \*)&current\_elem, 1);

h\_graph[i] = current\_elem;

}

//graph\_reader.close();

// Run empty task on cuda - it will decrease time of first run

int threadNum = std::min(n, uint32\_t(32));

dim3 gridSize(n / threadNum, n / threadNum, 1);

dim3 cudaBlockSize(threadNum, threadNum, 1);

WakeGpuKernel << <1, cudaBlockSize >> >(32);

// Blocked Floyd-Warshall algorithm on cuda

FloydBlocked(h\_graph, h\_floydResult, n);

// Write Floyd results to file

std::fstream result\_writer(argv[2], std::fstream::out | std::fstream::binary);

for (size\_t i = 0; i < n \* n; ++i) {

result\_writer.write((char\*)&h\_floydResult[i], 4);

}

result\_writer.close();

delete[] h\_graph;

delete[] h\_floydResult;

return 0;

}

## ПРИЛОЖЕНИЕ Б

### Модификация многостадийного блочного алгоритма с использованием текстурной памяти

#include <chrono>

#include <cuda.h>

#include <fstream>

#include <iostream>

#include <algorithm>

#include <iostream>

#include <stdint.h>

#include <stdio.h>

#include <cuda\_runtime.h>

#include <device\_functions.h>

#include <device\_launch\_parameters.h>

#define TILE\_SIZE 32

#define BASE\_STAGE\_SIZE\_STEP2 8

#define STAGE\_SIZE\_STEP2 BASE\_STAGE\_SIZE\_STEP2

#define STAGE\_COUNT\_STEP2 TILE\_SIZE / STAGE\_SIZE\_STEP2

#define BASE\_STAGE\_SIZE\_STEP3 32

#define STAGE\_SIZE\_STEP3 BASE\_STAGE\_SIZE\_STEP3

#define STAGE\_COUNT\_STEP3 TILE\_SIZE / STAGE\_SIZE\_STEP3

#define HANDLE\_ERROR(status) \

{ \

if (status != cudaSuccess) \

{ \

printf("%s failed at line %d \nError message: %s \n", \

\_\_FILE\_\_, \_\_LINE\_\_, cudaGetErrorString(status)); \

exit(EXIT\_FAILURE); \

} \

}

\_\_global\_\_ void WakeGpuKernel(int reps)

{

int idx = blockIdx.x \* blockDim.x + threadIdx.x;

if (idx >= reps) return;

}

\_\_global\_\_ void CalculateLeadBlock(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter)

{

const int locI = threadIdx.y;

const int locJ = threadIdx.x;

const int glI = TILE\_SIZE \* blockedIter + locI;

const int glJ = TILE\_SIZE \* blockedIter + locJ;

if (glI >= n || glJ >= n ||

glI >= TILE\_SIZE \* (blockedIter + 1) || glI < TILE\_SIZE \* blockedIter ||

glJ >= TILE\_SIZE \* (blockedIter + 1) || glJ < TILE\_SIZE \* blockedIter)

{

return;

}

\_\_shared\_\_ uint32\_t leadBlock[TILE\_SIZE \* TILE\_SIZE];

leadBlock[locI \* TILE\_SIZE + locJ] = graph[glI \* n + glJ];

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < TILE\_SIZE; ++locIter) {

uint32\_t newPathLen = leadBlock[locI \* TILE\_SIZE + locIter]

+ leadBlock[locIter \* TILE\_SIZE + locJ];

if (newPathLen < leadBlock[locI \* TILE\_SIZE + locJ]) {

leadBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

graph[glI \* n + glJ] = leadBlock[locI \* TILE\_SIZE + locJ];

}

\_\_global\_\_ void CalculateLeadRowAndColumn(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter)

{

if (threadIdx.y \* TILE\_SIZE + threadIdx.x > TILE\_SIZE \* TILE\_SIZE

|| blockIdx.x == blockedIter)

{

return;

}

int blockPosI, blockPosJ;

if (blockIdx.y == 0) {

// This is lead row

blockPosI = blockedIter \* TILE\_SIZE;

blockPosJ = blockIdx.x \* TILE\_SIZE;

}

else {

// This is lead column

blockPosI = blockIdx.x \* TILE\_SIZE;

blockPosJ = blockedIter \* TILE\_SIZE;

}

int locI = threadIdx.y;

int locJ = threadIdx.x;

int glI = blockPosI + threadIdx.y;

int glJ = blockPosJ + threadIdx.x;

\_\_shared\_\_ uint32\_t leadBlock[TILE\_SIZE \* STAGE\_SIZE\_STEP2];

\_\_shared\_\_ uint32\_t curBlock[TILE\_SIZE \* TILE\_SIZE];

curBlock[locI \* TILE\_SIZE + locJ] = graph[glI \* n + glJ];

size\_t leadBlockOffset = blockedIter \* TILE\_SIZE;

if (blockIdx.y == 0) {

// This is lead row

#pragma unroll

for (size\_t stage = 0; stage < TILE\_SIZE / STAGE\_SIZE\_STEP2; ++stage) {

if (locI / STAGE\_SIZE\_STEP2 == stage) {

leadBlock[locJ \* STAGE\_SIZE\_STEP2 + (locI % STAGE\_SIZE\_STEP2)] =

graph[(leadBlockOffset + locJ) \* n + leadBlockOffset + locI];

}

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < STAGE\_SIZE\_STEP2; ++locIter) {

uint32\_t newPathLen = curBlock[(stage \* STAGE\_SIZE\_STEP2 + locIter) \* TILE\_SIZE + locJ]

+ leadBlock[locI \* STAGE\_SIZE\_STEP2 + locIter];

if (newPathLen < curBlock[locI \* TILE\_SIZE + locJ]) {

curBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

}

}

else {

// This is lead column

#pragma unroll

for (size\_t stage = 0; stage < TILE\_SIZE / STAGE\_SIZE\_STEP2; ++stage) {

if (locI / STAGE\_SIZE\_STEP2 == stage) {

leadBlock[(locI % STAGE\_SIZE\_STEP2) \* TILE\_SIZE + locJ] =

graph[(leadBlockOffset + locI) \* n + leadBlockOffset + locJ];

}

\_\_syncthreads();

#pragma unroll

for (size\_t locIter = 0; locIter < STAGE\_SIZE\_STEP2; ++locIter) {

uint32\_t newPathLen = curBlock[locI \* TILE\_SIZE + stage \* STAGE\_SIZE\_STEP2 + locIter]

+ leadBlock[locIter \* TILE\_SIZE + locJ];

if (newPathLen < curBlock[locI \* TILE\_SIZE + locJ]) {

curBlock[locI \* TILE\_SIZE + locJ] = newPathLen;

}

\_\_syncthreads();

}

}

}

graph[glI \* n + glJ] = curBlock[locI \* TILE\_SIZE + locJ];

}

texture<uint32\_t, 1, cudaReadModeElementType> texLeadCol;

\_\_global\_\_ void CalculateRestBlocks(uint32\_t \*graph, uint32\_t n,

uint32\_t blockedIter, uint32\_t rowIter)

{

\_\_shared\_\_ uint32\_t leadRow[TILE\_SIZE \* STAGE\_SIZE\_STEP3];

uint32\_t curBlockElem;

if (blockIdx.x == blockedIter)

{

return;

}

int blockPosY = rowIter \* TILE\_SIZE;

int blockPosX = blockIdx.x \* TILE\_SIZE;

#pragma unroll

for (int stage = 0; stage < STAGE\_COUNT\_STEP3; ++stage) {

size\_t leadBlocksOffset = blockedIter \* TILE\_SIZE;

int locY = threadIdx.y;

int locX = threadIdx.x;

if (threadIdx.y / STAGE\_SIZE\_STEP3 == stage) {

leadRow[(threadIdx.y % STAGE\_SIZE\_STEP3) \* TILE\_SIZE + locX] =

graph[(leadBlocksOffset + threadIdx.y) \* n + (blockPosX + locX)];

}

\_\_syncthreads();

int gl = (blockPosY + locY) \* n + blockPosX + locX;

curBlockElem = graph[gl];

#pragma unroll

for (int locIter = 0; locIter < STAGE\_SIZE\_STEP3; ++locIter) {

uint32\_t newPathLen = tex1Dfetch(texLeadCol, locY \* TILE\_SIZE + stage \* STAGE\_SIZE\_STEP3 + locIter)

//c\_leadCol[locY \* TILE\_SIZE + stage \* STAGE\_SIZE\_STEP3 + locIter]

+ leadRow[locIter \* TILE\_SIZE + locX];

if (newPathLen < curBlockElem) {

curBlockElem = newPathLen;

}

}

graph[gl] = curBlockElem;

\_\_syncthreads();

}

}

\_\_global\_\_ void PrintGraph(uint32\_t \*graph, uint32\_t n) {

printf("Graph\n");

for (int i = 0; i < n; ++i) {

for (int j = 0; j < n; ++j) {

printf("%d ", graph[i \* n + j]);

}

printf("\n");

}

}

\_\_host\_\_ void FloydBlocked(uint32\_t \*h\_graph,

uint32\_t \*h\_floydResult,

uint32\_t n)

{

// Copy graph to device global memory

std::chrono::steady\_clock::time\_point start = std::chrono::steady\_clock::now();

std::chrono::duration<double> totalTimeSpentOnCopy;

cudaError\_t cudaStatus;

uint32\_t \*d\_graph;

cudaMalloc(&d\_graph, sizeof(uint32\_t)\* n \* n);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaMemcpy(d\_graph, h\_graph, sizeof(uint32\_t)\* n \* n, cudaMemcpyHostToDevice);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

// declare and allocate memory

uint32\_t \*d\_leadColBuffer;

cudaMalloc(&d\_leadColBuffer, TILE\_SIZE \* TILE\_SIZE \* sizeof(uint32\_t));

size\_t offset;

cudaBindTexture(&offset, texLeadCol, d\_leadColBuffer, cudaCreateChannelDesc(32, 0, 0, 0, cudaChannelFormatKindUnsigned), TILE\_SIZE \* TILE\_SIZE \* sizeof(uint32\_t));

dim3 firstStepGridSize(1, 1, 1);

dim3 firstStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // no EXTRACOUNTX

dim3 secondStepGridSize((n - 1) / TILE\_SIZE + 1, 2, 1);

dim3 secondStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // no EXTRACOUNTX, EXTRACOUNTY

dim3 thirdStepGridSize((n - 1) / TILE\_SIZE + 1, 1, 1);

dim3 thirdStepBlockSize(TILE\_SIZE, TILE\_SIZE, 1); // TILE\_SIZE \* TILE\_SIZE threads

cudaEvent\_t stepFinishedEvent;

cudaEventCreate(&stepFinishedEvent);

for (uint32\_t blockedIteration = 0; blockedIteration < n / TILE\_SIZE; ++blockedIteration) {

CalculateLeadBlock << <firstStepGridSize, firstStepBlockSize >> >

(d\_graph, n, blockedIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

CalculateLeadRowAndColumn << <secondStepGridSize, secondStepBlockSize >> >

(d\_graph, n, blockedIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

for (uint32\_t rowIteration = 0; rowIteration < n / TILE\_SIZE; ++rowIteration) {

if (rowIteration == blockedIteration)

continue;

std::chrono::steady\_clock::time\_point startCopy = std::chrono::steady\_clock::now();

for (int threadRowIteration = 0; threadRowIteration < TILE\_SIZE; ++threadRowIteration) {

cudaMemcpy(

d\_leadColBuffer + TILE\_SIZE \* threadRowIteration,

d\_graph + (rowIteration \* n \* TILE\_SIZE + threadRowIteration \* n + blockedIteration \* TILE\_SIZE),

sizeof(uint32\_t) \* TILE\_SIZE,

cudaMemcpyDeviceToDevice);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

}

totalTimeSpentOnCopy += std::chrono::steady\_clock::now() - startCopy;

CalculateRestBlocks << <thirdStepGridSize, thirdStepBlockSize >> >

(d\_graph, n, blockedIteration, rowIteration);

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

cudaEventRecord(stepFinishedEvent);

cudaEventSynchronize(stepFinishedEvent);

}

}

cudaStatus = cudaGetLastError();

HANDLE\_ERROR(cudaStatus);

// Copy results to host

cudaMemcpy(h\_floydResult, d\_graph, sizeof(int)\* n \* n, cudaMemcpyDeviceToHost);

// Calculate all time used by cuda, and print it to console

/\*std::chrono::steady\_clock::duration duration = std::chrono::duration\_cast<std::chrono::milliseconds>

(std::chrono::steady\_clock::now() - start);\*/

std::chrono::duration<double> duration = std::chrono::steady\_clock::now() - start;

std::cout << n << ";" << duration.count() << ";" << totalTimeSpentOnCopy.count() << std::endl;

cudaUnbindTexture(texLeadCol);

cudaFree(d\_graph);

}

\_\_host\_\_ int main(int argc, char \*\*argv) {

if (argc < 3) {

std::cout << "usage: " << argv[0] << " graph\_path results\_path" << std::endl;

return 1;

}

// Read vertex count and all graph

uint32\_t n;

//std::fstream graph\_reader(argv[1], std::fstream::in | std::fstream::binary);

FILE\* fin = fopen(argv[1], "r");

fscanf(fin, "%d", &n);

//graph\_reader.read((char\*)&n, 4);

if (n % TILE\_SIZE != 0) {

std::cout << "Number of vertex shoud be divided by tile size (just for easier implementation). "

<< "Tile size: " << TILE\_SIZE << ". Vertex's count: " << n << "."

<< std::endl;

//graph\_reader.close();

return 1;

}

uint32\_t \*h\_graph = new uint32\_t[n \* n];

uint32\_t \*h\_floydResult = new uint32\_t[n \* n];

for (size\_t i = 0; i < n \* n; ++i) {

uint32\_t current\_elem = 0;

fscanf(fin, "%d", &current\_elem);

//graph\_reader.read((char \*)&current\_elem, 1);

h\_graph[i] = current\_elem;

}

//graph\_reader.close();

// Run empty task on cuda - it will decrease time of first run

int threadNum = std::min(n, uint32\_t(32));

dim3 gridSize(n / threadNum, n / threadNum, 1);

dim3 cudaBlockSize(threadNum, threadNum, 1);

WakeGpuKernel << <1, cudaBlockSize >> >(32);

// Blocked Floyd-Warshall algorithm on cuda

FloydBlocked(h\_graph, h\_floydResult, n);

// Write Floyd results to file

std::fstream result\_writer(argv[2], std::fstream::out | std::fstream::binary);

for (size\_t i = 0; i < n \* n; ++i) {

result\_writer.write((char\*)&h\_floydResult[i], 4);

}

result\_writer.close();

delete[] h\_graph;

delete[] h\_floydResult;

return 0;

}